Contact: Jolien Creighton (Librarian) jolien@gravity.phys.uwm.edu

RCS Revision: 1.45 Date: 2007/03/25 18:04:48 UTC — Compiled: 2008/03/16
Contributors
Contents

Preface xv

Manual I Installation Guide 1

1 Pre-Installation 3
1 Obtaining required software 5
2 Obtaining LAL from CVS 7

2 Installing LAL 8

Manual II User Guide 10

3 LAL Standard Package 11

3 Package std 12
3.1 Header LALConfig.h .................................................. 13
3.2 Header LALStdlib.h .................................................. 14
3.3 Header LALRCSID.h .................................................. 15
3.4 Header LALDatatypes.h .............................................. 16
3.4.1 Primitive datatypes .............................................. 17
3.4.2 Atomic datatypes codes ........................................... 19
3.4.3 Aggregate datatypes ............................................. 20
3.4.4 Structured datatypes ............................................. 22
3.4.5 The LAL universal status structure LALStatus ............... 25
3.5 Header LALStatusMacros.h ........................................ 26
3.5.1 Status-reporting objects ......................................... 26
3.5.2 Using the status tools .......................................... 28
3.5.3 Compilation flags ................................................ 32
3.5.4 Sample header: LALPrimer.h .................................... 35
3.5.5 Sample module: LALPrimer.c ................................... 36
3.5.6 Sample program: LALPrimerTest.c ............................... 37
3.6 Header LALConstants.h ............................................. 38
3.7 Header LALStdio.h .................................................. 41
3.7.1 Module LALStd.c ................................................ 42
3.8 Header LALVersion.h ................................................ 43
3.8.1 Program LALVersionTest.c ...................................... 45
3.9 Header LALMalloc.h ............................................... 46
3.9.1 Module LALMalloc.c ............................................. 47
3.9.2 Program LALMallocTest.c ...................................... 51
3.10 Header LALError.h ................................................ 52
3.10.1 Module LALError.c .............................................. 53
3.11 Header LALGSL.h .................................................. 56
  3.11.1 Module LALGSL.c ........................................... 57
  3.11.2 Program LALGSLTest.c .................................. 58
3.12 Header StringInput.h ........................................... 59
  3.12.1 Module StringToken.c .................................... 61
  3.12.2 Module StringConvert.c .................................. 62
3.13 Header Grid.h .................................................. 64
  3.13.1 Module Grid.c ............................................. 65

4 Package sample .................................................... 67
  4.1 Header LALSample.h .......................................... 68
    4.1.1 Module LALSample.c ..................................... 69
    4.1.2 Program LALSampleTest.c............................... 70

4 General Packages .................................................. 71

5 Package date ....................................................... 72
  5.1 Header Date.h ................................................ 73
    5.1.1 Module Julian.c .......................................... 76
    5.1.2 Module DateString.c ..................................... 78
    5.1.3 Module LMST1.c .......................................... 79
    5.1.4 Module SecsToLALDate.c .................................. 82
    5.1.5 Module GPSToUTC.c ....................................... 84
    5.1.6 Module GPSToFloat.c ..................................... 86
    5.1.7 Module GPSToINT8.c ...................................... 87
    5.1.8 Module IncrementGPS.c ................................... 88
    5.1.9 Module GPSTimeNow.c .................................... 90
    5.1.10 Module Playground.c .................................... 91
  5.2 Header TimeDelay.h ........................................... 92
    5.2.1 Module TimeDelay.c ...................................... 93

6 Package factories ................................................ 95
  6.1 Header AVFactories.h ......................................... 96
    6.1.1 Module VectorFactories.c ................................. 97
    6.1.2 Module ArrayFactories.c ................................. 99
    6.1.3 XLAL Functions .......................................... 101
    6.1.4 Program VectorFactoriesTest.c ......................... 103
    6.1.5 Program ArrayFactoriesTest.c ......................... 104
  6.2 Header SeqFactories.h ........................................ 105
    6.2.1 Module VectorSequenceFactories.c .................... 106
    6.2.2 XLAL Functions .......................................... 107
    6.2.3 Program VectorSequenceFactoriesTest.c ............. 108
    6.2.4 Program ArraySequenceFactoriesTest.c ............. 109

7 Package tools ..................................................... 110
  7.1 Header Units.h ................................................. 111
    7.1.1 Module UnitDefs.c ....................................... 112
    7.1.2 Module UnitNormalize.c .................................. 114
    7.1.3 Module UnitRaise.c ....................................... 115
    7.1.4 Module UnitMultiply.c ................................... 116
    7.1.5 Module UnitCompare.c ..................................... 117
    7.1.6 XLAL Functions .......................................... 118
    7.1.7 Program UnitsTest.c ..................................... 119
  7.2 Header DetectorSite.h ........................................ 120
  7.3 Header LALDetectors.h ........................................ 121
    7.3.1 Module CreateDetector.c .................................. 124
    7.3.2 Program DetectorSiteTest.c ............................. 128
  7.4 Header DetResponse.h ......................................... 129
5 General Mathematical and Signal Analysis Packages

9 Package clremoval

9.1 Header CLR.h .................................................. 259
  9.1.1 Module HarmonicFinder.c .................................. 261
  9.1.2 Module RefInterference.c .................................. 262
  9.1.3 Module CleanAll.c .......................................... 263
  9.1.4 Program CLRTest.c ......................................... 264

10 Package fft

10.1 Header RealFFT.h ............................................. 267
  10.1.1 Module RealFFT.c .......................................... 268
  10.1.2 XLAL Functions ........................................... 272
  10.1.3 Program RealFFITest.c .................................... 274
  10.2 Header ComplexFFT.h ....................................... 275
  10.2.1 Module ComplexFFT.c ..................................... 276
  10.2.2 Program ComplexFFITest.c ................................. 279
  10.3 Header TimeFreqFFT.h ...................................... 280
  10.3.1 Module TimeFreqFFT.c ..................................... 282
  10.3.2 Program TimeFreqFFITest.c ............................... 285

11 Package stats

11.1 Header LALMoment.h ......................................... 288
  11.1.1 Module LALMoment.c ...................................... 289
  11.1.2 Program LALMomentTest.c ................................. 290
  11.2 Header LALCorrelation.h .................................... 291
  11.2.1 Module LALCorrelation.c ................................. 292
  11.2.2 Program LALCorrelationTest.c .......................... 293

12 Package tdfilter

12.1 Header ZPGFilter.h .......................................... 295
  12.1.1 Module CreateZPGFilter.c ................................ 296
  12.1.2 Module DestroyZPGFilter.c ............................... 297
  12.1.3 Module BilinearTransform.c .............................. 298
  12.2 Header IIRFilter.h .......................................... 300
  12.2.1 Module CreateIIRFilter.c ................................ 302
  12.2.2 Module DestroyIIRFilter.c ............................... 304
  12.2.3 Module IIRFilter.c ........................................ 305
  12.2.4 Module IIRFilterVector.c ................................. 306
  12.2.5 Module IIRFilterVectorR.c ............................... 307
  12.2.6 Program IIRFilterTest.c ................................ 308
  12.3 Header BandPassTimeSeries.h ............................... 311
  12.3.1 Module ButterworthTimeSeries.c ......................... 312
  12.3.2 Program BandPassTest.c ................................ 315
  12.4 Header LFC.h ................................................ 316

13 Package utilities

13.1 Header Random.h ............................................. 319
  13.1.1 Module Random.c .......................................... 320
  13.1.2 Program RandomTest.c .................................... 322
  13.1.3 Module MersenneRandom.c ................................. 323
  13.1.4 Program MersenneRandomTest.c ......................... 325
  13.2 Header FindRoot.h .......................................... 327
  13.2.1 Module FindRoot.c ........................................ 328
  13.2.2 Program FindRootTest.c ................................ 330
  13.3 Header Integrate.h ........................................... 331
  13.3.1 Module Integrate.c ........................................ 333
  13.3.2 Program IntegrateTest.c ................................. 335
  13.4 Header Interpolate.h ....................................... 336
## Core Data Analysis Packages

### 13.12 Module Interpolate.c
- Interpolate.c
- InterpolateTest.c

### 13.4.2 Program InterpolateTest.c
- InterpolateTest.c

### 13.5 Header Sort.h
- Sort.h
- SortTest.c

### 13.6 Header ODE.h
- ODE.h
- ODE.c
- ODETest.c

### 13.7 Header Dirichlet.h
- Dirichlet.h
- Dirichlet.c
- DirichletTest.c

### 13.8 Header CoarseGrainFrequencySeries.h
- CoarseGrainFrequencySeries.h
- CoarseGrainFrequencySeries.c
- SCoarseGrainFrequencySeriesTest.c
- SCoarseGrainFrequencySeriesSeriesTest.c
- DCoarseGrainFrequencySeriesTest.c
- DCoarseGrainFrequencySeriesSeriesTest.c
- CCoarseGrainFrequencySeriesTest.c
- CCoarseGrainFrequencySeriesSeriesTest.c

### 13.9 Header MatrixUtils.h
- MatrixUtils.h
- MatrixOps.c
- DetInverse.c
- DetInverseInternal.c
- DetInverseTest.c
- Eigen.c
- EigenInternal.c
- EigenTest.c

### 13.10 Header LALRunningMedian.h
- LALRunningMedian.h
- LALRunningMedian.c
- LALRunningMedianTest.c

### 13.11 Header RngMedBias.h
- RngMedBiasTest.c

### 13.12 Module RngMedBias.c
- RngMedBias.c

### 14 Package window

## 6 Core Data Analysis Packages

### 15 Package inject
- Inject.h
- InjectVector.c
- InjectTimeSeries.c
- BasicInjectTest.c
- InjectTest.c
- InjectVector.h

### 15.2 Header SimulateCoherentGW.h
- SimulateCoherentGW.h
- SimulateCoherentGW.c

### 15.3 Header SimulateInspiral.h
- SimulateInspiral.h
- SimulateInspiral.c

### 15.4 Header SimulateSE.h
- SimulateSE.h
- SimulateSE.c

### 15.5 Header SimulatePopcorn.h
- SimulatePopcorn.h
- SimulatePopcorn.c

### 15.6 Header GenerateInspiral.h
- GenerateInspiral.h
- GenerateInspiral.c

### 15.7 Header GeneratePPNInspiral.h
- GeneratePPNInspiral.h
- GeneratePPNInspiral.c
- GeneratePPNInspiralCorInspiral.c
- GeneratePPNInspiralTruncInspiral.c
- GeneratePPNInspiralParams.c
- GenerateInspiralSmooth.c
- GeneratePPNInspiralTest.c
- GeneratePPNInspiralTest.c
7 Burst Packages

16 Package block

16.1 Header BlockRho.h

16.1.1 Module BlockRho2.c

17 Package burstsearch

17.1 Header TFTransform.h

17.1.1 Module CreateRealDFTParams.c

18 Package ring

18.1 Header Ring.h

18.1.1 Module Ring.c

19 Package slopefilters

19.1 Header SlopeDetectorFilter.h

19.1.1 Module SlopeDetectorFilter.c

19.1.2 Program SlopeDetectorFilterTest.c

20 Package: tfclusters

20.1 Header TFClusters.h

20.1.1 Module TFClusters.c

20.1.2 Program TFClustersTest1.c

20.2 Header TFCThresholds.h

20.2.1 Module TFCThresholds.c

21 Package waveburst

21.1 Header LALWavelet.h

21.1.1 Module LALWavelet.c

8 Inspiral Packages

22 Package bank

22.1 Template Placement for Binary Inspiral Searches

22.1.1 Coarse Grid Algorithm : the square placement

22.1.2 Coarse Grid Algorithm : the hexagonal placement

22.1.3 Coarse Grid Algorithm : the hybrid hexagonal placement

22.1.4 Coarse Grid algorithm : non physical placement (BCV)

22.2 Header LALInspiralBank.h

22.2.1 Module LALInspiralCreateCoarseBank.c

22.2.2 Module LALInspiralCreateBCVBank.c

22.2.3 Module LALInspiralBCVcutBank.c

22.2.4 Module LALInspiralBankUtils.c

22.2.5 Module InspiralSpinBank.c
23.1 Conventions

23.1.1 Conventions for Discrete Quantities
23.1.2 The Discrete Fourier Transform
23.1.3 Power Spectral Densities

23.2 Header FindChirpDatatypes.h

23.3 Header FindChirp.h

23.3.1 Module FindChirpLinkedList.c
23.3.2 Module FindChirpMemory.c
23.3.3 Module FindChirpData.c
23.3.4 Module FindChirpTemplate.c
23.3.5 Module FindChirpFilter.c
23.3.6 Matched Filtering Using Post-Newtonian Templates
23.3.7 Module FindChirpFilterOutputVeto.c
23.3.8 Module FindChirpSimulation.c

23.4 Header FindChirpSP.h

23.4.1 Module FindChirpSPData.c
23.4.2 Module FindChirpSPTemplate.c

23.5 Header FindChirpTD.h

23.5.1 Module FindChirpTDData.c
23.5.2 Module FindChirpTDTemplate.c

23.6 Header FindChirpChisq.h

23.6.1 Module FindChirpChisqInit.c
23.6.2 Module FindChirpChisq.c

23.7 Header FindChirpBCV.h

23.7.1 Module FindChirpBCVData.c
23.7.2 Module FindChirpBCVTemplate.c
23.7.3 Module FindChirpBCVChisq.c
23.7.4 Module FindChirpBCVFilter.c

23.8 Header FindChirpBCVSpin.h

23.8.1 Module FindChirpBCVSpinData.c
23.8.2 Module FindChirpBCVSpinTemplate.c
23.8.3 Module FindChirpBCVSpinFilter.c
## 24 Package inspiral

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>24.1</td>
<td>Taylor and Pade approximants</td>
</tr>
<tr>
<td>24.2</td>
<td>Stationary Phase Approximation</td>
</tr>
<tr>
<td>24.2.1</td>
<td>Amplitude in the Fourier domain</td>
</tr>
<tr>
<td>24.3</td>
<td>Detection template family</td>
</tr>
<tr>
<td>24.3.1</td>
<td>DTF for non-spinning sources</td>
</tr>
<tr>
<td>24.3.2</td>
<td>DTF for spinning sources</td>
</tr>
<tr>
<td>24.4</td>
<td>Effective one-body approach</td>
</tr>
<tr>
<td>24.5</td>
<td>Spinning Modulated Chirps</td>
</tr>
</tbody>
</table>

### Module Headers

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LALInspiralParameterCalc.c</td>
<td></td>
</tr>
<tr>
<td>LALInspiralAmplitude.c</td>
<td></td>
</tr>
<tr>
<td>LALInspiralWaveLength.c</td>
<td></td>
</tr>
<tr>
<td>LALInspiralChooseModel.c</td>
<td></td>
</tr>
<tr>
<td>LALInspiralSetup.c</td>
<td></td>
</tr>
<tr>
<td>LALInspiralInit.c</td>
<td></td>
</tr>
<tr>
<td>LALInspiralWaveTaper.c</td>
<td></td>
</tr>
<tr>
<td>LALInspiralWave.c</td>
<td></td>
</tr>
<tr>
<td>LALInspiralWave1.c</td>
<td></td>
</tr>
<tr>
<td>LALInspiralWave1Templates.c</td>
<td></td>
</tr>
<tr>
<td>LALInspiralWave2.c</td>
<td></td>
</tr>
<tr>
<td>LALInspiralWave2Templates.c</td>
<td></td>
</tr>
<tr>
<td>LALInspiralWave3.c</td>
<td></td>
</tr>
<tr>
<td>LALInspiralWave3Templates.c</td>
<td></td>
</tr>
<tr>
<td>LALInspiralStationaryPhaseApprox1.c</td>
<td></td>
</tr>
<tr>
<td>LALInspiralStationaryPhaseApprox2.c</td>
<td></td>
</tr>
<tr>
<td>LALEOBWaveform.c</td>
<td></td>
</tr>
<tr>
<td>LALEOBWaveformTemplates.c</td>
<td></td>
</tr>
<tr>
<td>LALBCVWaveform.c</td>
<td></td>
</tr>
<tr>
<td>LALInspiralDerivatives.c</td>
<td></td>
</tr>
<tr>
<td>LALInspiralVelocity.c</td>
<td></td>
</tr>
<tr>
<td>LALInspiralPhasing1.c</td>
<td></td>
</tr>
<tr>
<td>LALInspiralPhiofVIntegrand.c</td>
<td></td>
</tr>
<tr>
<td>LALInspiralPhasing2.c</td>
<td></td>
</tr>
<tr>
<td>LALInspiralPhasing3.c</td>
<td></td>
</tr>
<tr>
<td>LALInspiralTofV.c</td>
<td></td>
</tr>
<tr>
<td>LALInspiralToFVIntegrand.c</td>
<td></td>
</tr>
<tr>
<td>LALInspiralTiming2.c</td>
<td></td>
</tr>
<tr>
<td>LALInspiralFrequency3.c</td>
<td></td>
</tr>
<tr>
<td>LALRungeKutta4.c</td>
<td></td>
</tr>
<tr>
<td>LALInspiralParseParameters.c</td>
<td></td>
</tr>
</tbody>
</table>

### Test program

- LALGenerateInspiralWaveform.c

## 25 Package noisemodels

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LALNoiseModels.h</td>
<td></td>
</tr>
<tr>
<td>LALNoiseSpectralDensity.c</td>
<td></td>
</tr>
<tr>
<td>LALInspiralWaveCorrelate.c</td>
<td></td>
</tr>
<tr>
<td>LALInspiralWaveNormalise.c</td>
<td></td>
</tr>
<tr>
<td>LALInspiralWaveNormaliseLSO.c</td>
<td></td>
</tr>
<tr>
<td>LALAddVectors.c</td>
<td></td>
</tr>
<tr>
<td>LALAdvLIGOPsd.c</td>
<td></td>
</tr>
<tr>
<td>LALLIGOIPsd.c</td>
<td></td>
</tr>
<tr>
<td>LALAddLIGOIPsd.c</td>
<td></td>
</tr>
<tr>
<td>LALRandomInspiralSignal.c</td>
<td></td>
</tr>
<tr>
<td>LALColoredNoise.c</td>
<td></td>
</tr>
</tbody>
</table>

### Test program

- LALAddVectors.c
9 Pulsar Packages

26 Package houghpulsar: The Hough transform

26.1 Header LUT.h
26.1.1 Module Stereographic.c
26.1.2 Module PatchGrid.c
26.1.3 Module ParamPLUT.c
26.1.4 Module NDPARAMPLUT.c
26.1.5 Module ConstructPLUT.c
26.1.6 Program TestConstructPLUT.c
26.1.7 Program TestNDConstructPLUT.c
26.2 Header PHMD.h
26.2.1 Module Peak2PHMD.c
26.2.2 Program TestPeak2PHMD.c
26.2.3 Program TestNDPeak2PHMD.c
26.3 Header HoughMap.h
26.3.1 Module HoughMap.c
26.3.2 Program TestHoughMap.c
26.3.3 Program TestNDHoughMap.c
26.4 Header LALHough.h
26.4.1 Module DriveHough.c
26.4.2 Program TestDriveHough.c
26.4.3 Program TestDriveNDHough.c
26.5 Header Statistics.h
26.5.1 Module Statistics.c
26.5.2 Program TestStatistics.c
26.6 Header Velocity.h
26.6.1 Module Velocity.c
26.6.2 Program TestVelocity.c

27 Package pulsar: common routines

27.1 Header PulsarTimes.h
27.1.1 Module GetEarthTimes.c
27.1.2 Module TBaryptolemaic.c
27.1.3 Module TSpin.c
27.1.4 Module TComp.c
27.1.5 Module DTephemeris.c
27.2 Header LALBarycenter.h
27.2.1 Module LALBarycenter.c
27.3 Header LalinItBarycenter.h
27.3.1 Program LALInitBarycenter.c
27.4 Header FlatMesh.h
27.4.1 Module FlatMesh.c
27.4.2 Program FlatMeshTest.c
27.4.3 Program DircatedMeshTest.c
27.5 Header TwobMesh.h
27.5.1 Module TwobMesh.c
31.1.13 Program StochasticInverseNoiseTest.c ........................................ 899
31.1.14 Module StochasticOmegaGW.c .......................................................... 901
31.1.15 Program StochasticOmegaGWTest.c .................................................. 902
31.1.16 Module OverlapReductionFunction.c .............................................. 905
31.1.17 Program OverlapReductionFunctionTest.c ....................................... 907
31.2 Header AstroOmega.h ........................................................................ 911

11 LAL Support Interface ................................................................. 918

32 Package support .............................................................................. 919
32.1 Header FileIO.h .............................................................................. 920
32.1.1 Module FileIO.c ............................................................................ 921
32.2 Header PrintVector.h ...................................................................... 922
32.2.1 Module PrintVector.c ..................................................................... 923
32.2.2 Program PrintVectorTest.c .......................................................... 924
32.3 Header PrintFTSeries.h .................................................................. 925
32.3.1 Module PrintTimeSeries.c ............................................................. 926
32.3.2 Program PrintFTSeriesTest.c ....................................................... 927
32.3.3 Module PrintFrequencySeries.c .................................................... 928
32.4 Header ReadFTSeries.h .................................................................. 930
32.4.1 Module ReadFrequencySeries.c .................................................... 931
32.4.2 Module ReadTimeSeries.c ............................................................. 932
32.4.3 Program ReadFTSeriesTest.c ....................................................... 933
32.5 Header ReadNoiseSpectrum.h .......................................................... 935
32.5.1 Module ReadNoiseSpectrum.c ........................................................ 937
32.6 Header StreamInput.h ..................................................................... 938
32.6.1 Module StreamVectorInput.c ......................................................... 939
32.6.2 Module StreamVectorSequenceInput.c ........................................... 941
32.6.3 Module StreamSequenceInput.c ..................................................... 943
32.6.4 Module StreamSeriesInput.c .......................................................... 945
32.6.5 Module StreamGridInput.c ............................................................. 949
32.6.6 Program StreamInputTest.c .......................................................... 952
32.6.7 Program StreamSeriesInputTest.c ................................................. 954
32.7 Header StreamOutput.h .................................................................... 955
32.7.1 Module StreamSeriesOutput.c ........................................................ 956
32.7.2 Module StreamGridOutput.c ........................................................... 958
32.8 Header LALInitBarycenter.h ............................................................. 960
32.8.1 Module LALInitBarycenter.c .......................................................... 961
32.9 Header LALXMGRInterface.h ............................................................. 962
32.9.1 Module LALXMGRInterface.c ......................................................... 964
32.10 Header LIGOLwXML.h .................................................................... 965
32.10.1 Module LIGOLwXML.c ................................................................. 966
32.11 Header LIGOLwXMLHeaders.h .......................................................... 968
32.12 Header LIGOLwXMLRead.h .............................................................. 969
32.12.1 Module CreateMetaTableDir.c ....................................................... 970
32.12.2 Module LIGOLwXMLRead.c .......................................................... 971
32.12.3 Module LIGOLwXMLRingdownRead.c ........................................... 974
32.13 Header SegmentsIO.h ..................................................................... 975
32.13.1 Module SegmentsIO.c ................................................................. 976
32.13.2 Program SegmentsIOTest.c .......................................................... 977
32.14 Header ConfigFile.h ...................................................................... 978
32.14.1 Module ConfigFile.c ..................................................................... 979
32.14.2 Program ConfigFileTest.c ............................................................ 981
32.15 Header LALMathematica.h ............................................................... 982
32.15.1 Module LALMath3DPlot.c ............................................................. 985
32.15.2 Module LALMathNDPlot.c ............................................................. 987
32.15.3 Program LALMath3DPlotTest.c ..................................................... 988
32.15.4 Program LALMathNDPlotTest.c .................................................... 989
### 12 LAL Framedata Interface (optional)

<table>
<thead>
<tr>
<th>Package</th>
<th>991</th>
</tr>
</thead>
<tbody>
<tr>
<td>framedata</td>
<td></td>
</tr>
<tr>
<td>- 33.1 Header FrameCache.h</td>
<td>992</td>
</tr>
<tr>
<td>- 33.1.1 Module FrameCache.c</td>
<td>994</td>
</tr>
<tr>
<td>- 33.2 Header FrameStream.h</td>
<td>996</td>
</tr>
<tr>
<td>- 33.2.1 Module FrameStream.c</td>
<td>999</td>
</tr>
<tr>
<td>- 33.2.2 Module FrameSeries.c</td>
<td>1001</td>
</tr>
<tr>
<td>- 33.2.3 Program FrameStreamTest.c</td>
<td>1007</td>
</tr>
<tr>
<td>- 33.3 Header FrameCalibration.h</td>
<td>1008</td>
</tr>
<tr>
<td>- 33.3.1 Module FrameCalibration.c</td>
<td>1009</td>
</tr>
<tr>
<td>- 33.3.2 Program FrameCalibrationTest.c</td>
<td>1011</td>
</tr>
</tbody>
</table>

### 13 LAL MPI Interface (optional)

<table>
<thead>
<tr>
<th>Package</th>
<th>1012</th>
</tr>
</thead>
<tbody>
<tr>
<td>comm</td>
<td></td>
</tr>
<tr>
<td>- 34.1 Header Comm.h</td>
<td>1013</td>
</tr>
<tr>
<td>- 34.1.1 Module SendRecv.c</td>
<td>1014</td>
</tr>
<tr>
<td>- 34.1.2 Module Exchange.c</td>
<td>1016</td>
</tr>
<tr>
<td>- 34.1.3 Program SendRecvTest.c</td>
<td>1021</td>
</tr>
<tr>
<td>- 34.1.4 Program ExchangeTest.c</td>
<td>1026</td>
</tr>
<tr>
<td>- 34.1.5 Program ExchangeTest.c</td>
<td>1027</td>
</tr>
</tbody>
</table>

### Manual III Coding and Documentation Instructions

<table>
<thead>
<tr>
<th>14 How to Develop Code for LAL</th>
<th>1028</th>
</tr>
</thead>
<tbody>
<tr>
<td>35 An Introduction to the LIGO/LSC Algorithm Library (LAL)</td>
<td>1029</td>
</tr>
<tr>
<td>- 35.1 The LAL webpage</td>
<td>1030</td>
</tr>
<tr>
<td>36 Notes about coding</td>
<td>1031</td>
</tr>
<tr>
<td>37 The directory structure of the LAL</td>
<td>1032</td>
</tr>
<tr>
<td>- 37.1 Schematic Diagram of Directory Structure</td>
<td>1033</td>
</tr>
</tbody>
</table>

### 38 Documenting your code

| 38.1 Use LATEX | 1034 |
| 38.2 The layout of the documentation | 1035 |
| 38.3 Documentation for a single package versus a comprehensive LAL manual | 1035 |
| 38.4 What about figures? | 1036 |
| 38.5 Autodocumentation and Indexing | 1036 |
| 38.5.1 Autodocumentation requirements | 1036 |
| 38.5.2 Indexing requirements | 1036 |

### 39 Package samplepackage

| 39.1 Header SampleHeader1.h | 1037 |
| 39.1.1 Module SampleModule1.c | 1038 |

### 15 LALDoc: Automated Documentation

<table>
<thead>
<tr>
<th>15 LALDoc: Automated Documentation</th>
<th>1039</th>
</tr>
</thead>
<tbody>
<tr>
<td>40 The automatic documentation system</td>
<td>1043</td>
</tr>
<tr>
<td>- 40.1 A four step introduction to the code parser laldoc</td>
<td>1044</td>
</tr>
<tr>
<td>- 40.2 Brief Description of the parser laldoc</td>
<td>1044</td>
</tr>
<tr>
<td>- 40.2.1 The laldoc command line</td>
<td>1044</td>
</tr>
<tr>
<td>- 40.2.2 The three laldoc environments</td>
<td>1045</td>
</tr>
<tr>
<td>- 40.2.3 How laldoc handles the output files</td>
<td>1045</td>
</tr>
<tr>
<td>- 40.3 Examples of how to use the three environments in laldoc</td>
<td>1045</td>
</tr>
</tbody>
</table>
40.3.1 The `<lalVerbatim>` environment ........................................ 1045
40.3.2 The `<lalLaTeX>` environment ........................................... 1046
40.3.3 The `<lalErrTable>` environment, for printing a table of the error codes and warnings 1046
40.4 How the documentation is automatically built .......................... 1046
Preface

Preface to the first edition:
The formal document governing the LAL-code and documentation is the LIGO Data Analysis System Numerical Algorithms Library Specification and Style Guide http://www.ligo.caltech.edu/docs/T/T990030-07.pdf (called the LAL-Spec through out). Like a constitution describing a government, the guidelines given in the LAL-Spec are quite general. What is given in this document is a more like the criminal code: the nut-n-bolts explanation of how to put the LAL-Spec into practice. Like a constitution, if there are any inconsistencies, the LAL-Spec takes precedence over any statements in this document. Also like a constitution, the LAL-Spec is to be taken very seriously, but it is not suicide pact: if there are real problems with the Spec, there is a formal process to change it.

The first part of this document gives a brief introduction to the LAL and some coding and documentation instructions. The second part is the documentation of the LAL code itself.

Preface to the second edition:
By necessity, the first edition of the L S D contained long lists of instructions that specified the layout of the documentation and code; however many of these rules have migrated to the LAL-Spec itself, and thus there is less need for such rules. Furthermore, now there is a much larger body of example code to help guide people in how to follow the LAL rules. Therefore Part I of this new edition of the L S D is considerably shorter than its predecessor.
Manual I

Installation Guide
This section describes how to perform a basic install of LAL. This is not quite a minimal install—support for the (optional) frame library is included—but it is probably the most useful basic installation. It is assumed that you are obtaining LAL from CVS rather than from a distribution tarball. If you are using a distribution tarball then the installation will be somewhat more straightforward.

This section assumes you are using a Unix system with standard GNU tools in place, e.g., gcc, GNU m4, cvs, etc. The instructions are written for use with a Bourne-like shell (e.g., bash) rather than a C-shell (e.g., tcsh). Really the only difference is that you need to change statements like

```
MYENV=value
export MYENV
```

(Bourne shell syntax) to

```
setenv MYENV value
```

(C-shell syntax).

The part of the instructions in this document are extracted from the shell scripts `lal-preinstall-x.sh` and `lal-install-x.sh` in the doc subdirectory in LAL. Also extracted are plain-text `lal-preinstall.txt` and `lal-install.txt`, and nicely-formatted shell scripts `lal-preinstall.sh` and `lal-install.sh`. The shell scripts are a superset of the instructions here. They also set up some useful configuration files. Rather than copying the instructions in this document, you can simply edit the scripts `lal-preinstall.sh` and `lal-install.sh` (to specify the install locations, etc.) and run these scripts instead. (Run `lal-preinstall.sh` first and `lal-install.sh` second.)
Section 1

Pre-Installation
This pre-installation is required to obtain the necessary tools to build LAL from a CVS checkout. If you are building LAL from a distributed tarball then you will not need to install automake or autoconf, but you will still need fftw3 and gsl. The frame software is optional and metaio software is optional in LAL, but it is very useful so these instructions assume that you will want to build LAL to interface with these libraries.
Chapter 1

Obtaining required software

Perform the pre-install for LAL. The following commands will download, build, and install the software required to build LAL. The software includes the following:

- pkgconfig-0.15.0
- fftw-3.0.1 (both single- and double-precision libraries required)
- gsl-1.5
- libframe-6.14 (optional but recommended)
- libmetaio-5.4 (optional but recommended)

If this software is already on your system, you can use the existing software. If some of the software is missing, you can use the appropriate part of these instructions to install that component. Pre-compiled versions of the software are also available for installation. See the RPMs section for instructions on obtaining these. The software is installed in the directory `LSCSOFT_LOCATION`. If this variable is not set, it will be installed in `$HOME/opt/lscsoft` by default. To install in some other location, set `LSCSOFT_LOCATION` to that location.

The software is installed in the directory `LSCSOFT_LOCATION`. If this variable is not set, it will be installed in `$HOME/opt/lscsoft` by default. To install in some other location, set `LSCSOFT_LOCATION` to that location. The commands listed below are appropriate for a Bourne-shell (e.g., bash); they will need to be modified appropriately for C-shells (e.g., tcsh).

```
LSCSOFT_PERSISTENT=$LSCSOFT_LOCATION:-"$HOME/opt/lscsoft"
LSCSOFT_INDIR=$LSCSOFT_PERSISTENT/include
LSCSOFT_LIBDIR=$LSCSOFT_PERSISTENT/lib
LSCSOFT_SRCDIR=$LSCSOFT_PERSISTENT/src
LSCSOFT_ETCDIR=$LSCSOFT_PERSISTENT/etc
LSCSOFT_TMPDIR=$LSCSOFT_PERSISTENT/tmp
```

This is where to get sources:

```
LALSRCURL=http://www.lsc-group.phys.uwm.edu/lal/sources
```

**setup directories**

```
mkdir -p $LSCSOFT_PERSISTENT
mkdir -p $LSCSOFT_INDIR
mkdir -p $LSCSOFT_LIBDIR
mkdir -p $LSCSOFT_SRCDIR
mkdir -p $LSCSOFT_ETCDIR
mkdir -p $LSCSOFT_TMPDIR
```

**get required autoconf, automake, fftw3, frame, gsl, and metaio you can use lynx -dump or wget -o- instead of curl**

```
curl $LALSRCURL/pkgconfig-0.15.0.tar.gz > $LSCSOFT_TMPDIR/pkgconfig-0.15.0.tar.gz
```

```
curl $LALSRCURL/fftw-3.0.1.tar.gz > $LSCSOFT_TMPDIR/fftw-3.0.1.tar.gz
```

```
curl $LALSRCURL/gsl-1.5.tar.gz > $LSCSOFT_TMPDIR/gsl-1.5.tar.gz
```

```
```

```
curl $LALSRCURL/libmetaio-5.4.tar.gz > $LSCSOFT_TMPDIR/libmetaio-5.4.tar.gz
```

**unpack these archives in LSCSOFT_SRCDIR**
cd $LSCSOFT_SRCDIR
tar -zxvf $LSCSOFT_TMPDIR/pkgconfig-0.15.0.tar.gz
tar -zxvf $LSCSOFT_TMPDIR/fftw-3.0.1.tar.gz
tar -zxvf $LSCSOFT_TMPDIR/gsl-1.5.tar.gz
tar -zxvf $LSCSOFT_TMPDIR/libframe-6.14.tar.gz
tar -zxvf $LSCSOFT_TMPDIR/libmetaio-5.4.tar.gz

build and install pkg-config

cd $LSCSOFT_SRCDIR/pkgconfig-0.15.0
./configure --prefix=$LSCSOFT_PREFIX
make
make install

build and install fftw3

cd $LSCSOFT_SRCDIR/fftw-3.0.1
./configure --prefix=$LSCSOFT_PREFIX --enable-shared --enable-float
make  # note: ignore fail... the build fails on MacOSX, but not seriously
make install  # note: ignore fail
make distclean
./configure --prefix=$LSCSOFT_PREFIX --enable-shared
make  # note: ignore fail
make install  # note: ignore fail

build and install gsl

cd $LSCSOFT_SRCDIR/gsl-1.5
./configure --prefix=$LSCSOFT_PREFIX
make
make install

build and install libframe

cd $LSCSOFT_SRCDIR/libframe-6.14
./configure --prefix=$LSCSOFT_PREFIX --disable-octave
make
make install

build and install libmetaio

cd $LSCSOFT_SRCDIR/libmetaio-5.4
./configure --prefix=$LSCSOFT_PREFIX
make
make install

To setup your environment to use the software that has been installed please add the following to your .profile if you use a bourne shell (e.g. bash):

```
LSCSOFT_LOCATION=$HOME/opt/lscsoft  # i: change this as appropriate
export LSCSOFT_LOCATION
if [ -f $LSCSOFT_LOCATION/etc/lscsoft-user-env.sh ]; then
  . $LSCSOFT_LOCATION/etc/lscsoft-user-env.sh
fi
```

If you are using a C shell (e.g., tcsh), instead add these lines to your .login:

```
setenv LSCSOFT_LOCATION $HOME/opt/lscsoft  # i: change this as appropriate
if ( -r $LSCSOFT_LOCATION/etc/lscsoft-user-env.csh ) then
  source $LSCSOFT_LOCATION/etc/lscsoft-user-env.csh
endif
```
Chapter 2

Obtaining LAL from CVS

This describes how to obtain a distribution of LAL from CVS, and to prepare LAL to be built. Make sure that you are in the directory where you want to download LAL.

LSCSOFT_LOCATION must be set — make sure this software is available:

```bash
PATH=$LSCSOFT_LOCATION/bin:$PATH
LD_LIBRARY_PATH=$LSCSOFT_LOCATION/lib:$LD_LIBRARY_PATH
export PATH LD_LIBRARY_PATH
```

set LSCSOFT_CVSUSER to your username and make sure you run cvs login!

```bash
LSCSOFT_CVSUSER="anonymous"
LSCSOFT_CVSROOT=":pserver:$LSCSOFT_CVSUSER@gravity.phys.uwm.edu:2402/usr/local/cvs/lscsoft"
```

get LAL and LALApps from CVS

```bash
cvs -d$LSCSOFT_CVSROOT checkout -d lal-current lal
```

now prepare LAL to be configured

```bash
cd lal-current
./00boot
```

LAL is now ready to be configured, built, and installed.
Section 2

Installing LAL
Perform the installation of LAL. The following commands will build and install LAL. This requires the software installed in the pre-install to have been built and installed. LAL will be built in the directory specified by the environment variable `LAL_LOCATION`. If the software was not installed in a standard location, such as `/usr` or `/usr/local`, then the location of the software must be specified. To do so, the location of this software is given by the environment variable `LSCSOFT_LOCATION` which must be set before these commands are executed.

```
LD_LIBRARY_PATH=$LSCSOFT_LOCATION/lib:$LD_LIBRARY_PATH
PKG_CONFIG_PATH=$LSCSOFT_LOCATION/lib/pkgconfig:$PKG_CONFIG_PATH
export PATH LD_LIBRARY_PATH PKG_CONFIG_PATH

LAL_PREFIX=$LAL_LOCATION

build and install LAL (the `--with-gcc-flags` tells gcc to use flags that will produce warnings about possibly non-portable code)
```

```
./configure --prefix=$LAL_PREFIX --with-gcc-flags
make
make install
```

Now LAL has been built and installed. To use LAL you may want to set various environment variables. If you are using a Bourne shell, put the following in your `.profile` file:

```
LAL_LOCATION=<value of "LAL_PREFIX">
export LAL_LOCATION
. $LAL_LOCATION/etc/lal-user-env.sh
```

If you are using a C shell, put the following in your `.login` file:

```
setenv LAL_LOCATION <value of "LAL_PREFIX">
source $LAL_LOCATION/etc/lal-user-env.csh
```
Manual II
User Guide
Section 3

LAL Standard Package
Chapter 3

Package std

This package contains headers providing basic datatypes, constants, and macros that support the LAL standard.
3.1 Header LALConfig.h

Defines configuration macro constants.

Synopsis

```c
#include <lal/LALConfig.h>
```

This header (which is not technically in the `std` package; rather it is generated directly in the `include/lal` directory during configuration) is included in essentially every other header file. It contains macro constants that are defined at configuration time. They are:

- **LAL_VERSION**: Constant string containing the version of LAL.
- **LAL_VERSION_MAJOR**: Integer representing the major version number of LAL.
- **LAL_VERSION_MINOR**: Integer representing the minor version number of LAL.
- **LAL_CONFIGURE_ARGS**: Constant string containing the arguments given to the configure script.
- **LAL_CONFIGURE_DATE**: Constant string containing the date when LAL was configured.
- **LAL_CVS_TAG**: Constant string containing the CVS tag used to checkout LAL (blank if none).
- **LAL_SIZEOF_DOUBLE**: Integer representing the size of a double precision floating point number.
- **LAL_SIZEOF_FLOAT**: Integer representing the size of a single precision floating point number.
- **LAL_SIZEOF_INT**: Integer representing the size of an integer.
- **LAL_SIZEOF_LONG**: Integer representing the size of a long integer.
- **LAL_SIZEOF_LONG_LONG**: Integer representing the size of a long long integer.
- **LAL_SIZEOF_SHORT**: Integer representing the size of a short integer.
- **LAL_NDEBUG**: Defined if debugging is turned off (use the configure argument `--disable-debug` to do this).
- **NOLALMACROS**: Defined if status macros are replaced by functions (where possible) (use the configure argument `--disable-macros` to do this).
- **LAL_PTHREAD_LOCK**: Defined if POSIX thread mutex locking is to be used for threadsafety (use the configure argument `--enable-pthread-lock` to do this).
- **LAL_FRAME_ENABLED**: Defined if LAL frame-format data reading routines will be compiled (use the configure argument `--enable-frame` to do this).
- **LAL_MPI_ENABLED**: Defined if LAL MPI routines will be compiled (use the configure argument `--enable-mpi` to do this).
3.2 Header **LALStdlib.h**

Includes the standard LAL header files.

**Synopsis**

```c
#include <lal/LALStdlib.h>
```

This header is the overall header for the std package. It provides the datatypes, constants, and macros required by most LAL functions, by including the following header files in the std package:

```c
#include <lal/LALRCSID.h>
#include <lal/LALDatatypes.h>
#include <lal/LALStatusMacros.h>
```

**LALStdlib.h** also includes function prototype headers for certain standard modules used by many LAL routines:

```c
#include <stdio.h>
#include <stdarg.h>
#include <lal/LALMalloc.h>
```

Author: J. D. E. Creighton, T. D. Creighton

$Id: LALStdlib.h,v 1.12 2007/06/08 14:41:53 bema Exp$
3.3 Header LALRCSID.h

Provides macros for assigning an RCS ID string to a file.

Synopsis

```
#include <lal/LALRCSID.h>
```

This header defines a pair of macros `RCSID()` and `NRCSID()`, which are used to assign Revision Control System (RCS) ID strings to a given source file. Whenever a file is added or changed in the central LAL distribution, the RCS software searches for symbols of the form `$Id$` or `$Id: ... $`, and expands them into strings that give information about the file's name, version number, when it was last modified and by whom. For an example of an RCS ID string, look at the `$Id: ... $` line at the bottom of this page.

The macro `RCSID()` is called as follows:

```
RCSID($Id$);
```

This assigns the RCS ID string to a variable `static const char *rcsid` in a given module or header. This variable will be loaded onto the stack whenever code from that module or header is used. This can be used as a diagnostic tool by debuggers.

The macro `NRCSID()` is called in the following manner:

```
NRCSID(MYFILEC,$Id$);
```

This assigns the RCS ID string to the variable `MYFILEC`, as above. Standard LAL naming conventions are that the variable name should be the file name converted to capital letters, with file extensions but without periods. Thus the module `MyFile.c` should store its ID in the variable `MYFILEC`, while the header `MyHeader.h` should store it in `MYHEADERH`.

LAL convention dictates that all modules and header files must call `NRCSID()` using the naming convention above. The call must be made after all subsidiary header files have been included (notably this header file, or `LALStdlib.h` which includes this header file), but before any actual functions or function prototypes. In modules containing LAL functions, the RCS ID string will typically be assigned to the `LALStatus` structure for those functions; see the documentation for the header `LALStatusMacros.h`.

This header is included automatically by all standard LAL headers, including `LALDatatypes.h`, `LALStatusMacros.h`, and `LALStdlib.h`. Thus if you have included any of the standard LAL headers, you will have gotten `LALRCSID.h` as well, and don't need to `#include` it separately. However, including it separately is not an error, as this and all LAL headers are required to have double-include protection.
3.4 Header LALDatatypes.h

Provides the basic LAL datatypes.

Synopsis

#include <lal/LALDatatypes.h>

This header defines the standard data types and data structures that are used throughout LAL. They fall into three general categories: *primitive* datatypes, *aggregates* of primitive datatypes, and *structured* datatypes. The LAL status structure is a special case of a structured datatype that is used in every standard LAL function.

This header file is automatically included by the header LALStdlib.h. In turn, this header file starts by including the header LALAtomicDatatypes.h, which is discussed in the following section.
3.4.1 Primitive datatypes

The primitive datatypes are defined in a separate header **LALAtomicDatatypes.h**, which is included by **LALDatatypes.h**. This is done in order to facilitate the interface between LAL and non-LAL modules. By including just **LALAtomicDatatypes.h**, a non-LAL module can ensure that it is using the same arithmetic standard as LAL, without being burdened by LAL’s more specialized structures.

Primitive datatypes are those that conceptually store a single number or quantity. They include both the atomic datatypes and the complex datatypes.

Atomic datatypes

Atomic LAL datatypes are platform-independent datatypes corresponding to the basic types in the C/C++ language. However, since the C/C++ types are not necessarily the same across platforms, the actual mapping between LAL and C/C++ datatypes may be different on different platforms. The following table lists the LAL atomic datatypes, their size and range, and the C/C++ datatype to which they usually correspond.

<table>
<thead>
<tr>
<th>Type</th>
<th>Bytes</th>
<th>Range</th>
<th>Usual C/C++ type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR</td>
<td>1</td>
<td>'\0' to '\255'</td>
<td>char</td>
</tr>
<tr>
<td>UCHAR</td>
<td>1</td>
<td>'\0' to '\255'</td>
<td>unsigned char</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>1</td>
<td>0 or positive</td>
<td>unsigned char</td>
</tr>
<tr>
<td>INT2</td>
<td>2</td>
<td>$-2^{15}$ to $2^{15} - 1$</td>
<td>short</td>
</tr>
<tr>
<td>INT4</td>
<td>4</td>
<td>$-2^{31}$ to $2^{31} - 1$</td>
<td>int or long</td>
</tr>
<tr>
<td>INT8</td>
<td>8</td>
<td>$-2^{63}$ to $2^{63} - 1$</td>
<td>long</td>
</tr>
<tr>
<td>UINT2</td>
<td>2</td>
<td>0 to $2^{16} - 1$</td>
<td>unsigned short</td>
</tr>
<tr>
<td>UINT4</td>
<td>4</td>
<td>0 to $2^{32} - 1$</td>
<td>unsigned int or long</td>
</tr>
<tr>
<td>UINT8</td>
<td>8</td>
<td>0 to $2^{64} - 1$</td>
<td>unsigned long long</td>
</tr>
<tr>
<td>REAL4</td>
<td>4</td>
<td>$-3.4 \times 10^{-38}$ to $3.4 \times 10^{38}$</td>
<td>float</td>
</tr>
<tr>
<td>REAL8</td>
<td>8</td>
<td>$-1.8 \times 10^{308}$ to $1.8 \times 10^{308}$</td>
<td>double</td>
</tr>
</tbody>
</table>

The unsigned character and integer datatypes store their values according to the usual binary system. For signed characters and integers, setting the most-significant bit indicates that the number formed from the remaining bits should be added to the lower value of the range. The **REAL4** and **REAL8** datatypes should store values according to the IEEE Standard 754 for Binary Floating-Point Arithmetic, which gives them the following precisions and dynamic ranges:

<table>
<thead>
<tr>
<th></th>
<th>REAL4</th>
<th>REAL8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum positive subnormal</td>
<td>$1.4 \times 10^{-35}$</td>
<td>$4.9 \times 10^{-324}$</td>
</tr>
<tr>
<td>Minimum positive normal</td>
<td>$1.2 \times 10^{-38}$</td>
<td>$2.2 \times 10^{-308}$</td>
</tr>
<tr>
<td>Maximum finite normal</td>
<td>$3.4 \times 10^{38}$</td>
<td>$1.8 \times 10^{308}$</td>
</tr>
<tr>
<td>Minimum fractional difference</td>
<td>$6.0 \times 10^{-8}$</td>
<td>$1.1 \times 10^{-16}$</td>
</tr>
<tr>
<td>Significant decimal digits</td>
<td>6–9</td>
<td>15–17</td>
</tr>
</tbody>
</table>

The minimum positive subnormal is the smallest positive representable number. The minimum positive normal is the smallest positive number that can be represented with full precision; that is, one whose mantissa lies in the range $[0.5,1)$. The maximum finite normal is the largest representable number other than the reserved code for $+\infty$. The minimum fractional difference is the smallest fractional difference between consecutive representable numbers, or half the difference between 1 and the next representable number. Significant decimal digits gives the number of decimal digits used to represent the binary number in decimal notation: the first is the maximum number of digits that are guaranteed not to change upon conversion to binary, the second is the number of digits required to represent a unique binary quantity.

Complex datatypes

LAL represents complex numbers as structures with two floating-point fields, storing the real and imaginary parts. These are considered primitive datatypes (rather than aggregate or structured datatypes) because they conceptually represent a single number. Furthermore, atomic and complex datatypes are treated equivalently by LAL aggregate and structured datatypes.

**COMPLEX8**

This structure stores a single-precision complex number in 8 bytes of memory. The fields are:
REAL4 re The real part.

REAL4 im The imaginary part.

COMPLEX16

This structure stores a double-precision complex number in 16 bytes of memory. The fields are:

REAL8 re The real part.

REAL8 im The imaginary part.
3.4.2 Atomic datatypes codes

The following constants specify the size, in bytes, of the atomic datatype.

<table>
<thead>
<tr>
<th>Name</th>
<th>Octal Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAL_1_BYTE_TYPE_SIZE</td>
<td>000</td>
<td>1 byte type</td>
</tr>
<tr>
<td>LAL_2_BYTE_TYPE_SIZE</td>
<td>001</td>
<td>2 byte type</td>
</tr>
<tr>
<td>LAL_4_BYTE_TYPE_SIZE</td>
<td>002</td>
<td>4 byte type</td>
</tr>
<tr>
<td>LAL_8_BYTE_TYPE_SIZE</td>
<td>003</td>
<td>8 byte type</td>
</tr>
<tr>
<td>LAL_16_BYTE_TYPE_SIZE</td>
<td>004</td>
<td>16 byte type</td>
</tr>
<tr>
<td>LAL_TYPE_SIZE_MASK</td>
<td>007</td>
<td>Mask for byte type size fields</td>
</tr>
</tbody>
</table>

The constant **LAL_TYPE_SIZE_MASK** is useful in extracting the size information from other type attributes. For example, the size, in bytes, of an atomic datatype can be found using something like the following:

```cpp
UINT4 code = LAL_S_TYPE_CODE;
UINT4 size = 1U << ( code & LAL_TYPE_SIZE_MASK );
```

The following constants are flags describing the type attributes. A type is either an integer or a floating-point, either purely real or complex, and, if integer, is either signed or unsigned.

<table>
<thead>
<tr>
<th>Name</th>
<th>Octal Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAL_FLTPT_TYPE_FLAG</td>
<td>010</td>
<td>Floating-point (not integer) type</td>
</tr>
<tr>
<td>LAL_CMPLX_TYPE_FLAG</td>
<td>020</td>
<td>Complex (not purely real) type</td>
</tr>
<tr>
<td>LAL_UNSGN_TYPE_FLAG</td>
<td>040</td>
<td>Unsigned (no sign info) type</td>
</tr>
</tbody>
</table>

To get the actual type, these flags are combined together and with the type size constants using the bitwise-or operator (|). For example, an eight-byte floating point number would be **LAL_8_BYTE_TYPE_SIZE | LAL_FLTPT_TYPE_FLAG**. Conceivably you could have a complex type made from a pair of unsigned one-byte integers that would be specified as **LAL_1_BYTE_TYPE_SIZE | LAL_CMPLX_TYPE_FLAG | LAL_UNSGN_TYPE_FLAG**. Fortunately, there are none of these in LAL. Attributes of a particular type can be extracted using the bitwise-and operator. For example:

```cpp
UINT4 code = LAL_S_TYPE_CODE;
UINT4 isfloat = ( code & LAL_FLTPT_TYPE_FLAG );
UINT4 iscomplex = ( code & LAL_CMPLX_TYPE_FLAG );
```

The following constants correspond to the types that actually exist in LAL. Their enumeration is the type **LALTYPECODE**.

<table>
<thead>
<tr>
<th>Name</th>
<th>Octal Value</th>
<th>Corresponding Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAL_CHAR_TYPE_CODE</td>
<td>000</td>
<td>CHAR</td>
</tr>
<tr>
<td>LAL_I2_TYPE_CODE</td>
<td>001</td>
<td>INT2</td>
</tr>
<tr>
<td>LAL_I4_TYPE_CODE</td>
<td>002</td>
<td>INT4</td>
</tr>
<tr>
<td>LAL_I8_TYPE_CODE</td>
<td>003</td>
<td>INT8</td>
</tr>
<tr>
<td>LAL_UCHAR_TYPE_CODE</td>
<td>040</td>
<td>UCHAR</td>
</tr>
<tr>
<td>LAL_U2_TYPE_CODE</td>
<td>041</td>
<td>UINT2</td>
</tr>
<tr>
<td>LAL_U4_TYPE_CODE</td>
<td>042</td>
<td>UINT4</td>
</tr>
<tr>
<td>LAL_U8_TYPE_CODE</td>
<td>043</td>
<td>UINT8</td>
</tr>
<tr>
<td>LAL_S_TYPE_CODE</td>
<td>012</td>
<td>REAL4</td>
</tr>
<tr>
<td>LAL_D_TYPE_CODE</td>
<td>013</td>
<td>REAL8</td>
</tr>
<tr>
<td>LAL_C_TYPE_CODE</td>
<td>033</td>
<td>COMPLEX8</td>
</tr>
<tr>
<td>LAL_Z_TYPE_CODE</td>
<td>034</td>
<td>COMPLEX16</td>
</tr>
</tbody>
</table>
3.4.3 Aggregate datatypes

These datatypes store arbitrarily large sets or collections of primitive datatypes. At this level there is no physical interpretation assigned to the objects (such as names or units); the aggregate datatypes simply collect and arrange the primitive datatypes. The following types of aggregate datatypes are defined: vectors, arrays, sequences, vector sequences, and array sequences.

<datatype>Vector

This structure stores an ordered set of \( n \) elements of type \(<datatype>\), which can be any primitive datatype. The data are to be interpreted as being a point in an \( n\)-dimensional vector space. The fields are:

- \texttt{UINT4 length}  The number of data \( n \).
- \texttt{<datatype> *data}  Pointer to the data array. The data are stored sequentially as \( \texttt{data[0,\ldots,n-1]} \).

<datatype>Array

This structure stores a set of elements of type \(<datatype>\), which can be any primitive datatype, arranged as an \( m\)-dimensional array. That is, each element can be thought of as having \( m \) indicies, \( A_{i_0 \cdots i_{m-1}} \), where each index \( i_k \) runs over its own range \( 0,\ldots,n_k - 1 \). The total number of elements is then \( N = n_0 \times \cdots \times n_{m-1} \). In memory the array is “flattened” so that the elements are stored sequentially in a contiguous block. The fields are:

- \texttt{UINT4 Vector *dimLength}  Pointer to a vector of length \( m \), storing the index ranges \( (n_0,\ldots,n_{m-1}) \).
- \texttt{<datatype> *data}  Pointer to the data array. The data element \( A_{i_0 \cdots i_{m-1}} \) is stored as \( \texttt{data[i_m-1 + n_{m-2} \times (i_{m-2} + n_{m-3} \times (\cdots (i_1 + n_0 \times i_0) \cdots))] \); that is, the index of \( \texttt{data[]} \) runs over the entire range of an index \( i_k \) before incrementing \( i_k \).

<datatype>Sequence

This structure stores an ordered set of \( l \) elements of type \(<datatype>\), which can be any primitive datatype. It is identical to \(<datatype>Vector\), except that the elements are to be interpreted as \( l \) consecutive elements rather than the components of an \( l\)-dimensional vector. The fields are:

- \texttt{UINT4 length}  The number of data \( l \).
- \texttt{<datatype> *data}  Pointer to the data array. The data are stored sequentially as \( \texttt{data[0,\ldots,l-1]} \).

<datatype>VectorSequence

This structure stores an ordered set of \( l \) elements of type \(<datatype>Vector\), where \(<datatype>\) can be any primitive datatype. Mathematically the sequence can be written as \( \{ \vec{v}^{(0)},\ldots,\vec{v}^{(l-1)} \} \), where each element \( \vec{v}^{(j)} = (v_j^{(0)},\ldots,v_j^{(n-1)}) \) is a vector of length \( n \). In memory the elements are “flattened”; that is, they are stored sequentially in a contiguous block of memory. The fields are:

- \texttt{UINT4 length}  The number of vectors \( l \).
- \texttt{UINT4 vectorLength}  The length \( n \) of each vector.
- \texttt{<datatype> *data}  Pointer to the data array. The data element \( \vec{v}^{(j)}_i \) is stored as \( \texttt{data[j \times n + i]} \); that is, the index of \( \texttt{data[]} \) runs over the internal index of each vector element before incrementing to the next vector element.
This structure stores an ordered set of \( l \) elements of type `<datatype>Array`, where `<datatype>` can be any primitive datatype. The indexing of an array sequence can get quite complicated; it helps to read first the documentation for data arrays, above. Mathematically the data can be written as a set \( \{ A^{(j)}_{i_0\ldots i_{m-1}} \} \), where the sequence number \( j \) runs from 0 to \( l - 1 \), and each array index \( i_k \) runs over its own range \( 0, \ldots, n_k - 1 \). The total number of data in a given array element is then \( N = n_0 \times \cdots \times n_{m-1} \), and the total number of data in the sequence is \( N \times l \). In memory the array is “flattened” so that the elements are stored sequentially in a contiguous block. The fields are:

- **UINT4 length** The number \( l \) of array elements in the sequence.
- **UINT4 arrayDim** The number of data \( N \) (not the number of indecies \( m \)) in each array element of the sequence.
- **UINT4Vector *dimLength** Pointer to a vector of length \( m \), storing the index ranges \((n_0, \ldots, n_{m-1})\).
- **<datatype> *data** Pointer to the data. The element \( A^{(j)}_{i_0\ldots i_{m-1}} \) is stored as \( \text{data}[j \times N + i_{m-1} + n_{m-2} \times (i_{m-2} + n_{m-3} \times (\cdots (i_1 + n_0 \times i_0) \cdots))] \); that is, the index of `data[]` runs over the internal indecies of each array element before incrementing to the next array element.
3.4.4 Structured datatypes

These datatypes embed primitive and aggregate datatypes inside structures that define their physical meaning. Most of these structures are wrappers for aggregate datatypes that store a physical quantity as a function of time or frequency. Other structures store specific physical information, such as the GPS time, or the factored response function of a filter.

LIGOTimeGPS

This structure stores the time, to nanosecond precision, synchronized to the Global Positioning System time reference. The zero time for the GPS standard is the moment of midnight beginning January 6, 1980, UTC. The LIGOTimeGPS structure can represent times up to 68 years on either side of this epoch. (Note that this is better than an equivalently-sized REAL8 representation of time, which can maintain nanosecond precision only for times within 104 days of its reference point. However, the REAL8 representation does allow one to cover arbitrarily long timescales at correspondingly lower precision.) The fields are:

- INT4 gpsSeconds The number of seconds since the GPS reference time.
- INT4 gpsNanoSeconds The number of nanoseconds since the last GPS second.

The macro LIGOTIMEGPSZERO can be used to statically initialize a LIGOTimeGPS object, for example:

```c
LIGOTimeGPS epoch = LIGOTIMEGPSZERO;
```

LALUnit

This structure stores units in the mksA system (plus Kelvin, Strain, and ADC Count). It also stores an overall power-of-ten scaling factor. The fields are:

- INT2 powerOfTen The power $p$ of ten scaling factor.
- INT2 unitNumerator[LALNumUnits] Array of unit numerators, $N_i$, $i = 0 \ldots LALNumUnits - 1$.
- INT2 unitDenominatorMinusOne[LALNumUnits] Array of unit denominators-minus-one, $D_i$, $i = 0 \ldots LALNumUnits - 1$.

Thus, the units are given by

$$10^p \times \text{m}^{N_0/(1+D_0)} \times \text{kg}^{N_1/(1+D_1)} \times \text{s}^{N_2/(1+D_2)} \times \text{A}^{N_3/(1+D_3)} \times \text{K}^{N_4/(1+D_4)} \times \text{strain}^{N_5/(1+D_5)} \times \text{count}^{N_6/(1+D_6)}$$

(3.1)

The indexes of the units can be specified using the constants LALUnitIndexMeter, LALUnitIndexKiloGram, LALUnitIndexSecond, LALUnitIndexAmpere, LALUnitIndexKelvin, LALUnitIndexStrain, LALUnitIndexADCCount, while LALNumUnits is the total number of units.

<datatype>TimeSeries

This structure represents a sequence of data of type <datatype> (where <datatype> can be any primitive datatype), sampled over uniform time intervals $t_0, t_0 + \Delta t, \ldots, t_0 + l\Delta t$. Essentially this is a <datatype>Sequence with extra fields defining the sample times and the type of data being sampled. The raw data may also have been heterodyned; that is, multiplied by a sinusoid of some frequency $f_0$, low-pass filtered, and resampled, in order to extract the behaviour in a small bandwidth about $f_0$. The fields are:

- CHAR name[LALNameLength] The name of the data series (i.e. the type of data being sampled).
- LIGOTimeGPS epoch The start time $t_0$ of the data series.
- REAL8 deltaT The sampling interval $\Delta t$, in seconds.
- REAL8 f0 The heterodyning frequency $f_0$, in hertz.
- LALUnit sampleUnits The physical units of the quantity being sampled.
- <datatype>Sequence *data The sequence of sampled data.
<datatype>TimeVectorSeries

Like <datatype>TimeSeries, above, except that the sampled data are of type type <datatype>Vector (where <datatype> can be any primitive datatype). The fields are:

CHAR name[LALNameLength] The name of the data series (i.e. the type of data being sampled).

LIGOTimeGPS epoch The start time of the data series.

REAL8 deltaT The sampling interval, in seconds.

REAL8 f0 The heterodyning frequency, in hertz.

LALUnit sampleUnits The physical units of the quantity being sampled.

<datatype>VectorSequence *data The sequence of sampled data.

<datatype>TimeArraySeries

Like <datatype>TimeSeries, above, except that the sampled data are of type type <datatype>Array (where <datatype> can be any primitive datatype). The fields are:

CHAR name[LALNameLength] The name of the data series (i.e. the type of data being sampled).

LIGOTimeGPS epoch The start time of the data series.

REAL8 deltaT The sampling interval, in seconds.

REAL8 f0 The heterodyning frequency, in hertz.

LALUnit sampleUnits The physical units of the quantity being sampled.

<datatype>ArraySequence *data The sequence of sampled data.

<datatype>FrequencySeries

This structure represents a frequency spectrum of data of type <datatype> (where <datatype> can be any primitive datatype), sampled over uniform frequency intervals $f_0, f_0 + \Delta f, \ldots, f_0 + l\Delta f$. Essentially this is a <datatype>Sequence with extra fields defining the sample frequencies, the timestamp of the spectrum, and the type of data being sampled. The fields are:

CHAR name[LALNameLength] The name of the data series (i.e. the type of data being sampled).

LIGOTimeGPS epoch The start time of the time series from which the spectrum was calculated.

REAL8 f0 The lowest frequency $f_0$ being sampled, in hertz.

REAL8 deltaF The frequency sampling interval $\Delta f$, in hertz.

LALUnit sampleUnits The physical units of the quantity being sampled.

<datatype>Sequence *data The sequence of sampled data.

<datatype>ZPGFilter

This structure stores the complex frequency response of a filter or transfer function in a factored form, where <datatype> can be either COMPLEX8 or COMPLEX16. One defines a (dimensionless) complex frequency variable $\zeta(f\Delta t)$, where $\Delta t$ is the time sampling interval of the data to which the filter will be applied (in the case of a digital filter), or some other reference timescale (in the case of an analog filter). The complex response function can then be given (or approximated) as $H(f) = g \times \prod_k (\zeta - z_k)/\prod_l (\zeta - p_l)$, where $z_k$ are the complex zeros, $p_l$ are the complex poles, and $g$ is the complex gain of the response function. Some common complex frequency representations are the $z$-plane representation $\zeta(f\Delta t) = \exp(2\pi if\Delta t)$, which maps the Nyquist interval $f \in [0, 1/(2\Delta t))$ onto the upper-half unit circle in $\zeta$, and the $w$-plane representation $\zeta(f\Delta t) = \tan(\pi f\Delta t)$, which maps the Nyquist interval onto the positive real axis in $\zeta$. The fields of <datatype>ZPGFilter are:
CHAR name[LALNameLength]  The name of the filter or transfer function. This should also mention its complex frequency representation.

REAL8 deltaT  The sampling time or reference timescale $\Delta t$ for the filter, in seconds. If zero, it will be treated as being equal to the sampling interval of the data being filtered.

<typename>Vector *zeros  Pointer to a vector storing the zeros $z_k$ of the filter.

<typename>Vector *poles  Pointer to a vector storing the poles $p_k$ of the filter.

<typename> gain  The gain $g$ of the filter.
3.4.5 The LAL universal status structure \texttt{LALStatus}

This structure is the means by which LAL functions report their success or failure; it provides a useful mechanism for tracking progress and errors through nested function calls. The error reporting structure is a linked list of \texttt{LALStatus} structures, with each node corresponding to a given function in the current calling sequence. When a function terminates successfully, its node is dropped from the list. If a function encounters an error, it must still return control to the calling routine, reporting the error through its \texttt{LALStatus}. The calling routine must either deal with the error (pruning the linked list if it succeeds), or else return an error itself. A fatal error will thus return a linked list of \texttt{LALStatus} structures to the top-level routine, where the tail of the list identifies the source of the error, and the intermediate nodes identify the sequence of nested function calls that led to the error. The fields of the \texttt{LALStatus} are as follows:

\begin{verbatim}
INT4 statusCode  A numerical code identifying the type of error, or 0 for nominal status.
const CHAR *statusDescription  A description of the current status or error.
volatile const CHAR *Id  The RCS ID string of the source file of the current function.
const CHAR *function  The name of the current function.
const CHAR *file  The name of the source file of the current function.
INT4 line  The line number in the source file where the current statusCode was set.
LALStatus *statusPtr  Pointer to the next node in the list; NULL if this function is not reporting a subroutine error.
INT4 level  The current level in the nested calling sequence.
\end{verbatim}
3.5 Header LALStatusMacros.h

Provides macros for handling the LAL status structure.

Synopsis

```
#include <lal/LALStatusMacros.h>
```

This header provides macros and functions for tracking and reporting the runtime status of a program. The intent is simultaneously to standardize the error reporting, and to make the reporting as transparent as possible to people coding individual routines.

3.5.1 Status-reporting objects

LAL routines make use of two objects in reporting their current status: the status structure `LALStatus`, and the global integer `lalDebugLevel`. These two objects are described in the following sections.

The `LALStatus` structure

LAL routines store their current execution status in a linked list of structures of type `LALStatus`, with each node in the list representing a subroutine in the current calling sequence. The `LALStatus` structure is described in Sec. 3.4.5 of the header `LALDatatypes.h`, but for completeness, we explain its fields below:

- **INT4 statusCode**: A code indicating the exit status of a function. 0 represents a normal exit. Negative values are reserved for certain standard error types. The authors of individual functions should assign positive values to the various ways in which their code can fail.

- **const CHAR *statusDescription**: An explanatory string corresponding to the numerical status code.

- **volatile const CHAR *Id**: A character string identifying the source file and version number of the function being reported on.

- **const CHAR *function**: The name of the function.

- **const CHAR *file**: The file name of the `.c` file containing the function code.

- **INT4 line**: The line number in the `.c` file of the instruction where any error was reported.

- **LALStatus *statusPtr**: A recursive pointer to another status pointer. This structure is used to report an error in a subroutine of the current function. Thus if an error occurs in a deeply-nested routine, the status structure returned to the main program will be the head of a linked list of status structures, one for each nested level, with the tail structure reporting the actual error that caused the overlying routines to fail.

- **INT4 level**: The nested-function level where any error was reported.

In almost all circumstances the programmer will not have to access this structure directly, relying instead on the macros defined in this header. The exception is the `statusCode` field, which the programmer may want to query directly.

The `statusCode` field is set to a nonzero value any time an error condition arises that would lead to abnormal termination of the current function. Programmers can assign positive error codes to the various types of error that may be encountered in their routines. Additionally, the following following status codes are reserved to report certain standard conditions:
<table>
<thead>
<tr>
<th>Code</th>
<th>Message</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Nominal execution; the function returned</td>
<td>successfully.</td>
</tr>
<tr>
<td>-1</td>
<td>Recursive error</td>
<td>The function aborted due to failure of a subroutine.</td>
</tr>
<tr>
<td>-2</td>
<td>INITSTATUS: non-null status pointer</td>
<td>The status structure passed to the function had a non-NULL statusPtr field,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>which blocks the function from calling subroutines (it is symptomatic of</td>
</tr>
<tr>
<td></td>
<td></td>
<td>something screwy going on in the calling routine).</td>
</tr>
<tr>
<td>-4</td>
<td>ATTACHSTATUSPTR: memory allocation error</td>
<td>The function was unable to allocate a statusPtr field to pass down to a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>subroutine.</td>
</tr>
<tr>
<td>-8</td>
<td>DETACHSTATUSPTR: null status pointer</td>
<td>The statusPtr field could not be deallocated at the end of all subroutine</td>
</tr>
<tr>
<td></td>
<td></td>
<td>calls; one of the subroutines must have lost it or set it to NULL.</td>
</tr>
<tr>
<td>-16</td>
<td>INITSTATUS: non-zero xlalErrno</td>
<td>The xlalError variable is non-zero, which suggests that an error in an</td>
</tr>
<tr>
<td></td>
<td></td>
<td>XLAL routine has occurred and has not been handled.</td>
</tr>
<tr>
<td>-16</td>
<td>RETURN: untrapped XLAL error code</td>
<td>The xlalError variable is non-zero, which indicates that an error in an</td>
</tr>
<tr>
<td></td>
<td></td>
<td>XLAL routine has occurred and has not been handled.</td>
</tr>
</tbody>
</table>

The `lalDebugLevel` variable is a global variable, set at runtime, that determines how much and what kind of debugging information will be reported. It is declared as an `extern int` in the header `LALStatusMacros.h`, and is therefore accessible in any standard LAL module that includes this header. Note, however, that it is declared to be of the C type `int`, which is usually but not always a 32-bit integer (on some systems it may only be 16 bits).

The value of `lalDebugLevel` should be thought of not as a number, but as a bit mask, wherein each bit in the binary representation turns on or off a specific type of status reporting. At present, there are five types of status reporting, each associated with a bit in `lalDebugLevel`.

### Error messages
Tell the operator that a computation has terminated abnormally, and has failed to produce an acceptable result. Normally this is associated with assigning a non-zero `statusCode`; an error message is printed automatically whenever a function exits with non-zero `statusCode`.

### Warning messages
Tell the user that a computation is working, but with unusual behaviour that might indicate an unreliable or meaningless result. Warnings do not normally result in a non-zero `statusCode`.

### Information messages
Tell the operator that the computation is proceeding as expected, and simply provide additional information about its progress.

### Tracing messages
Are printed automatically a subroutine is called or returned; they simply track the current sequence of function calls.

### Memory information messages
Are a special type of information message; they tell the operator when and how much memory is allocated or freed from the memory heap.

The module `LAEError.c` defines functions for printing each of these types of status message. Each type of message is turned on by setting the corresponding bit in `lalDebugLevel` to 1, and is suppressed by setting the bit to 0. This header file contains flags with numerical values designed to switch on the appropriate bits. Combinations of bits can be switched on by combining these flags using the bitwise-or operator, `|`. The flags are defined as follows:
### 3.5.2 Using the status tools

The following summarizes everything the common programmer needs to know in order to follow LAL standard error reporting. It can be treated as a primer on LAL coding conventions.

#### LAL function calls

All functions should have return type void. The first argument of any function should be a pointer to a structure of type LALStatus. Thus:

```c
void MyFunction( LALStatus *stat, ... )
```

Since the function has no return code, it must report all errors or failure through the status structure. A function that is passed a NULL pointer in place of the status pointer should terminate the program with a SIGABRT signal, as this is its only way to report the error. However, this is one of the few circumstances under which a function should deliberately raise a signal. In all other cases the error should be trapped, reported in the status structure, and control returned to the calling routine.

#### Assigning an RCS $Id$ string

Every source file should have a unique character string identifying that version of that file. The standard convention, for a file MyFile.c, is to declare a string MYFILEC at the top of the module using the macro NRCSID() (defined in the include file LALRCSID.h):

```c
NRCSID( MYFILEC, $Id$ );
```

where $Id$ is expanded by RCS to give the full name and version number of the source file.

#### Initializing the status structure

The first instruction in any function, after variable declarations, should be the macro INITSTATUS(), which takes three arguments: the function’s status pointer, the function name (a string literal) and the module’s RCS $Id$ string.

```c
INITSTATUS( stat, "MyFunction", MYFILEC );
```
This macro checks that a valid status pointer has been passed to the function, and if so, initializes the other fields to indicate (by default) nominal execution. If stat is null, the macro causes the program to terminate with a SIGABRT signal, as described above.

### Normal return from a function

Upon completion, the function should issue the macro `RETURN()`, which takes one argument: the function’s status pointer.

```
RETURN( stat );
```

This takes the place of any return statements. If `stat->statusCode` is non-zero, the macro calls `LALError()` (see `LALError.c`) to log `stat->statusDescription` and other information, depending on implementation and the value of `lalDebugLevel`. Typically `RETURN()` is used only for successful completion, with other macros `ABORT()`, `ASSERT()`, `CHECKSTATUSPTR()`, and `TRY()` being used to report failure. However, it is possible for the programmer to assign the fields of `*stat` by hand, and then issue `RETURN()`.

### Abnormal return from a function

The standard method to terminate a function unsuccessfully is with the `ABORT()` macro, which takes three arguments: the status pointer, the status code, and the status description string. Normally the various error codes and descriptions will be constants defined in the function’s header file `MyHeader.h`:

```
ABORT( stat, MYHEADERH_EMYERR, MYHEADERH_MSGEMYERR );
```

where the error code `MYHEADERH_EMYERR` and the error message `MYHEADERH_MSGEMYERR` are defined in `MyHeader.h`. This standard LAL naming convention for error messages prevents namespace conflicts between different header files. Like `RETURN()`, `ABORT()` correctly handles any status logging required by the implementation and the `lalDebugLevel`. Note that `ABORT()` does not raise a SIGABRT signal, but instead returns control to the calling routine.

### Error checking within a function

Another way to indicate an unsuccessful termination is with the macro `ASSERT()`, which takes as arguments a test statement, a status pointer, a status code, and a status description. The statement

```
ASSERT( assertion, stat, MYHEADERH_EMYERR, MYHEADERH_MSGEMYERR );
```

One subtle but important point is that the `ASSERT()` should be used only to trap coding errors, rather than runtime errors, which would be trapped using `ABORT()`. In other words, the assertion should always test true in the final debugged program. This is vital because certain compilation flags will remove all `ASSERT()` macros at compile time, in order to speed execution of the final code. See Sec. 3.5.3, below.

Programmers should also be aware that using `ASSERT()` to exit a function in normal runtime can have serious side effects. For example, it is an error to allocate dynamic memory to local variables in a function and then fail to free it before returning. Thus, if you have dynamically allocated memory, you cannot then use `ASSERT()` for runtime error checking, as this does not permit you to free the memory before returning. Instead, you must explicitly check the assertion, and, if it fails, free the memory and call `ABORT()`.

### Calling subroutines

If the function is to call other LAL functions as subroutines, four more macros are used to report possible errors arising in these routines. The macros are `ATTATCHSTATUSPTR()`, `DETATCHSTATUSPTR()`, `CHECKSTATUSPTR()`, and `TRY()`. The usage of these macros is as follows.

1. First, before any subroutines are called, the function must call the macro `ATTATCHSTATUSPTR()` which takes as its argument the status pointer of the current function:

```
ATTATCHSTATUSPTR( stat );
```
This allocates `stat->statusPtr`, which is the status pointer that will be handed down into any and all subroutines. If the pointer has already been allocated, `ATTACHSTATUSPTR()` will raise a `SIGABRT`, as this is symptomatic of a coding error.

In most cases `ATTACHSTATUSPTR()` need only be called once in a given function, immediately after `INITSTATUS()`, no matter how many subroutine calls that function makes. The exception is if the function deals with (or ignores) errors reported by its subroutines. In that case, the function should detach the status pointer using `DETACHSTATUSPTR()` (below), and then re-attach it.

The macro `ATTACHSTATUSPTR()` sets the status code to be −1 and the status message to be "Recursive error". These flags are unset when `DETACHSTATUSPTR()` (below) is called. This is so that a use of `RETURN()` prior to detaching the status pointer will yield an error.

2. When a subroutine is called, it should be handed the `statusPtr` field of the calling function's status structure, to report its own errors. The calling function should test the returned status code, and either attempt to deal with any abnormal returns, or abort with status code −1. The macro `CHECKSTATUSPTR()` simplifies the latter case. It takes one argument: the status pointer of the current function (not the subroutine).

```c
MySubroutine( stat->statusPtr, ... );
CHECKSTATUSPTR( stat );
```

The `TRY()` macro is a somewhat more streamlined approach but with equivalent results. It takes two arguments. The first is the subroutine call, and the second is the status pointer. Thus:

```c
TRY( MySubroutine( stat->statusPtr, ... ), stat );
```

The only practical difference between these two approaches is that `TRY()` also reports the name of the failed subroutine call when logging errors.

Similar caveats apply when using `CHECKSTATUSPTR()` and `TRY()` as when using `ASSERT()`, in that these macros can force an immediate return with no additional housekeeping instructions. For instance, if you have dynamically-allocated local memory, you should explicitly check the `statusPtr->statusCode` field to see if a subroutine failed, then free the memory and call `ABORT()` to exit.

If the calling routine attempts to work around an error reported from a subroutine, and the attempt fails, the routine should not use `CHECKSTATUSPTR()` to exit with status code −1. Instead, it should call `ABORT()` with an appropriate (positive) code and message to indicate how the attempted workaround failed.

3. After all subroutines have been called, but before any `RETURN()` statement, the function must call the `DETACHSTATUSPTR()` macro, with the status pointer of the current function (not the subroutines) as its argument:

```c
DETACHSTATUSPTR( stat );
```

This simply deallocates `stat->statusPtr` (and any subsequent structures in the list), and sets it to `NULL`. It is an error to exit the function with non-`NULL` `statusPtr`, unless the exit was due to a subroutine failure. `ABORT()` and `ASSERT()` check for this automatically; the only place you normally need to call `DETACHSTATUSPTR()` is immediately before `RETURN()`. This macro also sets the status code and the status message to nominal values.

Additionally, if a function successfully works around an error reported by a subroutine, it should call `DETACHSTATUSPTR()` and `ATTACHSTATUSPTR()` to create a fresh status pointer before calling another subroutine.

### Cleaning up after subroutine failure

Although they are convenient, the `TRY()` and `CHECKSTATUSPTR()` macros have a serious drawback in that they may cause the calling function to return immediately. If the calling function had previously allocated any local memory storage, this memory will be cast adrift, with no means of accessing or subsequently freeing it (short of terminating the runtime process). Such a memory leak is a violation of the LAL function standard.
The macros `BEGINFAIL()` and `ENDFAIL()` allow a function to test the return code of a subroutine, and, if that indicates a failure, to execute one or more “cleanup” instructions before itself returning. Each macro takes a single argument: the current function’s status pointer. The macros must occur in matched pairs, and use the same syntax as a `do ... while` statement: they either span a single instruction, or a block of instructions enclosed in braces.

For example, if a function had allocated memory to some pointer `localPointer`, any subsequent call to a subroutine `LALSubroutine()` would take the following form:

```c
LALSubroutine( stat->statusPtr, ... );
BEGINFAIL( stat )
    LALFree( localPointer );
ENDFAIL( stat );
```

For another example, if a function had to create three vectors `*vector1, *vector2, *vector3`, the allocation would look something like this:

```c
TRY( LALSCreateVector( stat->statusPtr, &vector1, 100 ), stat );
LALSCreateVector( stat->statusPtr, &vector2, 100 );
BEGINFAIL( stat )
    TRY( LALSDestroyVector( stat->statusPtr, &vector1 ), stat );
ENDFAIL( stat );
LALSCreateVector( stat->statusPtr, &vector3, 100 );
BEGINFAIL( stat ) {
    TRY( LALSDestroyVector( stat->statusPtr, &vector1 ), stat );
    TRY( LALSDestroyVector( stat->statusPtr, &vector2 ), stat );
} ENDFAIL( stat );
```

As indicated above, the cleanup instructions can include calls to other LAL routines. The `BEGINFAIL( stat )` macro call first checks `stat->statusPtr` to see if a subroutine error has occurred. If it has, the macro detaches and saves that pointer, then attaches a new `stat->statusPtr` to be used in calls to the cleanup routines. After the cleanup instructions have been executed, the `ENDFAIL( stat )` macro call reattaches the saved status pointer and returns with a subroutine error code. In this way, the returned status list indicates where the original failure occurred, rather than giving an uninformative report from the last cleanup routine.

Of course a second failure in one of the cleanup routines can cause serious problems. If the routine was called using a `TRY()` macro, it will force an immediate return from the calling function, with a status code and status list indicating how the cleanp routine failed. The original status list saved by `BEGINFAIL()` is lost. While this loss does constitute a memory leak, the failure of a cleanup routine in itself indicates that there are serious problems with the memory management.

It is possible to nest `BEGINFAIL() . . . ENDFAIL()` blocks, but this is unlikely to serve any useful purpose. Once cleanup routines start to fail, it is probably beyond the scope of the LAL function to deal with the resulting memory leaks.

**Issuing status messages**

The module `LALError.c` defines the functions `LALError()`, `LALWarning()`, `LALInfo()`, and `LALTrace()` to issue various types of status message. This is the preferred means of printing status messages, since each type of message can be activated or suppressed by setting `lalDebugLevel` appropriately. In fact, `LALError()` and `LALTrace()` are called automatically by the status macros whenever they are required, so most LAL modules will explicitly invoke only the `LALWarning()` and `LALInfo()` functions.

`LALStatusMacros.h` provides a macro, `REPORTSTATUS()`, which is used to report the current state of the `LALStatus` list. It takes a status pointer as its argument:

```c
REPORTSTATUS( stat );
```

This macro iteratively prints the contents of `stat` and all subsequent structures in the list to the error log.

The action of `REPORTSTATUS()` is not suppressed by any value of `lalDebugLevel`. Therefore, as a rule, it should only be called by test programs, not by LAL routines intended for use in production code.
3.5. Header LALStatusMacros.h

Setting the initial LALStatus structure and global lalDebugLevel

As mentioned above, any module including LALStatusMacros.h includes the global variable lalDebugLevel as an extern int. At least one module in the final executable program must have a global declaration of int lalDebugLevel (not extern int), and assign lalDebugLevel a value. In most cases lalDebugLevel will be declared in the module containing the main() function, and will be assigned a value on declaration or from command-line arguments to main(). Alternatively, if the LAL functions are to be embedded in a non-LAL program, lalDebugLevel can be declared and set in the topmost module that calls LAL functions.

A LALStatus structure should also be declared as a local variable in the main() function of a LAL program, or in the topmost function calling LAL functions withing a non-LAL program, to pass in its LAL function calls. The structure must be empty (all fields set to zero) before being passed into a function. The LALStatus structure need only be declared and initialized once, no matter how many LAL functions are called.

Thus a typical LAL program might look something like the following:

```c
int lalDebugLevel = 1;

int main( int argc, char **argv )
{
    static LALStatus stat;
    MyFunction( &stat );
    REPORTSTATUS( &stat );
    return stat.statusCode;
}
```

Please note that all status macros described above can force a return from the calling routine. This is a Bad Thing if the calling routine is main(), since main() must normally return int rather than void. It is therefore recommended that none of these macros except REPORTSTATUS() be used at the top level.

Non-conformant functions

These standards apply only to functions that will be publicly available in the LAL libraries. Within a module, a programmer may define and use subroutines that do not conform to the LAL function standards, provided these routines are only visible within that module. Such functions should be declared as static to ensure this. A publicly-visible non-conformant function requires special dispensation.

3.5.3 Compilation flags

LAL provides two flags that can be used to exclude or modify debugging code at compile time. Although these flags are typically #defined or #undefined globally and can affect many modules (notably modules in the support package), their primary effect is on the debugging and status-reporting tools defined in this header. The two flags are named NDEBUG and NOLALMACROS.

The NDEBUG flag

Setting the NDEBUG (or LAL_NDEBUG) flag turns off debugging and error-reporting code, in order to get condensed production-line programs. As far as error reporting is concerned, setting the NDEBUG flag at compile time is similar to setting lalDebugLevel equal to zero at runtime, in that it suppresses all status messages and memory leak detection. However, the NDEBUG flag accomplishes this by telling the compiler preprocessor to remove the relevant code from the object file, thus eliminating frequent and unnecessary tests on lalDebugLevel. When debugging is turned off, the global integer variable lalNoDebug is non-zero; otherwise it is zero.

Compiling with the NDEBUG flag set also removes all ASSERT() macros from the object code, in keeping with the philosophy that ASSERT() statements should only be used to catch coding bugs, not runtime errors.

The NOLALMACROS flag

Setting the NOLALMACROS flag replaces the status-handling macros described above with actual functions that accomplish the same results. These functions are defined in the module LALError.c. Function calls introduce computational and memory overheads that are absent in macros, since macro replacement occurs
at compile time. However, there are circumstances in which one might want to use function calls rather than
macro replacement.

   For example, debuggers typically cannot step through the individual instructions within a macro. If a
   conflict somehow arose between a particular piece of code and its status macros, this conflict would be easier
to catch and resolve by replacing the macros with function calls into which the debugger could step.

Using the compilation flags

There are three ways to set these flags when compiling LAL programs or libraries.

   When compiling your own modules, the flags can be set using one or more \#define statements within
the module or its header file:

\#define NDEBUG
\#define NOLALMACROS

To restrict the scope of these flags, they should later be unset using the corresponding \#undef statements.

   Alternatively, these can be set in the Makefile or when compiling. The syntax for most UNIX C compilers
is something like the following:

   > gcc ... -DNDEBUG -DNOLALMACROS ...

   If you want to compile a large number of modules, or the entire library, under the effects of one or more
of these flags, you will not want to go through and modify every header or Makefile. Instead, you may
add either -DNDEBUG or -DNOLALMACROS (or both) to the environment variable CPPFLAGS. They will then
automatically be set for all compilations done in that environment. The command for doing this in sh or
bash shells is:

   > CPPFLAGS="$CPPFLAGS -DNDEBUG -DNOLALMACROS"

   while in csh or tcsh shells it is:

   > setenv CPPFLAGS "$CPPFLAGS -DNDEBUG -DNOLALMACROS"

   Note that if you plan to do further LAL code development on the same system, you may want to keep two
versions of the library around: one with the flag(s) set and one without.

Notes

Why are the status handling routines written as macros rather than functions? There are three good reasons.

   First, many of the handling routines must be able to force an exit from the function calling them. This
cannot be done if the routine is in its own function, except by raising signal flags (which is a Bad Thing
according to LAL standards).

   Second, it is useful for these routines to assign a status structure’s file and line fields using the __FILE__
and __LINE__ macros. If the routine is its own function, then these will just give the file and line number
where the error handling routine is defined. If the routine is a macro, then these will give the file and line
number where the macro was called, which is much more interesting.

   Third, by expanding macros at compile time, the runtime performance of the resulting code is marginally
better. Most of these macros will, under nominal conditions, reduce to a single conditional test of an
integer value, with no additional overhead from function calling and parameter passing. Thus programmers
can be encouraged to include extensive error trapping in all their routines, without having to worry about
compromising performance.

   It should be mentioned that, for these reasons, compiling a module with the NOLALMACROS flag above
does not actually eliminate the status handling macros. Rather, the macros are modified to call specialized
functions that do most (but not all) of the processing.

Example: A LAL primer

The following sections give a sample program program LALPrimerTest.c, along with its supporting header
file LALPrimer.h and function module LALPrimer.c. The program itself is trivial to the point of silliness:
it takes two arguments from the command line and computes their ratio. (Optionally it can take a third
command line argument to set the lalDebugLevel.) It is intended simply to illustrate how to use the LAL
status structure and macros in an actual, complete piece of code.

   For a more fully developed sample program, see the package hello. That package also demonstrates how
to document a package using the autodocumentation utilities, which the LALPrimer routines ignore.
Author: Creighton, J. D. E. and Creighton, T. D.
$Id: LALStatusMacros.h,v 1.25 2007/06/05 14:41:52 bema Exp$
3.5.4 Sample header: LALPrimer.h

```c
#ifndef _LALPRIMER_H /* Double-include protection. */
#define _LALPRIMER_H

#include <lal/LALDatatypes.h>

#undef _cplusplus /* C++ protection. */
extern "C" {
#define LALPRIMERH_ENULL 1
#define LALPRIMERH_EDIV0 2
#define LALPRIMERH_MSGENULL "Null pointer"
#define LALPRIMERH_MSGEDIV0 "Division by zero"

void
REAL4Invert( LALStatus *status, REAL4 *output, REAL4 input );

void
REAL4Divide( LALStatus *status, REAL4 *output, REAL4 numer, REAL4 denom);

#undef _cplusplus
}
#endif /* C++ protection. */
#undef Double-include protection. */
```

NRCSID( LALPRIMERH, "$Id: LALPrimer.h,v 1.5 2007/06/08 14:41:52 bema Exp "$ );
3.5.5 Sample module: LALPrimer.c

#include <lal/LALStdlib.h>
#include <lal/LALPrimer.h>

NRCSID( LALPRIMERC, "$Id: LALPrimer.c,v 1.5 2007/06/08 14:41:53 bema Exp $" );

void
REAL4Invert( LALStatus *stat, REAL4 *output, REAL4 input )
  /* Computes the inverse of a REAL4 number. */
{
  INITSTATUS( stat, "REAL4Invert", LALPRIMERC );
  /* This traps coding errors in the calling routine. */
  ASSERT( output != NULL, stat, LALPRIMERH_ENULL, LALPRIMERH_MSGENULL );
  /* This traps runtime errors. */
  if ( input == 0.0 )
    ABORT( stat, LALPRIMERH_EDIV0, LALPRIMERH_MSGEDIV0 );
  *output = 1.0/input;
  RETURN( stat );
}

void
REAL4Divide( LALStatus *stat, REAL4 *output, REAL4 numer, REAL4 denom )
  /* Computes the ratio of two REAL4 numbers. */
{
  INITSTATUS( stat, "REAL4Divide", LALPRIMERC );
  ATTATCHSTATUSPTR( stat );
  TRY( REAL4Invert( stat->statusPtr, output, denom ), stat );
  *output *= numer;
  DETATCHSTATUSPTR( stat );
  RETURN( stat );
}
3.5.6 Sample program: LALPrimerTest.c

```c
#include <stdlib.h>
#include <lal/LALStdlib.h>
#include <lal/LALPrimer.h>

NRCSID( LALPRIMERTESTC, "$Id: LALPrimerTest.c,v 1.5 2007/06/08 14:41:54 bema Exp "$ );

int lalDebugLevel = 0;

int main( int argc, char **argv )
    /* Divides two numbers given on the command input line. */
{
    static LALStatus stat;
    REAL4 ratio;

    /* Parse input line. */
    if ( argc == 4 )
        lalDebugLevel = atoi( argv[3] );
    else if ( argc != 3 )
    {
        fprintf( stderr, "Usage: %s numer denom [ lalDebugLevel ]\n",
                 argv[0] );
        return 0; /* so that test script won't fail */
    }

    /* Compute ratio. */
    REAL4Divide( &stat, &ratio, atof( argv[1] ), atof( argv[2] ) );
    if ( stat.statusCode && ( lalDebugLevel > 0 ) )
        fprintf( stderr,
                 "Error[0] 1: program %s, file %s, line %i, %s\n"
                 " Function REAL4Divide() failed\n",
                 argv[0], __FILE__, __LINE__, LALPRIMERTESTC );

    /* Print result. */
    if ( !stat.statusCode )
        fprintf( stdout, "\nRatio = %f\n", ratio );
    REPORTSTATUS( &stat );
    return stat.statusCode;
}
```
3.6 Header **LALConstants.h**

Provides standard numerical constants for LAL.

**Synopsis**

```
#include <lal/LALConstants.h>
```

This header defines a number of useful numerical constants for use in LAL routines. These constants come in three basic flavours: arithmetic and mathematical constants, fundamental (or defined) physical constants, and measured astrophysical and cosmological parameters.

Note that, unlike the other headers in the *std* package, this header is *not* included automatically by the header **LALStdlib.h**. Include it explicitly if you need any of these constants.

**Mathematical constants**

The following constants define the precision and range of floating-point arithmetic in LAL. They are taken from the IEEE standard 754 for binary arithmetic. All numbers are dimensionless.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAL_REAL4_MANT</td>
<td>24</td>
<td>Bits in REAL4 mantissa</td>
</tr>
<tr>
<td>LAL_REAL4_MAX</td>
<td>$3.40282347 \times 10^{38}$</td>
<td>Largest REAL4</td>
</tr>
<tr>
<td>LAL_REAL4_MIN</td>
<td>$1.17549435 \times 10^{-38}$</td>
<td>Smallest positive REAL4</td>
</tr>
<tr>
<td>LAL_REAL4_EPS</td>
<td>$1.19209290 \times 10^{-7}$</td>
<td>$2^{-(\text{LAL_REAL4_MANT} - 1)}$</td>
</tr>
<tr>
<td>LAL_REAL8_MANT</td>
<td>53</td>
<td>Bits in REAL8 mantissa</td>
</tr>
<tr>
<td>LAL_REAL8_MAX</td>
<td>$1.7976931348623157 \times 10^{308}$</td>
<td>Largest REAL8</td>
</tr>
<tr>
<td>LAL_REAL8_MIN</td>
<td>$2.250738585072014 \times 10^{-308}$</td>
<td>Smallest positive REAL8</td>
</tr>
<tr>
<td>LAL_REAL8_EPS</td>
<td>$2.204460492503131 \times 10^{-16}$</td>
<td>$2^{-(\text{LAL_REAL8_MANT} - 1)}$</td>
</tr>
</tbody>
</table>

**LAL_REAL4_EPS** and **LAL_REAL8_EPS** can be thought of as the difference between 1 and the next representable REAL4 or REAL8 number.

The following are fundamental mathematical constants. They are mostly taken from the GNU C *math.h* header (with the exception of **LAL_TWOPi**, which was computed using Maple). All numbers are dimensionless.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAL_E</td>
<td>2.7182818284590452353602874713526625</td>
<td>$e$</td>
</tr>
<tr>
<td>LAL_LOG2E</td>
<td>1.44269504088898634073599246810018922</td>
<td>$\log_2 e$</td>
</tr>
<tr>
<td>LAL_LOG10E</td>
<td>0.434294481903251827651128918916055</td>
<td>$\log_{10} e$</td>
</tr>
<tr>
<td>LAL_LN2</td>
<td>0.69314718055994530491723214581766</td>
<td>$\log_2 2$</td>
</tr>
<tr>
<td>LAL_LN10</td>
<td>2.30258509299445684017991454683642</td>
<td>$\log_{10} 10$</td>
</tr>
<tr>
<td>LAL_SQRT2</td>
<td>1.414213562373095048801688724209681</td>
<td>$\sqrt{2}$</td>
</tr>
<tr>
<td>LAL_SQRT1_2</td>
<td>0.7071067811865475244008443621048490</td>
<td>$1/\sqrt{2}$</td>
</tr>
<tr>
<td>LAL_GAMMA</td>
<td>0.5772156649015328606065120900824024</td>
<td>$\gamma$</td>
</tr>
<tr>
<td>LAL_PI</td>
<td>3.141592653589793238462433832795029</td>
<td>$\pi$</td>
</tr>
<tr>
<td>LAL_TWOPi</td>
<td>6.283185307179586476925867665590058</td>
<td>$2\pi$</td>
</tr>
<tr>
<td>LAL_PI_2</td>
<td>1.57079632674896619231231691397514</td>
<td>$\pi/2$</td>
</tr>
<tr>
<td>LAL_PI_4</td>
<td>0.78539816339748483096156608458198757</td>
<td>$\pi/4$</td>
</tr>
<tr>
<td>LAL_1_PI</td>
<td>0.3183098861837906715377675267450287</td>
<td>$1/\pi$</td>
</tr>
<tr>
<td>LAL_2_PI</td>
<td>0.6366197723675813430755350534900574</td>
<td>$2/\pi$</td>
</tr>
<tr>
<td>LAL_2_SQRTPI</td>
<td>1.1283791670955125738961589031215452</td>
<td>$2/\sqrt{\pi}$</td>
</tr>
<tr>
<td>LAL_PI_180</td>
<td>1.7453292519943295769236907684886127</td>
<td>$180/\pi$</td>
</tr>
<tr>
<td>LAL_180_PI</td>
<td>57.29577951308232087679815481465170</td>
<td>$\pi/180$</td>
</tr>
</tbody>
</table>

**Physical constants**

The following physical constants are defined to have exact values. The values of $c$ and $g$ are taken from [1], $\rho_{\text{atm}}$ is from [2], while $\epsilon_0$ and $\mu_0$ are computed from $c$ using exact formulae. They are given in the SI units shown.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>2.99792458 \times 10^8 m/s</td>
<td>$c$</td>
</tr>
<tr>
<td>g</td>
<td>9.80665 \times 10^{-1} m/s</td>
<td>$g$</td>
</tr>
<tr>
<td>$\rho_{\text{atm}}$</td>
<td>1.225 \times 10^{-3} kg/m^3</td>
<td>$\rho_{\text{atm}}$</td>
</tr>
<tr>
<td>$\epsilon_0$</td>
<td>8.8541878176 \times 10^{-12} C^2/N m^2</td>
<td>$\epsilon_0$</td>
</tr>
<tr>
<td>$\mu_0$</td>
<td>4\pi \times 10^{-7} H/m</td>
<td>$\mu_0$</td>
</tr>
</tbody>
</table>
### Astrophysical parameters

The following parameters are derived from measured properties of the Earth and Sun. The values are taken from [1], except for the obliquity of the ecliptic plane and the eccentricity of Earth's orbit, which are taken from [2]. All values are given in the SI units shown.

#### Table 1: Astrophysical parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAL_REARTH_SI</td>
<td>6.378140 × 10^9 m</td>
<td>Earth equatorial radius</td>
</tr>
<tr>
<td>LAL_AWGS84_SI</td>
<td>6.378137 × 10^9 m</td>
<td>Semimajor axis of WGS-84 Reference Ellipsoid</td>
</tr>
<tr>
<td>LAL_BWGS84_SI</td>
<td>6.356752314 × 10^6 m</td>
<td>Semiminor axis of WGS-84 Reference Ellipsoid</td>
</tr>
<tr>
<td>LAL_MASSH_SI</td>
<td>5.97370 × 10^24 kg</td>
<td>Earth mass</td>
</tr>
<tr>
<td>LAL_IEARTH</td>
<td>0.000992804 rad</td>
<td>Obliquity of the ecliptic (2000)</td>
</tr>
<tr>
<td>LAL_EARTH</td>
<td>0.0167</td>
<td>Earth orbital eccentricity</td>
</tr>
<tr>
<td>LAL_RSUN_SI</td>
<td>6.960 × 10^8 m</td>
<td>Solar equatorial radius</td>
</tr>
<tr>
<td>LAL_MSSUN_SI</td>
<td>1.98892 × 10^30 kg</td>
<td>Solar mass</td>
</tr>
<tr>
<td>LAL_MRSUN_SI</td>
<td>1.47662504 × 10^3 m</td>
<td>Geometrized solar mass (length)</td>
</tr>
<tr>
<td>LAL_MTSUN_SI</td>
<td>4.92549095 × 10^-6 s</td>
<td>Geometrized solar mass (time)</td>
</tr>
<tr>
<td>LAL_LSUN_SI</td>
<td>3.846 × 10^26 W</td>
<td>Solar luminosity</td>
</tr>
<tr>
<td>LAL_AU_SI</td>
<td>1.4959787066 × 10^11 m</td>
<td>Astronomical unit</td>
</tr>
<tr>
<td>LAL_PC_SI</td>
<td>3.0856775807 × 10^16 m</td>
<td>Parsec</td>
</tr>
<tr>
<td>LAL_YRTROP_SI</td>
<td>31.556.925.2 s</td>
<td>Tropical year (1994)</td>
</tr>
<tr>
<td>LAL_YRSID_SI</td>
<td>31.558.149.8 s</td>
<td>Sidereal year (1994)</td>
</tr>
<tr>
<td>LAL_DAYSID_SI</td>
<td>86.164.090.53s</td>
<td>Mean sidereal day</td>
</tr>
<tr>
<td>LAL_LYB_SI</td>
<td>9.46052817 × 10^15 m</td>
<td>c x tropical year (1994)</td>
</tr>
</tbody>
</table>
The following cosmological parameters are derived from measurements of the Hubble expansion rate and of the cosmic background radiation (CBR). Data are taken from [1]. In what follows, the normalized Hubble constant \( h_0 \) is equal to the actual Hubble constant \( H_0 \) divided by \( \langle H \rangle = 100 \text{ km s}^{-1} \text{Mpc}^{-1} \). Thus the Hubble constant can be written as:

\[
H_0 = \langle H \rangle h_0.
\]

Similarly, the critical energy density \( \rho_c \) required for spatial flatness is given by:

\[
\rho_c = \langle \rho \rangle h_0^2.
\]

Current estimates give \( h_0 \) a value of around 0.65, which is what is assumed below. All values are in the SI units shown.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAL_H0_SI</td>
<td>( 2 \times 10^{-18} \text{s}^{-1} )</td>
<td>Approx. Hubble constant ( H_0 )</td>
</tr>
<tr>
<td>LAL_H0FAC_SI</td>
<td>( 3.2407792903 \times 10^{-18} \text{s}^{-1} )</td>
<td>( H_0/h_0 )</td>
</tr>
<tr>
<td>LAL_RHOC_SI</td>
<td>( 7 \times 10^{-10} \text{J m}^{-3} )</td>
<td>Approx. critical energy density ( \rho_c )</td>
</tr>
<tr>
<td>LAL_RHOCFAC_SI</td>
<td>( 1.68860 \times 10^{-9} \text{J m}^{-3} )</td>
<td>( \rho_c/h_0^2 )</td>
</tr>
<tr>
<td>LAL_TCBR_SI</td>
<td>2.726K</td>
<td>CBR temperature</td>
</tr>
<tr>
<td>LAL_VCBR_SI</td>
<td>( 3.695 \times 10^5 \text{m s}^{-1} )</td>
<td>Solar velocity with respect to CBR</td>
</tr>
<tr>
<td>LAL_RHOCBR_SI</td>
<td>( 4.177 \times 10^{-14} \text{J m}^{-3} )</td>
<td>Energy density of CBR</td>
</tr>
<tr>
<td>LAL_NCBR_SI</td>
<td>( 4.109 \times 10^8 \text{m}^{-3} )</td>
<td>Number density of CBR photons</td>
</tr>
<tr>
<td>LAL_SCBR_SI</td>
<td>( 3.993 \times 10^{-14} \text{J K}^{-1} \text{m}^{-3} )</td>
<td>Entropy density of CBR</td>
</tr>
</tbody>
</table>
3.7 Header LALStdio.h

Provides LAL functions similar to the non-file functions in `<stdio.h>`.

Synopsis

```c
#include <lal/LALStdio.h>
#include <lal/FileIO.h>
```

This header provides the LALsnprintf function.
3.7.1 Module LALStd.c

LAL replacement routines for standard C functions. At present, this just includes replacements for `snprintf`.

**Prototypes**

```c
int LALSnprintf( char *str, size_t size, const char *fmt, ... );
int LALVsnprintf( char *str, size_t size, const char *fmt, va_list ap );
```

**Description**

The routines `LALSnprintf()` and `LALVsnprintf()` are wrappers for `snprintf()` and `vsnprintf()`, if these functions are available, otherwise they are simply `sprintf()` or `vsprintf()`. It is strongly recommended that `LALSnprintf()` and `LALVsnprintf()` be used rather than `sprintf()` and `vsprintf()` as the latter are prone to buffer-overflow problems.
3.8 Header LALVersion.h

Provides routines for reporting the LAL version.

Synopsis

```
#include <lal/LALVersion.h>
```

This header covers the routines for reporting the LAL version.

Global variables

```
extern const char *lalVersion;
extern const int lalVersionMajor;
extern const int lalVersionMinor;
extern const char *lalConfigureArgs;
extern const char *lalConfigureDate;
extern const char *lalCVSTag;
```

These constant variables are set at compile time and included into the LAL library. They contain the information about the version of LAL and the configuration information.

Macros

```
#define LALVersionRequired( major, minor, micro )
   ( LAL_VERSION_MAJOR > ( major ) ||
   ( LAL_VERSION_MAJOR == ( major ) &&
   ( LAL_VERSION_MINOR > ( minor ) ||
   ( LAL_VERSION_MINOR == ( minor ) &&
   LAL_VERSION_MICRO >= ( micro )
   )
   )
   )
```

This macro returns 0 (false) if you do not have the require version of LAL, or 1 (true) if you do.

Error conditions

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>1</td>
<td>&quot;Null string pointer.&quot;</td>
</tr>
<tr>
<td>SIZE</td>
<td>2</td>
<td>&quot;Zero string size.&quot;</td>
</tr>
<tr>
<td>SPRN</td>
<td>4</td>
<td>&quot;Error in snprintf.&quot;</td>
</tr>
<tr>
<td>SHRT</td>
<td>8</td>
<td>&quot;String too short.&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants LALVERSIONH_E<name>, and the status descriptions in LALVERSIONH_MSGE<name>. The source code with these messages is in LALVersion.h on line 1.110.
Module **LALVersion.c**
Routine that returns the version of LAL.

**Prototypes**

```c
void LALVersion( LALStatus *status, CHAR *message, UINT4 size, INT4 verbose )
```

**Description**

This function writes a version message into the string buffer of specified size (and is truncated if the buffer is too small). Configuration information is also provided if the verbose flag is set.
3.8.1 Program LALVersionTest.c

Prints the version and configure options of the LAL library being used.

Usage

LALVersionTest

Description

This program prints the current version of LAL. If the version information in the library differs from the version information in the header file, this program prints the two versions and exits with code 1. This is useful for determining which version of the LAL library and header files you are linking to.

Exit codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success, normal exit.</td>
</tr>
<tr>
<td>1</td>
<td>Version info in library disagrees with header file.</td>
</tr>
<tr>
<td>2</td>
<td>Subroutine failed.</td>
</tr>
</tbody>
</table>
3.9 Header LALMalloc.h

Provides standard LAL memory allocation/deallocation routines.

Synopsis

```
#include <lal/LALMalloc.h>
```

This header covers routines that replace the standard `malloc()`, `calloc()`, `realloc()`, and `free()`. All memory allocation and deallocation in LAL should use these replacement functions. If the `NDEBUG` flag is set at compile time, the LAL routines are `#defined` to be the same as the standard C routines.
3.9.1 Module **LALMalloc.c**

LAL memory allocation routines.

**Macros**

```c
#if defined NDEBUG || defined LAL_NDEBUG
#define LALMalloc malloc
#define LALMallocShort malloc
#define LALMallocLong( n, file, line ) malloc( n )
#define LALCalloc calloc
#define LALCallocShort calloc
#define LALCallocLong( m, n, file, line ) calloc( m, n )
#define LALRealloc realloc
#define LALReallocShort realloc
#define LALReallocLong( p, n, file, line ) realloc( p, n )
#define LALFree free
#define LALCheckMemoryLeaks()
#else
#define LALMalloc( n ) LALMallocLong( n, __FILE__, __LINE__ )
#define LALCalloc( m, n ) LALCallocLong( m, n, __FILE__, __LINE__ )
#define LALRealloc( p, n ) LALReallocLong( p, n, __FILE__, __LINE__ )
#define LALFree ()
#endif
```

**Prototypes**

```c
1.743 LALMalloc.c
void * LALMallocShort( size_t n )

1.752 LALMalloc.c
void * LALMallocLong( size_t n, const char *file, int line )

1.783 LALMalloc.c
void * LALCallocShort( size_t m, size_t n )

1.782 LALMalloc.c
void * LALCallocLong( size_t m, size_t n, const char *file, int line )

1.825 LALMalloc.c
void * LALReallocShort( void *p, size_t n )

1.834 LALMalloc.c
void * LALReallocLong( void *q, size_t n, const char *file, const int line )

1.889 LALMalloc.c
void LALFree( void *q )

1.910 LALMalloc.c
void LALCheckMemoryLeaks( void )
```
Description

These functions are the LAL replacements for malloc(), calloc(), realloc(), and free(), with extra functionality to check for memory leaks (i.e. unfreed memory and segmentation violations). The LALMallocLong(), LALCallocLong(), and LALReallocLong() functions have two extra arguments giving the file name and line number of the calling statement; LALMallocShort(), LALCallocShort(), and LALReallocShort() do not have these extra arguments, and are merely call the corresponding long alloc functions with a file name of "unknown" and a line number of -1 (they are useful if you want to replace hooks to malloc(), calloc(), and realloc() of an external package that provides suitable hooks). LALMalloc(), LALCalloc(), and LALRealloc() are actually macros which call the functions LALMallocLong(), LALCallocLong(), and LALReallocLong with the appropriate file name and line number information. In practice, it is almost sufficient to use LALMalloc(), LALCalloc(), and LALRealloc() as you would malloc(), calloc(), and realloc().

Any time an object is freed, LALFree() checks to make sure that the memory bounds were not overwritten, and that the memory address is valid. The function LALCheckMemoryLeaks() is to be called at the end of a program when all the allocated memory is expected to have been freed. If there is memory that has been allocated but not freed then this routine reports an error. Whenever a memory leak is detected, the routines raise a segmentation violation signal SIGSEGV. (The signal is raised using the signal raising hook lalRaiseHook, which can be reset to a different handler if desired.)

Memory leak detection adds significant computational overhead to a program. It also requires the use of static memory, making the code non-thread-safe (but it can be made posix-thread-safe using the --enable-pthread-lock configure option). Production code should suppress memory leak detection at runtime by setting the global lalDebugLevel equal to zero or by setting the LALNMEMDBG bit of lalDebugLevel, or at compile time by compiling all modules with the NDEBUG flag set or by using the --disable-debug configure option. This causes LALCheckMemoryLeaks() to do nothing, and the other functions to revert to their standard C counterparts. In addition, you can turn off individual components of the memory debugging tools. Setting the LALNMEMPAD bit of lalDebugLevel prevents the allocation routines from "padding out" the arrays in an effort to detect buffer overflows. Setting the LALNMEMTRK bit of lalDebugLevel prevents tracking the allocations/frees. Setting the LALMEMINFO bit of lalDebugLevel produces copious output describing each memory allocation and deallocation.

Algorithm

When buffer overflow detection is active, LALMalloc() allocates, in addition to the requested memory, storage at the beginning of the object where a magic number and the size of the object is recorded, and padding at the end of the object. The number of allocations and the total size of allocated memory are stored in static memory. When LALFree() is executed, the padding at the end of the object and the magic number are examined to see if the bounds of the object were over-written. The total number of allocations and the total memory allocated are decreased. LALCheckMemoryLeaks() is called when all memory should have been freed. If the number of allocations or the total memory allocated is not zero, this routine reports an error.

When memory tracking is active, LALMalloc() keeps a linked list containing information about each allocation: the memory address, the size of the allocation, and the file name and line number of the calling statement. Subsequent calls to LALFree() make sure that the address to be freed was correctly allocated. In addition, in the case of a memory leak in which some memory that was allocated was not freed, LALCheckMemoryLeaks() prints a list of all allocations and the information about the allocations.

When any of these routines encounter an error, they will issue an error message using LALPrintError() and will raise a SIGSEGV signal, which will normally cause execution to terminate. The signal is raised using the hook lalRaiseHook, which can be set to perform a different action if desired.

These routines also issue status messages indicating how much memory is being allocated or freed with each function call. These memory information messages are considered a distinct class of status message, and can be activated or suppressed independently of other status messages. See the discussion in LALStatusMacros.h.

When lalDebugLevel is set to zero or the LALNMEMDBG bit is set, or when compiled with the NDEBUG flag set, these functions revert to their standard system versions, and LALCheckMemoryLeaks() does nothing.

Uses

lalDebugLevel
lalRaiseHook
LALPrintError()

Notes
Memory leak detection only occurs when lalDebugLevel ≠ 0. To turn on leak detection independent of error reporting, simply switch on the most-significant bit of lalDebugLevel, which is reserved not to be associated with any type of status message. See the discussion in LALStatusMacros.h for more information about lalDebugLevel.

It is assumed that pointers of type size_t * have the most restrictive alignment. If this is not true, then this code may not work except in non-debugging mode. (It will probably produce bus errors.)

Debugging memory leak tips
Programs should end by calling LALCheckMemoryLeaks(). This will ensure that all memory that has been allocated has been freed. Making sure that all memory allocated is freed is a good idea in order to make sure (i) that memory isn’t being “lost” (which may mean that the computer will run out of memory when the program is run under more extensive use), (ii) that array bounds are not being exceeded (since this will usually overwrite the pad area at the end of the array, and this overwrite is detected when the array is freed). LALCheckMemoryLeaks() should pass silently—if it doesn’t, then there is probably some memory that has not been freed; LALCheckMemoryLeaks() will give information about where this memory was allocated.

The most common problem (after forgetting to free some memory) is overwriting of array bounds. When this is detected, LALFree() reports the memory address that was overwritten, as well as the address of the array that LALFree() attempted to free. In order to find out where the overwrite occurs, run the program in the debugger and stop the execution as soon as the array that is being overwritten has been allocated. The LALMalloc module has some secret memory debugging tools (for use in debugging only!). One is the global variable lalMemDbgUsrPtr, which is of type char *. Set this variable to be equal to the memory address where the overwrite occurs. Then watch the contents of the variable to find out where the overwrite occurs. This is done in gdb using the commands:

```
set var lalMemDbgUsrPtr=0x20f530
watch *lalMemDbgUsrPtr
cont
```

where 0x20f530 is the corrupted memory address. The program will run until the value of this address is changed, thereby allowing you to find out where in the program the overwrite occurs.

If you don’t know where the memory was allocated, you can locate this too. To do so, set lalMemDbgUsrPtr to be the address of the array. Then, every time LALMalloc() is called, it sets the value of the global variable lalIsMemDbgRetPtr to be one zero if the array address produced by LALMalloc() is not the address in lalMemDbgUsrPtr, and one if it is. Then you can watch the value of lalIsMemDbgRetPtr in a debugger until it changes to one, which stops execution at that point. (Note: it is possible that a given address is allocated, then freed, the allocated again—you may need to watch lalIsMemDbgRetPtr for a while.)

Here’s an example debugging session: first we run the program, identify the address of the array whose bounds are being overwritten, and find out where that array is allocated.

```
(gdb) run
LALFree error: array bounds overwritten
Byte 4 past end of array has changed
Corrupted address: 0x1cf530
Array address: 0x1cf528
```

Program received signal SIGSEGV, Segmentation fault.
0x9001b46c in kill ()
(gdb) list 1,11
1     #include <lal/LALStdlib.h>
2     int main( void )
3     {
4         char *s;
5         lalDebugLevel = 1;
6         s = LALMalloc( 5 );
```
7 s[8] = 'x';
8 LALFree( s );
9 LALCheckMemoryLeaks();
10 return 0;
11 }
(gdb) break 5
Breakpoint 1 at 0x1b60: file bad.c, line 5.
(gdb) run

Breakpoint 1, main () at bad.c:5
5 lalDebugLevel = 1;
(gdb) set var lalMemDbgUsrPtr = 0x1cf528
(gdb) watch lalIsMemDbgRetPtr
Hardware watchpoint 2: lalIsMemDbgRetPtr
(gdb) cont
Continuing.
Hardware watchpoint 2: lalIsMemDbgRetPtr

Old value = 0
New value = 1
0x0088d63c in LALMallocLong (n=5, file=0x1ff8 "bad.c", line=6) at LALMalloc.c:575
575 lalIsMemDbgPtr = lalIsMemDbgRetPtr = ( lalMemDbgRetPtr == lalMemDbgUsrPtr );
(gdb) up
#1 0x00001b84 in main () at bad.c:6
6 s = LALMalloc( 5 );

So here is where the memory is allocated. We want to find out where the memory is being corrupted.

(gdb) set var lalMemDbgUsrPtr = 0x1cf530
(gdb) watch *lalMemDbgUsrPtr
Hardware watchpoint 3: *lalMemDbgUsrPtr
(gdb) cont
Continuing.
Hardware watchpoint 3: *lalMemDbgUsrPtr

Old value = -25
New value = 120 'x'
main () at bad.c:8
8 LALFree( s );
(gdb) list
3 {
4       char *s;
5       lalDebugLevel = 1;
6       s = LALMalloc( 5 );
7       s[8] = 'x';
8       LALFree( s );
9       LALCheckMemoryLeaks();
10     return 0;
11 }

Notice that the program has stopped just after the line in which the array bounds were overwritten.
3.9.2 Program LALMallocTest.c

Tests the routines in LALMalloc.h.

Usage

LALMallocTest

Description

This program has ugly code for testing the LAL memory allocation and freeing routines.

Exit codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success.</td>
</tr>
<tr>
<td>1</td>
<td>Failure.</td>
</tr>
</tbody>
</table>

Uses

lalDebugLevel
LALMalloc()
LALCalloc()
LALRealloc()
LALFree()
LALCheckMemoryLeaks()

Notes
3.10 Header LALError.h

Provides routines to report and handle errors.

Synopsis

```c
#include <lal/LALError.h>
```

This header covers routines that print status messages, and that allow functions to abort.
Error handling routines for LAL. These should not be invoked in production code, except in very specific circumstances.

Prototypes

```c
int LALPrintError( const char *fmt, ... )
int (*lalRaiseHook)( int, const char *, ... ) = LALRaise;
int LALRaise( int sig, const char *fmt, ... )
void (*lalAbortHook)( const char *, ... ) = LALAbort;
void LALAbort( const char *fmt, ... )
int LALError( LALStatus *status, const char *statement )
int LALWarning( LALStatus *status, const char *warning )
int LALInfo( LALStatus *status, const char *info )
int LALTrace( LALStatus *status, int exitflg )
```

Description

These functions cause LAL to print status messages and perform basic error handling. Their implementation is quite simple but may be altered in the future to provide reasonable behaviour when integrated with other systems (e.g., LDAS). As a general rule, LALWarning() and LALInfo() are the only routines that programmers should use in their own modules; the other routines are used internally by LAL. Descriptions of the individual functions are as follows.

LALPrintError() prints a formatted string to some designated output device (usually the stderr stream), returning the number of characters printed, or negative if an error occurred. The format of the argument list is the same as for the standard C routine printf(). By funneling all LAL error printing through this one routine, it is easier to adapt LAL to implementations that have particular I/O or error-logging requirements. Most LAL routines should use LALError(), LALWarning(), and LALInfo() to report their status, rather than calling LALPrintError() directly.

LALRaise() prints a formatted string to an error logging device, as above, and then raises the requested signal. Standard LAL routines should not terminate execution, but should instead return control to the calling routine, reporting errors through their LALStatus structure. Programmers should never invoke LALRaise() explicitly. A hook to a LALRaise()-type function, lalRaiseHook, is provided, should the user wish to change the default behavior of LALRaise() (i.e., the LAL library always uses lalRaiseHook rather than LALRaise, but lalRaiseHook is set to LALRaise by default).

LALAbort() prints a formatted string to an error logging device, as above, and then terminates program execution. Usually this is done by raising a SIGABRT signal, but this can change in implementations that have different requirements. Standard LAL routines should not terminate execution, but should instead return control to the calling routine, reporting errors through their LALStatus structure. The exception is when a function receives a NULL status pointer, in which case it has no option but to abort. This is done automatically by the INITSTATUS() macro (see LALStatusMacros.h), so programmers should never need to invoke LALAbort() explicitly. A hook to a LALAbort()-type function, lalAbortHook, is provided, should the user wish to change the default behavior of LALAbort() (i.e., the LAL library always uses lalAbortHook rather than LALAbort, but lalAbortHook is set to LALAbort by default).
LALError() prints the statement string to the error log, provided that the value of the global lalDebugLevel is set to allow error messages. It returns the number of characters printed. This is the standard LAL routine for printing error messages. However, LALError() is called automatically by the status-handling macros (see LALStatusMacros.h) whenever a LAL function returns with non-zero error code. Since an error is, by definition, a condition that would cause a routine to terminate abnormally, LAL programmers will generally not have to call LALError() explicitly.

LALWarning() prints the warning string to the error log, provided that the value of the global lalDebugLevel is set to allow warning messages. It returns the number of characters printed. A warning message is less serious than an error message: it indicates that computation is proceeding successfully, but with unusual or unexpected behaviour that may invalidate the results of the computation.

LALInfo() prints the info string to the error log, provided that the value of the global lalDebugLevel is set to allow information messages. It returns the number of characters printed. An information message indicates that a computation is proceeding normally, and simply provides additional information about its progress.

LALTrace() prints a message providing information, taken from the status structure, about the function currently being executed; it is used to track the progress of execution through nested function calls. It returns the number of characters printed. The message begins with the word Enter (if exitflg = 0) or Leave (if exitflg ≠ 0), to indicate whether the flow of execution has just entered or is about to leave the function. Tracking information is printed only if the value of the global lalDebugLevel is set to allow it. LALTrace() is called automatically by the status macros when entering or leaving a function (see LALStatusMacros.h), so LAL programmers need never invoke it explicitly.

Algorithm

The functions LALError(), LALWarning(), LALInfo(), and LALTrace() print status messages depending on the value of the global lalDebugLevel. Specifically, each type of status message is associated with a particular bit in lalDebugLevel. If the value of the bit is 1, that type status message will be printed; if it is 0, that type of message will be suppressed. See the documentation in LALStatusMacros.h for information about how to set the value of lalDebugLevel.

These four functions are also suppressed if a module is compiled with the NDEBUG flag set. In this case, however, the function calls are actually removed from the object code (i.e. they are replaced with the integer 0, representing their return value). This is used to generate streamlined production code. Again, see the LALStatusMacros.h documentation for more discussion of this compilation flag.

Macro replacement functions

When a LAL module is compiled with the flag NOLALMACROS set, the usual status-handling macros defined in LALStatusMacros.h are replaced with function calls to specialized support functions that perform the same operations. These functions are necessarily global in scope, and so we provide their prototype declarations below. However, they will never be invoked explicitly in any LAL function, so we will not bother with additional usage information.

```
int LALInitStatus( LALStatus *status, const char *function, const char *id, const char *file, const int line )
int LALPrepareReturn( LALStatus *status, const char *file, const int line )
int LALAttachStatusPtr( LALStatus *status, const char *file, const int line )
int LALDetachStatusPtr( LALStatus *status, const char *file, const int line )
int LALPrepareAbort( LALStatus *status, const INT4 code, const char *mesg, const char *file, const int line )
```
int
LALPrepareAssertFail( LALStatus *status, const INT4 code, const char *msg,
const char *statement, const char *file, const int line )

int
LALCheckStatusPtr( LALStatus *status, const char *statement, const char *file,
const int line )

void
FREESTATUSPTR( LALStatus *status )

void
REPORTSTATUS( LALStatus *status )

Uses
lalDebugLevel

Notes
3.11 Header LALGSL.h

Provides macros for integrating the GSL error handler with the LAL status structure.

Synopsis

#include <lal/LALGSL.h>

This header provides macros and functions for tracking and reporting the runtime status of a GSL calls. The intent is simultaneously to standardize the error reporting, and to make the reporting as transparent as possible to people coding individual routines.

Please always use these macros when making a GSL call within LAL. This will ensure that the LAL functions always have the same behaviour and will also ensure that the LAL functions are reenterant and threadsafe (when LAL is configured appropriately).

GSL function calls

The behaviour of GSL functions depends on the error handler that has been assigned. In order that LAL functions always have the same behaviour, it is necessary to use a LAL-specific GSL error handler. This error handler populates a LAL status structure with the GSL error message and code so that GSL functions behave much the same way as LAL functions. After the GSL functions are called, the error handler needs to be restored to the original handler so that the program calling the LAL routine has the same error handler after the GSL function was called as it did before the LAL function was called.

This module provides a simple set of macros and the default LAL GSL error handler. The macros provide a standard way to assign the LAL GSL error handler before a GSL function call and to restore the original handler after the call.

Note that changing the GSL error handler is not a thread-safe action. Therefore it is necessary to block other threads from performing GSL function calls while one thread has changed the handler. These macros ensure that such blocking is done for GSL function calls within other LAL routines if LAL is configured with the --enable-pthread-lock flag. See below for instructions on how to make other GSL function calls outside LAL thread-safe when used with LAL.

ATTATCHSTATUSPTR( status );
CALLGSL( gsl_function( x ), status );
CHECKSTATUSPTR( status );
DETATCHSTATUSPTR( status );

Note that the LAL function must attach (and detach) a status pointer as if a LAL routine were called. Note also that you need to use the CHECKSTATUSPTR macro to check the status of the call. The equivalent to the TRY macro for GSL functions is the TRYGSL macro, which is used as follows:

ATTATCHSTATUSPTR( status );
TRYGSL( gsl_function( x ), status );
DETATCHSTATUSPTR( status );

If you are using GSL functions both in LAL and in the calling program, and you are worried about thread-safety, the GSL function calls outside of LAL need to be blocked so that they do not access the GSL error handler while it has been changed to the LAL GSL error handler in a LAL function. To do this, you need to do the following:

#include<lal/LALGSL.h>
...
LALGSL_PTHREAD_MUTEX_LOCK;
gsl_function( x );
LALGSL_PTHREAD_MUTEX_UNLOCK;

This ensures that gsl_function is not called while a LAL routine is calling a GSL function in a different thread. You can do this even if you don’t always run your code with multiple threads. If you configure LAL without the --enable-pthread-lock flag, the macros LALGSL_PTHREAD_MUTEX_LOCK and LALGSL_PTHREAD_MUTEX_UNLOCK do nothing.
3.11.1 Module LALGSL.c

LAL GSL error handler.

Prototypes

extern LALStatus *lalGSLGlobalStatusPtr;
#include <lal/LALConfig.h>
#ifdef LAL_PTHREAD_LOCK
#include <pthread.h>
extern pthread_mutex_t lalGSLPthreadMutex;
#endif

void
LALGSLErrorHandler(
    const char *reason,
    const char *file,      
    int line,              
    int my_gsl_error
)

Description

The function LALGSLErrorHandler is the standard GSL error handler for GSL functions called within LAL. Its function is to take the GSL error code and information and translate them into equivalent aspects of the LAL status structure. The status structure that is currently identified by lalGSLGlobalStatusPtr is populated. This global variable is to be set to the current status pointer prior to invocation of the GSL function. In addition, the GSL error handler must be set to the LAL GSL error handler prior to the invocation of the GSL function. Both of these tasks can be done with the macros provided in the header LALGSL.h [3.11] However, doing so is not thread-safe. Thus the macros use the mutex lalGSLPthreadMutex to block other threads from making GSL calls from within LAL functions while one thread has set the GSL error handler and global status pointer. This mutex must also be used to block any non-LAL GSL function calls from other threads or else they may be called with the LAL GSL error handler in effect.
3.11.2 Program LALGSLTest.c

This program tests the LAL macros for GSL function calls. It makes sure that a nominal status is returned if the GSL function succeeds, and that an error code is returned if the GSL function fails.
3.12 Header StringInput.h

Provides routines to parse CHARVectors into other LAL datatypes.

Synopsis

```c
#include "StringInput.h"
```

This header provides prototypes for routines that construct LAL data structures using the data from a character string. As in standard C, a string is a block of non-null bytes of arbitrary length, terminated by a null byte '\0', and referred to by a value of type `CHAR *` pointing to the first byte in the string. It is not to be confused with a CHARVector, a LAL structure referring to a block of data of a specified length, which may or may not contain one or more instances of '\0'.

In general, the routines under this header will have string inputs of type `const CHAR *` (in order to allow, for instance, string literals to be used as inputs), but will allocate CHARVector structures to store string outputs. Unless otherwise specified, these outputs are guaranteed to contain at least one '\0' character, so their data fields are valid strings. It is the responsibility of the calling routine to ensure that the string input contains a terminating '\0' within the memory segment pointed to by the CHAR * input, in order to avoid segmentation violation.

These routines are intended to work in conjunction with the functions in StreamInput.h to add LAL robustness to otherwise ad-hoc data input routines. However, the functions in StringInput.h are fully LAL-compliant and use only LAL types, so they are included in liblal proper.

Error conditions

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUL</td>
<td>1</td>
<td>&quot;Unexpected null pointer in arguments&quot;</td>
</tr>
<tr>
<td>OUT</td>
<td>2</td>
<td>&quot;Output handle points to a non-null pointer&quot;</td>
</tr>
<tr>
<td>MEM</td>
<td>3</td>
<td>&quot;Memory allocation error&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants STRINGINPUTH_E<name>, and the status descriptions in STRINGINPUTH_MSGE<name>. The source code with these messages is in StringInput.h on line 1.89.

Constants

The following constants are format strings that can be used by the various C `scanf()` or `printf()` functions to parse or write sequences of characters corresponding to base LAL datatypes. Since the C datatypes (`short`, `int`, `long`, `long long`, `float`, `double`, etc.) do not have fixed mappings to LAL base datatypes (`INT2`, `INT4`, `INT8`, `REAL4`, `REAL8`, etc.), the appropriate format strings for each LAL datatype must be determined at configuration time and set at compile time.

These format strings give only the conversion character preceded by any length modifier according to the type (`short`, `long`, etc.). In particular they do not contain the initial '%' character that initiates the conversion specification. However, being `#defined` string literals, they can be combined with "%" string literals or more complicated format strings through implicit concatenation. Thus to scan string for a UINT4 number n one would write:

```c
sscanf( string, "%" LAL_UINT4_FORMAT, &n );
```

Similarly, to print a REAL8 number x with 12 digits following the decimal place, one could use the following:

```c
printf( "%12.12" LAL_REAL8_FORMAT, x );
```

Of course, floating-point numbers are more commonly printed using the "%e" conversion specifier, which does not generally require type-dependent length modifiers.
### Types

**Structure TokenList**

This structure stores a number of null-terminated strings of arbitrary length. The entire list is stored flattened in a CHARVector, and individual tokens are pointed to by a CHAR *[] handle. The fields are:

**UINT4 nTokens** The number of tokens in the list.

**CHAR **tokens** A list of pointers to the individual tokens. The elements tokens[0..nTokens-1] point to tokens, and the element tokens[nTokens] is explicitly NULL (as is the convention for an argv argument list).

**CHARVector **list** The flattened list of tokens, separated by (and terminated with) ‘\0’ characters.

<table>
<thead>
<tr>
<th>Name</th>
<th>Usual value</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAL_INT2_FORMAT</td>
<td>&quot;hd&quot;</td>
</tr>
<tr>
<td>LAL_INT4_FORMAT</td>
<td>&quot;d&quot; or &quot;ld&quot;</td>
</tr>
<tr>
<td>LAL_INT8_FORMAT</td>
<td>&quot;ld&quot; or &quot;lld&quot;</td>
</tr>
<tr>
<td>LAL_UINT2_FORMAT</td>
<td>&quot;hu&quot;</td>
</tr>
<tr>
<td>LAL_UINT4_FORMAT</td>
<td>&quot;u&quot; or &quot;lu&quot;</td>
</tr>
<tr>
<td>LAL_UINT8_FORMAT</td>
<td>&quot;lu&quot; or &quot;llu&quot;</td>
</tr>
<tr>
<td>LAL_REAL4_FORMAT</td>
<td>&quot;f&quot;</td>
</tr>
<tr>
<td>LAL_REAL8_FORMAT</td>
<td>&quot;lf&quot;</td>
</tr>
</tbody>
</table>
3.12.1 Module **StringToken.c**

Converts a string into a series of tokens, for use by other routines.

**Prototypes**

```c

void LALCreateTokenList( LALStatus *stat,
                        TokenList **list,
                        const CHAR *string,
                        const CHAR *delimiters )

void LALDestroyTokenList( LALStatus *stat,
                          TokenList **list )
```

**Description**

The routine `LALCreateTokenList()` parses `*string` as a sequence of tokens (substrings of non-null characters that do not appear in `delimiters`), separated by delimiters (substrings consisting only of characters that appear in `delimiters`), and terminated by the null character `\0`. The structure `**list` is created, storing the sequence of tokens as a list null-terminated character strings.

The output `list` should be a non-NULL handle that points to the value `NULL` (i.e. `list!=NULL` but `*list=NULL`). Even if no tokens were found, `*list` will be created, but will have `(*list)->nTokens=0, (*list)->tokens[0]=NULL, and (*list)->list=NULL`. Note that this is not an error, so the calling routine need not guarantee in advance that `string` contain any non-delimiter characters.

The routine `LALDestroyTokenList()` destroys a list of tokens as created by `LALCreateTokenList()`, setting `*list` to `NULL`.

**Algorithm**

The `LALCreateTokenList()` function is not particularly memory-efficient, requiring internal storage up to twice the length of `*string`. It first creates a working copy of `string->data`, and replaces all occurrences of characters appearing in `*delimiters` with `\0`, while at the same time keeping track of the number and total length of all tokens. It then allocates a contiguous block of memory to store all the tokens (separated by and terminated with single `\0` characters), and a set of `CHAR *` pointers to point to the individual tokens in this block. Then the routine proceeds through the working copy one last time, copying tokens into the token list and setting the token pointers accordingly, before destroying the working copy.

**Uses**

```c

LALMalloc() LALCHARCreateVector()
LALFree() LALCHARDestroyVector()
```

**Notes**

Author: Creighton, T. D.

$Id: StringToken.c,v 1.2 2007/06/08 14:41:53 bema Exp $
3.12.2 Module StringConvert.c

Converts a string into a numerical value.

Prototypes

```c
void LALStringToU2( LALStatus *stat, UINT2 *value, const CHAR *string, CHAR **endptr )
void LALStringToU4( LALStatus *stat, UINT4 *value, const CHAR *string, CHAR **endptr )
void LALStringToU8( LALStatus *stat, UINT8 *value, const CHAR *string, CHAR **endptr )
void LALStringToI2( LALStatus *stat, INT2 *value, const CHAR *string, CHAR **endptr )
void LALStringToI4( LALStatus *stat, INT4 *value, const CHAR *string, CHAR **endptr )
void LALStringToI8( LALStatus *stat, INT8 *value, const CHAR *string, CHAR **endptr )
void LALStringToS( LALStatus *stat, REAL4 *value, const CHAR *string, CHAR **endptr )
void LALStringToD( LALStatus *stat, REAL8 *value, const CHAR *string, CHAR **endptr )
void LALStringToC( LALStatus *stat, COMPLEX8 *value, const CHAR *string, CHAR **endptr )
void LALStringToZ( LALStatus *stat, COMPLEX16 *value, const CHAR *string, CHAR **endptr )
void LALStringToGPS( LALStatus *stat, LIGOTimeGPS *value, const CHAR *string, CHAR **endptr )
```

Description

These routines parse the string `*string` and compute a numerical value `*value` of the appropriate datatype. If `endptr` is not `NULL`, then after conversion `*endptr` will point to the character after the last character used in the conversion. The routine will always return without error if the arguments are valid, regardless of the contents of `string`; failure to parse a number is indicated by `*endptr` being set equal to `string` (provided `endptr` is not `NULL`).

For integer or floating-point conversion to occur, `string` must consist of zero or more whitespace characters followed by a number in any standard base-ten integer or floating-point representation (described in detail below); the conversion will stop as soon as the routine encounters a character that is not part of the number. For instance, parsing the string "123bad" will return `*value`=123, and `*endptr` will point to the substring "bad"; it is up to the calling routine to determine whether this is an acceptable input. By contrast, parsing the string "bad" will leave `*value` unchanged and `*endptr` pointing to the start of the original string. In general, if the routine returns with `*endptr` not equal to `string` and `**endptr` is a whitespace or `'\0'` character, then the format was unambiguously acceptable.

Complex conversion is essentially equivalent to performing floating-point conversion on `string` to get the real part, and again on `*endptr` to get the imaginary part. Normally this means that an acceptable format is two floating-point representations separated by (and possibly preceded with) whitespace, although the intervening whitespace can be omitted if the separation between the two numbers is unambiguous. Thus the string "-1.0+3.5" will be unambiguously read as $-1.0 + 3.5i$, but "-1.03.5" will be read as $-1.03 + 0.5i$ (since the conversion of the real part stops at the second `.' character), which may or may not be the intended conversion.

GPS conversion is similar to floating-point conversion, but the result is stored in a `LIGOTimeGPS` structure as two integer values representing seconds and nanoseconds. The `LALStringToGPS` function does not convert the string to an intermediate `REALS` value, but parses the string specially to retain the full precision of the string representation to the nearest nanosecond.
Algorithm

These functions (other than LALStringToGPS) emulate the standard C functions `strtol()`, `strtoul()`, and `strtod()`, except that they follow LAL calling conventions and return values of the appropriate LAL datatypes. For integer conversion, only base-ten (decimal) representations are supported. Otherwise, the valid format is as for the corresponding C functions, which we summarize below:

A string to be converted to an `INTn` (where \( n = 2, 4, \) or \( 8 \)) consists of zero or more whitespace characters as determined by `isspace()`, followed optionally by a single `+'` or `-'` character, followed by one or more decimal digits; conversion stops at the first non-digit character after this. If the result would overflow or underflow the `INTn` representation, then the value is set to \( LAL_{\text{INT}}n_{\text{MAX}} = 2^{8n-1} - 1 \) or \( LAL_{\text{INT}}n_{\text{MIN}} = -2^{8n-1} \), respectively.

A string to be converted to a `UINTn` follows the same format, except that a leading negative sign character `-'` is ignored (the routine will compute the magnitude of the result), and the return value is capped at \( LAL_{\text{UINT}}n_{\text{MAX}} = 2^{8n-1} - 1 \).

A string to be converted to a floating-point number (\( \text{REAL4} \) or \( \text{REAL8} \)) consists of zero or more whitespace characters as determined by `isspace()`, followed optionally by a single `+'` or `-'` character, followed by a sequence of one or more decimal digits optionally containing a decimal point `.`, optionally followed by an exponent. An exponent consists of a single `E` or `e` character, followed optionally by a single `+'` or `-'` character, followed by a sequence of one or more decimal digits. If the converted value would overflow, \( \pm LAL_{\text{REAL}}n_{\text{MAX}} \) is returned, as appropriate. If the value would underflow, 0 is returned.

A string to be converted to a complex number (\( \text{COMPLEX8} \) or \( \text{COMPLEX16} \)) consists of two floating-point format substrings concatenated together, where the first character of the second substring cannot be interpreted as a continuation of the first number. Usually this means that the second substring will contain at least one leading whitespace character, though this is not strictly necessary. Overflow or underflow is dealt with as above.

A string to be converted to a GPS time can have the format of an integer or a floating-point number, as described above. The optional exponent in the floating-point form is supported, and both positive and negative GPS times are permitted. If the result would overflow the LIGOTimeGPS representation (too far in the future), then the `gpsSeconds` and `gpsNanoSeconds` fields are set to `LAL_INT4_MAX` and `999999999`, respectively. For an underflow (too far in the past), the fields are set to `LAL_INT4_MIN` and 0.

Internally, the floating-point conversion routines call `strtod()`, then cap and cast the result as necessary. The complex conversion routines simply call their floating-point counterpart twice. The integer routines call an internal function `LALStringToU8AndSign()`, which does what you would expect, then cap and cast the result as necessary. (The C routines `strtol()` and `strtoul()` are not used as they are not guaranteed to have 8-byte precision.)

Uses

Notes
3.13 Header Grid.h

Provides a structured datatype for a multidimensional rectilinear grid.

Synopsis

#include <lal/Grid.h>

This header provides a new structured datatype storing data on a multidimensional rectilinear grid. It is in some sense a generalization of the series datatypes (frequency series, time series, etc.), representing evenly-sampled data over some physical parameter space.

Error conditions

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUL</td>
<td>1</td>
<td>&quot;Unexpected null pointer in arguments&quot;</td>
</tr>
<tr>
<td>OUT</td>
<td>2</td>
<td>&quot;Output handle points to a non-null pointer&quot;</td>
</tr>
<tr>
<td>MEM</td>
<td>3</td>
<td>&quot;Memory allocation error&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants GRIDH_E<name>, and the status descriptions in GRIDH_MSGE<name>. The source code with these messages is in Grid.h on line 1.67.

Types

Structure <datatype>Grid

This structure is a generalization of the LAL series types, storing data on an m-dimensional rectangular grid on a physical parameter space. The values on the grid are of type <datatype> which can be any LAL primitive numerical datatype (INT2, INT4, INT8, UINT2, UINT4, UINT8, REAL4, REAL8, COMPLEX8, COMPLEX16). The data are stored in an array of dimension $M \geq m$: if $M = m$, then the structure stores a single value for each grid point; if $M > m$, then the structure stores a vector or array of values on the “tangent space” of each grid point. We refer to $m$ as the grid dimension and $M$ as the data dimension. The fields of the structure are:

- CHAR name[LALNameLength] A name identifying the grid and/or the data being sampled.
- LALUnit sampleUnits The physical units of the quantities on the grid.
- LALUnit *dimUnits The physical units of the grid axes. This must be allocated as an array of length $m$.
- REAL8Vector *offset A vector $p_0$ specifying the location of the grid point indexed by $(0,\ldots,0)$. Must have dimensionality $m$.
- REAL8Vector *interval The vector $\Delta p$ specifying the grid spacing in each dimension. Must have dimensionality $m$.
- <datatype>Array *data Pointer to an array storing the data values at the corresponding grid points. The data dimension $M = \text{data->dimLength->length}$ must be greater than or equal to the grid dimension $m = \text{offset->length=interval->length}$. An index $i = (i_0,\ldots,i_{M-1})$, where $i_k$ are integers from 0 to $\text{data->dimLength->data}_k$, specifies a grid point located at $p = p_0 + \sum_{k=0}^{n-1} e_k i_k \Delta p_k$ if $M = m$, or an array element $A_{i_{m-\cdots-1}}$ at that grid point if $M > m$. The values in $\text{data->data}$ are the value stored at each grid point (or array element at each grid point), arranged in the manner discussed in LALDatatypes.h.
3.13.1 Module Grid.c

Creates or destroys a LAL grid structure.

Prototypes

```c
void LAL<typecode>CreateGrid( LALStatus *stat,
                               <datatype>Grid **grid,
                               UINT4Vector *dimLength,
                               UINT4 dimension )

void LAL<typecode>DestroyGrid( LALStatus *stat,
                               <datatype>Grid **grid )
```

Description

These routines create or destroy a `<datatype>Grid` structure. The input vector `dimLength` stores the lengths of each dimension of the grid and of the array at each grid point: in the notation defined in `Grid.h`, `dimLength->length = M`. The parameter `dimension` gives the dimension `m` of the physical grid space; if `M > m`, then the remaining dimensions refer to a tangent space at each grid point. When creating a grid, the routines allocate space for all the internal vectors and arrays, but no data are filled in, with the exception of the `(*grid)->data->dimLength` vector (which will contain exactly the same data as the `dimLength` input parameter). When calling the `LAL<typecode>CreateGrid()` routines, or on returning from the `LAL<typecode>DestroyGrid()` routines, `grid` should be a non-NULL handle to a NULL-valued pointer.

For each of these prototype templates there are in fact 10 separate routines corresponding to all the numerical atomic datatypes `<datatype>` referred to by `<typecode>`:

<table>
<thead>
<tr>
<th><code>&lt;typecode&gt;</code></th>
<th><code>&lt;datatype&gt;</code></th>
<th><code>&lt;typecode&gt;</code></th>
<th><code>&lt;datatype&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>I2</td>
<td>INT2</td>
<td>U2</td>
<td>UINT2</td>
</tr>
<tr>
<td>I4</td>
<td>INT4</td>
<td>U4</td>
<td>UINT4</td>
</tr>
<tr>
<td>I8</td>
<td>INT8</td>
<td>U8</td>
<td>UINT8</td>
</tr>
<tr>
<td>S</td>
<td>REAL4</td>
<td>C</td>
<td>COMPLEX8</td>
</tr>
<tr>
<td>D</td>
<td>REAL8</td>
<td>Z</td>
<td>COMPLEX16</td>
</tr>
</tbody>
</table>

Algorithm

Uses

- `lalDebugLevel`
- `LALMalloc()`
- `LALFree()`
- `LALDCreateVector()`
- `LALDDestroyVector()`
- `LAL<typecode>CreateArray()`
- `LAL<typecode>DestroyArray()`

Notes
References


Chapter 4

Package *sample*

This package contains templates for LAL headers and modules, as well as a fully-autodocumenting example program based on the primer in the *std* package.
4.1 Header LALSample.h

Example header for LAL.

Synopsis

```c
#include <lal/LALSample.h>
```

This header provides two trivial functions to divide real numbers. It exists primarily to demonstrate documentation and coding standards for LAL headers.

Error codes

<table>
<thead>
<tr>
<th>name</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>1</td>
<td>&quot;Arguments contained an unexpected null pointer&quot;</td>
</tr>
<tr>
<td>DIV0</td>
<td>2</td>
<td>&quot;Division by zero&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants LALSAMPLEH_E<name>, and the status descriptions in LALSAMPLEH_MSGE<name>. The source code with these messages is in LALSample.h on line 1.61.
4.1.1 Module LALSample.c

Example module for LAL.

Prototypes

```c
void LALREAL8Invert( LALStatus *stat, REAL8 *output, REAL8 input )
void LALREAL8Divide( LALStatus *stat, REAL8 *output, REAL8 numer, REAL8 denom )
```

Description

This module exists to demonstrate documentation and coding standards for LAL modules, using two trivial functions. `LALREAL8Invert()` computes `*output = 1/input`; if `input = 0`, it leaves `*output` unchanged and returns an error. `LALREAL8Divide()` computes `*output = numer/denom`, calling `LALREAL8Invert()` as a subroutine. This allows us to demonstrate LAL error handling through nested function calls.

Algorithm

Uses

Notes
4.1.2 Program LALSampTest.c

Example program for LAL.

Usage

LALSampTest [numer denom [lalDebugLevel]]

Description

This program demonstrates LAL coding and documentation standards for test programs. It reads two numbers numer and denom from the command line, computes their quotient using the function LALREAL8Divide(), and prints the result to stdout. It is run with lalDebugLevel = 0, unless set by the optional third argument.

Exit codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOM</td>
<td>0</td>
<td>&quot;Nominal exit&quot;</td>
</tr>
<tr>
<td>ARG</td>
<td>1</td>
<td>&quot;Error parsing command-line arguments&quot;</td>
</tr>
<tr>
<td>SUB</td>
<td>2</td>
<td>&quot;Subroutine returned error&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants LALSAMPLETESTC_E<name>, and the status descriptions in LALSAMPLETESTC_MSGE<name>. The source code with these messages is in LALSampTest.c on line 1.73.

Uses

lalDebugLevel
LALPrintError()
LALREAL8Divide()

Notes
Section 4

General Packages
Chapter 5

Package date

This package provides routines related to date and time manipulations (Date.h). It also provides routines to compute the difference in time of arrival of a signal at two detector locations, and also between a detector and the center of the Earth-fixed frame (TimeDelay.h).
5.1 Header Date.h

Provides routines for manipulating date and time information.

Synopsis

```
#include <lal/Date.h>
```

This header covers routines for manipulating date and time information. The various time systems are discussed in [4].

Error conditions

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULLINPUT</td>
<td>1</td>
<td>&quot;Input is NULL&quot;</td>
</tr>
<tr>
<td>NULLOUTPUT</td>
<td>2</td>
<td>&quot;Output is NULL&quot;</td>
</tr>
<tr>
<td>DATETOOEARY</td>
<td>3</td>
<td>&quot;Date too early: Julian Day can only be computed for dates &gt;= 1900-03-01&quot;</td>
</tr>
<tr>
<td>RANGEGPSABS</td>
<td>4</td>
<td>&quot;Input time out of range: only able to accurately convert times between 1980-Jan-06 00:00:00 UTC (GPS 0) and 2006-Jun-30 23:59:59 UTC (GPS 835747212)&quot;</td>
</tr>
<tr>
<td>BUFTTOOOSMALL</td>
<td>5</td>
<td>&quot;Output timestamp string too small: min. size = 26&quot;</td>
</tr>
<tr>
<td>ASCTIMEFAIL</td>
<td>6</td>
<td>&quot;asctimeUNDERSCOREr() failed&quot;</td>
</tr>
<tr>
<td>GPSDATETOOEARY</td>
<td>7</td>
<td>&quot;Date too early: GPS time only defined for times on or after 1980-Jan-06 00:00:00 UTC&quot;</td>
</tr>
<tr>
<td>FORMATPARAMOUTOFRANGE</td>
<td>8</td>
<td>&quot;Format parameter out of range: must be one of LALLEAPSECunderscoreTAIUTC or LALLEAPSECunderscoreGPSUTC&quot;</td>
</tr>
<tr>
<td>ACCPARAMOUTOFRANGE</td>
<td>9</td>
<td>&quot;Accuracy parameter out of range: must be one of LALLEAPSECunderscoreSTRIC or LALLEAPSECunderscoreLOOSE&quot;</td>
</tr>
<tr>
<td>DECRTIMETOOLARGE</td>
<td>10</td>
<td>&quot;Decrement amount too large: GPS time cannot be decremented to before the start of the GPS epoch.&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants DATEH_<name>, and the status descriptions in DATEH_MSGE<name>. The source code with these messages is in Date.h on line 1.124.

Structures

Types

Enumeration LALMSTUnits

This enumerated type is used as a parameter for Mean Sidereal Time routines to specify the units in which to return the Mean Sidereal Time. The allowed values are:

- MST_SEC: arc-seconds
- MST_HRS: arc-hours (i.e. units of Right Ascension)
- MST_DEG: degrees
- MST_RAD: radians
Enumeration **LALMonth**

This enumerated type is used to define mnemonic symbols for the **LALUnixDate** month field (`tm_mon`). The allowed values are:

- `LALMONTH_JAN` January
- `LALMONTH_FEB` February
- `LALMONTH_MAR` March
- `LALMONTH_APR` April
- `LALMONTH_MAY` May
- `LALMONTH_JUN` June
- `LALMONTH_JUL` July
- `LALMONTH_AUG` August
- `LALMONTH_SEP` September
- `LALMONTH_OCT` October
- `LALMONTH_NOV` November
- `LALMONTH_DEC` December

Enumeration **LALLeapSecAccuracy**

This enumerated type is used as a parameter for `LALGPStoUTC()`, `LALUTCtoGPS()`, and `LALLeapSecs()` to specify if complete accuracy is required in use of leap seconds. The allowed values are:

- `LALLEAPSEC_LOOSE` may miss leap seconds
- `LALLEAPSEC STRICT` require all leap seconds

If strict accuracy is selected, the code will **ABORT** if leap second data is not current. Otherwise, a warning will be printed, and the code will continue execution.

Enumeration **LALGPSCompareResult**

This enumerated type is used as the output type for `LALCompareGPS()`. The allowed values are:

- `LALGPS_EARLIER` GPS1 < GPS2
- `LALGPS_EQUAL` GPS1 = GPS2
- `LALGPS_LATER` GPS1 > GPS2

Enumeration **LALLeapSecFormat**

This enumerated type is used as a parameter for `LALLeapSecs()` to specify whether TAI-UTC or GPS-UTC should be returned. TAI-UTC is the total number of leap seconds added to UTC since the TAI epoch. GPS-UTC is the total number of leap seconds added since the GPS epoch. These two quantities are related by: TAI-UTC = GPS-UTC + 19.

- `LALLEAPSEC_TAIUTC` return TAI-UTC
- `LALLEAPSEC_GPSUTC` return GPS-UTC

Structure **LALUnixDate**

This structure is just the standard Unix `tm` structure, described in the man page for `ctime(3)`. We shall **always** ignore the daylight savings time field, `tm_isdst`.

Structure **LALTimeInterval**

This structure is used for storing intervals of `LIGOTimeGPS` and `LIGOTimeUnix` times. The fields are:

- `INT4 seconds` Integral part of the time interval
- `INT4 nanoSeconds` Residual nanoseconds (i.e. fractional part, in nanoseconds)
Structure LALDate
This structure is an extension of LALUnixDate to include residual nanosecond information. The fields are:

LALUnixDate unixDate Unix date in struct tm format
INT4 residualNanoSeconds Residual nanoseconds

Structure LALPlaceAndGPS
This structure stores pointers to a LALDetector and a LIGOTimeGPS. Its sole purpose is to aggregate these structures for passing to functions. The fields are:

LALDetector *p_detector Pointer to a detector
LIGOTimeGPS *p_gps Pointer to a GPS time structure

Structure LALPlaceAndDate
Like LALPlaceAndGPS, this structure aggregates a pointer to a detector and a pointer to a date. This is another convenience structure, used in calling LALLMST1(). The fields are:

LALDetector *p_detector Pointer to a detector
LALDate *p_date Pointer to a date

Structure LALLeapSecFormatAndAcc
This structure aggregates the LALLeapSecFormat and LALLeapSecAccuracy parameters for passing to LALLeapSecs().

The format field specifies whether LALLeapSecs() returns TAI-UTC or GPS-UTC. The accuracy field specifies whether a warning/error should be produced if the function is given an input GPS time that may result in a leap second not being accounted for.

Structure LALMSTUnitsAndAcc
Type LALMSTUnitsAndAcc
This structure aggregates the LALMSTUnits and LALLeapSecAccuracy parameters for passing to LALGPSstoGMST1() and LALGPSstoLMST1().
5.1. Module Julian.c

Converts between Gregorian date and Julian Days/Dates.

Prototypes

```c
void LALJulianDay (LALStatus *status,
                    INT4    *jDay,
                    const LALDate *date)
```

```c
void LALJulianDate (LALStatus *status,
                    REAL8   *jDateOut,
                    const LALDate *date)
```

```c
void LALModJulianDate (LALStatus *status,
                       REAL8   *modJDate,
                       const LALDate *date)
```

Description

These routines compute Julian Day, Julian Date, and Modified Julian Date for a given Gregorian date and time UTC. Julian Day and Modified Julian Day are integer number of days; Julian Date and Modified Julian Date are decimal number of days.

Algorithms

See [4] and [6] for details. First, some definitions:

- Mean Julian Year = 365.25 days
- Julian Epoch = 1 Jan 4713 BCE, 12:00 GMT (4713 BC Jan 01d.5 GMT)
- Fundamental Epoch J2000.0 = 2001-01-01.5 TDB

Julian Date is the amount of time elapsed since the Julian Epoch, measured in days and fractions of a day. There are a couple of complications arising from the length of a year: the Tropical Year is 365.2422 days. First, the Gregorian correction where 10 days (1582-10-05 through 1582-10-14) were eliminated. Second, leap years: years ending with two zeroes (e.g. 1700, 1800) are leap only if divisible by 400; so, 400 civil years contain (400 $\times$ 365.25) $-$ 3 = 146097 days. So, the Julian Date of J2000.0 is JD 2451545.0, and thus the Julian Epoch = J2000.0 + (JD - 2451545)/365.25, i.e. number of years elapsed since J2000.0.

One algorithm for computing the Julian Day is from [3] based on a formula in [4] where the algorithm is due to [2] and “compactified” by P. M. Muller and R. N. Wimberly. The formula is

\[
jd = 367 \times y - 7 \times (y + (m + 9)/12)/4 - 3 \times ((y + (m - 9)/7)/100 + 1)/4 + 275 \times m/9 + d + 1721029
\]

where \(jd\) is the Julian day number, \(y\) is the year, \(m\) is the month (1-12), and \(d\) is the day (1-31). This formula is valid only for JD $\geq$ 0, i.e. after -4713 Nov 23 = 4712 BCE Nov 23.

A shorter formula from the same reference, but which only works for dates since 1900-March is:

\[
jd = 367 \times y - 7 \times (y + (m + 9)/12)/4 + 275 \times m/9 + d + 1721014
\]

We will use this shorter formula since there is unlikely to be any analyzable data from before 1900-Mar.

Uses

Notes

Suppose we would like to get the Julian Date for today. The following program would accomplish this:
#include <lal/LALStdlib.h>
#include <lal/Date.h>

NRCSID (TESTJULIANDAYC, "Id");

int
main(int argc, char *argv[])
{
    time_t now;
    Status status = {0};
    LALDate date;
    REAL8 jDate;

    INITSTATUS (&status, "TestJulianDay", TESTJULIANDAYC);

    now = time(NULL);
    gmtime_r(&now, &(date.unixDate));

    date.unixDate.tm_sec = ltime->tm_sec;
    date.unixDate.tm_min = ltime->tm_min;
    date.unixDate.tm_hour = ltime->tm_hour;
    date.unixDate.tm_mday = ltime->tm_mday;
    date.unixDate.tm_mon = ltime->tm_mon;
    date.unixDate.tm_year = ltime->tm_year;
    date.unixDate.tm_wday = ltime->tm_wday;
    date.unixDate.tm_yday = ltime->tm_yday;
    date.unixDate.tm_isdst = 0;

    LALJulianDate(&status, &jDate, &date);
    printf("Julian Date = %10.1f\n", jDate);

    return 0;
}
5.1.2 Module **DateString.c**

Returns a formatted string for the date and time in ISO 8601 format, given the date in an **LALDate** structure.

**Prototypes**

```c
void LALDateString (LALStatus *status, CHARVector *timestamp, const LALDate *date)
```

**Description**

Returns a formatted string for the date and time in ISO 8601 format, given the date in an **LALDate** structure. A date and time in ISO 8601 format looks like `2001-03-04 17:03:52` for March 4, 2001, 5:03:52 pm. The advantage of this format, besides avoiding Y2K issues, is that a numerical-order sort of dates will result in a chronologically ordered sort. This routine is a replacement for **strftime(3c)**.

**Algorithms**

Trivial.

**Uses**

Suppose we would like to form a timestamp string for the current time. The following program would accomplish this:

```c
#include <lal/Date.h>
int main(int argc, char *argv[])
{
    static LALStatus status;
    LIGOTimeGPS gpstime;
    LALDate laldate;
    LALUnixDate utimestruct;
    CHARVector *utc = NULL;
    time_t ticks;

    INITSTATUS (status, "printone", TESTUTOGPSC);
    LALCHARCreateVector(status, &utc, (UINT4)64);
    ticks = time(NULL);
    gmtime_r(&ticks, &(laldate->unixDate));
   DateString(status, utc, &laldate);
    printf("%s\n", utc->data);
    CHARDestroyVector(status, &utc);
    RETURN (status);
}
```

**Notes**

5.1.3 Module LMST1.c

Routines related to Local and Greenwich Mean Sidereal Time (LMST1 and GMST1) computations.

Prototypes

```c
void LALGMST1 (LALStatus *status,
               REAL8 *p_gmst, /* output - GMST1 */
               const LALDate *p_date, /* input - date and time */
               LALMSTUnits outunits) /* GMST1 units */
```

```c
void LALGPStoGMST1( LALStatus *status,
                    REAL8 *p_gmst, /* output - GMST1 */
                    const LIGOTimeGPS *p_gps, /* input - GPS time */
                    const LALMSTUnitsAndAcc *pUnitsAndAcc) /* GMST1 units and accuracy */
```

```c
void LALLMST1 (LALStatus *status,
               REAL8 *p_lmst, /* output - LMST1 */
               const LALPlaceAndDate *p_place_and_date, /* input - place and date */
               LALMSTUnits outunits) /* LMST1 units */
```

```c
void LALGPStoLMST1( LALStatus *status,
                    REAL8 *p_lmst, /* output - LMST1 */
                    const LALPlaceAndGPS *p_place_and_gps, /* input - place and GPS */
                    const LALMSTUnitsAndAcc *pUnitsAndAcc) /* LMST1 units and accuracy */
```

Description

The routines in this module compute Mean Sidereal Time in a choice of units: seconds, hours, degrees, or radians. LMST1 is offset from GMST1 by the longitude of the observing post.

- **LALGMST1()** computes GMST1 given a Gregorian date UTC in an LALDate structure.
- **LALGPStoGMST1()** computes GMST1 given a GPS time.
- **LALLMST1()** computes LMST1 given an observing location (in a LALDetector structure) and a Gregorian date UTC.
- **LALGPStoLMST1()** computes LMST1 given an observing location and a GPS time.

All the routines will output GMST1 or LMST1 in the units and leap second accuracy specified by the pUnitsAndAcc argument. The sidereal time produced is within 1 sidereal second of values published in the Almanac.

Algorithms

The basic definitions and formulae are from [4], Ch. 2, Sec. 24, and also Sec. B of the Astronomical Almanac. The formula computes GMST1 for 0h UT1. To compute GMST1 for other times, a simple linear interpolation is done. The implementation used in **LALGMST1()** is from [5]. Since 1984-Jan-01, GMST has been related to UT1 as follows:

\[
\text{GMST of 0h UT1} = 24110^\circ.54841 + 8640184^\circ.812866 T_u + 0^\circ.093104 T_u^2 - 6.2 \times 10^{-6} T_u^3
\]

where \( T_u = d_u/36525 \), \( d_u \) is the number of days of Universal Time elapsed since JD 2451545.0 UT1 (January 1, 2000, at 12:00 UT1), taking on values of ±0.5, ±1.5, ±2.5, ±3.5, ...
Here is a simple example:

```c
#include <stdlib.h>
#include <lal/LALStdlib.h>
#include <lal/Date.h>

INT4 debuglevel = 2;

NRCSID (TESTLMSTC, "Id");

int
main(int argc, char *argv[])
{

    LALDate date;
    LALDate mstdate;
    REAL8 gmsthours, lmsthours;
    REAL8 gmstsecs;
    REAL8 longitude;
    LALMSTUnitsAndAcc units_and_acc;
    time_t timer;
    CHAR timestamp[64], tmpstr[64];
    Status status = {0};

    if (argc == 1)
    {
        printf("Usage: TestUTCtoGPS debug_level -- debug_level = [0,1,2]\n");
        return 0;
    }

    if (argc == 2)
        debuglevel = atoi(argv[1]);

    INITSTATUS(&status, "TestLMST", TESTLMSTC);

    printf("TEST of GMST1 routine\n");
    printf("================\n");

    // Check against the Astronomical Almanac:
    // For 1994-11-16 0h UT - Julian Date 2449672.5, GMST 03h 39m 21.2738s
    date.unixDate.tm_sec = 0;
    date.unixDate.tm_min = 0;
    date.unixDate.tm_hour = 0;
    date.unixDate.tm_mday = 16;
    date.unixDate.tm_mon = LALMONTH_NOV;
    date.unixDate.tm_year = 94;
    longitude = 0.;
    LALGMST1(&status, &gmsthours, &date, MST_HRS);
    LALLMST1(&status, &lmsthours, &date, longitude, MST_HRS);

    LALGMST1(&status, &gmstsecs, &date, MST_SEC);
    LALSecsToLALDate(&status, &mstdate, gmstsecs);
    strftime(timestamp, 64, "%Hh %Mm %S", &(mstdate.unixDate));
    sprintf(tmpstr, "%fs", mstdate.residualNanoSeconds * 1.e-9);
    strcat(timestamp, tmpstr+1);
```
printf("gmsthours = %f = %s\n", gmsthours, timestamp);
printf(" expect: 3.655728 = 03h 39m 20.6222s \n");

return(0);
}

From [4]:

Universal Time (UT) is the measure of time used as the basis for all civil time-keeping. It conforms closely to the mean diurnal motion of the Sun. The apparent diurnal motion of the Sun involves both the nonuniform diurnal rotation of the Earth and the motion of the Earth in its orbit around the Sun. Although it would be possible to define a system of time measurement in terms of the hour angle of the Sun, such a system could never be related precisely to sidereal time and could not, therefore, be determined by observations of star transits. As a result, Universal Time is directly related to sidereal time by means of a numerical formula. It does not refer to the motion of the Earth and is not precisely related to the hour angle of the Sun.

Universal Time at any instant can be derived from observations of the diurnal motion of the stars or radio sources. The uncorrected observed rotational timescale, which is dependent on the place of observation, is designated UT0. Correcting this timescale for the shift in longitude of the observing station caused by polar motion produces the UT1 timescale, which is independent of observing location, but is influenced by the slightly variable rotation of the Earth.

5.1.4 Module SecsToLALDate.c

Converts time in seconds to time in an LALDate structure.

Prototypes

```c
void
LALSecsToLALDate(LALStatus *status,
                  LALDate *date,  /* output - date */
                  REAL8 seconds) /* input - time in seconds since 0h */
```

Description

This routine converts a time of day in decimal seconds since 0h (midnight) to an LALDate structure. Of course, the date information is not present.

Algorithms

Uses

A simple example:

```c
#include <stdlib.h>
#include <lal/LALStdlib.h>
#include <lal/Date.h>

INT4 debuglevel = 2;
NRCSID (TESTLMSTC, "Id");

int main(int argc, char *argv[]) {  
  LALDate date;
  LALDate mstdate;
  REAL8 gmstsecs;
  CHAR timestamp[64], tmpstr[64];
  Status status = {0};

  INITSTATUS(&status, "TestLMST", TESTLMSTC);

  printf("TEST of GMST1 routine\n");
  printf("=*=*=*=*=*=*=*=*=*=*=*=*=*\n");

  date.unixDate.tm_sec = 0;
  date.unixDate.tm_min = 0;
  date.unixDate.tm_hour = 0;
  date.unixDate.tm_mday = 16;
  date.unixDate.tm_mon = LALMONTH_NOV;
  date.unixDate.tm_year = 94;

  GMST1(&status, &gmstsecs, &date, MST_SEC);
  SecsToLALDate(&status, &mstdate, gmstsecs);
  strftime(timestamp, 64, "%Hh %Mm %S", &mstdate.unixDate);
  sprintf(tmpstr, "%fs", mstdate.residualNanoSeconds * 1.e-9);
  strcat(timestamp, tmpstr+1);

  printf("gmst = %s\n", timestamp);
  return 0;
}
```
printf(" expect: 03h 39m 20.6222s \n");

return(0);
}

Notes
5.1.5 Module GPStoUTC.c

Converts between GPS time (in seconds and nanoseconds) and UTC in a `LALDate` structure.

Prototypes

```c
void GPStoUTC (LALStatus *status,
               LALDate *p_utcDate, /* output - date */
               const LIGOTimeGPS *p_gpsTime, /* input - GPS seconds */
               const LALLeapSecAccuracy *p_accuracy) /* accuracy of
                  leap-second accounting: LALLeapSec_OFFSET, or LALLeapSec_OFFSET */

void UTCtoGPS (LALStatus *status,
               LIGOTimeGPS *p_gpsTime, /* output - GPS seconds */
               const LALDate *p_utcDate, /* input - date in UTC */
               const LALLeapSecAccuracy *p_accuracy) /* accuracy of
                  leap-second accounting: LALLeapSec_OFFSET, or LALLeapSec_OFFSET */

void LeapSecs (LALStatus *status,
               INT4 *p_leapSecs, /* output - GPS-UTC,
                  i.e. the number of
                  leap-seconds introduced
                  since the GPS epoch
                  1980-Jan-06 */
               const LIGOTimeGPS *p_gpsTime, /* input - GPS time */
               const LALLeapSecFormatAndAcc *p_formatAndAcc) /* format and
                  accuracy parameters */
```

Description

`LALGPStoUTC()` and `LALUTCtoGPS` convert time in GPS seconds and nanoseconds (`LIGOTimeGPS`) and time in UTC (`LALDate`), taking into account leap seconds until 2006-Dec-31 23:59:59 UTC.

`LALLeapSecs()` returns the number of leap seconds introduced since the GPS epoch 1980-Jan-06, abbreviated GPS-UTC.

Algorithms

The conversion from GPS to UTC is copied directly from GRASP [7]. It does the conversion by counting TAI seconds starting from the Unix epoch origin, 1970-Jan-01 00:00:00 UTC. A static table of leap seconds is compiled in: this must be updated whenever a new leap second is introduced. The latest leap second included is 2006-Jan-01.

The conversion from UTC to GPS is done by counting the amount of elapsed time since the GPS epoch origin, 1980-Jan-06 00:00:00 UTC. Again, leap seconds are accounted for by a static table (different from the one used in GPS to UTC) which must be updated whenever a new leap second is introduced. The latest leap second included is 2006-Jan-01.

Uses

Notes

These routines will not work for times before 1980-01-06 00:00:00 UTC (GPS 0). The latest leap second that can be accounted for is the one added at the end of 2005-Dec. information until 2006-Dec-31.

**Example:** To convert a GPS time to UTC, and then back to GPS:

```c
#include <lal/LALStdlib.h>
#include <lal/Date.h>

struct tm *gmtime_r( const time_t *, struct tm * );
char *asctime_r( const struct tm *, char *, int );

int main(int argc, char *argv[])
{
    static LALStatus status;
    LIGOTimeGPS gps = {615081613, 123456789};
    LALDate date;
    LALLeapSecAccuracy accuracy = LALLEAPSEC_STRICT;
    CHARVector *timestamp = NULL;

    LALCHARCreateVector(&status, &timestamp, (UINT4)128);
    LALGPStoUTC(&status, &date, &gps, &accuracy);
    LALDateString(&status, timestamp, &date);
    printf("GPS (%d, %d) = %s\n", gps.gpsSeconds, gps.gpsNanoSeconds, timestamp->data);
    LALUTCtoGPS(&status, &gps, &date, &accuracy);
    printf("%s = GPS (%d, %d)\n", timestamp->data, gps.gpsSeconds, gps.gpsNanoSeconds);
    return 0;
}
```

For an example of how `LALLeapSecs()` is used, see the test program `TestLeapSecs.c` in the `packages/date/test` directory.
5.1.6 Module GPStoFloat.c

Converts between LIGOTimeGPS and REAL8 formats, and also to/from LALTimeInterval and REAL8 formats.

Prototypes

```c
void LALGPStoFloat( LALStatus *stat, REAL8 *p_flt_time, /* output - floating point GPS seconds */
                const LIGOTimeGPS *p_gps_time) /* input - GPS seconds */

void XLALFloatToGPS( LIGOTimeGPS *p_gps_time, /* output - GPS time */
                const REAL8 flt_time) /* input - floating point GPS seconds */

void LALFloatToGPS( LALStatus *stat, LIGOTimeGPS *p_gps_time, /* output - GPS time */
                const REAL8 *p_flt_time) /* input - floating point GPS seconds */

void LALFloatToInterval(LALStatus *status,
                LALTimeInterval *pInterval, /* output: deltaT in LALTimeInterval format */
                const REAL8 *pDeltaT) /* input: time interval in floating point */

void LALIntervalToFloat(LALStatus *status,
                REAL8 *pDeltaT, /* output: floating point time interval */
                const LALTimeInterval *pInterval) /* input: LALTimeInterval */
```

Description

This module contains two routines, one of which converts from LIGOTimeGPS to REAL8, and the other, from REAL8 to LIGOTimeGPS. Accuracy is on par with what one expects from a typical IEEE-compliant machine epsilon; thus, conversion into the REAL8 values incurs an error of approximately 1.e-7.

- LALGPStoFloat() converts a LIGOTimeGPS to REAL8
- LALFloatToGPS() converts a time in REAL8 to a LIGOTimeGPS
- LALFloatToInterval() converts a time interval in REAL8 to a LALTimeInterval
- LALIntervalToFloat() converts a time interval in LALTimeInterval to REAL8
5.1.7 Module GPStoINT8.c

Converts between LIGOTimeGPS and INT8 formats.

Prototypes

LIGOTimeGPS *
XLALINT8stoGPS (  
    LIGOTimeGPS *output,  
    INT8 input
)

void
LALINT8stoGPS (  
    LALStatus *status,  
    LIGOTimeGPS *output,  
    const INT8 *input
)

INT8
XLALGPStoINT8 (  
    const LIGOTimeGPS *input
)

void
LALGPStoINT8 (  
    LALStatus *status,  
    INT8 *output,  
    const LIGOTimeGPS *input
)

Description

This module contains two LAL routines and their XLAL counterparts. One pair of routines converts from LIGOTimeGPS to INT8 nanoseconds, and the other, from INT8 nanoseconds to LIGOTimeGPS.

Algorithm

Uses

Notes
5.1.8 Module *IncrementGPS.c*

Routines to perform arithmetic and comparisons on *LIGOTimeGPS* and *LALTimeInterval* data types.

Prototypes

```c
void LALIncrementGPS (LALStatus *status, LIGOTimeGPS *pIncrementedGPS, /* output */
            const LIGOTimeGPS *pInitialGPS, /* input: GPS time */
            const LALTimeInterval *pDeltaT) /* input: interval to increment by */

void LALDecrementGPS (LALStatus *status, LIGOTimeGPS *pDecrementedGPS, /* output */
            const LIGOTimeGPS *pInitialGPS, /* input: GPS time */
            const LALTimeInterval *pDeltaT) /* input: interval to decrement by */

void LALDeltaGPS (LALStatus *status, LALTimeInterval *pDeltaGPS, /* output: GPS1 - GPS2 */
            const LIGOTimeGPS *pGPS1, /* input: GPS1 */
            const LIGOTimeGPS *pGPS2) /* input: GPS2 */

void LALCompareGPS(LALStatus *status, LALGPSCompareResult *pResult, /* output: -1 => GPS1 < GPS2
  0 => GPS1 = GPS2 1 => GPS1 > GPS2 */
            const LIGOTimeGPS *pGPS1, /* input: GPS1 */
            const LIGOTimeGPS *pGPS2) /* input: GPS2 */

LIGOTimeGPS * XLALAddFloatToGPS( LIGOTimeGPS *gps, REAL8 deltaT )

void LALAddFloatToGPS( LALStatus *status, LIGOTimeGPS *outputGPS, /* outputGPS = startGPS + deltaT */
            const LIGOTimeGPS *startGPS, /* input: GPS time */
            REAL8 deltaT /* input: interval to increment by in sec */
)

REAL8 XLALDeltaFloatGPS( const LIGOTimeGPS *GPS1, const LIGOTimeGPS *GPS2
)

void LALDeltaFloatGPS (LALStatus *status, REAL8 *deltaT, /* GPS1 - GPS2 */
            const LIGOTimeGPS *GPS1, /* input: GPS1 */
            const LIGOTimeGPS *GPS2) /* input: GPS2 */
```
Description

This module contains a few utility routines to perform comparisons and arithmetic on LIGOTimeGPS GPS times. These routines do not convert the GPS times they operate on into a floating point representation.

- `LALIncrementGPS()` increments a GPS time by a time interval
- `LALDecrementGPS()` decrements a GPS time by a time interval
- `LALAddFloatToGPS()` adds a REAL8 interval (in seconds) to a GPS time
- `XLALAddFloatToGPS()` adds a REAL8 interval (in seconds) to a GPS time
- `LALDeltaGPS()` returns the difference between two GPS times as a `LALTimeInterval`.
- `LALDeltaFloatGPS()` returns the difference between two GPS times in seconds as a `REAL8`.
- `XLALDeltaFloatGPS()` returns the difference between two GPS times in seconds as a `REAL8`.
- `LALCompareGPS()` compares two GPS times, and returns a `LALGPSCompareResult` indicating if the first GPS time is earlier than, equal to, or later than the second GPS time

Algorithm

Uses

Notes

In the `LALIncrementGPS()`, `LALDecrementGPS()` and `LALAddFloatToGPS()` routines, it is legal to pass a pointer to the same `LIGOTimeGPS` structure as input and output, e.g.

```c
LALStatus status;
LIGOTimeGPS gps;
LALTimeInterval interval;

...

gps.gpsSeconds = 10;
gps.gpsNanoSeconds = 25;
interval.seconds = 13;
interval.nanoSeconds = 1000;

LALIncrementGPS(&status, &gps, &gps, &interval);
```
5.1.9 Module GPSTimeNow.c

Routine to convert the current unix system clock time into a LIGOTimeGPS structure.

Prototypes

```
LIGOTimeGPS *
XLALGPSTimeNow (  
    LIGOTimeGPS *gpstime  
)

void
LALGPSTimeNow (  
    LALStatus       *status,  
    LIGOTimeGPS     *gpstime,  
    const LALLeapSecAccuracy *accuracy  
)
```

Description

This module contains a single function that converts the current unix system time as returned by the `time()` function to GPS seconds. The leap second accuracy is determined by the `accuracy` argument.

Algorithm

Uses

Notes

If the system clock time is incorrect, the returned GPS time will obviously be wrong by the same amount.
5.1.10 Module Playground.c

Determines if a given time (or segment) is playground data.

Prototypes

```c
int XLALINT8NanoSecIsPlayground (INT8 *ns)

void LALINT8NanoSecIsPlayground (LALStatus *status, INT4 *playground, INT8 *ns)

void LALGPSIsPlayground (LALStatus *status, INT4 *playground, LIGOTimeGPS *gpstime)

void LALSegmentIsPlayground (LALStatus *status, INT4 *playground, LIGOTimeGPS *gpsStart, LIGOTimeGPS *gpsEnd)
```

Description

This module contains two routines to determine if a given time is in the data designated as playground or not. The first routines takes input as `INT8` nanoseconds and the second as a `LIGOTimeGPS` structure. The third routine decides if some or all of a given time interval is playground or not.

Algorithm

The playground algorithm is given in LIGO technical document T030020-01. Briefly, $t$ is playground if

$$t - 729273613\%6370 < 600.$$  \hspace{1cm} (5.1)

Uses

Notes

Author: Brown D. A.

$Id: Playground.c,v 1.5 2007/06/08 14:41:43 bema Exp$
5.2 Header TimeDelay.h

Provides routine to compute time delay between two detectors.

Synopsis

```
#include <lal/TimeDelay.h>
```

This header provides prototypes of routines to compute the difference in time for a signal to arrive at two detectors. The routine is a direct translation of the Maple worksheet by Anderson, et al., available at http://dirac.utb.edu/~warren/unprot/beam_patterns.tar.gz.

Error conditions

```
<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUL</td>
<td>1</td>
<td>&quot;Unexpected null pointer in arguments&quot;</td>
</tr>
</tbody>
</table>
```

The status codes in the table above are stored in the constants `TIMEDELAYH_E<name>`, and the status descriptions in `TIMEDELAYH_MSGE<name>`. The source code with these messages is in `TimeDelay.h` on line 1.74.

Structures

**TwoDetsTimeAndASource**

This structure stores two pointers to `LALPlaceAndGPS` structures, and a pointer to a `SkyPosition` structure. The fields are:

- `LALPlaceAndGPS *p_det_and_time1` The first detector and GPS
- `LALPlaceAndGPS *p_det_and_time2` The second detector and GPS
- `SkyPosition *p_source` The source location (equatorial coordinates in decimal radians)

**DetTimeAndASource**

This structure stores one pointer to a `LALPlaceAndGPS` structure, and a pointer to a `SkyPosition` structure. The fields are:

- `LALPlaceAndGPS *p_det_and_time` The detector and GPS
- `SkyPosition *p_source` The source location (equatorial coordinates in decimal radians)
5.2.1 Module TimeDelay.c

Computes difference in arrival time of the same signal at two different detectors.

Prototypes

```c
double
XLALArrivalTimeDiff(
    const double detector1_earthfixed_xyz_metres[3],
    const double detector2_earthfixed_xyz_metres[3],
    const double source_right_ascension_radians,
    const double source_declination_radians,
    const LIGOTimeGPS *gpstime
)

void
LALTimeDelay(
    LALStatus *stat,
    REAL8 *p_time_diff,
    const TwoDetsTimeAndASource *p_dets_time_and_source
)

INT8
XLALLightTravelTime(
    const LALDetector *aDet,
    const LALDetector *bDet
)
```

Description

The function LALTimeDelay() computes the difference in time of arrival of a signal at two detectors from the same source. The two detectors and the source are passed in a `TwoDetsTimeAndASource` structure. The time delay is defined to be \( \delta t = t_2 - t_1 \) where \( t_1 \) is the time the signal arrives at the first detector and \( t_2 \) is the time the signal arrives at the second detector.

The function LALTimeDelayFromEarthCenter() computes difference in arrival time of the same signal at detector and at center of Earth-fixed frame. Equivalent to LALTimeDelay() with detector 1 set to the geocenter.

The function XLALLightTravelTime() computes the light travel time between two detectors and returns the answer in `INT8` nanoseconds.

The function XLALPopulateAccuracyParams creates an instance of InspiralAccuracyList populated with the light-travel times between the detectors, using just the previous function. The function XLALPopulateAccuracyParamsExt, however, creates an instance of InspiralAccuracyList populated with the real travel time of a putative signal for the given time and the given sky location (in right ascension and declination, both given in degrees).

Algorithm


Note that GPS time is passed with both the detectors. The GPS time of the second detector is ignored, and the GPS time for the first detector is taken to be the time when the signal arrives at the center of the Earth. In practice, this time will be the time of detection of a signal at the first detector, but, as in Anderson, et al., we make this approximation as it makes little difference. This time is used to compute a GMST which gives us the orientation of the Earth.

Uses

Notes

Author: Chin, David <dwchin@umich.edu> +1-734-709-9119, Kipp Cannon <kipp@gravity.phys.uwm.edu>

$Id: TimeDelay.c,v 1.22 2008/01/18 23:32:00 kipp Exp$
References


Chapter 6

Package **factories**

This package provides routines for creating and destroying the LAL aggregate datatypes.
6.1 Header AVFactories.h

Provides prototype and status code information for use of CreateVector, CreateArray, ResizeVector, ResizeArray, DestroyVector and DestroyArray

Synopsis
#include <lal/AVFactories.h>

Error conditions

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LENGTH</td>
<td>1</td>
<td>&quot;Illegal length.&quot;</td>
</tr>
<tr>
<td>VPTR</td>
<td>2</td>
<td>&quot;Null vector/array handle.&quot;</td>
</tr>
<tr>
<td>UPTR</td>
<td>4</td>
<td>&quot;Non-null vector/array pointer.&quot;</td>
</tr>
<tr>
<td>DPTR</td>
<td>8</td>
<td>&quot;Null vector/array data.&quot;</td>
</tr>
<tr>
<td>MALLOC</td>
<td>16</td>
<td>&quot;Malloc failure.&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants AVFACTORIESH_E<name>, and the status descriptions in AVFACTORIESH_MSGE<name>. The source code with these messages is in AVFactories.h on line 1.59.
6.1.1 Module VectorFactories.c

Create/destroy (datatype)Vector objects.

Prototypes

```c
void LALZCreateVector ( LALStatus *status, COMPLEX16Vector **vector, UINT4 length )
void LALZResizeVector ( LALStatus *status, COMPLEX16Vector **vector, UINT4 length )
void LALZDestroyVector ( LALStatus *status, COMPLEX16Vector **vector )
void LALCCreateVector ( LALStatus *status, COMPLEX8Vector **vector, UINT4 length )
void LALCResizeVector ( LALStatus *status, COMPLEX8Vector **vector, UINT4 length )
void LALCDestroyVector ( LALStatus *status, COMPLEX8Vector **vector )
void LALDCreateVector ( LALStatus *status, REAL8Vector **vector, UINT4 length )
void LALDResizeVector ( LALStatus *status, REAL8Vector **vector, UINT4 length )
void LALDDestroyVector ( LALStatus *status, REAL8Vector **vector )
void LALSCreateVector ( LALStatus *status, REAL4Vector **vector, UINT4 length )
void LALSResizeVector ( LALStatus *status, REAL4Vector **vector, UINT4 length )
void LALSDestroyVector ( LALStatus *status, REAL4Vector **vector )
void LALI2CreateVector ( LALStatus *status, INT2Vector **vector, UINT4 length )
void LALI2ResizeVector ( LALStatus *status, INT2Vector **vector, UINT4 length )
void LALI2DestroyVector ( LALStatus *status, INT2Vector **vector )
void LALI4CreateVector ( LALStatus *status, INT4Vector **vector, UINT4 length )
void LALI4ResizeVector ( LALStatus *status, INT4Vector **vector, UINT4 length )
void LALI4DestroyVector ( LALStatus *status, INT4Vector **vector )
void LALI8CreateVector ( LALStatus *status, INT8Vector **vector, UINT4 length )
void LALI8ResizeVector ( LALStatus *status, INT8Vector **vector, UINT4 length )
void LALI8DestroyVector ( LALStatus *status, INT8Vector **vector )
void LALU2CreateVector ( LALStatus *status, UINT2Vector **vector, UINT4 length )
void LALU2ResizeVector ( LALStatus *status, UINT2Vector **vector, UINT4 length )
void LALU2DestroyVector ( LALStatus *status, UINT2Vector **vector )
void LALU4CreateVector ( LALStatus *status, UINT4Vector **vector, UINT4 length )
void LALU4ResizeVector ( LALStatus *status, UINT4Vector **vector, UINT4 length )
void LALU4DestroyVector ( LALStatus *status, UINT4Vector **vector )
void LALU8CreateVector ( LALStatus *status, UINT8Vector **vector, UINT4 length )
void LALU8ResizeVector ( LALStatus *status, UINT8Vector **vector, UINT4 length )
void LALU8DestroyVector ( LALStatus *status, UINT8Vector **vector )
void LALCHARCreateVector ( LALStatus *status, CHARVector **vector, UINT4 length )
void LALCHARResizeVector ( LALStatus *status, CHARVector **vector, UINT4 length )
void LALCHARDestroyVector ( LALStatus *status, CHARVector **vector )
void LALCreateVector ( LALStatus *status, REAL4Vector **vector, UINT4 length )
void LALResizeVector ( LALStatus *status, REAL4Vector **vector, UINT4 length )
void LALDestroyVector ( LALStatus *status, REAL4Vector **vector )
```
Description

The CreateVector family of functions create a (datatype) Vector of the appropriate dimensions.

The ResizeVector family of functions changes the amount of storage allocated by the CreateVector functions.

The DestroyVector family of functions return the storage allocated by the CreateVector functions to the system.

Algorithm

Uses

LALMalloc()
LALFree()

Notes
6.1.2 Module ArrayFactories.c

Create/destroy (datatype)Array objects.

Prototypes

```c
void LALZCreateArray ( LALStatus *status, COMPLEX16Array **array, UINT4Vector *dimLength )
void LALZResizeArray ( LALStatus *status, COMPLEX16Array **array, UINT4Vector *dimLength )
void LALZDestroyArray ( LALStatus *status, COMPLEX16Array **array )
void LALCCreateArray ( LALStatus *status, COMPLEX8Array **array, UINT4Vector *dimLength )
void LALCResizeArray ( LALStatus *status, COMPLEX8Array **array, UINT4Vector *dimLength )
void LALCDestroyArray ( LALStatus *status, COMPLEX8Array **array )
void LALDCreateArray ( LALStatus *status, REAL8Array **array, UINT4Vector *dimLength )
void LALDResizeArray ( LALStatus *status, REAL8Array **array, UINT4Vector *dimLength )
void LALDDestroyArray ( LALStatus *status, REAL8Array **array )
void LALSCreateArray ( LALStatus *status, REAL4Array **array, UINT4Vector *dimLength )
void LALSResizeArray ( LALStatus *status, REAL4Array **array, UINT4Vector *dimLength )
void LALSDestroyArray ( LALStatus *status, REAL4Array **array )
void LALI2CreateArray ( LALStatus *status, INT2Array **array, UINT4Vector *dimLength )
void LALI2ResizeArray ( LALStatus *status, INT2Array **array, UINT4Vector *dimLength )
void LALI2DestroyArray ( LALStatus *status, INT2Array **array )
void LALI4CreateArray ( LALStatus *status, INT4Array **array, UINT4Vector *dimLength )
void LALI4ResizeArray ( LALStatus *status, INT4Array **array, UINT4Vector *dimLength )
void LALI4DestroyArray ( LALStatus *status, INT4Array **array )
void LALI8CreateArray ( LALStatus *status, INT8Array **array, UINT4Vector *dimLength )
void LALI8ResizeArray ( LALStatus *status, INT8Array **array, UINT4Vector *dimLength )
void LALI8DestroyArray ( LALStatus *status, INT8Array **array )
void LALU2CreateArray ( LALStatus *status, UINT2Array **array, UINT4Vector *dimLength )
void LALU2ResizeArray ( LALStatus *status, UINT2Array **array, UINT4Vector *dimLength )
void LALU2DestroyArray ( LALStatus *status, UINT2Array **array )
void LALU4CreateArray ( LALStatus *status, UINT4Array **array, UINT4Vector *dimLength )
void LALU4ResizeArray ( LALStatus *status, UINT4Array **array, UINT4Vector *dimLength )
void LALU4DestroyArray ( LALStatus *status, UINT4Array **array )
void LALU8CreateArray ( LALStatus *status, UINT8Array **array, UINT4Vector *dimLength )
void LALU8ResizeArray ( LALStatus *status, UINT8Array **array, UINT4Vector *dimLength )
void LALU8DestroyArray ( LALStatus *status, UINT8Array **array )
void LALCreateArray ( LALStatus *status, REAL4Array **array, UINT4Vector *dimLength )
void LALResizeArray ( LALStatus *status, REAL4Array **array, UINT4Vector *dimLength )
void LALDestroyArray ( LALStatus *status, REAL4Array **array )
```
Description

The CreateArray family of functions create a (datatype) Array of the appropriate dimensions.

The DestroyArray family of functions return the storage allocated by the CreateArray functions to the system.

Algorithm

Uses

LALMalloc()
LALFree()

Notes
6.1.3 XLAL Functions

Synopsis

```
#include <lal/AVFactories.h>

REAL4Vector * XLALCreateVector(UINT4 length);
REAL4Vector * XLALResizeVector(REAL4Vector *vector, UINT4 length);
void XLALDestroyVector(REAL4Vector *vector, UINT4 length);

<vectype> * XLALCreate<vectype>(UINT4 length);
<vectype> * XLALResize<vectype>(<vectype> *vector, UINT4 length);
void XLALDestroy<vectype>(<vectype> *vector);

REAL4Array * XLALCreateArrayL(UINT4 ndim, ...);
REAL4Array * XLALCreateArrayV(UINT4 ndim, UINT4 *dims);
REAL4Array * XLALCreateArray(UINT4Vector *dimLength);
REAL4Array * XLALResizeArrayL(REAL4Array *array, UINT4 ndim, ...);
REAL4Array * XLALResizeArrayV(REAL4Array *array, UINT4 ndim, UINT4 *dims);
REAL4Array * XLALResizeArray(REAL4Array *array, UINT4Vector *dimLength);
void XLALDestroyArray(REAL4Array *array);

<arrtype> * XLALCreate<arrtype>L(UINT4 ndim, ...);
<arrtype> * XLALCreate<arrtype>V(UINT4 ndim, UINT4 *dims);
<arrtype> * XLALCreate<arrtype>(UINT4Vector *dimLength);
<arrtype> * XLALResize<arrtype>L(<arrtype> *array, UINT4 ndim, ...);
<arrtype> * XLALResize<arrtype>V(<arrtype> *array, UINT4 ndim, UINT4 *dims);
<arrtype> * XLALResize<arrtype>(<arrtype> *array, UINT4Vector *dimLength);
void XLALDestroy<arrtype>(<arrtype> *array);
```

Here `<vectype>` is one of COMPLEX16Vector, COMPLEX8Vector, REAL8Vector, REAL4Vector, INT8Vector, INT4Vector, INT2Vector, UINT8Vector, UINT4Vector, UINT2Vector, or CHARVector, and `<arrtype>` is one of COMPLEX16Array, COMPLEX8Array, REAL8Array, REAL4Array, INT8Array, INT4Array, INT2Array, UINT8Array, UINT4Array, or UINT2Array.

Description

The XLALCreate<type>Vector functions create vectors of the specified `length` number of objects of type `<type>`. The function XLALCreateVector is the same as XLALCreateREAL4Vector.

The XLALDestroy<type>Vector functions deallocate the memory allocation pointed to by `vector` including its contents. The function XLALDestroyVector is the same as XLALDestroyREAL4Vector.

The XLALResize<type>Vector functions resize the supplied vector `vector` to the new size `length`. If `vector` is NULL then this is equivalent to XLALCreate<type>Vector. If `length` is zero then this is equivalent to XLALDestroy<type>Vector and the routine returns NULL. Otherwise, the amount of data in the vector is reallocated using LALRealloc. The function XLALResizeVector is the same as XLALResizeREAL4Vector.

The XLALCreate<type>Array XLALCreate<type>ArrayL XLALCreate<type>ArrayV all create an object of type `<type>Array`. They differ in the way that the dimensions of the array are specified. The function XLALCreate<type>Array allocates an array with dimensions specified by the UINT4Vector `dimLength` which is a vector of dimension lengths for the array. The function XLALCreate<type>ArrayV provides these dimensions with two arguments: `ndim` is the number of dimensions and `dims` is an array of UINT4 values for the dimensions. The function XLALCreate<type>ArrayL also specifies the dimensions as arguments. Here, the first argument, `ndim`, is the number of dimensions, and this is followed by `ndim` arguments that provide the dimensions. Note that for this function, a maximum of 16 dimensions can be provided (that is, `ndim` cannot be more than 16 and there cannot be more than 16 arguments after the first). The XLALCreateArray XLALCreateArrayL XLALCreateArrayV functions are equivalent to the XLALCreateREAL4Array XLALCreateREAL4ArrayL XLALCreateREAL4ArrayV functions respectively.

The XLALDestroy<type>Array functions deallocate the memory allocation pointed to by `array` including its contents. The function XLALDestroyArray is the same as XLALDestroyREAL4Array.

The XLALResize<type>Array XLALResize<type>ArrayL XLALResize<type>ArrayV functions resize the provided array `array`. The arguments after the first are interpreted in the same
way as for the `XLALCreate<type>Array` `XLALCreate<type>ArrayL` `XLALCreate<type>ArrayV` functions. If `array` is `NULL`, the resize functions are equivalent to the corresponding create function. If `ndim` is zero for `XLALResize<type>ArrayL` or `XLALResize<type>ArrayV`, or if `dimLength` is `NULL` for `XLALResize<type>Array`, then these functions are equivalent to `XLALDestroy<type>Array`. The `XLALResizeArray` `XLALResizeArrayL` `XLALResizeArrayV` functions are equivalent to the `XLALResizeREAL4Array` `XLALResizeREAL4ArrayL` `XLALResizeREAL4ArrayV` functions respectively.

Return Values

If successful, the create and resize functions return a pointer to the same data that was passed to the function. The resize functions will return a `NULL` pointer if the size of the new object was zero. Upon failure these routines will return `NULL` and will set `xlalErrno` to one of these values: `XLAL_ENOMEM` if a memory allocation failed, `XLAL_EBADLEN` if an invalid length was provided (for example, a zero-size allocation with a create function), `XLAL_EINVAL` if an invalid argument is provided (for example, if the pointer to an array of dimensions is `NULL`).

The destroy function does not return any value. If the function is passed a `NULL` pointer, it will set `xlalErrno` to `XLAL_EFAULT`. If the function is passed an object that appears corrupted (e.g., a vector with zero length or will a `NULL` data pointer) it will set `xlalErrno` to `XLAL_EINVAL`. 
6.1.4 Program VectorFactoriesTest.c

A program to test create/destroy vector routines.

Usage

VectorFactoriesTest [options]

Options:

- h     print help
- q     quiet: run silently
- v     verbose: print extra information
- d level set lalDebugLevel to level

Description

Exit codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success, normal exit.</td>
</tr>
<tr>
<td>1</td>
<td>Subroutine failed.</td>
</tr>
</tbody>
</table>

Algorithm

Uses

lalDebugLevel
TYPECODECreateVector()
TYPECODEResizeVector()
TYPECODEDestroyVector()

Notes
6.1.5 Program ArrayFactoriesTest.c

A program to test create/destroy array routines.

Usage

ArrayFactoriesTest [options]
Options:
  -h     print help
  -q     quiet: run silently
  -v     verbose: print extra information
  -d level set lalDebugLevel to level

Description

Exit codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success, normal exit.</td>
</tr>
<tr>
<td>1</td>
<td>Subroutine failed.</td>
</tr>
</tbody>
</table>

Algorithm

Uses

lalDebugLevel
TYPECODECreateArray()
TYPECODEDestroyArray()

Notes
6.2 Header **SeqFactories.h**

Provides prototype and status code information for use of CreateVectorSequence and DestroyVectorSequence.

**Synopsis**

```c
#include <lal/SeqFactories.h>
```

**Error conditions**

<table>
<thead>
<tr>
<th>name</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLENGTH</td>
<td>1</td>
<td>&quot;Illegal sequence length.&quot;</td>
</tr>
<tr>
<td>VLENGTH</td>
<td>2</td>
<td>&quot;Illegal vector length.&quot;</td>
</tr>
<tr>
<td>ALENGTH</td>
<td>4</td>
<td>&quot;Illegal array dimension.&quot;</td>
</tr>
<tr>
<td>VPTR</td>
<td>8</td>
<td>&quot;Null sequence handle.&quot;</td>
</tr>
<tr>
<td>UPTR</td>
<td>16</td>
<td>&quot;Non-null sequence pointer.&quot;</td>
</tr>
<tr>
<td>DPTR</td>
<td>32</td>
<td>&quot;Null sequence data.&quot;</td>
</tr>
<tr>
<td>INPTR</td>
<td>64</td>
<td>&quot;Null input pointer.&quot;</td>
</tr>
<tr>
<td>MALLOC</td>
<td>128</td>
<td>&quot;Malloc failure.&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants **SEQFACTORIESH_E<name>**, and the status descriptions in **SEQFACTORIESH_MSGE<name>**. The source code with these messages is in **SeqFactories.h** on line 1.68.

**Structures**

**CreateVectorSequenceIn**

This structure stores the input required for creating a vector sequence. This input includes the length of the sequence (i.e., the number of vectors) and the length of each vector. The fields are:

- `UINT4 length` The sequence length.
- `UINT4 vectorLength` The length of each vector in the sequence.

**CreateArraySequenceIn**

This structure stores the input required for creating an array sequence. This input includes the length of the sequence (i.e., the number of array) and the dimensions of each array index. The fields are:

- `UINT4 length` The sequence length.
- `UINT4Vector *dimLength` The dimensions of each array index (the same for every array in the sequence).
6.2.1 Module VectorSequenceFactories.c

Create/destroy (datatype)VectorSequence objects.

Prototypes

void LALZCreateVectorSequence ( LALStatus *status, COMPLEX16VectorSequence **vseq, CreateVectorSequenceIn *in )
void LALZDestroyVectorSequence ( LALStatus *status, COMPLEX16VectorSequence **vseq )
void LALCCreateVectorSequence ( LALStatus *status, COMPLEX8VectorSequence **vseq, CreateVectorSequenceIn *in )
void LALCDestroyVectorSequence ( LALStatus *status, COMPLEX8VectorSequence **vseq )
void LALDCreateVectorSequence ( LALStatus *status, REAL8VectorSequence **vseq, CreateVectorSequenceIn *in )
void LALDDestroyVectorSequence ( LALStatus *status, REAL8VectorSequence **vseq )
void LALSCreateVectorSequence ( LALStatus *status, REAL4VectorSequence **vseq, CreateVectorSequenceIn *in )
void LALSDestroyVectorSequence ( LALStatus *status, REAL4VectorSequence **vseq )
void LL12CreateVectorSequence ( LALStatus *status, INT2VectorSequence **vseq, CreateVectorSequenceIn *in )
void LL12DestroyVectorSequence ( LALStatus *status, INT2VectorSequence **vseq )
void LL14CreateVectorSequence ( LALStatus *status, INT4VectorSequence **vseq, CreateVectorSequenceIn *in )
void LL14DestroyVectorSequence ( LALStatus *status, INT4VectorSequence **vseq )
void LL8CreateVectorSequence ( LALStatus *status, INT8VectorSequence **vseq, CreateVectorSequenceIn *in )
void LL8DestroyVectorSequence ( LALStatus *status, INT8VectorSequence **vseq )
void LLU2CreateVectorSequence ( LALStatus *status, UINT2VectorSequence **vseq, CreateVectorSequenceIn *in )
void LLU2DestroyVectorSequence ( LALStatus *status, UINT2VectorSequence **vseq )
void LLU4CreateVectorSequence ( LALStatus *status, UINT4VectorSequence **vseq, CreateVectorSequenceIn *in )
void LLU4DestroyVectorSequence ( LALStatus *status, UINT4VectorSequence **vseq )
void LLU8CreateVectorSequence ( LALStatus *status, UINT8VectorSequence **vseq, CreateVectorSequenceIn *in )
void LLU8DestroyVectorSequence ( LALStatus *status, UINT8VectorSequence **vseq )
void LLCHARCreateVectorSequence ( LALStatus *status, CHARVectorSequence **vseq, CreateVectorSequenceIn *in )
void LLCHARDestroyVectorSequence ( LALStatus *status, CHARVectorSequence **vseq )
void LLCreateVectorSequence ( LALStatus *status, REAL4VectorSequence **vseq, CreateVectorSequenceIn *in )
void LLDestroyVectorSequence ( LALStatus *status, REAL4VectorSequence **vseq )

Description

The CreateVectorSequence family of functions create a (datatype)VectorSequence of the appropriate dimensions.

The DestroyVectorSequence family of functions return the storage allocated by the CreateVectorSequence functions to the system.

Algorithm

Uses

LALMalloc()
LALFree()

Notes

Revision: $Id: VectorSequenceFactories.m4,v 1.4 2001/08/21 04:19:49 jolien Exp$
6.2.2 XLAL Functions

Synopsis

REAL4VectorSequence * XLALCreateVectorSequence(UINT4 length, UINT4 veclen);
void XLALCreateVectorSequence(REAL4VectorSequence *vecseq);

<vecseqtype> * XLALCreate<vecseqtype>(UINT4 length, UINT4 veclen);
void XLALCreate<vecseqtype>(<vecseqtype> *vecseq);

Here <vecseqtype> is one of COMPLEX16VectorSequence, COMPLEX8VectorSequence,
REAL8VectorSequence, REAL4VectorSequence, INT8VectorSequence, INT4VectorSequence,
INT2VectorSequence, UINT8VectorSequence, UINT4VectorSequence, UINT2VectorSequence, or
CHARVectorSequence.

Description

The XLALCreate<type>VectorSequence functions create vector sequences of type <type>, length \verb|length|+, and vector length veclen. The function XLALCreateVectorSequence is the same as XLALCreateREAL4VectorSequence.

The XLALDestroy<type>VectorSequence functions deallocate the memory allocation pointed to by vecseq including its contents. The function XLALDestroyVectorSequence is the same as XLALDestroyREAL4VectorSequence.

Return Values

The create functions return a pointer to the created vector sequence if successful; upon failure they will return NULL and set xlalErrno to one of the following values: XLAL_ENOMEM if memory allocation failed, or XLAL_EBADLEN if the requested length or veclen is zero.

The destroy functions do not have a return value. They can fail if they are passed a NULL pointer, in which case xlalErrno is set to XLAL_EFAULT, or if the vector sequence passed to the destroy routine has zero length, vector length, or NULL data pointer then xlalErrno is set to XLAL_EINVAL.
6.2.3 Program VectorSequenceFactoriesTest.c

A program to test create/destroy vector sequence routines.

Usage

VectorSequenceFactoriesTest [options]
Options:
   -h  print help
   -q  quiet: run silently
   -v  verbose: print extra information
   -d level set lalDebugLevel to level

Description

Exit codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success, normal exit.</td>
</tr>
<tr>
<td>1</td>
<td>Subroutine failed.</td>
</tr>
</tbody>
</table>

Algorithm

Uses

lalDebugLevel
TYPECODECreateVectorSequence()
TYPECODEDestroyVectorSequence()
6.2.4 Program `ArraySequenceFactoriesTest.c`

A program to test create/destroy array sequence routines.

Usage

```
ArraySequenceFactoriesTest [options]
```

Options:

- `-h` print help
- `-q` quiet: run silently
- `-v` verbose: print extra information
- `-d level` set `lalDebugLevel` to `level`

Description

Exit codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success, normal exit.</td>
</tr>
<tr>
<td>1</td>
<td>Subroutine failed.</td>
</tr>
</tbody>
</table>

Algorithm

Uses

- `lalDebugLevel`
- `TYPECODECreateArraySequence()`
- `TYPECODEDestroyArraySequence()`

Notes
Chapter 7

Package *tools*

This package contains the general purpose LAL tools.
7.1 Header Units.h

Provides prototypes for manipulation of units and declares extern constants for the basic and derived SI units.

Synopsis

#include <lal/Units.h>

This header provides prototypes for functions to manipulate the LALUnit structure. It also defines extern constants for a set of predefined units, which are designed to make the structure easier to use. For instance, to determine whether a quantity has units of strain per root hertz, one constructs the unit "strain per root hertz" from the predefined lalStrainUnit and lalHertzUnit constant structures using the LALUnitRaise() and LALUnitMultiply() functions, then compares that to the unit structure in question using the LALUnitCompare() function.

The LALUnit datatype itself is included in the header LALDatatypes.h, and defines a unit in terms of an integer power of ten multiplier along with rational powers of the basic SI units (meters, kilograms, seconds, Amperes, and Kelvins) and two custom units (strain and ADC counts).

Error conditions

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULLPIN</td>
<td>1</td>
<td>&quot;Null pointer to input&quot;</td>
</tr>
<tr>
<td>NULLPOUT</td>
<td>2</td>
<td>&quot;Null pointer to output&quot;</td>
</tr>
<tr>
<td>NULLPD</td>
<td>3</td>
<td>&quot;Null pointer to data member of vector&quot;</td>
</tr>
<tr>
<td>NULLPPARAM</td>
<td>4</td>
<td>&quot;Null pointer to parameters&quot;</td>
</tr>
<tr>
<td>STRINGSIZE</td>
<td>5</td>
<td>&quot;Output string too short&quot;</td>
</tr>
<tr>
<td>OVERFLOW</td>
<td>6</td>
<td>&quot;Exponent outside of (U)INT2 bounds&quot;</td>
</tr>
<tr>
<td>NONINT</td>
<td>7</td>
<td>&quot;Non-integer power of ten&quot;</td>
</tr>
<tr>
<td>PARSE</td>
<td>8</td>
<td>&quot;Error parsing unit string&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants UNITSH_E<name>, and the status descriptions in UNITSH_MSGE<name>. The source code with these messages is in Units.h on line 1.214.

Structures

LALUnitPair

Consists of a pair of unit structures; used as an input structure for the LALUnitCompare() and LALUnitMultiply() functions. The fields are:

LALUnit *unitOne The first unit.

LALUnit *unitTwo The second unit.

RAT4

A four-byte rational number, used as a parameter structure for LALUnitRaise(). The fields are:

INT2 numerator The numerator.

UINT2 denominatorMinusOne One less than the denominator.
## 7.1.1 Module `UnitDefs.c`

Defines basic and derived SI units and a function to produce a text string corresponding to a unit structure.

### Prototypes

```c
void LALUnitAsString( LALStatus *status,
                     CHARVector *output,
                     const LALUnit *input )
```

```c
void LALParseUnitString ( LALStatus *status,
                         LALUnit *output,
                         const CHARVector *input )
```

### Description

`LALUnitAsString()` converts the unit structure `*input` into a text string which is stored in the character vector `*output`. Note that the resulting text string is expressed solely in terms of the basic units (m, kg, s, A, K, strain and counts), and is thus not necessarily the most convenient way to check the units of a quantity. A better method is to construct a unit structure containing the expected units, then compare that to the actual units using `LALUnitCompare()`.

`LALParseUnitString()` reconstructs the original `LALUnit` structure from the string output by `LALUnitAsString()`. It is very sensitive to the exact format of the string and is not intended for use in parsing user-entered strings.

### Algorithm

`LALUnitAsString()` moves through the unit structure, appending the appropriate text to the string as it goes along.

`LALParseUnitString()` moves through the input string, one character at a time, building an `LALUnit` structure as it goes along, so long as it encounters precisely the syntax expected.

### Uses

None.

### Notes

#### Predefined Units

This file also defines a number of `constant` unit structures (declared `extern` in `Units.h`). Zeroth is `lalDimensionlessUnit`, which is simply a `LALUnit` structure to be associated with a unitless quantity.

First, the relevant fundamental SI units and two custom units of use in gravitational wave detection:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Name</th>
<th>Abbr.</th>
<th>Physical Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lalMeterUnit</code></td>
<td>meter</td>
<td>m</td>
<td>length</td>
</tr>
<tr>
<td><code>lalKiloGramUnit</code></td>
<td>kilogram</td>
<td>kg</td>
<td>mass</td>
</tr>
<tr>
<td><code>lalSecondUnit</code></td>
<td>second</td>
<td>s</td>
<td>time</td>
</tr>
<tr>
<td><code>lalAmpereUnit</code></td>
<td>ampere</td>
<td>A</td>
<td>electric current</td>
</tr>
<tr>
<td><code>lalKelvinUnit</code></td>
<td>kelvin</td>
<td>K</td>
<td>thermodynamic temperature</td>
</tr>
<tr>
<td><code>lalStrainUnit</code></td>
<td>strain</td>
<td>ε</td>
<td>gravitational strain</td>
</tr>
<tr>
<td><code>lalADCCountUnit</code></td>
<td>ADC count</td>
<td>count</td>
<td>A-to-D converter counts</td>
</tr>
</tbody>
</table>

Next, the named derived units in the SI:\[1\]:

[1]: SI units reference URL
<table>
<thead>
<tr>
<th>Constant</th>
<th>Name</th>
<th>Abbr.</th>
<th>Physical Quantity</th>
<th>Def.</th>
<th>Fundamental</th>
</tr>
</thead>
<tbody>
<tr>
<td>lalHertzUnit</td>
<td>hertz</td>
<td>Hz</td>
<td>frequency</td>
<td>s⁻¹</td>
<td>s⁻¹</td>
</tr>
<tr>
<td>lalNewtonUnit</td>
<td>newton</td>
<td>N</td>
<td>force</td>
<td>kg·m/s²</td>
<td>m kg s⁻²</td>
</tr>
<tr>
<td>lalPascalUnit</td>
<td>pascal</td>
<td>Pa</td>
<td>pressure</td>
<td>N/m²</td>
<td>m⁻¹ kg s⁻²</td>
</tr>
<tr>
<td>lalJouleUnit</td>
<td>joule</td>
<td>J</td>
<td>energy</td>
<td>N·m</td>
<td>m² kg s⁻²</td>
</tr>
<tr>
<td>lalWattUnit</td>
<td>watt</td>
<td>W</td>
<td>power</td>
<td>J/s</td>
<td>m² kg s⁻³</td>
</tr>
<tr>
<td>lalCoulombUnit</td>
<td>coulomb</td>
<td>C</td>
<td>electric charge</td>
<td>A·s</td>
<td>s A</td>
</tr>
<tr>
<td>lalVoltUnit</td>
<td>volt</td>
<td>V</td>
<td>potential</td>
<td>W/A</td>
<td>m² kg s⁻³ A⁻¹</td>
</tr>
<tr>
<td>lalOhmUnit</td>
<td>ohm</td>
<td>Ω</td>
<td>resistance</td>
<td>V/A</td>
<td>m² kg s⁻³ A⁻²</td>
</tr>
<tr>
<td>lalFaradUnit</td>
<td>farad</td>
<td>F</td>
<td>capacitance</td>
<td>C/V</td>
<td>m⁻² kg⁻¹ s⁴ A²</td>
</tr>
<tr>
<td>lalWeberUnit</td>
<td>weber</td>
<td>Wb</td>
<td>magnetic flux</td>
<td>V·s</td>
<td>m² kg s⁻² A⁻¹</td>
</tr>
<tr>
<td>lalHenryUnit</td>
<td>henry</td>
<td>H</td>
<td>inductance</td>
<td>V·s/A</td>
<td>m² kg s⁻² A⁻²</td>
</tr>
<tr>
<td>lalTeslaUnit</td>
<td>tesla</td>
<td>T</td>
<td>magnetic flux density</td>
<td>Wb/m²</td>
<td>kg s⁻² A⁻¹</td>
</tr>
</tbody>
</table>

The powers of ten (SI prefixes)

<table>
<thead>
<tr>
<th>Constant</th>
<th>Prefix</th>
<th>Abbr.</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>lalYottaUnit</td>
<td>yotta</td>
<td>Y</td>
<td>10²⁴</td>
</tr>
<tr>
<td>lalZettaUnit</td>
<td>zetta</td>
<td>Z</td>
<td>10²¹</td>
</tr>
<tr>
<td>lalExaUnit</td>
<td>exa</td>
<td>E</td>
<td>10¹⁸</td>
</tr>
<tr>
<td>lalPetaUnit</td>
<td>peta</td>
<td>P</td>
<td>10¹⁵</td>
</tr>
<tr>
<td>lalTeraUnit</td>
<td>tera</td>
<td>T</td>
<td>10¹²</td>
</tr>
<tr>
<td>lalGigaUnit</td>
<td>giga</td>
<td>G</td>
<td>10⁹</td>
</tr>
<tr>
<td>lalMegaUnit</td>
<td>mega</td>
<td>M</td>
<td>10⁶</td>
</tr>
<tr>
<td>lalKiloUnit</td>
<td>kilo</td>
<td>k</td>
<td>10³</td>
</tr>
<tr>
<td>lalHectoUnit</td>
<td>hecto</td>
<td>h</td>
<td>10²</td>
</tr>
<tr>
<td>lalDekaUnit</td>
<td>deka</td>
<td>da</td>
<td>10¹</td>
</tr>
<tr>
<td>lalDeciUnit</td>
<td>deci</td>
<td>d</td>
<td>10⁻¹</td>
</tr>
<tr>
<td>lalCentiUnit</td>
<td>centi</td>
<td>c</td>
<td>10⁻²</td>
</tr>
<tr>
<td>lalMilliUnit</td>
<td>milli</td>
<td>m</td>
<td>10⁻³</td>
</tr>
<tr>
<td>lalMicroUnit</td>
<td>micro</td>
<td>µ</td>
<td>10⁻⁶</td>
</tr>
<tr>
<td>lalNanoUnit</td>
<td>nano</td>
<td>n</td>
<td>10⁻⁹</td>
</tr>
<tr>
<td>lalPicoUnit</td>
<td>pico</td>
<td>p</td>
<td>10⁻¹²</td>
</tr>
<tr>
<td>lalFemtoUnit</td>
<td>femto</td>
<td>f</td>
<td>10⁻¹⁵</td>
</tr>
<tr>
<td>lalAttoUnit</td>
<td>atto</td>
<td>a</td>
<td>10⁻¹⁸</td>
</tr>
<tr>
<td>lalZeptoUnit</td>
<td>zepto</td>
<td>z</td>
<td>10⁻²¹</td>
</tr>
<tr>
<td>lalYoctoUnit</td>
<td>yocto</td>
<td>y</td>
<td>10⁻²⁴</td>
</tr>
</tbody>
</table>

And finally a couple of convenient scaled units:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Name</th>
<th>Abbr.</th>
<th>Def.</th>
<th>Fundamental</th>
</tr>
</thead>
<tbody>
<tr>
<td>lalGramUnit</td>
<td>gram</td>
<td>g</td>
<td>10⁻³ kg</td>
<td>10⁻⁵ kg</td>
</tr>
<tr>
<td>lalAttoStrainUnit</td>
<td>attostrain</td>
<td>aε</td>
<td>10⁻¹⁸ε</td>
<td>10⁻¹⁸ε</td>
</tr>
<tr>
<td>lalPicoFaradUnit</td>
<td>picofarad</td>
<td>pF</td>
<td>10⁻¹² F</td>
<td>10⁻¹² m⁻² kg⁻¹ s⁴ A²</td>
</tr>
</tbody>
</table>
7.1.2 Module UnitNormalize.c

Brings an LALUnit structure into standard form by reducing all of the rational exponents into LCD form.

Prototypes

```c
void LALUnitNormalize (LALStatus *status, LALUnit *output, const LALUnit *input)
```

Description

Since the LALUnit structure stores the rational powers of the fundamental units as numerator and denominator, it is possible to represent the same units in different ways, e.g., m^2 versus m^{4/2}. This function reduces all of those fractions to convert the structure to its simplest form.

Algorithm

The rational powers are reduced using Euclid’s algorithm[2].

Uses

None.

Notes

Note that the functions LALUnitRaise(), LALUnitMultiply(), and LALUnitCompare() all call LALUnitNormalize() themselves, so there is usually no need to call it explicitly.
7.1.3 Module UnitRaise.c

Raises an LALUnit structure to a specified rational power.

Prototypes

void LALUnitRaise (LALStatus *status, LALUnit *output, const LALUnit *input, const RAT4 *power)

Description

This function raises the LALUnit structure *input to the rational power *power. In this way, units such as $s^{1/2}$ and $m^{-1}$ can be created using existing units.

Algorithm

The function first multiplies the overall power of ten $input->powerOfTen$ by the rational number *power, checking to make sure that the resulting power is still an integer. It then multiplies each of the rational powers in *input by *power by naïve multiplication of rational numbers

$$\left(\frac{N_1}{1+D_1}\right) \left(\frac{N_2}{1+D_2}\right) = \frac{N_1N_2}{1+(1+D_1)(1+D_2)-1}$$

and then calls LALUnitNormalize() to bring the result into standard form.

Uses

LALUnitNormalize()

Notes
7.1.4 Module **UnitMultiply.c**

Multiplies two LALUnit structures.

Prototypes

```c
void LALUnitMultiply (LALStatus *status, LALUnit *output, const LALUnitPair *input)
```

This function multiplies together the LALUnit structures *(input->unitOne)* and *(input->unitTwo)*, thus allowing a module to *e.g.*, multiply two REAL8TimeSeries and give the resulting REAL8TimeSeries the correct units.

Algorithm

The function first adds together the overall powers of ten in the two input unit structures, then adds each of the corresponding rational powers in *(input->unitOne)* and *(input->unitTwo)* by naïve addition of rational numbers

\[
\frac{N_1}{1 + D_1} + \frac{N_2}{1 + D_2} = \frac{N_1(1 + D_2) + N_2(1 + D_1)}{1 + (1 + D_1)(1 + D_2) - 1}
\]

and then calls `LALUnitNormalize()` to bring the result into standard form.

Uses

`LALUnitNormalize()`

Notes
7.1.5 Module UnitCompare.c

Compares two LALUnit structures, returning true if they are equivalent, false otherwise.

Prototypes

```c
void LALUnitCompare (LALStatus *status, BOOLEAN *output, const LALUnitPair *input)
```

Description

This function determines whether the units represented by *(input->unitOne) and *(input->unitTwo) are the same (both dimensionally and in the power-of-ten prefactor). In this way, programs and programmers can verify that quantities have the expected units.

Algorithm

The function first uses LALUnitNormalize() to bring both unit structures into standard form, then compares the powers of ten and the numerator and denominator of each exponent of a fundamental unit in turn.

Uses

LALUnitNormalize()

Notes
### 7.1.6 XLAL Functions

**Synopsis**

```c
#include <lal/Units.h>

char * XLALUnitAsString( char *string, UINT4 length, const LALUnit *input );
LALUnit * XLALParseUnitString( LALUnit *output, const char *string );
int XLALUnitNormalize( LALUnit *unit );
int XLALUnitCompare( const LALUnit *unit1, const LALUnit *unit2 );
LALUnit * XLALUnitMultiply( LALUnit *output, const LALUnit *unit1,
                        const LALUnit *unit2 );
LALUnit * XLALUnitRaiseRAT4( LALUnit *output, const LALUnit *input,
                        const RAT4 *power );
LALUnit * XLALUnitRaiseINT2( LALUnit *output, const LALUnit *input,
                        INT2 power );
LALUnit * XLALUnitSquare( LALUnit *output, const LALUnit *input );
LALUnit * XLALUnitSqrt( LALUnit *output, const LALUnit *input );
```

**Description**

- **XLALUnitAsString** converts a LALUnit structure into a character string of maximum length length (including NUL termination) representation of the units. The inverse function, **XLALParseUnitString** parses a character string to produce a LALUnit structure; if output is NULL, memory for the output is allocated. If the input string is NULL or is empty then the output units are dimensionless: lalDimensionlessUnit.
- **XLALUnitNormalize** puts a LALUnit structure into normal form by simplifying all unit exponent fractions to their simplest form.
- **XLALUnitCompare** compares two LALUnit structures: they are the same if their normal forms are identical.
- **XLALUnitMultiply** multiplies two LALUnit structures. The result is put into normal form.
- **XLALUnitRaiseRAT4** raises a LALUnit structure to a rational power given by the RAT4 structure power.
- **XLALUnitRaiseINT2** raises a LALUnit structure to an integer power power.
- **XLALUnitSquare** produces the square of a LALUnit structure.
- **XLALUnitSqrt** produces the square-root of a LALUnit structure.

**Return Values**

- **XLALUnitAsString** returns the pointer to the input string, which is populated with the unit string if successful. If there is a failure, **XLALUnitAsString** returns a NULL pointer and xlalErrno is set to one of the following values: XLAL_EFAULT if one of the input pointers is NULL or XLAL_EBADLEN if the length of the string is insufficient for the unit string.
- **XLALParseUnitString** returns the pointer output upon return or a pointer to newly allocated memory if output was NULL; on failure, **XLALParseUnitString** returns NULL and sets xlalErrno to one of the following values: XLAL_ENOMEM if the routine was unable to allocate memory for the output or XLAL_EFAILED if the routine was unable to parse the unit string.
- **XLALUnitNormalize** returns 0 upon success or XLAL_FAILURE if the input pointer is NULL, in which case xlalErrno is set to XLAL_EFAULT
- **XLALUnitCompare** returns 1 if the the normal form of the two unit structures are the same or 0 if they are different. It returns XLAL_FAILURE and xlalErrno is set to XLAL_EFAULT if one of the input pointers is NULL.
- **XLALUnitMultiply** and **XLALUnitRaiseRAT4** and **XLALUnitRaiseINT2** and **XLALUnitSquare** and **XLALUnitSqrt** all return a pointer to the output unit structure output upon success or NULL upon failure. If there is a failure, xlalErrno is set to one of the following values: XLAL_EFAULT if one of the input pointers is NULL, XLAL_ERANGE if one of the unit powers exceeds the allowed range, or XLAL EINVAL (for the raise functions only) if the unit power would not be an integer.
7.1.7 Program UnitsTest.c

Test Suite for unit manipulation programs

Usage

\verbatim
UnitsTest [options]
\end{verbatim}

Options:
  -h       print help
  -q       quiet: run silently
  -v       verbose: print extra information
  -d level set lalDebugLevel to level

Description

This program tests the various units-manipulation routines, as well as the pre-defined units defined in Units.h. For each successful test, it prints “PASS” to standard output.

Exit codes

<table>
<thead>
<tr>
<th>name</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOM</td>
<td>0</td>
<td>&quot;Nominal exit&quot;</td>
</tr>
<tr>
<td>CHK</td>
<td>1</td>
<td>&quot;Error checking failed to catch bad data&quot;</td>
</tr>
<tr>
<td>FLS</td>
<td>2</td>
<td>&quot;Incorrect answer for valid data&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants UNITSTESTC_\<name\>, and the status descriptions in UNITSTESTC_MSGE\<name\>. The source code with these messages is in UnitsTest.c on line 1.114.

Uses

LALCHARCreateVector()
LALUnitAsString()
LALParseUnitString()
LALCHARDestroyVector()
LALUnitMultiply()
LALUnitRaise()
LALUnitNormalize()
LALUnitCompare()

Notes
7.2 Header DetectorSite.h

This is a dummy header which includes LALDetectors.h, for backwards compatibility.

/* Legacy code: should now use data in LALDetectors.h */
enum
{
    LALDetectorIndexLHODIFF = LAL_LHO_4K_DETECTOR,
    LALDetectorIndexLLODIFF = LAL_LLO_4K_DETECTOR,
    LALDetectorIndexVIRGODIFF = LAL_VIRGO_DETECTOR,
    LALDetectorIndexGEO600DIFF = LAL_GEO_600_DETECTOR,
    LALDetectorIndexTAMA300DIFF = LAL_TAMA_300_DETECTOR,
    LALDetectorIndexCIT40DIFF = LAL_CIT_40_DETECTOR,
    LALNumCachedDetectors = LAL_NUM_DETECTORS
};
7.3 Header LALDetectors.h

This header defines structures to hold the basic data describing a gravitational wave detector.

Synopsis

```c
#include <lal/LALDetectors.h>
```

According to the common frame format specification [3] the geometry of an interferometric detector will be stored in a `FrDetector` structure, specifying the location of the detector vertex and the orientation of its arms in geodetic coordinates suited to geographical surveying. Resonant bars and other sorts of detectors, if they write their data to frames, are expected to fill this structure with their location and orientation in some way suited to the detector type.

For most data analysis tasks, however, any gravitational wave detector can be described by its location in an Earth-fixed rotating reference frame, as well as a response tensor $d^{ab}$, constant in the same frame, which defines the “strain” $h$ measured by the detector in terms of the metric perturbation $h_{ab}$ as

$$h = h_{ab} d^{ab}.$$  \hfill (7.1)

This header defines a `LALFrDetector` structure which contains essentially the same information as the `FrDetector` structure, as well as a `LALDetector` structure which contains the Cartesian coordinates of the detector along with the components of the response tensor $d^{ab}$ in the same coordinate system.

The Geodetic Coordinate System

Geodetic coordinates are spheroidal coordinates based on the WGS-84 Earth Model, which is an oblate spheroid with equatorial radius $a = 6.378137 \times 10^6$ m and polar radius $b = 6.356752314 \times 10^6$ m. Any point in space can be located according to its longitude, latitude, and elevation. The longitude $\lambda$ is the angle between the half-plane bounded by the symmetry axis of the reference ellipsoid containing the point in question and the half-plane plane containing the Prime Meridian; it is measured in radians, increases to the East, and ranges from $-\pi$ to $\pi$. The latitude $\beta$ is the angle between the ray which is normal to the ellipsoid and passes through the point in question and the equatorial plane; it is measured in radians, increases to the North, and ranges from $-\pi/2$ to $\pi/2$. The elevation $h$ is the signed distance along this ray from the reference ellipsoid to the point in question. This coordinate system is described in more detail in [4].

Altitude and Azimuth Angles

The `LALFrDetector` structure stores the directions along the two arms of an interferometer in an altitude/azimuth representation with respect to the local tangent plane to the reference ellipsoid, known as the local horizontal. The altitude $A$ is the angle the direction vector makes with the horizontal, $A > 0$ meaning above horizontal, $A < 0$ below. The azimuth angle $\zeta$ is found by projecting the direction onto the local horizontal plane, then measuring the angle clockwise from North to this projected direction.

The Cartesian Coordinate System

The position vector and response tensor contained in the `LALDetector` structure are defined in a simple orthonormal coordinate system with its origin at the center of the earth, an $x^1$ axis which pierces the Earth’s surface at the intersection of the equator and the prime meridian, an $x^2$ axis which pierces the earth’s surface at $\pi/2$ radians East longitude on the equator, and an $x^3$ axis which pierces the Earth’s surface at the North Pole. The coordinates $x^1$, $x^2$, $x^3$ correspond to the Earth-fixed coordinates $X_E$, $Y_E$, $Z_E$ defined in [4], respectively.

The relationship between geodetic and Cartesian coordinates is given by

$$x^1 = \left( \frac{a^2}{\sqrt{a^2 \cos^2 \beta + b^2 \sin^2 \beta}} + h \right) \cos \beta \cos \lambda,$$ \hfill (7.2)

$$x^2 = \left( \frac{a^2}{\sqrt{a^2 \cos^2 \beta + b^2 \sin^2 \beta}} + h \right) \cos \beta \sin \lambda,$$ \hfill (7.3)

$$x^3 = \left( \frac{b^2}{\sqrt{a^2 \cos^2 \beta + b^2 \sin^2 \beta}} + h \right) \sin \beta.$$ \hfill (7.4)
Error conditions

<table>
<thead>
<tr>
<th>name</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULLP</td>
<td>1</td>
<td>&quot;Null pointer&quot;</td>
</tr>
<tr>
<td>TYPE</td>
<td>2</td>
<td>&quot;Unsupported detector type&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants LALDETECTORS\_E<name>, and the status descriptions in LALDETECTORS\_MSGE<name>. The source code with these messages is in LALDetectors.h on line 1.234.

The LALDetectorType enumeration

Since data from bars as well as interferometers can be written to frames, we need an additional piece of information to interpret the site geometry data specified in the LALFrDetector structure; for instance, is the x arm really the x arm or is it the long axis of a bar? The LALDetectorType enumeration provides a way to keep track of that.

The possible values are (each value is prefaced by LALDETECTORTYPE.):

- LALDETECTORTYPE\_ABSENT No FrDetector associated with the structure
- LALDETECTORTYPE\_IFODIFF Interferometer in differential mode
- LALDETECTORTYPE\_IFOXARM Interferometer in one-armed mode (X arm)
- LALDETECTORTYPE\_IFOYARM Interferometer in one-armed mode (Y arm)
- LALDETECTORTYPE\_IFOCOMM Interferometer in common mode
- LALDETECTORTYPE\_CYLBAR Cylindrical bar

Cached Detectors

In practice, we will often be working with fixed unchanging site geometry, e.g., for the LIGO interferometers; to avoid constantly reconstructing the corresponding LALDetectors, we should define some constant LALDetectors describing them. Those are stored in a constant array of LALDetector structures known as lalCachedDetectors, which is declared extern in this header and defined in CreateDetector.c (see Sec. 7.3.1).

The LALCreateDetector() routine will first look through the lalCachedDetectors array for a LALDetector structure with matching type and frDetector.name fields; if it finds one, it returns a copy of that; if not, it creates one.

The header LALDetectors.h also defines an enumeration of the indices of the known detectors:

For example, the LALDetector representing LIGO Hanford in differential mode is lalCachedDetectors[LALDetectorIndexLHODIFF].

Structures

Structure LALFrDetector

The LALFrDetector structure holds site geometry information in the same format as the FrDetector structure defined in the frames spec. The fields are:

- CHAR name[LALNameLength] A unique identifying string.
- REAL8 vertexLongitudeRadians The geodetic longitude $\lambda$ of the vertex, in radians.
- REAL8 vertexLatitudeRadians The geodetic latitude $\beta$ of the vertex, in radians.
- REAL4 vertexElevation The height of the vertex above the reference ellipsoid, in meters.
- REAL4 xArmAltitudeRadians The angle $A_X$ up from the local tangent plane of the reference ellipsoid to the X arm, in radians.
- REAL4 xArmAzimuthRadians The angle $\zeta_X$ clockwise from North to the projection of the X arm into the local tangent plane of the reference ellipsoid, in radians.
REAL4 yArmAltitudeRadians  The angle $\mathcal{A}_Y$ up from the local tangent plane of the reference ellipsoid to the $Y$ arm, in radians.

REAL4 yArmAzimuthRadians  The angle $\zeta_Y$ clockwise from North to the projection of the $Y$ arm into the local tangent plane of the reference ellipsoid, in radians.

Structure LALDetector

The LALDetector structure is intended to be the way that detector geometry information is passed to LAL routines. This structure describes a detector geometry in a way independent of the type of detector. The fields are:

REAL8 location[3]  The three components, in an Earth-fixed Cartesian coordinate system, of the position vector from the center of the Earth to the detector, in meters.

REAL4 response[3][3]  The Earth-fixed Cartesian components of the detector’s response tensor $d_{ab}$.

LALDetectorType type  The type of detector (e.g., IFO in differential mode, cylindrical bar, etc.)

LALFrDetector frDetector  The original LALFrDetector structure from which this was created.
7.3.1 Module CreateDetector.c

Creates a LALDetector structure from a LALFrDetector structure and the type of detector.

Prototypes

```c
void LALCreateDetector( LALStatus *status, LALDetector *output, const LALFrDetector *input, const LALDetectorType type )
```

Description

This routine takes the site geometry described in the LALFrDetector structure, along with a LALDetectorType parameter, and constructs the Cartesian detector location and response tensor needed to fill the LALDetector output.

The detector type is needed because different types of detectors have different response tensors. In each case the response tensor is determined by the unit vectors \( \hat{u}_X \) and \( \hat{u}_Y \) which are constant in an Earth-fixed rotating reference frame and point in the “X arm” and “Y arm” directions, respectively; the headings of these directions in a local frame at the detector are stored in the LALFrDetector structure.

The detector types recognized are (all names are prefaced by LALDETECTORTYPE):

- **IFODIFF** An interferometer in differential mode. The response tensor is given by \( d^{ab} = \frac{1}{4}(u_X^a u_X^b - u_Y^a u_Y^b) \). Note that this is the preferred form even in the two arms of the detector are not perpendicular (e.g., at the GEO600 site).
- **IFOXARM** An interferometer in one-armed mode with the X arm active. The response tensor is given by \( d^{ab} = \frac{1}{2} u_X^a u_X^b \).
- **IFOYARM** An interferometer in one-armed mode with the Y arm active. The response tensor is given by \( d^{ab} = \frac{1}{2} u_Y^a u_Y^b \).
- **IFOCOMM** An interferometer in common mode. The response tensor is given by \( d^{ab} = \frac{1}{2}(u_X^a u_X^b + u_Y^a u_Y^b) \).
- **CYLBAR** A cylindrical bar detector. In this case the “X arm” is actually the symmetry axis of the bar, and the “Y arm” is ignored. The response tensor is \( d^{ab} = u_X^a u_X^b \).

In each of these cases, the basic transformation needed is to express a unit vector \( \hat{u} \) in terms of its components in the Earth-fixed basis \( \{\hat{e}_1, \hat{e}_2, \hat{e}_3\} \). The altitude angle \( A \) and azimuth angle \( \zeta \) allow us to express the unit vector \( \hat{u} \) corresponding to a direction in terms of an orthonormal basis consisting of a vector \( \hat{e}_E \) pointing due East within the local horizontal, a vector \( \hat{e}_N \) pointing due North within the local horizontal, and an upward-pointing vector \( \hat{e}_U \) normal to the local horizontal plane.

The relationship is

\[
\hat{u} = (\hat{e}_E \sin \zeta + \hat{e}_N \cos \zeta) \cos A + \hat{e}_U \sin A .
\]

(7.5)

Since the local horizontal is defined as the tangent plane to the reference ellipsoid at the point with the detector’s latitude \( \beta \) and longitude \( \lambda \), the local basis is related to the orthonormal basis \( \{\hat{e}_\rho, \hat{e}_\lambda, \hat{e}_z\} \) of a cylindrical coordinate system (related to the Earth-fixed Cartesian coordinates by \( x^1 = \rho \cos \lambda, x^2 = \rho \sin \lambda, x^3 = z \), so that \( \hat{e}_\rho \) points away from the Earth’s axis, \( \hat{e}_\lambda \) points in the direction of increasing longitude, and \( \hat{e}_z \) points in the direction of increasing \( x^3 \) by

\[
\hat{e}_E = \hat{e}_\lambda \\
\hat{e}_N = -\hat{e}_\rho \sin \beta + \hat{e}_z \cos \beta \\
\hat{e}_U = \hat{e}_\rho \cos \beta + \hat{e}_z \sin \beta
\]

(7.6) (7.7) (7.8)

It is then straightforward to relate the cylindrical basis vectors to those in the Earth-fixed Cartesian system by

\[
\hat{e}_\rho = \hat{e}_1 \cos \lambda + \hat{e}_2 \sin \lambda \\
\hat{e}_\lambda = -\hat{e}_1 \sin \lambda + \hat{e}_2 \cos \lambda \\
\hat{e}_z = \hat{e}_3
\]

(7.9) (7.10) (7.11)

1. These form a right-handed basis, providing an answer to the age-old question “What’s Up?”: “East cross North.”
To express \( \hat{u} \) in the Cartesian basis, we need \( \hat{u} \cdot \hat{e}_1 \), \( \hat{u} \cdot \hat{e}_2 \), and \( \hat{u} \cdot \hat{e}_3 \). We first observe that
\[
\hat{u} \cdot \hat{e}_E = \cos A \sin \zeta \tag{7.12}
\]
\[
\hat{u} \cdot \hat{e}_N = \cos A \cos \zeta \tag{7.13}
\]
\[
\hat{u} \cdot \hat{e}_U = \sin A \tag{7.14}
\]
then that
\[
\hat{u} \cdot \hat{e}_\rho = (\hat{u} \cdot \hat{e}_N)(\hat{e}_N \cdot \hat{e}_\rho) + (\hat{u} \cdot \hat{e}_U)(\hat{e}_U \cdot \hat{e}_\rho) = -(\hat{u} \cdot \hat{e}_N) \sin \beta + (\hat{u} \cdot \hat{e}_U) \cos \beta \tag{7.15}
\]
\[
\hat{u} \cdot \hat{e}_\lambda = \hat{u} \cdot \hat{e}_E \tag{7.16}
\]
\[
\hat{u} \cdot \hat{e}_z = (\hat{u} \cdot \hat{e}_N)(\hat{e}_N \cdot \hat{e}_z) + (\hat{u} \cdot \hat{e}_U)(\hat{e}_U \cdot \hat{e}_z) = (\hat{u} \cdot \hat{e}_N) \cos \beta + (\hat{u} \cdot \hat{e}_U) \sin \beta \tag{7.17}
\]
and finally that
\[
\hat{u} \cdot \hat{e}_1 = (\hat{u} \cdot \hat{e}_\rho)(\hat{e}_\rho \cdot \hat{e}_1) + (\hat{u} \cdot \hat{e}_\lambda)(\hat{e}_\lambda \cdot \hat{e}_1) = (\hat{u} \cdot \hat{e}_\rho) \cos \lambda - (\hat{u} \cdot \hat{e}_\lambda) \sin \lambda \tag{7.18}
\]
\[
\hat{u} \cdot \hat{e}_2 = (\hat{u} \cdot \hat{e}_\rho)(\hat{e}_\rho \cdot \hat{e}_2) + (\hat{u} \cdot \hat{e}_\lambda)(\hat{e}_\lambda \cdot \hat{e}_2) = (\hat{u} \cdot \hat{e}_\rho) \sin \lambda + (\hat{u} \cdot \hat{e}_\lambda) \cos \lambda \tag{7.19}
\]
\[
\hat{u} \cdot \hat{e}_3 = \hat{u} \cdot \hat{e}_z \tag{7.20}
\]

**Cached Detectors**

To avoid repeatedly calculating the Cartesian coordinates and response tensor of known detectors, the constant array `lalCachedDetectors[]` contains the site geometry and response tensors of the most commonly used detectors. These are defined in this file and listed in Table 7.1.

**Algorithm**

`LALCreateDetector()` first checks the `lalCachedDetectors[]` array to see if the specified type and the name in the input `LALFrDetector` match any of the predefined constant detectors. If so, it returns a copy of the constant detector (not just a pointer to the constant).

If not, it calculates the Cartesian coordinates \( \{x^1, x^2, x^3\} \) of the detector location defined by (7.2–7.4); in particular, it calculates the denominator \( \sqrt{a^2 \cos^2 \beta + b^2 \sin^2 \beta} \) and the distance from the axis
\[
\rho = \left( \frac{a^2}{\sqrt{a^2 \cos^2 \beta + b^2 \sin^2 \beta}} + h \right) \cos \beta \tag{7.21}
\]
as intermediate steps.

It then calculates the Cartesian components of the unit vectors \( \hat{u}_X \) and \( \hat{u}_Y \) in the arm directions from the altitude and azimuth angles by use of a `static` function which implements (7.12–7.20). (Depending on the detector type specified, only the unit vector(s) which are actually needed are calculated.) Using this components it constructs \( d^{ab} \) according to the formula appropriate to the detector type.

The calculation of \( x^a \) is done to double precision, that of \( d^{ab} \) to single precision.

**Uses**

`LALCreateDetector()`

**Notes**

- The conventions in the `LALFrDetector` structure are an extension of those used in version 4 of the frame specification [3] and do not agree with the conventions used by LDAS, which are an anticipation of the yet-to-be-released version 5 frame spec. The `LALFrDetector` structure will change once a final version 5 frame spec is available.

- If the location and response tensor information for a `LALDetector` are filled in by hand (e.g., for testing purposes), the `type` field should be set to `LALDETECTORTYPE_ABSENT`.

- The range of `LALDetectorTypes` could be expanded to include the monopole and five quadrupole modes for a spherical resonant detector [6, 7, 8, 9].
Table 7.1: Predefined gravitational wave detectors, contained in the `lalCachedDetectors[]` array. The LIGO site data come directly from [4], including the Cartesian position vectors $x^a$ and the response tensor $d^{ab}$, which was derived from the quoted components of the detector frame basis vectors $\hat{u}_X$ and $\hat{u}_Y$. The data on the other detectors comes from [5].
• At the moment, this code still writes some diagnostics to standard output. These are supposed to be removed once it’s been tested.
7.3.2 Program DetectorSiteTest.c

Tests the detector response and site parameter structures and the routine to create one from the other.

Usage

\verbatim
DetectorSiteTest [options]
Options:
  -h print help
  -q quiet: run silently
  -v verbose: print extra information
  -d level set lalDebugLevel to level
\endverbatim

Description

Right now the test routine does very little. It contains a static function \texttt{PrintLALDetector()} which will print the fields of a \texttt{LALDetector} to standard output in the same format that would be used for a C initialization. This function is not currently called. It also contains a static function \texttt{CheckDetector()} which extracts the \texttt{LALFrDetector} and type from a \texttt{LALDetector}, changes the name of the \texttt{LALFrDetector} (in case it’s one of the predefined constant detectors), constructs a new \texttt{LALDetector} and compares the values of the fields of the old and new structures. The program currently performs this check for the two LIGO sites.

Exit codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOM</td>
<td>0</td>
<td>&quot;Nominal exit&quot;</td>
</tr>
<tr>
<td>CHK</td>
<td>1</td>
<td>&quot;Error checking failed to catch bad data&quot;</td>
</tr>
<tr>
<td>FLS</td>
<td>2</td>
<td>&quot;Incorrect answer for valid data&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants \texttt{DETECTORSITETESTC_E<name>}, and the status descriptions in \texttt{DETECTORSITETESTC_MSGE<name>}. The source code with these messages is in \texttt{DetectorSiteTest.c} on line 1.120.

Uses

\texttt{LALCreateDetector()}

Notes
7.4 Header `DetResponse.h`

Provides routines to compute gravitational wave detector response to polarized planar gravitational wave originating from a given source, detected at a given time.

7.4.1 Synopsis

```c
#include <lal/DetResponse.h>
```

Error conditions

<table>
<thead>
<tr>
<th>name</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULLINPUT</td>
<td>1</td>
<td>&quot;Input is NULL&quot;</td>
</tr>
<tr>
<td>NULLOUTPUT</td>
<td>2</td>
<td>&quot;Output is NULL&quot;</td>
</tr>
<tr>
<td>SRCNOTEQUATORIAL</td>
<td>3</td>
<td>&quot;Source coordinates not in Equatorial system&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants `DETRESPONSEH_E<name>`, and the status descriptions in `DETRESPONSEH_MSGE<name>`. The source code with these messages is in `DetResponse.h` on line 1.83.

Types and Structures

**Structure LALSource**

This structure contains gravitational wave source position (in Equatorial coordinates), and orientation angle. The orientation is measured counter-clockwise with respect to the “line of ascending nodes”, i.e. counter-clockwise with respect to a line perpendicular to the source’s meridian and extending westwards. For a source in the Northern celestial hemisphere, and an observer in the Northern hemisphere standing such that they are facing South, this angle is measured counter-clockwise from a 3 o’clock position (pointing West) at the source. The polarization convention is chosen such that if the source orientation were zero, the source would be a pure +-polarized source.

The fields are:

- **CHAR *name** Name of source
- **SkyPosition equatorialCoords** Equatorial coordinates of source
- **REAL8 orientation** Orientation angle (ψ) of source: counter-clockwise angle x-axis makes with a line perpendicular to meridian of source in Westward direction (i.e. North of West), in decimal radians.

**Structure LALDetAndSource**

This structure aggregates a pointer to a `LALDetector` and a `LALSource`. Its sole function is to allow the user to pass detector and source parameters to the functions `LALComputeDetAMResponse()` and `LALComputeDetAMResponseSeries()`.

The fields are:

- **LALDetector *pDetector** Pointer to `LALDetector` object containing information about the detector
- **LALSource *pSource** Pointer to `LALSource` object containing information about the source
Structure **LALDetAMResponse**

This structure encapsulates the detector AM (beam pattern) coefficients for one source at one instance in time. The fields are:

- **REAL4 plus** Detector response to +-polarized gravitational radiation
- **REAL4 cross** Detector response to \(\times\)-polarized gravitational radiation
- **REAL4 scalar** Detector response to scalar gravitational radiation (NB: ignored at present – scalar response computation is not yet implemented)

Structure **LALDetAMResponseSeries**

This structure aggregates together three **REAL4Vectors** containing time series of detector AM response. Since these quantities are dimensionless, they cannot be accurately stored in a **TimeSeries** structure. However, **REAL4Vectors** may be conveniently converted to **TimeSeries** format.

- **REAL4TimeSeries *pPlus** Pointer to a **REAL4TimeSeries** containing detector response to +-polarized gravitational radiation over a span of time
- **REAL4TimeSeries *pCross** Pointer to a **REAL4TimeSeries** containing detector response to \(\times\)-polarized gravitational radiation over a span of time
- **REAL4TimeSeries *pScalar** Pointer to a **REAL4TimeSeries** containing detector response to scalar gravitational radiation over a span of time. (NB: This is unused for the moment. Response to scalar gravitational radiation is not yet implemented.)

Structure **LALGPSandAcc**

This structure aggregates GPS time and leap second accuracy requirement for converting GPS time to sidereal time (implicitly used in **LALComputeDetAMResponse()**).

- **LIGOTimeGPS gps** The GPS time
- **LALLeapSecAccuracy accuracy** The required accuracy for handling leap seconds

Structure **LALTimeIntervalAndNSample**

This structure encapsulates time and sampling information for computing a **LALDetAMResponseSeries**. Its fields correspond to some fields of the **TimeSeries** structures for easy conversion.

- **LIGOTimeGPS epoch** The start time \(t_0\) of the time series
- **REAL8 deltaT** The sampling interval \(\Delta t\), in seconds
- **UINT4 nSample** The total number of samples to be computed
- **LALLeapSecAccuracy accuracy** The required accuracy for handling leap seconds
7.4.2 Module *DetResponse.c*

Computes the response of a detector.

**Prototypes**

```c
void XLALComputeDetAMResponse(
    double *fplus, /**< Returned value of F+ */
    double *fcross, /**< Returned value of Fx */
    REAL4 D[3][3], /**< Detector response 3x3 matrix */
    const double ra, /**< Right ascension of source (radians) */
    const double dec, /**< Declination of source (radians) */
    const double psi, /**< Polarization angle of source (radians) */
    const double gmst /**< Greenwich mean sidereal time (radians) */
)
```

```c
void LALComputeDetAMResponse(LALStatus * status, LALDetAMResponse * pResponse, const LALDetAndSource * pDetAndSrc, const LALGPSandAcc * pGPSandAcc)
```

```c
void LALComputeDetAMResponseSeries(LALStatus * status, LALDetAMResponseSeries * pResponseSeries, const LALDetAndSource * pDetAndSource, const LALTimeIntervalAndNSample * pTimeInfo)
```

**Description**

These routines compute the antenna beam pattern for all supported detector types. *XLALComputeDetAMResponse()* computes the response at one instance in time, and *XLALComputeDetAMResponseSeries()* computes a vector of response for some length of time.

**Algorithm**

This code is a translation of the algorithm in the Maple worksheet by Anderson, *et al.* [5]. We compute the $h$-tensors for $+$ and $\times$-polarized in the Earth-fixed frame, and then contract them (take the scalar product) with the detector response tensors as described in the *DetectorSite.h* section of the *tools* package.

*DetectorSite.h* in the *tools* package provides predefined *LALDetector* structures representing most current detectors, including LIGO (Hanford and Livingston), and GEO.

**Uses**

*LALGPStoGMST1()*

**Notes**

For examples of usage, please see the test programs in the *test* directory.
7.5 Header Calibration.h

Synopsis

```
#include <lal/Calibration.h>
```

Error conditions

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>001</td>
<td>&quot;Null pointer&quot;</td>
</tr>
<tr>
<td>SIZE</td>
<td>002</td>
<td>&quot;Invalid size&quot;</td>
</tr>
<tr>
<td>SZMM</td>
<td>004</td>
<td>&quot;Size mismatch&quot;</td>
</tr>
<tr>
<td>ZERO</td>
<td>010</td>
<td>&quot;Zero factor&quot;</td>
</tr>
<tr>
<td>TIME</td>
<td>020</td>
<td>&quot;Time out of range&quot;</td>
</tr>
<tr>
<td>UNIT</td>
<td>040</td>
<td>&quot;Incompatible units&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants CALIBRATIONH_E<name>, and the status descriptions in CALIBRATIONH_MSGE<name>. The source code with these messages is in Calibration.h on line 1.68.

Structures

**Type CalibrationType**

```
typedef enum
{
    CalibrationAmplitude = 001,
    CalibrationOffset = 002,
    CalibrationDelay = 004,
    CalibrationTransfer = 010,
    CalibrationZPG = 020
} CalibrationType;
```

**Type CalFactors**

```
typedef struct tagCalFactors
{
    COMPLEX16 alpha;
    COMPLEX16 alphabeta;
    COMPLEX16 beta;
    COMPLEX16 exc;
    COMPLEX16 asq;
    COMPLEX16 darm;
} CalFactors;
```

**Type UpdateFactorsParams**

```
typedef struct tagUpdateFactorsParams
{
    REAL8 lineFrequency;
    COMPLEX16 openloop;
```
```c
typedef struct
tagCalibrationRecord
{
    CHAR          name[LALNameLength];
    LIGOTimeGPS   epoch;
    REAL8         duration;
    CHAR          reference[LALNameLength];
    LALUnit       units;
    UINT4         type;
    REAL8         conversion;
    REAL8         offset;
    REAL8         delay;
    COMPLEX8FrequencySeries *transfer;
    REAL8Vector   *zeros;
    REAL8Vector   *poles;
    REAL8         gain;
}
CalibrationRecord;
```

The type `CalibrationFunctions` contains two calibration functions, the sensing function $C(f)$ and the response function $R(f)$. While the response function is the function that is most often wanted, the sensing function is needed in updating calibration from one epoch to another.

```c
typedef struct
tagCalibrationUpdateParams
{
    LIGOTimeGPS  epoch;
    LIGOTimeGPS  duration;
    CHAR         *ifo;
    COMPLEX8    alpha;
    COMPLEX8    alphabeta;
    COMPLEX8TimeSeries *openLoopFactor;
    COMPLEX8TimeSeries *sensingFactor;
}
CalibrationUpdateParams;
```
The type `CalibrationUpdateParams` contains two time series representing an overall gain factor for the open-loop gain function $H(f)$ and the sensing function $C(f)$. These transfer functions are known to change (by an overall factor) with time, and these two factors can be tracked using the injected calibration lines. The factors are stored in this structure as (very-slowly varying) time series, and are to be used in updating the calibration functions described previously. (The response function can be computed from the open-loop gain and the sensing function. It is simply $R(f) = [1 + H(f)]/C(f)$.) In addition, this structure contains the present epoch and the duration of the data to be calibrated to identify the particular set of factors (from those recorded in the time series) to use.
### 7.5.1 Module `ComputeTransfer.c`

Computes the transfer function from zero-pole-gain representation.

#### Prototypes

```c
void LALComputeTransfer( LALStatus *stat, CalibrationRecord *calrec )

void LALUpdateCalibration( LALStatus *status, CalibrationFunctions *output, CalibrationFunctions *input, CalibrationUpdateParams *params )

void LALResponseConvert( LALStatus *status, COMPLEX8FrequencySeries *output, COMPLEX8FrequencySeries *input )

INT4 XLALResponseConvert( COMPLEX8FrequencySeries *output, COMPLEX8FrequencySeries *input )
```

#### Description

A transfer function can either be specified as a list of coefficients or a list of poles and zeros. The function `LALComputeTransfer()` computes the frequency representation of the transfer function `calrec->transfer` described by the zeroes, poles and gain in `*calrec`. The memory for the frequency series should be allocated before calling this routine which uses `calrec->transfer->deltaF` and `calrec->transfer->data->npoints`.

The routine `LALUpdateCalibration()` updates the response function and the sensing function from some reference functions to the current functions using information about the calibration lines. The two calibration lines yield two constants (as a slowly-varying function of time) that are used as coefficients to the reference response and sensing functions to compute the current response and sensing functions. These coefficients are stored in time series in the parameter structure, along with the current epoch and duration for which the calibration functions are to be computed. If the duration is zero, the calibration factors are the first ones at or after the given epoch. If the duration is non-zero, then the calibration factors are an average of all calibrations between epoch and epoch + duration.

The routine `LALResponseConvert()` takes a given frequency series and converts it to a new frequency series by performing the following steps: (i) the original frequency series is interpolated (using linear interpolation of the real and imaginary parts independently) to the frequencies required in the output frequency series; (ii) if the output frequency series has units that are the inverse of those of the input frequency series, the data is inverted; (iii) the data is scaled by an appropriate power of ten computed from the input units and the output units. For example you can convert from strain per count to counts per atto-strain.

#### Algorithm

The transfer function is deduced from the poles and zeros as follows:

\[
T(f) = c_{gain} \prod_{i} \text{zero}(f, z_i) \prod_{i} \text{pole}(f, p_i) \quad (7.22)
\]
where
\[
\text{zero}(f, z) = \begin{cases} 
  (if/z) + 1 & \text{when } z \neq 0 \\
  if & \text{when } z = 0 
\end{cases}
\] (7.23)

and
\[
\text{pole}(f, p) = \begin{cases} 
  (1/ij) + 1 & \text{when } p \neq 0 \\
  1 & \text{when } p = 0 
\end{cases}
\] (7.24)

For both the transfer function and the pole-zero notation the units for frequency is Hz rather than rad/s (angular frequency). In particular, poles and zeros are specified by their location in frequency space.

To update the response function from one epoch to another, two functions are needed. These are the sensing function \( C(f) \) and the response function \( R(f) \), which are related by
\[
R(f) = \frac{1 + H(f)}{C(f)}
\] (7.25)

where \( H(f) \) is the open-loop gain function. If the sensing function and the open-loop gain function are known at some reference time \( C_0(f) \) and \( H_0(f) \) then the sensing function and open-loop gain function can be calculated at a later time. They are \( C(f) = \alpha C_0(f) \) and \( H(f) = \alpha \beta H_0(f) \) where \( \alpha \) and \( \beta \) are slowly varying coefficients that account for overall changes in the gains of the sensing function and the open-loop gain. The coefficients \( \alpha \) and \( \alpha \beta \) can be determined, as slowly-varying functions of time, by monitoring the two injected calibration lines. Thus, an updated sensing function and response function can be computed from reference sensing function and response function, \( C_0(f) \) and \( R_0(f) \) via:
\[
C(f) = \alpha C_0(f)
\] (7.26)

and
\[
R(f) = \frac{1 + \alpha \beta [C_0(f) R_0(f) - 1]}{\alpha C_0(f)}
\] (7.27)

where \( \alpha \) and \( \beta \) are those values of the coefficients that are appropriate for the particular epoch.

Uses

Notes
The DC component of \texttt{calrec->transfer} is always filled with \( 1 + i0 \). In most cases, this should be irrelevant for gravitational wave data analysis, but care should be taken if DC is relevant when this function is used.
7.6 Header ResampleTimeSeries.h

Provides routines to resample a time series. At present only integer downsampling or REAL4TimeSeries by a power of two is supported.

Synopsis

#include <lal/ResampleTimeSeries.h>

This header covers routines that resample time series by applying a low pass filter and decimating the resulting time series. Further documentation is given in the individual routines’ modules.

Error conditions

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>1</td>
<td>&quot;Null pointer&quot;</td>
</tr>
<tr>
<td>NNUL</td>
<td>2</td>
<td>&quot;Non-null pointer&quot;</td>
</tr>
<tr>
<td>ZERO</td>
<td>3</td>
<td>&quot;Length of input time series is zero&quot;</td>
</tr>
<tr>
<td>RATE</td>
<td>4</td>
<td>&quot;Sample rate is zero&quot;</td>
</tr>
<tr>
<td>UPSM</td>
<td>5</td>
<td>&quot;Cannot upsample&quot;</td>
</tr>
<tr>
<td>HIGH</td>
<td>6</td>
<td>&quot;Input sample rate is greater than 32kHz&quot;</td>
</tr>
<tr>
<td>LOG2</td>
<td>7</td>
<td>&quot;Only power-of-two resampling is available&quot;</td>
</tr>
<tr>
<td>FILT</td>
<td>8</td>
<td>&quot;Unknown filter type&quot;</td>
</tr>
<tr>
<td>INVD</td>
<td>9</td>
<td>&quot;Invalid or non-integer resample factor&quot;</td>
</tr>
<tr>
<td>LDAS</td>
<td>10</td>
<td>&quot;Input resample factor with LDAS FIR&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants RESAMPLETIMESERIESH_<name>, and the status descriptions in RESAMPLETIMESERIESH_MSGE<name>. The source code with these messages is in ResampleTimeSeries.h on line 1.83.

Types

Enum ResampleTSFilter

This enum type contains the different low pass filters available to prevent power above the new Nyquist frequency entering the resampled time series due to aliasing. The options are:

defaultButterworth An IIR butterworth filter of order 20 with attenuation 0.1 at the new Nyquist frequency. See the package tdfilters for documentation of butterworth filters in LAL.

LDASfirLP For downsampling by a factor of 2, 4 or 8 an implementation of the FIR filter used by the LDAS datacondAPI resample(). This is provided for testing the result of standalone codes and codes running under LDAS. The LDAS filter provided here has filter order parameter 10, so the order of the filter is \(2 \times 10 \times q\) where \(q\) is the resampling ratio.

Union ResampleTSFilterParams

This union is provided so that the code can store the parameters of the filter in a place accessible by the user for user designed low pass filters. This is not presently implemented and this structure may be ignored.

butterworth A structure of type PassBandParamStruc used to store the parameters of the butterworth filter used to perform low pass filtering.

iirfilter A structure of type REAL8IIRFilter used to store the parameters of the IIR or FIR filter used to perform low pass filtering.
Structure `tagResampleTimeSeriesParams`

This structure controls the behaviour of the resampling function.

`deltaT` The sample interval desired in the down sampled time series.

`filterType` The type of filter with which to perform the low pass filtering.

`filterParams` Filter parameters for the low pass filter. Presently ignored.
7.6.1 Module ResampleTimeSeries.c

Downsamples a time series in place by an integer power of two.

Prototypes

```c
void ResampleTimeSeriesREAL4TimeSeries(  
    LALStatus *status,  
    REAL4TimeSeries *ts,  
    ResampleTSParms *params
)
```

Description

The routine `LALResampleREAL4TimeSeries()` provided functionality to downsample a time series in place by an integer factor which is a power of two. Upsampling, non-integer resampling and resampling by a factor which is not a power of two are currently unsupported. Attempts to use this methods will cause the function to abort with an error message.

On entry the input time series `ts` should contain the data to be resampled, with the data, length and sample interval of the time series populated. The parameter structure `params` should have the value of `deltaT` set to the desired value sample interval of the output data. The type of filter used to perform the low pass filter should be set in the parameter structure field `filterType`. The parameter structure field `filterParams` may be ignored at present. It is designed to allow user defined low pass filters and is not currently implemented. The resampling function will behave correctly even if this contains garbage on entry.

On exit, the time series `ts` will contain the resampled time series. The length of the time series will be reduced by the resampling ratio and the sample interval will be set correctly.

There is no time shift in the output time series when either the `defaultButterworth` or `LDASfirLP` low pass filter types are used. The timestamp of each point in the output time series is the same as the timestamp of the corresponding point in input time series.

There will be corrupted data at the start and end of the time series when either the `defaultButterworth` or `LDASfirLP` low pass filter types are used. This is caused by corruption due to the low pass filter. Users should take care to truncate these points from the time series before using the output in subsequent filtering applications.

Algorithm

The input time series is first low passed to remove any power above the new Nyquist frequency. There are two available low pass filters:

1. `defaultButterworth` The input time series has a time domain low pass filter applied by the `LALDButterworthREAL4TimeSeries()` function from the tdfilters package. The filter order is 20 and the attenuation is 0.1 at the new Nyquist frequency. Since the butterworth filter is applied forwards and backwards there is no time delay or phase shift of the output data. The disadvantage of the butterworth filter is that since it is an IIR filter, it is not possible to determine the exact length of corrupted data at the start and end of the output time series. Care should be taken to avoid using these regions of data in subsequent filtering.

2. `LDASfirLP` The input time series has a time domain low pass filter applied by the `LALDIIRFilterREAL4Vector()` function from the tdfilters package. This applies an FIR filter with coefficients generated by the LDAS `firlp()` dataconditioning API action. FIR coefficients are available for downsampling by a factor of 2, 4 or 8. An attempt to downsample by any other ratio will cause an error. The FIR coefficients are equivalent to the default `resample()` action in the dataconditioning API.

The FIR filter causes the a time delay of 10 points in the output time series and corruption of the first `n` points. `n` is given by the order of the filter by `n = 2 \times 10 \times q`, where `q` is the resampling ratio. To account for this, we do the following:

(a) The first `n/2` points of the time series are deleted.
(b) The whole time series is shifted $n/2$ points “left” to remove the time delay.

(c) The first and last $n/2$ points of the output time series are set to zero to make sure that the corrupted data contains sensible numbers and not garbage or Inf.

This means that there is no time delay in the output time series, but the first and last $n/2q$ points are set to zero. Care should be taken to avoid using these regions of data in subsequent filtering. If the debug level is set to LALWARNING, a message is printed reporting how many points at the start and end of time series are corrupted.

The filter coefficients used were produced by LDAS-CIT running version 0.7.0 of LDAS. See the LDAS dataconditioning API documentation for more information.

Uses

lalDebugLevel
LALInfo()
LALDButterworthREAL4TimeSeries()
LALDCreateVector()
LALDIIRFilterREAL4Vector()
LALSnprintf()
LALWarning()
LALDestroyVector()
LALReallocl()
memmove()

Notes
7.6.2 Program ResampleTimeSeriesTest.c

Tests the routines in ResampleTimeSeries.h

Usage

Usage: ./ResampleTimeSeriesTest [options] defaults shown in brackets
Options:
- h  print this message
- d level set lalDebugLevel to level
- v  verbose: print extra information
- n points number of points in the raw time series (1048576)
- i freq sample rate of input time series (16384)
- o freq sample rate of output time series (4096)
- f freq frequency of sine wave to inject as input (1000.0)
- r type type of filter to use in resampling (ldas)
               [ldas|butterworth]

Description

Tests the resampling functions by injecting a sine wave of a given frequency into a time series and down-
sampling it. The raw and output data are returned as frame files for plotting in matlab.

Sample Results

Figures 7.1–7.3 show the results of various tests using this program.

Figure 7.1: The left figure shows a 10 Hz sine wave generated at 16384 Hz resampled to 4096 Hz. The
right figure shows a 100 Hz sine wave generated at 16384 Hz resampled to 4096 Hz. Note that there is no
attenuation, time delay or phase shift of the output. FIXME the legend in the right figure is wrong. It
should say 100 Hz, not 10 Hz the output.
Figure 7.2: A 100Hz sine wave generated at 16384 Hz resampled to 8192 Hz. The left plot shows the start of the time series and the right plot the end. Note the corruption of points due to the time domain filtering.

Figure 7.3: The left figure shows a 1000 Hz sine wave generated at 16384 Hz resampled to 4096 Hz. The right figure shows a 1000Hz sine wave generated at 16384 Hz resampled to 2048 Hz. Note that there is no attenuation, time delay or phase shift of the output at 4096 Hz, however there is attenuation and phase shift of the output at 2048 Hz. This is due to the fact that the signal is very close to the output Nyquist frequency. Care should be taken to downsample to a suitable rate to avoid this type of attenuation.

### 7.7 Header LIGOMetadataTables.h

Provides data type definitions for the LIGO metadata database tables. Only columns that are available for the user defined values are available in the structures. For example the `process_id` column in the `sngl_inspiral` table is not in the corresponding LAL structure as it is generated by the database on insertion of events, not by the user.

#### Synopsis

```c
#include <lal/LIGOMetadataTables.h>
```

#### Types

Type `MetadataTableType`

```c
typedef enum
{
  no_table,
  process_table,
  process_params_table,
...```
search_summary_table,
search_summvars_table,
sngl_burst_table,
sngl_inspiral_table,
sngl_inspiral_table_bns,
sngl_inspiral_table_bcv,
sngl_ringdown_table,
sngl_transdata_table,
multi_inspiral_table,
sim_inspiral_table,
sim_ringdown_table,
summ_value_table,
sim_inst_params_table,
coinc_inspiral_table,
coinc_ringdown_table,
stochastic_table,
stoch_summ_table,
ext_triggers_table,
filter_table
}

MetadataTableType;

The MetadataTableType contains an enum type for each of the possible database tables that are provided.

Type ProcessTable

typedef enum
{
    LAL_IFO_G1,
    LAL_IFO_H1,
    LAL_IFO_H2,
    LAL_IFO_L1,
    LAL_IFO_T1,
    LAL_IFO_V1,
    LAL_NUMIFO,
    LAL_UNKNOWNIFO = -1
} InterferometerNumber;

The InterferometerNumber contains an enum type for describing the interferometer.

Type ProcessIDColumn

typedef struct
tagProcessIDColumn
{
    UINT8 id;
    CHAR textId[LIGOMETA_UNIQUE_MAX];
    UCHAR dbUniqueId[LIGOMETA_DBUNIQUE_MAX];
} ProcessIDColumn;

The ProcessIDColumn contains the process_id for a given row in a table.

Type EventIDColumn

typedef struct
tagEventIDColumn
{
    struct tagEventIDColumn *next;
}
The `EventIDColumn` structure is used to link different tables that refer to the same event. The database column `event_id` is a ID generated by the database that is generally unique within a given event table (e.g. `sngl_inspiral` or `sngl_burst`), but may be shared across tables. For example, an entry in the `sngl_burst` may have several entries in the `sngl_transdata` table with the same event ID or a `sngl_inspiral` many have a corresponding entry in the `sngl_burst` with the same event ID, e.g. for an inspiral-ringdown search.

When the LAL XML output routines are called, the generate XML that the database can interpret to set the `event_id` columns correctly in the tables.

If the user does not wish to make use of the event ID at the level of the search, for example if the user is generating a list of inspiral events and does not want to link these to any other tables, the `event_id` pointer in the `SnglInpiralTable` may be set to NULL. In this case the XML output routines will generate the correct XML to tell the database to generate a unique event ID for each inspiral event, but these should not be shared across tables.

```c
typedef struct tagProcessTable
{
    struct tagProcessTable *next;
    CHAR program[LIGOMETA_PROGRAM_MAX];
    CHAR version[LIGOMETA_VERSION_MAX];
    CHAR cvs_repository[LIGOMETA_CVS_REPOSITORY_MAX];
    LIGOTimeGPS cvs_entry_time;
    CHAR comment[LIGOMETA_COMMENT_MAX];
    INT4 is_online;
    CHAR node[LIGOMETA_NODE_MAX];
    CHAR username[LIGOMETA_USERNAME_MAX];
    LIGOTimeGPS start_time;
    LIGOTimeGPS end_time;
    INT4 jobid;
    CHAR domain[LIGOMETA_DOMAIN_MAX];
    INT4 unix_procid;
    CHAR ifos[LIGOMETA_IFOS_MAX];
} ProcessTable;
```

Document table.

**Type ProcessParamsTable**

```c
typedef struct tagProcessParamsTable
{
    struct tagProcessParamsTable *next;
    CHAR program[LIGOMETA_PROGRAM_MAX];
    CHAR param[LIGOMETA_PARAM_MAX];
} ProcessParamsTable;
```
CHAR             type[LIGOMETA_TYPE_MAX];
CHAR             value[LIGOMETA_VALUE_MAX];
}
ProcessParamsTable;

Document table.

Type SearchSummaryTable

/* shared object is lalapps, lalwrapper_cvs_tag is lalapps cvs tag */
/* lal_cvs tag is populated automatically */
typedef struct
tagSearchSummaryTable
{
    struct tagSearchSummaryTable *next;
    CHAR             comment[LIGOMETA_COMMENT_MAX];
    LIGOTimeGPS      in_start_time;
    LIGOTimeGPS      in_end_time;
    LIGOTimeGPS      out_start_time;
    LIGOTimeGPS      out_end_time;
    INT4             nevents;
    INT4             nnodes;
    CHAR             ifos[LIGOMETA_IFOS_MAX];
} SearchSummaryTable;

Document table.

Type SearchSummvarsTable

typedef struct
tagSearchSummvarsTable
{
    struct tagSearchSummvarsTable *next;
    CHAR             name[LIGOMETA_NAME_MAX];
    CHAR             string[LIGOMETA_STRING_MAX];
    REAL8            value;
} SearchSummvarsTable;

Document table.

Type SnglBurstTable

typedef struct
tagSnglBurstTable
{
    struct tagSnglBurstTable *next;
    CHAR             ifo[LIGOMETA_IFO_MAX];
    CHAR             search[LIGOMETA_SEARCH_MAX];
    CHAR             channel[LIGOMETA_CHANNEL_MAX];
    LIGOTimeGPS      start_time;
    LIGOTimeGPS      peak_time;
    REAL4            duration;
    REAL4            central_freq;
    REAL4            bandwidth;
    REAL4            amplitude;
    REAL4            snr;
    REAL4            confidence;
REAL4  string_cluster_t;
UINT4  event_id;
}
SnglBurstTable;

Document table.

Type SnglInspiralTable

typedef struct
tagSnglInspiralTable
{
  struct tagSnglInspiralTable *next;
  CHAR     ifo[LIGOMETAIFO_MAX];
  CHAR     search[LIGOMETASEARCH_MAX];
  CHAR     channel[LIGOMETACHANNEL_MAX];
  LIGOTimeGPS end_time;
  REAL8    end_time_gmst;
  LIGOTimeGPS impulse_time;
  REAL8    template_duration;
  REAL8    event_duration;
  REAL4    amplitude;
  REAL4    eff_distance;
  REAL4    coa_phase;
  REAL4    mass1;
  REAL4    mass2;
  REAL4    mchirp;
  REAL4    mtotal;
  REAL4    eta;
  REAL4    kappa;
  REAL4    chi;
  REAL4    tau0;
  REAL4    tau2;
  REAL4    tau3;
  REAL4    tau4;
  REAL4    tau5;
  REAL4    ttotal;
  REAL4    psi0;
  REAL4    psi3;
  REAL4    alpha;
  REAL4    alpha1;
  REAL4    alpha2;
  REAL4    alpha3;
  REAL4    alpha4;
  REAL4    alpha5;
  REAL4    alpha6;
  REAL4    beta;
  REAL4    f_final;
  REAL4    snr;
  REAL4    chisq;
  INT4     chisq_dof;
  REAL4    bank_chisq;
  INT4     bank_chisq_dof;
  REAL4    cont_chisq;
  INT4     cont_chisq_dof;
  REAL8    sigmasq;
  REAL4    rsqveto_duration;
  REAL4    Gamma[10];  /* metric co-efficients */
  EventIDColumn *event_id;
}
Type \texttt{SnglRingdownTable}

```c
typedef struct
  tagSnglRingdownTable
{
  struct tagSnglRingdownTable *next;
  CHAR ifo[LIGOMETAIFO_MAX];
  CHAR channel[LIGOMETA_CHANNEL_MAX];
  LIGOTimeGPS start_time;
  REAL8 start_time_gmst;
  REAL4 frequency;
  REAL4 quality;
  REAL4 phase;
  REAL4 mass;
  REAL4 spin;
  REAL4 epsilon;
  INT4 num_clust_trigs;
  REAL4 ds2_H1H2;
  REAL4 ds2_H1L1;
  REAL4 ds2_H2L1;
  REAL4 amplitude;
  REAL4 snr;
  REAL4 eff_dist;
  REAL8 sigma_sq;
  EventIDColumn *event_id;
} SnglRingdownTable;
```

This structure contains the required information for generating a ringdown template and storing ringdown triggers. The fields are:

- **ifo** The interferometer in which the trigger is found.
- **channel** The interferometer channel in which the ringdown is found.
- **start\_time** The GPS start time of the ringdown trigger.
- **frequency** The central frequency of the ringdown waveform (in Hz).
- **quality** The quality factor $Q$ of the ringdown waveform.
- **phase** The initial phase of the ringdown in radians. Zero is a cosine-phase ringdown; $-\pi/2$ is a sine-phase ringdown.
- **mass** The mass $M$ of the black hole (in solar masses, $M_\odot$).
- **spin** The dimensionless spin parameter of the black hole $\hat{a}$ where the spin is $S = \hat{a}GM^2/c$ ($G$ is Newton constant and $c$ is the speed of light).
- **epsilon** The fractional mass loss $\epsilon$ of the initial black hole mass in ringdown radiation.
- **amplitude** The peak strain amplitude $h_0$ of the waveform, given by

$$ h_0 = \sqrt{\frac{5}{2}} \epsilon \left(\frac{GM}{c^2 r}\right) Q^{-\frac{1}{2}} F(Q)^{-\frac{1}{4}} g(a)^{-\frac{1}{2}}, \quad (7.28) $$

where $F(Q) = 1 + 7/(24Q^2)$, $g(a) = 1 - 0.63(1 - a)^{3/10}$ and $r$ is the distance to the trigger.
**snr**  The signal-to-noise ratio of the trigger.

**eff_dist**  The effective distance to the trigger source in megaparsecs (Mpc).

**sigma_sq**  The variance of the matched filter used to find the trigger.

**Type**  `MultiInspiralTable`

define struct
tagMultiInspiralTable
{
    struct tagMultiInspiralTable *next;
    CHAR ifos[LIGOMETA_IFOS_MAX];
    CHAR search[LIGOMETA_SEARCH_MAX];
    LIGOTimeGPS end_time;
    REAL8 end_time_gmst;
    LIGOTimeGPS impulse_time;
    REAL4 amplitude;
    REAL4 eff_distance;
    REAL4 ifo1_eff_distance;
    REAL4 ifo2_eff_distance;
    REAL4 coa_phase;
    REAL4 mass1;
    REAL4 mass2;
    REAL4 mchirp;
    REAL4 eta;
    REAL4 tau0;
    REAL4 tau2;
    REAL4 tau3;
    REAL4 tau4;
    REAL4 tau5;
    REAL4 ttotal;
    REAL4 snr;
    REAL4 ifo1_snr;
    REAL4 ifo2_snr;
    REAL4 chisq;
    INT4 chisq_dof;
    REAL4 bank_chisq;
    INT4 bank_chisq_dof;
    REAL4 cont_chisq;
    INT4 cont_chisq_dof;
    REAL4 sigmasq;
    REAL4 ligo_axis_ra;
    REAL4 ligo_axis_dec;
    REAL4 ligo_angle;
    REAL4 ligo_angle_sig;
    REAL4 inclination;
    REAL4 polarization;
    REAL4 null_statistic;
    COMPLEX8 h1quad;
    COMPLEX8 h2quad;
    COMPLEX8 l1quad;
    COMPLEX8 g1quad;
    COMPLEX8 v1quad;
    COMPLEX8 t1quad;
    EventIDColumn *event_id;
}
MultiInspiralTable;

Document table.
Type SimInspiralTable

typedef struct
tagSimInspiralTable
{
    struct tagSimInspiralTable *next;
    CHAR waveform[LIGOMETA_WAVEFORM_MAX];
    LIGOTimeGPS geocent_end_time;
    LIGOTimeGPS h_end_time;
    LIGOTimeGPS l_end_time;
    LIGOTimeGPS g_end_time;
    LIGOTimeGPS t_end_time;
    LIGOTimeGPS v_end_time;
    REAL8 end_time_gmst;
    CHAR source[LIGOMETA_SOURCE_MAX];
    REAL4 mass1;
    REAL4 mass2;
    REAL4 eta;
    REAL4 distance;
    REAL4 longitude;
    REAL4 latitude;
    REAL4 inclination;
    REAL4 coa_phase;
    REAL4 polarization;
    REAL4 psi0;
    REAL4 psi3;
    REAL4 alpha;
    REAL4 alpha1;
    REAL4 alpha2;
    REAL4 alpha3;
    REAL4 alpha4;
    REAL4 alpha5;
    REAL4 alpha6;
    REAL4 beta;
    REAL4 spin1x;
    REAL4 spin1y;
    REAL4 spin1z;
    REAL4 spin2x;
    REAL4 spin2y;
    REAL4 spin2z;
    REAL4 theta0;
    REAL4 phi0;
    REAL4 f_lower;
    REAL4 f_final;
    REAL4 mchirp;
    REAL4 eff_dist_h;
    REAL4 eff_dist_l;
    REAL4 eff_dist_g;
    REAL4 eff_dist_t;
    REAL4 eff_dist_v;
    EventIDColumn *event_id;
    INT4 numrel_mode_min;
    INT4 numrel_mode_max;
    CHAR numrel_data[LIGOMETA_STRING_MAX];
}
SimInspiralTable;

Document table.
Type `CoincInspiralTable`  

```c
typedef struct tagCoincInspiralTable {
    struct tagCoincInspiralTable *next;
    CHAR ifos[LIGOMETA_IFOS_MAX];
    INT4 numIfos;
    SnglInspiralTable *snglInspiral[LAL_NUMIFO];
    SimInspiralTable *simInspiral;
} CoincInspiralTable;
```

The `CoincInspiralTable` contains a set of pointers to `SnglInspiral` tables. The length of the array is set to `LAL_NUMIFO` which is a value of the enum `InterferometerNumber` described above. This enum also provides an easy way to access the `SnglInspiral` corresponding to each ifo. In addition the table, it contains a field in which to store the number and names of the IFOs which have non-null `SnglInspiral` tables. Finally, it contains a pointer to an associated `SimInspiral`. This table is meant to provide a simple way to manipulate coincident triggers.

Type `SimBurst`  

```c
typedef struct tagSimBurst {
    struct tagSimBurst *next;
    long process_id;
    char waveform[LIGOMETA_WAVEFORM_MAX];
    REAL8 ra;
    REAL8 dec;
    REAL8 psi;
    LIGOTimeGPS time_geocent_gps;
    REAL8 time_geocent_gmst;
    REAL8 duration;
    REAL8 frequency;
    REAL8 bandwidth;
    REAL8 q;
    REAL8 pol_ellipse_angle;
    REAL8 pol_ellipse_e;
    REAL8 amplitude;
    REAL8 hrss;
    REAL8 egw_over_rsquared;
    /* FIXME: this should be unsigned long */
    long waveform_number;
    long simulation_id;
} SimBurst;
```

Document table.

Type `SimRingTable`  

```c
typedef struct tagSimRingdownTable {
    struct tagSimRingdownTable *next;
    CHAR waveform[LIGOMETA_WAVEFORM_MAX];
    CHAR coordinates[LIGOMETA_COORDINATES_MAX];
    LIGOTimeGPS geocent_start_time;
    LIGOTimeGPS h_start_time;
    LIGOTimeGPS l_start_time;
    REAL8 start_time_gmst;
} SimRingTable;
```
REAL4 longitude;
REAL4 latitude;
REAL4 distance;
REAL4 inclination;
REAL4 polarization;
REAL4 frequency;
REAL4 quality;
REAL4 phase;
REAL4 mass;
REAL4 spin;
REAL4 epsilon;
REAL4 amplitude;
REAL4 eff_dist_h;
REAL4 eff_dist_l;
REAL4 hrss;
REAL4 hrss_h;
REAL4 hrss_l;
EventIDColumn *event_id;
} SimRingdownTable;

This structure contains the required information for generating a ringdown injection. The fields are:

- **waveform**  Description of the type of waveform to inject.
- **coordinates** Type of coordinate system to perform injection.
- **geocent_start_time** The GPS start time of the waveform at the center of the earth.
- **h_start_time** The GPS start time of the waveform at the LIGO Hanford Observatory.
- **l_start_time** The GPS start time of the waveform at the LIGO Livingston Observatory.
- **start_time_gmst** The start time of the waveform at the sidereal time.
- **longitude** The longitude of the source defined in the stated coordinate system.
- **latitude** The latitude of the source defined in the stated coordinate system.
- **distance** The distance to the source defined in the stated coordinate system in megaparsecs (Mpc).
- **inclination** The inclination angle $\iota$ of the source.
- **polarization** The polarization angle $\Psi$ of the source.
- **frequency** The central frequency of the ringdown waveform (in Hz).
- **quality** The quality factor $Q$ of the ringdown waveform.
- **phase** The initial phase of the ringdown signal $\phi_0$.
- **mass** The mass $M$ of the black hole (in solar masses, $M_\odot$).
- **spin** The dimensionless spin parameter of the black hole $\hat{a}$ where the spin is $S = \hat{a}GM^2/c$ ($G$ is Newton constant and $c$ is the speed of light).
- **epsilon** The fractional mass loss $\epsilon$ of the initial black hole mass in ringdown radiation.
- **amplitude** The peak strain amplitude $h_0$ of the waveform, given by

$$h_0 = \sqrt{\frac{5}{2}} \epsilon \left(\frac{GM}{c^2r}\right) Q^{-\frac{3}{2}} F(Q)^{-\frac{1}{2}} g(a)^{-\frac{1}{2}},$$

(7.29)

where $F(Q) = 1 + 7/(24Q^2)$, $g(a) = 1 - 0.63(1 - a)^{3/10}$ and $r$ is the distance to the trigger.

- **h_eff_dist** The effective distance to the source from the LIGO Hanford Observatory in megaparsecs (Mpc).
**l_eff_dist** The effective distance to the source from the LIGO Livingston Observatory in megaparsecs (Mpc).

**hrss** Define.

**hrss.h** Define.

**hrss.l** Define.

**Type CoincRingdownTable**

```c
typedef struct
tagCoincRingdownTable
{
    struct tagCoincRingdownTable *next;
    CHAR ifos[LIGOMETA_IFOS_MAX];
    INT4 numIfos;
    SnglRingdownTable *snglRingdown[LAL_NUM_IFO];
    SimRingdownTable *simRingdown;
}
CoincRingdownTable;
```

Document table.

**Type SnglTransdataTable**

```c
typedef struct
tagSnglTransdataTable
{
    CHAR event_table[LIGOMETA_TRANSDATA_EVENT_TABLE_MAX];
    CHAR ifo[LIGOMETA_IFO_MAX];
    CHAR name[LIGOMETA_TRANSDATA_NAME_MAX];
    INT4 dimensions;
    INT4 x_bins;
    REAL8 x_start;
    REAL8 x_end;
    CHAR x_units[LIGOMETA_TRANSDATA_UNITS_MAX];
    INT4 y_bins;
    REAL8 y_start;
    REAL8 y_end;
    CHAR y_units[LIGOMETA_TRANSDATA_UNITS_MAX];
    CHAR data_type[LIGOMETA_TRANSDATA_DATA_MAX];
    CHAR data_units[LIGOMETA_TRANSDATA_DATA_MAX];
    UCHAR *trans_data; /* must be big Endian */
    INT4 transdata_length;
    EventIDColumn *event_id;
}
SnglTransdataTable;
```

Document table.

**Type SummValueTable**

```c
typedef struct
tagSummValueTable
{
    struct tagSummValueTable *next;
    CHAR program[LIGOMETA_PROGRAM_MAX];
    CHAR frameset_group[LIGOMETA_FRAMESETG_MAX];
    CHAR segment_group[LIGOMETA_SEGMENTG_MAX];
    INT4 version;
```
LIGOTimeGPS  start_time;
LIGOTimeGPS  end_time;
CHAR        ifo[LIGOMETAIFO_MAX];
CHAR        name[LIGOMETA_SUMMVALUE_NAME_MAX];
REAL4       value;
REAL4       error;
INT4        intvalue;
CHAR        comment[LIGOMETA_SUMMVALUE_COMM_MAX];
}
SummValueTable;

Document table.

**Type** SimInstParamsTable

typedef struct
tagSimInstParamsTable
{
    struct tagSimInstParamsTable *next;
    CHAR        name[LIGOMETA_SIMINSTPARAMS_NAME_MAX];
    CHAR        comment[LIGOMETA_SIMINSTPARAMS_COMM_MAX];
    REAL8       value;
}
SimInstParamsTable;

Document table.

**Type** StochasticTable

typedef struct
tagStochasticTable
{
    struct tagStochasticTable *next;
    CHAR        ifo_one[LIGOMETAIFO_MAX];
    CHAR        ifo_two[LIGOMETAIFO_MAX];
    CHAR        channel_one[LIGOMETA_CHANNEL_MAX];
    CHAR        channel_two[LIGOMETA_CHANNEL_MAX];
    LIGOTimeGPS start_time;
    LIGOTimeGPS duration;
    REAL8       f_min;
    REAL8       f_max;
    REAL8       cc_stat;
    REAL8       cc_sigma;
}
StochasticTable;

The **StochasticTable** contains output parameters relevant for the stochastic search. The IFOs, channels, start time, duration, minimum and maximum frequency, cross correlation statistic and theoretical variance are stored.

**Type** StochSummTable

typedef struct
tagStochSummTable
{
    struct tagStochSummTable *next;
    CHAR        ifo_one[LIGOMETAIFO_MAX];
    CHAR        ifo_two[LIGOMETAIFO_MAX];
    CHAR        channel_one[LIGOMETA_CHANNEL_MAX];
    CHAR        channel_two[LIGOMETA_CHANNEL_MAX];
LIGOTimeGPS start_time;
LIGOTimeGPS end_time;
REAL8 f_min;
REAL8 f_max;
REAL8 y_opt;
REAL8 error;
}
StochSummTable;

The **StochSummTable** contains a summary of a stochastic search. It can be used to fully summarise the parameters used for a full search.

**Type ExtTriggerTable**

typedef struct
tagExtTriggerTable
{
    struct tagExtTriggerTable *next;
    CHAR det_alts[LIGOMETA_STD];
    CHAR det_band[LIGOMETA_STD];
    CHAR det_fluence[LIGOMETA_STD];
    CHAR det_fluence_int[LIGOMETA_STD];
    CHAR det_name[LIGOMETA_STD];
    CHAR det_peak[LIGOMETA_STD];
    CHAR det_peak_int[LIGOMETA_STD];
    CHAR det_snr[LIGOMETA_STD];
    INT4 email_time;
    REAL4 event_dec;
    REAL4 event_dec_err;
    CHAR event_epoch[LIGOMETA_STD];
    CHAR event_err_type[LIGOMETA_STD];
    REAL4 event_ra;
    REAL4 event_ra_err;
    INT4 start_time;
    INT4 start_time_ns;
    CHAR event_type[LIGOMETA_STD];
    REAL4 event_z;
    REAL4 event_z_err;
    CHAR notice_comments[LIGOMETA_STD];
    CHAR notice_id[LIGOMETA_STD];
    CHAR notice_sequence[LIGOMETA_STD];
    INT4 notice_time;
    CHAR notice_type[LIGOMETA_STD];
    CHAR notice_url[LIGOMETA_STD];
    REAL4 obs_fov_dec;
    REAL4 obs_fov_dec_width;
    REAL4 obs_fov_ra;
    REAL4 obs_fov_ra_width;
    REAL4 obs_loc_ele;
    REAL4 obs_loc_lat;
    REAL4 obs_loc_long;
    REAL4 ligo_fave_lho;
    REAL4 ligo_fave_llo;
    REAL4 ligo_delay;
    INT4 event_number_gcn;
    CHAR event_number_grb[8];
    INT4 event_status;
    EventIDColumn *event_id;
}
ExtTriggerTable;

Document table.

**Type** FilterTable

typedef struct
tagFilterTable
{
   struct tagFilterTable *next;
   CHAR program[LIGOMETA_PROGRAM_MAX];
   INT4 start_time;
   CHAR filter_name[LIGOMETA_FILTER_NAME_MAX];
   CHAR comment[LIGOMETA_SUMMVALUE_COMM_MAX];
} FilterTable;

Document table.

**Type** MetadataTable

typedef union
tagMetadataTable
{
   ProcessTable *processTable;
   ProcessParamsTable *processParamsTable;
   SearchSummaryTable *searchSummaryTable;
   SearchSummvarsTable *searchSummvarsTable;
   SnglBurstTable *snglBurstTable;
   SnglInspiralTable *snglInspiralTable;
   SnglRingdownTable *snglRingdownTable;
   MultiInspiralTable *multiInspiralTable;
   SimInspiralTable *simInspiralTable;
   SimBurst *simBurst;
   SimRingdownTable *simRingdownTable;
   SnglTransdataTable *snglTransdataTable;
   SummValueTable *summValueTable;
   SimInstParamsTable *simInstParamsTable;
   StochasticTable *stochasticTable;
   StochSummTable *stochSummTable;
   ExtTriggerTable *extTriggerTable;
   FilterTable *filterTable;
} MetadataTable;

Document table.
7.8 Header LIGOMetadataUtils.h

Provides functions for manipulating the LAL structures that correspond to the LIGO metadata database tables defined in LIGOMetadataTables.h.

Synopsis

#include <lal/LIGOMetadataUtils.h>

This header provides prototypes for routines that perform processing on the LAL structures that correspond to the LIGO metadata database tables defined in LIGOMetadataTables.h, such as sorting and eliminating duplicates. The functions specific to a particular metadata table (e.g. sngl_inspiral, sngl_burst, etc.) are all prototyped in this header.

Types

None.

Error codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>1</td>
<td>&quot;Null pointer&quot;</td>
</tr>
<tr>
<td>NNUL</td>
<td>2</td>
<td>&quot;Non-null pointer&quot;</td>
</tr>
<tr>
<td>TIME</td>
<td>3</td>
<td>&quot;Invalid GPS Time&quot;</td>
</tr>
<tr>
<td>COOR</td>
<td>4</td>
<td>&quot;Invalid Coordinate System&quot;</td>
</tr>
<tr>
<td>SGAP</td>
<td>5</td>
<td>&quot;Gap in Search Summary Input&quot;</td>
</tr>
<tr>
<td>SDUB</td>
<td>6</td>
<td>&quot;Repeated data in Search Summary Input&quot;</td>
</tr>
<tr>
<td>TEST</td>
<td>7</td>
<td>&quot;Unknown parameter test for sorting events&quot;</td>
</tr>
<tr>
<td>DET</td>
<td>8</td>
<td>&quot;Unknown detector&quot;</td>
</tr>
<tr>
<td>DIST</td>
<td>9</td>
<td>&quot;No horizon distance for consistency cut&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants LIGOMETADATAUTILSH_E<name>, and the status descriptions in LIGOMETADATAUTILSH_MSGE<name>. The source code with these messages is in LIGOMetadataUtils.h on line 1.102.

Types

typedef enum
{
    unspecified_data_type,
    playground_only,
    exclude_play,
    all_data
} LALPlaygroundDataMask;

The LALPlaygroundDataMask contains an enum type for describing the subset of data to be used, playground_only, exclude_play and all_data.

Type LALPlaygroundDataMask

Type SnglInspiralParameterTest

typedef enum
{
    unspecified_test,
    no_test,
    m1_and_m2,
    psi0_and_psi3,
    }
mchirp_and_eta, 
tau0_and_tau3, 
ellipsoid
}
SnglInspiralParameterTest;

The SnglInspiralParameterTest contains an enum type for each of the tests of mass parameters which
are used.

**Type SnglInspiralAccuracy**

typedef struct
tagSnglInspiralAccuracy
{
 INT4 match;
 REAL4 epsilon;
 REAL4 kappa;
 INT8 dt;
 REAL4 dm;
 REAL4 deta;
 REAL4 dmchirp;
 REAL4 dmchirpHi;
 REAL4 highMass;
 REAL4 dpsio;
 REAL4 dpsis3;
 REAL4 dtau0;
 REAL4 dtau3;
 SnglInspiralParameterTest test;
 INT4 exttrig;
}
SnglInspiralAccuracy;

The SnglInspiralAccuracy structure contains parameters used for testing coincidence between two or
more single inspiral tables. These include a timing accuracy dt, five mass accuracies dm (used for testing
mass1 and mass2), deta, dmchirp, dpsio and dpsis3. It also includes the parameters kappa and epsilon
which are used for testing consistency of effective distance.

**Type SnglInspiralClusterChoice**

typedef struct
tagInspiralAccuracyList
{
 INT4 match;
 SnglInspiralParameterTest test;
 SnglInspiralAccuracy ifoAccuracy[LAL_NUM_IFO];
 INT8 lightTravelTime[LAL_NUM_IFO][LAL_NUM_IFO];
 REAL4 iotaCutH1H2;
 REAL4 iotaCutH1L1;
 REAL8 eMatch;
 INT4 exttrig;
}
InspiralAccuracyList;

The InspiralAccuracyList structure contains parameter accuracies for each of the six global interferometers. These are stored in the SnglInspiralAccuracy structure. The accuracies stored should be the accuracy with which each instrument can determine the given parameter. It also contains a match which is set to 1 to signify that coincidence criteria are satisfied and 0 otherwise. Finally, the SnglInspiralParameterTest
must be specified.
Type `SnglInspiralClusterChoice`:

```c
typedef struct
tagCoincInspiralBittenLParams
{
    REAL4 param_a[LAL_NUMIFO];
    REAL4 param_b[LAL_NUMIFO];
} CoincInspiralBittenLParams;
```

The `CoincInspiralBittenLParams` structure contains the bitten L parameter for each of the six global interferometers. These are stored in the `param_a` and `param_b` structure.

### CoincInspiralStatistic

```c
typedef enum
{
    none,
    snr_and_chisq,
    snrsq_over_chisq,
    snr
}
SnglInspiralClusterChoice;
```

The `SnglInspiralClusterChoice` provides three choices for clustering a single inspiral table. The `snr` clustering returns the trigger with the greatest signal to noise ratio; `snr_and_chisq` replaces the existing trigger if the new trigger has both a greater snr and a smaller chi squared value; `snrsq_over_chisq` selects the trigger with the largest value of snr squared divided by the chi squared.

```c
typedef enum
{
    no_stat,
    snrsq,
    effective_snrsq,
    s3_snr_chisq_stat,
    bitten_l,
    bitten_lsq
}
CoincInspiralStatistic;
```

The `CoincInspiralStatistic` provides two choices for clustering a single inspiral table. The `snrsq` clustering returns the trigger with the greatest summed snr² from all instruments. The `snr_chisq_stat` replaces selects the trigger with the largest value of the snr and chisqa statistic and the `bitten_l` returns the minimum among the summed snr² from all instruments and the \( a \times snr_i - b \) in each detector. The parameters \( a \) and \( b \) must be provided by the user.

```c
typedef struct
tagSnglInspiralBCVCalphafCut
{
    REAL4 h1_lo;
    REAL4 h1_hi;
    REAL4 h2_lo;
    REAL4 h2_hi;
    REAL4 l1_lo;
    REAL4 l1_hi;
    REAL4 lsi0cut;
}
SnglInspiralBCVCalphafCut;
```

The `SnglInspiralBCVCalphafCut` provides entries for cutting single IFO triggers generated with the BCVC code. For each LSC IFO there is a field `lo` and `hi` which corresponds to the area allowing triggers.

Author: Brown, D. A. and Fairhurst, S.

$Id: LIGOMetadataUtils.h,v 1.142 2008/02/27 00:59:36 nvf Exp $
7.8.1 Module LIGOMetadataUtils.c

General routines for manipulating LIGO metadatabase tables.

Prototypes

```c
int XLALCountProcessTable(ProcessTable *head)
int XLALCountProcessParamsTable(ProcessParamsTable *head)
int XLALCountMultiInspiralTable(MultiInspiralTable *head)

int XLALIFONumber(const char *ifo)

void XLALReturnIFO(char *ifo, InterferometerNumber IFONumber)

void XLALReturnDetector(LALDetector *det, InterferometerNumber IFONumber)

void LALPlaygroundInSearchSummary(LALStatus *status, SearchSummaryTable *ssTable, LIGOTimeGPS *inPlayTime, LIGOTimeGPS *outPlayTime)

int XLALPlaygroundInSearchSummary(SearchSummaryTable *ssTable, LIGOTimeGPS *inPlayTime, LIGOTimeGPS *outPlayTime)

int LALCompareSearchSummaryByInTime(const void *a, const void *b)

int LALCompareSearchSummaryByOutTime(const void *a, const void *b)

int XLALTimeSortSearchSummary(SearchSummaryTable **summHead, int(*comparfunc)(const void *, const void *))
```
void LALTimeSortSearchSummary(
    LALStatus *status,
    SearchSummaryTable **summHead,
    int(*comparfunc) (const void *, const void *)
)

SearchSummaryTable *
XLALIfoScanSearchSummary(
    SearchSummaryTable *input,
    CHAR *ifos
)

void LALIfoScanSearchSummary(
    LALStatus *status,
    SearchSummaryTable **output,
    SearchSummaryTable *input,
    CHAR *ifos
)

void LALDistanceScanSummValueTable(
    LALStatus *status,
    SummValueTable *summValueList,
    LIGOTimeGPS gps,
    CHAR *ifo,
    REAL4 *distance
)

void LALCheckOutTimeFromSearchSummary(
    LALStatus *status,
    SearchSummaryTable *summList,
    CHAR *ifo,
    LIGOTimeGPS *startTime,
    LIGOTimeGPS *endTime
)

void LALIfoScanSummValue(
    LALStatus *status,
    SummValueTable **output,
    SummValueTable *input,
    CHAR *ifo
)

int LALCompareSummValueByTime(
    const void *a,
    const void *b
)

int XLALTimeSortSummValue(
    SummValueTable **summHead,
    int(*comparfunc) (const void *, const void *)
)

void LALTimeSortSummValue(
    LALStatus *status,
    SummValueTable **summHead,
    int(*comparfunc) (const void *, const void *)
)
Description

The function **LALPlaygroundInSearchSummary()** determines the amount of time in the search summary table `ssTable` that overlaps with playground data. The time between `in_start_time` and `in_end_time` that overlaps with playground is returned in `inPlayTime` and the time between `out_start_time` and `out_end_time` that overlaps with playground is returned in `outPlayTime`.

**LALCompareSearchSummaryByInTime()** is a function to compare the in times in two search summary tables. It returns 1 if the `in_start_time` of the first table is after the `in_start_time` of the second and -1 if it is before. If the two `in_start_times` are identical, the test is repeated on the `in_end_times`. If these are also equal, the comparison returns 0. **LALCompareSearchSummaryByOutTime()** operates in a similar manner, but uses the out, rather than in, times.

**LALTimeSortSearchSummary()** will time sort a linked list of search summary tables. You can sort on in our out start time depending which `comparfunc` is specified.

**LALIfoScanSearchSummary()** steps through a linked list of search summary tables and returns a pointer `output` to a linked list of those tables whose `ifos` field matches the string `ifos`.

**LALIfoScanSummValue(), LALCompareSummValueByTime()** and **LALTimeSortSummValue()** performs the same functions as described above. The only difference being that they act on summ value tables.

**LALCheckOutTimeFromSearchSummary()** verifies that all times between the specified `startTime` and `endTime` have been searched precisely once for the given `ifo`.

Finally, **LALDistanceScanSummValueTable()** scan a summ value table searching for a trigger belonging to a given `ifo` and englobing a give GPS time.

Algorithm

None.

Uses

LALGPStoINT8, LALCalloc, LALMalloc, LALFree.

Notes

Author: Brown, D. A.

$Id: LIGOMetadataUtils.c,v 1.30 2007/06/08 14:41:57 bema Exp$
### 7.8.2 Module SnglInspiralUtils.c

Provides a set of utilities for manipulating `snglInspiralTables`.

**Prototypes**

```c
void
LALFreeSnglInspiral (  
    LALStatus  *status,  
    SnglInspiralTable **eventHead  
)

int
XLALFreeSnglInspiral (  
    SnglInspiralTable **eventHead  
)

void
LALSOrtSnglInspiral (  
    LALStatus  *status,  
    SnglInspiralTable **eventHead,  
    int(*comparfunc)  (const void *, const void *)  
)

SnglInspiralTable *
XLALSOrtSnglInspiral (  
    SnglInspiralTable *eventHead,  
    int(*comparfunc)  (const void *, const void *)  
)

int
LALCompareSnglInspiralByMass (  
    const void *a,  
    const void *b  
)

int
LALCompareSnglInspiralByPsi (  
    const void *a,  
    const void *b  
)

int
LALCompareSnglInspiralByTime (  
    const void *a,  
    const void *b  
)

int
LALCompareSnglInspiralByID (  
    const void *a,  
    const void *b  
)

void
LALCompareSnglInspiral (  
    LALStatus  *status,  
    SnglInspiralTable *aPtr,  
    SnglInspiralTable *bPtr,  
    SnglInspiralAccuracy *params  
)
```
void LALCompareInspirals (  
  LALStatus *status,  
  SnglInspiralTable *aPtr,  
  SnglInspiralTable *bPtr,  
  InspiralAccuracyList *params  
)

int XLALCompareInspirals (  
  SnglInspiralTable *aPtr,  
  SnglInspiralTable *bPtr,  
  InspiralAccuracyList *params  
)

void LALClusterSnglInspiralTable (  
  LALStatus *status,  
  SnglInspiralTable **inspiralEvent,  
  INT8 dtimeNS,  
  SnglInspiralClusterChoice clusterchoice  
)

int XLALClusterSnglInspiralTable (  
  SnglInspiralTable **inspiralList,  
  INT8 dtimeNS,  
  SnglInspiralClusterChoice clusterchoice  
)

void LALTimeCutSingleInspiral (  
  LALStatus *status,  
  SnglInspiralTable **eventHead,  
  LIGOTimeGPS *startTime,  
  LIGOTimeGPS *endTime  
)

SnglInspiralTable * XLALTimeCutSingleInspiral (  
  SnglInspiralTable *eventHead,  
  LIGOTimeGPS *startTime,  
  LIGOTimeGPS *endTime  
)

void LALSNRCutSingleInspiral (  
  LALStatus *status,  
  SnglInspiralTable **eventHead,  
  REAL4 snrCut  
)

SnglInspiralTable * XLALSNRCutSingleInspiral (  
  SnglInspiralTable *eventHead,  
  REAL4 snrCut  
)
SnglInspiralTable *
XLALReqCutSingleInspiral (  
    SnglInspiralTable *eventHead,  
    REAL4 rsqVetoTimeThresh,  
    REAL4 rsqMaxSnr,  
    REAL4 rsqAboveSnrCoeff,  
    REAL4 rsqAboveSnrPow
)

SnglInspiralTable *
XLALVetoSingleInspiral (  
    SnglInspiralTable *eventHead,  
    LALSegList *vetoSegs,  
    CHAR *ifo
)

void
LALFreeMultiInspiral (  
    LALStatus *status,  
    MultiInspiralTable **eventHead
)

void
LALCompareRingdowns (  
    LALStatus *status,  
    SnglRingdownTable *aPtr,  
    SnglRingdownTable *bPtr,  
    RingdownAccuracyList *params
)

SnglRingdownTable *
XLALIfoCutSingleRingdown(  
    SnglRingdownTable **eventHead,  
    char *ifo
)

**Description**

The function `LALFreeSnglInspiral()` and free the memory associated to a single inspiral table. The single inspiral table may point to a linked list of EventIDColumns. Thus, it is necessary to free all event ids associated with the single inspiral.

The function `LALSortSnglInspiral()` and `XLALSortSnglInspiral()` sorts a list of single inspiral tables. The function simply calls `qsort` with the appropriate comparison function, `comparfunc`. It then ensures that the head of the sorted list is returned. There then follow several comparison functions for single inspiral tables. `LALCompareSnglInspiralByMass()` first compares the `mass1` entry of the two inspiral tables, returning 1 if the first mass is larger and -1 if the second is larger. In the case that the `mass1` fields are equal, a similar comparison is performed on `mass2`. If these also agree, 0 is returned. `LALCompareSnglInspiralByPsi()` compares the `Psi0` and `Psi3` fields in two single inspiral tables. The function is analogous to the mass comparison described above. `LALCompareSnglInspiralByTime` compares the end times of two single inspiral tables, returning 1 if the first time is larger, 0 if equal and -1 if the second time is larger.

`LALCompareSnglInspiral()` tests whether two single inspiral tables pass a coincidence test. The coincidence parameters are given by `params` which is a `SnglInspiralAccuracy` structure. It tests first that the `ifo` fields are different. If they are, it then tests for time and mass coincidence, where mass coincidence may be any one of `psi0` and `psi3`, `m1` and `m2`, `mchirp` and `eta`. Finally, if the test is on `m1` and `m2`, consistency of effective distances is also checked. If the two single inspiral tables pass coincidences the `params.match` is set to 1, otherwise it is set to zero.

`LALClusterSnglInspiralTable()` clusters single inspiral triggers within a time window `dtimeNS`. The triggers are compared either by `snr`, `snr_and_chisq` or `snrsq_over_chisq`. The " loudest" trigger, as determined by the selected algorithm, within each time window is returned.
LALTimeCutSingleInspiral() and XLALTimeCutSingleInspiral() takes in a linked list of single inspiral tables and returns only those which occur after the given startTime and before the endTime.

LALSNR CutSingleInspiral() and XLALSNR CutSingleInspiral() take in a linked list of single inspiral tables and returns only those triggers which have snr values above a specific snrCut.

XLALRsqCutSingleInspiral() performs the R-squared veto on a linked list of single inspiral tables. Triggers whose snr is less than rsqSnrMax and whose rsqVeto_duration is greater than rsqVetoThresh or (optional) whose snr is greater than rsqSnrMax and whose rsqVeto_duration is greater than rsqAboveSnrCoeff \times snr^{rsqAboveSnrPow}

XLAVetoSingleInspiral() takes in a linked list of single inspiral tables and a list of segments and returns only those triggers which do not lie in within the vetoSegs.

LALBCVCVetoSingleInspiral() takes in a linked list of single inspiral tables and returns only those triggers which have alphaF/SNR values below a specific threshold and alphaF value between alphaF-hi and alphaF-lo values. It is relevant for the BCVC or BCVU search only.

XLALalphaFCutSingleInspiral() takes in a linked list of single inspiral tables and returns only those triggers which have alphaF values below a specific alphaFcut. It is relevant for the BCV search only.

LALIfoCutSingleInspiral() scans through a linked list of single inspiral tables and returns those which are from the requested ifo. On input, eventHead is a pointer to the head of a linked list of single inspiral tables. On output, this list contains only single inspirals from the requested ifo. XLALIfoCutSingleInspiral() works similarly, although slightly differently. This function returns the list of events from the specified ifo, while on completion, eventHead contains the list of events from other ifos.

LALIfoCountSingleInspiral() scans through a linked list of single inspiral tables and counts the number which are from the requested IFO. This count is returned as numTrigs.

LALTimeSlideSingleInspiral() performs a time slide on the triggers contained in the triggerList. The time slide for each instrument is specified by slideTimes[LAL_NUMIFO]. If startTime and endTime are specified, then the time slide is performed on a ring. If the slide takes any trigger outside of the window [startTime,endTime], then the trigger is wrapped to be in this time window.

LALPlayTestSingleInspiral() and test whether single inspiral events occurred in playground or non-playground times. It then returns the requested subset of events which occurred in the times specified by dataType which must be one of playground_only, exclude_play or all_data.

LALCreateTrigBank() takes in a list of single inspiral tables and returns a template bank. The function tests whether a given template produced multiple triggers. If it did, only one copy of the template is retained. Triggers are tested for coincidence in m1 and m2 or psi0 and psi3.

LALIncaCoincidenceTest() performs a coincidence test between triggers from two interferometers. It tests pairs of events for both time and mass coincidence and returns two equal length lists of coincident events. Note that if an event in one detector is coincident with several events in the other detector, the output lists will contain several copies of this event.

LALTamaCoincidenceTest() also performs a coincidence test between triggers from two interferometers, but with a slightly different coincidence test. First, it locates all triggers in the second instrument which are coincident with triggers in the first instrument. Then, it clusters these triggers using the appropriate clusterchioce. Finally, it tests for mass coincidence between the first trigger and the clustered trigger from the second instrument.

Algorithm
None.

Uses
LALCalloc, LALFree, LALGPStoINT8, LALINT8NanoSecIsPlayground.

Notes
7.8.3 Module CoincInspiralUtils.c

Blah.

Prototypes

```c
void LALCreateTwoIFOCoincList(
    LALStatus *status,
    CoincInspiralTable **coincOutput,
    SnglInspiralTable *snglInput,
    InspiralAccuracyList *accuracyParams
)

void LALCreateNIFOCoincList(
    LALStatus *status,
    CoincInspiralTable **coincHead,
    InspiralAccuracyList *accuracyParams,
    INT4 N
)

void LALAddSnglInspiralToCoinc(
    LALStatus *status,
    CoincInspiralTable **coincPtr,
    SnglInspiralTable *snglInspiral
)

CoincInspiralTable * XLALAddSnglInspiralToCoinc(
    CoincInspiralTable *coincInspiral,
    SnglInspiralTable *snglInspiral
)

void LALSnglInspiralCoincTest(
    LALStatus *status,
    CoincInspiralTable *coincInspiral,
    SnglInspiralTable *snglInspiral,
    InspiralAccuracyList *accuracyParams
)

void XLALSnglInspiralCoincTest(
    CoincInspiralTable *coincInspiral,
    SnglInspiralTable *snglInspiral,
    InspiralAccuracyList *accuracyParams
)

void LALExtractSnglInspiralFromCoinc(
    LALStatus *status,
    SnglInspiralTable **snglPtr,
    CoincInspiralTable *coincInspiral,
    LIGOTimeGPS *gpsStartTime,
    INT4 slideNum
)
```
SnglInspiralTable *
XLALExtractSnglInspiralFromCoinc(
    CoincInspiralTable *coincInspiral,
    LIGOTimeGPS *gpsStartTime,
    INT4 slideNum
)

int
XLALRecreateCoincFromSngls(
    CoincInspiralTable **coincPtr,
    SnglInspiralTable *snglInspiral
)

int
XLALGenerateCoherentBank(
    SnglInspiralTable **coherentBank,
    CoincInspiralTable *coincInput,
    CHAR *ifos
)

void
XLALInspiralPsi0Psi3CutBCVC(
    CoincInspiralTable **coincInspiral
)

void
XLALInspiralIotaCutBCVC(
    CoincInspiralTable **coincInspiral,
    InspiralAccuracyList *accuracyParams
)

void
LALInspiralDistanceCutCleaning(
    LALStatus *status,
    CoincInspiralTable **coincInspiral,
    InspiralAccuracyList *accuracyParams,
    REAL4 snrThreshold,
    SummValueTable **summValueList,
    LALSegList *vetoSegsH1,
    LALSegList *vetoSegsH2
)

void
XLALInspiralDistanceCutBCVC(
    CoincInspiralTable **coincInspiral,
    InspiralAccuracyList *accuracyParams
)

void
XLALInspiralDistanceCut(
    CoincInspiralTable **coincInspiral,
    InspiralAccuracyList *accuracyParams
)

void
LALCoincCutSnglInspiral(
    LALStatus *status,
    SnglInspiralTable **eventHead
)
```c
int XLALClusterCoincInspiralTable(
    CoincInspiralTable **coincList,
    INT8 dtimeNS,
    CoincInspiralStatistic coincStat,
    CoincInspiralBittenLParams *bittenLParams
)

int XLALCompareCoincInspiralByTime(
    const void *a,
    const void *b
)

CoincInspiralTable *
XLALSortCoincInspiral(
    CoincInspiralTable *eventHead,
    int(*comparfunc) (const void *, const void *)
)

int XLALCoincInspiralIfos(
    CoincInspiralTable *coincInspiral,
    char *ifos
)

int XLALCoincInspiralIfosDiscard(
    CoincInspiralTable **coincHead,
    char *ifos
)

UINT8 XLALCoincInspiralIdNumber(
    CoincInspiralTable *coincInspiral
)

void XLALInspiralSNRCutBCV2(
    CoincInspiralTable **coincInspiral
)

INT4 XLALCountCoincInspiral( CoincInspiralTable *head )

CoincInspiralTable *
XLALStatCutCoincInspiral(
    CoincInspiralTable *eventHead,
    CoincInspiralStatistic coincStat,
    CoincInspiralBittenLParams *bittenLParams,
    REAL4 statCut
)

SnglInspiralTable *
XLALCompleteCoincInspiral(
    CoincInspiralTable *eventHead,
    int ifoList[LAL_NUM_IFO]
)

CoincInspiralTable *
XLALPlayTestCoincInspiral(
    CoincInspiralTable *eventHead,
    LALPlaygroundDataMask *dataType
)
```
CoincInspiralTable *
XLALMeanMassCut(
    CoincInspiralTable *eventHead,
    char *massCut,
    REAL4 massRangeLow,
    REAL4 massRangeHigh,
    REAL4 mass2RangeLow,
    REAL4 mass2RangeHigh
)

void
XLALPopulateAccuracyParams(
    InspiralAccuracyList *accuracyParams
)

void
XLALPopulateAccuracyParamsExt(
    InspiralAccuracyList *accuracyParams,
    const LIGOTimeGPS *gpstime,
    const REAL8 ra_deg,
    const REAL8 dec_deg
)

UINT8
XLALCoincRingdownIdNumber(
    CoincRingdownTable *coincRingdown
)

CoincRingdownTable *
XLALStatCutCoincRingdown(
    CoincRingdownTable *eventHead,
    CoincInspiralStatistic coincStat,
    CoincInspiralBittenLParams *bittenLParams,
    REAL4 statCut
)

void
LALCoincCutSnglInspiral(
    LALStatus *status,
    SnglInspiralTable **eventHead
)

Description
LALCreateTwoIFOCoincList() takes in a linked list of single inspiral tables and returns a list of two instrument coincidences. The coincidence requirements are given by the accuracyParams. When single inspirals from two different instruments are found to be coincident, the code creates a new coincInspiralTable and uses LALAddSnglInspiralToCoinc() to add the single inspirals to the coinc. The function returns coincOutput which is a pointer to the head of a linked list of CoincInspiralTables.

LALCreateNIFOCoincList() takes linked list of CoincInspiralTables, assumed to contain (N-1) ifo coincidences and creates all N ifo coincidences. Both the input and output list of CoincInspiralTables are passed as coincHead.

LALRemoveRepeatedCoincs() will remove any lower order coincidences if they are contained in a higher order coincidence. For example, if an H1-L1 double coincident trigger is also part of an H1-H2-L1 triple coincident trigger, the double coincident trigger will be removed. The head of the list of coincident triggers is passed and returned as coincHead.

XLALFreeCoincInspiral() LALFreeCoincInspiral() and free the memory associated to the CoincInspiralTable pointed to by coincPtr. This entails freeing the CoincInspiralTable as well as any eventIds which point to the coinc.
LALAddSnglInspiralToCoinc() and XLALAddSnglInspiralToCoinc() add a pointer to a single inspiral table to a coinc inspiral table. Upon entry, if coincPtr points to a NULL coinc inspiral table, the table is created before a pointer to the single inspiral table is added. Additionally, an eventId table is created for the single inspiral table. This points to both the single and coinc inspirals. If an eventId already exists for the single inspiral, another eventId table is added to the linked list. The linked list of.eventIds associated to a single inspiral table allow us to easily determine which coincident events each single is a part of.

LALSnglInspiralCoincTest() tests for coincidence between a single inspiral and a coinc inspiral. It works by testing for coincidence between each non-null entry in the coinc inspiral and the single. If all members of the coinc are found to be coincident with the single, the accuracyParams.match is set to 1, otherwise to 0.

LALExtractSnglInspiralFromCoinc() extracts the information from a linked list of coincInspiralTables and returns it as a linked list of snglInspiralTables. Thus, the output snglPtr is a pointer to a linked list of single inspiral tables. That list contains only single inspirals which are found in coincidence. In order to preserve the coincidence information, we assign to each coincident event an integer value. This is stored in the UINTS id field of the eventIdColumn of each single inspiral which forms part of the coincidence. The id is set equal to $10^9 \times \text{gpsStartTime} + 10^5 \times \text{slideNum} + \text{event number}$. We do not assign multiple id values to a given single inspiral table, but instead make multiple copies of the table, each with a unique id.

XLALRecreateCoincFromSngls() is used to recreate a list of coinc inspirals from a list of snglInspiralTables with populated eventIdColumn. The code searches for entries in snglInspiral which have the same numerical value of the id field in the eventIdColumn.

XLALGenerateCoherentBank() is used to generate a coherent bank from a list of coincInspiralTables. The coherent bank has the same mass parameters for each ifo. These are currently chosen as the mass parameters of the trigger in the coinc with the highest snr. If the ifos field is not NULL, then a template is generated for every ifo in ifos. If it is NULL then templates are only generated for those ifos which have triggers in the coinc.

XLALInspiralDistanceCut() is used to perform a distance cut between the triggers in a coincidence. The distance cut analyzes triggers from two different instruments. It determines which instrument was the most sensitive by comparing the sigmasq values of the two triggers, the instrument with the greatest range is designated ifo A, the other ifo B. It then discards and triggers for which

$$\frac{|\text{distB} - \text{distA}|}{\text{distA}} > \frac{\epsilon_{\text{snr}B}}{\text{snr}B} + \kappa_{\text{snr}B}$$

LALCoincCutSnglInspiral() extracts all single inspirals from a specific ifo which are in coinc inspirals. The output snglPtr is a pointer to a linked list of single inspiral tables. That list contains only single inspirals from the specified ifo which are found in coincidence.

XLALCountCoincInspiral() scans through a linked list of coincidence inspiral table and counts the number of events. This count is returned as numTrigs.

XLALCompleteCoincInspiral() scans through a linked list of coincidence inspirals and checks whether the coincs contain a trigger from every ifo in the ifoList with a non-zero value. If a trigger does not exist, it is added at the appropriate time for the appropriate ifo, with zero snr. The code returns a linked list of new single inspirals which were created in the process of completing the coincs.

Algorithm
None.

Uses
None.

Notes

Author: Fairhurst, S.

$Id: CoincInspiralUtils.c,v 1.62 2007/12/05 18:45:23 dkeppel Exp $
7.8.4 Module SimInspiralUtils.c

Provides a set of utilities for manipulating simInspiralTables.

Prototypes

```c
void XLALPlayTestSimInspiral(
    SimInspiralTable **eventHead,
    LALPlaygroundDataMask *dataType
)

int XLALSimInspiralInSearchedData(
    SimInspiralTable **eventHead,
    SearchSummaryTable **summList
)

int XLALSimInspiralChirpMassCut(
    SimInspiralTable **eventHead,
    REAL4 minChirpMass,
    REAL4 maxChirpMass
)

int XLALSimInspiralCompMassCut(
    SimInspiralTable **eventHead,
    REAL4 minCompMass,
    REAL4 maxCompMass,
    REAL4 minCompMass2,
    REAL4 maxCompMass2
)

int XLALSimInspiralTotalMassCut(
    SimInspiralTable **eventHead,
    REAL4 minTotalMass,
    REAL4 maxTotalMass
)

void LALGalacticInspiralParamsToSimInspiralTable(
    LALStatus *status,
    SimInspiralTable *output,
    GalacticInspiralParamStruc *input,
    RandomParams *params
)

void LALInspiralSiteTimeAndDist(
    LALStatus *status,
    SimInspiralTable *output,
    LALDetector *detector,
    LIGOTimeGPS *endTime,
    REAL4 *effDist,
    SkyPosition *skyPos
)
```
The function `LALPopulateSimInspiralSiteInfo()` calculates detector end time (`endTime`) and effective distance (`effDist`) for an inspiral signal from a specific location in the sky (`skyPos`) assumed to be given in equatorial coordinates. The detector end time is obtained by using `LALTimeDelayFromEarthCenter()`, while the effective distance requires calculation of the detector response, calculated using `LALComputeDetAMResponse()`.

The function `LALPopulateSimInspiralSiteInfo()` populates the end time and effective distance for each of the interferometer sites. The sky location (in equatorial coordinates) is assumed to be already contained in the input `SimInspiralTable`. The end time and effective distance for each site is calculated by calling `LALPopulateSimInspiralSiteTimeAndDist()` once for each of the detectors, and setting the `detector` appropriately.
Algorithm

None.

Uses

LALGetInspiralParams, LALGPSGtoGMST1, LALTimeDelayFromEarthCenter, LALAddFloatToGPS, LALComputeDetAMResponse.

Notes
7.9 Header CoincInspiralEllipsoid.h

Provides function definitions for performing inspiral coincidence analysis using error ellipsoids.

Synopsis

```c
#include <lal/CoincInspiralEllipsoid.h>

typedef struct tagTriggerErrorList {
    SnglInspiralTable *trigger;
    gsl_matrix *err_matrix;
    gsl_vector *position;
    struct tagTriggerErrorList *next;
} TriggerErrorList;
```

The `TriggerErrorList` is a linked list used within e-thinca. It contains pointers to the `SnglInspiralTable` for a given trigger, and its associated error matrix and position vector.
7.9.1 Module CoincInspiralEllipsoid.c

Blah.

Prototypes

```c
void LALCreateTwoIFOCoincListEllipsoid(
    LALStatus *status,
    CoincInspiralTable **coincOutput,
    SnglInspiralTable *snglInput,
    InspiralAccuracyList *accuracyParams
);

void XLALSnglInspiralCoincTestEllipsoid(
    CoincInspiralTable *coincInspiral,
    SnglInspiralTable *snglInspiral,
    InspiralAccuracyList *accuracyParams
);

INT2 XLALCompareInspiralsEllipsoid(
    TriggerErrorList *aPtr,
    TriggerErrorList *bPtr,
    fContactWorkSpace *workSpace,
    InspiralAccuracyList *params
);

REAL8 XLALCalculateEThincaParameter(
    SnglInspiralTable *table1,
    SnglInspiralTable *table2,
    InspiralAccuracyList* accuracyParams
);

REAL8 XLALEThincaParameterForInjection(
    SimInspiralTable *injection,
    SnglInspiralTable *trigger
);
```

Description

`LALCreateTwoIFOCoincListEllipsoid()` takes in a linked list of single inspiral tables and returns a list of two instrument coincidences. To determine coincidence, the triggers are modelled as ellipsoids in the parameter space. Triggers are deemed to be coincident if these ellipsoids are found to overlap. The ellipsoid scaling factor is given within the `accuracyParams` structure. When single inspirals from two different instruments are found to be coincident, the code creates a new `coincInspiralTable` and uses `LALAddSnglInspiralToCoinc()` to add the single inspirals to the coin. The function returns `coincOutput` which is a pointer to the head of a linked list of `CoincInspiralTable`s.

`XLALSnglInspiralCoincTestEllipsoid()` is used in the creation of multiple IFO coincident events. It is called by `LALCreateNIFOCoincList()` when the coincidence test is set to be ellipsoid. Unlike in other coincidence tests, coincidence here is determined by the use of event ids as opposed to calling the comparison function. This is because the test for ellipsoid overlap uses matrix inversions and function maximizations, which are potentially costly operations. If all members of the coinc are found to be coincident with the single, then `accuracyParams.match` is set to 1, otherwise to 0.

`XLALCompareInspiralsEllipsoid()` checks for the overlap of ellipsoids associated with two single inspiral tables. The ellipsoid scaling factor is provided by `accuracyParams`. If the ellipsoids are found to overlap, 1 is returned; otherwise 0 is returned.

`XLALCalculateEThincaParameter()` calculates the maximum e-thinca parameter between two single inspiral tables. It does this using a bisection method, and uses `XLALCompareInspiralsEllipsoid()` to
check for overlap. The maximum value for the e-thinca parameter is returned. If the two triggers do not overlap for an e-thinca parameter of 2.0, the triggers are not coincident, and an error is thrown.

XLALCalculateEThincaParameterForInjection() takes in a SnglInspiralTable and a SimInspiralTable, and returns the e-thinca parameter between the trigger and the injection. This amounts to calculating the square of the metric distance between the two points in $(t_C, \tau_0, \tau_3)$ space.

Algorithm
None.

Uses

Notes

Author: Craig Robinson
$Id: CoincInspiralEllipsoid.c,v 1.19 2007/06/08 14:41:57 bema Exp$
7.9.2 Module GetErrorMatrixFromSnglInspiral.c

Blah.

Prototypes

```c
#include "CoincInspiralEllipsoid.h"

gsl_matrix * XLALGetErrorMatrixFromSnglInspiral(SnglInspiralTable *event,
REALSE eMatch
);

gsl_vector * XLALGetPositionFromSnglInspiral(SnglInspiralTable *table);

int XLALSetTimeInPositionVector(gsl_vector *position,
REALSE time);
```

Description

**XLALGetErrorMatrixFromSnglInspiral()** takes in a **SnglInspiralTable**, and a value for the \( e \)-thinca parameter. It returns a **gsl_matrix** containing the the metric scaled appropriately for the given \( e \)-thinca parameter.

**XLALGetPositionFromSnglInspiral()** takes in a **SnglInspiralTable**, and returns the position vector associated with the trigger in \((t_C, \tau_0, \tau_3)\) space.

**XLALSetTimeInPositionVector()** sets the time co-ordinate in the given position vector to **time**. It returns zero upon successful completion.

Algorithm

None.

Uses

Notes
7.10 Header `TemplateBankGeneration.h`

Synopsis

```c
#include <lal/TemplateBankGeneration.h>
```

This header file includes all the necessary types and function prototypes for LALNDTemplateBank() and LALMakeTemplateBank(). NDTemplateBank() provides a general way to tile a parameter space with a constant metric (currently only for less than 12 dimensions). MakeTemplateBank() provides a general way for applications to generate a template bank with suitable I/O.
### Error codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>1</td>
<td>&quot;Unexpected NULL pointer to an input type&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants `TEMPLATEBANKGENERATIONH_E<name>`, and the status descriptions in `TEMPLATEBANKGENERATIONH_MSGE<name>`. The source code with these messages is in `TemplateBankGeneration.h` on line 1.142.
Types

typedef union {
    InspiralTmpltBankCInput *InspiralInput; /* Searches 100-199 */
    PulsarTmpltBankCInput *PulsarInput; /* Searches 200-299 */
    BurstTmpltBankCInput *BurstInput; /* Searches 300-399 */
} MakeTemplateBankInput;

typedef struct {
    /* InspiralCoarseBankIn Parameters */
    REAL4 mMin; /* minimum component mass */
    REAL4 mMax; /* maximum component mass */
    REAL8 MMax; /* Maximum total Mass */
    REAL4 psi0Min; /* minimum value of psi0 */
    REAL4 psi0Max; /* maximum value of psi0 */
    REAL4 psi3Min; /* minimum value of psi3 */
    REAL4 psi3Max; /* maximum value of psi3 */
    REAL4 alpha; /* BCV amplitude correction */
    INT4 numFcutTemplates; /* num tmpls in fcut direction */
    REAL4 mmCoarse; /* minimum requested match */
    REAL4 fLower; /* upper frequency cutoff */
    REAL4 fUpper; /* post-Newtonian order */
    REAL8 frequencySeries shf; /* used for PSD stuff */
    REAL8 tSampling; /* Sampling Rate */
    REAL8 etamin; /* Minimum Eta */
    REAL8 LowGM; /* ??????????????? */
    REAL8 HighGM; /* ??????????????? */
    Approximant approximant; /* approximation method */
    CoordinateSpace space; /* coordinate space used */
    InspiralBankMassRange massRange;
    INT4 ntiles; /* number of templates made */
    REAL4 mmFine; /* not implemented? */
    INT4 iflso; /* not implemented? */
} InspiralTmpltBankCInput;

/* InspiralTemplate Parameters */

/* InspiralSpinBank Additional Parameters */
    REAL4 mTwoMin; /* minimum mass smaller body */
    REAL4 mTwoMax; /* maximum mass smaller body */
    REAL4 betaMin; /* minimum beta spin parameter */
    REAL4 betaMax; /* maximum beta spin parameter */
} InspiralTmpltBankCInput; /* InspiralTmpltBankCInput */

typedef struct {
    INT4 placeholder;
} PulsarTmpltBankCInput; /* not used yet */

typedef struct {
    INT4 placeholder;
} BurstTmpltBankCInput; /* not used yet */

typedef enum {
    /* Binary Inspiral Searches 100-199 */
    BCVType,
    BCVSpinType,
PrecessingType,
/* Pulsar Searches 200-299 */
Pulsar,
/* Burst Searches 300-399 */
Burst
/* Other Searches 400-499 Etc... */
} TemplateBankType;

typedef struct {
  REAL4 minCoordinates[12];  /* Psi0, Psi3, etc. */
  REAL4 maxCoordinates[12];  /* Psi0, Psi3, etc. */
  REAL4 minParameters[12];  /* mass, eta, etc. */
  REAL4 maxParameters[12];  /* mass, eta, etc. */
  INT2 dimension;  /* 3D?, 4D? -> ND! */
  REAL4 mm;  /* mismatch */
  TemplateBankType type;  /* what is the search? */
  REAL8FrequencySeries* PSD;  /* Power Spec. Density */
  REAL4 f0;  /* Moment scaling freq */
} NDTemplateBankInput;

typedef struct NDTemplateBankOutput{
  REAL4 coordinateVals[12];
  REAL4 parameterVals[12];
  struct NDTemplateBankOutput *next;
} NDTemplateBankOutput;

Notes

- No notes yet.
7.10.1 Module **NDTemplateBank.c**

This module handles template bank generation for up to searches with <= 12 dimensional parameter spaces.

**Prototypes**

```c
LALNDTemplateBank( LALStatus *status,
                   NDTemplateBankInput *input,
                   NDTemplateBankFunctionPtrs *functionPtrs,
                   NDTemplateBankOutput **output)
```

**Description**

This module tiles up to a 12 dimensional space when given a metric and a function that determines the search region.

**Algorithm**

The algorithm first draws a rectilinear box in the primed coordinates which includes the distorted box, then steps through along the directions of the primed coordinates. At each point it tests if the point lies within the distorted box. If the point is inside the distorted box, the algorithm adds a template to the linked list. If not, it continues.

**Uses**

- `LALCalloc()`
- `LALFree()`

**Notes**

* Author: Hanna, C. R.
* $Id: NDTemplateBank.c,v 1.5 2007/06/08 14:41:57 bema Exp $
7.10.2 Module MakeTemplateBank.c

Prototypes

LALMakeTemplateBank(
    LALStatus *status,
    TemplateBankType *type,
    MakeTemplateBankInput *input,
    MetadataTable *table)

Description

Notes

- No notes yet.
7.11 Sequence Manipulation

This is a suite of functions for creating, destroying, and manipulating LAL sequences. For example `XLALCreateREAL4Sequence()` is available for creating sequences of REAL4 data.

7.11.1 Creation Functions

Name

`XLALCreate`*sequence type*, `LALCreate`*sequence type*

Synopsis

```c
#include <lal/Sequence.h>

COMPLEX8Sequence *XLALCreateCOMPLEX8Sequence ( size_t length );

COMPLEX16Sequence *XLALCreateCOMPLEX16Sequence ( size_t length );

REAL4Sequence *XLALCreateREAL4Sequence ( size_t length );

REAL8Sequence *XLALCreateREAL8Sequence ( size_t length );

INT2Sequence *XLALCreateINT2Sequence ( size_t length );

INT4Sequence *XLALCreateINT4Sequence ( size_t length );

INT8Sequence *XLALCreateINT8Sequence ( size_t length );

UINT2Sequence *XLALCreateUINT2Sequence ( size_t length );

UINT4Sequence *XLALCreateUINT4Sequence ( size_t length );

UINT8Sequence *XLALCreateUINT8Sequence ( size_t length );
```

Description

These functions create LAL sequences. The parameter `length` specifies the length of the desired sequence. The return value is a pointer to the newly created sequence or `NULL` on failure.
7.11.2 Destruction Functions

Name

`XLALDestroy`\texttt{sequencetype}(),

Synopsis

```c
#include <lal/Sequence.h>

void XLALDestroyCOMPLEX8Sequence (COMPLEX8Sequence *sequence);

void XLALDestroyCOMPLEX16Sequence (COMPLEX16Sequence *sequence);

void XLALDestroyREAL4Sequence (REAL4Sequence *sequence);

void XLALDestroyREAL8Sequence (REAL8Sequence *sequence);

void XLALDestroyINT2Sequence (INT2Sequence *sequence);

void XLALDestroyINT4Sequence (INT4Sequence *sequence);

void XLALDestroyINT8Sequence (INT8Sequence *sequence);

void XLALDestroyUINT2Sequence (UINT2Sequence *sequence);

void XLALDestroyUINT4Sequence (UINT4Sequence *sequence);

void XLALDestroyUINT8Sequence (UINT8Sequence *sequence);
```

Description

These functions free all memory associated with a LAL sequence. It is safe to pass \texttt{NULL} to these functions.

Author

Kipp Cannon <kipp@gravity.phys.uwm.edu>
7.11.3 Cutting Functions

Name

XLALCutsequence\texttt{type()}, XLALCopysequence\texttt{type()},

Synopsis

```c
#include <lal/Sequence.h>

COMPLEX8Sequence *XLALCutCOMPLEX8Sequence (COMPLEX8Sequence *sequence, size_t first, size_t length);

COMPLEX8Sequence *XLALCopyCOMPLEX8Sequence (COMPLEX8Sequence *sequence);

COMPLEX16Sequence *XLALCutCOMPLEX16Sequence (COMPLEX16Sequence *sequence, size_t first, size_t length);

COMPLEX16Sequence *XLALCopyCOMPLEX16Sequence (COMPLEX16Sequence *sequence);

REAL4Sequence *XLALCutREAL4Sequence (REAL4Sequence *sequence, size_t first, size_t length);

REAL4Sequence *XLALCopyREAL4Sequence (REAL4Sequence *sequence);

REAL8Sequence *XLALCutREAL8Sequence (REAL8Sequence *sequence, size_t first, size_t length);

REAL8Sequence *XLALCopyREAL8Sequence (REAL8Sequence *sequence);

INT2Sequence *XLALCutINT2Sequence (INT2Sequence *sequence, size_t first, size_t length);

INT2Sequence *XLALCopyINT2Sequence (INT2Sequence *sequence);
```
INT4Sequence *XLALCutINT4Sequence ( 
    INT4Sequence *sequence, 
    size_t first, 
    size_t length
);

INT4Sequence *XLALCopyINT4Sequence ( 
    INT4Sequence *sequence
);

INT8Sequence *XLALCutINT8Sequence ( 
    INT8Sequence *sequence, 
    size_t first, 
    size_t length
);

INT8Sequence *XLALCopyINT8Sequence ( 
    INT8Sequence *sequence
);

UINT2Sequence *XLALCutUINT2Sequence ( 
    UINT2Sequence *sequence, 
    size_t first, 
    size_t length
);

UINT2Sequence *XLALCopyUINT2Sequence ( 
    UINT2Sequence *sequence
);

UINT4Sequence *XLALCutUINT4Sequence ( 
    UINT4Sequence *sequence, 
    size_t first, 
    size_t length
);

UINT4Sequence *XLALCopyUINT4Sequence ( 
    UINT4Sequence *sequence
);

UINT8Sequence *XLALCutUINT8Sequence ( 
    UINT8Sequence *sequence, 
    size_t first, 
    size_t length
);

UINT8Sequence *XLALCopyUINT8Sequence ( 
    UINT8Sequence *sequence
);

Description
These functions create a new sequence by extracting a section of an existing sequence.

Author
Kipp Cannon <kipp@gravity.phys.uwm.edu>

7.11.4 Shifting Functions

Name
XLALShiftSequenceType();
Synopsis

#include <lal/Sequence.h>

void XLALShiftCOMPLEX8Sequence (COMPLEX8Sequence *sequence, int count);

void XLALShiftCOMPLEX16Sequence (COMPLEX16Sequence *sequence, int count);

void XLALShiftREAL4Sequence (REAL4Sequence *sequence, int count);

void XLALShiftREAL8Sequence (REAL8Sequence *sequence, int count);

void XLALShiftINT2Sequence (INT2Sequence *sequence, int count);

void XLALShiftINT4Sequence (INT4Sequence *sequence, int count);

void XLALShiftINT8Sequence (INT8Sequence *sequence, int count);

void XLALShiftUINT2Sequence (UINT2Sequence *sequence, int count);

void XLALShiftUINT4Sequence (UINT4Sequence *sequence, int count);

void XLALShiftUINT8Sequence (UINT8Sequence *sequence, int count);

Description

These functions shift the samples in a sequence, with zeros being placed in the space that is freed.

Author

Kipp Cannon <kipp@gravity.phys.uwm.edu>
7.11.5 Resizing Functions

Name
XLALResize$sequence$type(), XLALShrink$sequence$type(),

Synopsis

```c
#include <lal/Sequence.h>

COMPLEX8Sequence *XLALResizeCOMPLEX8Sequence (COMPLEX8Sequence *sequence, int first, size_t length);

COMPLEX8Sequence *XLALShrinkCOMPLEX8Sequence (COMPLEX8Sequence *sequence, size_t first, size_t length);

COMPLEX16Sequence *XLALResizeCOMPLEX16Sequence (COMPLEX16Sequence *sequence, int first, size_t length);

COMPLEX16Sequence *XLALShrinkCOMPLEX16Sequence (COMPLEX16Sequence *sequence, size_t first, size_t length);

REAL4Sequence *XLALResizeREAL4Sequence (REAL4Sequence *sequence, int first, size_t length);

REAL4Sequence *XLALShrinkREAL4Sequence (REAL4Sequence *sequence, size_t first, size_t length);

REAL8Sequence *XLALResizeREAL8Sequence (REAL8Sequence *sequence, int first, size_t length);

REAL8Sequence *XLALShrinkREAL8Sequence (REAL8Sequence *sequence, size_t first, size_t length);

INT2Sequence *XLALResizeINT2Sequence (INT2Sequence *sequence, int first, size_t length);

```
INT2Sequence *XLALShrinkINT2Sequence (  
    INT2Sequence *sequence,  
    size_t first,  
    size_t length  
);  

INT4Sequence *XLALResizeINT4Sequence (  
    INT4Sequence *sequence,  
    int first,  
    size_t length  
);  

INT4Sequence *XLALShrinkINT4Sequence (  
    INT4Sequence *sequence,  
    size_t first,  
    size_t length  
);  

INT8Sequence *XLALResizeINT8Sequence (  
    INT8Sequence *sequence,  
    int first,  
    size_t length  
);  

INT8Sequence *XLALShrinkINT8Sequence (  
    INT8Sequence *sequence,  
    size_t first,  
    size_t length  
);  

UINT2Sequence *XLALResizeUINT2Sequence (  
    UINT2Sequence *sequence,  
    int first,  
    size_t length  
);  

UINT2Sequence *XLALShrinkUINT2Sequence (  
    UINT2Sequence *sequence,  
    size_t first,  
    size_t length  
);  

UINT4Sequence *XLALResizeUINT4Sequence (  
    UINT4Sequence *sequence,  
    int first,  
    size_t length  
);  

UINT4Sequence *XLALShrinkUINT4Sequence (  
    UINT4Sequence *sequence,  
    size_t first,  
    size_t length  
);  

UINT8Sequence *XLALResizeUINT8Sequence (  
    UINT8Sequence *sequence,  
    int first,  
    size_t length  
);  

UINT8Sequence *XLALShrinkUINT8Sequence (  
    UINT8Sequence *sequence,  
    size_t first,  
    size_t length  
);
Description

The resize functions alter the size of an existing sequence. The sequence is adjusted to have the specified length, and that part of the original sequence starting at sample first is used to fill the new sequence. If first is negative, then the start of the new sequence is padded by that many samples. If part of the new sequence does not correspond to some part of the original sequence, then those samples are set to 0.

The shrink functions, originally, could only handle the special case in which the new sequence is wholly contained in the original sequence. Now the shrink functions are wrappers for the resize functions and are only retained for backwards compatibility.

Author
Kipp Cannon <kipp@gravity.phys.uwm.edu>

7.11.6 Summing Functions

Name
XLAL<datatype>Sum(), XLAL<datatype>SumSquares(), XLAL<sequencetype>Sum() XLAL<sequencetype>SumSquares()

Synopsis

```c
#include <lal/Sequence.h>

COMPLEX8 XLALCOMPLEX8Sum (  
    const COMPLEX8 *data,  
    size_t first,  
    size_t count  
);

REAL4 XLALCOMPLEX8SumSquares (  
    const COMPLEX8 *data,  
    size_t first,  
    size_t count  
);

COMPLEX8 XLALCOMPLEX8SequenceSum (  
    const COMPLEX8Sequence *sequence,  
    size_t first,  
    size_t count  
);

REAL4 XLALCOMPLEX8SequenceSumSquares (  
    const COMPLEX8Sequence *sequence,  
    size_t first,  
    size_t count  
);

COMPLEX16 XLALCOMPLEX16Sum (  
    const COMPLEX16 *data,  
    size_t first,  
    size_t count  
);

REAL8 XLALCOMPLEX16SumSquares (  
    const COMPLEX16 *data,  
    size_t first,  
    size_t count  
);
```
COMPLEX16 XLALCOMPLEX16SequenceSum (
    const COMPLEX16Sequence *sequence,
    size_t first,
    size_t count
);

REAL8 XLALCOMPLEX16SequenceSumSquares (
    const COMPLEX16Sequence *sequence,
    size_t first,
    size_t count
);

REAL4 XLALREAL4Sum (
    const REAL4 *data,
    size_t first,
    size_t count
);

REAL4 XLALREAL4SequenceSumSquares (
    const REAL4Sequence *sequence,
    size_t first,
    size_t count
);

REAL4 XLALREAL4SequenceSum (
    const REAL4Sequence *sequence,
    size_t first,
    size_t count
);

REAL8 XLALREAL8Sum (
    const REAL8 *data,
    size_t first,
    size_t count
);

REAL8 XLALREAL8SequenceSumSquares (
    const REAL8Sequence *sequence,
    size_t first,
    size_t count
);

REAL8 XLALREAL8SequenceSum (
    const REAL8Sequence *sequence,
    size_t first,
    size_t count
);

REAL8 XLALREAL8SequenceSumSquares (
    const REAL8Sequence *sequence,
    size_t first,
    size_t count
);
INT2 XLALINT2Sum (
    const INT2 *data,
    size_t first,
    size_t count
);

UINT2 XLALINT2SumSquares (
    const INT2 *data,
    size_t first,
    size_t count
);

INT2 XLALINT2SequenceSum (
    const INT2Sequence *sequence,
    size_t first,
    size_t count
);

UINT2 XLALINT2SequenceSumSquares (
    const INT2Sequence *sequence,
    size_t first,
    size_t count
);

INT4 XLALINT4Sum (
    const INT4 *data,
    size_t first,
    size_t count
);

UINT4 XLALINT4SumSquares (
    const INT4 *data,
    size_t first,
    size_t count
);

INT4 XLALINT4SequenceSum (
    const INT4Sequence *sequence,
    size_t first,
    size_t count
);

UINT4 XLALINT4SequenceSumSquares (
    const INT4Sequence *sequence,
    size_t first,
    size_t count
);

INT8 XLALINT8Sum (
    const INT8 *data,
    size_t first,
    size_t count
);

UINT8 XLALINT8SumSquares (
    const INT8 *data,
    size_t first,
    size_t count
);
INT8 XLALINT8SequenceSum (  
    const INT8Sequence *sequence,  
    size_t first,  
    size_t count  
);

UINT8 XLALINT8SequenceSumSquares (  
    const INT8Sequence *sequence,  
    size_t first,  
    size_t count  
);

UINT2 XLALUINT2Sum (  
    const UINT2 *data,  
    size_t first,  
    size_t count  
);

UINT2 XLALUINT2SumSquares (  
    const UINT2 *data,  
    size_t first,  
    size_t count  
);

UINT2 XLALUINT2SequenceSum (  
    const UINT2Sequence *sequence,  
    size_t first,  
    size_t count  
);

UINT2 XLALUINT2SequenceSumSquares (  
    const UINT2Sequence *sequence,  
    size_t first,  
    size_t count  
);

UINT4 XLALUINT4Sum (  
    const UINT4 *data,  
    size_t first,  
    size_t count  
);

UINT4 XLALUINT4SumSquares (  
    const UINT4 *data,  
    size_t first,  
    size_t count  
);

UINT4 XLALUINT4SequenceSum (  
    const UINT4Sequence *sequence,  
    size_t first,  
    size_t count  
);

UINT4 XLALUINT4SequenceSumSquares (  
    const UINT4Sequence *sequence,  
    size_t first,  
    size_t count  
);
UINT8 XLALUINT8Sum (const UINT8 *data, size_t first, size_t count);

UINT8 XLALUINT8SumSquares (const UINT8 *data, size_t first, size_t count);

UINT8 XLALUINT8SequenceSum (const UINT8Sequence *sequence, size_t first, size_t count);

UINT8 XLALUINT8SequenceSumSquares (const UINT8Sequence *sequence, size_t first, size_t count);

Description

The XLAL datatype Sum() and XLAL datatype SumSquares() functions sum the elements and squares of the elements, respectively, in an array.

The XLAL sequence type Sum() and XLAL sequence type SumSquares() functions sum the elements and the squares of the elements, respectively in a sequence. Bounds checking is performed.

In all cases, the return value is the sum, and these functions cannot fail. In the case of the sequence-related functions, if the sum extends beyond the bounds of the sequence, then the missing values are assumed to be 0.

Bugs

Because the LAL library must conform to the C89 specification, aggregate data types cannot be returned from functions so the COMPLEX8 and COMPLEX16 versions of the sum functions (not sum-of-squares functions) are commented out at this time.

Author

Kipp Cannon <kipp@gravity.phys.uwm.edu>

7.11.7 Conjugate Functions

Name

XLAL datatype Conjugate()

Synopsis

#include <lal/Sequence.h>

COMPLEX8Sequence *XLALConjugateCOMPLEX8Sequence (COMPLEX8Sequence *series);

COMPLEX16Sequence *XLALConjugateCOMPLEX16Sequence (COMPLEX16Sequence *series);
Description

These functions replace a sequence with its complex conjugate.

Author

Kipp Cannon <kipp@gravity.phys.uwm.edu>
7.12 Frequency Series Manipulation

This is a suite of functions for creating, destroying, and manipulating LAL frequency series. One pair of functions (the XLAL version and its LAL counterpart) is available for each method and frequency series type. For example XLALCreateREAL4FrequencySeries() is available for creating frequency series of REAL4 data, and the LAL-stype wrapper LALCreateREAL4FrequencySeries() is provided which is equivalent to the XLAL version in all respects except that it adheres to the LAL calling conventions (eg. it takes a LALStatus pointer as its first argument, its return type is void, etc.).

7.12.1 Creation Functions

Name

XLALCreatefrequencyseriestype(), LALCreatefrequencyseriestype()

Synopsis

#include <lal/FrequencySeries.h>

COMPLEX8FrequencySeries *XLALCreateCOMPLEX8FrequencySeries (const CHAR *name, const LIGOTimeGPS *epoch, REAL8 f0, REAL8 deltaF, const LALUnit *sampleUnits, size_t length);

void LALCreateCOMPLEX8FrequencySeries (LALStatus *status, COMPLEX8FrequencySeries **output, const CHAR *name, LIGOTimeGPS epoch, REAL8 f0, REAL8 deltaF, LALUnit sampleUnits, size_t length);

COMPLEX16FrequencySeries *XLALCreateCOMPLEX16FrequencySeries (const CHAR *name, const LIGOTimeGPS *epoch, REAL8 f0, REAL8 deltaF, const LALUnit *sampleUnits, size_t length);

void LALCreateCOMPLEX16FrequencySeries (LALStatus *status, COMPLEX16FrequencySeries **output, const CHAR *name, LIGOTimeGPS epoch, REAL8 f0, REAL8 deltaF, LALUnit sampleUnits, size_t length);
REAL4FrequencySeries *XLALCreateREAL4FrequencySeries (  
    const CHAR *name,
    const LIGOTimeGPS *epoch,
    REAL8 f0,
    REAL8 deltaF,
    const LALUnit *sampleUnits,
    size_t length
);

void LALCreateREAL4FrequencySeries (  
    LALStatus *status,
    REAL4FrequencySeries **output,
    const CHAR *name,
    LIGOTimeGPS epoch,
    REAL8 f0,
    REAL8 deltaF,
    LALUnit sampleUnits,
    size_t length
);

REAL8FrequencySeries *XLALCreateREAL8FrequencySeries (  
    const CHAR *name,
    const LIGOTimeGPS *epoch,
    REAL8 f0,
    REAL8 deltaF,
    const LALUnit *sampleUnits,
    size_t length
);

void LALCreateREAL8FrequencySeries (  
    LALStatus *status,
    REAL8FrequencySeries **output,
    const CHAR *name,
    LIGOTimeGPS epoch,
    REAL8 f0,
    REAL8 deltaF,
    LALUnit sampleUnits,
    size_t length
);

INT2FrequencySeries *XLALCreateINT2FrequencySeries (  
    const CHAR *name,
    const LIGOTimeGPS *epoch,
    REAL8 f0,
    REAL8 deltaF,
    const LALUnit *sampleUnits,
    size_t length
);

void LALCreateINT2FrequencySeries (  
    LALStatus *status,
    INT2FrequencySeries **output,
    const CHAR *name,
    LIGOTimeGPS epoch,
    REAL8 f0,
    REAL8 deltaF,
    LALUnit sampleUnits,
    size_t length
);
INT4FrequencySeries *XLALCreateINT4FrequencySeries (  
    const CHAR *name,  
    const LIGOTimeGPS *epoch,  
    REAL8 f0,  
    REAL8 deltaF,  
    const LALUnit *sampleUnits,  
    size_t length  
);

void LALCreateINT4FrequencySeries (  
    LALStatus *status,  
    INT4FrequencySeries **output,  
    const CHAR *name,  
    LIGOTimeGPS epoch,  
    REAL8 f0,  
    REAL8 deltaF,  
    LALUnit sampleUnits,  
    size_t length  
);

INT8FrequencySeries *XLALCreateINT8FrequencySeries (  
    const CHAR *name,  
    const LIGOTimeGPS *epoch,  
    REAL8 f0,  
    REAL8 deltaF,  
    const LALUnit *sampleUnits,  
    size_t length  
);

void LALCreateINT8FrequencySeries (  
    LALStatus *status,  
    INT8FrequencySeries **output,  
    const CHAR *name,  
    LIGOTimeGPS epoch,  
    REAL8 f0,  
    REAL8 deltaF,  
    LALUnit sampleUnits,  
    size_t length  
);

UINT2FrequencySeries *XLALCreateUINT2FrequencySeries (  
    const CHAR *name,  
    const LIGOTimeGPS *epoch,  
    REAL8 f0,  
    REAL8 deltaF,  
    const LALUnit *sampleUnits,  
    size_t length  
);

void LALCreateUINT2FrequencySeries (  
    LALStatus *status,  
    UINT2FrequencySeries **output,  
    const CHAR *name,  
    LIGOTimeGPS epoch,  
    REAL8 f0,  
    REAL8 deltaF,  
    LALUnit sampleUnits,  
    size_t length  
);
These functions create LAL frequency series. An XLAL function returns a pointer to the newly created series or NULL on failure. The LAL counterpart accepts the address of a pointer which it fills with the address of the newly created series or NULL on failure. Additionally, the LAL wrapper provides standard LAL-style error checking via a LALStatus pointer.

Author

Kipp Cannon <kipp@gravity.phys.uwm.edu>

7.12.2 Destruction Functions

Name

XLALDestroyfrequencyseriestype(), LALDestroyfrequencyseriestype()
Synopsis

#include <lal/FrequencySeries.h>

void XLALDestroyCOMPLEX8FrequencySeries (COMPLEX8FrequencySeries *series);

void LALDestroyCOMPLEX8FrequencySeries (LALStatus *status,
                                           COMPLEX8FrequencySeries *series);

void XLALDestroyCOMPLEX16FrequencySeries (COMPLEX16FrequencySeries *series);

void LALDestroyCOMPLEX16FrequencySeries (LALStatus *status,
                                           COMPLEX16FrequencySeries *series);

void XLALDestroyREAL4FrequencySeries (REAL4FrequencySeries *series);

void LALDestroyREAL4FrequencySeries (LALStatus *status,
                                       REAL4FrequencySeries *series);

void XLALDestroyREAL8FrequencySeries (REAL8FrequencySeries *series);

void LALDestroyREAL8FrequencySeries (LALStatus *status,
                                       REAL8FrequencySeries *series);

void XLALDestroyINT2FrequencySeries (INT2FrequencySeries *series);

void LALDestroyINT2FrequencySeries (LALStatus *status,
                                     INT2FrequencySeries *series);

void XLALDestroyINT4FrequencySeries (INT4FrequencySeries *series);

void LALDestroyINT4FrequencySeries (LALStatus *status,
                                    INT4FrequencySeries *series);

void XLALDestroyINT8FrequencySeries (INT8FrequencySeries *series);
void LALDestroyINT8FrequencySeries (  
    LALStatus *status,  
    INT8FrequencySeries *series  
);  

void XLALDestroyUINT2FrequencySeries (  
    UINT2FrequencySeries *series  
);  

void LALDestroyUINT2FrequencySeries (  
    LALStatus *status,  
    UINT2FrequencySeries *series  
);  

void XLALDestroyUINT4FrequencySeries (  
    UINT4FrequencySeries *series  
);  

void LALDestroyUINT4FrequencySeries (  
    LALStatus *status,  
    UINT4FrequencySeries *series  
);  

void XLALDestroyUINT8FrequencySeries (  
    UINT8FrequencySeries *series  
);  

void LALDestroyUINT8FrequencySeries (  
    LALStatus *status,  
    UINT8FrequencySeries *series  
);  

Description  
These functions free all memory associated with a LAL frequency series. It is safe to pass NULL to these functions.

Author  
Kipp Cannon <kipp@gravity.phys.uwm.edu>

7.12.3 Cutting Functions  
Name  
XLALCut<frequencyseriestype>();

Synopsis  
#include <lal/FrequencySeries.h>  

COMPLEX8FrequencySeries *XLALCutCOMPLEX8FrequencySeries (  
    const COMPLEX8FrequencySeries *series,  
    size_t first,  
    size_t length  
);  

COMPLEX16FrequencySeries *XLALCutCOMPLEX16FrequencySeries (  
    const COMPLEX16FrequencySeries *series,  
    size_t first,  
    size_t length  
);
REAL4FrequencySeries *XLALCutREAL4FrequencySeries (  
    const REAL4FrequencySeries *series,  
    size_t first,  
    size_t length 
);  

REAL8FrequencySeries *XLALCutREAL8FrequencySeries (  
    const REAL8FrequencySeries *series,  
    size_t first,  
    size_t length 
);  

INT2FrequencySeries *XLALCutINT2FrequencySeries (  
    const INT2FrequencySeries *series,  
    size_t first,  
    size_t length 
);  

INT4FrequencySeries *XLALCutINT4FrequencySeries (  
    const INT4FrequencySeries *series,  
    size_t first,  
    size_t length 
);  

INT8FrequencySeries *XLALCutINT8FrequencySeries (  
    const INT8FrequencySeries *series,  
    size_t first,  
    size_t length 
);  

UINT2FrequencySeries *XLALCutUINT2FrequencySeries (  
    const UINT2FrequencySeries *series,  
    size_t first,  
    size_t length 
);  

UINT4FrequencySeries *XLALCutUINT4FrequencySeries (  
    const UINT4FrequencySeries *series,  
    size_t first,  
    size_t length 
);  

UINT8FrequencySeries *XLALCutUINT8FrequencySeries (  
    const UINT8FrequencySeries *series,  
    size_t first,  
    size_t length 
);  

Description
These functions create a new frequency series by extracting a section of an existing frequency series.

Author
Kipp Cannon <kipp@gravity.phys.uwm.edu>

7.12.4 Resizing Functions
Name
XLALResize<frequencyseriestype>(), XLALShrink<frequencyseriestype>(), LALShrink<frequencyseriestype>()
Synopsis

#include <lal/FrequencySeries.h>

COMPLEX8FrequencySeries *XLALResizeCOMPLEX8FrequencySeries (COMPLEX8FrequencySeries *series, int first, size_t length);

COMPLEX8FrequencySeries *XLALShrinkCOMPLEX8FrequencySeries (COMPLEX8FrequencySeries *series, size_t first, size_t length);

COMPLEX16FrequencySeries *XLALResizeCOMPLEX16FrequencySeries (COMPLEX16FrequencySeries *series, int first, size_t length);

COMPLEX16FrequencySeries *XLALShrinkCOMPLEX16FrequencySeries (COMPLEX16FrequencySeries *series, size_t first, size_t length);

REAL4FrequencySeries *XLALResizeREAL4FrequencySeries (REAL4FrequencySeries *series, int first, size_t length);

REAL4FrequencySeries *XLALShrinkREAL4FrequencySeries (REAL4FrequencySeries *series, size_t first, size_t length);

REAL8FrequencySeries *XLALResizeREAL8FrequencySeries (REAL8FrequencySeries *series, int first, size_t length);

REAL8FrequencySeries *XLALShrinkREAL8FrequencySeries (REAL8FrequencySeries *series, size_t first, size_t length);

INT2FrequencySeries *XLALResizeINT2FrequencySeries (INT2FrequencySeries *series, int first, size_t length);

INT2FrequencySeries *XLALShrinkINT2FrequencySeries (INT2FrequencySeries *series, size_t first, size_t length);
INT4FrequencySeries *XLALResizeINT4FrequencySeries (  
    INT4FrequencySeries *series,  
    int first,  
    size_t length  
);  

INT4FrequencySeries *XLALShrinkINT4FrequencySeries (  
    INT4FrequencySeries *series,  
    size_t first,  
    size_t length  
);  

INT8FrequencySeries *XLALResizeINT8FrequencySeries (  
    INT8FrequencySeries *series,  
    int first,  
    size_t length  
);  

INT8FrequencySeries *XLALShrinkINT8FrequencySeries (  
    INT8FrequencySeries *series,  
    size_t first,  
    size_t length  
);  

UINT2FrequencySeries *XLALResizeUINT2FrequencySeries (  
    UINT2FrequencySeries *series,  
    int first,  
    size_t length  
);  

UINT2FrequencySeries *XLALShrinkUINT2FrequencySeries (  
    UINT2FrequencySeries *series,  
    size_t first,  
    size_t length  
);  

UINT4FrequencySeries *XLALResizeUINT4FrequencySeries (  
    UINT4FrequencySeries *series,  
    int first,  
    size_t length  
);  

UINT4FrequencySeries *XLALShrinkUINT4FrequencySeries (  
    UINT4FrequencySeries *series,  
    size_t first,  
    size_t length  
);  

UINT8FrequencySeries *XLALResizeUINT8FrequencySeries (  
    UINT8FrequencySeries *series,  
    int first,  
    size_t length  
);  

UINT8FrequencySeries *XLALShrinkUINT8FrequencySeries (  
    UINT8FrequencySeries *series,  
    size_t first,  
    size_t length  
);
**Description**

These functions resize an existing frequency series. The new frequency series will have the given length, and its contents will consist of that part of the original time series that started at sample \( \text{first} \). If \( \text{first} \) is negative, then the new time series is padded at the start by that many samples. The frequency series' heterodyne frequency, \( f_0 \), is adjusted appropriately.

The “Shrink” functions accept non-negative values for the parameter \( \text{first} \), and are retained only for historical purposes. New code should use the “Resize” variants.

**Author**

Kipp Cannon <kipp@gravity.phys.uwm.edu>

### 7.12.5 Addition Functions

**Name**

\texttt{XLALAdd\_frequencyseriestype()}

**Synopsis**

```c
#include <lal/FrequencySeries.h>

COMPLEX8FrequencySeries *XLALAddCOMPLEX8FrequencySeries (  
    COMPLEX8FrequencySeries *arg1,  
    const COMPLEX8FrequencySeries *arg2
);

COMPLEX16FrequencySeries *XLALAddCOMPLEX16FrequencySeries (  
    COMPLEX16FrequencySeries *arg1,  
    const COMPLEX16FrequencySeries *arg2
);

REAL4FrequencySeries *XLALAddREAL4FrequencySeries (  
    REAL4FrequencySeries *arg1,  
    const REAL4FrequencySeries *arg2
);

REAL8FrequencySeries *XLALAddREAL8FrequencySeries (  
    REAL8FrequencySeries *arg1,  
    const REAL8FrequencySeries *arg2
);

INT2FrequencySeries *XLALAddINT2FrequencySeries (  
    INT2FrequencySeries *arg1,  
    const INT2FrequencySeries *arg2
);

INT4FrequencySeries *XLALAddINT4FrequencySeries (  
    INT4FrequencySeries *arg1,  
    const INT4FrequencySeries *arg2
);

INT8FrequencySeries *XLALAddINT8FrequencySeries (  
    INT8FrequencySeries *arg1,  
    const INT8FrequencySeries *arg2
);

UINT2FrequencySeries *XLALAddUINT2FrequencySeries (  
    UINT2FrequencySeries *arg1,  
    const UINT2FrequencySeries *arg2
);
```

---

1. 109 FrequencySeries.h
2. 1.316 FrequencySeries.h
3. 1.319 FrequencySeries.h
4. 1.417 FrequencySeries.h
5. 1.515 FrequencySeries.h
6. 1.613 FrequencySeries.h
7. 1.711 FrequencySeries.h
8. 1.819 FrequencySeries.h
**Description**

These functions add the second argument to the first argument, returning a pointer to the first argument on success or NULL on failure. The two series must have the same epoch and frequency resolution, and have units that differ only by a dimensionless factor.

**Author**

Kipp Cannon <kipp@gravity.phys.uwm.edu>

### 7.12.6 Conjugation Functions

**Name**

`XLALConjugate` for `frequencyseriestype()`

**Synopsis**

```
#include <lal/FrequencySeries.h>

COMPLEX8FrequencySeries *XLALConjugateCOMPLEX8FrequencySeries (COMPLEX8FrequencySeries *series);

COMPLEX16FrequencySeries *XLALConjugateCOMPLEX16FrequencySeries (COMPLEX16FrequencySeries *series);
```

**Description**

These functions replace a frequency series with its complex conjugate.

**Author**

Kipp Cannon <kipp@gravity.phys.uwm.edu>

### 7.12.7 Multiplication Functions

**Name**

`XLALMultiply` for `frequencyseriestype()`
Synopsis

```c
#include <lal/FrequencySeries.h>

COMPLEX8FrequencySeries *XLALMultiplyCOMPLEX8FrequencySeries (  
    COMPLEX8FrequencySeries *arg1,  
    const COMPLEX8FrequencySeries *arg2  
);

COMPLEX16FrequencySeries *XLALMultiplyCOMPLEX16FrequencySeries (  
    COMPLEX16FrequencySeries *arg1,  
    const COMPLEX16FrequencySeries *arg2  
);

REAL4FrequencySeries *XLALMultiplyREAL4FrequencySeries (  
    REAL4FrequencySeries *arg1,  
    const REAL4FrequencySeries *arg2  
);

REAL8FrequencySeries *XLALMultiplyREAL8FrequencySeries (  
    REAL8FrequencySeries *arg1,  
    const REAL8FrequencySeries *arg2  
);

INT2FrequencySeries *XLALMultiplyINT2FrequencySeries (  
    INT2FrequencySeries *arg1,  
    const INT2FrequencySeries *arg2  
);

INT4FrequencySeries *XLALMultiplyINT4FrequencySeries (  
    INT4FrequencySeries *arg1,  
    const INT4FrequencySeries *arg2  
);

INT8FrequencySeries *XLALMultiplyINT8FrequencySeries (  
    INT8FrequencySeries *arg1,  
    const INT8FrequencySeries *arg2  
);

UINT2FrequencySeries *XLALMultiplyUINT2FrequencySeries (  
    UINT2FrequencySeries *arg1,  
    const UINT2FrequencySeries *arg2  
);

UINT4FrequencySeries *XLALMultiplyUINT4FrequencySeries (  
    UINT4FrequencySeries *arg1,  
    const UINT4FrequencySeries *arg2  
);

UINT8FrequencySeries *XLALMultiplyUINT8FrequencySeries (  
    UINT8FrequencySeries *arg1,  
    const UINT8FrequencySeries *arg2  
);
```

Description

These functions multiply the first argument by the second argument, returning a pointer to the first argument on success or NULL on failure. The two series must have the same epoch and frequency resolution, and have units that differ only by a dimensionless factor.

Author

Kipp Cannon <kipp@gravity.phys.uwm.edu>
7.13 Time Series Manipulation

This is a suite of functions for creating, destroying, and manipulating LAL time series. One pair of functions (the XLAL version and its LAL counterpart) is available for each method and series type. For example `XLALCreateREAL4TimeSeries()` is available for creating time series of REAL4 data, and the LAL-stype wrapper `LALCreateREAL4TimeSeries()` is provided which is equivalent to the XLAL version in all respects except that it adheres to the LAL calling conventions (e.g. it takes a `LALStatus` pointer as its first argument, its return type is `void`, etc.).

7.13.1 Creation Functions

Name

`XLALCreate timeseriesType()`, `LALCreate timeseriesType()`

Synopsis

```c
#include <lal/TimeSeries.h>

COMPLEX8TimeSeries *XLALCreateCOMPLEX8TimeSeries (  
    const CHAR *name,  
    const LIGOTimeGPS *epoch,  
    REAL8 f0,  
    REAL8 deltaT,  
    const LALUnit *sampleUnits,  
    size_t length
);

void LALCreateCOMPLEX8TimeSeries (  
    LALStatus *status,  
    COMPLEX8TimeSeries **output,  
    const CHAR *name,  
    LIGOTimeGPS epoch,  
    REAL8 f0,  
    REAL8 deltaT,  
    LALUnit sampleUnits,  
    size_t length
);

COMPLEX16TimeSeries *XLALCreateCOMPLEX16TimeSeries (  
    const CHAR *name,  
    const LIGOTimeGPS *epoch,  
    REAL8 f0,  
    REAL8 deltaT,  
    const LALUnit *sampleUnits,  
    size_t length
);

void LALCreateCOMPLEX16TimeSeries (  
    LALStatus *status,  
    COMPLEX16TimeSeries **output,  
    const CHAR *name,  
    LIGOTimeGPS epoch,  
    REAL8 f0,  
    REAL8 deltaT,  
    LALUnit sampleUnits,  
    size_t length
);
```
REAL4TimeSeries *XLALCreateREAL4TimeSeries( 
    const CHAR *name,
    const LIGOTimeGPS *epoch,
    REAL8 f0,
    REAL8 deltaT,
    const LALUnit *sampleUnits,
    size_t length
);

void LALCreateREAL4TimeSeries ( 
    LALStatus *status,
    REAL4TimeSeries **output,
    const CHAR *name,
    LIGOTimeGPS epoch,
    REAL8 f0,
    REAL8 deltaT,
    LALUnit sampleUnits,
    size_t length
);

REAL8TimeSeries *XLALCreateREAL8TimeSeries ( 
    const CHAR *name,
    const LIGOTimeGPS *epoch,
    REAL8 f0,
    REAL8 deltaT,
    const LALUnit *sampleUnits,
    size_t length
);

void LALCreateREAL8TimeSeries ( 
    LALStatus *status,
    REAL8TimeSeries **output,
    const CHAR *name,
    LIGOTimeGPS epoch,
    REAL8 f0,
    REAL8 deltaT,
    LALUnit sampleUnits,
    size_t length
);

INT2TimeSeries *XLALCreateINT2TimeSeries ( 
    const CHAR *name,
    const LIGOTimeGPS *epoch,
    REAL8 f0,
    REAL8 deltaT,
    const LALUnit *sampleUnits,
    size_t length
);

void LALCreateINT2TimeSeries ( 
    LALStatus *status,
    INT2TimeSeries **output,
    const CHAR *name,
    LIGOTimeGPS epoch,
    REAL8 f0,
    REAL8 deltaT,
    LALUnit sampleUnits,
    size_t length
);
INT4TimeSeries *XLALCreateINT4TimeSeries (  
    const CHAR *name,  
    const LIGOTimeGPS *epoch,  
    REAL8 f0,  
    REAL8 deltaT,  
    const LALUnit *sampleUnits,  
    size_t length  
);

void LALCreateINT4TimeSeries (  
    LALStatus *status,  
    INT4TimeSeries **output,  
    const CHAR *name,  
    LIGOTimeGPS epoch,  
    REAL8 f0,  
    REAL8 deltaT,  
    LALUnit sampleUnits,  
    size_t length  
);

INT8TimeSeries *XLALCreateINT8TimeSeries (  
    const CHAR *name,  
    const LIGOTimeGPS *epoch,  
    REAL8 f0,  
    REAL8 deltaT,  
    const LALUnit *sampleUnits,  
    size_t length  
);

void LALCreateINT8TimeSeries (  
    LALStatus *status,  
    INT8TimeSeries **output,  
    const CHAR *name,  
    LIGOTimeGPS epoch,  
    REAL8 f0,  
    REAL8 deltaT,  
    LALUnit sampleUnits,  
    size_t length  
);

UINT2TimeSeries *XLALCreateUINT2TimeSeries (  
    const CHAR *name,  
    const LIGOTimeGPS *epoch,  
    REAL8 f0,  
    REAL8 deltaT,  
    const LALUnit *sampleUnits,  
    size_t length  
);

void LALCreateUINT2TimeSeries (  
    LALStatus *status,  
    UINT2TimeSeries **output,  
    const CHAR *name,  
    LIGOTimeGPS epoch,  
    REAL8 f0,  
    REAL8 deltaT,  
    LALUnit sampleUnits,  
    size_t length  
);
UINT4TimeSeries *XLALCreateUINT4TimeSeries (
    const CHAR *name,
    const LIGOTimeGPS *epoch,
    REAL8 f0,
    REAL8 deltaT,
    const LALUnit *sampleUnits,
    size_t length
);

void LALCreateUINT4TimeSeries (
    LALStatus *status,
    UINT4TimeSeries **output,
    const CHAR *name,
    LIGOTimeGPS epoch,
    REAL8 f0,
    REAL8 deltaT,
    LALUnit sampleUnits,
    size_t length
);

UINT8TimeSeries *XLALCreateUINT8TimeSeries (
    const CHAR *name,
    const LIGOTimeGPS *epoch,
    REAL8 f0,
    REAL8 deltaT,
    const LALUnit *sampleUnits,
    size_t length
);

void LALCreateUINT8TimeSeries (
    LALStatus *status,
    UINT8TimeSeries **output,
    const CHAR *name,
    LIGOTimeGPS epoch,
    REAL8 f0,
    REAL8 deltaT,
    LALUnit sampleUnits,
    size_t length
);

Description

These functions create LAL frequency series. An XLAL function returns a pointer to the newly created series or NULL on failure. The LAL counterpart accepts the address of a pointer which it fills with the address of the newly created series or NULL on failure. Additionally, the LAL wrapper provides standard LAL-style error checking via a LALStatus pointer.

Author

Kipp Cannon <kipp@gravity.phys.uwm.edu>

7.13.2 Destruction Functions

Name

XLALDestroytimeseriesType(), LALDestroytimeseriesType()
Synopsis

#include <lal/TimeSeries.h>

void XLALDestroyCOMPLEX8TimeSeries (COMPLEX8TimeSeries *series);

void LALDestroyCOMPLEX8TimeSeries (LALStatus *status, COMPLEX8TimeSeries *series);

void XLALDestroyCOMPLEX16TimeSeries (COMPLEX16TimeSeries *series);

void LALDestroyCOMPLEX16TimeSeries (LALStatus *status, COMPLEX16TimeSeries *series);

void XLALDestroyREAL4TimeSeries (REAL4TimeSeries *series);

void LALDestroyREAL4TimeSeries (LALStatus *status, REAL4TimeSeries *series);

void XLALDestroyREAL8TimeSeries (REAL8TimeSeries *series);

void LALDestroyREAL8TimeSeries (LALStatus *status, REAL8TimeSeries *series);

void XLALDestroyINT2TimeSeries (INT2TimeSeries *series);

void LALDestroyINT2TimeSeries (LALStatus *status, INT2TimeSeries *series);

void XLALDestroyINT4TimeSeries (INT4TimeSeries *series);

void LALDestroyINT4TimeSeries (LALStatus *status, INT4TimeSeries *series);

void XLALDestroyINT8TimeSeries (INT8TimeSeries *series);
Description
These functions free all memory associated with a LAL time series. It is safe to pass NULL to these functions.

Author
Kipp Cannon <kipp@gravity.phys.uwm.edu>

7.13.3 Cutting Functions

Name
XLALCut\textit{timeseriestype()}\texttt{()}, LALCut\textit{timeseriestype()}\texttt{()}

Synopsis
\begin{verbatim}
#include <lal/TimeSeries.h>

COMPLEX8TimeSeries *XLALCutCOMPLEX8TimeSeries (  
    const COMPLEX8TimeSeries *series,  
    size_t first,  
    size_t length  
);

COMPLEX16TimeSeries *XLALCutCOMPLEX16TimeSeries (  
    const COMPLEX16TimeSeries *series,  
    size_t first,  
    size_t length  
);
\end{verbatim}
REAL4TimeSeries *XLALCutREAL4TimeSeries (  
    const REAL4TimeSeries *series,  
    size_t first,  
    size_t length  
);  

REAL8TimeSeries *XLALCutREAL8TimeSeries (  
    const REAL8TimeSeries *series,  
    size_t first,  
    size_t length  
);  

INT2TimeSeries *XLALCutINT2TimeSeries (  
    const INT2TimeSeries *series,  
    size_t first,  
    size_t length  
);  

INT4TimeSeries *XLALCutINT4TimeSeries (  
    const INT4TimeSeries *series,  
    size_t first,  
    size_t length  
);  

INT8TimeSeries *XLALCutINT8TimeSeries (  
    const INT8TimeSeries *series,  
    size_t first,  
    size_t length  
);  

UINT2TimeSeries *XLALCutUINT2TimeSeries (  
    const UINT2TimeSeries *series,  
    size_t first,  
    size_t length  
);  

UINT4TimeSeries *XLALCutUINT4TimeSeries (  
    const UINT4TimeSeries *series,  
    size_t first,  
    size_t length  
);  

UINT8TimeSeries *XLALCutUINT8TimeSeries (  
    const UINT8TimeSeries *series,  
    size_t first,  
    size_t length  
);  

Description

These functions create a new time series by extracting a section of an existing time series.

Author

Kipp Cannon <kipp@gravity.phys.uwm.edu>

7.13.4 Resizing Functions

Name

XLALResize\textit{timeseriestype}() ,  
LALResize\textit{timeseriestype}() ,  
XLALShrink\textit{timeseriestype}() ,  
LALShrink\textit{timeseriestype}()
Synopsis

#include <lal/TimeSeries.h>

COMPLEX8TimeSeries *XLALResizeCOMPLEX8TimeSeries (  
    COMPLEX8TimeSeries *series,  
    int first,  
    size_t length  
);

void LALResizeCOMPLEX8TimeSeries (  
    LALStatus *status,  
    COMPLEX8TimeSeries *series,  
    int first,  
    size_t length  
);

COMPLEX8TimeSeries *XLALShrinkCOMPLEX8TimeSeries (  
    COMPLEX8TimeSeries *series,  
    size_t first,  
    size_t length  
);

COMPLEX16TimeSeries *XLALResizeCOMPLEX16TimeSeries (  
    COMPLEX16TimeSeries *series,  
    int first,  
    size_t length  
);

void LALResizeCOMPLEX16TimeSeries (  
    LALStatus *status,  
    COMPLEX16TimeSeries *series,  
    int first,  
    size_t length  
);

COMPLEX16TimeSeries *XLALShrinkCOMPLEX16TimeSeries (  
    COMPLEX16TimeSeries *series,  
    size_t first,  
    size_t length  
);

REAL4TimeSeries *XLALResizeREAL4TimeSeries (  
    REAL4TimeSeries *series,  
    int first,  
    size_t length  
);

void LALResizeREAL4TimeSeries (  
    LALStatus *status,  
    REAL4TimeSeries *series,  
    int first,  
    size_t length  
);

REAL4TimeSeries *XLALShrinkREAL4TimeSeries (  
    REAL4TimeSeries *series,  
    size_t first,  
    size_t length  
);
REAL8TimeSeries *XLALResizeREAL8TimeSeries (REAL8TimeSeries *series, int first, size_t length);
void LALResizeREAL8TimeSeries (LALStatus *status, REAL8TimeSeries *series, int first, size_t length);
REAL8TimeSeries *XLALShrinkREAL8TimeSeries (REAL8TimeSeries *series, size_t first, size_t length);
void LALResizeINT2TimeSeries (LALStatus *status, INT2TimeSeries *series, int first, size_t length);
INT2TimeSeries *XLALResizeINT2TimeSeries (INT2TimeSeries *series, int first, size_t length);
void LALResizeINT4TimeSeries (LALStatus *status, INT4TimeSeries *series, int first, size_t length);
INT4TimeSeries *XLALResizeINT4TimeSeries (INT4TimeSeries *series, int first, size_t length);
void LALResizeINT8TimeSeries (LALStatus *status, INT8TimeSeries *series, int first, size_t length);
INT8TimeSeries *XLALResizeINT8TimeSeries (INT8TimeSeries *series, int first, size_t length);
void LALResizeINT8TimeSeries (  
    LALStatus *status,   
    INT8TimeSeries *series,   
    int first,   
    size_t length 
);  

INT8TimeSeries *XLALShrinkINT8TimeSeries (  
    INT8TimeSeries *series,  
    size_t first,  
    size_t length 
);  

UINT2TimeSeries *XLALResizeUINT2TimeSeries (  
    UINT2TimeSeries *series,  
    int first,  
    size_t length 
);  

void LALResizeUINT2TimeSeries (  
    LALStatus *status,  
    UINT2TimeSeries *series,  
    int first,  
    size_t length 
);  

UINT2TimeSeries *XLALShrinkUINT2TimeSeries (  
    UINT2TimeSeries *series,  
    size_t first,  
    size_t length 
);  

UINT4TimeSeries *XLALResizeUINT4TimeSeries (  
    UINT4TimeSeries *series,  
    int first,  
    size_t length 
);  

void LALResizeUINT4TimeSeries (  
    LALStatus *status,  
    UINT4TimeSeries *series,  
    int first,  
    size_t length 
);  

UINT4TimeSeries *XLALShrinkUINT4TimeSeries (  
    UINT4TimeSeries *series,  
    size_t first,  
    size_t length 
);  

UINT8TimeSeries *XLALResizeUINT8TimeSeries (  
    UINT8TimeSeries *series,  
    int first,  
    size_t length 
);
void LALResizeUINT8TimeSeries (  
    LALStatus *status,  
    UINT8TimeSeries *series,  
    int first,  
    size_t length  
);  

UINT8TimeSeries *XLALShrinkUINT8TimeSeries (  
    UINT8TimeSeries *series,  
    size_t first,  
    size_t length  
);  

Description

These functions resize an existing time series. The new time series will have the given length, and its contents  
will consist of that part of the original time series that started at sample first. If first is negative, then the new  
time series is padded at the start by that many samples. The time series' epoch is adjusted appropriately.

Author

Kipp Cannon <kipp@gravity.phys.uwm.edu>

7.13.5 Addition Functions

Name

XLALAddtimeseriestype ()

Synopsis

#include <lal/TimeSeries.h>

COMPLEX8TimeSeries *XLALAddCOMPLEX8TimeSeries (  
    COMPLEX8TimeSeries *arg1,  
    const COMPLEX8TimeSeries *arg2  
);  

COMPLEX16TimeSeries *XLALAddCOMPLEX16TimeSeries (  
    COMPLEX16TimeSeries *arg1,  
    const COMPLEX16TimeSeries *arg2  
);  

REAL4TimeSeries *XLALAddREAL4TimeSeries (  
    REAL4TimeSeries *arg1,  
    const REAL4TimeSeries *arg2  
);  

REAL8TimeSeries *XLALAddREAL8TimeSeries (  
    REAL8TimeSeries *arg1,  
    const REAL8TimeSeries *arg2  
);  

INT2TimeSeries *XLALAddINT2TimeSeries (  
    INT2TimeSeries *arg1,  
    const INT2TimeSeries *arg2  
);  

INT4TimeSeries *XLALAddINT4TimeSeries (  
    INT4TimeSeries *arg1,  
    const INT4TimeSeries *arg2  
);
Description

These functions add the second argument to the first argument, returning a pointer to the first argument on success or NULL on failure. The two series must have the same heterodyne frequency and time resolution, and have units that differ only by a dimensionless factor.

Author

Kipp Cannon <kipp@gravity.phys.uwm.edu>
7.14 Header `Segments.h`

Provides data types and functions for manipulating lists of “segments” (GPS time intervals).

**Synopsis**

```
#include <lal/Segments.h>
```

This header defines data structures for segments and lists of segments, as well as prototypes for functions that manipulate them.

A segment is a time interval with a start time and an end time. The end time must be equal to or later than the start time. If the end time is equal to the start time, then the segment represents a point in time. If the end time is later than the start time, then the segment represents a half-open time interval, inclusive of its starting point and exclusive of its ending point.

All of the segment list manipulation functions are XLAL functions. They handle error conditions by invoking the current XLAL error handler and setting `xlalErrno` to a nonzero value.

**Error conditions**

<table>
<thead>
<tr>
<th><code>xlalErrno</code></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XLALEFAULT</td>
<td>Null pointer passed for some argument</td>
</tr>
<tr>
<td>XLAL EINVAL</td>
<td>Attempted to use an uninitialized segment list structure</td>
</tr>
<tr>
<td>XLAL EDOM</td>
<td>Pair of GPS times does not represent a valid segment</td>
</tr>
</tbody>
</table>

**Structures**

```c
typedef struct
  tagLALSeg
{
  LIGOTimeGPS start; /**< Beginning time of the segment */
  LIGOTimeGPS end; /**< Ending time of the segment */
  INT4 id; /**< Identifier (segment ID, array index, etc.) for user */
} LALSeg;
```

```c
typedef struct
  tagLALSegList
{
  LALSeg *segs; /**< Pointer to array of segments (LALSeg structures) */
  size_t arraySize; /**< Size of array for which memory is allocated */
  UINT4 length; /**< Number of segments in this segment list */
  UINT4 dplaces; /**< Decimal places (0,3,6,9) to format GPS times */
  UINT4 sorted; /**< Flag to indicate whether segment list is sorted */
  UINT4 disjoint; /**< Flag to indicate whether segment list is disjoint */
  UINT4 initMagic; /**< Internal value to help ensure list was initialized */
  LALSeg *lastFound; /**< Internal record of last segment found by a search */
} LALSegList;
```

**Notes**

A `LALSegList` must be initialized before it is used. Initialization leaves it in an “empty” state, containing no segments. They also must be "cleared" after using `XLALSegListClear`, and freed with `LALFree` if it was dynamically allocated. Segments can then be added to the list through an “append” operation. The information about each segment appended is copied to a memory location managed by the `LALSegList` object. In fact, the segments are stored in the form of an array of `LALSeg` structures, with the `segs` field of the segment list structure being the base address of the array. This allows the segments to be accessed directly using a pointer as an iterator, as in the following example code:
LALSegList mylist;
LALSeg *segp;
...
/* (Append segments to the segment list 'mylist' here) */
...
for ( segp=mylist.segs; segp<mylist.segs+mylist.length; segp++ ) {
    printf( "The end time of the segment is GPS %d.%09d\n",
            segp->end.gpsSeconds, segp->end.gpsNanoSeconds );
}

... or by using an integer array index, as in the following example code:

LALSegList mylist;
LALSeg *segp;
INT4 iseg;
LIGOTimeGPS startgps;
...
/* (Append segments to the segment list 'mylist' here) */
...
for ( iseg=0; iseg<mylist.length; iseg++ ) {
    /* One way to access the segment... */
    startgps = mylist.segs[iseg].start;
    printf( "The start time of the segment is GPS %d.%09d\n",
            startgps.gpsSeconds, startgps.gpsNanoSeconds );

    /* Another way to access the segment... */
    segp = mylist.segs + iseg;
    printf( "The end time of the segment is GPS %d.%09d\n",
            segp->end.gpsSeconds, segp->end.gpsNanoSeconds );
}

Note that if the segment list is empty, then the segs field will be NULL and the length field will be 0. So be careful not to dereference the segs pointer unless you know that the length is nonzero.

A segment list is considered “sorted” if the segments are in ascending (or at least non-descending) order according to the comparison done by the XLALSegCmp function. A segment list is considered “disjoint” if no two segments in the list overlap, although they may touch at an endpoint due to the half-open nature of the time intervals represented by segments. The LALSegList structure includes fields which record whether the segment list is sorted and/or disjoint, and these are used to search the segment list more efficiently when possible. Note that a segment list could in principle be disjoint but not sorted, but that case is not of interest for the code; the disjoint field in the structure specifically means that the list is sorted and disjoint.

Also all segments in a segment list can be time-shifted using XLALSegListShift.
7.14.1 Module Segments.c

Functions for manipulating segment lists.

Prototypes

```
INT4
XLALSegSet( LALSeg *seg, const LIGOTimeGPS *start, const LIGOTimeGPS *end,
            const INT4 id )

LALSeg *
XLALSegCreate( const LIGOTimeGPS *start, const LIGOTimeGPS *end,
               const INT4 id )

int
XLALGPSInSeg( const void *pgps, const void *pseg )

int
XLALSegCmp( const void *pseg0, const void *pseg1 )

INT4
XLALSegListInit( LALSegList *seglist )

INT4
XLALSegListClear( LALSegList *seglist )

INT4
XLALSegListAppend( LALSegList *seglist, const LALSeg *seg )

INT4
XLALSegListSort( LALSegList *seglist )

INT4
XLALSegListCoalesce( LALSegList *seglist )

LALSeg *
XLALSegListSearch( LALSegList *seglist, const LIGOTimeGPS *gps )

INT4
XLALSegListShift( LALSegList *seglist, const LIGOTimeGPS *gps )
```

Functions for handling segments

The first few functions listed above deal with segments:

The function `XLALSegSet()` sets the start time, the end time, and the id of a segment. The id can be any integer and is solely for the use of the user, e.g. to store a segment ID code or an index into some array containing additional information about the segment. `XLALSegSet()` checks to make sure the segment is valid, i.e. the end time is later than or equal to the start time; an error occurs if this condition is not true.

The function `XLALSegCreate()` is similar to `XLALSegSet()` except that it allocates memory for a new segment structure rather than setting the fields of an existing segment structure. It returns a pointer to the new LALSeg structure. When the structure is no longer needed, its pointer should be passed to `LALFree()`.

`XLALGPSInSeg` is designed to be usable as a comparison function for `bsearch()` and therefore returns a negative value, 0, or a positive value depending on whether the GPS time (the first argument) is before the beginning of, within, or after the end of the segment (the second argument). Note that a segment is a half-open interval, so the GPS time is considered to be within the segment if it is equal to the start time of the segment but not if it is equal to the end time of the segment.

`XLALSegCmp` is designed to be usable as a comparison function for `qsort()` and therefore returns a negative value, 0, or a positive value depending on whether the first argument is less than, equal to, or greater than the second. The comparison is based on the start time of the segments, unless these are equal, in which case the end times are compared. Therefore, two segments are considered equal only if their start and end times are identical.
Functions for handling segment lists

The rest of the functions listed above deal with segment lists:

The function `XLALSegListInit()` must be called to initialize a segment list structure before that structure can be used.

The function `XLALSegListClear()` must be called when you are done with a segment list, in order to free memory that was allocated to store the segments in the list. (Strictly speaking, this is only necessary if the list contains one or more segments.) The function leaves the segment list in a valid state, but containing no segments. After calling `XLALSegListClear()`, it is OK to re-use the segment list structure (by using `XLALSegListAppend()` to add segments to it, etc.). You do not have to call `XLALSegListInit()` again before re-using it, but it is OK to do so. If you do call `XLALSegListInit()` again, you must do so after calling `XLALSegListClear()`!

The function `XLALSegListAppend()` appends a segment to a segment list. It first checks to make sure the segment is valid, i.e. the end time is later than or equal to the start time; an error occurs if this condition is not true. The input segment information is copied into an array of `LALSeg` structures maintained "internally" by the segment list. The function takes care of extending this array when necessary. It also checks whether the segment being appended preserves the "sorted" and/or "disjoint" properties of the segment list. An empty segment list has the "sorted" and "disjoint" properties to start with, and as long as segments are appended in ascending time order and do not overlap, it retains those properties.

The function `XLALSegListSort()` sorts the segments in a segment list into forward time order. If the list is already sorted, then this function returns promptly.

The function `XLALSegListCoalesce()` first sorts the segments in a segment list (if not already sorted) and then joins together segments which overlap or touch (i.e. share endpoints). The result is a segment list which is sorted and is guaranteed to not have any overlapping segments; thus it is "disjoint". (Note, however, that a disjoint segment list is not necessarily coalesced, since segments which touch at an endpoint are considered disjoint but will be joined by `XLALSegListCoalesce()`.) If the list has the "disjoint" property to begin with, then this function returns promptly. Each segment in the output list is assigned the id value taken from the first segment in the input list (after it has been sorted) among those which were joined to make it.

The function `XLALSegListSearch()` determines which segment in the list, if any, contains the GPS time passed to this function. It returns a pointer to a segment containing the time, if there is one, otherwise it returns NULL. If more than one segment in the list contains the time, then this function returns a pointer to one of the segments which contains it, not necessarily the first such segment in the list. (This is not an issue if the list is "disjoint", which guarantees that it has no overlapping segments.) The following code shows how the `XLALSegListSearch()` function might be used:

```c
LALSegList mylist;
LIGOTimeGPS tgps, startgps;
LALSeg *segp;

/* (Fill the segment list 'mylist' with segments here) */
/* (Set the gps time, 'tgps', to search for) */

segp = XLALSegListSearch( &mylist, &tgps );
if ( segp ) {
    startgps = segp->start;
    printf( "That time is within a segment which starts at GPS time %d.%09d\n",
            startgps.gpsSeconds, startgps.gpsNanoSeconds );
} else {
    printf( "That time is not within any segment in the list\n" );
}
```

The search algorithm used by the `XLALSegListSearch()` function depends on whether the segment list is "sorted" and/or "disjoint". If the segment list has both of these properties, then the function can use a binary search to locate the segment containing the time, or to determine that there is no such segment. (Therefore, it is a good idea to pass a segment list to `XLALSegListCoalesce` before using it with `XLALSegListSearch`, unless segment list ordering or distinct segments which touch/overlap are meaningful for what you are doing, which is sometimes the case.) Otherwise, it must use a linear search, although a "sorted" list can still be searched slightly more efficiently than an un-sorted list. In all cases, `XLALSegListSearch()` first
checks whether the segment found by the last successful search contains the specified time, and returns that promptly if so.

Error codes and return values

Each XLAL function listed above, if it succeeds, sets `xlalErrno` to 0 before returning. If it fails, it invokes the current XLAL error handler, sets `xlalErrno` to the appropriate XLAL error code, and returns a particular value as noted below:

- Functions which return an integer status code (`XLALSegSet`, `XLALSegListInit`, `XLALSegListClear`, `XLALSegListAppend`, `XLALSegListSort`, `XLALSegListCoalesce`) return `XLAL_SUCCESS` if successful or `XLAL_FAILURE` if an error occurs.

- `XLALGPSInSeg` and `XLALSegCmp` normally return a comparison value (negative, 0, or positive). If a NULL pointer is passed to either function, it sets `xlalErrno` to a nonzero value and returns `XLAL_FAILURE`. `XLAL_FAILURE` happens to be equal to −1, so this looks like a valid comparison value if one neglects to check `xlalErrno`.

- `XLALSegCreate` normally returns a pointer to the created segment. If an error occurs, it returns NULL.

- `XLALSegListSearch` returns a pointer to a segment in the list which contains the time being searched for, or NULL if there is no such segment. If more than one segment in the list contains the time, then this function returns a pointer to one of the segments which contains it, not necessarily the first such segment in the list. (This is not an issue if the list is “disjoint”, which guarantees that it has no overlapping segments.) If no segment in the list contains the time, then this function returns NULL; however, this is not really an error, so `xlalErrno` will be equal to zero in this case. If a real error occurs—e.g., the function arguments are invalid—then this function returns NULL and sets `xlalErrno` to a nonzero value.
7.14.2 Program SegmentsTest.c

Tests the segment and segment list manipulation functions.

Usage

SegmentsTest [ lalDebugLevel ]

The default value of lalDebugLevel is 4.

Description

This program tests the various XLAL functions which deal with segments and segment lists.

If the lalDebugLevel argument is omitted, the test program sets it to 4 to turn on info messages only. Note that this default value does not enable LAL/XLAL error messages, since many of the tests intentionally create error conditions and verify that the proper error codes are generated. If you want to turn on the LAL/XLAL error and warning messages, specify a lalDebugLevel value of 7, or 23 if you also want informational messages related to memory checking. If you specify 0, then no messages will be printed under any circumstances. However, in all cases, the return status of the program will be 0 if all tests passed, 1 if one or more tests failed.

* Author: Peter Shawhan
* Revision: $Id: SegmentsTest.c,v 1.2 2007/06/08 14:41:59 bema Exp $
References


Chapter 8

Package vectorops

This package contains routines for manipulating vectors.
8.1 Header VectorOps.h

Basic vector manipulation operations.

Synopsis

```c
#include <lal/VectorOps.h>
```

Error conditions

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>1</td>
<td>&quot;Null pointer&quot;</td>
</tr>
<tr>
<td>SIZE</td>
<td>2</td>
<td>&quot;Invalid input size&quot;</td>
</tr>
<tr>
<td>SZMM</td>
<td>4</td>
<td>&quot;Size mismatch&quot;</td>
</tr>
<tr>
<td>SAME</td>
<td>8</td>
<td>&quot;Input/Output data vectors are the same&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants VECTOROPSH_E<name>, and the status descriptions in VECTOROPSH_MSGE<name>. The source code with these messages is in VectorOps.h on line 1.69.
8.1.1 Module VectorMultiply.c

Multiply two vectors.

Prototypes

```c
void LALCCVectorDivide (  
    LALStatus *status,  
    COMPLEX8Vector *out,  
    const COMPLEX8Vector *in1,  
    const COMPLEX8Vector *in2  
)

void LALZZVectorDivide (  
    LALStatus *status,  
    COMPLEX16Vector *out,  
    const COMPLEX16Vector *in1,  
    const COMPLEX16Vector *in2  
)

void LALCCVectorMultiply (  
    LALStatus *status,  
    COMPLEX8Vector *out,  
    const COMPLEX8Vector *in1,  
    const COMPLEX8Vector *in2  
)

void LALZZVectorMultiply (  
    LALStatus *status,  
    COMPLEX16Vector *out,  
    const COMPLEX16Vector *in1,  
    const COMPLEX16Vector *in2  
)

void LALCCVectorMultiplyConjugate (  
    LALStatus *status,  
    COMPLEX8Vector *out,  
    const COMPLEX8Vector *in1,  
    const COMPLEX8Vector *in2  
)

void LALZZVectorMultiplyConjugate (  
    LALStatus *status,  
    COMPLEX16Vector *out,  
    const COMPLEX16Vector *in1,  
    const COMPLEX16Vector *in2  
)

void LALSCVectorMultiply (  
    LALStatus *status,  
    COMPLEX8Vector *out,  
    const REAL4Vector *in1,  
    const COMPLEX8Vector *in2  
)
```
void LALDZVectorMultiply ( LALStatus *status, COMPLEX16Vector *out, const REAL8Vector *in1, const COMPLEX16Vector *in2 )

void LALSSVectorMultiply ( LALStatus *status, REAL4Vector *out, const REAL4Vector *in1, const REAL4Vector *in2 )

void LALDDVectorMultiply ( LALStatus *status, REAL8Vector *out, const REAL8Vector *in1, const REAL8Vector *in2 )

**Description**

Let \( u \), \( v \), and \( w \) be objects of type \texttt{COMPLEX8Vector}, and let \( a \), \( b \), and \( c \) be objects of type \texttt{REAL4Vector}.

The \texttt{LALCCVectorMultiply( \&status, \&w, \&u, \&v )} function computes \( w.data[i] = u.data[i] \times v.data[i] \).

The \texttt{LALCCVectorMultiplyConjugate( \&status, \&w, \&u, \&v )} function computes \( w.data[i] = u.data[i] \times \overline{v.data[i]} \).

The \texttt{LALCCVectorDivide( \&status, \&w, \&u, \&v )} function computes \( w.data[i] = u.data[i]/v.data[i] \).

The \texttt{LALSCVectorMultiply( \&status, \&w, \&a, \&v )} function computes \( w.data[i] = a.data[i] \times v.data[i] \).

The \texttt{LALSSVectorMultiply( \&status, \&c, \&a, \&b )} function computes \( c.data[i] = a.data[i] \times b.data[i] \).

The double-precision multiply routines (with \texttt{D} or \texttt{Z} names) work similarly.

**Algorithm**

The algorithm for complex division is described in Sec. 5.4 of Ref. [1]. The formula used is:

\[
\frac{a + ib}{c + id} = \begin{cases} 
\frac{[a+b(d/c)]+i[b-a(d/c)]}{c+d(-d/c)} & |c| \geq |d| \\
\frac{[a(c/d)+b]+i[b(c/d)-a]}{c(c/d)+d} & |c| < |d|. 
\end{cases}
\]
8.1.2 Module VectorPolar.c

Convert complex vector components from rectangular coordinates to polar coordinates.

Prototypes

```c
void LALCVectorAbs( LALStatus *status, REAL4Vector *out, const COMPLEX8Vector *in )
void LALZVectorAbs( LALStatus *status, REAL8Vector *out, const COMPLEX16Vector *in )
void LALCVectorAngle( LALStatus *status, REAL4Vector *out, const COMPLEX8Vector *in )
void LALZVectorAngle( LALStatus *status, REAL8Vector *out, const COMPLEX16Vector *in )
void LALUnwrapREAL4Angle( LALStatus *status, REAL4Vector *out, const REAL4Vector *in )
void LALUnwrapREAL8Angle( LALStatus *status, REAL8Vector *out, const REAL8Vector *in )
```

Description

Let \( u \) be an object of type \( \text{COMPLEX8Vector} \), and let \( a \) and \( b \) be objects of type \( \text{REAL4Vector} \).

The \( \text{LALCVectorAbs}( \&\text{status}, \&a, \&u ) \) function computes the magnitude of a complex vector \( u \).

\[
a.data[i] = \sqrt{u.data[i].re^2 + v.data[i].im^2}.
\]

The \( \text{LALCVectorAngle}( \&\text{status}, \&a, \&u ) \) function computes the phase angle of a complex vector \( u \) in the interval \([-\pi, \pi]\) radians.

\[
a.data[i] = \text{atan2}(u.data[i].im,v.data[i].re).
\]

The \( \text{LALUnwrapREAL4Angle}( \&\text{status}, \&a, \&b ) \) function corrects the radian phase angles of a real vector \( b \) by adding multiples of \( \pm\pi \) when the absolute jumps between consecutive angle elements are greater than \( \pi \) radians. This function detects branch cut crossings, but it can be fooled by sparse, rapidly changing phase values.

The double-precision functions are similar.
Algorithm

The algorithm for LALUnwrapREAL4Angle and LALUnwrapREAL8Angle (Inspired from the MATLAB function unwrap):

```c
a = in->data;
b = out->data;
n = out->length;

cumsum = 0.0;
phaseI = *a;
*b = phaseI;
--n;

while (n-- > 0)
{
    ++a;
    ++b;
    phaseII = *a;
diffph = phaseII - phaseI;
    phaseI = phaseII;

    cumsum += LAL_TWOPI*( (diffph < -LAL_PI) - (diffph > LAL_PI) );

    *b = phaseII + cumsum;
}
```

Notes

For the LALUnwrapREAL4Angle and LALUnwrapREAL8Angle functions, a, and b should not point to the same memory location (a != b).
8.1.3 Program VectorOpsTest.c

Tests the routines in VectorOps.h. Exercises some of the error conditions and makes sure that they work.

Usage

VectorOpsTest [options]

Options:
- `-h` print help
- `-q` quiet: run silently
- `-v` verbose: print extra information
- `-d level` set lalDebugLevel to level

Description

Exit codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success, normal exit.</td>
</tr>
<tr>
<td>1</td>
<td>Subroutine failed.</td>
</tr>
</tbody>
</table>
8.2 Header VectorIndexRange.h

Routines to slice up vectors/sequences by specifying starting and ending indices, inclusive. VectorIndexHole() is "complementary" to VectorIndexRange(), in the sense that it returns the two segments of the vector that would remain after VectorIndexRange() acts.

Synopsis

```c
#include <lal/VectorIndexRange.h>
```

Error conditions

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARG</td>
<td>1</td>
<td>&quot;Error parsing command-line arguments&quot;</td>
</tr>
<tr>
<td>CHK</td>
<td>2</td>
<td>&quot;Error checking failed to catch bad data&quot;</td>
</tr>
<tr>
<td>FLS</td>
<td>3</td>
<td>&quot;Incorrect answer for valid data&quot;</td>
</tr>
<tr>
<td>USE</td>
<td>4</td>
<td>&quot;Bad user-entered data&quot;</td>
</tr>
<tr>
<td>NULL</td>
<td>5</td>
<td>&quot;Null Pointer.&quot;</td>
</tr>
<tr>
<td>ALOC</td>
<td>6</td>
<td>&quot;Memory Allocation Error&quot;</td>
</tr>
<tr>
<td>FPMS</td>
<td>7</td>
<td>&quot;Filter Parameter Structure Error&quot;</td>
</tr>
<tr>
<td>NUMZ</td>
<td>8</td>
<td>&quot;Incorrect number of command line arguments&quot;</td>
</tr>
<tr>
<td>LNTH</td>
<td>9</td>
<td>&quot;Vector/Array of Improper Length&quot;</td>
</tr>
<tr>
<td>NNUL</td>
<td>10</td>
<td>&quot;Non-Null Pointer that should be NULL&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants VECTORINDEXRANGEH_E<name>, and the status descriptions in VECTORINDEXRANGEH_MSGE<name>. The source code with these messages is in VectorIndexRange.h on line 1.67.
8.2.1 Module VectorIndexRange.c

Description

Algorithms

Uses

- LAL*CreateVector

Notes

Prototypes

```c
void LALCHARVectorIndexRange ( LALStatus *status, CHARVector **result,
                                CHARVector *A, const UINT4Vector *indexVector )

void LALI2VectorIndexRange ( LALStatus *status, INT2Vector **result,
                                INT2Vector *A, const UINT4Vector *indexVector )

void LALI4VectorIndexRange ( LALStatus *status, INT4Vector **result,
                                INT4Vector *A, const UINT4Vector *indexVector )

void LALI8VectorIndexRange ( LALStatus *status, INT8Vector **result,
                                INT8Vector *A, const UINT4Vector *indexVector )

void LALU2VectorIndexRange ( LALStatus *status, UINT2Vector **result,
                                UINT2Vector *A, const UINT4Vector *indexVector )

void LALU4VectorIndexRange ( LALStatus *status, UINT4Vector **result,
                                UINT4Vector *A, const UINT4Vector *indexVector )

void LALU8VectorIndexRange ( LALStatus *status, UINT8Vector **result,
                                UINT8Vector *A, const UINT4Vector *indexVector )

void LALSVectorIndexRange ( LALStatus *status, REAL4Vector **result,
                                REAL4Vector *A, const UINT4Vector *indexVector )

void LALDVectorIndexRange ( LALStatus *status, REAL8Vector **result,
                                REAL8Vector *A, const UINT4Vector *indexVector )

void LALCVectorIndexRange ( LALStatus *status, COMPLEX8Vector **result,
                                COMPLEX8Vector *A, const UINT4Vector *indexVector )

void LALZVectorIndexRange ( LALStatus *status, COMPLEX16Vector **result,
                                COMPLEX16Vector *A, const UINT4Vector *indexVector )
```

Author: David Chin <dwchin@umich.edu> +1-734-709-9119

$Id: VectorIndexRangeC.m4,v 1.2 2003/08/14 15:09:15 dwchin Exp $
8.3 Header Matrix.h

Matlab Routines to handle Matrices & Vectors.

Synopsis

```c
#include <lal/Matrix.h>
```

Error conditions

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARG</td>
<td>1</td>
<td>&quot;Error parsing command-line arguments&quot;</td>
</tr>
<tr>
<td>CHK</td>
<td>2</td>
<td>&quot;Error checking failed to catch bad data&quot;</td>
</tr>
<tr>
<td>FLS</td>
<td>3</td>
<td>&quot;Incorrect answer for valid data&quot;</td>
</tr>
<tr>
<td>USE</td>
<td>4</td>
<td>&quot;Bad user-entered data&quot;</td>
</tr>
<tr>
<td>NULL</td>
<td>5</td>
<td>&quot;Null Pointer.&quot;</td>
</tr>
<tr>
<td>ALOC</td>
<td>6</td>
<td>&quot;Memory Allocation Error&quot;</td>
</tr>
<tr>
<td>FPMS</td>
<td>7</td>
<td>&quot;Filter Parameter Structure Error&quot;</td>
</tr>
<tr>
<td>NUMZ</td>
<td>8</td>
<td>&quot;Incorrect number of command line arguments&quot;</td>
</tr>
<tr>
<td>LNTH</td>
<td>9</td>
<td>&quot;Vector/Array of Improper Length&quot;</td>
</tr>
<tr>
<td>NNUL</td>
<td>10</td>
<td>&quot;Non-Null Pointer that should be NULL&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above did not obey the LAL naming convention, i.e. the code does not use the file name in all caps as the prefix for the error names. Consult the source code for the full names. Better yet, fix the code. The source code with these messages is in Matrix.h on line 1.85.
8.3.1 Module MatrixMultiply.c

This file is dedicated to reproducing the matlab function ".*". This file has several declarations of the same function taking all forms of available input. This being said, I have yet to script the complex actions and their counterparts.

Description

This file is to help make the conversion from Matlab to c much earier. In this file, we have created all of the versions of .* that we plan on using.

Algorithms

The algorithm is the same as it is in matlab. The dot in front of an operator in matlab signifies that if either or both of the operands are vectors, then the operation will be carried out member by member. For instance

```
vector a[25]; vector b[25]; vector c[25];
c = a .* b;
```

The result of this is:
```
c[0] = a[0] * b[0]; c[1] = a[1] * b[1]; . . . . . . . .
```

etc.

Uses

- LALDCreateVector

Notes

At the current time none of the operations have been specified for neither the complex datatypes nor the unsigned datatypes.

Prototypes

```c
void LALDDotStarDVector ( LALStatus *status, REAL8Vector **result,
REAL8 B, REAL8Vector *A )

void LALDVectorDotStarDVector ( LALStatus *status, REAL8Vector **result,
REAL8Vector *B, REAL8Vector *A )

void LALDDotStarDArray ( LALStatus *status, REAL8Array **result,
REAL8 A, REAL8Array *B )

void LALDArrayDotStarDArray ( LALStatus *status, REAL8Array **result,
REAL8Array *A, REAL8Array *B )

void LALDDotStarSVector ( LALStatus *status, REAL4Vector **result,
REAL8 B, REAL4Vector *A )

void LALDVectorDotStarSVector ( LALStatus *status, REAL8Vector **result,
REAL8Vector *B, REAL4Vector *A )

void LALDDotStarSArray ( LALStatus *status, REAL4Array **result,
REAL8 A, REAL4Array *B )

void LALDArrayDotStarSArray ( LALStatus *status, REAL8Array **result,
REAL8Array *A, REAL4Array *B )
```
void LALDDotStarI2Vector ( LALStatus *status, INT2Vector **result, REAL8 B, INT2Vector *A )

void LALDVectorDotStarI2Vector ( LALStatus *status, REAL8Vector **result, REAL8Vector *B, INT2Vector *A )

void LALDDotStarI2Array ( LALStatus *status, INT2Array **result, REAL8 A, INT2Array *B )

void LALDArrayDotStarI2Array ( LALStatus *status, REAL8Array **result, REAL8Array *A, INT2Array *B )

void LALDDotStarI4Vector ( LALStatus *status, INT4Vector **result, REAL8 B, INT4Vector *A )

void LALDVectorDotStarI4Vector ( LALStatus *status, REAL8Vector **result, REAL8Vector *B, INT4Vector *A )

void LALDDotStarI4Array ( LALStatus *status, INT4Array **result, REAL8 A, INT4Array *B )

void LALDArrayDotStarI4Array ( LALStatus *status, REAL8Array **result, REAL8Array *A, INT4Array *B )

void LALDDotStarI8Vector ( LALStatus *status, INT8Vector **result, REAL8 B, INT8Vector *A )

void LALDVectorDotStarI8Vector ( LALStatus *status, REAL8Vector **result, REAL8Vector *B, INT8Vector *A )

void LALDDotStarI8Array ( LALStatus *status, INT8Array **result, REAL8 A, INT8Array *B )

void LALDArrayDotStarI8Array ( LALStatus *status, REAL8Array **result, REAL8Array *A, INT8Array *B )

void LALSDotStarSVector ( LALStatus *status, REAL4Vector **result, REAL4 B, REAL4Vector *A )

void LALSVectorDotStarSVector ( LALStatus *status, REAL4Vector **result, REAL4Vector *B, REAL4Vector *A )

void LALSDotStarSArray ( LALStatus *status, REAL4Array **result, REAL4 A, REAL4Array *B )

void LALSArrayDotStarSArray ( LALStatus *status, REAL4Array **result, REAL4Array *A, REAL4Array *B )

void LALSDotStarI2Vector ( LALStatus *status, INT2Vector **result, REAL4 B, INT2Vector *A )
void LALSVectorDotStarI2Vector ( LALStatus *status, REAL4Vector **result, REAL4Vector *B, INT2Vector *A )

void LALSDotStarI2Array ( LALStatus *status, INT2Array **result, REAL4 A, INT2Array *B )

void LALSArrayDotStarI2Array ( LALStatus *status, REAL4Array **result, REAL4Array *A, INT2Array *B )

void LALSVectorDotStarI4Vector ( LALStatus *status, REAL4Vector **result, REAL4Vector *B, INT4Vector *A )

void LALSDotStarI4Vector ( LALStatus *status, INT4Vector **result, INT4Vector *A )

void LALSVectorDotStarI4Vector ( LALStatus *status, REAL4Vector **result, REAL4Vector *B, INT4Vector *A )

void LALSDotStarI4Array ( LALStatus *status, INT4Array **result, REAL4 A, INT4Array *B )

void LALSArrayDotStarI4Array ( LALStatus *status, REAL4Array **result, REAL4Array *A, INT4Array *B )

void LALSDotStarI8Vector ( LALStatus *status, INT8Vector **result, REAL4 B, INT8Vector *A )

void LALSVectorDotStarI8Vector ( LALStatus *status, REAL4Vector **result, REAL4Vector *B, INT8Vector *A )

void LALSDotStarI8Array ( LALStatus *status, INT8Array **result, REAL4 A, INT8Array *B )

void LALSArrayDotStarI8Array ( LALStatus *status, REAL4Array **result, REAL4Array *A, INT8Array *B )

void LALI2DotStarI2Vector ( LALStatus *status, INT2Vector **result, INT2 B, INT2Vector *A )

void LALI2VectorDotStarI2Vector ( LALStatus *status, INT2Vector **result, INT2Vector *B, INT2Vector *A )

void LALI2DotStarI2Array ( LALStatus *status, INT2Array **result, INT2 A, INT2Array *B )

void LALI2ArrayDotStarI2Array ( LALStatus *status, INT2Array **result, INT2Array *A, INT2Array *B )

void LALI4DotStarI2Vector ( LALStatus *status, INT4Vector **result, INT4 B, INT2Vector *A )

void LALI4VectorDotStarI2Vector ( LALStatus *status, INT4Vector **result, INT4Vector *B, INT2Vector *A )
void LALI4DotStarI2Array ( LALStatus *status, INT2Array **result,
  INT4 A, INT2Array *B )

void LALI4ArrayDotStarI2Array ( LALStatus *status, INT4Array **result,
  INT4Array *A, INT2Array *B )

void LALI4DotStarI4Vector ( LALStatus *status, INT4Vector **result,
  INT4 B, INT4Vector *A )

void LALI4VectorDotStarI4Vector ( LALStatus *status, INT4Vector **result,
  INT4Vector *B, INT4Vector *A )

void LALI4DotStarI4Array ( LALStatus *status, INT4Array **result,
  INT4 A, INT4Array *B )

void LALI4ArrayDotStarI4Array ( LALStatus *status, INT4Array **result,
  INT4Array *A, INT4Array *B )

void LALI8DotStarI2Vector ( LALStatus *status, INT2Vector **result,
  INT8 B, INT2Vector *A )

void LALI8VectorDotStarI2Vector ( LALStatus *status, INT8Vector **result,
  INT8Vector *B, INT2Vector *A )

void LALI8DotStarI2Array ( LALStatus *status, INT2Array **result,
  INT8 A, INT2Array *B )

void LALI8ArrayDotStarI2Array ( LALStatus *status, INT8Array **result,
  INT8Array *A, INT2Array *B )

void LALI8DotStarI4Vector ( LALStatus *status, INT4Vector **result,
  INT8 B, INT4Vector *A )

void LALI8VectorDotStarI4Vector ( LALStatus *status, INT8Vector **result,
  INT8Vector *B, INT4Vector *A )

void LALI8DotStarI4Array ( LALStatus *status, INT4Array **result,
  INT8 A, INT4Array *B )

void LALI8ArrayDotStarI4Array ( LALStatus *status, INT8Array **result,
  INT8Array *A, INT4Array *B )

void LALI8DotStarI8Vector ( LALStatus *status, INT8Vector **result,
  INT8 B, INT8Vector *A )

void LALI8VectorDotStarI8Vector ( LALStatus *status, INT8Vector **result,
  INT8Vector *B, INT8Vector *A )

void LALI8DotStarI8Array ( LALStatus *status, INT8Array **result,
  INT8 A, INT8Array *B )

void LALI8ArrayDotStarI8Array ( LALStatus *status, INT8Array **result,
  INT8Array *A, INT8Array *B )
8.3.2 Module MatrixDivide.c

This file is dedicated to reproducing the matlab function "./". This file has several declarations of the same function taking all forms of available input. This being said, I have yet to script the complex actions and their counterparts.

Description

This file is to help make the conversion from Matlab to c much easier. In this file, we have created all of the versions of ./ that we plan on using.

Algorithms

The algorithm is the same as it is in matlab. The dot in front of an operator in matlab signifies that if either or both of the operands are vectors, then the operation will be carried out member by member. For instance:

```c
vector a[25]; vector b[25]; vector c[25];
c = a . / b;
```

The result of this is:
```c
c[0] = a[0] / b[0]; c[1] = a[1] / b[1]; . . . . . .
```

etc.

Uses

- LALDCreateVector

Notes

At the current time none of the operations have been specified for neither the complex datatypes nor the unsigned datatypes.

Prototypes

```c
void LALDDotSlashDVector ( LALStatus *status, REAL8Vector **result, REAL8 A, REAL8Vector *B )

void LALDVectorDotSlashD ( LALStatus *status, REAL8Vector **result, REAL8Vector *A, REAL8 B )

void LALDVectorDotSlashDVector ( LALStatus *status, REAL8Vector **result, REAL8Vector *A, REAL8Vector *B )

void LALDDotSlashDArray ( LALStatus *status, REAL8Array **result, REAL8 A, REAL8Array *B )

void LALDArrayDotSlashD ( LALStatus *status, REAL8Array **result, REAL8Array *A, REAL8 B )

void LALDArrayDotSlashDArray ( LALStatus *status, REAL8Array **result, REAL8Array *A, REAL8Array *B )

void LALDDotSlashSVector ( LALStatus *status, REAL4Vector **result, REAL8 A, REAL4Vector *B )

void LALDVectorDotSlashS ( LALStatus *status, REAL4Vector **result, REAL4Vector *A, REAL8 B )
```
void LALDVectorDotSlashSVector ( LALStatus *status, REAL8Vector **result, 
   REAL8Vector *A, REAL4Vector *B )

void LALDDotSlashSArray ( LALStatus *status, REAL4Array **result, 
   REAL8 A, REAL4Array *B )

void LALDArrayDotSlashS ( LALStatus *status, REAL8Array **result, 
   REAL8Array *A, REAL4 B )

void LALDArrayDotSlashSArray ( LALStatus *status, REAL8Array **result, 
   REAL8Array *A, REAL4Array *B )

void LALDVectorDotSlashI2Vector ( LALStatus *status, INT2Vector **result, 
   REAL8 A, INT2Vector *B )

void LALVectorDotSlashI2 ( LALStatus *status, INT2Vector **result, 
   INT2Vector *A, REAL8 B )

void LALDVectorDotSlashI2Vector ( LALStatus *status, REAL8Vector **result, 
   REAL8Vector *A, INT2Vector *B )

void LALDVectorDotSlashI2Array ( LALStatus *status, INT2Array **result, 
   REAL8 A, INT2Array *B )

void LALDArrayDotSlashI2 ( LALStatus *status, REAL8Array **result, 
   REAL8Array *A, INT2 B )

void LALDArrayDotSlashI2Array ( LALStatus *status, REAL8Array **result, 
   REAL8Array *A, INT2Array *B )

void LALDVectorDotSlashI4Vector ( LALStatus *status, INT4Vector **result, 
   REAL8 A, INT4Vector *B )

void LALVectorDotSlashI4 ( LALStatus *status, INT4Vector **result, 
   INT4Vector *A, REAL8 B )

void LALDVectorDotSlashI4Vector ( LALStatus *status, REAL8Vector **result, 
   REAL8Vector *A, INT4Vector *B )

void LALDVectorDotSlashI4Array ( LALStatus *status, INT4Array **result, 
   REAL8 A, INT4Array *B )

void LALDArrayDotSlashI4 ( LALStatus *status, REAL8Array **result, 
   REAL8Array *A, INT4 B )

void LALDArrayDotSlashI4Array ( LALStatus *status, REAL8Array **result, 
   REAL8Array *A, INT4Array *B )

void LALDVectorDotSlashI8Vector ( LALStatus *status, INT8Vector **result, 
   REAL8 A, INT8Vector *B )
void LALDVectorDotSlashI8 ( LALStatus *status, INT8Vector **result, INT8Vector *A, REAL8 B )

void LALDVectorDotSlashI8Vector ( LALStatus *status, REAL8Vector **result, REAL8Vector *A, INT8Vector *B )

void LALDDotSlashI8Array ( LALStatus *status, INT8Array **result, REAL8 A, INT8Array *B )

void LALDArrayDotSlashI8 ( LALStatus *status, REAL8Array **result, REAL8Array *A, INT8 B )

void LALDArrayDotSlashI8Array ( LALStatus *status, REAL8Array **result, REAL8Array *A, INT8Array *B )

void LALSVectorDotSlashS ( LALStatus *status, REAL4Vector **result, REAL4Vector *A, REAL4 B )

void LALSVectorDotSlashSVector ( LALStatus *status, REAL4Vector **result, REAL4Vector *A, REAL4Vector *B )

void LALSArrayDotSlashS ( LALStatus *status, REAL4Array **result, REAL4Array *A, REAL4 B )

void LALSArrayDotSlashSArray ( LALStatus *status, REAL4Array **result, REAL4Array *A, REAL4Array *B )

void LALSDotSlashI2Vector ( LALStatus *status, INT2Vector **result, REAL4 A, INT2Vector *B )

void LALSVectorDotSlashI2 ( LALStatus *status, INT2Vector **result, INT2Vector *A, REAL4 B )

void LALSVectorDotSlashI2Vector ( LALStatus *status, REAL4Vector **result, REAL4Vector *A, INT2Vector *B )

void LALSDotSlashI2Array ( LALStatus *status, INT2Array **result, REAL4 A, INT2Array *B )

void LALSArrayDotSlashI2 ( LALStatus *status, REAL4Array **result, REAL4Array *A, INT2 B )

void LALSArrayDotSlashI2Array ( LALStatus *status, REAL4Array **result, REAL4Array *A, INT2Array *B )
void LALSDotSlashI4Vector ( LALStatus *status, INT4Vector **result,
REAL4 A, INT4Vector *B )

void LALSVectorDotSlashI4 ( LALStatus *status, INT4Vector **result,
INT4Vector *A, REAL4 B )

void LALSVectorDotSlashI4Vector ( LALStatus *status, REAL4Vector **result,
REAL4Vector *A, INT4Vector *B )

void LALSDotSlashI4Array ( LALStatus *status, INT4Array **result,
REAL4 A, INT4Array *B )

void LALSArrayDotSlashI4 ( LALStatus *status, REAL4Array **result,
REAL4Array *A, INT4 B )

void LALSArrayDotSlashI4Array ( LALStatus *status, REAL4Array **result,
REAL4Array *A, INT4Array *B )

void LALSDotSlashI8Vector ( LALStatus *status, INT8Vector **result,
REAL4 A, INT8Vector *B )

void LALSVectorDotSlashI8 ( LALStatus *status, INT8Vector **result,
INT8Vector *A, REAL4 B )

void LALSVectorDotSlashI8Vector ( LALStatus *status, REAL4Vector **result,
REAL4Vector *A, INT8Vector *B )

void LALSDotSlashI8Array ( LALStatus *status, INT8Array **result,
REAL4 A, INT8Array *B )

void LALSArrayDotSlashI8 ( LALStatus *status, REAL4Array **result,
REAL4Array *A, INT8 B )

void LALSArrayDotSlashI8Array ( LALStatus *status, REAL4Array **result,
REAL4Array *A, INT8Array *B )

void LALI2DotSlashI2Vector ( LALStatus *status, INT2Vector **result,
INT2 A, INT2Vector *B )

void LALI2VectorDotSlashI2 ( LALStatus *status, INT2Vector **result,
INT2Vector *A, INT2 B )

void LALI2VectorDotSlashI2Vector ( LALStatus *status, INT2Vector **result,
INT2Vector *A, INT2Vector *B )

void LALI2DotSlashI2Array ( LALStatus *status, INT2Array **result,
INT2 A, INT2Array *B )

void LALI2ArrayDotSlashI2 ( LALStatus *status, INT2Array **result,
INT2Array *A, INT2 B )
void LALI2ArrayDotSlashI2Array ( LALStatus *status, INT2Array **result, 
INT2Array *A, INT2Array *B )

void LALI4DotSlashI2Vector ( LALStatus *status, INT2Vector **result, 
INT4 A, INT2Vector *B )

void LALI4VectorDotSlashI2 ( LALStatus *status, INT2Vector **result, 
INT2Vector *A, INT4 B )

void LALI4VectorDotSlashI2Vector ( LALStatus *status, INT4Vector **result, 
INT4Vector *A, INT2Vector *B )

void LALI4DotSlashI2Array ( LALStatus *status, INT2Array **result, 
INT4 A, INT2Array *B )

void LALI4ArrayDotSlashI2 ( LALStatus *status, INT4Array **result, 
INT4Array *A, INT2 B )

void LALI4ArrayDotSlashI2Array ( LALStatus *status, INT4Array **result, 
INT4Array *A, INT2Array *B )

void LALI4DotSlashI2Vector ( LALStatus *status, INT4Vector **result, 
INT4 A, INT4Vector *B )

void LALI4VectorDotSlashI4 ( LALStatus *status, INT4Vector **result, 
INT4Vector *A, INT4 B )

void LALI4VectorDotSlashI4Vector ( LALStatus *status, INT4Vector **result, 
INT4Vector *A, INT4Vector *B )

void LALI4DotSlashI4Array ( LALStatus *status, INT4Array **result, 
INT4 A, INT4Array *B )

void LALI4ArrayDotSlashI4 ( LALStatus *status, INT4Array **result, 
INT4Array *A, INT4 B )

void LALI4ArrayDotSlashI4Array ( LALStatus *status, INT4Array **result, 
INT4Array *A, INT4Array *B )

void LALI8DotSlashI2Vector ( LALStatus *status, INT2Vector **result, 
INT8 A, INT2Vector *B )

void LALI8VectorDotSlashI2 ( LALStatus *status, INT2Vector **result, 
INT2Vector *A, INT8 B )

void LALI8VectorDotSlashI2Vector ( LALStatus *status, INT8Vector **result, 
INT8Vector *A, INT2Vector *B )

void LALI8DotSlashI2Array ( LALStatus *status, INT2Array **result, 
INT8 A, INT2Array *B )
void LALIS8ArrayDotSlashI2 ( LALStatus *status, INT8Array **result,
INT8Array *A, INT2 B )

void LALIS8ArrayDotSlashI2Array ( LALStatus *status, INT8Array **result,
INT8Array *A, INT2Array *B )

void LALISDotSlashI4Vector ( LALStatus *status, INT4Vector **result,
INT8 A, INT4Vector *B )

void LALISVectorDotSlashI4 ( LALStatus *status, INT4Vector **result,
INT4Vector *A, INT8 B )

void LALISVectorDotSlashI4Vector ( LALStatus *status, INT8Vector **result,
INT8Vector *A, INT4Vector *B )

void LALISDotSlashI8Vector ( LALStatus *status, INT8Vector **result,
INT8 A, INT8Vector *B )

void LALISVectorDotSlashI8 ( LALStatus *status, INT8Vector **result,
INT8Vector *A, INT8 B )

void LALISVectorDotSlashI8Vector ( LALStatus *status, INT8Vector **result,
INT8Vector *A, INT8Vector *B )

void LALISDotSlashI8Array ( LALStatus *status, INT8Array **result,
INT8 A, INT8Array *B )

void LALISArrayDotSlashI8 ( LALStatus *status, INT8Array **result,
INT8Array *A, INT8 B )

void LALISArrayDotSlashI8Array ( LALStatus *status, INT8Array **result,
INT8Array *A, INT8Array *B )
8.3.3 Module MatrixPower.c

This file is dedicated to reproducing the matlab function ".^". This file has several declarations of the same function taking all forms of available input. This being said, I have yet to script the complex actions and their counterparts.

Description

This file is to help make the conversion from Matlab to c much easier. In this file, we have created all of the versions of .^ that we plan on using.

Algorithms

The algorithm is the same as it is in matlab. The dot in front of an operator in matlab signifies that if either or both of the operands are vectors, then the operation will be carried out member by member. For instance

```c
vector a[25]; vector b[25]; vector c[25];
c = a .^ b;
```

The result of this is:

```
c[0] = a[0]^b[0]; c[1] = a[1]^b[1]; . . . . . . .
```

etc.

Uses

- LALDCreateVector

Notes

At the current time none of the operations have been specified for neither the complex datatypes nor the unsigned datatypes.

Prototypes

```c
void LALDDotPowerDVector ( LALStatus *status, REAL8Vector **result,
REAL8 A, REAL8Vector *B )

void LALDVectorDotPowerD ( LALStatus *status, REAL8Vector **result,
REAL8Vector *A, REAL8 B )

void LALDVectorDotPowerDVector ( LALStatus *status, REAL8Vector **result,
REAL8Vector *B, REAL8Vector *A )

void LALDDotPowerDArray ( LALStatus *status, REAL8Array **result,
REAL8 A, REAL8Array *B )

void LALDArrayDotPowerD ( LALStatus *status, REAL8Array **result,
REAL8Array *A, REAL8 B )

void LALDArrayDotPowerDArray ( LALStatus *status, REAL8Array **result,
REAL8Array *A, REAL8Array *B )

void LALDDotPowerSVector ( LALStatus *status, REAL8Vector **result,
REAL8 A, REAL4Vector *B )

void LALDVectorDotPowerS ( LALStatus *status, REAL8Vector **result,
REAL8Vector *A, REAL4 B )
```
void LALDVectorDotPowerSVector ( LALStatus *status, REAL8Vector **result, REAL8Vector *B, REAL4Vector *A )

void LALDDotPowerSArray ( LALStatus *status, REAL8Array **result, REAL8 A, REAL4Array *B )

void LALDArrayDotPowerS ( LALStatus *status, REAL8Array **result, REAL8Array *A, REAL4 B )

void LALDVectorDotPowerSArray ( LALStatus *status, REAL8Array **result, REAL8Array *A, REAL4Array *B )

void LALDVectorDotPowerI2Vector ( LALStatus *status, REAL8Vector **result, REAL8 A, INT2Vector *B )

void LALDVectorDotPowerI2 ( LALStatus *status, REAL8Vector **result, REAL8Vector *A, INT2 B )

void LALDVectorDotPowerI2Vector ( LALStatus *status, REAL8Vector **result, REAL8Vector *B, INT2Vector *A )

void LALDVectorDotPowerI2Array ( LALStatus *status, REAL8Array **result, REAL8 A, INT2Array *B )

void LALDVectorDotPowerI2 ( LALStatus *status, REAL8Array **result, REAL8Array *A, INT2 B )

void LALDVectorDotPowerI2Array ( LALStatus *status, REAL8Array **result, REAL8Array *A, INT2Array *B )

void LALDDotPowerI4Vector ( LALStatus *status, REAL8Vector **result, REAL8 A, INT4Vector *B )

void LALDVectorDotPowerI4 ( LALStatus *status, REAL8Vector **result, REAL8Vector *A, INT4 B )

void LALDVectorDotPowerI4Vector ( LALStatus *status, REAL8Vector **result, REAL8Vector *B, INT4Vector *A )

void LALDDotPowerI4Array ( LALStatus *status, REAL8Array **result, REAL8 A, INT4Array *B )

void LALDVectorDotPowerI4 ( LALStatus *status, REAL8Array **result, REAL8Array *A, INT4 B )

void LALDArrayDotPowerI4Array ( LALStatus *status, REAL8Array **result, REAL8Array *A, INT4Array *B )

void LALDDotPowerI8Vector ( LALStatus *status, REAL8Vector **result, REAL8 A, INT8Vector *B )
void LALDVectorDotPowerI8 ( LALStatus *status, REAL8Vector **result,
REAL8Vector *A, INT8 B )

void LALDVectorDotPowerI8Vector ( LALStatus *status, REAL8Vector **result,
REAL8Vector *B, INT8Vector *A )

void LALDDotPowerI8Array ( LALStatus *status, REAL8Array **result,
REAL8 A, INT8Array *B )

void LALDArrayDotPowerI8 ( LALStatus *status, REAL8Array **result,
REAL8Array *A, INT8 B )

void LALDArrayDotPowerI8Array ( LALStatus *status, REAL8Array **result,
REAL8Array *A, INT8Array *B )

void LALSDotPowerSVector ( LALStatus *status, REAL8Vector **result,
REAL4 A, REAL4Vector *B )

void LALSVectorDotPowerS ( LALStatus *status, REAL8Vector **result,
REAL4Vector *A, REAL4 B )

void LALSVectorDotPowerSVector ( LALStatus *status, REAL8Vector **result,
REAL4Vector *B, REAL4Vector *A )

void LALSDotPowerSArray ( LALStatus *status, REAL8Array **result,
REAL4 A, REAL4Array *B )

void LALSArrayDotPowerS ( LALStatus *status, REAL8Array **result,
REAL4Array *A, REAL4 B )

void LALSArrayDotPowerSArray ( LALStatus *status, REAL8Array **result,
REAL4Array *A, REAL4Array *B )

void LALSDotPowerI2Vector ( LALStatus *status, REAL8Vector **result,
REAL4 A, INT2Vector *B )

void LALSVectorDotPowerI2 ( LALStatus *status, REAL8Vector **result,
REAL4Vector *A, INT2 B )

void LALSVectorDotPowerI2Vector ( LALStatus *status, REAL8Vector **result,
REAL4Vector *B, INT2Vector *A )

void LALSDotPowerI2Array ( LALStatus *status, REAL8Array **result,
REAL4 A, INT2Array *B )

void LALSArrayDotPowerI2 ( LALStatus *status, REAL8Array **result,
REAL4Array *A, INT2 B )

void LALSArrayDotPowerI2Array ( LALStatus *status, REAL8Array **result,
REAL4Array *A, INT2Array *B )
void LALSDotPowerI4Vector ( LALStatus *status, REAL8Vector **result, REAL4 A, INT4Vector *B )

void LALSVectorDotPowerI4 ( LALStatus *status, REAL8Vector **result, REAL4Vector *A, INT4 B )

void LALSVectorDotPowerI4Vector ( LALStatus *status, REAL8Vector **result, REAL4Vector *B, INT4Vector *A )

void LALSDotPowerI4Array ( LALStatus *status, REAL8Array **result, REAL4 A, INT4Array *B )

void LALSArrayDotPowerI4 ( LALStatus *status, REAL8Array **result, REAL4Array *A, INT4 B )

void LALSArrayDotPowerI4Array ( LALStatus *status, REAL8Array **result, REAL4Array *A, INT4Array *B )

void LALSDotPowerI8Vector ( LALStatus *status, REAL8Vector **result, REAL4 A, INT8Vector *B )

void LALSVectorDotPowerI8 ( LALStatus *status, REAL8Vector **result, REAL4Vector *A, INT8 B )

void LALSVectorDotPowerI8Vector ( LALStatus *status, REAL8Vector **result, REAL4Vector *B, INT8Vector *A )

void LALSDotPowerI8Array ( LALStatus *status, REAL8Array **result, REAL4 A, INT8Array *B )

void LALSArrayDotPowerI8 ( LALStatus *status, REAL8Array **result, REAL4Array *A, INT8 B )

void LALSArrayDotPowerI8Array ( LALStatus *status, REAL8Array **result, REAL4Array *A, INT8Array *B )

void LALI2DotPowerI2Vector ( LALStatus *status, REAL8Vector **result, INT2 A, INT2Vector *B )

void LALI2VectorDotPowerI2 ( LALStatus *status, REAL8Vector **result, INT2Vector *A, INT2 B )

void LALI2VectorDotPowerI2Vector ( LALStatus *status, REAL8Vector **result, INT2Vector *B, INT2Vector *A )

void LALI2DotPowerI2Array ( LALStatus *status, REAL8Array **result, INT2 A, INT2Array *B )

void LALI2ArrayDotPowerI2 ( LALStatus *status, REAL8Array **result, INT2Array *A, INT2 B )
void LALI2ArrayDotPowerI2Array ( LALStatus *status, REAL8Array **result, INT2Array *A, INT2Array *B )

void LALI4DotPowerI2Vector ( LALStatus *status, REAL8Vector **result, INT4 A, INT2Vector *B )

void LALI4VectorDotPowerI2 ( LALStatus *status, REAL8Vector **result, INT4Vector *A, INT2 B )

void LALI4VectorDotPowerI2Vector ( LALStatus *status, REAL8Vector **result, INT4Vector *A, INT2Vector *B )

void LALI4DotPowerI2Array ( LALStatus *status, REAL8Array **result, INT4 A, INT2Array *B )

void LALI4ArrayDotPowerI2 ( LALStatus *status, REAL8Array **result, INT4Array *A, INT2 B )

void LALI4ArrayDotPowerI2Array ( LALStatus *status, REAL8Array **result, INT4Array *A, INT2Array *B )

void LALI4DotPowerI4Vector ( LALStatus *status, REAL8Vector **result, INT4 A, INT4Vector *B )

void LALI4VectorDotPowerI4 ( LALStatus *status, REAL8Vector **result, INT4Vector *A, INT4 B )

void LALI4VectorDotPowerI4Vector ( LALStatus *status, REAL8Vector **result, INT4Vector *A, INT4Vector *B )

void LALI4DotPowerI4Array ( LALStatus *status, REAL8Array **result, INT4 A, INT4Array *B )

void LALI4ArrayDotPowerI4 ( LALStatus *status, REAL8Array **result, INT4Array *A, INT4 B )

void LALI4ArrayDotPowerI4Array ( LALStatus *status, REAL8Array **result, INT4Array *A, INT4Array *B )

void LALI8DotPowerI2Vector ( LALStatus *status, REAL8Vector **result, INT8 A, INT2Vector *B )

void LALI8VectorDotPowerI2 ( LALStatus *status, REAL8Vector **result, INT8Vector *A, INT2 B )

void LALI8VectorDotPowerI2Vector ( LALStatus *status, REAL8Vector **result, INT8Vector *A, INT2Vector *B )

void LALI8DotPowerI2Array ( LALStatus *status, REAL8Array **result, INT8 A, INT2Array *B )
void LALI8ArrayDotPowerI2 ( LALStatus *status, REAL8Array **result, INT8Array *A, INT2 B )

void LALI8ArrayDotPowerI2Array ( LALStatus *status, REAL8Array **result, INT8Array *A, INT2Array *B )

void LALI8DotPowerI4Vector ( LALStatus *status, REAL8Vector **result, INT8Vector *A, INT4Vector *B )

void LALI8VectorDotPowerI4 ( LALStatus *status, REAL8Vector **result, INT8Vector *A, INT4 B )

void LALI8VectorDotPowerI4Vector ( LALStatus *status, REAL8Vector **result, INT8Vector *B, INT4Vector *A )

void LALI8DotPowerI4Array ( LALStatus *status, REAL8Array **result, INT8 A, INT4Array *B )

void LALI8ArrayDotPowerI4 ( LALStatus *status, REAL8Array **result, INT8Array *A, INT4 B )

void LALI8ArrayDotPowerI4Array ( LALStatus *status, REAL8Array **result, INT8Array *A, INT4Array *B )

void LALI8DotPowerI8Vector ( LALStatus *status, REAL8Vector **result, INT8 A, INT8Vector *B )

void LALI8VectorDotPowerI8 ( LALStatus *status, REAL8Vector **result, INT8Vector *A, INT8 B )

void LALI8VectorDotPowerI8Vector ( LALStatus *status, REAL8Vector **result, INT8Vector *B, INT8Vector *A )

void LALI8DotPowerI8Array ( LALStatus *status, REAL8Array **result, INT8 A, INT8Array *B )

void LALI8ArrayDotPowerI8 ( LALStatus *status, REAL8Array **result, INT8Array *A, INT8 B )

void LALI8ArrayDotPowerI8Array ( LALStatus *status, REAL8Array **result, INT8Array *A, INT8Array *B )
8.3.4 Module MiscMatlab.c

This file reproduces the last few matlab functions that we needed for our purposes. It creates useable forms of cumsum, sum, max, and finally an implementation of the array addressing in matlab. Matlab has an easy of inverting a vector, (end: -1: 1) and the final function, FlipVector returns a result vector that has been flipped in that same manner.

Description
This file reproduces the last few matlab functions that we needed for our purposes. It creates useable forms of cumsum, sum, max, and finally an implementation of the array addressing in matlab. Matlab has an easy of inverting a vector, (end: -1: 1) and the final function, FlipVector returns a result vector that has been flipped in that same manner.

Algorithms
The algorithms are the same as in matlab. Flip vector was discussed above. Sum takes the sum of all of the elements in a vector. Cum sum takes an input vector:

```
vector input[25]; vector output[25];
```

etc

Uses
- LALDCreateVector

Notes
At the current time none of the operations have been specified for neither the complex datatypes nor the unsigned datatypes.
Also, the prototypes are out of order as I have used m4 to create all of the functions from one codebase.

Prototypes

```c
void LALDCumSum ( LALStatus *status, REAL8Vector **result, REAL8Vector *data )
void LALDSum ( LALStatus *status, REAL8 *result, REAL8Vector *data )
void LALDMax ( LALStatus *status, REAL8 *result, REAL8Vector *data, INT4 *myindex )
void LALDFlipVector ( LALStatus *status, REAL8Vector **result, REAL8Vector *data )
void LALSCumSum ( LALStatus *status, REAL4Vector **result, REAL4Vector *data )
void LALSSum ( LALStatus *status, REAL4 *result, REAL4Vector *data )
void LALSMax ( LALStatus *status, REAL4 *result, REAL4Vector *data, INT4 *myindex )
void LALSFlipVector ( LALStatus *status, REAL4Vector **result, REAL4Vector *data )
void LALI2CumSum ( LALStatus *status, INT2Vector **result, INT2Vector *data )
void LALI2Sum ( LALStatus *status, INT2 *result, INT2Vector *data )
```
void LALI2Max (LALStatus *status, INT2 *result, INT2Vector *data, INT4 *myindex )

void LALI2FlipVector (LALStatus *status, INT2Vector **result, INT2Vector *data )

void LALI4CumSum ( LALStatus *status, INT4Vector **result, INT4Vector *data )

void LALI4Sum ( LALStatus *status, INT4 *result, INT4Vector *data )

void LALI4Max (LALStatus *status, INT4 *result, INT4Vector *data, INT4 *myindex )

void LALI4FlipVector (LALStatus *status, INT4Vector **result, INT4Vector *data )

void LALI8CumSum ( LALStatus *status, INT8Vector **result, INT8Vector *data )

void LALI8Sum ( LALStatus *status, INT8 *result, INT8Vector *data )

void LALI8Max (LALStatus *status, INT8 *result, INT8Vector *data, INT4 *myindex )

void LALI8FlipVector (LALStatus *status, INT8Vector **result, INT8Vector *data )
References

Section 5

General Mathematical and Signal Analysis Packages
Chapter 9

Package clremoval

This package provides routines for finding line harmonics, generating a reference interference signal, and removing all the interference harmonics. As an example, these routines can be useful for removing all those lines corresponding to the electricity supply frequency and its harmonics.

The technique used is the so-called coherent line removal (CLR) [1]-[5]. CLR is an algorithm able to remove interference present in the data while preserving the stochastic detector noise. CLR works when the interference is present in many harmonics, as long as they remain coherent with one another. CLR can remove the external interference without removing any ‘single line’ signal buried by the harmonics. The algorithm works even when the interference frequency changes. CLR can be used to remove all harmonics of periodic or broad-band signals (e.g., those which change frequency in time), even when there is no external reference source. CLR requires little a priori knowledge of the signals we want to remove.

The package is organized under the header CLR.h and the modules HarmonicFinder.c, RefInterference.c and CleanAll.c.
9.1 Header CLR.h

Provides routines for finding line harmonics, generating a reference interference signal, and removing all the interference harmonics.

Synopsis

\#include "CLR.h"

The principal of CLR is the following:

We assume that the interference has the form

\[ y(t) = \sum_n a_n m(t)^n + (a_n m(t))^* , \tag{9.1} \]

where \(a_n\) are complex amplitudes and \(m(t)\) is a nearly monochromatic function near a frequency \(f_0\). The idea is to use the information in the different harmonics of the interference to construct a function \(M(t)\) that is as close a replica as possible of \(m(t)\) and then construct a function close to \(y(t)\) which is subtracted from the output of the system cancelling the interference. The key is that real gravitational wave signals will not be present with multiple harmonics and that \(M(t)\) is constructed from many frequency bands with independent noise. Hence, CLR will little affect the statistics of the noise in any one band and any gravitational wave signal masked by the interference can be recovered without any disturbance.

We assume that the data produced by the system is just the sum of the interference plus noise

\[ x(t) = y(t) + n(t) , \tag{9.2} \]

where \(y(t)\) is given by Eq. (29.5) and the noise \(n(t)\) in the detector is a zero-mean stationary stochastic process. The procedure consists in defining a set of functions \(\tilde{z}_k(\nu)\) in the frequency domain as

\[ \tilde{z}_k(\nu) = \begin{cases} \hat{x}(\nu) & \nu_k < \nu < \nu_{fk} \\ 0 & \text{elsewhere} \end{cases} , \tag{9.3} \]

where \((\nu_k, \nu_{fk})\) correspond to the upper and lower frequency limits of the harmonics of the interference and \(k\) denotes the harmonic considered. These functions are equivalent to

\[ \tilde{z}_k(\nu) = a_k m^k(\nu) + \tilde{n}_k(\nu) , \tag{9.4} \]

where \(\tilde{n}_k(\nu)\) is the noise in the frequency band of the harmonic considered. Their inverse Fourier transforms yield

\[ z_k(t) = a_k m(t)^k + n_k(t) . \tag{9.5} \]

Since \(m(t)\) is supposed to be a narrow-band function near a frequency \(f_0\), each \(z_k(t)\) is a narrow-band function near \(kf_0\). Then, we define

\[ B_k(t) = [z_k(t)]^{1/k} , \tag{9.6} \]

that can be rewritten as

\[ B_k(t) = (a_k)^{1/k} m(t) \beta_k(t) , \quad \beta_k(t) = \left[ 1 + \frac{n_k(t)}{a_k m(t)^k} \right]^{1/k} . \tag{9.7} \]

All these functions, \(\{B_k(t)\}\), are almost monochromatic around the fundamental frequency, \(f_0\), but they differ basically by a certain complex amplitude. These factors, \(\Gamma_k\), can easily be calculated, and we can construct a set of functions \(\{b_k(t)\}\)

\[ b_k(t) = \Gamma_k B_k(t) , \tag{9.8} \]

such that, they all have the same mean value. Then, \(M(t)\) can be constructed as a function of all \(\{b_k(t)\}\) in such a way that it has the same mean and minimum variance. If \(M(t)\) is linear with \(\{b_k(t)\}\), the statistically the best is

\[ M(t) = \left( \sum_k \frac{b_k(t)}{\text{Var}[\beta_k(t)]} \right) / \left( \sum_k \frac{1}{\text{Var}[\beta_k(t)]} \right) , \tag{9.9} \]

where

\[ \text{Var}[\beta_k(t)] = \frac{\langle n_k(t)n_k(t)^* \rangle}{k^2 |a_k m(t)^k|^2} + \text{corrections} . \tag{9.10} \]
In practice, we approximate
\[ |a_k m(t)^k|^2 \approx |z_k(t)|^2 , \tag{9.11} \]
and we assume stationary noise. Therefore,
\[ \langle n_k(t)n_k(t)^* \rangle = \int_{\nu_k} S(\nu) d\nu , \tag{9.12} \]
where \( S(\nu) \) is the power spectral density of the noise.

Finally, it only remains to determine the amplitude of the different harmonics, which can be obtained applying a least square method.

### Error conditions

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>1</td>
<td>&quot;Null pointer&quot;</td>
</tr>
<tr>
<td>SIZE</td>
<td>2</td>
<td>&quot;Invalid input size&quot;</td>
</tr>
<tr>
<td>SZMM</td>
<td>4</td>
<td>&quot;Size mismatch&quot;</td>
</tr>
<tr>
<td>INT</td>
<td>6</td>
<td>&quot;Invalid interval&quot;</td>
</tr>
<tr>
<td>SAME</td>
<td>8</td>
<td>&quot;Input/Output data vectors are the same&quot;</td>
</tr>
<tr>
<td>FREQ</td>
<td>10</td>
<td>&quot;Invalid frequency&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants `CLRH_E<name>` , and the status descriptions in `CLRH_MSGE<name>` . The source code with these messages is in `CLR.h` on line 1.251.

### Structures

**struct REAL4TVectorCLR**

This structure stores the time domain data and other information needed in order to remove all the line interference harmonics. The fields are:

- **UINT4 length** The number of elements in data.
- **REAL4 *data** The (real) time domain data.
- **REAL4 deltaT** The sample spacing in time (in seconds).
- **REAL4 fLine** The interference fundamental frequency \( f_0 \) (in Hz), e.g., 60 Hz.

**struct REAL4FVectorCLR**

This structure stores the spectrum, \( |\tilde{x}(\nu)|^2 \), and other information needed to find the location of the line harmonics. The fields are:

- **UINT4 length** The number of elements in data.
- **REAL4 *data** The (real) frequency domain data, e.g., \( |\tilde{x}(\nu)|^2 \).
- **REAL4 deltaF** The \( \Delta F \) offset between samples (in Hz).
- **REAL4 fLine** The interference fundamental frequency \( f_0 \) (in Hz), e.g., 60 Hz.
9.1.1 Module HarmonicFinder.c

Given certain harmonic indices \( \{k\} \) finds the frequency interval location (in bins) of the interference (around \( k \times f_0 \)).

Prototypes

```c
void LALHarmonicFinder (LALStatus *status,
    INT4Vector *out,  // harmonic index and location, size 3*l */
    REAL4FVectorCLR *in2,  /* |x(f)|^2, data + information */
    INT4Vector *in1)   // the harmonic index, size l */
```

Description

This routine determines the lower and upper frequency limit (in bins) of each harmonic line considered, \( (\nu_{ik}, \nu_{fk}) \), from the power spectrum.

The harmonic indices are given as an input *in1.

*in1->length* Number of harmonics.

*in1->data* List of harmonics to consider, e.g., \( \{k\} = \{3, 5, 9, 11\ldots\} \).

The power spectrum, \( |\tilde{x}(\nu)|^2 \), together with the approximate frequency \( f_0 \) (in Hz) of the interference fundamental harmonic and the frequency resolution are also given as an input *in2*.

*in2->length* The number of elements in *in2->data*.

*in2->data* The spectrum, \( |\tilde{x}(\nu)|^2 \).

*in2->deltaF* The \( \Delta F \) offset between samples (in Hz).

*in2->fLine* The interference fundamental frequency \( f_0 \) (in Hz), e.g., 60 Hz.

The output *out* is a vector whose length is \( out->length = 3 \times in1->length \), and contains for each considered harmonic, in the following order, its index \( k \) and the bin location of \( \nu_{ik} \) and \( \nu_{fk} \).

*out->length* The number of elements in *out->data*.

*out->data* \( \{k, \nu_{ik}, \nu_{fk}\} \), e.g., \( \{3, 9868, 9894, 5, 16449, 16487, 9, 29607, 29675\ldots\} \).

Algorithm

It looks for the location of interference harmonics assuming that the fundamental harmonic is located somewhere in the interval \( in2->fLine - 0.7 \) Hz and \( in2->fLine + 0.7 \) Hz. First, the power spectrum is smoothed by averaging neighboring bins. Then, the corresponding frequency intervals of the harmonics considered are reduced by finding the central bin position of the lines and their standard deviation. This is done using the smooth power spectrum as a probability density distribution. The limits of the lines are set initially at 1 or 2 sigma from the central bin location and, later, they are moved until they hit a local minimum in a selected interval. See the code for details.

Uses

- LALSCreateVector()
- LALSDestroyVector()

Notes
9.1.2 Module `RefInterference.c`

Generates a reference interference signal.

Prototypes

```c
void LALRefInterference (LALStatus *status,  
    COMPLEX8Vector *out, /* M(t), size n */  
    COMPLEX8Vector *in1, /* x(f), size n/2+1 */  
    INT4Vector *par)  
```

Description

Given the complex vector `*in1` of length `n/2+1`, containing the Fourier transform of the data $\tilde{x}(\nu)$,

- `in1->length` The number of elements in `in1->data = n/2 + 1`
- `in1->data` The data $\tilde{x}(\nu)$

and given another vector `*par` containing the information related to the harmonics from which we want to construct the reference signal (i.e., indices, and initial and final frequency bin locations),

- `par->length` The number of elements in `par->data`. This is equal to three times the number of harmonics that will be used to construct the reference signal $M(t)$.
- `par->data` $\{k, \nu_{ik}, \nu_{fk}\}$, e.g., $\{3, 9868, 9894, 5, 16449, 16487, 9, 29607, 29675, \ldots\}$,

it generates the time domain $M(t)$ reference interference signal, `*out`. This is a complex vector of length $n$.

- `out->length` The number of elements in `out->data = n`
- `out->data` $M(t)$ complex data.

$M(t)$ corresponds to a nearly monochromatic function near the frequency $f_0$, that is implicit in the information given in `*par`.

Algorithm

Described before.

Uses

- `LALSCreateVector()`
- `LALCCreateVector()`
- `LALZCreateVector()`
- `LALCreateReverseComplexFFTPlan()`
- `LALCOMPLEX8VectorFFT()`
- `LALSDestroyVector()`
- `LALCDestroyVector()`
- `LALZDestroyVector()`
- `LALDestroyComplexFFTPlan()`

Notes

The harmonics selected to construct the reference signal should not be (if possible) buried with other strong lines, such as violin modes. Choose the strongest harmonics, those that clearly stand over the noise level.

Author: Sintes, A. M.

$Id: RefInterference.c,v 1.4 2007/06/08 14:41:43 bema Exp$
9.1.3 Module CleanAll.c

Gets data cleaned from line harmonic interference given a time domain reference signal.

Prototypes

```c
void LALCleanAll (LALStatus *status,
                  REAL4Vector *out, /* clean data */
                  COMPLEX8Vector *in2, /* M(t), ref. interference */
                  REAL4TVectorCLR *in1) /* x(t), data + information */
```

Description

This routine cleans data in the time domain from line harmonic interference (from the first harmonic up to
the Nyquist frequency). The inputs are:

* `*in1` the time domain data of type `REAL4TVectorCLR`, containing also the interference fundamental
  frequency `f_0` and the sampling spacing. This information is needed in order to obtain the total number
  of harmonics contained in the data.

  `in1->length` The number of elements in `in1->data = n`.

  `in1->data` The (real) time domain data, `x(t)`.

  `in1->deltaT` The sample spacing in seconds.

  `in1->fLine` The interference fundamental frequency `f_0` (in Hz), e.g., 60 Hz.

  * `*in2` the time domain reference signal (a complex vector).

  `in2->length` The number of elements in `in2->data = n`.

  `in2->data` The `M(t)` complex data.

  The output `*out` is a real vector containing the clean data.

  `out->length` The number of elements in `out->data = n`.

  `out->data` The clean (real) time domain data.

Algorithm

It takes the reference signal `M(t)` and, for all possible harmonics `j` (`j = 1,...,floor(1.0/fabs( 2.02*
`in1->deltaT  *  in1->fLine))`), from the fundamental frequency up to the Nyquist frequency, constructs
`M(t)^j`, performs a least-squares fit, i.e., minimizes the power `|x(t) - \rho_j M(t)^j|^2` with respect to `\rho_j`, and
subtracts `\rho_j M(t)^j` from the original data, `x(t)`.

Uses

- `LALDCreateVector()`
- `LALZCreateVector()`
- `LALDDestroyVector()`
- `LALZDestroyVector()`

Notes

Author: Sintes, A. M.

$Id: CleanAll.c,v 1.3 2007/06/08 14:41:43 bema Exp$
9.1.4 Program CLRTest.c

Test for CLR operations.

Usage

CLRTest

Description

This program is just an example of the usage of the different prototypes.

The program reads some data from the file CLRindata.asc, finds the position of several harmonics, builds a reference signal, cleans the initial data of all interference harmonics and writes the clean data into the file CLRoutdata.asc.

Exit codes

Uses

lalDebugLevel
LALI4CreateVector()
LALCCreateVector()
LALSCreateVector()
LALMalloc()
LALFree()
LALCreateForwardRealFFTPlan()
LALRealPowerSpectrum()
LALForwardRealFFT()
LALDestroyRealFFTPlan()
LALI4DestroyVector()
LALCDestroyVector()
LALSDestroyVector()
LALHarmonicFinder()
LALRefInterference()
LALCleanAll()
LALCreateReverseComplexFFTPlan()
LALCOMPLEX8VectorFFT()
LALDestroyComplexFFTPlan()

Notes

Take this program just as an example, build your own one and feed it with the data of your interest. The CLR functions work on stretches of data from a few seconds up to a couple of minutes.
References


Chapter 10

Package \texttt{fft}

This package contains various routines for performing FFTs.
10.1 Header RealFFT.h

Performs real-to-complex and complex-to-real FFTs.

Synopsis

```c
#include <lal/RealFFT.h>
```

Perform real-to-complex and complex-to-real fast Fourier transforms of vectors, and sequences of vectors using the package FFTW [I].

Error conditions

<table>
<thead>
<tr>
<th>name</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>1</td>
<td>&quot;Null pointer&quot;</td>
</tr>
<tr>
<td>NNUL</td>
<td>2</td>
<td>&quot;Non-null pointer&quot;</td>
</tr>
<tr>
<td>SIZE</td>
<td>4</td>
<td>&quot;Invalid input size&quot;</td>
</tr>
<tr>
<td>SZMM</td>
<td>8</td>
<td>&quot;Size mismatch&quot;</td>
</tr>
<tr>
<td>SLEN</td>
<td>16</td>
<td>&quot;Invalid/mismatched sequence lengths&quot;</td>
</tr>
<tr>
<td>SAME</td>
<td>32</td>
<td>&quot;Input/Output data vectors are the same&quot;</td>
</tr>
<tr>
<td>SIGN</td>
<td>64</td>
<td>&quot;Incorrect plan sign&quot;</td>
</tr>
<tr>
<td>DATA</td>
<td>128</td>
<td>&quot;Bad input data: DC/Nyquist should be real&quot;</td>
</tr>
<tr>
<td>ALOC</td>
<td>256</td>
<td>&quot;Memory allocation failed&quot;</td>
</tr>
<tr>
<td>FFTW</td>
<td>512</td>
<td>&quot;Error in FFTW&quot;</td>
</tr>
<tr>
<td>SNGL</td>
<td>1024</td>
<td>&quot;FFTW library is not single-precision&quot;</td>
</tr>
<tr>
<td>INTL</td>
<td>2048</td>
<td>&quot;Error in Intel FFT library&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants \texttt{REALFFTH\_E<name>}, and the status descriptions in \texttt{REALFFTH\_MGE<name>}. The source code with these messages is in \texttt{RealFFT.h} on line \texttt{1.87}.

Structures

```c
typedef struct tagREAL4FFTPlan REAL4FFTPlan;
typedef struct tagREAL8FFTPlan REAL8FFTPlan;
#define tagRealFFTPlan tagREAL4FFTPlan
#define RealFFTPlan REAL4FFTPlan
```

This structure contains the parameters necessary for performing an FFT of a given size and direction. The contents should not be manually adjusted.
10.1.1 Module RealFFT.c

Functions for performing real FFTs.

Prototypes

```c
void LALCreateForwardREAL4FFTPlan(  
    LALStatus *status,  
    REAL4FFTPlan **plan,  
    UINT4 size,  
    INT4 measure  
 );

void LALCreateReverseREAL4FFTPlan(  
    LALStatus *status,  
    REAL4FFTPlan **plan,  
    UINT4 size,  
    INT4 measure  
 );

void LALDestroyREAL4FFTPlan(  
    LALStatus *status,  
    REAL4FFTPlan **plan  
 );

void LALForwardREAL4FFT(  
    LALStatus *status,  
    COMPLEX8Vector *output,  
    REAL4Vector *input,  
    REAL4FFTPlan *plan  
 );

void LALReverseREAL4FFT(  
    LALStatus *status,  
    REAL4Vector *output,  
    COMPLEX8Vector *input,  
    REAL4FFTPlan *plan  
 );

void LALREAL4PowerSpectrum (  
    LALStatus *status,  
    REAL4Vector *spec,  
    REAL4Vector *data,  
    REAL4FFTPlan *plan  
 );

void LALREAL4VectorFFT(  
    LALStatus *status,  
    REAL4Vector *output,  
    REAL4Vector *input,  
    REAL4FFTPlan *plan  
 );
```
```c
void LALCreateForwardREAL8FFTPlan(
    LALStatus *status,
    REAL8FFTPlan **plan,
    UINT4 size,
    INT4 measure
)

void LALCreateReverseREAL8FFTPlan(
    LALStatus *status,
    REAL8FFTPlan **plan,
    UINT4 size,
    INT4 measure
)

void LALDestroyREAL8FFTPlan(
    LALStatus *status,
    REAL8FFTPlan **plan
)

void LALForwardREAL8FFT(
    LALStatus *status,
    COMPLEX16Vector *output,
    REAL8Vector *input,
    REAL8FFTPlan *plan
)

void LALReverseREAL8FFT(
    LALStatus *status,
    REAL8Vector *output,
    COMPLEX16Vector *input,
    REAL8FFTPlan *plan
)

void LALREAL8PowerSpectrum ( 
    LALStatus *status,
    REAL8Vector *spec,
    REAL8Vector *data,
    REAL8FFTPlan *plan
)

void LALREAL8VectorFFT( 
    LALStatus *status,
    REAL8Vector *output,
    REAL8Vector *input,
    REAL8FFTPlan *plan
)
```

Description

This package provides a LAL-style interface with the FFTW fast Fourier transform package [1]. The routines `LALCreateForwardREAL8FFTPlan()` and `LALCreateReverseREAL8FFTPlan()` create plans for computing the forward (real-to-complex) and reverse (complex-to-real) FFTs of a specified size.
The optimum plan is either estimated (reasonably fast) if the measure flag is zero, or measured (can be time-consuming, but gives better performance) if the measure flag is non-zero. The routine \texttt{LALDestroyRealFFTPlan()} destroys any of these flavours of plans.

The routines \texttt{LALForwardRealFFT()} and \texttt{LALReverseRealFFT()} perform the forward (real-to-complex) and reverse (complex-to-real) FFTs using the plans. The discrete Fourier transform \(H_k, k = 0 \ldots [n/2]\) (\(n/2\) rounded down), of a vector \(h_j, j = 0 \ldots n - 1\), of length \(n\) is defined by

\[
H_k = \sum_{j=0}^{n-1} h_j e^{-2\pi i jk/n}
\]

and, similarly, the inverse Fourier transform is defined by

\[
h_j = \frac{1}{n} \sum_{k=0}^{n-1} H_k e^{2\pi i jk/n}
\]

where \(H_k\) for \([n/2] < k < n\) can be obtained from the relation \(H_k = H_{n-k}^*\). The present implementation of the reverse FFT omits the factor of \(1/n\).

The routines in this package require that the vector \(h_j, j = 0 \ldots n - 1\) be real; consequently, \(H_k = H_{n-k}^*\) (\(0 \leq k \leq [n/2]\)), i.e., the negative frequency Fourier components are the complex conjugate of the positive frequency Fourier components when the data is real. Therefore, one need compute and store only the first \([n/2] + 1\) components of \(H_k\); only the values of \(H_k\) for \(k = 0 \ldots [n/2]\) are returned (integer division is rounded down, e.g., \([7/2] = 3\)).

The routine \texttt{LALRealPowerSpectrum()} computes the power spectrum \(P_k = 2|H_k|^2, k = 1 \ldots [(n-1)/2]\), \(P_0 = |H_0|^2\), and \(P_{n/2} = |H_{n/2}|^2\) if \(n\) is even, of the data \(h_j, j = 0 \ldots n - 1\). The factor of two except at DC and Nyquist accounts for the power in negative frequencies.

The routine \texttt{LALREAL4VectorFFT()} is essentially a direct calls to FFTW routines without any re-packing of the data. This routine should not be used unless the user understands the packing used in FFTW.

**Operating Instructions**

```c
const UINT4 n = 32;
static LALStatus status;
RealFFTPlan *pfwd = NULL;
RealFFTPlan *prev = NULL;
REAL4Vector *hvec = NULL;
COMPLEX8Vector *Hvec = NULL;
REAL4Vector *Pvec = NULL;
LALCreateForwardRealFFTPlan( &status, &pfwd, n );
LALCreateReverseRealFFTPlan( &status, &prev, n );
LALCreateVector( &status, &hvec, n );
LALCreateVector( &status, &Hvec, n/2 + 1 );
LALCreateVector( &status, &Pvec, n/2 + 1 );
<assign data>
LALRealPowerSpectrum( &status, Pvec, hvec, pfwd );
LALForwardRealFFT( &status, Hvec, hvec, pfwd );
LALReverseRealFFT( &status, hvec, Hvec, pinv );
LALDestroyRealFFTPlan( &status, &pfwd );
LALDestroyRealFFTPlan( &status, &prev );
LALSDestroyVector( &status, &hvec );
LALSDestroyVector( &status, &Hvec );
LALSDestroyVector( &status, &Pvec );
```

**Algorithm**

The FFTW \[1\] is used.
Uses

Notes

1. The sign convention used here is the opposite of Numerical Recipes [1], but agrees with the one used by FFTW [1] and the other LIGO software components.

2. The result of the reverse FFT must be multiplied by $1/n$ to recover the original vector. This is unlike the Numerical Recipes [1] convolution where the factor is $2/n$ for real FFTs. This is different from the datacondAPI where the normalization constant is applied by default.

3. The size $n$ of the transform can be any positive integer; the performance is $O(n \log n)$. However, better performance is obtained if $n$ is the product of powers of 2, 3, 5, 7, and zero or one power of either 11 or 13. Transforms when $n$ is a power of 2 are especially fast. See Ref. [1].

4. All of these routines leave the input array undamaged. (Except for LALREAL4VectorFFT.)

5. LALMalloc() is used by all the fftw routines.
10.1.2 XLAL Functions

Synopsis

```c
#include <lal/RealFFT.h>

REAL4FFTPlan * XLALCreateREAL4FFTPlan( UINT4 size, int fwdflg, int measurelvl );
REAL4FFTPlan * XLALCreateForwardREAL4FFTPlan( UINT4 size, int measurelvl );
REAL4FFTPlan * XLALCreateReverseREAL4FFTPlan( UINT4 size, int measurelvl );
void XLALDestroyREAL4FFTPlan( REAL4FFTPlan *plan );

int XLALREAL4ForwardFFT( COMPLEX8Vector *output, REAL4Vector *input, REAL4FFTPlan *plan );
int XLALREAL4ReverseFFT( REAL4Vector *output, COMPLEX8Vector *input, REAL44FFTPlan *plan );
int XLALREAL4VectorFFT( REAL4Vector *output, REAL4Vector *input, REAL4FFTPlan *plan );
int XLALREAL4PowerSpectrum( REAL4Vector *spec, REAL4Vector *data, REAL4FFTPlan *plan );

REAL8FFTPlan * XLALCreateREAL8FFTPlan( UINT4 size, int fwdflg, int measurelvl );
REAL8FFTPlan * XLALCreateForwardREAL8FFTPlan( UINT4 size, int measurelvl );
REAL8FFTPlan * XLALCreateReverseREAL8FFTPlan( UINT4 size, int measurelvl );
void XLALDestroyREAL8FFTPlan( REAL8FFTPlan *plan );

int XLALREAL8ForwardFFT( COMPLEX16Vector *output, REAL8Vector *input, REAL8FFTPlan *plan );
int XLALREAL8ReverseFFT( REAL8Vector *output, COMPLEX16Vector *input, REAL8FFTPlan *plan );
int XLALREAL8VectorFFT( REAL8Vector *output, REAL8Vector *input, REAL8FFTPlan *plan );
int XLALREAL8PowerSpectrum( REAL8Vector *spec, REAL8Vector *data, REAL8FFTPlan *plan );
```

Description

The REAL4 routines are described below. These use single-precision FFTs, i.e., they convert REAL4Vectors into COMPLEX8Vectors and vice-versa. The REAL8 versions of the routines are the same but they are double-precision versions, i.e., they convert REAL8Vectors into COMPLEX16Vectors.

The routine XLALCreateREAL4FFTPlan creates a REAL4FFTPlan structure to perform FFTs of vectors of length size. If fwdflg is non-zero then the plan is created to perform forward (real-to-complex) FFTs with a negative exponential sign. Otherwise the plan is created to perform reverse (complex-to-real) FFTs with a positive exponential sign. The value of measurelvl determines how much optimization of the plan FFTW will do with the most optimization taking the most amount of time. Reasonable values for measurelvl would be 0 for the fastest plan creation (FFTW does not measure the speed of any transform with this level but rather estimates which plan will be the fastest) or 1 to measure a few likely plans to determine the fastest.

XLALCreateForwardREAL4FFTPlan is equivalent to XLALCreateREAL4FFTPlan with fwdflg set to 1.
XLALCreateReverseREAL4FFTPlan is equivalent to XLALCreateREAL4FFTPlan with fwdflg set to 0.

XLALDestroyREAL4FFTPlan is used to destroy the plan, freeing all memory that was allocated in the structure as well as the structure itself. It can be used on either forward or reverse plans.

XLALREAL4ForwardFFT and XLALREAL4ReverseFFT perform forward (real to complex) and reverse (complex to real) transforms respectively. The plan supplied to these routines must be correctly generated for the direction of the transform. I.e., XLALREAL4ForwardFFT cannot be supplied with a plan generated by XLALCreateReverseREAL4FFTPlan.

XLALREAL4VectorFFT is a low-level routine that transforms a real vector to a half-complex real vector (with a forward plan) or a half-complex real vector to a real vector (with a reverse plan). If you’re not sure what this means, don’t use this routine. The input and output vectors (and their data) must be distinct pointers.
**XLALREAL4PowerSpectrum** computes a real power spectrum of the input real vector and a forward FFT plan.

**Return Values**

Upon success, **XLALCreateREAL4FFTPlan**, **XLALCreateForwardREAL4FFTPlan**, and **XLALCreateReverseREAL4FFTPlan** return a pointer to a newly-allocated FFT plan. Upon failure, they return a **NULL** pointer and set **xlalErrno** to one of the following values: **XLAL_EBADLEN** if **size** is not greater than zero, **XLAL_ENOMEM** if a memory allocation failed, or **XLAL_EFAILED** if the FFTW plan creation routine failed.

**XLALDestroyREAL4FFTPlan** does not return any value but, upon failure, it will set **xlalErrno** to one of the following values: **XLAL_EFAULT** if the routine is provided a **NULL** pointer, or **XLAL_EINVAL** if the contents of the plan are invalid (e.g., if the routine is provided a plan that had been previously destroyed).

**XLALREAL4ForwardFFT**, **XLALREAL4ReverseFFT**, **XLALREAL4VectorFFT**, and **XLALREAL4PowerSpectrum** return the value 0 upon success; upon failure they return **XLAL_FAILURE** and set **xlalErrno** to one of the following values: **XLAL_EFAULT** if one of the input pointers is **NULL**, **XLAL_EINVAL** if the input, output, or plan structures appears invalid or if the routine is passed a plan for the wrong transform directions or if the input and output data pointers are not distinct for **XLALREAL4VectorFFT**, **XLAL_EBADLEN** if the input and output vectors and the plan have incompatible lengths, **XLAL_ENOMEM** if a memory allocation of temporary internal memory fails.

As before, the **REAL8** versions of these routines behave the same way but for double-precision transforms.
10.1.3 Program RealFFTTest.c

Tests the routines in RealFFT.h.

Usage

RealFFTTest [options]
Options:
  -h      print this message
  -q      quiet: run silently
  -v      verbose: print extra information
  -d level set lalDebugLevel to level
  -m trials set number of random trials
  -n size  set size of FFTs

Use the -n option to specify the size of the test transform and the -m option to specify the number of test transforms of that size. (Default is to test transforms of size 1 to 128 in unit steps and then powers of two up to 65536.)

Description

Exit codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success, normal exit.</td>
</tr>
<tr>
<td>1</td>
<td>Subroutine failed.</td>
</tr>
</tbody>
</table>

Uses

Notes
10.2 Header ComplexFFT.h

Performs complex-to-complex FFTs.

Synopsis

#include <lal/ComplexFFT.h>

Perform complex-to-complex fast Fourier transforms of vectors using the package FFTW [1].

Error conditions

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>1</td>
<td>&quot;Null pointer&quot;</td>
</tr>
<tr>
<td>NNUL</td>
<td>2</td>
<td>&quot;Non-null pointer&quot;</td>
</tr>
<tr>
<td>SIZE</td>
<td>4</td>
<td>&quot;Invalid input size&quot;</td>
</tr>
<tr>
<td>SZMM</td>
<td>8</td>
<td>&quot;Size mismatch&quot;</td>
</tr>
<tr>
<td>SLEN</td>
<td>16</td>
<td>&quot;Invalid/mismatched sequence lengths&quot;</td>
</tr>
<tr>
<td>SAME</td>
<td>32</td>
<td>&quot;Input/Output data vectors are the same&quot;</td>
</tr>
<tr>
<td>ALOC</td>
<td>64</td>
<td>&quot;Memory allocation failed&quot;</td>
</tr>
<tr>
<td>FFTW</td>
<td>128</td>
<td>&quot;Error in FFTW&quot;</td>
</tr>
<tr>
<td>SNGL</td>
<td>256</td>
<td>&quot;FFTW library is not single-precision&quot;</td>
</tr>
<tr>
<td>INTL</td>
<td>512</td>
<td>&quot;Error in Intel FFT library&quot;</td>
</tr>
<tr>
<td>SIGN</td>
<td>1024</td>
<td>&quot;Unknown sign of transform in plan&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants COMPLEXFFTH_E<name>, and the status descriptions in COMPLEXFFTH_MSGE<name>. The source code with these messages is in ComplexFFT.h on line l.82.

Structures

typedef struct tagCOMPLEX8FFTPlan COMPLEX8FFTPlan;
#define tagComplexFFTPlan tagCOMPLEX8FFTPlan
#define ComplexFFTPlan COMPLEX8FFTPlan

This structure contains the parameters necessary for performing an FFT of a given size and direction. The contents should not be manually adjusted.
10.2.1 Module ComplexFFT.c

Functions for performing complex FFTs.

Prototypes

```c
void LALCreateForwardCOMPLEX8FFTPlan(
    LALStatus *status,
    COMPLEX8FFTPlan **plan,
    UINT4 size,
    INT4 measure
);

void LALCreateReverseCOMPLEX8FFTPlan(
    LALStatus *status,
    COMPLEX8FFTPlan **plan,
    UINT4 size,
    INT4 measure
);

void LALDestroyCOMPLEX8FFTPlan(
    LALStatus *status,
    COMPLEX8FFTPlan **plan
);

void LALCOMPLEX8VectorFFT(
    LALStatus *status,
    COMPLEX8Vector *output,
    COMPLEX8Vector *input,
    COMPLEX8FFTPlan *plan
);

void LALCreateForwardCOMPLEX16FFTPlan(
    LALStatus *status,
    COMPLEX16FFTPlan **plan,
    UINT4 size,
    INT4 measure
);

void LALCreateReverseCOMPLEX16FFTPlan(
    LALStatus *status,
    COMPLEX16FFTPlan **plan,
    UINT4 size,
    INT4 measure
);

void LALDestroyCOMPLEX16FFTPlan(
    LALStatus *status,
    COMPLEX16FFTPlan **plan
);
```

1.435
ComplexFFT.c

1.472
ComplexFFT.c

1.509
ComplexFFT.c

1.536
ComplexFFT.c

1.604
ComplexFFT.c

1.641
ComplexFFT.c

1.678
ComplexFFT.c

1.725
ComplexFFT.c
void LALCOMPLEX16VectorFFT (
    LALStatus *status,
    COMPLEX16Vector *output,
    COMPLEX16Vector *input,
    COMPLEX16FFTPlan *plan
)

Description

This package provides a LAL-style interface with the FFTW fast Fourier transform package [1].

The routines LALCreateForwardComplexFFTPlan() and LALCreateReverseComplexFFTPlan() create plans for computing the forward and reverse FFTs of a given size. The optimum plan is either estimated (reasonably fast) if the measure flag is zero, or measured (can be time-consuming, but gives better performance) if the measure flag is non-zero. The routine LALDestroyComplexFFTPlan() destroys either of these flavours of plans.

The routine LALCOMPLEX8VectorFFT() performs either the forward or reverse FFT depending on the plan. The discrete Fourier transform \( H_k, k = 0 \ldots n - 1 \) of a vector \( h_j, j = 0 \ldots n - 1 \), of length \( n \) is defined by

\[
H_k = \sum_{j=0}^{n-1} h_j e^{-2\pi i j k / n}
\]

and, similarly, the inverse Fourier transform is defined by

\[
h_j = \frac{1}{n} \sum_{k=0}^{n-1} H_k e^{2\pi i j k / n}.
\]

However, the present implementation of the reverse FFT omits the factor of \( 1/n \). The input and output vectors must be distinct.

Operating Instructions

const UINT4 n = 17;
static LALStatus status;
ComplexFFTPlan *pfwd = NULL;
ComplexFFTPlan *prev = NULL;
COMPLEX8Vector *avec = NULL;
COMPLEX8Vector *bvec = NULL;
COMPLEX8Vector *cvec = NULL;

LALCreateForwardComplexFFTPlan( &status, &pfwd, n, 0 );
LALCreateReverseComplexFFTPlan( &status, &prev, n, 0 );
LALCCreateVector( &status, &avec, n );
LALCCreateVector( &status, &bvec, n );
LALCCreateVector( &status, &cvec, n );

<assign data>

LALCOMPLEX8VectorFFT( &status, bvec, avec, pfwd );
LALCOMPLEX8VectorFFT( &status, cvec, bvec, prev );
LALDestroyComplexFFTPlan( &status, &pfwd );
LALDestroyComplexFFTPlan( &status, &prev );
LALDestroyVector( &status, &avec );
LALDestroyVector( &status, &bvec );
LALDestroyVector( &status, &cvec );
Algorithm
The FFTW [1] is used.

Uses

Notes

1. The sign convention used here is the opposite of the definition in Numerical Recipes [1], but agrees with the one used by FFTW [1] and the other LIGO software components.

2. The result of the inverse FFT must be multiplied by $1/n$ to recover the original vector. This is different from the datacondAPI where the factor is applied by default.

3. The size $n$ of the transform can be any positive integer; the performance is $O(n \log n)$. However, better performance is obtained if $n$ is the product of powers of 2, 3, 5, 7, and zero or one power of either 11 or 13. Transforms when $n$ is a power of 2 are especially fast. See Ref. [1].

4. LALMalloc() is used by all the fftw routines.

5. The input and output vectors for +LALCOMPLEX8VectorFFT()+ must be distinct.
10.2.2 Program ComplexFFTTest.c

Tests the routines in ComplexFFT.h.

Usage

ComplexFFTTest [options]
Options:
- h print this message
- q quiet: run silently
- v verbose: print extra information
- d level set lalDebugLevel to level

Description

Exit codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success, normal exit.</td>
</tr>
<tr>
<td>1</td>
<td>Subroutine failed.</td>
</tr>
</tbody>
</table>

Uses

Notes
10.3 Header *TimeFreqFFT.h*

Performs real-to-complex, complex-to-real FFTs and average power spectrum estimation.

**Synopsis**

```c
#include <lal/TimeFreqFFT.h>
```

Perform time-to-frequency and frequency-to-time fast Fourier transforms. Also provides a function to compute mean and median power spectra with user specified windowing.

**Error conditions**

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>1</td>
<td>&quot;Null pointer&quot;</td>
</tr>
<tr>
<td>SIZE</td>
<td>2</td>
<td>&quot;Invalid size&quot;</td>
</tr>
<tr>
<td>RATE</td>
<td>4</td>
<td>&quot;Invalid rate&quot;</td>
</tr>
<tr>
<td>SIGN</td>
<td>4</td>
<td>&quot;Incorrect plan sign&quot;</td>
</tr>
<tr>
<td>ALLOC</td>
<td>16</td>
<td>&quot;Pointer has already been allocated, should be null&quot;</td>
</tr>
<tr>
<td>POSARG</td>
<td>32</td>
<td>&quot;Argument must be positive&quot;</td>
</tr>
<tr>
<td>MALLOC</td>
<td>64</td>
<td>&quot;Malloc failure&quot;</td>
</tr>
<tr>
<td>INCOMP</td>
<td>128</td>
<td>&quot;Incompatible arguments&quot;</td>
</tr>
<tr>
<td>NNUL</td>
<td>256</td>
<td>&quot;Non-null pointer&quot;</td>
</tr>
<tr>
<td>ZSEG</td>
<td>512</td>
<td>&quot;Segment length is zero&quot;</td>
</tr>
<tr>
<td>ZOVR</td>
<td>1024</td>
<td>&quot;Overlap length is zero&quot;</td>
</tr>
<tr>
<td>MISM</td>
<td>2048</td>
<td>&quot;Mismatch between segment, overlap and data length&quot;</td>
</tr>
<tr>
<td>UAVG</td>
<td>4096</td>
<td>&quot;Unknown average power spectrum method&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants `TIMEFREQFFTH_E<name>`, and the status descriptions in `TIMEFREQFFTH_MSGE<name>`. The source code with these messages is in *TimeFreqFFT.h* on line 1.109.

**Types**

**Enum type** *AvgSpecMethod*

This type determines the method the type of average that will be used to compute the power spectrum estimate by the `LALREAL4AverageSpectrum()` function. The function computes a series of (possibly overlapping) power spectra and computes the average using one of the following methods:

- **useUnity** A constant PSD of value unity will be returned independent of the input data given. This is used for testing purposes.
- **useMean** The arithmetic mean of the individual power spectra computed will be used to compute the output power spectrum.
- **useMedian** The median value of the individual power spectra computed will be used to compute the output power spectrum.

**NumberAvgSpecMethods** gives the number of defined methods.

**Structure** *AveragSpectrumParams*

This structure controls the behaviour of the `LALREAL4AverageSpectrum()` function.

- **REAL4Window *window** The windowing function to use when computing the individual power spectra from the input time series. The input time series is broken into smaller time series to compute power spectra for the estimate. The length of these time series is determined by the `length` parameter of the window vector.
UINT4 overlap  The overlap between successive time series used to compute the power spectra.

AvgSpecMethod method  The method of computing the average as describe above.

RealFFTPlan *plan  The FFT plan to be used in the computation of the power spectrum.
10.3.1 Module **TimeFreqFFT.c**

Functions for time to frequency Fourier transforms.

**Prototypes**

```c
void LALTimeFreqRealFFT(
    LALStatus *status,
    COMPLEX8FrequencySeries *freq,
    REAL4TimeSeries *time,
    RealFFTPlan *plan
)

void LALFreqTimeRealFFT(
    LALStatus *status,
    REAL4TimeSeries *time,
    COMPLEX8FrequencySeries *freq,
    RealFFTPlan *plan
)

void LALREAL4AverageSpectrum(
    LALStatus *status,
    REAL4FrequencySeries *fSeries,
    REAL4TimeSeries *tSeries,
    AverageSpectrumParams *params
)

void LALCOMPLEX8AverageSpectrum(
    LALStatus *status,
    COMPLEX8FrequencySeries *fSeries,
    REAL4TimeSeries *tSeries0,
    REAL4TimeSeries *tSeries1,
    AverageSpectrumParams *params
)

void LALTimeFreqComplexFFT(
    LALStatus *status,
    COMPLEX8FrequencySeries *freq,
    COMPLEX8TimeSeries *time,
    ComplexFFTPlan *plan
)

void LALFreqTimeComplexFFT(
    LALStatus *status,
    COMPLEX8TimeSeries *time,
    COMPLEX8FrequencySeries *freq,
    ComplexFFTPlan *plan
)
```
Description

The routines `LALTimeFreqRealFFT()` and `LALTimeFreqComplexFFT()` transform time series \( h_j, 0 \leq j < n \), into a frequency series \( \tilde{h}_k \). For `LALTimeFreqRealFFT()`,

\[
\tilde{h}_k = \Delta t \times H_k \quad \text{for} \quad 0 \leq k \leq \lfloor n/2 \rfloor.
\]

The packing covers the range from dc (inclusive) to Nyquist (inclusive if \( n \) is even). For `LALTimeFreqComplexFFT()`,

\[
\tilde{h}_k = \Delta t \left\{ \begin{array}{ll}
H_{k+(n+1)/2} & \text{for} \quad 0 \leq k < \lfloor n/2 \rfloor, \\
H_{k-n/2} & \text{for} \quad \lfloor n/2 \rfloor \leq k < n.
\end{array} \right.
\]

The packing covers the range from negative Nyquist (inclusive if \( n \) is even) up to (but not including) positive Nyquist. Here \( H_k \) is the DFT of \( h_j \):

\[
H_k = \sum_{j=0}^{n-1} h_j e^{-2\pi ijk/n}.
\]

The units of \( \tilde{h}_k \) are equal to the units of \( h_j \) times seconds.

The routines `LALFreqTimeRealFFT()` and `LALFreqTimeComplexFFT()` perform the inverse transforms from \( \tilde{h}_k \) back to \( h_j \). This is done by shuffling the data, performing the reverse DFT, and multiplying by \( \Delta f \).

The routine `LALREAL4AverageSpectrum()` uses Welch’s method to compute the average power spectrum of the time series stored in the input structure `tSeries` and return it in the output structure `fSeries`. A Welch PSD estimate is defined by an FFT length, overlap length, choice of window function and averaging method. These are specified in the parameter structure; the FFT length is obtained from the length of the `REAL4Window` in the parameters.

On entry the parameter structure `params` must contain a valid `REAL4Window` generated by `LALCreateREAL4Window()`, an integer that determines the overlap as described below and a forward FFT plan for transforming data of the specified window length into the time domain. The method used to compute the average must also be set.

If the length of the window is \( N \), then the FFT length is defined to be \( N/2 - 1 \). The input data of length \( M \) is divided into \( i \) segments which overlap by \( o \), where

\[
i = \frac{M - o}{N - o}.
\]

The PSD of each segment is obtained. The Welch PSD estimate is the average of these \( i \) sub-estimates. The average is computed using the mean or median method, as specified in the parameter structure.

Note: the return PSD estimate is a one-sided power spectral density normalized as defined in the conventions document. When the averaging method is choosen to be mean and the window type Hann, the result is the same as returned by the LDAS dataCondAPI `psd()` action for a real sequence without detrending.

Operating Instructions

```c
const UINT4 n = 65536;
const REAL4 dt = 1.0 / 16384.0;
static LALStatus status; compute average power spectrum
static REAL4TimeSeries x;
static COMPLEX8FrequencySeries X;
static COMPLEX8TimeSeries z;
static COMPLEX8FrequencySeries Z;
RealFFTPlan *fwdRealPlan = NULL;
RealFFTPlan *revRealPlan = NULL;
ComplexFFTPlan *fwdComplexPlan = NULL;
ComplexFFTPlan *revComplexPlan = NULL;
LALSCreateVector( &status, &x.data, n );
LALCCreateVector( &status, &X.data, n / 2 + 1 );
LALCCreateVector( &status, &z.data, n );
LALCCreateVector( &status, &Z.data, n );
LALCreateForwardRealFFTPlan( &status, &fwdRealPlan, n, 0 );
```
LALCreateReverseRealFFTPlan( &status, &revRealPlan, n, 0 );
LALCreateForwardComplexFFTPlan( &status, &fwdComplexPlan, n, 0 );
LALCreateReverseComplexFFTPlan( &status, &revComplexPlan, n, 0 );

x.f0 = 0;
x.deltaT = dt;
x.sampleUnits = lalMeterUnit;
strncpy( x.name, "x", sizeof( x.name ) );

z.f0 = 0;
z.deltaT = dt;
z.sampleUnits = lalVoltUnit;
strncpy( z.name, "z", sizeof( z.name ) );

LALCreateReverseRealFFTPlan( &status, &revRealPlan, n, 0 );
LALCreateForwardComplexFFTPlan( &status, &fwdComplexPlan, n, 0 );
LALCreateReverseComplexFFTPlan( &status, &revComplexPlan, n, 0 );

LALCreateReverseRealFFTPlan( &status, &revRealPlan, n, 0 );
LALCreateForwardComplexFFTPlan( &status, &fwdComplexPlan, n, 0 );
LALCreateReverseComplexFFTPlan( &status, &revComplexPlan, n, 0 );

LALDestroyRealFFTPlan( &status, &fwdRealPlan );
LALDestroyRealFFTPlan( &status, &revRealPlan );
LALDestroyComplexFFTPlan( &status, &fwdComplexPlan );
LALDestroyComplexFFTPlan( &status, &revComplexPlan );
LALCDestroyVector( &status, &Z.data );
LALCDestroyVector( &status, &z.data );
LALCDestroyVector( &status, &x.data );
LALSDestroyVector( &status, &X.data );

Notes

1. The routines do not presently work properly with heterodyned data, i.e., the original time series data should have $f_0$ equal to zero.
10.3.2 Program TimeFreqFFTTest.c

Tests the routines in TimeFreqFFT.h.

Usage

```
TimeFreqFFTTest [options]
Options:
    -h    print this message
    -q    quiet: run silently
    -v    verbose: print extra information
    -d level set lalDebugLevel to level
```

Description

Exit codes

```
<table>
<thead>
<tr>
<th>Code</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success, normal exit.</td>
</tr>
<tr>
<td>1</td>
<td>Subroutine failed.</td>
</tr>
<tr>
<td>2</td>
<td>PSD estimation tolerance exceeded</td>
</tr>
</tbody>
</table>
```

Notes
References


Chapter 11

Package *stats*

This package contains statistical routines.
11.1 Header LALMoment.h

The LALDMoment() and LALSMoment() associated header file.
(S - single precision)
(D - double precision)

Synopsis

#include <lal/LALMoment.h>

This header provides the prototype for the LALDMoment() and LALSMoment() function.

Error codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>1</td>
<td>&quot;NULL pointer.&quot;</td>
</tr>
<tr>
<td>NNUL</td>
<td>2</td>
<td>&quot;Non-NULL pointer.&quot;</td>
</tr>
</tbody>
</table>
| LNTH   | 3    | "Must have more than one data point."
| SEGZ   | 4    | "Invalid number of segments"         |
| NUMZ   | 5    | "Invalid number of points in segment"|
| ALOC   | 6    | "Memory Allocation Error"            |

The status codes in the table above are stored in the constants LALMOMENT_E<name>, and the status descriptions in LALMOMENT_MSGE<name>. The source code with these messages is in LALMoment.h on line 1.88.
11.1.1 Module LALMoment.c

Routine to compute various moments of data.

Prototypes

```c
void LALDMoment (  
    LALStatus *status,  
    REAL8 *result,  
    REAL8Sequence *data,  
    INT4 whichMoment  
)
```

```c
void LALSMoment (  
    LALStatus *status,  
    REAL4 *result,  
    REAL4Sequence *data,  
    INT4 whichMoment  
)
```

Description

The data structure passed in is either a REAL8 or a REAL4 Sequence. The only parameter is which moment to calculate. The function sums the data, calculates the average and then it returns the average for the first moment, it returns the variance for the second moment, and it returns the n-th moment about the mean for higher order moments.

Algorithm

- Find the mean (here referred to as $\overline{x}$).
- Sum, over all the elements, the quantity: $(x[k] - \overline{x})^n$.
- Divide the sum just made by N-1. Call it moment-n
- If n is greater than 2:
  - Sum, over all the elements, the quantity: $(x[k] - \overline{x})^n$.
  - Divide the sum just made by N. Call it moment-n
- Return moment-n

Uses

Determination of a specific moment of a set of data.

Notes

- Moments less than two are not allowed.
- The result structure must be Non-NULL when passed in.
- The function assumes that the length member of the data passed in is correct.
11.1.2 Program LALMomentTest.c

A program to test LALMoment(). - Note only the double precision is tested because both are derived from the same code.

Usage

./LALMomentTest [options]

Options:
- h print usage message
- q quiet: run silently
- v verbose: print extra information
- d level set lalDebugLevel to level

This program tests the function LALMoment(), which calculates the moment of a given data set. First, it tests that the correct error codes are generated for the following error conditions (tests in italics are not performed if LAL_NEDEBUG is set, as the corresponding checks in the code are made using the ASSERT macro):

- null pointer to output structure
- null pointer to input structure
- null pointer to data member of input structure
- null pointer to data member of data member of input structure
- zero length

It then verifies that the correct moment (value and units) is generated for each of the following simple test cases:

1. data set all same value, find moments 2-5.
2. mixed data set, find moments 2-5.
3. evenly distributed data set, find moments 2-5.

For each successful test (both of these valid data and the invalid ones described above), it prints “PASS” to standard output; if a test fails, it prints “FAIL”.

Exit codes

Uses

LALDMoment()
LALSMoment()

Notes
11.2 Header LALCorrelation.h

[One sentence briefly defining scope of the header]

Synopsis

#include <lal/LALCorrelation.h>

[Generic documentation on the header; this is the main place to document any stuff not specific to the module]

Error conditions

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULLP</td>
<td>1</td>
<td>&quot;Null pointer&quot;</td>
</tr>
<tr>
<td>START</td>
<td>2</td>
<td>&quot;Time series do not start simultaneously&quot;</td>
</tr>
<tr>
<td>SAMPLING</td>
<td>3</td>
<td>&quot;Time series are not sampled with the same rate&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants LALCORRELATIONH_E<name>, and the status descriptions in LALCORRELATIONH_MSGE<name>. The source code with these messages is in LALCorrelation.h on line 1.78.

Structures

[Document here any structures defined in the header. Also include any of them in the index; e.g.:]
11.2.1 Module LALCorrelation.c

[A one-line description of the function(s) defined in this module.]

Prototypes

```c
void LALCorrelation( LALStatus *status, OutputCorrelation **out, 
                     const InputCorrelation *input, 
                     const CorrelationParams *params);
```

Description

LALCorrelation is designed to compute a time shifted correlation between two time series given in input-one and input-two. The maximum time shift in nanoseconds is given in params-maxTimeShiftNan. The output consists of a correlation for each time shift in the range out-timeShiftedCorrelation, maximum and minimum values of correlations and corresponding time shifts. The original intention is to use this function to test coincendence bursts found in two detectors for correlation. For this to work one must apply a response function to the raw time series in order to get rid of hardware specific contributions to each time series. The signature of the coincendence event is a clear maximum above some threshold in the graph of correlation vs time shift (no more than 10 ms).

One might, of course, try to use the code to search for any correlations in the data caused by any kind of gravitational waves but that seems to be too computationally expensive.

Algorithm

Just a straightforward computation of correlation for different time shifts. This computation is applied to time series of the length originalLength – maxShift.

Uses

Notes

One must figure out how to prefilter the raw data, what length of time series is appropriate to use, what threshold on the maximum correlation value should be applied to declare a good correlation.
11.2.2 Program LALCorrelationTest.c

[One-line description of test program]

Usage

LALCorrelationTest

Description

Exit codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOM</td>
<td>0</td>
<td>&quot;Nominal exit&quot;</td>
</tr>
<tr>
<td>CHK</td>
<td>1</td>
<td>&quot;Error checking failed to catch bad data&quot;</td>
</tr>
<tr>
<td>FLS</td>
<td>2</td>
<td>&quot;Incorrect answer for valid data&quot;</td>
</tr>
<tr>
<td>IO</td>
<td>3</td>
<td>&quot;Input/output error&quot;</td>
</tr>
<tr>
<td>UP</td>
<td>4</td>
<td>&quot;Unexpected parameter values&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants LALCORRELATIONTESTC_E<name>, and the status descriptions in LALCORRELATIONTESTC_MSGE<name>. The source code with these messages is in LALCorrelationTest.c on line 1.80.

Uses

LALCorrelationTest()
Chapter 12

Package tdfilter

This package covers LAL routines for constructing and applying digital time-domain filters. It is organized under three headers. The ZPGFilter.h header provides routines for manipulating filters in the “zeros, poles, gain” representation, which is typically the simplest way of representing a filter response. These routines create and destroy ZPG filters, and can transform the complex variable used to represent them. The IIRFilter.h header provides routines for creating actual time-domain filters from the ZPG representation, and applying these filters to data. The BandPassTimeSeries.h header provides routines for actual implementation of these utilities to the specific task of high- or low-pass filtering of a data stream. These routines also serve as an example for the more general task of designing time-domain filters with any desired frequency response.

The module ButterworthTimeSeries.c provides specific advice and guidelines for building a numerically stable time-domain filter, but the general procedure is as follows. (1) Decide on the desired filter response, and express it as a rational function of the frequency variable $w = \tan(\pi f/\Delta t)$ (which maps the Nyquist interval onto the positive real axis). (2) Factorize this rational function into zeros and poles, restricting oneself to the upper half of the $w$ complex plane. Assign these to one or more objects of type <datatype>ZPGFilter. (3) Use WToZ<datatype>ZPGFilter() to transform the filter to the more conventional $z = \exp(2\pi if/\Delta t)$ frequency variable. (4) Use the routines in IIRFilter.h to create IIR filters from the ZPG filters, and to apply them to data.
12.1 Header ZPGFilter.h

Provides routines to manipulate ZPG filters.

Synopsis

#include <lal/ZPGFilter.h>

This header covers routines that create, destroy, and transform objects of type <datatype>ZPGFilter, where <datatype> is either COMPLEX8 or COMPLEX16. Generically, these data types can be used to store any rational complex function in a factored form. Normally this function is a filter response, or “transfer function” \( T(z) \), expressed in terms of a complex frequency parameter \( z = \exp(2\pi if\Delta t) \), where \( \Delta t \) is the sampling interval. The rational function is factored as follows:

\[
T(f) = g \times \prod_{k}(z - a_k) \prod_{l}(z - b_l)
\]

where \( g \) is the gain, \( a_k \) are the (finite) zeros, and \( b_l \) are the (finite) poles. It should be noted that rational functions always have the same number of zeros as poles if one includes the point \( z = \infty \); any excess in the number of finite zeros or poles in the rational expression simply indicates that there is a corresponding pole or zero of that order at infinity. It is also worth pointing out that the “gain” is just the overall prefactor of this rational function, and is not necessarily equal to the actual gain of the transfer function at any particular frequency.

Another common complex frequency space is the \( w \)-space, obtained from the \( z \)-space by the bilinear transformation:

\[
w = i \left( \frac{1 - z}{1 + z} \right) = \tan(\pi f \Delta t), \quad z = \frac{1 + iw}{1 - iw}.
\]

Other variables can also be used to represent the complex frequency plane. The <datatype>ZPGFilter structure can be used to represent the transfer function in any of these spaces by transforming the coordinates of the zeros and poles, and incorporating any residual factors into the gain. Care must be taken to include any zeros or poles that are brought in from infinity by the transformation, and to remove any zeros or poles which were sent to infinity. Thus the number of zeros and poles of the <datatype>ZPGFilter is not necessarily constant under transformations! Routines invoking the <datatype>ZPGFilter data types should document which complex variable is assumed.

Error conditions

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUL</td>
<td>1</td>
<td>&quot;Unexpected null pointer in arguments&quot;</td>
</tr>
<tr>
<td>OUT</td>
<td>2</td>
<td>&quot;Output handle points to a non-null pointer&quot;</td>
</tr>
<tr>
<td>MEM</td>
<td>3</td>
<td>&quot;Memory allocation error&quot;</td>
</tr>
<tr>
<td>BAD</td>
<td>4</td>
<td>&quot;Bad filter parameters&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants ZPGFILTERH_E<name>, and the status descriptions in ZPGFILTERH_MSGE<name>. The source code with these messages is in ZPGFilter.h on line 1.103.

Structures
12.1.1 Module CreateZPGFilter.c

Creates ZPG filter objects.

Prototypes

```c
void LALCreateCOMPLEX8ZPGFilter( LALStatus *stat,
                                   COMPLEX8ZPGFilter **output,
                                   INT4 numZeros,
                                   INT4 numPoles )
```

```c
void LALCreateCOMPLEX16ZPGFilter( LALStatus *stat,
                                   COMPLEX16ZPGFilter **output,
                                   INT4 numZeros,
                                   INT4 numPoles )
```

Description

These functions create an object **output, of type COMPLEX8ZPGFilter or COMPLEX16ZPGFilter, having numZeros zeros and numPoles poles. The values of those zeros and poles are not set by these routines (in general they will start out as garbage). The handle passed into the functions must be a valid handle (i.e. output=NULL), but must not point to an existing object (i.e. *output=)=NULL).

Algorithm

Uses

- LALMalloc()
- LALFree()
- LALCreateVector()
- LALCDestroyVector()
- LALZCreateVector()
- LALZDestroyVector()

Notes

Author: Creighton, T. D.

$Id: CreateZPGFilter.c,v 1.12 2007/06/08 14:41:56 bema Exp $
12.1.2 Module DestroyZPGFilter.c

Destroys ZPG filter objects.

Prototypes

```c
void LALDestroyCOMPLEX8ZPGFilter( LALStatus *stat,
                                   COMPLEX8ZPGFilter **input )

void LALDestroyCOMPLEX16ZPGFilter( LALStatus *stat,
                                    COMPLEX16ZPGFilter **input )
```

Description

These functions destroy an object **output of type COMPLEX8ZPGFilter or COMPLEX16ZPGFilter, and set *output to NULL.

Algorithm

Uses

LALFree()
LALCDestroyVector()
LALZDestroyVector()

Notes

Author: Creighton, T. D.

$Id: DestroyZPGFilter.c,v 1.11 2007/06/08 14:41:56 bema Exp $
12.1.3 Module BilinearTransform.c

Transforms the complex frequency coordinate of a ZPG filter.

Prototypes

```c
void LALWToZCOMPLEX8ZPGFilter( LALStatus *stat, COMPLEX8ZPGFilter *filter )
```

```c
void LALWToZCOMPLEX16ZPGFilter( LALStatus *stat, COMPLEX16ZPGFilter *filter )
```

Description

These functions perform an in-place bilinear transformation on an object *filter of type
<datatype>ZPGFilter, transforming from \( w \) to \( z = \frac{1 + iw}{1 - iw} \). Care is taken to ensure that zeros and poles at \( w = \infty \) are correctly transformed to \( z = -1 \), and zeros and poles at \( w = -i \) are correctly transformed to \( z = \infty \). In addition to simply relocating the zeros and poles, residual factors are also incorporated into the gain of the filter (i.e. the leading coefficient of the rational function).

Algorithm

The vectors filter->zeros and filter->poles only record those zeros and poles that have finite value. If one includes the point \( \infty \) on the complex plane, then a rational function always has the same number of zeros and poles: a number num that is the larger of z->zeros->length or z->poles->length. If one or the other vector has a smaller length, then after the transformation that vector will receive additional elements, with a complex value of \( z = -1 \), to bring its length up to num. However, each vector will then lose those elements that previously had values \( w = -i \), (which are sent to \( z = \infty \),) thus possibly decreasing the length of the vector. These routines handle this by simply allocating a new vector for the transformed data, and freeing the old vector after the transformation.

When transforming a zero \( w_k \) on the complex plane, one makes use of the identity:

\[
(w - w_k) = -(w_k + i) \times \frac{z - z_k}{z + 1},
\]

and similarly, when transforming a pole at \( w_k \),

\[
(w - w_k)^{-1} = -(w_k + i)^{-1} \times \frac{z + 1}{z - z_k},
\]

where \( z = \frac{1 + iw}{1 - iw} \) and \( z_k = \frac{1 + iw_k}{1 - iw_k} \). If there are an equal number of poles and zeros being transformed, then the factors of \( z + 1 \) will cancel; otherwise, the remaining factors correspond to the zeros or poles at \( z = -1 \) brought in from \( w = \infty \). The factor \( (z - z_k) \) represents the new position of the transformed zero or pole. The important factor to note, though, is the factor \(-(w_k + i)^{\pm 1} \). This factor represents the change in the gain filter->gain. When \( w_k = -i \), the transformation is slightly different:

\[
(w + i) = \frac{2i}{z + 1};
\]

thus the gain correction factor is \( 2i \) (rather than 0) in this case.

The algorithm in this module computes and stores all the gain correction factors before applying them to the gain. The correction factors are sorted in order of absolute magnitude, and are multiplied together in small- and large-magnitude pairs. In this way one reduces the risk of overrunning the floating-point dynamical range during intermediate calculations.

As a similar precaution, the routines in this module use the algorithm discussed in the VectorOps package whenever they perform complex division, to avoid intermediate results that may be the product of two large numbers. When transforming \( z = \frac{1 + iw}{1 - iw} \), these routines also test for special cases (such as \( w \) purely imaginary) that have qualitatively significant results (\( z \) purely real), so that one doesn’t end up with, for instance, an imaginary part of \( 10^{-12} \) instead of 0.
Uses

LALI4CreateVector()  LALI4DestroyVector()
LALSCreateVector()  LALDCreateVector()
LALSDestroyVector()  LALDDestroyVector()
LALCCreateVector()  LALZCreateVector()
LALCDestroyVector()  LALZDestroyVector()
LALCVectorAbs()  LALZVectorAbs()
LALSHeapIndex()  LALDHeapIndex()

Notes

Author: Creighton, T. D.
$Id: BilinearTransform.c,v 1.13 2007/06/08 14:41:56 bema Exp $
12.2  Header  IIRFilter.h

Provides routines to make and apply IIR filters.

Synopsis

#include <lal/IIRFilter.h>

This header covers routines that create, destroy, and apply generic time-domain filters, given by objects of type <datatype>IIRFilter, where <datatype> is either REAL4 or REAL8.

An IIR (Infinite Impulse Response) filter is a generalized linear causal time-domain filter, in which the filter output \( y_n = y(t_n) \) at any sampled time \( t_n = t_0 + n \Delta t \) is a linear combination of the input \( x \) and output \( y \) at previous sampled times:

\[
y_n = \sum_{k=0}^{M} c_k x_{n-k} + \sum_{l=1}^{N} d_l y_{n-l} .
\]

The coefficients \( c_k \) are called the direct filter coefficients, and the coefficients \( d_l \) are the recursive filter coefficients. The filter order is the larger of \( M \) or \( N \), and determines how far back in time the filter must look to determine its next output. However, the recursive nature of the filter means that the output can depend on input arbitrarily far in the past; hence the name “infinite impulse response”. Nonetheless, for a well-designed, stable filter, the actual filter response to an impulse should diminish rapidly beyond some characteristic timescale.

Note that nonrecursive FIR (Finite Impulse Response) filters are considered a subset of IIR filters, having \( N = 0 \).

For practical implementation, it is convenient to express the bilinear equation above as two linear equations involving an auxiliary sequence \( w \):

\[
w_n = x_n + \sum_{l=1}^{N} d_l w_{n-l} ,
\]

\[
y_n = \sum_{k=0}^{M} c_k w_{n-k} .
\]

The equivalence of this to the first expression is not obvious, but can be proven by mathematical induction. The advantage of the auxiliary variable representation is twofold. First, when one is feeding data point by point to the filter, the filter needs only “remember” the previous \( M \) or \( N \) (whichever is larger) values of \( w \), rather than remembering the previous \( M \) values of \( x \) and the previous \( N \) values of \( y \). Second, when filtering a large stored data vector, the filter response can be computed in place: one first runs forward through the vector replacing \( x \) with \( w \), and then backward replacing \( w \) with \( y \).

Although the IIR filters in these routines are explicitly real, one can consider formally their complex response. A sinusoidal input can thus be written as \( x_n = X \exp(2\pi f n \Delta t) = X z^n \), where \( X \) is a complex amplitude and \( z = \exp(2\pi f \Delta t) \) is a complex parameterization of the frequency. By linearity, the output must also be sinusoidal: \( y_m = Y \exp(2\pi f m \Delta t) = Y z^m \). Putting these into the bilinear equation, one can easily compute the filter’s complex transfer function:

\[
T(z) = \frac{Y}{X} = \frac{\sum_{k=0}^{M} c_k z^{-k}}{1 - \sum_{l=1}^{N} d_l z^{-l}}
\]

This can be readily converted to and from the “zeros, poles, gain” representation of a filter, which expresses \( T(z) \) as a factored rational function of \( z \).

It should also be noted that, in the routines covered by this header, I have adopted the convention of including a redundant recursive coefficient \( d_0 \), in order to make the indexing more intuitive. For formal correctness \( d_0 \) should be set to \(-1\), although the filtering routines never actually use this coefficient.

Error conditions

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUL</td>
<td>1</td>
<td>&quot;Unexpected null pointer in arguments&quot;</td>
</tr>
<tr>
<td>OUT</td>
<td>2</td>
<td>&quot;Output handle points to a non-null pointer&quot;</td>
</tr>
<tr>
<td>MEM</td>
<td>3</td>
<td>&quot;Memory allocation error&quot;</td>
</tr>
<tr>
<td>PAIR</td>
<td>4</td>
<td>&quot;Input has unpaired nonreal poles or zeros&quot;</td>
</tr>
</tbody>
</table>
The status codes in the table above are stored in the constants `IIRFILTERH_E<name>`, and the status descriptions in `IIRFILTERH_MSGE<name>`. The source code with these messages is in `IIRFilter.h` on line 1.131.

**Types**

**Structure `<datatype>IIRFilter`**

This structure stores the direct and recursive filter coefficients, as well as the history of the auxiliary sequence $w$. `<datatype>` may be `REAL4` or `REAL8`. The length of the history vector gives the order of the filter. The fields are:

- **const CHAR *name** A user-assigned name.
- **REAL8 deltaT** The sampling time interval of the filter. If $\leq 0$, it will be ignored (i.e. it will be taken from the data stream).
- **<datatype>Vector *directCoef** The direct filter coefficients.
- **<datatype>Vector *recursCoef** The recursive filter coefficients.
- **<datatype>Vector *history** The previous values of $w$. 

---

Author: Creighton, T. D.

$Id: IIRFilter.h,v 1.13 2007/06/08 14:41:56 bema Exp $
12.2.1 Module CreateIIRFilter.c

Creates IIR filter objects.

Prototypes

```c
void LALCreateREAL4IIRFilter( LALStatus *stat, REAL4IIRFilter **output, COMPLEX8ZPGFilter *input )

void LALCreateREAL8IIRFilter( LALStatus *stat, REAL8IIRFilter **output, COMPLEX16ZPGFilter *input )
```

Description

These functions create an object **output** of type `<datatype>`IIRFilter, where `<datatype>` is REAL4 or REAL8. The filter coefficients are computed from the zeroes, poles, and gain of an input object *input*. The sampling time interval is taken directly from `input->deltaT`. The ZPG filter should express the factored transfer function in the $z = \exp(2\pi if)$ plane. Initially the output handle must be a valid handle (**output** ≠ NULL) but should not point to an existing object (**output** = NULL)

Algorithm

An IIR filter is a real time-domain filter, which imposes certain constraints on the zeros, poles, and gain of the filter transfer function. The function `Create<datatype>`IIRFilter() deals with the constraints either by aborting if they are not met, or by adjusting the filter response so that they are met. In the latter case, warning messages will be issued if the external parameter `lalDebugLevel` is set to allow such messages. The specific constraints, and how they are dealt with, are as follows:

First, the filter must be causal; that is, the output at any time can be generated entirely from the input at previous times. In practice this means that the number of (finite) poles in the $z$ plane must equal or exceed the number of (finite) zeros. If this is not the case, `Create<datatype>`IIRFilter() will add additional poles at $z = 0$. Effectively this is just adding a delay to the filter response in order to make it causal.

Second, the filter should be stable, which means that all poles should be located on or within the circle $|z| = 1$. This is not enforced by `Create<datatype>`IIRFilter(), which can be used to make unstable filters; however, warnings will be issued. (In some sense the first condition is a special case of this one, since a transfer function with more zeros than poles actually has corresponding poles at infinity.)

Third, the filter must be physically realizable; that is, the transfer function should expand to a rational function of $z$ with real coefficients. Necessary and sufficient conditions for this are that the gain be real, and that all zeros and poles either be real or come in complex conjugate pairs. The routine `Create<datatype>`IIRFilter() enforces this by using only the real part of the gain, and only the real or positive-imaginary zeros and poles; it assumes that the latter are paired with negative-imaginary conjugates. The routine will abort if this assumption results in a change in the given number of zeros or poles. If `lalDebugLevel` is set to allow warnings, the routine will actually check to see that each pair of nonreal poles or zeros are in fact complex conjugates, and will issue a warning if an unmatched pair is detected; however, the algorithm will then simply proceed as if the negative-imaginary points were relocated to the “correct” positions.

The code associated with the warning messages is potentially rather cumbersome for production algorithms; therefore, the value of `lalDebugLevel` is tested before performing any other tests associated with warning messages. Furthermore, this code block is surrounded with compiler directives to exclude the code entirely if the module is compiled with the NDEBUG flag set.

Uses

```c
lalDebugLevel
LALWarning() LALPrintError()
LALMalloc() LALFree()
LALSCreateVector() LALSDestroyVector()
LALDCreateVector() LALDDestroyVector()
```
Notes
12.2.2 Module DestroyIIRFilter.c

Destroys IIR filter objects.

Prototypes

```c
void
LALDestroyREAL4IIRFilter( LALStatus *stat,
                         REAL4IIRFilter **input )
```

```c
void
LALDestroyREAL8IIRFilter( LALStatus *stat,
                         REAL8IIRFilter **input )
```

Description

These functions destroy an object **input of type REAL4IIRFilter or REAL8IIRFilter, and set *input to NULL.

Algorithm

Uses

```c
void LALFree()
void LALSDestroyVector()
void LALDDestroyVector()
```

Notes
12.2.3 Module **IIRFilter.c**

Computes an instant-by-instant IIR filter response.

**Prototypes**

```c
void LALIIRFilterREAL4( LALStatus *stat, REAL4 *output, REAL4 input, REAL4IIRFilter *filter )
```

```c
void LALIIRFilterREAL8( LALStatus *stat, REAL8 *output, REAL8 input, REAL8IIRFilter *filter )
```

```c
REAL4 LALSIIRFilter( REAL4 x, REAL4IIRFilter *filter )
```

```c
REAL8 XLALIIRFilterREAL8( REAL8 x, REAL8IIRFilter *filter )
```

```c
REAL4 XLALIIRFilterREAL4( REAL4 x, REAL8IIRFilter *filter )
```

**Description**

These functions pass a time-domain datum to an object *filter* of type `REAL4IIRFilter` or `REAL8IIRFilter`, and return the filter response. This is done using the auxiliary data series formalism described in the header `IIRFilter.h`.

There are two pairs of routines in this module. The functions `IIRFilterReal4()` and `LALIIRFilterREAL8()` conform to the LAL standard, with status handling and error trapping; the input datum is passed in as `input` and the response is returned in `output`. The functions `LALSIIRFilter()` and `LALDIIRFilter()` are non-standard lightweight routines, which may be more suitable for multiple callings from the inner loops of programs; they have no status handling or error trapping. The input datum is passed in by the variable `x`, and the response is returned through the function’s return statement.

**Algorithm**

**Uses**

**Notes**
12.2.4 Module **IIRFilterVector.c**

Applies an IIR filter to a data stream.

**Prototypes**

```c
void LALIIRFilterREAL4Vector( LALStatus *stat,
                               REAL4Vector *vector,
                               REAL4IIRFilter *filter )

void LALIIRFilterREAL8Vector( LALStatus *stat,
                               REAL8Vector *vector,
                               REAL8IIRFilter *filter )

void LALDIIRFilterREAL4Vector( LALStatus *stat,
                               REAL4Vector *vector,
                               REAL8IIRFilter *filter )
```

**Description**

These functions apply a generic time-domain filter given by an object `*filter` of type `REAL4IIRFilter` or `REAL8IIRFilter` to a list `*vector` of data representing a time series. This is done in place using the auxiliary data series formalism described in `IIRFilter.h`, so as to accommodate potentially large data series. To filter a piece of a larger dataset, the calling routine may pass a vector structure whose data pointer and length fields specify a subset of a larger vector.

The routine `LALDIIRFilterREAL4Vector()` applies a double-precision filter to single-precision data. It makes a single pass through the data, continuously updating the filter history at each step rather than storing the auxiliary array in-place. This reduces roundoff error by keeping all intermediate results to double-precision.

**Algorithm**

The implementation of `LALDIIRFilterREAL4Vector()` not only has lower truncation errors than `LALIIRFilterREAL4Vector()`, but also appears to be more computationally efficient, for reasons I have not yet determined; see the documentation for `IIRFilterTest.c`. These combine to suggest that `LALDIIRFilterREAL4Vector()` is the better overall algorithm for filtering `REAL4Vector`s.

**Uses**

- `LALMalloc()`
- `LALFree()`

**Notes**

Author: Creighton, T. D.

$Id: IIRFilterVector.c,v 1.12 2007/06/08 14:41:56 bema Exp $
12.2.5 Module IIRFilterVectorR.c
Applies a time-reversed IIR filter to a data stream.

Prototypes

```c
void LALIIRFilterREAL4VectorR( LALStatus *stat,
                                REAL4Vector *vector,
                                REAL4IIRFilter *filter )

void LALIIRFilterREAL8VectorR( LALStatus *stat,
                                REAL8Vector *vector,
                                REAL8IIRFilter *filter )

void LALDIIRFilterREAL4VectorR( LALStatus *stat,
                                 REAL4Vector *vector,
                                 REAL8IIRFilter *filter )
```

Description

These functions apply a generic time-domain filter `*filter` to a time series `*vector`, as with the routines `LALIIRFilterREAL4Vector()`, `LALIIRFilterREAL8Vector()`, and `LALDIIRFilterREAL4Vector()`, but do so in a time-reversed manner. By successively applying normal and time-reversed IIR filters to the same data, one squares the magnitude of the frequency response while canceling the phase shift. This can be significant when one wishes to preserve phase correlations across wide frequency bands.

Algorithm

Because these filter routines are inherently acausal, the `filter->history` vector is meaningless and unnecessary. These routines neither use nor modify this data array. They effectively treat the “future” data as zero.

(An alternative implementation would be to assume that the filter “history”, when invoked by these routines, stores the future values of the auxiliary sequence. This would allow a large vector to be broken into chunks and time-reverse filtered, yielding the same result as if the whole vector had been time-reverse filtered. I can switch to this implementation if there is any demand for it.)

Uses

Notes
12.2.6 Program IIRFilterTest.c

Tests the routines in IIRFilter.h.

Usage


Description

This program generates a time series vector, and passes it through a third-order Butterworth low-pass filter. By default, running this program with no arguments simply passes an impulse function to the filter routines, producing no output. All filter parameters are set from #defined constants. The following option flags are accepted:

- **-f** Specifies which filter(s) to be used: filtertag is a token containing one or more character codes from a to f and/or from A to D, each corresponding to a different filter:

  a = LALSIIIRFilter()
  b = LALIIRFilterREAL4()
  c = LALIIRFilterREAL4Vector()
  d = LALIIRFilterREAL4VectorR()
  e = LALDIIRFilterREAL4Vector()
  f = LALDIIRFilterREAL4VectorR()
  A = LALDIIRFilter()
  B = LALIIRFilterREAL8()
  C = LALIIRFilterREAL8Vector()
  D = LALIIRFilterREAL8VectorR()

  If not specified, -f abcd is assumed.

- **-o** Prints the input and output vectors to data files: out.0 stores the initial impulse, out.c the response computed using the filter with character code c (above). If not specified, the routines are exercised, but no output is written.

- **-t** Causes IIRFilterTest to fill the time series vector with Gaussian random deviates, and prints execution times for the various filter subroutines to stdout. To generate useful timing data, the default size of the time vector is increased (unless explicitly set, below).

- **-d** Changes the debug level from 0 to the specified value debuglevel.

- **-n** Sets the size of the time vectors to npts. If not specified, 4096 points are used (4194304 if the -t option was also given).

- **-r** Applies each filter to the data reps times instead of just once.

- **-w** Sets the characteristic frequency of the filter to freq in the w-plane (described in ZPGFilter.h). If not specified, -w 0.01 is assumed (i.e. a characteristic frequency of 2% of Nyquist).

Exit codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NORM</td>
<td>0</td>
<td>&quot;Normal exit&quot;</td>
</tr>
<tr>
<td>SUB</td>
<td>1</td>
<td>&quot;Subroutine failed&quot;</td>
</tr>
<tr>
<td>ARG</td>
<td>2</td>
<td>&quot;Error parsing arguments&quot;</td>
</tr>
<tr>
<td>BAD</td>
<td>3</td>
<td>&quot;Bad argument value&quot;</td>
</tr>
<tr>
<td>FILE</td>
<td>4</td>
<td>&quot;Could not create output file&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants IIRFILTERTESTC_<name>, and the status descriptions in IIRFILTERTESTC_MSGE_<name>. The source code with these messages is in IIRFilterTest.c on line 1.106.
Algorithm

A third-order Butterworth low-pass filter is defined by the following power response function:

\[ |T(w)|^2 = \frac{1}{1 + (w/w_c)^6}, \]

where the frequency parameter \( w = \tan(\pi f/\Delta t) \) maps the Nyquist interval onto the entire real axis, and \( w_c \) is the (transformed) characteristic frequency set by the -w option above. A stable time-domain filter has poles with \( |z| < 1 \) in the z-plane representation, which means that the \( w \)-plane poles should have \( \Im(w) > 0 \). We construct a transfer function using only the positive-imaginary poles of the power response function:

\[ T(w) = \frac{i w c^3}{\prod_{k=0}^{2} (w - w c e^{(2k+1)\pi/6})}, \]

where we have chosen a phase coefficient \( i \) in the numerator in order to get a purely real DC response. This ensures that the \( z \)-plane representation of the filter will have a real gain, resulting in a physically-realizable IIR filter.

The poles and gain of the transfer function \( T(w) \) are simply read off of the equation above, and are stored in a COMPLEX8ZPGFilter. This is transformed from the \( w \)-plane to the \( z \)-plane representation using LALWToZCOMPLEX8ZPGFilter(), and then used to create an IIR filter with LALCreateREAL4IIRFilter(). This in turn is used by the routines LALSIIRFilter(), LALIIRFilterREAL4(), LALIIRFilterREAL4Vector(), and LALIIRFilterREAL4VectorR() to filter a data vector containing either a unit impulse or white Gaussian noise (for more useful timing information).

Sample output

Running this program on a 1.3 GHz Intel machine with no optimization produced the following typical timing information:

```
> IIRFilterTest -r 5 -t -f abcdefABCD
Filtering 4194304 points 5 times:
Elapsed time for LALSIIRFilter(): 1.39 s
Elapsed time for LALDIIRFilter(): 1.79 s
Elapsed time for LALIIRFilterREAL4(): 2.86 s
Elapsed time for LALIIRFilterREAL8(): 3.25 s
Elapsed time for LALIIRFilterREAL4Vector(): 1.52 s
Elapsed time for LALIIRFilterREAL8Vector(): 2.13 s
Elapsed time for LALIIRFilterREAL4VectorR(): 1.33 s
Elapsed time for LALIIRFilterREAL8VectorR(): 1.96 s
Elapsed time for LALDIIRFilterREAL4VectorR(): 1.12 s
Elapsed time for LALDIIRFilterREAL8VectorR(): 1.06 s
```

From these results it is clear that the mixed-precision vector filtering routines are the most efficient, outperforming even the purely single-precision vector filtering routines by 20%–30%. This was unanticipated; by my count the main inner loop of the single-precision routines contain \( 2M + 2N + 1 \) dereferences and \( 2M + 2N - 3 \) floating-point operations per vector element, where \( M \) and \( N \) are the direct and recursive filter orders, whereas the mixed-precision routines contain \( 2\max\{M, N\} + 2M + 2 \) dereferences and \( 2M + 2N - 1 \) floating-point operations per element. However, most of the dereferences in the mixed-precision routines are to short internal arrays rather than to the larger data vector, which might cause some speedup.

Running the same command with the -t flag replaced with -o generates files containing the impulse response of the filters. The frequency-domain impulse response is shown in Fig. 12.1. This shows the steady improvement in truncation error from single- to mixed- to double-precision filtering.
### Uses

<table>
<thead>
<tr>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>lalDebugLevel</td>
<td>LALCreateVector()</td>
</tr>
<tr>
<td>LALSCreateVector()</td>
<td>LALDCreateVector()</td>
</tr>
<tr>
<td>LALSDestroyVector()</td>
<td>LALDDestroyVector()</td>
</tr>
<tr>
<td>LALCreateRandomParams()</td>
<td>LALDestroyRandomParams()</td>
</tr>
<tr>
<td>LALNormalDeviates()</td>
<td>LALPrintError()</td>
</tr>
<tr>
<td>LALCreateCOMPLEX8ZPGFilter()</td>
<td>LALCreateCOMPLEX16ZPGFilter()</td>
</tr>
<tr>
<td>LALDestroyCOMPLEX8ZPGFilter()</td>
<td>LALDestroyCOMPLEX16ZPGFilter()</td>
</tr>
<tr>
<td>LALCreateREAL4IIRFilter()</td>
<td>LALCreateREAL8IIRFilter()</td>
</tr>
<tr>
<td>LALDestroyREAL4IIRFilter()</td>
<td>LALDestroyREAL8IIRFilter()</td>
</tr>
<tr>
<td>LALWToZCOMPLEX8ZPGFilter()</td>
<td>LALWToZCOMPLEX16ZPGFilter()</td>
</tr>
<tr>
<td>LALSIIRFilter()</td>
<td>LALDIIRFilter()</td>
</tr>
<tr>
<td>LALIIRFilterREAL4()</td>
<td>LALIIRFilterREAL8()</td>
</tr>
<tr>
<td>LALIIRFilterREAL4Vector()</td>
<td>LALIIRFilterREAL8Vector()</td>
</tr>
<tr>
<td>LALIIRFilterREAL4VectorR()</td>
<td>LALIIRFilterREAL8VectorR()</td>
</tr>
<tr>
<td>LALDIIRFilterREAL4Vector()</td>
<td>LALDIIRFilterREAL4VectorR()</td>
</tr>
</tbody>
</table>

### Notes

Author: Creighton, T. D.

$Id: BandPassTest.c,v 1.16 2007/06/08 14:41:56 bema Exp $
12.3 Header BandPassTimeSeries.h

Provides routines to low- or high-pass filter a time series.

Synopsis

```
#include <lal/BandPassTimeSeries.h>
```

This header covers routines that apply a time-domain low- or high-pass filter to a data series of type `<datatype>TimeSeries`. Further documentation is given in the individual routines’ modules.

Error conditions

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>1</td>
<td>&quot;Unexpected null pointer in arguments&quot;</td>
</tr>
<tr>
<td>BAD</td>
<td>2</td>
<td>&quot;Bad filter parameters&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants `BANDPASSTIMESERIESH_<name>`, and the status descriptions in `BANDPASSTIMESERIESH_MSGE<name>`. The source code with these messages is in `BandPassTimeSeries.h` on line 1.65.

Types

Structure PassBandParamStruc

This structure stores data used for constructing a low- or high-pass filter: either the order and characteristic frequency of the filter, or the frequencies and desired attenuations at the ends of some transition band. In the latter case, a nonzero filter order parameter `n` indicates a maximum allowed order. The fields are:

```
CHAR *name A user-assigned name.

INT4 n The maximum desired filter order (actual order may be less if specified attenuations do not require a high order).

REAL8 f1, f2 The reference frequencies of the transition band.

REAL8 a1, a2 The minimal desired attenuation factors at the reference frequencies.
```
12.3.1 Module *ButterworthTimeSeries.c*

Applies a low- or high-pass Butterworth filter to a time series.

**Prototypes**

```c
void LALButterworthREAL4TimeSeries( LALStatus *stat,
                                     REAL4TimeSeries *series,
                                     PassBandParamStruc *params )
```

```c
void LALButterworthREAL8TimeSeries( LALStatus *stat,
                                     REAL8TimeSeries *series,
                                     PassBandParamStruc *params )
```

```c
void LALDButterworthREAL4TimeSeries( LALStatus *stat,
                                     REAL4TimeSeries *series,
                                     PassBandParamStruc *params )
```

**Description**

These routines perform an in-place time-domain band-pass filtering of a data sequence *series*, using a Butterworth filter generated from parameters *params*. The routines construct a filter with the square root of the desired amplitude response, which it then applied to the data once forward and once in reverse. This gives the full amplitude response with little or no frequency-dependent phase shift.

The routine `LALDButterworthREAL4TimeSeries()` applies a double-precision filter to single-precision data, using `LALDIIRFilterREAL4Vector()` and `LALDIIRFilterREAL4VectorR()`.

**Algorithm**

The frequency response of a Butterworth low-pass filter is easiest to express in terms of the transformed frequency variable \( w = \tan(\pi f \Delta t) \), where \( \Delta t \) is the sampling interval (i.e. \( \text{series->deltaT} \)). In this parameter, then, the power response (attenuation) of the filter is:

\[
|R|^2 = \sqrt{a} = \frac{1}{1 + \left(\frac{w}{w_c}\right)^2}^{2n},
\]

where \( n \) is the filter order and \( w_c \) is the characteristic frequency. We have written the attenuation as \( \sqrt{a} \) to emphasize that the full attenuation \( a \) is achieved only after filtering twice (once forward, once in reverse). Similarly, a Butterworth high-pass filter is given by

\[
|R|^2 = \sqrt{a} = \frac{1}{1 + \left(\frac{w_c}{w}\right)^2}^{2n}.
\]

If one is given a filter order \( n \), then the characteristic frequency can be determined from the attenuation at some any given frequency. Alternatively, \( n \) and \( w_c \) can both be computed given attenuations at two different frequencies.

Frequencies in *params* are assumed to be real frequencies \( f \) given in the inverse of the units used for the sampling interval \( \text{series->deltaT} \). In order to be used, the pass band parameters must lie in the ranges given below; if a parameter lies outside of its range, then it is ignored and the filter is calculated from the remaining parameters. If too many parameters are missing, the routine will fail. The acceptable parameter ranges are:

- \( \text{params->nMax} = 1, 2, \ldots \)
- \( \text{params->f1, f2} \in (0, \{2 \times \text{series->deltaT}\}^{-1}) \)
- \( \text{params->a1, a2} \in (0, 1) \)
If both pairs of frequencies and amplitudes are given, then \(a_1, a_2\) specify the minimal requirements on the attenuation of the filter at frequencies \(f_1, f_2\). Whether the filter is a low- or high-pass filter is determined from the relative sizes of these parameters. In this case the \(n_{\text{Max}}\) parameter is optional; if given, it specifies an upper limit on the filter order. If the desired attenuations would require a higher order, then the routine will sacrifice performance in the stop band in order to remain within the specified \(n_{\text{Max}}\).

If one of the frequency/attenuation pairs is missing, then the filter is computed using the remaining pair and \(n_{\text{Max}}\) (which must be given). The filter is taken to be a low-pass filter if \(f_1, a_1\) are given, and high-pass if \(f_2, a_2\) are given. If only one frequency and no corresponding attenuation is specified, then it is taken to be the characteristic frequency (i.e. the corresponding attenuation is assumed to be \(\sqrt{a} = 1/2\)). If none of these conditions are met, the routine will return an error.

The following table summarizes the decision algorithm. A • symbol indicates that the parameter is specified in the range given above. A ◦ symbol indicates that the parameter is “not given”, i.e. not specified in the valid range.

<table>
<thead>
<tr>
<th>(n_{\text{Max}})</th>
<th>(f_1)</th>
<th>(a_1)</th>
<th>(f_2)</th>
<th>(a_2)</th>
<th>Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>◦</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>Type of filter (low- or high-pass), (w_c), and (n) are computed from all four transition-band parameters.</td>
</tr>
<tr>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>Ditto, but if the resulting (n &gt; n_{\text{Max}}), (w_c) is computed from (n_{\text{Max}}) and the ((f, a)) pair with the larger (a).</td>
</tr>
<tr>
<td>•</td>
<td>•</td>
<td>•</td>
<td>◦</td>
<td>◦</td>
<td>Low-pass filter; (w_c) is computed from (n_{\text{Max}}, f_1), and (a_1).</td>
</tr>
<tr>
<td>•</td>
<td>•</td>
<td>◦</td>
<td>•</td>
<td>◦</td>
<td>Ditto; (a_2) is ignored.</td>
</tr>
<tr>
<td>•</td>
<td>◦</td>
<td>•</td>
<td>•</td>
<td>◦</td>
<td>Ditto; (f_2) is ignored.</td>
</tr>
<tr>
<td>•</td>
<td>◦</td>
<td>◦</td>
<td>○</td>
<td>○</td>
<td>Low-pass filter; (w_c) is computed as above with (a_1) treated as 1/4.</td>
</tr>
<tr>
<td>•</td>
<td>◦</td>
<td>○</td>
<td>◦</td>
<td>○</td>
<td>Ditto; (a_2) is ignored.</td>
</tr>
<tr>
<td>•</td>
<td>○</td>
<td>◦</td>
<td>•</td>
<td>◦</td>
<td>High-pass filter; (w_c) is computed from (n_{\text{Max}}, f_2), and (a_2).</td>
</tr>
<tr>
<td>•</td>
<td>○</td>
<td>◦</td>
<td>◦</td>
<td>•</td>
<td>Ditto; (a_1) is ignored.</td>
</tr>
<tr>
<td>•</td>
<td>○</td>
<td>◦</td>
<td>◦</td>
<td>◦</td>
<td>Ditto; (f_1) is ignored.</td>
</tr>
<tr>
<td>•</td>
<td>•</td>
<td>◦</td>
<td>○</td>
<td>○</td>
<td>High-pass filter; (w_c) is computed as above with (a_2) treated as 1/4.</td>
</tr>
<tr>
<td>•</td>
<td>◦</td>
<td>•</td>
<td>○</td>
<td>○</td>
<td>Ditto; (a_1) is ignored.</td>
</tr>
<tr>
<td>Other</td>
<td>Subroutine returns an error.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Once an order \(n\) and characteristic frequency \(w_c\) are known, the zeros and poles of a ZPG filter are readily determined. A stable, physically realizable Butterworth filter will have \(n\) poles evenly spaced on the upper half of a circle of radius \(w_c\); that is,

\[
R = \frac{(-iw_c)^n}{\prod_{k=0}^{n-1}(w - w_c e^{2\pi i (k+1/2)/n})}
\]

for a low-pass filter, and

\[
R = \frac{w^n}{\prod_{k=0}^{n-1}(w - w_c e^{2\pi i (k+1/2)/n})}
\]

for a high-pass filter. By choosing only poles on the upper-half plane, one ensures that after transforming to \(z\) the poles will have \(|z| < 1\). Furthermore, the phase factor \((-i)^n\) in the numerator of the low-pass filter is chosen so that the DC response is purely real; this ensures that the response function in the \(z\)-plane will have a real gain factor, and the resulting IIR filter will be physically realizable. The high-pass filter has a purely real response at Nyquist (\(w \to \infty\)), which similarly gives a physical IIR filter.

Although higher orders \(n\) would appear to produce better (i.e. sharper) filter responses, one rapidly runs into numerical errors, as one ends up adding and subtracting \(n\) large numbers to obtain small filter responses. One way around this is to break the filter up into several lower-order filters. The routines in this module do just that. Poles are paired up across the imaginary axis, (and combined with pairs of zeros at \(w = 0\) for high-pass filters,) to form \([n/2]\) second-order filters. If \(n\) is odd, there will be an additional first-order filter, with one pole at \(w = iw_c\) (and one zero at \(w = 0\) for a high-pass filter).

Each ZPG filter in the \(w\)-plane is first transformed to the \(z\)-plane by a bilinear transformation, and is then used to construct a time-domain IIR filter. Each filter is then applied to the time series. As mentioned in the description above, the filters are designed to give an overall amplitude response that is the square root of the desired attenuation; however, each time-domain filter is applied to the data stream twice: once in the normal sense, and once in the time-reversed sense. This gives the full attenuation with very little frequency-dependent phase shift.
Uses

`lalDebugLevel`  
`LALPrintError()`  
`LALCreateREAL4IIRFilter()`  
`LALCreateCOMPLEX8ZPGFilter()`  
`LALDestroyREAL4IIRFilter()`  
`LALDestroyCOMPLEX8ZPGFilter()`  
`LALWToZCOMPLEX8ZPGFilter()`  
`LALIIRFilterREAL4Vector()`  
`LALIIRFilterREAL4VectorR()`  

`LALWarning()`  
`LALCreateREAL8IIRFilter()`  
`LALCreateCOMPLEX16ZPGFilter()`  
`LALDestroyREAL8IIRFilter()`  
`LALDestroyCOMPLEX16ZPGFilter()`  
`LALWToZCOMPLEX16ZPGFilter()`  
`LALIIRFilterREAL8Vector()`  
`LALIIRFilterREAL8VectorR()`
12.3.2 Program BandPassTest.c

Tests time-domain high- and low-pass filters.

Usage

```bash
BandPassTest [-d debuglevel] [-i infile | -n npts dt offset] [-o outfile]
    [-f f1 f2 a1 a2 order]
```

Description

This program applies a Butterworth time-domain low-pass or high-pass filter to a time series, using the routine `LALDButterworthREAL4TimeSeries()`. The following option flags are accepted:

- `-d` Changes the default debug level from 0 to `debuglevel`.
- `-i` Reads the input time series from `infile` using the routine `LALSReadTSeries()`; see `StreamInput.h` for a description of the file format.
- `-n` Generates an input time series of length `npts` and sampling interval `dt`, containing just an impulse at sample index `offset`. If the `-i` option is also given, it overrides this option. If neither are given, `-n 4096 1.0 1024` is assumed.
- `-o` Writes the output time series to `outfile`, using the routine `LALSWriteTSeries()`; see `StreamOutput.h` for a description of the file format. If not specified, the routines are exercised, but no output is written.
- `-f` Sets the filter to have attenuation `a1` and `a2` at frequencies `f1` and `f2`, with a maximum filter order of `order`; see `ButterworthTimeSeries.c` for a description of how these values are interpreted. If not specified, `-f 0.01 0.015 0.9 0.1 20` is assumed.

Exit codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NORM</td>
<td>0</td>
<td>&quot;Normal exit&quot;</td>
</tr>
<tr>
<td>SUB</td>
<td>1</td>
<td>&quot;Subroutine failed&quot;</td>
</tr>
<tr>
<td>ARG</td>
<td>2</td>
<td>&quot;Error parsing arguments&quot;</td>
</tr>
<tr>
<td>BAD</td>
<td>3</td>
<td>&quot;Bad argument values&quot;</td>
</tr>
<tr>
<td>FILE</td>
<td>4</td>
<td>&quot;Could not open file&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants `BANDPASSTESTC_E<name>`, and the status descriptions in `BANDPASSTESTC_MSGE<name>`. The source code with these messages is in `BandPassTest.c` on line 1.79.

Uses

- `lalDebugLevel`
- `LALPrintError()`
- `LALSprintf()`
- `LALSCreateVector()`
- `LALSDestroyVector()`
- `LALSReadTSeries()`
- `LALSWriteTSeries()`
- `LALDButterworthREAL4TimeSeries()`
- `LALCheckMemoryLeaks()`

Notes
12.4 Header \texttt{LPC.h}

Functions for linear predictor filters.

Synopsis

\#include <lal/LPC.h>

Error Conditions

<table>
<thead>
<tr>
<th>name</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>1</td>
<td>&quot;invalid input&quot;</td>
</tr>
<tr>
<td>MEM</td>
<td>2</td>
<td>&quot;memory error&quot;</td>
</tr>
<tr>
<td>NUM</td>
<td>3</td>
<td>&quot;numerical error&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants \texttt{LPCH\_E<name>}, and the status descriptions in \texttt{LPCH\_MSGE<name>}. The source code with these messages is in \texttt{LPC.h} on line 1.60.
Compute the coefficients of a linear predictor filter.

Prototype

```c
void LALLPC(LALStatus *status,
    REAL4Vector *aout, /* returned filter coefficients */
    REAL4Vector *x, /* training data */
    UINT4 p        /* filter order */
) {
```

Description

Train a FIR filter `aout` of order `p` on the data `x`.

Uses

... Stabilizes a polynomial.

Prototype

```c
void LALPolystab(LALStatus *status,
    REAL4Vector *a)
```

Description

Reflects poles and zeroes of `a` inside the complex unit circle.

Uses

...
Chapter 13

Package *utilities*

This package contains various numerical utilities for use in LAL.
13.1 Header Random.h

Generates random numbers.

Synopsis

```c
#include <lal/Random.h>
```

This header covers the routines for generating random numbers.

Error conditions

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>1</td>
<td>&quot;Null pointer&quot;</td>
</tr>
<tr>
<td>NNUL</td>
<td>2</td>
<td>&quot;Non-null pointer&quot;</td>
</tr>
<tr>
<td>SIZE</td>
<td>4</td>
<td>&quot;Invalid size&quot;</td>
</tr>
<tr>
<td>LNTH</td>
<td>8</td>
<td>&quot;Must have more than one data point&quot;</td>
</tr>
<tr>
<td>SEGZ</td>
<td>16</td>
<td>&quot;Invalid number of segments&quot;</td>
</tr>
<tr>
<td>NUMZ</td>
<td>32</td>
<td>&quot;Invalid number of points in segment&quot;</td>
</tr>
<tr>
<td>ALOC</td>
<td>64</td>
<td>&quot;Memory Allocation Error&quot;</td>
</tr>
<tr>
<td>INIT</td>
<td>128</td>
<td>&quot;Params must be initialized with CreateParams first&quot;</td>
</tr>
<tr>
<td>ZERO</td>
<td>256</td>
<td>&quot;Output Vector length must be greater than zero&quot;</td>
</tr>
<tr>
<td>SEED</td>
<td>512</td>
<td>&quot;Improper seed value&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants RANDOMH_E<name>, and the status descriptions in RANDOMH_MSGE<name>. The source code with these messages is in Random.h on line 1.82.

Structures

```c
typedef struct tagRandomParams RandomParams;
```

This structure contains the parameters necessary for generating the current sequence of random numbers (based on the initial seed). The contents should not be manually adjusted.

```c
typedef struct tagMTRandomParams MTRandomParams;
```

This structure contains the parameters necessary for generating the current sequence of Mersenne twiser random numbers (based on the initial seed). The contents should not be manually adjusted.
13.1.1 Module Random.c

Functions for generating random numbers.

Prototypes

```c
void LALCreateRandomParams ( 
    LALStatus   *status, 
    RandomParams **params, 
    INT4         seed 
)

void LALDestroyRandomParams ( 
    LALStatus   *status, 
    RandomParams **params 
)

void LALUniformDeviate ( 
    LALStatus   *status, 
    REAL4       *deviate, 
    RandomParams *params 
)

void LALNormalDeviates ( 
    LALStatus   *status, 
    REAL4Vector *deviates, 
    RandomParams *params 
)
```

Description

The routines `LALCreateRandomParams()` and `LALDestroyRandomParams()` create and destroy a parameter structure for the generation of random variables. The creation routine requires a random number seed `seed`. If the seed is zero then a seed is generated using the current time.

The routine `LALUniformDeviate()` returns a single random deviate distributed uniformly between zero and unity.

The routine `LALNormalDeviates()` fills a vector with normal (Gaussian) deviates with zero mean and unit variance, whereas the function `XLALNormalDeviate` just returns one normal distributed random number.

Operating Instructions

```c
class LALStatus status;
static RandomParams *params;
static REAL4Vector *vector;
UINT4 i;
INT4 seed = 0;

LALCreateVector( &status, &vector, 9999 );
LALCreateRandomParams( &status, &params, seed );

/* fill vector with uniform deviates */
for ( i = 0; i < vector->length; ++i )
{
    LALUniformDeviate( &status, vector->data + i, params );
}
```
/ * fill vector with normal deviates */
LALNormalDeviates( &status, vector, params );

LALDestroyRandomParams( &status, &params );
LALDestroyVector( &status, &vector );

Algorithm
This is an implementation of the random number generators ran1 and gasdev described in Numerical Recipes [I].

Uses

Notes
13.1.2 Program *RandomTest.c*
Tests the routines in *Random.h*. Outputs random numbers to a file.

**Usage**

`RandomTest [options]`

**Options:**

- `-h` print this message
- `-q` quiet: run silently
- `-v` verbose: print extra information
- `-d level` set lalDebugLevel to level
- `-o` output random numbers to files

**Description**

**Exit codes**

<table>
<thead>
<tr>
<th>Code</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success, normal exit.</td>
</tr>
<tr>
<td>1</td>
<td>Subroutine failed.</td>
</tr>
</tbody>
</table>

**Uses**

**Notes**
13.1.3 Module **MersenneRandom.c**

Routine to get a random number based on the Mersenne Twister Algorithm.

**Prototypes**

```c
void LALCreateMTRandomParams
(
    LALStatus *status,
    REAL8 seed,
    MTRandomParams **params
)
```

```c
void LALDestroyMTRandomParams
(
    LALStatus *status,
    MTRandomParams **params
)
```

```c
void LALMersenneRandom
(
    LALStatus *status,
    REAL8 *output,
    MTRandomParams *params
)
```

```c
void LALMersenneRandomVector
(
    LALStatus *status,
    REAL8Vector *output,
    MTRandomParams *params
)
```

**Description**

This was implemented as in the paper listed below. We have provided two functions, each which may be called multiple times. One returns a single number and the other returns a vector of length prescribed by the Vector->length.

Below I have listed the abstract from the paper referenced below:

A new algorithm called Mersenne Twister (MT) is proposed for generating uniform pseudorandom numbers. For a particular choice of parameters, the algorithm provides a super astronomical period of $2^{19937} - 1$ and 623-dimensional equidistribution up to 32-bit accuracy, while using a working area of only 624 words. This is a new variant of the previously proposed generators, TGFSR, modified so as to admit a Mersenne-prime period. The characteristic polynomial has many terms. The distribution up to $v$ bits accuracy for $1 \leq v \leq 32$ is also shown to be good. An algorithm is also given that checks the primitivity of the characteristic polynomial of MT with computational complexity $O(p^2)$ where $p$ is the degree of the polynomial.

They implemented this generator in portable C-code. It passed several stringent statistical tests, including diehard. Its speed is comparable to other modern generators. Its merits are due to the efficient algorithms that are unique to polynomial calculations over the two-element field.

**Algorithm**

Please see paper listed below:


Copyright (C) 1997 Makoto Matsumoto and Takuji Nishimura. When you use this, send an email to: matumoto@math.keio.ac.jp with an appropriate reference to your work.
Uses
LALCreateMTRandomParams()
LALDestroyMTRandomParams()
LALMersenneRandom()
LALMersenneRandomVector()

Notes
Pulled from distributed source code:

A C-program for MT19937: Real number version (1998/4/6) genrand() generates one pseudorandom real number (double) which is uniformly distributed on [0,1]-interval, for each call. sgenrand(seed) set initial values to the working area of 624 words. Before genrand(), sgenrand(seed) must be called once. (seed is any 32-bit integer except for 0). Integer generator is obtained by modifying two lines.

Coded by Takuji Nishimura, considering the suggestions by Topher Cooper and Marc Rieffel in July-Aug. 1997.

Seed value MAY NOT EQUAL ZERO.
13.1.4 Program MersenneRandomTest.c

A program to test LALMersenneRandom() and LALMersenneRandomVector()

Usage

./MersenneRandomTest [options]
Options:
- `h`  print usage message
- `q`  quiet: run silently
- `v`  verbose: print extra information
- `d level`  set lalDebugLevel to level

This program tests the function LALMersenneRandom(), which generates a random number based on the Mersenne Twister algorithm.

First, it tests that the correct error codes are generated for the following error conditions passed to the function LALMersenneRandom() (tests in italics are not performed if LAL_NEDEBUG is set, as the corresponding checks in the code are made using the ASSERT macro):

- null pointer to output structure
- null pointer to params structure
- params not initialized

Second, it tests that the correct error codes are generated for the following error conditions passed to the function LALMersenneRandomVector() (tests in italics are not performed if LAL_NEDEBUG is set, as the corresponding checks in the code are made using the ASSERT macro):

- null pointer to output structure
- null pointer to params structure
- params not initialized
- outputVector->length = 0

Third, it verifies the output of the generator for each of the following simple test cases:

1. given a certain seed, does the output match the expected?
2. does calling the function again reinitialize it to the new seed properly?
3. does it create a vector of random numbers correctly?

For each successful test (both of these valid data and the invalid ones described above), it prints “PASS” to standard output; if a test fails, it prints “FAIL”.

Exit codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOM</td>
<td>0</td>
<td>&quot;Nominal exit&quot;</td>
</tr>
<tr>
<td>ARG</td>
<td>1</td>
<td>&quot;Error parsing command-line arguments&quot;</td>
</tr>
<tr>
<td>CHK</td>
<td>2</td>
<td>&quot;Error checking failed to catch bad data&quot;</td>
</tr>
<tr>
<td>FLS</td>
<td>3</td>
<td>&quot;Incorrect answer for valid data&quot;</td>
</tr>
<tr>
<td>USE</td>
<td>4</td>
<td>&quot;Bad user-entered data&quot;</td>
</tr>
<tr>
<td>NULL</td>
<td>5</td>
<td>&quot;Null Pointer.&quot;</td>
</tr>
<tr>
<td>ALOC</td>
<td>6</td>
<td>&quot;Memory Allocation Error&quot;</td>
</tr>
<tr>
<td>NMM</td>
<td>7</td>
<td>&quot;Random Number Mismatch&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants MERSENNERANDOMTESTC_E<name>, and the status descriptions in MERSENNERANDOMTESTC_MSGE<name>. The source code with these messages is in MersenneRandomTest.c on line 1.188.
Uses

LALMersenneRandom()
LALMersenneRandomVector()

Notes

• Vector must come in allocated
• params must be initialized before calls can be made.
13.2 Header FindRoot.h

Root finding routines.

Synopsis

#include <lal/FindRoot.h>

This header covers the routines for root finding.

Error conditions

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>1</td>
<td>&quot;Null pointer&quot;</td>
</tr>
<tr>
<td>IDOM</td>
<td>2</td>
<td>&quot;Invalid initial domain&quot;</td>
</tr>
<tr>
<td>MXIT</td>
<td>4</td>
<td>&quot;Maximum iterations exceeded&quot;</td>
</tr>
<tr>
<td>BRKT</td>
<td>8</td>
<td>&quot;Root not bracketed&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants FINDROOT_E<name>, and the status descriptions in FINDROOT_MSGE<name>. The source code with these messages is in FindRoot.h on line 1.79.

Structures

typedef void (REAL4LALFunction) (LALStatus *s, REAL4 *y, REAL4 x, void *p);
typedef void (REAL8LALFunction) (LALStatus *s, REAL8 *y, REAL8 x, void *p);

These are function pointers to functions that map real numbers to real numbers.

typedef struct
tagSFindRootIn
|
{  
  void (*function)(LALStatus *s, REAL4 *y, REAL4 x, void *p);  
  REAL4 xmax;  
  REAL4 xmin;  
  REAL4 xacc;
}  
SFindRootIn;

typedef struct
tagDFindRootIn
|
{  
  void (*function)(LALStatus *s, REAL8 *y, REAL8 x, void *p);  
  REAL8 xmax;  
  REAL8 xmin;  
  REAL8 xacc;
}  
DFindRootIn;

These are the input structures to the root finding routines. The fields are:

function  The function to find the root of.

xmax     The maximum value of the domain interval to look for the root.

xmin     The minimum value of the domain interval to look for the root.

xacc     The accuracy desired for the root.
13.2.1 Module FindRoot.c

Functions for root finding.

Prototypes

```c
void LALSBracketRoot (  
    LALStatus *status,  
    SFindRootIn *inout,  
    void *params  
)
```

```c
int XLALDBracketRoot(  
    REAL8 (*y)(REAL8, void *),  
    REAL8 *xmin,  
    REAL8 *xmax,  
    void *params  
)
```

```c
void LALDBracketRoot (  
    LALStatus *status,  
    DFindRootIn *inout,  
    void *params  
)
```

```c
void LALSbisectionFindRoot (  
    LALStatus *status,  
    REAL4 *root,  
    SFindRootIn *input,  
    void *params  
)
```

```c
REAL8 XLALDBisectionFindRoot (  
    REAL8 (*y)(REAL8, void *),  
    REAL8 xmin,  
    REAL8 xmax,  
    REAL8 xacc,  
    void *params  
)
```

```c
void LALDBisectionFindRoot (  
    LALStatus *status,  
    REAL8 *root,  
    DFindRootIn *input,  
    void *params  
)
```

Description

The routine `LALSBracketRoot()` expands the specified domain until a root is contained. The routine `LALDBracketRoot()` is the same but for a double-precision function.

The routine `LALSFindRoot()` bisects the domain (which must contain one root) until the root is found with the desired accuracy. The routine `LALDFindRoot()` is the same but for a double-precision function.
Operating Instructions

Suppose we want to find the root of the function \( y = F(x; y_0) = y_0 + x^2 \). Define the function:

```c
static void F( LALStatus *status, REAL4 *y, REAL4 x, void *y0 )
{
    INITSTATUS( status, "F", "Function F()" );
    ASSERT( y0, status, 1, "Null pointer" );
    y = *(REAL4 *)y0 + x*x;
    RETURN( status );
}
```

Then use the following code to bracket and find the root \( x_0 = 1 \) where \( F(x_0; y_0 = -1) = 0 \):

```c
static LALStatus status;
SFindRootIn input;
REAL4 y0;
REAL4 x0;

y0    = -1;
input.function = F;
input.xmin    = 0.1;
input.xmax    = 0.2;
input.xacc    = 1e-5;

/* expand domain until a root is bracketed */
LALSBracketRoot( &status, &input, &y0 );

/* bisect domain until root is found */
LALSBisectionFindRoot( &status, &x0, &input, &y0 );
```

Algorithm

This is an implementation of the root bracketing and bisection finding routines `zbrac` and `rtbis` in Numerical Recipes [1].

Uses

Notes
13.2.2 Program FindRootTest.c

Tests the routines in FindRoot.h.

Usage

FindRootTest [options]

Options:

- h       print this message
- q       quiet: run silently
- v       verbose: print extra information
- d level set lalDebugLevel to level

Description

Exit codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success, normal exit.</td>
</tr>
<tr>
<td>1</td>
<td>Subroutine failed.</td>
</tr>
</tbody>
</table>

Uses

Notes
13.3 Header *Integrate.h*

Integrates a function.

**Synopsis**

```
#include <lal/Integrate.h>
```

This header covers the routines for integrating a function.

**Error conditions**

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>1</td>
<td>&quot;Null pointer&quot;</td>
</tr>
<tr>
<td>TYPE</td>
<td>2</td>
<td>&quot;Unknown integral type&quot;</td>
</tr>
<tr>
<td>IDOM</td>
<td>4</td>
<td>&quot;Invalid domain&quot;</td>
</tr>
<tr>
<td>MXIT</td>
<td>8</td>
<td>&quot;Maximum iterations exceeded&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants `INTEGRATEH_E<name>`, and the status descriptions in `INTEGRATEH_MSGE<name>`. The source code with these messages is in `Integrate.h` on line 1.79.

**Structures**

```c
typedef enum
{
    ClosedInterval,
    OpenInterval,
    SingularLowerLimit,
    SingularUpperLimit,
    InfiniteDomainPow,
    InfiniteDomainExp
} IntegralType;

typedef struct
tagSIntegrateIn
{
    void (*function)(LALStatus *s, REAL4 *y, REAL4 x, void *p);
    REAL4 xmax;
    REAL4 xmin;
    IntegralType type;
} SIntegrateIn;

typedef struct
tagDIntegrateIn
{
    void (*function)(LALStatus *s, REAL8 *y, REAL8 x, void *p);
    REAL8 xmax;
    REAL8 xmin;
    IntegralType type;
} DIntegrateIn;
```

These are input structures to the integration routines. The fields are:

- **function** The function to integrate.
- **xmax** The maximum value of the domain of integration.
**xmax** The minimum value of the domain of integration.

**type** The type of integration. This is an enumerated type which can take on one of the following values:

<table>
<thead>
<tr>
<th>Enumeration constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClosedInterval</td>
<td>Evaluate integral on a closed interval</td>
</tr>
<tr>
<td>OpenInterval</td>
<td>Evaluate integral on an open interval</td>
</tr>
<tr>
<td>SingularLowerLimit</td>
<td>Integrate an inverse square-root singularity at lower limit</td>
</tr>
<tr>
<td>SingularUpperLimit</td>
<td>Integrate an inverse square-root singularity at upper limit</td>
</tr>
<tr>
<td>InfiniteDomainPow</td>
<td>Integrate an infinite domain with power-law falloff</td>
</tr>
<tr>
<td>InfiniteDomainExp</td>
<td>Integrate an infinite domain with exponential falloff</td>
</tr>
</tbody>
</table>

The types of integration are the following: I. **ClosedInterval** indicates that the integral should be computed on equal-spaced domain intervals including the boundary. II. **OpenInterval** indicates that the integral should be computed on intervals of the domain not including the boundary. III. **SingularLowerLimit** indicates that the integral should be evaluated on an open interval with a transformation so that an inverse-square-root singularity at the lower limit can be integrated. IV. **SingularUpperLimit** is the same as above but for a singularity at the upper limit. V. **InfiniteDomainPow** indicates that the integral should be evaluated over an semi-infinite domain—appropriate when both limits have the same sign (though one is very large) and when the integrand vanishes faster than $x^{-1}$ at infinity. VI. **InfiniteDomainExp** indicates that the integral should be evaluated over an infinite domain starting at $x_{\text{min}}$ and going to infinity ($x_{\text{max}}$ is ignored)—the integrand should vanish exponentially for large $x$. 
13.3.1 Module `Integrate.c`

Functions for generating random numbers.

Prototypes

```c
void LALSRombergIntegrate (  
    LALStatus *status,  
    REAL4 *result,  
    SIntegrateIn *input,  
    void *params  
)

void LALDRombergIntegrate (  
    LALStatus *status,  
    REAL8 *result,  
    DIntegrateIn *input,  
    void *params  
)
```

Description

The routine `LALSRombergIntegrate` performs the integral specified by the structure `input` and the result is returned as `result`. Any additional parameters (other than the integration variable `x`) can be passed as `params`. The routine `LALSRombergIntegrate` does not use `params` but just passes it to the integrand. The routine `LALDRombergIntegrate` is the same but for double precision.

Operating Instructions

The following program performs the integral \( \int_0^2 F(x) dx \) where \( F(x) = x^4 \log(x + \sqrt{x^2 + 1}) \).

```c
#include <math.h>
#include <lal/LALStdlib.h>
#include <lal/Integrate.h>

static void F( LALStatus *s, REAL4 *y, REAL4 x, void *p )  
{
    REAL4 x2 = x*x;  
    REAL4 x4 = x2*x2;  
    INITSTATUS( s, "F", "Function F()" );  
    ASSERT( !p, s, 1, "Non-null pointer" );  
    y = x4 * log( x + sqrt( x2 + 1 ) );  
    RETURN( s );
}

int main ()
{
    const REAL4 epsilon = 1e-6;  
    const long double expect = 8.153364119811650205L;  
    static LALStatus status;  
    SIntegrateIn intinp;  
    REAL4 result;

    intinp.function = F;  
    intinp.xmin = 0;  
    intinp.xmax = 2;  
    intinp.type = ClosedInterval;
```
LALSRombergIntegrate( &status, &result, &intinp, NULL );
if (fabs(result - expect) > epsilon * fabs(expect))
{
    /* integration did not achieve desired accuracy --- exit failure */
    return 1;
}
return 0;

Algorithm
This is an implementation of the Romberg integrating function qromb in Numerical Recipes [1].

Uses
These routines use the functions LALSPolynomialInterpolation() and LALDPolynomialInterpolation().

Notes
13.3.2 Program IntegrateTest.c

Tests the routines in Integrate.h by performing a suite of numerical integrations and checking the accuracy of the results.

Usage

IntegrateTest [options]

Options:

- `-h` print this message
- `-q` quiet: run silently
- `-v` verbose: print extra information
- `-d level` set lalDebugLevel to level

Description

Exit codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success, normal exit.</td>
</tr>
<tr>
<td>1</td>
<td>Subroutine failed.</td>
</tr>
</tbody>
</table>

Uses

Notes
Generates random numbers.

Synopsis

```c
#include <lal/Interpolate.h>
```

This header covers the routines for interpolation.

Error conditions

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>1</td>
<td>&quot;Null pointer&quot;</td>
</tr>
<tr>
<td>SIZE</td>
<td>2</td>
<td>&quot;Invalid size&quot;</td>
</tr>
<tr>
<td>ZERO</td>
<td>4</td>
<td>&quot;Zero divide&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants `INTERPOLATEH_E<name>`, and the status descriptions in `INTERPOLATEH_MSGE<name>`. The source code with these messages is in `Interpolate.h` on line 1.76.

Structures

```c
typedef struct
tagSInterpolateOut
{
   REAL4  y;
   REAL4  dy;
}
SInterpolateOut;

typedef struct
tagDInterpolateOut
{
   REAL8  y;
   REAL8  dy;
}
DInterpolateOut;
```

These structures contain the output of the interpolation. The two fields are:

- **y** The interpolated value.
- **dy** The estimated error in the interpolated value.

```c
typedef struct
tagSInterpolatePar
{
   UINT4  n;
   REAL4  *x;
   REAL4  *y;
}
SInterpolatePar;

typedef struct
tagDInterpolatePar
{
   UINT4  n;
   REAL8  *x;
   REAL8  *y;
}
DInterpolatePar;
```
These structures contain the interpolation parameters. These are the arrays of \( n \) domain values \( x[0] \ldots x[n-1] \) and their corresponding values \( y[0] \ldots y[n-1] \). The fields are:

- \( n \) The number of points in the arrays to use in the interpolation.
- \( x \) The array of domain values.
- \( y \) The array of values to interpolate.
13.4.1 Module Interpolate.c

Functions for generating random numbers.

Prototypes

```c
void LALSPolynomialInterpolation (  
    LALStatus *status,  
    SInterpolateOut *output,  
    REAL4 target,  
    SInterpolatePar *params
)

void LALDPolynomialInterpolation (  
    LALStatus *status,  
    DInterpolateOut *output,  
    REAL8 target,  
    DInterpolatePar *params
)
```

Description

The routine `LALSPolynomialInterpolation()` computes the interpolated \( y \) value `output` at the \( x \) value `target` by fitting a polynomial of order `params.n-1` to the data. The result `output` is of type `SInterpolateOut`, which contains the value `output.y` as well as an estimate of the error `output.dy`. The routine `LALDPolynomialInterpolation()` is the same but for double precision.

Operating Instructions

The following program fits a fourth-order polynomial to the five data points \( \{(0, 0), (1, 1), (2, 3), (3, 4), (4, 3)\} \), and interpolates the value at \( x = 2.4 \).

```c
#include <lal/LALStdlib.h>
#include <lal/Interpolate.h>

int main ()
{
    enum { ArraySize = 5 };  
    static LALStatus status;  
    REAL4 x[ArraySize] = {0,1,2,3,4};  
    REAL4 y[ArraySize] = {0,1,3,4,3};  
    REAL4 target = 2.4;  
    SInterpolatePar intpar = {ArraySize, x, y};  
    SInterpolateOut intout;

    LALSPolynomialInterpolation( &status, &intout, target, &intpar );

    return 0;
}
```

Algorithm

This is an implementation of the Neville algorithm, see `polint` in Numerical Recipes [1].

Uses

Notes

`$Id: Interpolate.c,v 1.10 2007/06/08 14:41:59 bema Exp $`
13.4.2 Program InterpolateTest.c

Tests the routines in Interpolate.h.

Usage

InterpolateTest [options]

Options:
- `h` print this message
- `q` quiet: run silently
- `v` verbose: print extra information
- `d level` set lalDebugLevel to level

Description

Exit codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success, normal exit.</td>
</tr>
<tr>
<td>1</td>
<td>Subroutine failed.</td>
</tr>
</tbody>
</table>

Uses

Notes
13.5 Header Sort.h

Provides routines for sorting, indexing, and ranking real vector elements.

13.5.1 Synopsis

#include <lal/Sort.h>

Error conditions

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUL</td>
<td>1</td>
<td>&quot;Null pointer&quot;</td>
</tr>
<tr>
<td>LEN</td>
<td>2</td>
<td>&quot;Length mismatch&quot;</td>
</tr>
<tr>
<td>MEM</td>
<td>3</td>
<td>&quot;Memory allocation error&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants SORTH_E<name>, and the status descriptions in SORTH_MSGE<name>. The source code with these messages is in Sort.h on line 1.79.

Structures
13.5.2 Module HeapSort.c

Sorts, indexes, or ranks vector elements using the heap sort algorithm.

Prototypes

```c
int XLALHeapSort( void *base, UINT4 nobj, UINT4 size, void *params,
int (*compar)(void *, const void *, const void *))
int XLALHeapIndex( INT4 *indx, void *base, UINT4 nobj, UINT4 size, void *params,
int (*compar)(void *, const void *, const void *))
int XLALHeapRank( INT4 *rank, void *base, UINT4 nobj, UINT4 size, void *params,
int (*compar)(void *, const void *, const void *))
void LALSHeapSort(LALStatus *stat,
REAL4Vector *vector)
void LALSHeapIndex(LALStatus *stat,
INT4Vector *idx,
REAL4Vector *vector)
void LALSHeapRank(LALStatus *stat,
INT4Vector *rank,
REAL4Vector *vector)
void LALDHeapSort(LALStatus *stat,
REAL8Vector *vector)
void LALDHeapIndex(LALStatus *stat,
INT4Vector *idx,
REAL8Vector *vector)
void LALDHeapRank(LALStatus *stat,
INT4Vector *rank,
REAL8Vector *vector)
```

Description

These routines sort a vector *data* (of type REAL4Vector or REAL8Vector) into ascending order using the in-place heapsort algorithm, or construct an index vector *index* that indexes *data* in increasing order (leaving *data* unchanged), or construct a rank vector *rank* that gives the rank order of the corresponding *data* element.

The relationship between sorting, indexing, and ranking can be a bit confusing. One way of looking at it is that the original array is ordered by index, while the sorted array is ordered by rank. The index array gives the index as a function of rank; i.e. if you’re looking for a given rank (say the 0th, or smallest element), the index array tells you where to look it up in the unsorted array:

```
unsorted_array[index[i]] = sorted_array[i]
```

The rank array gives the rank as a function of index; i.e. it tells you where a given element in the unsorted array will appear in the sorted array:

```
unsorted_array[j] = sorted_array[rank[j]]
```

Clearly these imply the following relationships, which can be used to construct the index array from the rank array or vice-versa:

```
index[rank[j]] = j
rank[index[i]] = i
```

The XLAL versions of these routines, XLALHeapSort, XLALHeapIndex, and XLALHeapRank, perform the same operations but on arrays of nobj generic objects of size size pointed to by base and using the comparison function compar. The function compar has the prototype

```c
int compar( void *p, const void *x, const void *y )
```

and returns -1 if x < y, 0 if x = y, and +1 if x > y. Here p (which may be NULL) is a pointer to additional data that may be used in the comparison function. This pointer is passed to the comparison function unchanged from the argument params of XLALHeapSort, XLALHeapIndex, and XLALHeapRank.
Algorithm

These routines use the standard heap sort algorithm described in Sec. 8.3 of Ref. [1].

The `LALSHeapSort()` and `LALDHeapSort()` routines are entirely in-place, with no auxiliary storage vector. The `LALSHeapIndex()` and `LALDHeapIndex()` routines are also technically in-place, but they require two input vectors (the data vector and the index vector), and leave the data vector unchanged. The `LALSHeapRank()` and `LALDHeapRank()` routines require two input vectors (the data and rank vectors), and also allocate a temporary index vector internally; these routines are therefore the most memory-intensive. All of these algorithms are $N \log_2(N)$ algorithms, regardless of the ordering of the initial dataset.

Note: if you can use `qsort`, you should.

Uses

LALI4CreateVector()
LALI4DestroyVector()

Notes
13.5.3 Program `SortTest.c`

A program to test sorting routines.

**Usage**

```
SortTest [-s seed] [-d [debug-level]] [-v]
```

**Description**

This test program creates rank and index arrays for an unordered list of numbers, and then sorts the list. The data for the list are generated randomly, and the output is to `stdout` if `-v` is specified (unless redirected). `SortTest` returns 0 if it executes successfully, and 1 if any of the subroutines fail.

The `-s` option sets the seed for the random number generator; if `seed` is set to zero (or if no `-s` option is given) then the seed is taken from the processor clock. The `-d` option increases the default debug level from 0 to 1, or sets it to the specified value `debug-level`.

**Exit codes**

<table>
<thead>
<tr>
<th>Code</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success, normal exit.</td>
</tr>
<tr>
<td>1</td>
<td>Subroutine failed.</td>
</tr>
</tbody>
</table>

**Algorithm**

Uses

- `lalDebugLevel`
- `CreateI4Vector()`
- `CreateSVector()`
- `DestroyI4Vector()`
- `DestroySVector()`
- `LALCreateRandomParams()`
- `LALDestroyRandomParams()`
- `LALUniformDeviate()`
- `LALPrintError()`

**Notes**
13.6 **Header ODE.h**

Routines for solving ordinary differential equations (ODEs).

**Synopsis**

```c
#include <lal/ODE.h>
```

These routines solve ordinary differential equations (ODEs) of the form:

\[ \dot{x} = f(x, t, \ldots) \]

where \( f \) is a specified vector-valued function.

**Error conditions**

<table>
<thead>
<tr>
<th>name</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>001</td>
<td>&quot;Null pointer.&quot;</td>
</tr>
<tr>
<td>SAME</td>
<td>002</td>
<td>&quot;Same data pointer.&quot;</td>
</tr>
<tr>
<td>SIZE</td>
<td>004</td>
<td>&quot;Invalid size.&quot;</td>
</tr>
<tr>
<td>SZMM</td>
<td>010</td>
<td>&quot;Size mismatch.&quot;</td>
</tr>
<tr>
<td>NSTP</td>
<td>020</td>
<td>&quot;Step number mismatch.&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants `ODEH_<name>`, and the status descriptions in `ODEH_MSGE_<name>`. The source code with these messages is in `ODE.h` on line 1.72.

**Structures**

```c
typedef struct
    tagREAL4ODEIndep
{
    REAL4 t;
    void *aux;
}
REAL4ODEIndep;
```

The independent variables of the ODE (parameters to the ODE function). The fields are:

- **t**: The independent parameter (e.g., time) that is evolved.
- **aux**: Storage for auxiliary variables used internally in the ODE routine.

```c
typedef struct
    tagREAL4ODEParams
{
    void (*ode)( LALStatus *, REAL4Vector *, REAL4Vector *, REAL4ODEIndep *);
    REAL4ODEIndep *indep;
    REAL4 tstep;
    REAL4Vector *xdot;
    REAL4Vector *xerr;
    REAL4 eps;
    REAL4VectorSequence *dx;
}
REAL4ODEParams;
```

The parameters for the ODE step integrator. The fields are:

- **ode**: Pointer to the function that computes the RHS of the ODE.
- **indep**: The independent variables of used by this function.
tstep  The suggested time step to use.

xdot  The value of the LHS of the ODE at the current time.

xerr  The estimated errors from the last ODE step.

eps  The allowed fractional error per step. If eps is less than LAL_REAL4_EPS, then the latter is used instead.

dx  Workspace storage for use in the step integrator. The length of the sequence depends on which integrator is used.
13.6.1 Module ODE.c

Routines to integrate ODEs.

Prototypes

```c
void LALSRungeKutta4(
    LALStatus *status,
    REAL4Vector *output,
    REAL4Vector *input,
    REAL4ODEParams *params
)
```

```c
void LALSRungeKutta5(
    LALStatus *status,
    REAL4Vector *output,
    REAL4Vector *input,
    REAL4ODEParams *params
)
```

```c
void LALSRungeKutta5Adapt(
    LALStatus *status,
    REAL4Vector *output,
    REAL4Vector *input,
    REAL4ODEParams *params
)
```

Description

The routines `LALSRungeKutta4`, `LALSRungeKutta5`, and `LALSRungeKutta5Adapt` are used to advance an ODE solution from one time step to the next. `LALSRungeKutta4` and `LALSRungeKutta5` advance by the specified time step. The former uses a 4th order Runge Kutta method while the latter, which uses a 5th order Runge Kutta method, also makes an estimate of the error accumulated in the step. `LALSRungeKutta5Adapt` uses `LALSRungeKutta5` to take a step and, if the error in the step is too larger (compared to the fractional error specified by `eps`), then it re-does that step with a finer step size; the step size is modified by the routine.

All the routines advance the time after each step.

The sequence length of the `dt` field of the parameter structure must be 4 for `LALSRungeKutta4` and 6 for `LALSRungeKutta5` and `LALSRungeKutta5Adapt`.

Operating Instructions

The following routine specifies the ODE for the Kepler problem:

\[
\frac{d}{dt}\left\{x, y, v_x, v_y\right\} = \left\{v_x, v_y, -x/r^3, -y/r^3\right\}
\]

```c
#include <math.h>
#include <lal/LALStdlib.h>

void Kepler( LALStatus *s, REAL4Vector *xdot, REAL4Vector *x, REAL4ODEIndep *p )
{
    REAL4 rsq = x->data[0] * x->data[0] + x->data[1] * x->data[1];
    REAL4 rcb = rsq * sqrt( rsq );
    xdot->data[0] = x->data[2];
    xdot->data[1] = x->data[3];
}
```
The following programs integrate the Kepler problem. The first program integrates a circular orbit with fixed step sizes:

```c
#include <math.h>
#include <stdio.h>
#include <string.h>
#include <lal/LALStdlib.h>
#include <lal/AVFactories.h>
#include <lal/SeqFactories.h>
#include <lal/ODE.h>

int main( void )
{
    const UINT4 ndim = 4;
    const UINT4 nstp = 4;
    const REAL4 tend = 100;
    const REAL4 tstep = 0.01;
    static REAL4ODEParams params;
    static REAL4ODEIndep indep;
    static REAL4Vector *x0;
    static REAL4Vector *x;
    static LALStatus status;
    CreateVectorSequenceIn seqin = { nstp, ndim };

    LALCreateVector( &status, &x, ndim );
    LALCreateVector( &status, &x0, ndim );
    LALCreateVector( &status, &params.xdot, ndim );
    LALCreateVectorSequence( &status, &params.dx, &seqin );

    params.ode = Kepler;
    params.indep = &indep;
    params.tstep = tstep;

    x->data[0] = 1;
    x->data[1] = 0;
    x->data[2] = 0;
    x->data[3] = 1;

    while ( indep.t < tend )
    {
        memcpy( x0->data, x->data, x->length * sizeof( *x->data ) );
        ( *params.ode )( &status, params.xdot, x0, &indep );
        LALSRungeKutta4( &status, x, x0, &params );
        printf( "%e\t%e\t", x->data[0], x->data[1] );
        printf( "%e\t%e\n", cos( indep.t ), sin( indep.t ) );
    }

    LALDestroyVectorSequence( &status, &params.dx );
    LALDestroyVector( &status, &params.xdot );
    LALDestroyVector( &status, &x0 );
    LALDestroyVector( &status, &x );

    return 0;
}
```

This second program integrates a highly-eccentric bound orbit with adaptive step sizes (and also computes the orbit using Kepler's method for comparison):

```c
#include <math.h>
```
#include <stdio.h>
#include <string.h>
#include <lal/LALStdlib.h>
#include <lal/LALConstants.h>
#include <lal/AVFactories.h>
#include <lal/SeqFactories.h>
#include <lal/ODE.h>

int main( void )
{
    const UINT4 ndim = 4;
    const UINT4 nstp = 6;
    const REAL4 nper = 3;
    static REAL4ODEParams params;
    static REAL4ODEIndep indep;
    static REAL4Vector *x0;
    static REAL4Vector *x;
    static LALStatus status;
    CreateVectorSequenceIn seqin = { nstp, ndim };
    REAL4 a;
    REAL4 v;
    REAL4 r;
    REAL4 e;
    REAL4 P;

    LALCreateVector( &status, &x, ndim );
    LALCreateVector( &status, &x0, ndim );
    LALCreateVector( &status, &params.xdot, ndim );
    LALCreateVector( &status, &params.xerr, ndim );
    LALCreateVectorSequence( &status, &params.dx, &seqin );

    params.eps = 1e-6;
    params.ode = Kepler;
    params.indep = &indep;
    params.tstep = 0.1;

    x->data[0] = r = 1;
    x->data[1] = 0;
    x->data[2] = 0;
    x->data[3] = v = 1.2;

    a = 1 / ( 2 / r - v * v );
    P = 2 * LAL_PI * a * sqrt( a );
    e = 1 - r / a;

    while ( indep.t < nper * P )
    {
        REAL4 fac = 1;
        REAL4 rad;
        REAL4 phi;
        REAL4 psi;
        REAL4 del;
        REAL4 M;

        memcpy( x0->data, x->data, x->length * sizeof( *x->data ) );
        ( *params.ode )( &status, params.xdot, x0, &indep );
        LALSRungeKutta5Adapt( &status, x, x0, &params );
        printf( "%e\t%e\t%e\t", indep.t, x->data[0], x->data[1] );
    }
psi = M = 2 * LAL_PI * indep.t / P;
del = psi - e * sin( psi ) - M;
while ( fabs( del ) > LAL_REAL4_EPS )
{
    psi += ( del < 0 ? 1 : -1 ) * ( fac *= 0.5 ) * e;
    del = psi - e * sin( psi ) - M;
}
rad = a * ( 1 - e * cos( psi ) );
phi = 2 * atan( sqrt(( 1 + e ) / ( 1 - e ) ) * tan( psi / 2 ) );
printf( "%.16e %.16e\n", rad * cos( phi ), rad * sin( phi ) );
}

LALDestroyVectorSequence( &status, &params.dx );
LALDestroyVector( &status, &params.xerr );
LALDestroyVector( &status, &params.xdot );
LALDestroyVector( &status, &x0 );
LALDestroyVector( &status, &x );

    return 0;
}

Algorithm

These routines are based on the methods presented in Numerical Recipes [1].

* Author: J. D. E. Creighton
* $Id: ODE.c,v 1.3 2007/06/08 14:41:59 bema Exp $
13.6.2 Program ODETest.c

Tests the routines in ODE.h by integrating Keplerian orbits. The orbits so integrated are output to files containing the integrated and expected orbits.

Usage

ODETest
13.7 Header Dirichlet.h

Provides prototype and error code information for \texttt{LALDirichlet()}, a routine which calculates the values of the Dirichlet kernel $D_N(x)$.

Synopsis

\begin{verbatim}
#include "Dirichlet.h"
\end{verbatim}

Error conditions

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULLPIN</td>
<td>1</td>
<td>&quot;Null pointer to input parameters&quot;</td>
</tr>
<tr>
<td>NVALUE</td>
<td>2</td>
<td>&quot;Dirichlet parameter $N$ less than or equal to zero&quot;</td>
</tr>
<tr>
<td>SIZE</td>
<td>3</td>
<td>&quot;Length parameter less than or equal to zero&quot;</td>
</tr>
<tr>
<td>DELTAX</td>
<td>4</td>
<td>&quot;Spacing of $x$ values less than or equal to zero&quot;</td>
</tr>
<tr>
<td>NULLPOUT</td>
<td>5</td>
<td>&quot;Null pointer to output vector&quot;</td>
</tr>
<tr>
<td>SIZEMM</td>
<td>6</td>
<td>&quot;Length of data member of output vector does not equal length specified in input parameters&quot;</td>
</tr>
<tr>
<td>NULLPDOUT</td>
<td>7</td>
<td>&quot;Null pointer to data member of output vector&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants \texttt{DIRICHLETH_E<name>}, and the status descriptions in \texttt{DIRICHLETH_MSGE<name>}. The source code with these messages is in \texttt{Dirichlet.h} on line 1.98.

Structures

\begin{verbatim}
struct DirichletParameters

Contains parameters that specify the Dirichlet kernel $D_N(x)$. The fields are:

\begin{verbatim}
UINT4 n  Dirichlet parameter $N$.
UINT4 length Specified length of output vector.
REAL8 deltaX Spacing of $x$ values.
\end{verbatim}
\end{verbatim}
13.7.1 Module **Dirichlet.c**

Calculates the values of the Dirichlet kernel.

**Prototypes**

```c
void LALDirichlet(LALStatus* status,
                  REAL4Vector* output,
                  const DirichletParameters* parameters )
```

**Description**

**LALDirichlet()** calculates the values of the Dirichlet kernel [2]:

\[
D_N(x) := \begin{cases} 
(-1)^x(N-1) & x = 0, \pm 1, \pm 2, \cdots \\
\sin(N\pi x) & \text{otherwise}
\end{cases}
\]

The magnitude of the Dirichlet kernel is \(1/N\) times the magnitude of the discrete Fourier transform of the discrete \(N\)-point rectangular window.

**Algorithm**

None.

**Uses**

None.

**Notes**

The Dirichlet kernel is needed for a rigorous (i.e., exact) calculation of the standard cross-correlation statistic, evaluated in discrete time. However, **LALStochasticOptimalFilter()** and other routines in the **stochastic** package, as currently implemented, do not make use of the Dirichlet kernel; these routines are expected to be used in the large observation time continuum limit approximation, for which \(D_N(x)\) can effectively be replaced by a Dirac delta function.
13.7.2 Program DirichletTest.c

Test suite for LALDirichlet().

Usage

./DirichletTest

Description

This program tests the function LALDirichlet(). It tests all error conditions listed in the Error codes table. It also writes to files the values of the Dirichlet kernel for three different valid test cases. See Figs. 13.1-13.3.

![Dirichlet Kernel $D_N(x)$ (N=10, $\Delta x=.01$)](image1)

Figure 13.1: Dirichlet kernel for $N = 10$, $\Delta x = .01$, and $0 \leq x \leq 1$.

![Dirichlet Kernel $D_N(x)$ (N=11, $\Delta x=.01$)](image2)

Figure 13.2: Dirichlet kernel for $N = 11$, $\Delta x = .01$, and $0 \leq x \leq 1$. 
Figure 13.3: Dirichlet kernel for $N = 10$, $\Delta x = .01$, and $0 \leq x \leq 2$.

Exit codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success, normal exit.</td>
</tr>
<tr>
<td>1</td>
<td>Subroutine failed.</td>
</tr>
</tbody>
</table>

Uses

- lalDebugLevel
- LALDirichlet()
- LALPrintVector()
- LALSCreateVector()
- LALSDestroyVector()
- LALCheckMemoryLeaks()

Notes
None.
13.8 Header `CoarseGrainFrequencySeries.h`

Provides prototype, structure and error code information for routines which coarse-grain a frequency series.

Synopsis

```c
#include <lal/CoarseGrainFrequencySeries.h>
```

Error conditions

<table>
<thead>
<tr>
<th>name</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULLPTR</td>
<td>1</td>
<td>&quot;Null pointer&quot;</td>
</tr>
<tr>
<td>SAMEPTR</td>
<td>2</td>
<td>&quot;Input and Output pointers the same&quot;</td>
</tr>
<tr>
<td>ZEROLEN</td>
<td>3</td>
<td>&quot;Zero length for data member of series&quot;</td>
</tr>
<tr>
<td>NONPOSDELTAF</td>
<td>4</td>
<td>&quot;Negative or zero frequency spacing&quot;</td>
</tr>
<tr>
<td>NEGFRMIN</td>
<td>5</td>
<td>&quot;Negative start frequency&quot;</td>
</tr>
<tr>
<td>MMHETERO</td>
<td>7</td>
<td>&quot;Mismatch in heterodyning frequencies&quot;</td>
</tr>
<tr>
<td>MMFMIN</td>
<td>8</td>
<td>&quot;Mismatch in start frequencies&quot;</td>
</tr>
<tr>
<td>MMDELTAF</td>
<td>9</td>
<td>&quot;Mismatch in frequency spacings&quot;</td>
</tr>
<tr>
<td>MMLEN</td>
<td>10</td>
<td>&quot;Mismatch in sequence lengths&quot;</td>
</tr>
<tr>
<td>OORCOARSE</td>
<td>16</td>
<td>&quot;Coarse-graining parameters out of range&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants `COARSEGRAINFREQUENCYSERIESH_E<name>`, and the status descriptions in `COARSEGRAINFREQUENCYSERIESH_MSGE<name>`. The source code with these messages is in `CoarseGrainFrequencySeries.h` on line 1.82.

Structures

```c
struct FrequencySamplingParams
```

Contains the parameters needed to specify the sampling of a frequency series:

- `UINT4 length` The number of points in the frequency series.
- `REAL8 f0` The start frequency of the frequency series.
- `REAL8 deltaF` The frequency spacing of the frequency series.
13.8.1 Module CoarseGrainFrequencySeries.c

“Coarse grains” a frequency series to produce a series with a lower frequency resolution.

Prototypes

```c
void LALSCoarseGrainFrequencySeries(LALStatus *status, REAL4FrequencySeries *output, const REAL4FrequencySeries *input, const FrequencySamplingParams *params);

void LALDCoarseGrainFrequencySeries(LALStatus *status, REAL8FrequencySeries *output, const REAL8FrequencySeries *input, const FrequencySamplingParams *params);

void LALCCoarseGrainFrequencySeries(LALStatus *status, COMPLEX8FrequencySeries *output, const COMPLEX8FrequencySeries *input, const FrequencySamplingParams *params);

void LALZCoarseGrainFrequencySeries(LALStatus *status, COMPLEX16FrequencySeries *output, const COMPLEX16FrequencySeries *input, const FrequencySamplingParams *params);
```

Description

These functions are designed to facilitate approximation of integrals such as

\[ \int g(f) h(f) \, df \]

when \( g(f) \) and \( h(f) \) are sampled with different frequency resolutions. If the frequency resolution were the same for both functions, e.g., a frequency spacing of \( \delta f \) and a start frequency of \( f_0 \), so that the \( k \)th element corresponded to a frequency \( f_k = f_0 + k\delta f \), the approximation would be defined as

\[ \int g(f) h(f) \, df \approx \delta f \sum_k g_k h_k \]

whose contribution from the \( k \)th element is\(^1\)

\[ \int_{f_k - \delta f/2}^{f_k + \delta f/2} g(f) h(f) \, df \approx \delta f g_k h_k . \]

The central idea in our definitions of coarse graining will thus be the correspondence

\[ h_k \approx \frac{1}{\delta f} \int_{f_k - \delta f/2}^{f_k + \delta f/2} h(f) \, df \]  \hspace{1cm} (13.1)\]

The purpose of this function is to obtain a frequency series \( \{h_k\} \) with start frequency \( f_0 \) and frequency spacing \( \delta f \) from a finer-grained frequency series \( \{h'_\ell\} \) with start frequency \( f'_0 \) and frequency spacing \( \delta f' \). Focussing on the \( k \)th element of the coarse-grained series, which represents a frequency range from \( f_k - \delta f/2 \) to \( f_k + \delta f/2 \), we consider the elements of the fine-grained series whose frequency ranges overlap with this. (Fig. 13.4) We define \( t^\text{min}_k \) and \( t^\text{max}_k \) to be the indices of the first and last elements of \( h'_\ell \) which overlap

\(^1\)It is important to make the limits of integration symmetric about \( f_k \) to maintain reality conditions when dealing with Fourier transforms of real quantities.
Figure 13.4: Coarse graining a frequency series

 completa  with the frequency range corresponding to $h_k$. These are most easily defined in terms of non-integer indices $\lambda_{k\text{ min}}$ and $\lambda_{k\text{ max}}$ which correspond to the locations of fine-grained elements which would exactly reach the edges of the coarse-grained element with index $k$. These are defined by

$$f_0 + \left( k - \frac{1}{2} \right) \delta f = f'_0 + \left( \lambda_{k\text{ min}} - \frac{1}{2} \right) \delta f'$$
$$f_0 + \left( k + \frac{1}{2} \right) \delta f = f'_0 + \left( \lambda_{k\text{ max}} + \frac{1}{2} \right) \delta f'$$

or, defining the offset $\Omega = (f_0 - f'_0)/\delta f'$ and the coarse graining ratio $\rho = \delta f/\delta f'$,

$$\lambda_{k\text{ min}} = \Omega + \left( k - \frac{1}{2} \right) \rho + \frac{1}{2}$$
$$\lambda_{k\text{ max}} = \Omega + \left( k + \frac{1}{2} \right) \rho - \frac{1}{2}$$

Examination of Fig. 13.4 shows that $\ell_{k\text{ min}}$ is the smallest integer not less than $\lambda_{k\text{ min}}$ and $\ell_{k\text{ max}}$ is the largest integer not greater than $\lambda_{k\text{ min}}$.

With these definitions, approximating the integral in (13.1) gives

$$h_k = \frac{1}{\rho} \left( (\ell_{k\text{ min}} - \lambda_{k\text{ min}})h'_{\ell_{k\text{ min}} - 1} + \sum_{\ell=\ell_{k\text{ min}}}^{\ell_{k\text{ max}}} h'_{\ell} + (\lambda_{k\text{ max}} - \ell_{k\text{ max}})h'_{\ell_{k\text{ max}} + 1} \right) \quad (13.2)$$

In the special case $f_0 = f'_0$, we assume both frequency series represent the independent parts of larger frequency series $\{h_k|k=-(N-1)\ldots(N-1)\}$ and $\{h'_\ell|\ell=-(N-1)\ldots(N-1)\}$ which obey $h_{-k} = h_k^*$ and $h'_{-\ell} = h'_\ell^*$ (e.g., fourier transforms of real data). In that case, the DC element of the coarse-grained series can be built out of both positive- and implied negative-frequency elements in the fine-grained series.

$$h_0 = \frac{1}{\rho} \left[ h'_0 + 2 \text{ Re} \left( \sum_{\ell=1}^{\ell_{k\text{ max}}} h'_{\ell} + (\lambda_{k\text{ max}} - \ell_{k\text{ max}})h'_{\ell_{k\text{ max}} + 1} \right) \right] \quad (13.3)$$

Algorithm

These routines move through the output series, using (13.2) to add up the contributions from the bins in the fine-grained series.

Uses

strncpy()

Notes

- The coarse graining ratio must obey $\rho \geq 1$ (so the coarse-grained frequency spacing must be less than the fine-grained one). Additionally, the bins in the fine-grained frequency series must *completely* overlap those in the coarse-grained frequency series. In particular, since the lowest frequency in the
first bin of the coarse-grained series is $f_{\text{min}} = f_0 - \delta f / 2$ and the last is $f_{\text{max}} = f_0 + (N - 1)\delta f + \delta f / 2$
(taking into account the width of the bins), the conditions are

\[
\begin{align*}
  f_0 - \delta f / 2 & \geq f_0' - \delta f' / 2 \\
  f_0 + \left(N - \frac{1}{2}\right)\delta f & \leq f_0' + \left(N' - \frac{1}{2}\right)\delta f'
\end{align*}
\]

(The special case $f_0 = f_0' = 0$ is an exception to the condition on the minimum frequency.)

- The routines return an error if either minimum frequency ($f_{\text{min}}$ or $f_{\text{min}}'$) is negative (unless $f_0 = 0$ or $f_0' = 0$, respectively).
13.8.2 Program SCoarseGrainFrequencySeriesTest.c

Test suite for LALS郭arseGrainFrequencySeries().

Usage

./SCoarseGrainFrequencySeriesTest
Options:
  -h  print usage message
  -q  quiet: run silently
  -v  verbose: print extra information
  -d level set lalDebugLevel to level
  -i filename read fine grained series from file filename
  -o filename print coarse grained series to file filename
  -n length input series contains length points
  -m length output series contains length points
  -e deltaF set coarse grained frequency spacing to deltaF
  -f f0 set start frequency of output to f0

Description

This program tests the routine LALS郭arseGrainFrequencySeries(), which coarse-grains a frequency series.

First, it tests that the correct error codes (cf. Sec. 13.8) are generated for the following error conditions (tests in *italics* are not performed if `LAL_NDEBUG` is set, as the corresponding checks in the code are made using the `ASSERT` macro):

- null pointer to output series
- null pointer to input series
- null pointer to data member of output series
- null pointer to data member of input series
- null pointer to data member of data member of input series
- null pointer to data member of data member of output series
- zero length
- negative frequency spacing
- zero frequency spacing

It then verifies that the correct values are obtained for some simple test cases

- \{h'_1\} = \{0, 1, 2, 3, 4, 5, 6, 7\}, \ f'_0 = f_0, \delta f' = 1, \delta f = 2, \ N = 3; \ \text{the expected output is}\ \{h_k\} = {1/2, 2, 4, 6}.
- \{h'_1\} = \{0, 1, 2, 3, 4, 5, 6, 7\}, \ f'_0 = f_0, \delta f' = 1, \delta f = 3, \ N = 3; \ \text{the expected output is}\ \{h_k\} = {2/3, 3, 6}.

For each successful test (both of these valid data and the invalid ones described above), it prints “PASS” to standard output; if a test fails, it prints “FAIL”. If the `filename` arguments are present, it also reads a frequency series from a file, calls LALS郭arseGrainFrequencySeries(), and writes the results to the specified output file.

Exit codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOM</td>
<td>0</td>
<td>&quot;Nominal exit&quot;</td>
</tr>
<tr>
<td>ARG</td>
<td>1</td>
<td>&quot;Error parsing command-line arguments&quot;</td>
</tr>
<tr>
<td>CHK</td>
<td>2</td>
<td>&quot;Error checking failed to catch bad data&quot;</td>
</tr>
<tr>
<td>FLS</td>
<td>3</td>
<td>&quot;Incorrect answer for valid data&quot;</td>
</tr>
<tr>
<td>USE</td>
<td>4</td>
<td>&quot;Bad user-entered data&quot;</td>
</tr>
</tbody>
</table>
The status codes in the table above are stored in the constants `SCOARSEGRAINFREQUENCYSERIES_TESTC_E<name>`, and the status descriptions in `SCOARSEGRAINFREQUENCYSERIES_TESTC_MSGE<name>`. The source code with these messages is in `SCoarseGrainFrequencySeriesTest.c` on line 1.212.

**Uses**

- `LALSCoarseGrainFrequencySeries()`
- `LALCheckMemoryLeaks()`
- `LALSPrintFrequencySeries()`
- `LALCreateVector()`
- `LALSDestroyVector()`
- `LALCHARCreateVector()`
- `LALCHARDestroyVector()`
- `LALUnitAsString()`
- `LALUnitCompare()`
- `getopt()`
- `printf()`
- `fprintf()`
- `freopen()`
- `fabs()`

**Notes**

- In addition to the error checks tested in this routine, the function checks for errors related to inconsistency of coarse graining parameters. Tests of these error checks are still to be added to this test program.

- No specific error checking is done on user-specified data. If `length` is missing, the resulting default will cause a bad data error.

- The length of the user-provided series must be specified, even though it could in principle be deduced from the input file, because the data sequences must be allocated before the `LALSReadFrequencySeries()` function is called.

- If one `filename` argument, but not both, is present, the user-specified data will be silently ignored.
13.8.3 Program DCoarseGrainFrequencySeriesTest.c

Test suite for LALDCoarseGrainFrequencySeries().

Usage

./DCoarseGrainFrequencySeriesTest

Options:

- `h` print usage message
- `q` quiet: run silently
- `v` verbose: print extra information
- `d level` set lalDebugLevel to level
- `i filename` read fine grained series from file filename
- `o filename` print coarse grained series to file filename
- `n length` input series contains length points
- `m length` output series contains length points
- `e deltaF` set coarse grained frequency spacing to deltaF
- `f f0` set start frequency of output to f0

Description

This program tests the routine LALDCoarseGrainFrequencySeries(), which coarse-grains a frequency series.

First, it tests that the correct error codes (cf. Sec. 13.8) are generated for the following error conditions (tests in italics are not performed if LAL_NDEBUG is set, as the corresponding checks in the code are made using the ASSERT macro):

- null pointer to output series
- null pointer to input series
- null pointer to data member of output series
- null pointer to data member of input series
- null pointer to data member of input series
- null pointer to data member of output series
- zero length
- negative frequency spacing
- zero frequency spacing

It then verifies that the correct values are obtained for some simple test cases:

- \( \{h'_k\} = \{0, 1, 2, 3, 4, 5, 6, 7\} \), \( f'_0 = f_0 \), \( \delta f' = 1 \), \( \delta f = 2 \), \( N = 3 \); the expected output is \( \{h_k\} = \{1/2, 2, 4, 6\} \).
- \( \{h'_k\} = \{0, 1, 2, 3, 4, 5, 6, 7\} \), \( f'_0 = f_0 \), \( \delta f' = 1 \), \( \delta f = 3 \), \( N = 3 \); the expected output is \( \{h_k\} = \{2/3, 3, 6\} \).

For each successful test (both of these valid data and the invalid ones described above), it prints “PASS” to standard output; if a test fails, it prints “FAIL”.

If the `filename` arguments are present, it also reads a frequency series from a file, calls LALDCoarseGrainFrequencySeries(), and writes the results to the specified output file.

Exit codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOM</td>
<td>0</td>
<td>&quot;Nominal exit&quot;</td>
</tr>
<tr>
<td>ARG</td>
<td>1</td>
<td>&quot;Error parsing command-line arguments&quot;</td>
</tr>
<tr>
<td>CHK</td>
<td>2</td>
<td>&quot;Error checking failed to catch bad data&quot;</td>
</tr>
<tr>
<td>FLS</td>
<td>3</td>
<td>&quot;Incorrect answer for valid data&quot;</td>
</tr>
<tr>
<td>USE</td>
<td>4</td>
<td>&quot;Bad user-entered data&quot;</td>
</tr>
</tbody>
</table>
The status codes in the table above are stored in the constants `DCOARSEGRAINFREQUENCYSERIESTE="name">`, and the status descriptions in `DCOARSEGRAINFREQUENCYSERIESTE="MSG">`. The source code with these messages is in `DCoarseGrainFrequencySeriesTest.c` on line 1.212.

Uses

- `LALDCoarseGrainFrequencySeries()`
- `LALCheckMemoryLeaks()`
- `LALDReadFrequencySeries()`
- `LALDPrintFrequencySeries()`
- `LALDCreateVector()`
- `LALDDestroyVector()`
- `LALCHARCreateVector()`
- `LALCHARDestroyVector()`
- `LALUnitAsString()`
- `LALUnitCompare()`
- `getopt()`
- `printf()`
- `fprintf()`
- `freopen()`
- `fabs()`

Notes

- In addition to the error checks tested in this routine, the function checks for errors related to inconsistency of coarse graining parameters. Tests of these error checks are still to be added to this test program.
- No specific error checking is done on user-specified data. If `length` is missing, the resulting default will cause a bad data error.
- The length of the user-provided series must be specified, even though it could in principle be deduced from the input file, because the data sequences must be allocated before the `LALDReadFrequencySeries()` function is called.
- If one `filename` argument, but not both, is present, the user-specified data will be silently ignored.
13.8.4 Program CCoarseGrainFrequencySeriesTest.c

Test suite for LALCCoarseGrainFrequencySeries().

Usage

./CCoarseGrainFrequencySeriesTest
Options:
- h print usage message
- q quiet: run silently
- v verbose: print extra information
- d level set lalDebugLevel to level
- i filename read fine grained series from file filename
- o filename print coarse grained series to file filename
- n length input series contains length points
- m length output series contains length points
- e deltaF set coarse grained frequency spacing to deltaF
- f f0 set start frequency of output to f0

Description

This program tests the routine LALCCoarseGrainFrequencySeries(), which coarse-grains a frequency series.

First, it tests that the correct error codes (cf. Sec. 13.8) are generated for the following error conditions (tests in italics are not performed if LAL_NDEBUG is set, as the corresponding checks in the code are made using the ASSERT macro):

- null pointer to output series
- null pointer to input series
- null pointer to data member of output series
- null pointer to data member of input series
- null pointer to data member of data member of input series
- null pointer to data member of data member of output series
- zero length
- negative frequency spacing
- zero frequency spacing

It then verifies that the correct values are obtained for some simple test cases

- \( \{h_k'\} = \{0, 1, 2, 3, 4, 5, 6, 7\}, f_0' = f_0, \delta f' = 1, \delta f = 2, N = 3; \) the expected output is \( \{h_k\} = \{1/2, 2, 4, 6\} \).

- \( \{h_k'\} = \{0, 1, 2, 3, 4, 5, 6, 7\}, f_0' = f_0, \delta f' = 1, \delta f = 3, N = 3; \) the expected output is \( \{h_k\} = \{2/3, 3, 6\} \).

- \( f_0' = 40, \delta f' = 1, \{h_k\} = \{f_k + i f_k^{-1}|k = 0, \ldots, 4\}, f_0 = 41, f_0 = f_0' , \delta f = 2. \delta f' = 3, N = ; \) the expected output is

\[
\{h_k'\} = \left\{ 41 + i \left( \frac{1}{40} + \frac{2}{41} + \frac{1}{42} \right) , 43 + i \left( \frac{1}{42} + \frac{2}{43} + \frac{1}{44} \right) \right\}
\]

For each successful test (both of these valid data and the invalid ones described above), it prints “PASS” to standard output; if a test fails, it prints “FAIL”.

If the filename arguments are present, it also reads a frequency series from a file, calls LALCCoarseGrainFrequencySeries(), and writes the results to the specified output file.
Exit codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOM</td>
<td>0</td>
<td>&quot;Nominal exit&quot;</td>
</tr>
<tr>
<td>ARG</td>
<td>1</td>
<td>&quot;Error parsing command-line arguments&quot;</td>
</tr>
<tr>
<td>CHK</td>
<td>2</td>
<td>&quot;Error checking failed to catch bad data&quot;</td>
</tr>
<tr>
<td>FLS</td>
<td>3</td>
<td>&quot;Incorrect answer for valid data&quot;</td>
</tr>
<tr>
<td>USE</td>
<td>4</td>
<td>&quot;Bad user-entered data&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants CCOARSEGRAINFREQUENCYSERIES_TESTC_E<name>, and the status descriptions in CCOARSEGRAINFREQUENCYSERIES_TESTC_MSGE<name>. The source code with these messages is in CCoarseGrainFrequencySeriesTest.c on line 1.245.

Uses

- LALCoarseGrainFrequencySeries()
- LALCheckMemoryLeaks()
- LALCReadFrequencySeries()
- LALCPrintFrequencySeries()
- LALCCreateVector()
- LALDestroyVector()
- LALCHARCreateVector()
- LALCHARDestroyVector()
- LALUnitAsString()
- LALUnitCompare()
- getopt()
- printf()
- fprintf()
- freopen()
- fabs()

Notes

- In addition to the error checks tested in this routine, the function checks for errors related to inconsistency of coarse graining parameters, as well as duplicate input and output pointers. Tests of these error checks are still to be added to this test program.

- No specific error checking is done on user-specified data. If length is missing, the resulting default will cause a bad data error.

- The length of the user-provided series must be specified, even though it could in principle be deduced from the input file, because the data sequences must be allocated before the LALCReadFrequencySeries() function is called.

- If one filename argument, but not both, is present, the user-specified data will be silently ignored.
13.8.5 Program ZCoarseGrainFrequencySeriesTest.c

Test suite for LALZCoarseGrainFrequencySeries().

Usage

./ZCoarseGrainFrequencySeriesTest

Options:
- h print usage message
- q quiet: run silently
- v verbose: print extra information
- d level set lalDebugLevel to level
- i filename read fine grained series from file filename
- o filename print coarse grained series to file filename
- n length input series contains length points
- m length output series contains length points
- e deltaF set coarse grained frequency spacing to deltaF
- f f0 set start frequency of output to f0

Description

This program tests the routine LALZCoarseGrainFrequencySeries(), which coarse-grains a frequency series.

First, it tests that the correct error codes (cf. Sec. 13.8) are generated for the following error conditions (tests in italics are not performed if LAL_NDEBUG is set, as the corresponding checks in the code are made using the ASSERT macro):

- null pointer to output series
- null pointer to input series
- null pointer to data member of output series
- null pointer to data member of input series
- null pointer to data member of data member of input series
- null pointer to data member of data member of output series
- zero length
- negative frequency spacing
- zero frequency spacing

It then verifies that the correct values are obtained for some simple test cases

- \{h'_k\} = \{0, 1, 2, 3, 4, 5, 6, 7\}, f'_0 = f_0, \delta f' = 1, \delta f = 2, N = 3; the expected output is \{h_k\} = \{1/2, 2, 4, 6\}.
- \{h'_k\} = \{0, 1, 2, 3, 4, 5, 6, 7\}, f'_0 = f_0, \delta f' = 1, \delta f = 3, N = 3; the expected output is \{h_k\} = \{2/3, 3, 6\}.
- f'_0 = 40, \delta f' = 1, \{h_k\} = \{f_k + i f_k^{-1}|k = 0,...,4\}, f_0 = 41, f_0 = f'_0, \delta f = 2. \delta f' = 3, N = ; the expected output is

\[ \{h'_k\} = \left\{ 41 + i \left( \frac{1}{40} + \frac{2}{41} + \frac{1}{42} \right), 43 + i \left( \frac{1}{42} + \frac{2}{43} + \frac{1}{44} \right) \right\} \]

For each successful test (both of these valid data and the invalid ones described above), it prints “PASS” to standard output; if a test fails, it prints “FAIL”.

If the filename arguments are present, it also reads a frequency series from a file, calls LALZCoarseGrainFrequencySeries(), and writes the results to the specified output file.
Exit codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOM</td>
<td>0</td>
<td>&quot;Nominal exit&quot;</td>
</tr>
<tr>
<td>ARG</td>
<td>1</td>
<td>&quot;Error parsing command-line arguments&quot;</td>
</tr>
<tr>
<td>CHK</td>
<td>2</td>
<td>&quot;Error checking failed to catch bad data&quot;</td>
</tr>
<tr>
<td>FLS</td>
<td>3</td>
<td>&quot;Incorrect answer for valid data&quot;</td>
</tr>
<tr>
<td>USE</td>
<td>4</td>
<td>&quot;Bad user-entered data&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants `ZCOARSEGRAINFREQUENCYSERIES_TESTC_E<name>`, and the status descriptions in `ZCOARSEGRAINFREQUENCYSERIES_TESTC_MSGE<name>`. The source code with these messages is in `ZCoarseGrainFrequencySeriesTest.c` on line 1.230.

Uses

- `LALZCoarseGrainFrequencySeries()`
- `LALCheckMemoryLeaks()`
- `LALZReadFrequencySeries()`
- `LALZPrintFrequencySeries()`
- `LALZCreateVector()`
- `LALZDestroyVector()`
- `LALCHARCreateVector()`
- `LALCHARDestroyVector()`
- `LALUnitAsString()`
- `LALUnitCompare()`
- `getopt()`
- `printf()`
- `fprintf()`
- `freopen()`
- `fabs()`

Notes

- In addition to the error checks tested in this routine, the function checks for errors related to inconsistency of coarse graining parameters, as well as duplicate input and output pointers. Tests of these error checks are still to be added to this test program.

- No specific error checking is done on user-specified data. If `length` is missing, the resulting default will cause a bad data error.

- The length of the user-provided series must be specified, even though it could in principle be deduced from the input file, because the data sequences must be allocated before the `LALZReadFrequencySeries()` function is called.

- If one `filename` argument, but not both, is present, the user-specified data will be silently ignored.
13.9 Header MatrixUtils.h

Provides routines to solve linear systems.

Synopsis

#include <lal/MatrixUtils.h>

This header covers routines to solve linear systems of equations and eigensystems, using algorithms adapted from Chapters 2 and 11 of Numerical Recipes [1]. The only routines at present are for computing eigenvalues and eigenvectors of real symmetric matrices. Routines for inverting or computing the determinant of arbitrary square matrices will likely follow.

Notation

A matrix is represented in LAL by a <datatype>Array structure with a dimLength->length field of 2; the dimLength->data field gives the dimensions \([M,N]\) of the matrix. Using the place-index notation common in tensor calculus, a matrix is a two-index tensor:

\[
A^a_b = \begin{bmatrix}
A^1_1 & A^1_2 & \cdots & A^1_N \\
A^2_1 & A^2_2 & \cdots & A^2_N \\
\vdots & \vdots & \ddots & \vdots \\
A^{M}_1 & A^{M}_2 & \cdots & A^{M}_N \\
\end{bmatrix},
\]

(13.4)

that is, the first (raised) index represents the row number and the second (lowered) index the column number. The standard C array structure declares this object as, say, float a[M][N], where the element \(A^i_j\) is stored in a[i][j]. The LAL array structure REAL4Array a stores data in a flattened block of memory, where the element \(A^{i}_j\) is stored in a.data[i*M+j].

A row vector is a matrix with only one row \((M = 1)\). In the above place-index notation, it is represented with a single lowered index:

\[
r_a = \begin{bmatrix}
 r_1 \\
 r_2 \\
 \vdots \\
 r_N \\
\end{bmatrix}.
\]

(13.5)

A column vector is a matrix with only one column \((N = 1)\). In the above place-index notation, it is represented with a single raised index:

\[
c^a = \begin{bmatrix}
 c^1 \\
 c^2 \\
 \vdots \\
 c^M \\
\end{bmatrix}.
\]

(13.6)

In LAL, both of these objects are conventionally represented as a LAL vector structure. Whether the object is to be used as a row or column vector must be determined from context; it is not specified by the datatype.

Properties

The basic matrix operations are addition, scalar multiplication, and vector multiplication. We assume the reader is familiar with these. In addition, we will refer to the following unary operations on square matrices:

**Inversion:** The inverse \((A^{-1})^a_b\) of a matrix \(A^a_b\) is one such that their matrix product is the identity matrix \(\delta^a_b\) (whose elements \(\delta^i_j\) are just the Kronecker delta function).

**Transposition:** The transpose \((A^T)^a_b\) of a matrix \(A^a_b\) is given by interchanging the indecies on each component: \((A^T)^i_j = A^j_i\).

**Conjugation:** The Hermitian conjugate (adjoint) \((A^\dagger)^a_b\) of a matrix \(A^a_b\) is given by interchanging the indecies and taking the complex conjugate of each component: \((A^\dagger)^i_j = A^j_i^*\).

A matrix that is identical to its own transpose is called *symmetric*. A matrix whose transpose is identical to the original matrix’s inverse is called *orthogonal*. A matrix that is identical to its own Hermitian conjugate is called *Hermitian* (or *self-adjoint*). A matrix whose Hermitian conjugate is identical to the original matrix’s inverse is called *unitary*.

At present, the routines under this header only deal with real matrices (i.e. matrices, vectors, and scalars whose components are all real). In this case, symmetric is equivalent to Hermitian, and orthogonal is equivalent to unitary.
Error conditions

<table>
<thead>
<tr>
<th>name</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUL</td>
<td>1</td>
<td>&quot;Unexpected null pointer in arguments&quot;</td>
</tr>
<tr>
<td>DIM</td>
<td>2</td>
<td>&quot;Bad matrix dimensions&quot;</td>
</tr>
<tr>
<td>ITER</td>
<td>3</td>
<td>&quot;Did not converge after maximum iterations&quot;</td>
</tr>
<tr>
<td>SING</td>
<td>4</td>
<td>&quot;Singular matrix&quot;</td>
</tr>
<tr>
<td>MEM</td>
<td>5</td>
<td>&quot;Memory allocation error&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants MATRIXUTILSH_E<name>, and the status descriptions in MATRIXUTILSH_MSGE<name>. The source code with these messages is in MatrixUtils.h on line 1149.
13.9.1 Module MatrixOps.c

Routines to perform basic matrix operations.

Prototypes

void
LAL<datatype>MatrixAdd( LALStatus *stat,
<datatype>Array *out,
<datatype>Array *in1,
<datatype>Array *in2 )

void
LAL<datatype>MatrixMultiply( LALStatus *stat,
<datatype>Array *out,
<datatype>Array *in1,
<datatype>Array *in2 )

void
LAL<datatype>MatrixTranspose( LALStatus *stat,
<datatype>Array *out,
<datatype>Array *in )

void
LALCMatrixAdjoint( LALStatus *stat, COMPLEX8Array *out, COMPLEX8Array *in1 )

void
LALZMatrixAdjoint( LALStatus *stat, COMPLEX16Array *out, COMPLEX16Array *in1 )

Description

The routines LAL<datatype>MatrixAdd() add the matrices *in1 and *in2 element-by-element, storing the result in *out. All of these matrices must have the same dimensionality. The addition may be performed in-place by pointing *out to the same structure as either *in1 or *in2.

The routines LAL<datatype>MatrixMultiply() perform matrix multiplication, contracting the columns of *in1 against the rows of *in2, and storing the result in *out. The number of columns of *in1 must equal the number of rows of *in2, and *out must have the same number of columns as *in1 and the same number of rows as *in2.

The routines LAL<datatype>MatrixTranspose() take the transpose of the matrix *in and store the result in *out. The number of rows of *out must equal the number of columns of *in, and vice-versa.

The routines LALCMatrixAdjoint() and LALZMatrixAdjoint() take the Hermitian conjugate (adjoint) of the matrix *in and store the result in *out: this involves transposing the matrix and taking the complex conjugate. The number of rows of *out must equal the number of columns of *in, and vice-versa.

Except for the adjoint routines, the prototype templates above in fact refer to 10 separate routines each, corresponding to all the numerical atomic datatypes <datatype> referred to by <typecode>:

<table>
<thead>
<tr>
<th>&lt;typecode&gt;</th>
<th>&lt;datatype&gt;</th>
<th>&lt;typecode&gt;</th>
<th>&lt;datatype&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>I2</td>
<td>INT2</td>
<td>U2</td>
<td>UINT2</td>
</tr>
<tr>
<td>I4</td>
<td>INT4</td>
<td>U4</td>
<td>UINT4</td>
</tr>
<tr>
<td>I8</td>
<td>INT8</td>
<td>U8</td>
<td>UINT8</td>
</tr>
<tr>
<td>S</td>
<td>REAL4</td>
<td>C</td>
<td>COMPLEX8</td>
</tr>
<tr>
<td>D</td>
<td>REAL8</td>
<td>Z</td>
<td>COMPLEX16</td>
</tr>
</tbody>
</table>

Algorithm

Matrix addition is simply carried through element-by-element. It involves one addition operation per element of the output matrix.

The matrix product $Z_{ab} = X_{ac}Y_{cb}$ of two matrices $X_{ac}$ and $Y_{cb}$ is given by the element formula $Z_{ij} = \sum_{k=1}^{N} X_{ik}Y_{kj}$, where $N$ is the number of columns of $X_{ac}$ and the number of rows of $Y_{cb}$. This can also be
used to compute the inner product of two vectors $x^a$ and $y^a$: simply store the transpose $(x^T)_a$ as a row vector (single-row matrix) as the first operand, and $y^a$ as a column vector (single-column matrix) as the second operand. To compute the vector outer product, simply transpose the second argument rather than the first. These computations involve $N$ additions and multiplications per element of the output matrix.

The transpose $(X^T)_{ab}$ of a matrix $X_{ab}$ is given by $(X^T)_{ij} = X_{ji}$. The adjoint $(X^\dagger)_{ab}$ of a complex matrix $X_{ab}$ is given by $(X^\dagger)_{ij} = X_{ji}^\ast$, where $\ast$ denotes complex conjugation. Transposition involves no arithmetic operations, just one assignment per element of the output. Conjugation involves one multiplication (negating the sign of the imaginary part) per element.
13.9.2 Module DetInverse.c

Routines to compute matrix determinants and inverses.

Prototypes

```c
void LALSMatrixDeterminant( LALStatus *stat, REAL4 *det, REAL4Array *matrix )
void LALSMatrixInverse( LALStatus *stat, REAL4 *det, REAL4Array *matrix, REAL4Array *inverse )
void LALSMatrixDeterminantErr( LALStatus *stat, REAL4 det[2], REAL4Array *matrix, REAL4Array *matrixErr )
void LALDMatrixDeterminant( LALStatus *stat, REAL8 *det, REAL8Array *matrix )
void LALDMatrixInverse( LALStatus *stat, REAL8 *det, REAL8Array *matrix, REAL8Array *inverse )
void LALDMatrixDeterminantErr( LALStatus *stat, REAL8 det[2], REAL8Array *matrix, REAL8Array *matrixErr )
```

Description

**LALSMatrixDeterminant**() and **LALDMatrixDeterminant**() compute the determinant *det* of the square matrix *matrix*. The internal computations will corrupt *matrix*; if you don’t want the input matrix to be changed, make a copy of it first.

**LALSMatrixInverse**() and **LALDMatrixInverse**() compute the inverse *inverse* of the square matrix *matrix*. If the pointer *det* is non-NULL, then the determinant is also computed and returned in *det*. The array *inverse* must already be allocated to the correct size. The internal computations will corrupt *matrix*; if you don’t want the input matrix to be changed, make a copy of it first.

**LALSMatrixDeterminantErr**() and **LALDMatrixDeterminantErr**() compute the determinant det[0] of the square matrix *matrix*. If *matrixErr* is non-NULL, it stores uncertainties in the corresponding components of *matrix*, and the resulting uncertainty in the determinant (computed by linear error propagation) is returned in det[1]. This routine is not highly optimized, and uses an internal matrix in its computations, so the contents of *matrix* and *matrixErr* will not be changed.

Algorithm

A linear system of equations is written in matrix terms as:

\[ M^a_{\vec{b}}x^b = v^a, \]  

(13.7)

where \( M^a_{\vec{b}} \) is a known matrix, \( v^a \) a known vector, and \( x^b \) an unknown vector that we wish to solve for.

A standard method for solving this is the method of LU decomposition, based on the fact that any non-singular square matrix \( M^a_{\vec{b}} \) can be decomposed into the product of a lower-triangular matrix \( L^a_{\vec{b}} \) and an upper-triangular matrix \( U^a_{\vec{b}} \):

\[
M^a_{\vec{b}} = L^a_{\vec{c}} U^c_{\vec{b}} = \begin{bmatrix} L_2^1 & 0 & \ldots & 0 \\ L_3^1 & 1 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ L_N^1 & L_N^2 & \ldots & 1 \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} & \cdots & U_{1N} \\ 0 & U_{22} & \cdots & U_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & U_{NN} \end{bmatrix}.
\]  

(13.8)

The linear system can then be broken up as \( L^a_{\vec{b}} y^b = v^a \) and \( U^b_{\vec{c}} x^c = y^b \), where these two equations are trivially invertible:

\[
y^i = v^i - \sum_{j=1}^i -1L_{ij}^j y^j, \quad i = 1, 2, \ldots, N,
\]  

(13.9)

\[
x^i = \frac{1}{U_{ii}^1} \left( y^i - \sum_{j=i+1}^N U_{ij}^j x_j \right), \quad i = N, N - 1, \ldots, 1,
\]  

(13.10)
where the calculations are arranged so that the computation of a given \( y^i \) or \( x^i \) requires only those values that have been computed already. This process of solving the linear system is called backsubstitution.

The determinant of \( M^a_b \) is simply the product of the diagonal elements of the decomposed matrix: \( |M^a_b| = \prod_{j=1}^{N} U^a_j \). The inverse matrix \((M^{-1})^a_b\) can be computed by performing a column-by-column backsubstitution of the identity matrix.

The routines in \texttt{DetInverse.c} simply call the routines in \texttt{DetInverseInternal.c}, first using \texttt{LALSLUDecomp()} or \texttt{LALDLUDecomp()} to perform an LU decomposition of the matrix, then either computing the determinant from the diagonal elements, or using \texttt{LALSLUBackSub()} or \texttt{LALDLUBackSub()} to determine the inverse by back-substitution of basis vectors. The routines that compute the determinant will also handle any “singular matrix” error code returned by the LU decomposition routines, returning zero as the determinant.

Since the diagonal components \( L^a_j \) are conventionally assigned to 1, they do not need to be explicitly stored. Therefore we can store both matrices \( L^a_b \) and \( U^a_b \) “in-place”, in the same memory block used for the input matrix \( M^a_b \). This the procedure taken by \texttt{LALSLUDecomp()} and \texttt{LALDLUDecomp()}; hence on return the routines in this module will leave the input \*\texttt{matrix} in this decomposed state. However, these routines also permute the rows of the input matrix, and the information on this permutation is not returned by the routines in \texttt{DetInverse.c}, so the information in \*\texttt{matrix} will be irretrievably mangled. If you want to do further work with the LU-decomposed matrix, call the routines in \texttt{DetInverseInternal.c} directly.

Computing the determinant is dominated by the cost of doing the LU decomposition, or of order \( N^3 / 3 \) operations. Computing the inverse requires an additional \( N \) back-substitutions, but since most of the vector elements are zero, this reduces the average cost of each back-substitution from \( N^2 \) to \( 2N^2 / 3 \), for a total operation count of \( N^3 \).

**Computing determinant uncertainties:** To determine the dependence of the determinant on any one element of the matrix, we take advantage of the fact that the determinant \(|M^a_b|\) can be written as:

\[
|M^a_b| = \sum_{j=1}^{N} (-1)^{i+j} M^i_j (C^i_j)^a_b \quad \text{for any } i \in [1, N],
\]

\[
= \sum_{i=1}^{N} (-1)^{i+j} M^i_j (C^i_j)^a_b \quad \text{for any } j \in [1, N],
\]

(13.11)

where

\[
(C^i_j)^a_b = \begin{bmatrix}
M^1_1 & \cdots & M^1_{j-1} & 0 & M^1_{j+1} & \cdots & M^1_N \\
\vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\
M^{i-1}_1 & \cdots & M^{i-1}_{j-1} & 0 & M^{i-1}_{j+1} & \cdots & M^{i-1}_N \\
0 & \cdots & 0 & 1 & 0 & \cdots & 0 \\
\vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\
M^N_1 & \cdots & M^N_{j-1} & 0 & M^N_{j+1} & \cdots & M^N_N 
\end{bmatrix}
\]

(13.12)

is called the co-matrix of the element \( M^i_j \).

Assuming all matrix elements are statistically independent, linear error propagation can give us the uncertainty in the determinant:

\[
E(|M^a_b|) = \sqrt{\sum_{i,j=1}^{N} \left( \frac{\partial |M^a_b|}{\partial M^i_j} E(M^i_j) \right)^2}
\]

\[
= \sqrt{\sum_{i,j=1}^{N} ((C^i_j)^a_b E(M^i_j))^2}. \quad (13.13)
\]

(13.14)

The routines \texttt{LALSMatrixDeterminantErr()} and \texttt{LALDMatrixDeterminantErr()} thus simply compute the determinant multiple times, successively zeroing out rows and columns of \*\texttt{matrix} and replacing the element at their intersection with the corresponding element of \*\texttt{matrixErr}, and take the square root of the sum of the squares of these determinants. As mentioned earlier, they use an internal matrix for all computations, so the input parameters are unchanged. The uncertainty requires evaluating \( N^2 \) determinants, so the operation count scales as \( N^5 \): this routine should \textit{not} be used for large matrices!
Uses

LALU4CreateVector() LALU4DestroyVector()
LALSCreateVector() LALSDestroyVector()
LALDCreateVector() LALDDestroyVector()
LALSLUDecom() LALSLUBackSub()
LALDLUDecom() LALDLUBackSub()

Notes
13.9.3 Module DetInverseInternal.c

Internal routines used to compute matrix determinants and inverses.

Prototypes

```c
void LALSLUDecomp( LALStatus *stat,
                    INT2 *sgn,
                    REAL4Array *matrix,
                    UINT4Vector *indx )
```

```c
void LALSLUBackSub( LALStatus *stat,
                    REAL4Vector *vector,
                    REAL4Array *matrix,
                    UINT4Vector *indx )
```

```c
void LALDLUDecomp( LALStatus *stat,
                    INT2 *sgn,
                    REAL8Array *matrix,
                    UINT4Vector *indx )
```

```c
void LALDLUBackSub( LALStatus *stat,
                    REAL8Vector *vector,
                    REAL8Array *matrix,
                    UINT4Vector *indx )
```

Description

These functions are called by the routines in DetInverse.c to compute the determinant and inverse of a nondegenerate square matrix *matrix*. They are useful routines in their own right, though, so they are made publically available.

LALSLUDecomp() and LALDLUDecomp() replace *matrix* with an LU decomposition of a row-wise permutation of itself. The output parameter *indx* stores the permutation, and the output *sgn* records the sign of the permutation.

LALSLUBackSub() and LALDLUBackSub() take the permuted LU-decomposed matrix returned by the above routine, and back-substitute the vector *vector* representing $v^a$ in Eq. (13.7), to compute the vector $x^b$. This is returned in-place in *vector*. The input parameter *indx* is the list of row permutations returned by the above routines.

Algorithm

LU decomposition is performed by Crout’s algorithm, described in Sec. 2.3 of [1]; the routines in this module are essentially re-implementations of the Numerical Recipes routines ludcmp() and lubksb(). For large $N$, their operation counts are approximately $N^3/3$ and $N^2$, respectively.

One difference between ludcmp() in [1] and the routines LALSLUDecomp() and LALDLUDecomp() in this module is the way in which singular matrices are handled. In [1], there is a distinction between between a manifestly singular matrix (where an entire row of the matrix is zero) and a numerically singular matrix (if a diagonal element in the decomposed matrix turns out to be zero). In the former case, they raise an error signal; in the latter, they replace the offending element with a “tiny” but nonzero number and continue. This treatment does not strike the present author as satisfactory.

Instead, the routines LALSLUDecomp() and LALDLUDecomp() will always return successfully, even with a singular matrix, but will not “adjust away” a numerical singularity. Instead, they will signal the presence of the singularity in two ways: First, they will set the permutation sign *sgn* to zero; second, they will set all elements of the *indx* vector to zero. This ensures that routines computing the determinant (whose sign depends on *sgn*) will correctly give a zero determinant, while the meaningless *indx* provides a simple sanity check for routines such as LALSLUBackSub() and LALDLUBackSub() that attempt to invert the linear system. Note that the returned value of *matrix* will be meaningless garbage.
13.9. Header MatrixUtils.h

Uses

LALMalloc() LALFree()

Notes
13.9.4 Program DetInverseTest.c

Computes the inverse and determinant of a matrix.

Usage


Description

This program computes the inverse and determinant of a square real matrix using the routines in DetInverse.c and DetInverseInternal.c. The following option flags are accepted:

- **-n** Generates a random symmetric size×size metric. If this option is not given, -n 3 is assumed. This option (or its default) is overridden by the -i option, below.

- **-i** Reads a matrix from an input file infile using the function LALSReadVector(). If the input file is specified as stdin, the data is read from standard input (not a file named stdin).

- **-o** Writes the determinant and inverse matrix to an output file outfile. If the output file is specified as stdout or stderr, the data is written to standard output or standard error (not to files named stdout or stderr).

- **-v** Specifies that the inverse matrix is to be computed as well the determinant.

- **-t** Specifies that the computation is to be timed; timing information is written to stderr.

- **-s** Specifies that the calculations are to be done to single-precision (REAL4) rather than double-precision (REAL8).

- **-d** Sets the debug level to debuglevel. If not specified, level 0 is assumed.

Input format: If an input file or stream is specified, it should consist of N consecutive lines of N whitespace-separated numbers, that will be parsed using LALDReadVector(), or LALSReadVector() if the -s option was given. The data block may be preceded by blank or comment lines (lines containing no parseable numbers), but once a parseable number is found, the rest should follow in a contiguous block. If the lines contain different numbers of data columns, or if there are fewer lines than columns, then an error is returned; if there are more lines than columns, then the extra lines are ignored.

Output format: If an output file or stream is specified, the input matrix is first written as N consecutive lines of N whitespace-separated numbers. This will be followed with a blank line, then a single number representing the determinant. If the -v option is specified, then another blank line will be appended to the output, followed by the inverse matrix written as N lines of N whitespace-separated numbers.

Exit codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NORM</td>
<td>0</td>
<td>&quot;Normal exit&quot;</td>
</tr>
<tr>
<td>SUB</td>
<td>1</td>
<td>&quot;Subroutine failed&quot;</td>
</tr>
<tr>
<td>ARG</td>
<td>2</td>
<td>&quot;Error parsing arguments&quot;</td>
</tr>
<tr>
<td>MEM</td>
<td>3</td>
<td>&quot;Out of memory&quot;</td>
</tr>
<tr>
<td>FILE</td>
<td>4</td>
<td>&quot;Could not open file&quot;</td>
</tr>
<tr>
<td>FMT</td>
<td>5</td>
<td>&quot;Bad input file format&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants DETINVERSETESTC_E<name>, and the status descriptions in DETINVERSETESTC_MSGE<name>. The source code with these messages is in DetInverseTest.c on line 1.100.
Algorithm

Uses

- lalDebugLevel
- LALPrintError() LALCheckMemoryLeaks()
- LALSCreateVector() LALSDestroyVector()
- LALDCreateVector() LALDDestroyVector()
- LALSCreateArray() LALSDestroyArray()
- LALDCreateArray() LALDDestroyArray()
- LALSMatrixDeterminant() LALSMatrixInverse()
- LALDMatrixDeterminant() LALDMatrixInverse()
- LALCreateRandomParams() LALDestroyRandomParams()
- LALUniformDeviate()

Notes
13.9.5 Module Eigen.c

Routines to compute eigenvalues and eigenvectors.

Prototypes

```c
void LALSSymmetricEigenVectors( LALStatus *stat, REAL4Vector *values, REAL4Array *matrix )

void LALSSymmetricEigenValues( LALStatus *stat, REAL4Vector *values, REAL4Array *matrix )

void LALDSymmetricEigenVectors( LALStatus *stat, REAL8Vector *values, REAL8Array *matrix )

void LALDSymmetricEigenValues( LALStatus *stat, REAL8Vector *values, REAL8Array *matrix )
```

Description

LALSSymmetricEigenVectors() and LALDSymmetricEigenVectors() compute the eigenvalues and eigenvectors of the square matrix *matrix*. The eigenvalues are stored in *values*, which must be pre-allocated to the same length as each of the dimensions of *matrix*. The eigenvectors are computed in-place and returned in *matrix*: on return, each column of *matrix* stores the eigenvector of the corresponding eigenvalue as a column vector. If you don’t want the input matrix to be changed, make a copy of it first.

LALSSymmetricEigenValues() and LALDSymmetricEigenValues() compute just the eigenvalues of the square matrix *matrix*, which is significantly faster than computing the eigenvectors. The eigenvalues are stored in *values*, which must be pre-allocated to the same length as each of the dimensions of *matrix*. However, *matrix* is still used as auxiliary storage for the in-place algorithm; if you don’t want the input matrix to be changed, make a copy of it first.

Algorithm

A square matrix $M_{ab}$ is said to have an eigenvalue $\lambda$ and associated eigenvector $x^a$ if the following equation holds:

$$M_{ab}x^b = \lambda x^a$$  \hspace{1cm} (13.15)

Generically an $N \times N$ matrix has $N$ distinct eigenvalues each with a unique (up to a multiplicative constant) associated eigenvector. In certain cases, though, the system is degenerate: that is, some of the $N$ eigenvalues may be the same, and the corresponding eigenvectors may not be unique.

We are concerned with the particular case of real, symmetric matrices, which occur in a broad class of problems. These matrices have certain useful properties. First, for any Hermitian matrix (including real symmetric matrices) the eigenvalues are all real. Second, for any matrix that commutes with its Hermitian conjugate (including Hermitian and unitary matrices, or, for real matrices, symmetric and orthogonal), the eigenvectors $x^{(i)}_a$ of distinct eigenvalues $\lambda^{(i)}$ are orthogonal: that is, $(x^{(i)^T}_a)x^{(j)}_a = 0$ if $\lambda^{(i)} \neq \lambda^{(j)}$. Furthermore, we note that if two or more linearly independent eigenvectors have the same eigenvalue, then any linear combination of them is also an eigenvector with that same eigenvalue: we are therefore free to choose a basis of eigenvectors that is orthonormal. Thirdly, for any matrix that commutes with its conjugate, the complete set of eigenvectors spans the entire $N$-dimensional vector space, so the orthonormal basis above is a complete orthonormal basis.

If we construct a square matrix $X^{a}_{b}$ whose columns are the orthonormal eigenvectors $X^{i}_{j} = x^{(i)}_a$, then it is clear (from the orthonormality of $x^{(i)}_a$) that $X^{a}_{b}$ is orthogonal. Furthermore, the eigenvalue equation \((13.15)\) gives:

$$ (X^{-1})^a_bM^a_{cd}X^c_d = \begin{bmatrix} \lambda^{(1)} & 0 & \cdots & 0 \\ 0 & \lambda^{(2)} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda^{(N)} \end{bmatrix} $$  \hspace{1cm} (13.16)
where $\lambda_{(i)}$ are the eigenvalues of the corresponding eigenvectors (with the possibility that some of these eigenvalues have the same value). That is, the matrix $M_{bc}^c$ can be diagonalized by an orthogonal similarity transformation, and the transformation matrix gives us the eigenvectors. The process of solving the eigenvalue equation is thus equivalent to the problem of diagonalizing the matrix.

For a general $N \times N$ symmetric matrix, there is no finite algorithm that exactly diagonalizes the matrix. Most numerical eigenvalue solvers use a process of iterative orthogonal transformations, where each transformation is chosen to reduce the sum of the squares of the off-diagonal elements: the matrix uniformly converges to a diagonal form under successive transformations. This convergence can be speeded if one first transforms the matrix into a tridiagonal form (where the only nonzero elements are on the diagonal or immediately adjacent to it); this transformation can be accomplished deterministically in a finite number of steps. This is the approach advocated in Chapter 11 of [1], and taken in this module.

The routines in `Eigen.c` simply call the routines in `EigenInternal.c`, first to reduce the matrix to tridiagonal form, and then to reduce it to diagonal form iteratively. The routines `LALSSymmetricEigenVectors()` and `LALDSymmetricEigenVectors()` call the routines `LALSSymmetricToTriDiagonal()` and `LALDSymmetricToTriDiagonal()` to tri-diagonalize the matrix, then `LALSTriDiagonalToDiagonal()` and `LALDTriDiagonalToDiagonal()` to diagonalize it. The routines `LALSSymmetricEigenValues()` and `LALDSymmetricEigenValues()` instead call `LALSSymmetricToTriDiagonal2()` or `LALDSymmetricToTriDiagonal2()`, which discard information about the transformation matrix itself and thus do not give the eigenvectors; however, they execute much faster.

### Uses

- `LALSCreateVector()`, `LALSDestroyVector()`
- `LALDCreateVector()`, `LALDDestroyVector()`
- `LALSSymmetricToTriDiagonal()`, `LALSTriDiagonalToDiagonal()`
- `LALDSymmetricToTriDiagonal()`, `LALDTriDiagonalToDiagonal()`
- `LALSSymmetricToTriDiagonal2()`, `LALSTriDiagonalToDiagonal2()`
- `LALDSymmetricToTriDiagonal2()`, `LALDTriDiagonalToDiagonal2()`

### Notes
13.9.6 Module `EigenInternal.c`

Internal routines used to compute eigenvalues and eigenvectors.

Prototypes

```c
void LALSSymmetricToTriDiagonal( LALStatus *stat,
                                   REAL4Vector *diag,
                                   REAL4Array *matrix,
                                   REAL4Vector *offDiag )

void LALSSymmetricToTriDiagonal2( LALStatus *stat,
                                   REAL4Vector *diag,
                                   REAL4Array *matrix,
                                   REAL4Vector *offDiag )

void LALSTriDiagonalToDiagonal( LALStatus *stat,
                                   REAL4Vector *diag,
                                   REAL4Array *matrix,
                                   REAL4Vector *offDiag )

void LALSTriDiagonalToDiagonal2( LALStatus *stat,
                                   REAL4Vector *diag,
                                   REAL4Array *matrix,
                                   REAL4Vector *offDiag )

void LALDSymmetricToTriDiagonal( LALStatus *stat,
                                   REAL8Vector *diag,
                                   REAL8Array *matrix,
                                   REAL8Vector *offDiag )

void LALDSymmetricToTriDiagonal2( LALStatus *stat,
                                   REAL8Vector *diag,
                                   REAL8Array *matrix,
                                   REAL8Vector *offDiag )

void LALDTriDiagonalToDiagonal( LALStatus *stat,
                                   REAL8Vector *diag,
                                   REAL8Array *matrix,
                                   REAL8Vector *offDiag )

void LALDTriDiagonalToDiagonal2( LALStatus *stat,
                                   REAL8Vector *diag,
                                   REAL8Array *matrix,
                                   REAL8Vector *offDiag )
```
Description

These functions are called by the routines in Eigen.c to compute eigenvalues and eigenvectors of a symmetric square matrix *matrix. They are provided because, in some circumstances, users may find it useful to work with the partially-reduced results.

LALSSymmetricToTriDiagonal() and LALDSymmetricToTriDiagonal() reduce the symmetric square matrix *matrix to tridiagonal form. The vectors *diag and *offDiag must be allocated to the same length as each matrix dimension; on return, *diag stores the diagonal elements and *offDiag the off-diagonal elements (offDiag->data[0] is meaningless and will be set to zero). The tri-diagonalization is done in-place, so that on return *matrix will store the transformation matrix that brings the original input matrix into tridiagonal form. If you don’t want the input matrix to be changed, make a copy of it first.

LALSTriDiagonalToDiagonal() and LALDTriDiagonalToDiagonal() take a symmetric tridiagonal matrix and compute its eigenvalues and eigenvectors. On input, *diag stores the diagonal elements of the matrix, and *offDiag the off-diagonal elements, with offDiag->data[0] arbitrary; on return, *diag will store the eigenvalues and *offDiag will store zeroes. The matrix *matrix should store the orthogonal transformation matrix that brought the original symmetric matrix into tri-diagonal form (as returned by the above routines), or the identity matrix if the tri-diagonal matrix is the original matrix of interest; on return, *matrix will store the orthogonal transformation matrix that diagonalizes the original matrix: its columns are the eigenvectors of the original matrix.

Algorithm

The tri-diagonalizing routines follow the Householder reduction method described in Sec. 11.2 of [1]; they are essentially re-implementations of the Numerical Recipes routine tred2(). For large N, their operation count is approximately $4N^3/3$, or $2N^3/3$ for the routines that ignore eigenvectors. These routines explicitly enforce symmetry, by only using the lower-left triangle of the matrix as input.

The diagonalizing routines follow the QL algorithm with implicit shifts described in Sec. 11.3 of [1]; they are essentially re-implementations of the Numerical Recipes routine tqli(). Depending on the number of iterations required, their operation count is roughly $\sim 30N^2$, plus $\sim 3N^3$ if eigenvectors are also being computed.

The diagonalizing routines can fail if they fail to converge rapidly enough. The discussion in [1] does not go into much detail about when this is likely to occur, except to note that degenerate eigenvalues converge more slowly. If the routines fail in this way, diag, matrix, and offDiag will all be left in indeterminate states.

Uses

Notes
13.9.7 Program EigenTest.c

Computes the eigenvalues and eigenvectors of a matrix.

Usage


Description

This program computes the eigenvalues and eigenvectors of a symmetric real matrix using the routines in Eigen.c and EigenInternal.c. The following option flags are accepted:

- **n** Generates a random symmetric size×size metric. If this option is not given, -n 3 is assumed. This option (or its default) is overridden by the -i option, below.

- **i** Reads a matrix from an input file infile using the function LALSReadVector(). If the input file is specified as stdin, the data is read from standard input (not a file named stdin).

- **o** Writes the eigenvalues (and eigenvectors, if -v is specified below) to an output file outfile. If the output file is specified as stdout or stderr, the data is written to standard output or standard error (not to files named stdout or stderr). The eigenvalues are written as a single row of whitespace-separated numbers, and the eigenvectors as a square matrix where each column is the eigenvector of the corresponding eigenvalue. If this option is not specified, no output is written.

- **v** Specifies that eigenvectors are to be computed as well as eigenvalues.

- **t** Specifies that the computation is to be timed; timing information is written to stderr.

- **s** Specifies that the calculations are to be done to single-precision (REAL4) rather than double-precision (REAL8).

- **d** Sets the debug level to debuglevel. If not specified, level 0 is assumed.

Input format: If an input file or stream is specified, it should consist of N consecutive lines of N whitespace-separated numbers, that will be parsed using LALDReadVector(), or LALSReadVector() if the -s option was given. The data block may be preceded by blank or comment lines (lines containing no parseable numbers), but once a parseable number is found, the rest should follow in a contiguous block. If the lines contain different numbers of data columns, or if there are fewer lines than columns, then an error is returned; if there are more lines than columns, then the extra lines are ignored.

Output format: If an output file or stream is specified, the input matrix is first written as N consecutive lines of N whitespace-separated numbers. This will be followed with a blank line, then a single line of N whitespace-separated numbers representing the eigenvalues. If the -v option is specified, another blank line will be appended to the output, followed by N lines of N columns specifying the eigenvectors: the column under each eigenvalue is the corresponding eigenvector.

Exit codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NORM</td>
<td>0</td>
<td>&quot;Normal exit&quot;</td>
</tr>
<tr>
<td>SUB</td>
<td>1</td>
<td>&quot;Subroutine failed&quot;</td>
</tr>
<tr>
<td>ARG</td>
<td>2</td>
<td>&quot;Error parsing arguments&quot;</td>
</tr>
<tr>
<td>MEM</td>
<td>3</td>
<td>&quot;Out of memory&quot;</td>
</tr>
<tr>
<td>FILE</td>
<td>4</td>
<td>&quot;Could not open file&quot;</td>
</tr>
<tr>
<td>FMT</td>
<td>5</td>
<td>&quot;Bad input file format&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants EIGENTESTC_E<name>, and the status descriptions in EIGENTESTC_MSGE<name>. The source code with these messages is in EigenTest.c on line 1.106.
Algorithm

Uses

lalDebugLevel
LALPrintError() LALCheckMemoryLeaks()
LALSCreateVector() LALSDestroyVector()
LALDCreateVector() LALDDestroyVector()
LALSCreateArray() LALSDestroyArray()
LALDCreateArray() LALDDestroyArray()
LALSSymmetricEigenValues() LALSSymmetricEigenVectors()
LALDSymmetricEigenValues() LALDSymmetricEigenVectors()
LALCreateRandomParams() LALDestroyRandomParams()
LALUniformDeviate()

Notes
13.10  Header LALRunningMedian.h

Provides routines to efficiently calculate the running median

Synopsis

#include <lal/LALRunningMedian.h>

This header covers routines to efficiently calculate the running median of REAL4 and REAL8 sequences

Error conditions

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MALOC1</td>
<td>1</td>
<td>&quot;Could not allocate indexblock&quot;</td>
</tr>
<tr>
<td>MALOC2</td>
<td>2</td>
<td>&quot;Could not allocate checks&quot;</td>
</tr>
<tr>
<td>MALOC3</td>
<td>3</td>
<td>&quot;Could not allocate checks4shift&quot;</td>
</tr>
<tr>
<td>MALOC4</td>
<td>4</td>
<td>&quot;Could not allocate nodeaddresses&quot;</td>
</tr>
<tr>
<td>MALOC5</td>
<td>5</td>
<td>&quot;Could not allocate first node&quot;</td>
</tr>
<tr>
<td>MALOC6</td>
<td>6</td>
<td>&quot;Could not allocate node&quot;</td>
</tr>
<tr>
<td>CV</td>
<td>7</td>
<td>&quot;Could not create output vector (LALCreateVector failed)&quot;</td>
</tr>
<tr>
<td>NULL</td>
<td>8</td>
<td>&quot;Invalid input: NULL pointer.&quot;</td>
</tr>
<tr>
<td>ZERO</td>
<td>9</td>
<td>&quot;Invalid input: block length must be &gt;2&quot;</td>
</tr>
<tr>
<td>LARGE</td>
<td>10</td>
<td>&quot;Invalid input: block length larger than input length&quot;</td>
</tr>
<tr>
<td>IMED</td>
<td>11</td>
<td>&quot;Invalid input: wrong size of median array&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants LALRUNNINGMEDIANH_E<name>, and the status descriptions in LALRUNNINGMEDIANH_MSGE<name>. The source code with these messages is in LALRunningMedian.h on line 1.90.

Structures

This is the parameter structure for the LALRunningMedian functions. Currently the only parameter supported is the blocksize, the number of elements a single median is calculated from. The current implementation requires the blocksize to be < 2.

typedef struct tagLALRunningMedianPar
{
    UINT4 blocksize;
} LALRunningMedianPar;
13.10.1 Module LALRunningMedian.c

Functions to efficiently calculate running medians

Prototypes

```c
void LALDRunningMedian( LALStatus *status,
             REAL8Sequence *medians,
             const REAL8Sequence *input,
             LALRunningMedianPar param)
```

```c
void LALSRunningMedian( LALStatus *status,
             REAL4Sequence *medians,
             const REAL4Sequence *input,
             LALRunningMedianPar param)
```

```c
void LALDRunningMedian2( LALStatus *status,
             REAL8Sequence *medians,
             const REAL8Sequence *input,
             LALRunningMedianPar param)
```

```c
void LALSRunningMedian2( LALStatus *status,
             REAL4Sequence *medians,
             const REAL4Sequence *input,
             LALRunningMedianPar param)
```

Description

The routine `LALDRunningMedian()` calculates the running medians of a REAL8Sequence. The routine `LALSRunningMedian()` does the same for a REAL4Sequence. `input` is a REAL4/REAL8Sequence containing the input array, `blocksize` is the length of the block the medians are calculated of. With `n` being the length of the input array and `b` being the blocksize, the medians array must have a length of `(n-b+1)`. `LALDRunningMedian2()` and `LALSRunningMedian2()` are a different implementation of the same algorithm. It should behave exactly like `LALDRunningMedian()`, but has proven to be a little faster and more stable. Check if it works for you.

Algorithm

For a detailed description of the algorithm see the LIGO document T-030168-00-D, Somya D. Mohanty: Efficient Algorithm for computing a Running Median

Uses

- `LALCalloc`
- `LALFree`

Notes
13.10.2 Program **LALRunningMedianTest.c**

Program to test running median function

**Usage**

```
LALRunningMedianTest [length blocksize [lalDebugLevel]]
```

**Description**

This program tests the LALRunningMedian functions. First the proper function of the input checks is tested. Then it reads an array size and a block size from the command line, fills an array of the given size with random numbers, computes medians of all blocks with blocksize using the LALRunningMedian functions and compares the results against individually calculated medians. The test is repeated with blocksize - 1 (to check for even/odd errors). The default values for array length and window width are 1024 and 512. If a value for lalDebugLevel is given, the program outputs the values of the input and median arrays to files, using the PrintVector functions from the support package.

**Exit codes**

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOM</td>
<td>0</td>
<td>&quot;Nominal exit&quot;</td>
</tr>
<tr>
<td>ARG</td>
<td>1</td>
<td>&quot;Error parsing command-line arguments&quot;</td>
</tr>
<tr>
<td>SUB</td>
<td>2</td>
<td>&quot;Subroutine returned error&quot;</td>
</tr>
<tr>
<td>ALOC</td>
<td>3</td>
<td>&quot;Could not allocate data space&quot;</td>
</tr>
<tr>
<td>FALSE</td>
<td>4</td>
<td>&quot;Medians mismatch&quot;</td>
</tr>
<tr>
<td>ERR</td>
<td>5</td>
<td>&quot;Subroutine returned wrong or no error&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants **LALRUNNINGMEDIANTESTC_E<name>**, and the status descriptions in **LALRUNNINGMEDIANTESTC_MSGE<name>**. The source code with these messages is in **LALRunningMedianTest.c** on line 1100.

**Uses**

- `lalDebugLevel`
- `LALPrintError()`
- `LALSCreateVector()`
- `LALSDestroyVector()`
- `LALSPrintVector()`
- `LALDCreateVector()`
- `LALDDestroyVector()`
- `LALDPrintVector()`
- `LALSRunningMedian()`
- `LALDRunningMedian()`
- `LALCheckMemoryLeaks()`

**Notes**
13.11 Header RngMedBias.h

Synopsis

#include <lal/RngMedBias.h>

Error conditions
13.12 Module **RngMedBias.c**

Routine for finding bias in median for exponential distribution To be used with any code which uses the running median to estimate PSD.

For the exponential distribution with unit mean and variance, the value of the median is $\log(2.0)$ in the limit of infinite sample size. Thus, if we are using the running median to estimate the PSD, there is a correction factor of $\log(2.0)$. However, for finite sample sizes (i.e. for finite block size values), there is a bias in the estimator of the median and the correction factor is different. This program returns the correct normalization factor for block sizes from 1 to 1000. For larger values it returns $\log(2.0)$ and returns an error for smaller values.

**Prototypes**

```c
void LALRngMedBias (LALStatus *status,
                     REAL8 *biasFactor,
                     INT4 blkSize)
```

**Description**

Uses

LALHO()

**Notes**

Author: Krishnan, B., Itoh, Y.

$Id: RngMedBias.c,v 1.4 2007/06/08 14:41:59 bema Exp$
References


Chapter 14

Package window

This package defines two data types, the REAL4Window and REAL8Window types. These are suitable for storing window function data, providing storage for a sequence of samples as well as metadata about the window such as the sum-of-square of the samples. The package also provides a collection of functions to construct a variety of pre-defined window functions.

The use of window functions in signal analysis is well documented in many places. Definitions of most of the window functions can be found in Numerical Recipes\[1\] equations 13.4.13-13.4.15. Definitions of the remaining windows can be found in Spectral analysis for physical applications\[2\] Section 6.11. Definition of the Kaiser window can be found in Discrete-time Signal Processing by Oppenheim and Schafer, p.474.

Synopsis

#include <lal/Window.h>

Types

REAL4Window (structure)

- REAL4Sequence data The window function’s samples.
- REAL8 sumofsquares The sum of the squares of the window function’s samples.
- REAL8 sum The sum of the window function’s samples.

REAL8Window (structure)

- REAL8Sequence data The window functions’ samples.
- REAL8 sumofsquares The sum of the squares of the window function’s samples.
- REAL8 sum The sum of the window function’s samples.

Functions

REAL8Window *XLALCreateRectangularREAL8Window(UINT4 length)

REAL8Window *XLALCreateHannREAL8Window(UINT4 length)

REAL8Window *XLALCreateWelchREAL8Window(UINT4 length)

REAL8Window *XLALCreateBartlettREAL8Window(UINT4 length)

REAL8Window *XLALCreateParzenREAL8Window(UINT4 length)

REAL8Window *XLALCreatePapoulisREAL8Window(UINT4 length)

REAL8Window *XLALCreateHammingREAL8Window(UINT4 length)
These functions create or destroy a time-domain window function in a vector of specified length. If you wish to construct a custom window, call \texttt{XLALCreateRectangularREAL8Window()} (or the \texttt{REAL4} version), then replace the samples inside it with your own, and update the \texttt{sumofsquares} and \texttt{sum} elements. If the window function proves useful, consider adding it here so that others can benefit.

It is convenient to describe the windows as functions on the normalized domain $y \in [-1,1]$. The window is zero outside this domain. The window functions defined in this package are as follows.

**Rectangle**

\[
w(y) = 1. \quad (14.1)
\]

**Hann**

\[
w(y) = \cos^2 \frac{\pi}{2} y. \quad (14.2)
\]

**Welch**

\[
w(y) = 1 - y^2. \quad (14.3)
\]

**Bartlett**

\[
w(y) = 1 - |y|. \quad (14.4)
\]

**Parzen**

\[
w(y) = \begin{cases} 1 - 6y^2(1 - |y|) & |y| \leq 1/2, \\ 2(1 - |y|)^3 & |y| > 1/2. \end{cases} \quad (14.5)
\]

**Papoulis**

\[
w(y) = \frac{1}{\pi} \sin \pi |y| + (1 - |y|) \cos \pi |y|. \quad (14.6)
\]
Hamming

\[ w(y) = 0.08 + 0.92 \cos \frac{\pi}{2} y. \] (14.7)

This is the same as the Hann window, but with an additional DC bias, or “foot,” of 0.08.

Kaiser

\[ w(y) = I_0 \left( \beta \sqrt{1 - y^2} \right) / I_0(\beta), \] (14.8)

where \( I_0(x) \) is the 0th order, modified Bessel function of the first kind. The shape parameter \( \beta \in [0, \infty] \) sets the sharpness of the central peak. \( \beta = 0 \) yields the rectangle window, \( \beta \to \infty \) yields a \( \delta \) function with a single non-zero sample in the middle. This window is difficult to compute for large \( \beta \), and an asymptotic approximation is used for \( \beta \geq 705 \). A linearly-interpolated transition occurs between \( \beta = 695 \) and \( \beta = 705 \). Finite-difference derivatives of the window with respect to \( \beta \) are unlikely to work well in this regime.

Creighton

\[ w(y) = \exp \left( -\beta \frac{y^2}{1-y^2} \right). \] (14.9)

This window function is based on a fairly standard \( C_\infty \) test function used in distribution theory, e.g. *Green’s Functions and Boundary Value Problems* [4], by Stakgold. The shape parameter \( \beta \in [0, \infty] \) sets the sharpness of the central peak. \( \beta = 0 \) yields the rectangle window, \( \beta \to \infty \) yields a \( \delta \) function with a single non-zero sample in the middle.

Tukey

\[ w(y) = \begin{cases} \sin^2 \left( \frac{\pi}{2} (|y| - 1)/\beta \right) & |y| \geq 1 - \beta, \\ 1 & |y| < 1 - \beta. \end{cases} \] (14.10)

The shape parameter \( \beta \in [0, 1] \) sets what fraction of the window is spanned by the tapers. \( \beta = 0 \) yields the rectangle window, \( \beta = 1 \) yields the Hann window.

Gauss

\[ w(y) = \exp \left( -\frac{1}{2} \beta^2 y^2 \right). \] (14.11)

The shape parameter \( \beta \in [0, \infty] \) sets the sharpness of the central peak. \( \beta = 0 \) yields the rectangle window, \( \beta \to \infty \) yields a \( \delta \) function with a single non-zero sample in the middle.

These window functions are shown in Fig. 14.1.

For a vector of length \( L \) (an integer), the mapping from integer array index \( i \) to normalized co-ordinate \( y \) is

\[ y(i) = \begin{cases} 0 & L \leq 1, \\ 2i/(L-1) - 1 & L > 1, \end{cases} \] (14.12)

where \( 0 \leq i < L \), and floating-point division is used. This agrees with J. G. Proakis and D. G. Manolakis, *Digital Signal Processing* [3], and MatLab. The first sample is \( y = -1 \), the last sample is \( y = +1 \). For odd-lengthed vectors, the middle sample is \( y = 0 \), while for even-lengthed vectors \( y = 0 \) occurs half-way between the two middle samples. Substituting \( y(i) \) into the definitions of the window functions above yields \( w(i) \), the value of the window function at the integer sample \( i \).

The Fourier transforms of the windows are shown as functions of \( 1/y \) in Fig. 14.2. Since the Fourier transform of windowed data is the Fourier transform of the data convolved with the Fourier transform of the window, Fig. 14.2 is the major guideline for selecting a window. One can see that windows with a narrow central lobe tend to have higher sidelobes, and windows which suppress their low-order sidelobes tend to have more power in the high-order sidelobes. The choice of window thus depends on whether one is trying to resolve nearby spectral features of comparable magnitude (suggesting a rectangular or a Welch window), to reduce spectral bias and low-order sidelobes (a Hamming or Kaiser window), or to measure a broad spectrum with a large dynamical range (a Creighton or a Papoulis window).
Figure 14.1: Various windows as functions of the normalized independent variable $y$, choosing $\beta = 6$ for the Kaiser window, $\beta = 2$ for the Creighton window, $\beta = 0.5$ for the Tukey window, and $\beta = 3$ for the Gauss window.

References


Figure 14.2: Frequency behaviour of various windows as functions of the inverse of the normalized independent variable $y$, choosing $\beta = 6$ for the Kaiser window, $\beta = 2$ for the Creighton window, $\beta = 0.5$ for the Tukey window, and $\beta = 3$ for the Gauss window.
Section 6

Core Data Analysis Packages
Chapter 15

Package \texttt{inject}

This package provides routines to simulate gravitational waves and their effect on a detector. Conceptually, this can be divided into three stages:

1. Generating the gravitational waveform as produced by the source. The routines currently available are:
   - \texttt{GeneratePPNInspiral.h} Provides routines to generate parametrized post-Newtonian inspiral waveforms up to 5/2 order.
   - \texttt{GenerateTaylorCW.h} Provides a routine to generate continuous quasiperiodic waveforms with Taylor-parametrized frequency evolution.
   - \texttt{GenerateSpinOrbitCW.h} Provides a routine to generate Taylor-parameterized waveforms, as above, with additional binary orbit Doppler modulations.

2. Simulating a detector’s theoretical response to an incoming gravitational wave. The routines currently available are:
   - \texttt{SimulateCoherentGW.h} Provides routines to simulate the detector response to a coherent wave with slowly-varying frequency and amplitude.
   - \texttt{SimulateSB.h} Provides routines to simulate the response of a pair of detectors to a stochastic gravitational-wave background.

3. Injecting the detector’s theoretical response with time into a (noisy) datastream. This is done by a single set of generic routines in \texttt{Inject.h}.

As the package evolves, new source types may be added under item 1, and other (perhaps more generic) ways of simulating the detector response may be added under item 2. Item 3, however, is unlikely to need much updating.

In addition to these basic divisions, the package may include routines that perform other useful tasks in signal injection or source simulation, such as combining signal generation, detector simulation, and injection into a single function call, or modelling astrophysical distributions of sources.
15.1 Header Inject.h

Provides routines to inject a signal into detector output.

Synopsis

#include <lal/Inject.h>

This header provides simple routines to inject a signal, stored as a floating-point time series, into an integer time series that represents the ADC output of a detector channel.

The basic concept at work here is that of dithering. That is, to add a real signal \( x(t_k) \) to the integer output \( n(t_k) \) of an ADC, we cannot simply compute \( x + n \) at each time \( t_k \) and round to the nearest integer. To see this, consider injecting a sinusoid with an amplitude less than half the ADC resolution, \(|x| < 0.5\). Then adding \( x + n \) and rounding will always give back \( n \), and the signal injection will have no effect on the output.

Instead, what we would like to do is to add \( x \) to the ADC input before rounding occurs, and then round to an integer. That is, the ADC input was actually \( n + d \), where \( d \in [-0.5, 0.5] \); we want to compute the rounded value of \( n + d + x \), which may occasionally be different from \( n \). In principle, a Fourier transforms can detect a sinusoidal signal with an output much smaller than the ADC resolution: by integrating enough data, one can eventually detect a statistically significant phase correlation between the occasional increments and decrements in \( n \).

Of course given the output time series \( n(t_k) \) we can only guess at the input series \( n(t_k) + d(t_k) \) that produced it. The simplest guess is to assume that each \( d(t_k) \) is an independent random variable with a flat distribution over the range \([-0.5, 0.5] \). This is a reasonable guess to make if the root mean square variation between successive output values \( n \) is a few or more ADC counts; i.e. if the dimensionless power spectral density \( \sqrt{S(f)} \) has a value of a few or more around the sampling frequency \( f \). This is almost always true of any detector designed to work at or near its noise limit: the input to the ADC will first be whitened so that \( \sqrt{S(f)} \) is nearly flat, and then amplified so that \( \sqrt{S(f)} \) is on the order of several (or more) ADC counts.

In the routines covered by this header we will take it for granted that the above is a reasonable approximation, and will not check for it. We will further assume that the signal to be injected has already been subjected to the same whitening and amplification, so that the units of \( x(t_k) \) are normalized ADC counts (although it is still a real number, not an integer).

The dithering routines should be used whenever one is injecting a signal into a time series representing raw digitized data. In some data storage specifications, ADC output is not stored as an integer, but as a floating-point number representing an integer value. Such data must be cast to integers before being passed to the digitizing routines.

This header also provides even simpler routines for injecting a signal into floating-point data, without dithering. These should only be used when the data is genuinely continuous in character. This can include data derived by applying floating-point operations on ADC channels (e.g. digital filters, linear combinations of channels, etc.), but not data that simply represents ADC output in floating-point format. The assumption here is that the numerical post-processing of the ADC data completely masks any statistical signatures of the digitization.

Error conditions

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUL</td>
<td>1</td>
<td>&quot;Unexpected null pointer in arguments&quot;</td>
</tr>
<tr>
<td>BAD</td>
<td>2</td>
<td>&quot;A sampling interval is (effectively) zero&quot;</td>
</tr>
<tr>
<td>UNIT</td>
<td>3</td>
<td>&quot;Input or output is not in units of ADC counts&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants \texttt{INJECTH_E<name>}, and the status descriptions in \texttt{INJECTH_MSGE<name>}. The source code with these messages is in \texttt{Inject.h} on line l.121.

Types
15.1.1 Module InjectVector.c

Injects a vector of floating-point numbers into a vector of integers, with dithering.

Prototypes

```c
void
LALSI2InjectVector( LALStatus *stat,
        INT2Vector *output,
        REAL4Vector *signal,
        RandomParams *params )

void
LALSSInjectVector( LALStatus *stat,
        REAL4Vector *output,
        REAL4Vector *signal )
```

Description

The function `LALSI2InjectVector()` (i.e. “Single-precision to INT2”) dithers the contents of *output, adds the contents of *signal, and rounds to the nearest integer, storing the result back in *output. If desired, the random parameters for the dithering can be created outside this routine and passed in as *params (see Random.h); if this pointer is NULL, the parameters will be generated internally.

The function `LALSSInjectVector()` (i.e. “Single-precision to single-precision”) simply adds the contents of *signal to *output where they overlap, without performing any dithering.

Algorithm

Dithering is done with a flat random distribution as described in Inject.h. Injected values outside the dynamic range of the output force the output to its “rails” of $-2^{8N-1}$ or $2^{8N-1} - 1$, where $N$ is the number of bytes in the integer. The two vectors need not be of equal length; the injection stops when either vector reaches its end.

If params is NULL, a RandomParams structure will be generated internally using a seed of zero (i.e. the current time will be used to initialize the pseudorandom sequence).

Uses

- LALCreateRandomParams()
- LALDestroyRandomParams()
- LALUniformDeviate()

Notes

Author: Creighton, T. D.

$Id: InjectVector.c,v 1.6 2007/06/08 14:41:47 bema Exp$
15.1.2 Module **InjectTimeSeries.c**

Injects a time series of floating-point numbers into a time series of integers, with dithering.

### Prototypes

```c
void LALSI2InjectTimeSeries( LALStatus *stat,
        INT2TimeSeries *output,
        REAL4TimeSeries *signal,
        RandomParams *params )
```

```c
void LALSSInjectTimeSeries( LALStatus *stat,
        REAL4TimeSeries *output,
        REAL4TimeSeries *signal )
```

### Description

The function `LALSI2InjectTimeSeries()` (i.e. “Single-precision to INT2”) dithers each sample in `*output`, adds the nearest time sample from `*signal`, and rounds to the nearest integer, storing the result back in `*output`. If desired, the random parameters for the dithering can be created outside this routine and passed in as `*params` (see `Random.h`); if this pointer is `NULL`, the parameters will be generated internally.

The function `LALSSInjectVector()` (i.e. “Single-precision to single-precision”) simply takes each sample from `*output` and adds the nearest corresponding time sample from `*signal`, without performing any dithering.

### Algorithm

The algorithm is as given in `InjectVector.c`, with the following additional considerations. Since the two time series each carry their own information about epoch and sampling interval, the value to be injected at a given point in `*output` is found by taking the nearest time sample in `*signal`. Injection is only performed over the range in times that `*output` and `*signal` overlap; other values in `*output` are untouched.

Previous versions of this algorithm found the value to be injected by interpolating the two nearest samples in `*signal`, which reduces high-frequency aliasing noise and ensures that the pre- and post-injection signals agree in timing to within a fraction of a sample. However, this interpolation effectively convolved the signal with a triangular function of width $2\Delta t$, where $\Delta t$ is the sampling interval of the `signal`. This has the effect of a low-pass filter with an attenuation factor of $\sim 0.8$ at frequencies $\sim 1/4\Delta t$. Since input signals are typically sampled at or near their Nyquist frequencies, this would represent an unacceptable level of attenuation. For this reason, the current version of the algorithm eliminates the interpolation procedure.

### Uses

- `LALCreateRandomParams()`
- `LALDestroyRandomParams()`
- `LALUniformDeviate()`

### Notes

Author: Creighton, T. D. 

$Id: InjectTimeSeries.c,v 1.10 2007/06/08 14:41:47 bema Exp $
15.1.3 Program BasicInjectTest.c

Injects inspiral signals into detector noise.

Usage

BasicInjectTest [-s sourcefile] [-r respfile] [-o outfile] [-e seed]
[-i infile | -n sec nsec npt dt sigma] [-d debuglevel]

Description

This program generates inspiral waveforms with specified parameters, and injects them into ADC data. The following option flags are accepted:

- **s**: Reads source information from the file `sourcefile`. If absent, it injects a single $1.4M_\odot - 1.4M_\odot$ inspiral, optimally oriented, at a distance of $10^{-5}$ solar Schwarzschild radii ($0.00002GM_\odot/c^2$).

- **r**: Reads a detector response function from the file `respfile`. If absent, it generates raw dimensionless strain.

- **i**: Generates ADC input from the file `infile`. This takes precedence over the `-n` option, below.

- **n**: Generates random ADC input data starting from a GPS epoch of `sec` seconds plus `nsec` nanoseconds, with `npt` data sampled at `dt` second intervals, with white Gaussian noise having standard deviation `sigma`. If neither `-i` (above) nor `-n` are given, the program assumes `-n 0 0 1048576 9.765625e-4 0.0`.

- **o**: Writes injected ADC data to the file `outfile`. If absent, the routines are exercised, but no output is written.

- **d**: Sets the debug level to `debuglevel`. If not specified, level 0 is assumed.

- **r**: Sets the random number seed to `randomseed`. If not specified, the seed is generated from the current time.

**Format for sourcefile**: The source file consists of any number of lines of data, each specifying a chirp waveform. Each line must begin with a character code (CHAR equal to one of 'i', 'f', or 'c'), followed by 6 whitespace-delimited numerical fields: the GPS epoch of the chirp (INT8 nanoseconds), the two binary masses (REAL4 $M_\odot$), the distance to the source (REAL4 kpc), and the source’s inclination and phase at coalescence (REAL4 degrees). The character codes have the following meanings:

  - **i**: The epoch represents the GPS time of the start of the chirp waveform.
  - **f**: The epoch represents the GPS time of the end of the chirp waveform.
  - **c**: The epoch represents the GPS time when the binaries would coalesce in the point-mass approximation.

Thus a typical input line for two $1.4M_\odot$ objects at 11 000 kpc inclined $30^\circ$ with an initial phase of $45^\circ$, coalescing at 315 187 245 GPS seconds, will have the following line in the input file:

```
c 315187245000000000 1.4 1.4 11000.0 30.0 45.0
```

**Format for respfile**: The response function $R(f)$ gives the real and imaginary components of the transformation from ADC output $o$ to tidal strain $h$ via $h(f) = R(f)\tilde{o}(f)$. It is inverted internally to give the detector transfer function $T(f) = 1/R(f)$. The format `respfile` is a header specifying the GPS epoch $t_0$ at which the response was taken (INT8 nanoseconds), the lowest frequency $f_0$ at which the response is given (REAL8 Hz), and the frequency sampling interval $\Delta f$ (REAL8 Hz):

```
# epoch = t_0
# f0 = f0
# deltaF = \Delta f
```

followed by two columns of REAL4 data giving the real and imaginary components of $R(f_0 + k\Delta f)$.
Format for **infile**: The input file consists of a header giving the GPS epoch \( t_0 \) of the first time sample (INT8 nanoseconds) and the sampling interval \( \Delta t \) (REAL8 seconds):

\[
\begin{align*}
\# \text{ epoch} &= t_0 \\
\# \text{ deltaT} &= \Delta t
\end{align*}
\]

followed by a single column of ADC data. The ADC data should be integers in the range of an INT2 (from \(-32768 \) to \(32767 \)), but is assumed to be written in floating-point notation in accordance with frame format.

The output file **outfile** containing injected data is written in the same format.

Exit codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NORM</td>
<td>0</td>
<td>&quot;Normal exit&quot;</td>
</tr>
<tr>
<td>SUB</td>
<td>1</td>
<td>&quot;Subroutine failed&quot;</td>
</tr>
<tr>
<td>ARG</td>
<td>2</td>
<td>&quot;Error parsing arguments&quot;</td>
</tr>
<tr>
<td>VAL</td>
<td>3</td>
<td>&quot;Input argument out of valid range&quot;</td>
</tr>
<tr>
<td>FILE</td>
<td>4</td>
<td>&quot;Could not open file&quot;</td>
</tr>
<tr>
<td>INPUT</td>
<td>5</td>
<td>&quot;Error reading file&quot;</td>
</tr>
<tr>
<td>MEM</td>
<td>6</td>
<td>&quot;Out of memory&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants BASICINJECTTESTC_E<name>, and the status descriptions in BASICINJECTTESTC_MSGE<name>. The source code with these messages is in BasicInjectTest.c on line 1.150.

Algorithm

Uses

- lalDebugLevel
- LALPrintError()
- LALCheckMemoryLeaks()
- LALMalloc()
- LALCheckMemoryLeaks()
- LALCreateRandomParams()
- LALDestroyRandomParams()
- LALI2CreateVector()
- LALI2DestroyVector()
- LALSCreateVector()
- LALSDestroyVector()
- LALCCreateVector()
- LALCDestroyVector()
- LALSReadVectorSequence()
- LALSDestroyVectorSequence()
- LALCCVectorDivide()
- LALGeneratePPNInspiral()
- LALSimulateCoherentGW()
- LALSI2InjectTimeSeries()
- LALNormalDeviates()

Notes

Author: Creighton, T. D.

$Id: BasicInjectTest.c,v 1.12 2007/06/08 14:41:48 bema Exp$
15.1.4 Program InjectTest.c

Injects an inspiral signal into detector noise.

Usage


Description

This program generates Galactic inspiral waveform signals and injects them into ADC data. The following option flags are accepted:

- \texttt{-s} Reads source data from the file \texttt{sourcefile}, whose format is given below. If not specified, no injections are performed.

- \texttt{-e} Specifies ephemeris files giving the location of the Earth and Sun with respect to a barycentric reference point for arrival times, whose format is as required by \texttt{LALInitBarycenter()} in the \texttt{support} package. If not specified, the Earth’s centre is treated as the barycentric reference point (this is not consistent for long signal durations).

- \texttt{-r} Sets the random number seed to \texttt{randomseed}. If not specified, the seed is generated from the current time.

- \texttt{-d} Sets the debug level to \texttt{debuglevel}. If not specified, level 0 is assumed.

- \texttt{-h} Prints out the usage message and exits.

After parsing these recognized options, there must be one remaining argument: the name of an index file \texttt{indxfile} specifying the data input and output files, along with detector information associated with each file; the format is given below.

Format for \texttt{sourcefile}:

The source file consists of one or more lines, each representing a particular source to be injected. Each line consists of the name of a routine for generating a waveform, followed by a set of numerical arguments required for setting the parameters for that generator. The generators currently supported are as follows:

\begin{verbatim}
LALGeneratePPNInspiral tc m1 m2 d inc ra dec psi phic fi ff
LALGenerateTaylorCW t0 t1 t2 a1 a2 ra dec psi phi0 f0 [f1 [...]]
LALGenerateSpinOrbitCW t0 t1 t2 a1 a2 ra dec psi phi0 f0 [f1 [...]] arg udot rp e
\end{verbatim}

where \texttt{tc} is the time of coalescence, \texttt{t0} is a reference time where system properties are specified, \texttt{t1} and \texttt{t2} are start and stop times for the signal (all given as \texttt{INT8} GPS nanoseconds), \texttt{m1} and \texttt{m2} are the component masses of a binary system (\texttt{REAL4} solar masses), \texttt{d} is the distance to the system (\texttt{REAL4} Mpc), \texttt{inc} is the inclination of the system (\texttt{REAL4} degrees), \texttt{a1} and \texttt{a2} are intrinsic GW amplitudes of the + and \times polarizations (\texttt{REAL4} strain), \texttt{ra} and \texttt{dec} are the right ascension and declination of the system (\texttt{REAL8} degrees), \texttt{psi} is the polarization angle of the system (\texttt{REAL4} degrees), \texttt{phic} is the wave phase at coalescence (\texttt{REAL4} degrees), \texttt{phi0} is the wave phase at the reference time (\texttt{REAL8} degrees), \texttt{f0} is the wave frequency at the reference time (\texttt{REAL8} Hz), \texttt{f1}, \texttt{f2}, \ldots, \texttt{f_k}, \ldots are the (optional) frequency-normalized spindown coefficients (\texttt{REAL8} s⁻¹), \texttt{arg} is the argument of the source’s orbital periapsis (\texttt{REAL4} degrees), \texttt{udot} is the orbital angular speed of the source at periapsis (\texttt{REAL4} Hz), \texttt{rp} is the normalized projected orbital periapsis distance (\texttt{r_p}/c)\sin i of the source (\texttt{REAL4} s), and \texttt{e} is the orbital eccentricity of the source (\texttt{REAL4}).

Format for \texttt{indxfile}:

The index file consists of one or more lines, each representing a stretch of data to be injected. Each line consists of four strings: The site name, the response function file name, the input file name, and the output file name (which may be the same as the input). These are treated as follows:

The detector name may be one of \texttt{LHO}, \texttt{LLO}, \texttt{VIRGO}, \texttt{GEO600}, \texttt{TAMA300}, or \texttt{CIT40}. Additionally, a name of '-' represents a fictitious detector with a fixed location in the barycentric frame, and responding purely to the signal’s + polarization.
The response function file name should give the location of a readable file relative to the current execution directory, which will be read using the routine \texttt{LALReadFSeries()}; see \texttt{StreamInput.h} for discussion of the file format. It should have a \texttt{sampleUnits} field set to "\texttt{strain count}^{-1}\"; however, incorrect units will generate only a warning. A response file name of \texttt{-} specifies a unit response (i.e. raw strain is injected into the data); note that this means that actual response files named \texttt{-} are not permitted.

The input file name should give the location of a readable file relative to the current execution directory, which will be read using the routine \texttt{LALSReadTimeSeries()}; see \texttt{StreamInput.h} for discussion of the file format. It should have a \texttt{sampleUnits} field set to "\texttt{count}\", but incorrect units will generate only a warning. The input file name may be replaced with a \texttt{-} followed by 4 whitespace-delimited numerical tokens \texttt{- epoch npt dt sigma}, indicating that a time series will be generated starting at (\texttt{INT8}) epoch GPS nanoseconds, consisting of (\texttt{UINT4}) npt points sampled at (\texttt{REAL8}) dt second intervals having Gaussian random noise with rms value (\texttt{REAL4}) sigma ADC counts. Again, an actual input file named \texttt{-} is therefore not permitted.

The output file name should give the location relative to the current execution directory, where the injected data will be written using the routine \texttt{LALSWriteTSeries()}; see \texttt{StreamInput.h} for discussion of the file format. A name of \texttt{-} specifies writing to standard output (\texttt{not} to a file named \texttt{-}). To suppress output, specify \texttt{/dev/null} as the output file.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|p{10cm}|}
\hline
\textbf{<name>} & \textbf{code} & \textbf{description} \\
\hline
NORM & 0 & "Normal exit" \\
SUB & 1 & "Subroutine failed" \\
ARG & 2 & "Error parsing arguments" \\
VAL & 3 & "Input argument out of valid range" \\
FILE & 4 & "Could not open file" \\
MEM & 5 & "Out of memory" \\
\hline
\end{tabular}
\end{table}

The status codes in the table above are stored in the constants \texttt{INJECTTESTC_E<name>}, and the status descriptions in \texttt{INJECTTESTC_MSGE<name>}. The source code with these messages is in \texttt{InjectTest.c} on line 1.160.

\section*{Algorithm}

The program first reads \texttt{sourcefile} using \texttt{LALReadVector()}, parses its arguments with \texttt{LALCreateTokenList()}, and generates a linked list of CoherentGW structures containing the waveforms to be injected: see below for a discussion of how the various generators' input lines are parsed. Blank lines are ignored, as are comments (a \texttt{#} or \texttt{\%} character causes the remainder of the line to be ignored). The waveforms will be held in memory together for the rest of the injection procedure; if this taxes the available system memory, break up \texttt{sourcefile} and run the program on each separate file, using the output from each run as the input for the next.

Once all the waveforms have been generated, the program starts reading \texttt{indxfile}: for each stretch of data specified, it reads in the input file and response function file using \texttt{LALSReadTSeries()} and \texttt{LALSReadFSeries()}, inverts the response function to get a transfer function, simulates and injects the waveforms using \texttt{LALSimulateCoherentGW()} and \texttt{LALSSInjectTimeSeries()}, and writes the resulting time series to the output file using \texttt{LALSWriteTSeries()}. If a given stretch of data has the same response function file name as the previous one, then the previous transfer function is reused rather than re-reading and re-inverting the data; thus, one should naturally arrange \texttt{indxfile} so that data stretches using the same response function are listed consecutively.

\texttt{LALGeneratePPNInspiral}: Most arguments are parsed from the corresponding tokens using the functions in \texttt{StringConvert.c}. However, two input parameters require some nontrivial computation. First, the generator requires a start time, while \texttt{sourcefile} specifies a coalescence time. The solution is to generate a waveform with an arbitrary start time (in this case zero GPS seconds), take the returned coalescence time, and adjust the start time accordingly. Second, \texttt{sourcefile} does not specify a sampling interval to be used in the generated waveform. Here the solution is to estimate an appropriate interval from the requested termination frequency or from the point of post-Newtonian breakdown for the specified masses. If a nonzero termination frequency \(f\) is specified (either positive or negative), then the maximum rate of change of frequency in the \texttt{(post\textsuperscript{0}.)Newtonian approximation is} \(f_{\text{max}} = 1.8\pi^{8/3}\tau^{5/3}f^{11/3}\), where \(\tau = \frac{Gm_{\text{tot}}}{c^3}\) is the relativistic
minimum timescale of the system. As explained in \texttt{GeneratePPNInspiral.c}, for linear interpolation of the waveform to be accurate to within $\pi/2$ radians, we would like $\Delta f/\Delta t \lesssim 2$ over any sampling interval. This implies that $\Delta t \approx \sqrt{2/\dot{f}_{\text{max}}}$, or:

$$\Delta t \approx 0.1403 \tau^{-5/6} f^{-11/6}.$$  

If the maximum frequency is given as zero (i.e. unspecified) or positive, then an additional constraint is imposed by the guaranteed post-Newtonian breakdown at or before $r = 2Gm_{\text{tot}}/c^2$, giving $\dot{f}_{\text{max}} = (2\sqrt{2}/40\pi)\tau^{-2}$. This implies that:

$$\Delta t_{\text{min}} \approx 7.697 \tau,$$

and we can write the previous expression as $\Delta t = \max\{\Delta t_{\text{min}}, 0.7685(\Delta t_{\text{min}})^{-5/6} f^{-11/6}\}$. When the waveform is actually generated, the maximum value of $\Delta f/\Delta t$ is returned, and, if greater than 2, $\Delta t$ is reduced accordingly and the waveform regenerated.

\texttt{LALGenerateTaylorCW}: Most arguments are parsed from the corresponding tokens using the routines in \texttt{StringConvert.c}. However, two input parameters require computation. First, the base frequency $f_0$ and spindown terms $f_1, \ldots, f_N$ need to be transformed from the reference time to the start time, by the following formula:

$$f_{0(\text{start})} = f_0 \left(1 + \sum_{k=1}^{N} f_k t^k\right),$$

$$f_{j(\text{start})} = \frac{\sum_{k=j}^{N} f_k t^k}{1 + \sum_{k=1}^{N} f_k t^k},$$

where $t = t_{\text{start}} - t_{\text{ref}}$ is the time shift. These calculations are done to double precision; even so, one is advised to specify the frequency and spindown terms at a reference time that is not too far from the times being considered. Roughly speaking, for phases to be accurate within a fraction of a cycle, this means that $1 + \sum_k |f_k t^k| \ll 10^{15}/|f_0| \tau$, where $\tau$ is the larger of $t_{\text{start}} - t_{\text{ref}}$ and $t_{\text{stop}} - t_{\text{ref}}$.

Second, the program must choose an appropriate sampling interval for the waveforms, such that the maximum $\Delta f/\Delta t$ over any interval is less than 2. This is fairly straightforward; one simply takes:

$$\Delta t \approx \left(0.5 f_0 \sum_{k=1}^{N} k |f_k T^{k-1}|\right)^{-1/2},$$

where $T = t_{\text{stop}} - t_{\text{start}}$ is the total signal length. If this gives $\Delta t > T$ (as for instance when there are no nonzero spindown terms), then $\Delta t$ is set equal to $T$.

\texttt{LALGenerateSpinOrbitCW}: Most arguments are parsed from the corresponding tokens using the routines in \texttt{StringConvert.c}. The reference time is assumed to correspond to the orbital epoch of the system, and the conversion from reference time to start time discussed above is performed automatically within the \texttt{LALGenerateSpinOrbitCW()} function. However, the program still needs to choose an appropriate sampling interval for the waveforms, such that the maximum $\Delta f/\Delta t$ over any interval is less than 2. We simply take the formula for $\Delta t$ above and add to the spindown terms a Doppler modulation at least as large as the maximum possible orbital acceleration, to get:

$$\Delta t \approx \left(0.5 f_0 \left[\dot{v}_p^2 r_p \sin i/c + \sum_{k=1}^{N} k |f_k T^{k-1}|\right]\right)^{-1/2},$$

where $\dot{v}_p$ is the angular speed at periapsis, $r_p \sin i/c$ is the projected, normalized periapsis distance, and $T = t_{\text{stop}} - t_{\text{start}}$ is the total signal length. If this gives $\Delta t > T$ then $\Delta t$ is set equal to $T$.

\textbf{Uses}

- \texttt{lalDebugLevel}
- \texttt{LALMalloc()}
- \texttt{LALCreateRandomParams()}
- \texttt{LALMalloc()}
- \texttt{LALFree()}
- \texttt{LALCreateRandomParams()}
- \texttt{LALDestroyRandomParams()}
- \texttt{LALCreateTokenList()}
- \texttt{LALDestroyTokenList()}
LALCHARCreateVector() LALCHARDestroyVector()
LALCCreateVector() LALCDestroyVector()
LALDCreateVector() LALDDestroyVector()
LALSDestroyVector() LALSDestroyVectorSequence()
LALCreateFSeries() LALSReadTSeries()
LALCHARReadVector() LALSWriteTSeries()
LALCCVectorDivide() LALNormalDeviates()
LALUnitRaise() LALUnitMultiply()
LALUnitCompare() LALStringToI8()
LALStringToS() LALStringToD()
LALGeneratePPNInspiral() LALGenerateTaylorCW()
LALGenerateSpinOrbitCW() LALSimulateCoherentGW()
LALSSInjectTimeSeries() LALInitBarycenter()
LALSnprintf() LALPrintError()
LALCheckMemoryLeaks()
15.2 Header SimulateCoherentGW.h

Provides routines to simulate generic gravitational waveforms originating from a particular source.

Synopsis

#include <lal/SimulateCoherentGW.h>

This header covers generic routines and structures to represent and simulate the effects of a plane gravitational wave propagating from a distinct point on the sky.

Any plane gravitational wave is specified by a direction \( \hat{n} \) to its apparent source (i.e. opposite to its direction of propagation), and by the instantaneous values \( h_+(t) \), \( h_\times(t) \) of its plus and cross polarizations as functions of (retarded) time \( t = t_0 + \hat{n} \cdot (x - x_0) \), where \( t_0 \) is the time measured at some local reference point \( x_0 \), and \( t \) is the time measured by a synchronized clock at \( x \). We adopt the standard meaning of the instantaneous strain amplitudes \( h_+, h_\times \): in some reference transverse \( x-y \) coordinate system oriented such that \( \hat{x} \times \hat{y} = -\hat{n} \) points in the direction of propagation, two free observers originally separated by a displacement \( (x, y) \) will experience an additional tidal displacement \( \delta x = (x h_+ + y h_\times)/2 \), \( \delta y = (x h_\times - y h_+)/2 \).

Quasiperiodic waves: Most astrophysical sources of gravitational radiation are described as quasiperiodic (or, less accurately, as “adiabatic”), in that they can be said to have an instantaneous frequency, amplitude, and polarization, all of which vary on timescales much longer than a wave period. Mathematically we write this as:

\[
\begin{align*}
h_+(t) &= A_1(t) \cos \Phi(t) + A_2(t) \sin \Phi(t) \\
h_\times(t) &= A_1(t) \sin \Phi(t) + A_2(t) \cos \Phi(t)
\end{align*}
\]

where \( \phi(t) = 2\pi \int f(t) \, dt \), and the evolution timescale \( \tau = \min(A/\dot{A}, B/\dot{B}, f/f) \) is much greater than \( h/\dot{h} \sim 1/f \). Obviously it is mathematically impossible for the physical functions \( h_+, h_\times \) to specify uniquely more than two other functions of time; we rely on the notion of quasiperiodicity to define “natural” choices of instantaneous frequency and amplitude. The ambiguity in this choice is on the order of the amount that these quantities change over a cycle.

While the above formula appears to have five degrees of freedom (two quadrature amplitudes \( A \) and \( B \) for each polarization, plus a common phase function \( \phi \) ), there is a degeneracy between the two quadrature amplitudes and a shift in phase. One could simply treat each polarization independently and represent the system with two amplitude functions \( A_+, A_\times \) and two phase functions \( \phi_+, \phi_\times \), but we would like to preserve the notion that the phases of the two waveforms derive from a single underlying instantaneous frequency. We therefore write the waveforms in terms of two polarization amplitudes \( A_1(t) \) and \( A_2(t) \), a single phase function \( \phi(t) \), and a polarization shift \( \Phi(t) \):

\[
\begin{align*}
h_+(t) &= A_1(t) \cos \Phi(t) \cos \phi(t) - A_2(t) \sin \Phi(t) \sin \phi(t) \\
h_\times(t) &= A_1(t) \sin \Phi(t) \cos \phi(t) + A_2(t) \cos \Phi(t) \sin \phi(t)
\end{align*}
\]

The physical meaning of these functions is shown in Fig. 15.1. There is a close relationship between the polarization shift \( \Phi \) and the orientation of the \( x-y \) coordinates used to define our polarization basis: if we rotate the \( x \) and \( y \) axes by an angle \( \Delta \psi \), we change \( \Phi \) by an amount \(-2\Delta \psi \). (The factor of 2 comes from the fact that the + and \( \times \) modes are quadrupolar: a + mode rotated 45° is a \( \times \) mode, and a mode rotated 90° is the opposite of itself.) We use the polarization angle \( \psi \) to define the orientation of the \( x \)-axis of the polarization basis relative to an Earth-fixed reference frame (see the coordinate conventions below). If \( \Phi \) is constant, one can redefine \( \psi \) such that \( \Phi = 0 \); however, when \( \Phi \) changes with time, we would nonetheless like our polarization basis to remain fixed. We therefore retain the constant \( \psi \) and the function \( \Phi(t) \) as distinct quantities.

Figure 15.1: Polarization phase diagram for a quasiperiodic gravitational wave. The phase point \( p(t) \) traces out the indicated ellipse in the \( h_+, h_\times \) plane; the parameters \( A_1, A_2, \) and \( \Phi \) remain roughly constant over many cycles in \( \phi \).
The advantage of this quasiperiodic representation of a gravitational wave is that a physical sampling of the parameters $A_1, A_2, \phi, \text{ and } \Phi$ need only be done on timescales \( \Delta t \lesssim \tau \), whereas the actual wave functions \( h_{+x} \) need to be sampled on timescales \( \Delta t \gtrsim 1/f \).

The following coordinate conventions are assumed:

1. Fig. 7 of [2] defines standard coordinate conventions for nonprecessing binaries, and by extension, for any fixed-axis rotating source: If \( \hat{z} \) points in the direction of wave propagation (away from the source), and \( \hat{I} \) points in the (constant) direction of the source’s angular momentum vector, then the \( x-y \) coordinates used to define the + and \( \times \) polarizations are given by \( \hat{x} = |\csc i| \hat{z} \times \hat{I} \) and \( \hat{y} = \hat{z} \times \hat{x} \), where \( i = \arccos(\hat{z} \cdot \hat{I}) \) is the inclination angle between \( \hat{I} \) and \( \hat{z} \). Such a system will generically have \( A_1(t) = A(t)(1+\cos^2 i) \), \( A_2(t) = 2A(t) \cos i \), \( \Phi(t) = 0 \), and \( f(t) > 0 \) (i.e. \( \phi(t) \) increasing with time). For precessing systems, prescriptions for \( \hat{x} \) and \( \hat{y} \) become ambiguous, but they must be fixed; the relations for \( A_1, A_2, \) and \( \Phi \) will no longer be maintained.

2. Appendix B of [3] defines a convention for the overall polarization angle \( \psi \): Let \( \hat{N} \) be the direction of the Earth’s north celestial pole, and define the direction of the ascending node \( \hat{\Omega} = |\csc \alpha| \hat{N} \times \hat{z} \), where \( \alpha \) is the right ascension of the source. Then \( \psi \) is the angle, right-handed about \( \hat{z} \), from \( \hat{\Omega} \) to \( \hat{x} \).

3. The direction of propagation of the wave is defined by the right ascension \( \alpha \) and declination \( \delta \) of the source, as seen from the point of measurement. See SkyCoordinates.h for a definition of these quantities. We expect that these will be effectively constant for almost any gravitational wave source of interest.

The polarization response: The relative strain induced in the test masses of a detector by a passing gravitational wave depends not only on the amplitudes \( h_{+x} \) of the gravitational wave, but also on the design of the detector and its orientation with relative to the \( x-y \) coordinate system used to define the + and \( \times \) polarizations. For a given detector, the response to each polarization thus depends on the right ascension \( \alpha \), declination \( \delta \), and polarization angle \( \psi \) of the source (which define the orientation of the + and \( \times \) polarization axes relative to the Earth), and on the time \( t \) (which determines the orientation of the detector as the Earth rotates). The strain \( h(t) \) induced in the detector is thus given by two polarization response functions \( F_{+x}(\alpha, \delta, \psi; t) \) by:

\[
 h(t) = h_+(t)F_+(\alpha, \delta, \psi; t) + h_\times(t)F_\times(\alpha, \delta, \psi; t) .
\]

We will not discuss the computation of these functions \( F_{+x} \), as these are covered under the header DetResponse.h.

The transfer function: All gravitational wave detectors incorporate a set of analog and digital filters that convert a gravitational excitation on the test masses into a measurable output time series. The effects of these functions are aggregated into a complex-valued transfer function \( T(f) \), which gives the instrumental response (in units of “counts” from an analog→digital converter) to gravitational waves of unit amplitude at the frequency \( f \). Specifically, if the strain exerted on the antenna is given by \( h(t) = \text{Re} [\mathcal{H}_e^{2\pi ift}] \) (where the complex amplitude \( \mathcal{H} \) includes the phase of the wave), then the ADC output of the instrument is given by:

\[
 o(t) = \text{Re} \left[ T(f) \mathcal{H}_e^{2\pi ift} \right] .
\]

The transfer function has a strong frequency dependence in order to “whiten” the highly-coloured instrumental noise, and thus preserve instrumental sensitivity across a broad band of frequencies.

We note that although the transfer function measures the response of the instrument to a gravitational wave, the term response function refers to inverse transformation of taking an instrumental response and computing a gravitational waveform; that is, \( R(f) = 1/T(f) \). This confusing bit of nomenclature arises from the fact that most data analysis deals with extracting gravitational waveforms from the instrumental output, rather than injecting waveforms into the output.

For quasiperiodic waveforms with a well-defined instantaneous frequency \( f(t) \) and phase \( \phi(t) \), we can compute the response of the instrument entirely in the time domain in the adiabatic limit: if our instrumental excitation is a linear superposition of waveforms \( h(t) = \text{Re} [\mathcal{H}(t)e^{i\phi(t)}] \), then the output is a superposition of waves of the form

\[
 o(t) \approx \text{Re} \left[ T\{f(t)\} \mathcal{H}(t) e^{i\phi(t)} \right] .
\]

This expression is approximate to the extent that \( T(f) \) varies over the range \( f \pm 1/\tau \), where \( \tau \) is the evolution timescale of \( \mathcal{H}(t) \) and \( f(t) \). Since the transfer function and polarization response (above) are linear operators, we can apply them in either order.
A note on terminology: We use the word “coherent” in the name of this header in the loosest possible sense, referring to any wave with a well-defined direction of propagation, whose wave amplitudes $h_{+,\times}$ are deterministic functions of retarded time. Given a knowledge of these parameters, such a waveform is amenable to “coherent” detection in a network of detectors, through time-shifted matched filtering.

However, coherence is often used to refer to a more restricted class of waveforms that are “effectively monochromatic” over some coherence timescale $t_{\text{coh}}$; i.e. in any timespan $t_{\text{coh}}$ there is a fixed-frequency sinusoid that is never more than $90^\circ$ out of phase with the waveform. This is more restrictive even than our concept of quasiperiodic waves; for smoothly-varying waveforms one has $t_{\text{coh}} \sim \dot{f}^{-1/2}$, which is much shorter than the evolution timescale $\tau \sim f/\dot{f}$ (provided $\tau \gg 1/f$, as we have assumed).

Error conditions

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUL</td>
<td>1</td>
<td>&quot;Unexpected null pointer in arguments&quot;</td>
</tr>
<tr>
<td>BAD</td>
<td>2</td>
<td>&quot;A sampling interval is (effectively) zero&quot;</td>
</tr>
<tr>
<td>SIG</td>
<td>3</td>
<td>&quot;Input signal must specify amplitude and phase functions&quot;</td>
</tr>
<tr>
<td>DIM</td>
<td>4</td>
<td>&quot;Amplitude must be a 2-dimensional vector&quot;</td>
</tr>
<tr>
<td>MEM</td>
<td>5</td>
<td>&quot;Memory allocation error&quot;</td>
</tr>
<tr>
<td>UNIT</td>
<td>6</td>
<td>&quot;Bad input units&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants \texttt{SIMULATECOHERENTGW\_E<name>}, and the status descriptions in \texttt{SIMULATECOHERENTGW\_MSGE<name>}. The source code with these messages is in \texttt{SimulateCoherentGW.h} on line L284.

Types

Structure \texttt{CoherentGW}

This structure stores a representation of a plane gravitational wave propagating from a particular point on the sky. Several alternate representations are permitted to allow a more natural characterization of quasiperiodic waveforms. The fields are:

\texttt{SkyPosition position} The location of the source in the sky. This should be in equatorial celestial coordinates, but routines may be able to do the conversion.

\texttt{REAL4 psi} The polarization angle $\psi$, in radians, as defined in Appendix B of [3].

\texttt{REAL4TimeVectorSeries *h} A time-sampled two-dimensional vector storing the waveforms $h_+ (t)$ and $h_{\times} (t)$, in dimensionless strain.

\texttt{REAL4TimeVectorSeries *a} A time-sampled two-dimensional vector storing the amplitudes $A_1 (t)$ and $A_2 (t)$, in dimensionless strain.

\texttt{REAL4TimeSeries *f} A time-sampled sequence storing the instantaneous frequency $f(t)$, in Hz.

\texttt{REAL8TimeSeries *phi} A time-sampled sequence storing the phase function $\phi(t)$, in radians.

\texttt{REAL4TimeSeries *shift} A time-sampled sequence storing the polarization shift $\Phi(t)$, in radians.

It is permissible to set only some of the \texttt{REAL4TimeSeries} or \texttt{REAL4TimeVectorSeries} fields above, but the waveform is treated as being zero except during those times when either \texttt{h}, or both \texttt{a} and \texttt{phi}, are defined. Where \texttt{shift} is not specified, it is assumed that $\Phi$ is zero; where \texttt{f} is not specified but \texttt{phi} is, $f(t)$ can be computed as $\dot{\phi}(t)/2\pi$. Where \texttt{f} and \texttt{phi} overlap, or where \texttt{h} and any other time series overlap, they must be defined consistently.

Structure \texttt{DetectorResponse}

This structure contains information required to determine the response of a detector to a gravitational waveform. The fields are:

\texttt{COMPLEX8FrequencySeries *transfer} The frequency-dependent transfer function of the interferometer, in ADC counts per unit strain amplitude at any given frequency. If absent, the response will be given in raw strain rather than ADC output.
LALDetector *site A structure storing site and polarization information, used to compute the polarization response and the propagation delay. If absent, the response will be computed to the plus mode waveform with no time delay.

EphemerisData *ephemerides A structure storing the positions, velocities, and accelerations of the Earth and Sun centres of mass, used to compute the propagation delay to the solar system barycentre. If absent, the propagation delay will be computed to the Earth centre (rather than a true barycentre).

LIGOTimeGPS heterodyneEpoch A reference time for heterodyned detector output time series, where the phase of the mixing signal is zero. This parameter is only used when generating detector output time series with nonzero heterodyne frequency $f_0$. (Note: This should really be a parameter stored in the TimeSeries structure along with $f_0$, but it isn’t, so we have to add it here.)
15.2.1 Module SimulateCoherentGW.c

Computes the response of a detector to a coherent gravitational wave.

Prototypes

```c
void LALSimulateCoherentGW( LALStatus *stat,
REAL4TimeSeries *output,
CoherentGW *signal,
DetectorResponse *detector )
```

Description

This function takes a quasiperiodic gravitational waveform given in \*signal, and estimates the corresponding response of the detector whose position, orientation, and transfer function are specified in \*detector. The result is stored in \*output.

The fields output->epoch, output->deltaT, and output->data must already be set, in order to specify the time period and sampling rate for which the response is required. If output->f0 is nonzero, idealized heterodyning is performed (an amount $2\pi f_0(t - t_0)$ is subtracted from the phase before computing the sinusoid, where $t_0$ is the heterodyning epoch defined in detector). For the input signal, signal->h is ignored, and the signal is treated as zero at any time for which either signal->a or signal->phi is not defined.

This routine will convert signal->position to equatorial coordinates, if necessary.

Algorithm

The routine first accounts for the time delay between the detector and the solar system barycentre, based on the detector position information stored in \*detector and the propagation direction specified in \*signal. Values of the propagation delay are precomputed at fixed intervals and stored in a table, with the intervals $\Delta T_{\text{delay}}$ chosen such that the value interpolated from adjacent table entries will never differ from the true value by more than some timing error $\sigma_T$. This implies that:

$$\Delta T_{\text{delay}} \leq \sqrt{\frac{8\sigma_T}{\max\{a/c\}}},$$

where $\max\{a/c\} = 1.32 \times 10^{-10} \text{s}^{-1}$ is the maximum acceleration of an Earth-based detector in the barycentric frame. The total propagation delay also includes Einstein and Shapiro delay, but these are more slowly varying and thus do not constrain the table spacing. At present, a 400s table spacing is hardwired into the code, implying $\sigma_T \approx 3 \mu\text{s}$, comparable to the stated accuracy of LALBarycenter().

Next, the polarization response functions of the detector $F_{+\times}(\alpha, \delta)$ are computed for every 10 minutes of the signal’s duration, using the position of the source in \*signal, the detector information in \*detector, and the function LALComputeDetAMResponseSeries(). Subsequently, the polarization functions are estimated for each output sample by interpolating these precomputed values. This guarantees that the interpolated value is accurate to $\sim 0.1\%$.

Next, the frequency response of the detector is estimated in the quasiperiodic limit as follows:

- At each sample point in \*output, the propagation delay is computed and added to the sample time, and the instantaneous amplitudes $A_1, A_2$, frequency $f$, phase $\phi$, and polarization shift $\Phi$ are found by interpolating the nearest values in signal->a, signal->f, signal->phi, and signal->shift, respectively. If signal->f is not defined at that point in time, then $f$ is estimated by differencing the two nearest values of $\phi$, as $f \approx \Delta \phi/2\pi \Delta t$. If signal->shift is not defined, then $\Phi$ is treated as zero.

- The complex transfer function of the detector the frequency $f$ is found by interpolating detector->transfer. The amplitude of the transfer function is multiplied with $A_1$ and $A_2$, and the phase of the transfer function is added to $\phi$.

- The plus and cross contributions $o_+, o_\times$ to the detector output are computed as in Eqs. 15.1 and 15.2 of SimulateCoherentGW.h, but using the response-adjusted amplitudes and phase.

- The final detector response $o$ is computed as $o = (o_+ F_+) + (o_\times F_\times)$.
A note on interpolation: Much of the computational work in this routine involves interpolating various time series to find their values at specific output times. The algorithm is summarized below.

Let \( A_j = A(t_A + j \Delta t_A) \) be a sampled time series, which we want to resample at new (output) time intervals \( t_k = t_0 + k \Delta t \). We first precompute the following quantities:

\[
\begin{align*}
t_{\text{off}} &= \frac{t_0 - t_A}{\Delta t_A}, \\
dt &= \frac{\Delta t}{\Delta t_A}.
\end{align*}
\]

Then, for each output sample time \( t_k \), we compute:

\[
\begin{align*}
t &= t_{\text{off}} + k \times dt, \\
j &= \lfloor t \rfloor, \\
f &= t - j,
\end{align*}
\]

where \( \lfloor x \rfloor \) is the “floor” function; i.e. the largest integer \( \leq x \). The time series sampled at the new time is then:

\[
A(t_k) = f \times A_{j+1} + (1 - f) \times A_j.
\]

Uses

- LALWarning()
- LALInfo()
- LALSCreateVector()
- LALSDestroyVector()
- LALDCreateVector()
- LALDDestroyVector()
- LALConvertSkyCoordinates()
- LALGeocentricToGeodetic()
- LALCVectorAbs()
- LALCVectorAngle()
- LALUnwrapREAL4Angle()

Notes

The major computational hit in this routine comes from computing the sine and cosine of the phase angle in Eqs. 15.1 and 15.2 of SimulateCoherentGW.h. For better online performance, these can be replaced by other (approximate) trig functions. Presently the code uses the native \texttt{libm} functions by default, or the function \texttt{sincosp()} in \texttt{libsunmath} if this function is available and the constant \texttt{ONLINE} is defined. Differences at the level of 0.01 begin to appear only for phase arguments greater than \( 10^{14} \) or so (corresponding to over 500 years between phase epoch and observation time for frequencies of around 1kHz).

To activate this feature, be sure that \texttt{sunmath.h} and \texttt{libsunmath} are on your system, and add \texttt{--with-extra-cppflags ONLINE} to the \texttt{--with-extra-cppflags} configuration argument. In future this flag may be used to turn on other efficient trig algorithms on other (non-Solaris) platforms.
15.3  Header SimulateInspiral.h

Provides a routine to inject inspirals into time series data.

Synopsis

#include <lal/SimulateInspiral.h>

The routines in GeneratePPNInspiral.h, SimulateCoherentGW.h, and Inject.h provide a powerful mechanism for simulating the instrumental response to a physical inspiral event, including such considerations as the polarization response and propagation delay for a particular instrument, which are necessary if one wants to model coincident detection in a network of detectors. In many cases, though, one simply wants to generate a signal with a given signal-to-noise ratio and a given coalescence time, and inject it into a time series. This header provides a streamlined interface to accomplish this with a minimum of fuss to the user.

In order to provide this streamlined interface, two calculations have to be internalized. First, the waveform must be time-shifted so that it coalesces at the specified time. This is straightforward and requires no explanation. Second, the waveform must be scaled to have some specified amplitude. To do this, we must first state what we mean by “amplitude”.

We define the characteristic detection amplitude $A_c$ of a gravitational-wave signal to be its root summed squared contribution to the sampled detector output. That is, if the detector output can be written as $o(t_k) = n(t_k) + s(t_k)$, where $n$ is the contribution due to noise, $s$ is the contribution due to signal, and $t_k = k\Delta t$ are the discrete time samples, then:

$$A_c = \sqrt{\sum_{k=-\infty}^{\infty} |s(t_k)|^2}.$$  

(15.3)

If $T(f)$ is the detector transfer function (such that a gravitational wave signal $\hat{h}(f)$ in the frequency domain produces an output $\hat{a}(f) = \hat{n}(f) + T(f)\hat{h}(f)$), the characteristic detection amplitude has the not-so-obvious relation that:

$$A^2_c = \int_{-\infty}^{\infty} df \frac{|T(f)\hat{h}(f)|^2}{S_o}.$$  

(15.4)

So why use this quantity to specify the signal amplitude? First, it is easy for a simulation/injection routine to calculate. Second, if we assume that $T(f)$ is a true whitening filter such that the output power spectral density is flat $S_o(f) = S_o =$constant, then the sampled noise output is uncorrelated noise with a mean of zero and a variance of $\sigma_n^2 = S_o/2\Delta t$. The intrinsic signal-to-noise power is then given by the simple relation:

$$(h|h) = 2 \int_{-\infty}^{\infty} df \frac{|\hat{h}(f)|^2}{S_h(f)}$$

$$= 2 \int_{-\infty}^{\infty} df \frac{|T(f)\hat{h}(f)|^2}{S_o}$$

$$= \frac{A^2_c}{\sigma_n^2}.$$

(15.5)

Thus to simulate a signal with an intrinsic signal-to-noise amplitude $\sqrt{(h|h)}$, simply fill a data vector with uncorrelated noise with variance $\sigma_n^2$ and inject a signal with a characteristic detection amplitude of $A_c = \sigma_n\sqrt{(h|h)}$.

We refer the reader to the Conventions section at the end of this header documentation for a more detailed derivation of these specifications.

Error conditions

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUL</td>
<td>1</td>
<td>&quot;Unexpected null pointer in arguments&quot;</td>
</tr>
<tr>
<td>MEM</td>
<td>2</td>
<td>&quot;Memory allocation error&quot;</td>
</tr>
<tr>
<td>DF</td>
<td>3</td>
<td>&quot;Transfer frequency interval is zero&quot;</td>
</tr>
<tr>
<td>BAD</td>
<td>4</td>
<td>&quot;Bad parameters: ac and deff are negative&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants SIMULATEINSPIRALH_E<name>, and the status descriptions in SIMULATEINSPIRALH_MSGE<name>. The source code with these messages is in SimulateInspiral.h on line 1.128.
Types

Structure SimulateInspiralParamStruc

This structure stores the parameters required to simulate a set of inspiral signal in white Gaussian noise. It can be part of a linked list of inspiral events, to allow for multiple injections. It consists of the following fields:

LIGOTimeGPS timeC The time of coalescence.

REAL4 phiC The wave phase at coalescence, in radians.

REAL4 mass1, mass2 The masses of the binary components, in M⊙.

REAL4 signalAmplitude, effDist The characteristic detection amplitude A_c, in ADC counts, and the effective distance in Mpc of an optimally-oriented source that would give that amplitude. A negative number means the quantity is unspecified. In general only one of these must be specified by the user; the simulation routine will set the other to be consistent with the first.

REAL4 fStart The lower cutoff frequency at which waveform generation will begin, in Hz. If ≤ 0, the cutoff frequency will be taken as the point where the instrument sensitivity function is ∼ 10⁻⁶ of its optimal value, as determined from the transfer function.

SimulateInspiralParamStruc *next Pointer to another inspiral event to be injected, or NULL if this is the last (or only) injection.

Conventions

We define here the conventions we use when talking about signal-to-noise ratios in coloured and white noise. You may also want to read the signal processing conventions in Secs. .1.1 and .1.2 of the findchirp package, since this section is essentially a summary and extension of those conventions.

Signal-to-noise definitions

We first reiterate the standard definitions (given in the findchirp package) of the Fourier transform pair:

$$\hat{a}(f) = \int_{-\infty}^{\infty} dt \ a(t) e^{-\pi if t} \quad \Rightarrow \quad a(t) = \int_{-\infty}^{\infty} df \ \hat{a}(f) e^{\pi if t},$$

(15.6)

of the power spectral density $S_n(f)$ of a stationary random process (noise) $n(t)$:

$$\langle \hat{n}(f) \hat{n}^*(f') \rangle = \frac{1}{2} S_n(f) \delta(f - f')$$

(15.7)

(where $\langle \ldots \rangle$ denotes an ensemble average over instantiations of the random process), and finally of the noise-weighted inner product $(a|b)$ of two time series $a(t)$ and $b(t)$:

$$(a|b) = \int_{-\infty}^{\infty} df \ \frac{\hat{a}(f)\hat{b}(f) + \hat{a}(f)^*\hat{b}(f)}{S_n(f)}.$$  (15.8)

In the case where the time series are all real and the noise has zero mean $\langle n(t) \rangle = 0$, the weighting on the inner product leads to the property that:

$$\langle (n|s)^2 \rangle = (s|s).$$  (15.9)

We call this quantity the intrinsic signal-to-noise power of the waveform $s(t)$, and its square root $\sqrt{(s|s)}$ the intrinsic signal-to-noise amplitude.

In the theory of signal processing, if one has a data stream $o(t) = n(t) + s(t)$ and can determine a matched filter $a(t) \propto s(t)$, one can then define a signal-to-noise estimator $r = (o|a)/\sqrt{(a|a)}$ with the property that $\langle r \rangle = \sqrt{(s|s)}$ and $\sigma_r = \sqrt{(r^2) - \langle r \rangle^2} = 1$ (these follow from Eq. (15.9) and $\langle n \rangle = 0$). This is the justification for calling $\sqrt{(s|s)}$ a signal-to-noise ratio.

However, in many cases one can only define a set of orthogonal waveforms $\{a_1, \ldots, a_N : (a_i|a_j) = (a_i|a_i)\delta_{ij}\}$ spanning the space of possible signals. Specifically, for inspiral signals, one does not know in advance the phase of the waveform, and so one must filter the data using two orthogonal waveforms $\{a_s, a_c\}$
(the sine and cosine quadratures) that are 90° out of phase. As described in the `FindChirp.h` header, the optimal statistic in this case is:

$$\rho^2 = \frac{(o|a_s)^2}{(a_s|a_s)} + \frac{(o|a_c)^2}{(a_c|a_c)}$$  \hspace{1cm} (15.10)

(where we have implicitly already maximized over any filter parameters, including time-of-arrival). This statistic no longer has unit variance, but instead has the property that:

$$\langle \rho^2 \rangle = 2 + \langle ss \rangle.$$  \hspace{1cm} (15.11)

By comparison, for the perfectly-matched filter one has $\langle r^2 \rangle = 1 + \langle ss \rangle$, which is why one often says that the search over phase halves the effective signal-to-noise power. However, this statement is ambiguous, as the two statistics $\rho$ and $r$ have completely different probability distributions in the presence of noise, and even in the absence of any signal we have $\langle \rho \rangle > 0$ (its value depends on the particular noise probability distribution).

### Specification to white noise

White noise is stationary zero-mean noise whose power spectral density $S_n(f)$ is independent of $f$. This property allows inner products to be expressed equivalently in the time domain as well as in the frequency domain:

$$\langle a|b \rangle = \int_{-\infty}^{\infty} dt \frac{a(t)b(t)^* + a(t)^*b(t)}{S_n}.$$  \hspace{1cm} (15.12)

If the white noise process $n(t)$ is discretely sampled at intervals $\Delta t$, we find that different time samples are uncorrelated:

$$\langle n(t_j)n(t_{j'})^* \rangle = \sigma_n^2\delta_{jj'},$$  \hspace{1cm} (15.13)

where $\sigma_n^2$ is the variance in the sampled noise. This can be proven using the definitions of the discrete inverse FFT and discrete power spectrum given in Eqs. (9) and (19) of the `findchirp` package:

$$\langle n(t_j)n(t_{j'})^* \rangle = \frac{1}{N^2} \sum_{k=0}^{N-1} \sum_{k'=0}^{N-1} e^{i2\pi(j-k')N} \langle \tilde{n}_k\tilde{n}_{k'} \rangle = \frac{1}{2N\Delta t} S_n \delta_{kk'},$$

whence we have $\sigma_n^2 = S_n/2\Delta t$. We note that the divergence as $\Delta t \to 0$ reflects the fact that the idealized white power spectral density integrates to an infinite amount of power as $f \to \infty$. Real noise processes always have some minimum timescale below which correlations between subsequent measurements are inevitable.

Reexpressing the inner product in Eq. (15.12) for discretely-sampled time series, we have:

$$\langle a|b \rangle = \sum_k \Delta t \frac{a(t_k)b(t_k)^* + a(t_k)^*b(t_k)}{S_n} = \frac{\text{Re} \left[ \sum_k a(t_k)b(t_k)^* \right]}{\sigma_n^2}.$$  \hspace{1cm} (15.14)

The intrinsic signal-to-noise amplitude also takes an intuitive form:

$$\sqrt{\langle ss \rangle} = \sqrt{\sum_k |s(t_k)|^2} / \sigma_n.$$  \hspace{1cm} (15.15)

This, then, is the key formula that we will use to determine the signal-to-noise amplitude of a whitened signal that is to be injected into white noise. It applies to any stationary white noise (recall that “white” implies zero mean), but we will almost always be concerned with white Gaussian noise, where the differential probability of a noise sample $n(t_k)$ lying in an infinitesimal range $(n, n+dn)$ is:

$$\frac{dP[n(t_k) \in (n, n+dn)]}{dn} = \frac{1}{\sqrt{2\pi\sigma_n^2}} e^{-n^2/2\sigma_n^2}.$$  \hspace{1cm} (15.16)
15.3.1 Module SimulateInspiral.c

Injests inspiral waveforms into detector output.

Prototypes

```
void LALSimulateInspiral( LALStatus *stat,
    REAL4TimeSeries *output,
    COMPLEX8FrequencySeries *transfer,
    SimulateInspiralParamStruc *params );
```

Description

This function generates a binary inspiral signal using the parameters in *params, simulates an instrument’s response to that signal using the instrument transfer function *transfer, and injects the resulting waveform into detector output stored in *output.

The *output time series should have all of its fields set to their desired values, and should have a data sequence already allocated; the function LALSimulateInspiral() simply adds the inspiral waveform on top of the existing data. The epoch and deltaT fields must be set, as they are used to determine the sample rate and time positioning of the injected signal. The sampleUnits field must be set to lalADCCountUnit for consistency.

The *transfer frequency series should define the complex frequency response function $T(f)$ taking the differential strain signal $\tilde{h}(f)$ to detector response $\tilde{o}(f) = T(f)\tilde{h}(f)$, and should have units of ADC counts per strain. It is treated as zero outside its frequency domain, and is linearly interpolated between its frequency samples.

The *params structure represents the parameters of an inspiral signal to be injected (if params->next=NULL), or the head of a linked list of parameter structures for multiple injections. For each structure, if the signalAmplitude field is $\geq 0$, the injected waveform will be scaled to give it the correct characteristic detection amplitude, and the effDist field is set appropriately. If signalAmplitude<0 and effDist>0, the waveform is injected with that effective distance, and the signalAmplitude field is set appropriately. If signalAmplitude<0 and effDist$\leq 0$, an error is returned (that and all subsequent injections are skipped).

An error is also returned (and no injections performed) if any of the fields of *output and *transfer are not set to usable values, including such things as wrong units or bad sampling intervals.

Usage

One of the most useful applications of this routine is to generate simulated noise containing a signal. The following code snippet generates white Gaussian noise with rms amplitude SIGMA, and injects a signal with intrinsic signal-to-noise ratio $\sqrt{(h|h)} = \text{SNR}$ into it, coalescing at a time DT seconds from the start of the time series, with a wave phase PHI at coalescence. The REAL4TimeSeries output and COMPLEX8FrequencySeries transfer structures are assumed to be defined and allocated outside of this block.

```
{  
    UINT4 i;
    SimulateInspiralParamStruc inspParams;
    RandomParams *randParams = NULL;

    /* Generate white Gaussian noise. */
    LALCreateRandomParams( status->statusPtr, &randParams, 0 );
    LALNormalDeviates( status->statusPtr, output.data, randParams );
    for ( i = 0; i < output.data->length; i++ )
        output.data->data[i] *= SIGMA;
    LALDestroyRandomParams( status->statusPtr, &randParams );

    /* Inject signal. */
    inspParams.timeC = output.epoch;
    ```
inspParams.timeC.gpsSeconds += DT;
inspParams.phiC = PHI; inspParams.mass1 = M1; inspParams.mass2 = M2;
inspParams.signalAmplitude = SNR*SIGMA;
inspParams.next = NULL;
LALSimulateInspiral( status->statusPtr, &output, &transfer, &inspParams );
}

Algorithm
The default mode of operation, when one specifies the desired amplitude, is as follows:
First, LALGeneratePPNInspiral() is called to generate the signal, placing the source at a distance of 1Mpc with optimal orientation. For lack of anything better, the amplitude and phase functions are sampled at the full sampling interval as the output data stream. This function call also returns the time at coalescence.
Second, the waveform produced by the signal in the detector output stream is calculated. The basic algorithm is the same as that in LALSimulateCoherentGW(), but we can simplify it significantly because we ignore polarization responses, time delays, and interpolation between time samples. Thus we have only to compute the effect of the frequency transfer function $T(f)$. As stated in the SimulateCoherentGW.h header, for quasiperiodic waveforms $h(t) = \text{Re}[H(t)e^{i\phi(t)}]$ we can approximate the instrument response (in the absence of noise) as:
$$o(t) \approx \text{Re}[T\{f(t)\}H(t)e^{i\phi(t)}].$$
In our case we are only sensitive to a single polarization (let’s say $h_+$), so we take $H(t) = A_+(t)$, where the phase of $H$ is absorbed into the coalescence phase of $\phi$. Then we can write the instrument response as:
$$o(t) \approx A_+(t)[T_{\text{re}}\{f(t)\}\cos\phi(t) - T_{\text{im}}\{f(t)\}\sin\phi(t)].$$
This calculation can be done in place and stored in one of the arrays for the amplitude, phase, or frequency functions, since they are already sampled at the correct rate and have the correct length.
Third, the characteristic detection amplitude is computed, and the whole waveform is scaled so that it has the correct value. Simultaneously, the effective distance is set to 1Mpc/(the scale factor). The epoch is also adjusted to give the waveform the correct coalescence time.
Finally, LALSSInjectTimeSeries() is called to inject the waveform into the output time series. The whole procedure is repeated for any other nodes in the linked list of parameters.
If any parameter structure specifies the effective distance in place of the characteristic detection amplitude, then the signal is injected with that effective distance and is not rescaled. The characteristic detection amplitude field is set to the measured value.

Uses
LALWarning()
LALDDestroyVector() LALFree()
LALSDestroyVector() LALSDestroyVectorSequence()
LALGenerateFPNInspiral() LALSSInjectTimeSeries()

Notes
15.4 Header \texttt{SimulateSB.h}

Provides prototype and error code information for the modules needed to simulate a stochastic background signal (whitened, if desired) in a pair of detectors, given the appropriate representations of the detector transfer function in each detector.

Synopsis

\#include <lal/SimulateSB.h>

Error conditions

<table>
<thead>
<tr>
<th>\texttt{name}</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{NULLP}</td>
<td>1</td>
<td>&quot;Null pointer&quot;</td>
</tr>
<tr>
<td>\texttt{NONPOSLEN}</td>
<td>2</td>
<td>&quot;Negative or zero length for data member of time series&quot;</td>
</tr>
<tr>
<td>\texttt{NONPOSDELTAF}</td>
<td>3</td>
<td>&quot;Negative or zero frequency spacing&quot;</td>
</tr>
<tr>
<td>\texttt{NONPOSDELTAT}</td>
<td>4</td>
<td>&quot;Negative or zero time spacing&quot;</td>
</tr>
<tr>
<td>\texttt{NEGFMN}</td>
<td>5</td>
<td>&quot;Negative start frequency&quot;</td>
</tr>
<tr>
<td>\texttt{MMTIME}</td>
<td>6</td>
<td>&quot;Mismatch in epochs&quot;</td>
</tr>
<tr>
<td>\texttt{MMHETERO}</td>
<td>7</td>
<td>&quot;Mismatch in heterodyning frequencies&quot;</td>
</tr>
<tr>
<td>\texttt{MMFMIN}</td>
<td>8</td>
<td>&quot;Mismatch in start frequencies&quot;</td>
</tr>
<tr>
<td>\texttt{MMDeltaF}</td>
<td>9</td>
<td>&quot;Mismatch in frequency spacings&quot;</td>
</tr>
<tr>
<td>\texttt{MMLLEN}</td>
<td>10</td>
<td>&quot;Mismatch in sequence lengths&quot;</td>
</tr>
<tr>
<td>\texttt{OORFREF}</td>
<td>11</td>
<td>&quot;Out of range reference frequency&quot;</td>
</tr>
<tr>
<td>\texttt{NONPOSOMEGA}</td>
<td>12</td>
<td>&quot;Negative stochastic background strength&quot;</td>
</tr>
<tr>
<td>\texttt{ALOC}</td>
<td>13</td>
<td>&quot;Memory allocation error&quot;</td>
</tr>
<tr>
<td>\texttt{NONZEROMHETERO}</td>
<td>14</td>
<td>&quot;Non-zero heterodyning frequency specified for real time series&quot;</td>
</tr>
<tr>
<td>\texttt{WRONGUNITS}</td>
<td>15</td>
<td>&quot;Inconsistent input units&quot;</td>
</tr>
<tr>
<td>\texttt{COMPTIME}</td>
<td>16</td>
<td>&quot;Time domain data complex instead of real&quot;</td>
</tr>
<tr>
<td>\texttt{NOTYETHETERO}</td>
<td>255</td>
<td>&quot;Non-zero heterodyning frequency not yet implemented&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants \texttt{SIMULATESBH\_E\_\texttt{name}}, and the status descriptions in \texttt{SIMULATESBH\_MSGE\_\texttt{name}}. The source code with these messages is in \texttt{SimulateSB.h} on line 1.101.

Structures

Structures associated with \texttt{SimulateSB.c} (Sec. 15.4.1)

\textbf{struct \texttt{SSSimStochBGOutput}}

Contains the output data produced by \texttt{LAL\texttt{SSSimStochBGTimeSeries}(). It comprises of a pair of (real) time-series simulated stochastic background signal in the outputs of a given pair of detectors. The fields are:

\texttt{REAL4TimeSeries *SSimStochBG1} Simulated stochastic background signal in the output of the first detector.

\texttt{REAL4TimeSeries *SSimStochBG2} Simulated stochastic background signal in the output of the second detector.

\textbf{struct \texttt{SSSimStochBGInput}}

Contains the input data needed by \texttt{LAL\texttt{SSSimStochBGTimeSeries}()} to calculate the whitened stochastic background signal in the output of a detector. The fields are:

\texttt{REAL4FrequencySeries *omegaGW} The spectrum \(\Omega_{GW}(f)\) of the stochastic gravitational-wave background.
COMPLEX8FrequencySeries *whiteningFilter1 The frequency-domain response function $\tilde{R}_1(f)$ for the first detector.

COMPLEX8FrequencySeries *whiteningFilter2 The frequency-domain response function $\tilde{R}_2(f)$ for the second detector.

struct SSSimStochBGParams
Contains the parameters used by LALSSSimStochBGTimeSeries() to compute the whitened stochastic background signal in the output of an interferometric detector. The fields are:

UINT4 length The number of points in the output time series.
REAL8 deltaT The temporal spacing of the output time series.
INT4 seed The random number seed for the stochastic simulation.

LALDetector *detector1 The site location and orientation information of first detector involved in the stochastic background search.

LALDetector *detector2 The site location and orientation information of second detector involved in the stochastic background search.

LALUnit SSimStochBGTimeSeries1Unit The unit field of the stochastic background, expressed as a Real4 time series, in detector 1.

LALUnit SSimStochBGTimeSeries2Unit The unit field of the stochastic background, expressed as a Real4 time series, in detector 2.
15.4.1 Module SimulateSB.c

Simulates whitened time-domain signal in a pair of detectors that arises purely from an isotropic and unpolarized stochastic background of gravitational radiation with the desired power spectrum, $\Omega_{GW}(f)$. This module will evolve beyond its present functionality to produce only real time-series signal for a pair of interferometric detectors.

Prototypes

```c
void LALSSSimStochBGTimeseries( LALStatus *status, SSSimStochBGOutput *output, SSSimStochBGInput *input, SSSimStochBGParams *params )
```

Description

The frequency domain strains $\tilde{h}_1(f_i)$ and $\tilde{h}_2(f_j)$ caused by the stochastic background in two detectors are random variables that have zero mean and that obey \[6\]:

$$\langle \tilde{h}_1^*(f_i)\tilde{h}_1(f_j) \rangle = \frac{3H_0^2T}{20\pi^2} \delta_{ij} f_i^{-3} \gamma_{11}(f_i) \Omega_{GW}(|f_i|)$$  \hspace{1cm} (15.17)

and

$$\langle \tilde{h}_2^*(f_i)\tilde{h}_2(f_j) \rangle = \frac{3H_0^2T}{20\pi^2} \delta_{ij} f_i^{-3} \gamma_{22}(f_i) \Omega_{GW}(|f_i|)$$  \hspace{1cm} (15.18)

and

$$\langle \tilde{h}_1^*(f_i)\tilde{h}_2(f_j) \rangle = \frac{3H_0^2T}{20\pi^2} \delta_{ij} f_i^{-3} \gamma_{12}(f_i) \Omega_{GW}(|f_i|) ,$$  \hspace{1cm} (15.19)

where $\langle \rangle$ denotes ensemble average, $T$ is the time of observation, and $\gamma_{AB}$ is the overlap reduction function \(\Upsilon\) of the detector pair comprising detectors $A$ and $B$. Above, $\tilde{h}_1(f_i)$ and $\tilde{h}_2(f_j)$ are the Fourier components of the gravitational strains $h_1(t)$ and $h_2(t)$ at the two detectors.

The Fourier components that obey the above relations are

$$\tilde{h}_1(f_i) = \sqrt{\frac{3H_0^2T}{40\pi^2} f_i^{-3/2} \Omega_{GW}^{1/2}(|f_i|) \sqrt{\gamma_{11}(f_i)}} (x_{1i} + iy_{1i})$$  \hspace{1cm} (15.20)

and

$$\tilde{h}_2(f_i) = \tilde{h}_1(f_i) \frac{\gamma_{12}(f_i)}{\gamma_{11}(f_i)} + \sqrt{\frac{3H_0^2T}{40\pi^2} f_i^{-3/2} \Omega_{GW}^{1/2}(|f_i|) \sqrt{\gamma_{22}(f_i)} - \frac{\gamma_{12}^2(f_i)}{\gamma_{11}(f_i)}} (x_{2i} + iy_{2i})$$  \hspace{1cm} (15.21)

where $x_{1i}$, $y_{1i}$, $x_{2i}$, and $y_{2i}$ are statistically independent real Gaussian random variables, each of zero mean and unit variance.

The routine assumes as inputs the data sample length, temporal spacing, stochastic background characteristics, detector locations, the appropriate representations of the detector response function in each detector, etc. The (frequency domain) response functions, $R_1(f_i)$ and $R_2(f_i)$ are used to whiten the strains $\tilde{h}_1(f_i)$ and $\tilde{h}_2(f_i)$, respectively, to obtain the whitened Fourier components:

$$\tilde{o}_1(f_i) = R_1(f_i)\tilde{h}_1(f_i)$$  \hspace{1cm} (15.22)

and

$$\tilde{o}_2(f_i) = R_2(f_i)\tilde{h}_2(f_i) .$$  \hspace{1cm} (15.23)

To obtain the whitened (real) outputs $o_1(t_i)$ and $o_2(t_i)$ in the time domain, the inverse Fourier transforms of the above frequency series are taken.
Algorithm

The routine `LALSSimStochBGTimeSeries()` produces only real time-series signal for a pair of interferometric detectors. It first inputs the frequency series describing the power spectrum of the stochastic background, $\Omega_{GW}(|f|)$, which the simulated signal is required to represent. It also inputs two `COMPLEX8` frequency series corresponding, respectively, to the two detector response functions. As parameters, it takes the two `LALDetector` structures corresponding to the two detectors in which the signal is to be mimicked. It also takes the time length (given in terms of the number of time data samples), the time spacing, a seed (for generating random numbers), and a couple of `LALUnit` structures for specifying the units of the two time-series signals that the routine outputs.

Using the specified power spectrum for the stochastic background, and a random number generator (of zero mean, unit variance Gaussian distributions), the routine produces $\tilde{h}_1(f_i)$ and $\tilde{h}_2(f_i)$. The response functions of the two detectors are then used to whiten the two strains in the Fourier domain. Their inverse transform is then taken to obtain at each detector the whitened simulated signal in the time domain.

Uses

- `LALStochasticOmegaGW()`
- `LALReverseRealFFT()`
- `LALNormalDeviates()`

Notes

- This routine does not yet support non-zero heterodyning frequencies.
15.5  Header  SimulatePopcorn.h

Provides prototype for simulating whitened time-domain signals in a pair of detectors that arises from low
duty cycle astrophysical backgrounds.

Synopsis

#include <lal/SimulatePopcorn.h>

Error conditions

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULLP</td>
<td>1</td>
<td>&quot;Null pointer&quot;</td>
</tr>
<tr>
<td>NONNULLFMIN</td>
<td>2</td>
<td>&quot;Non zero start frequency&quot;</td>
</tr>
<tr>
<td>MMDELTA</td>
<td>3</td>
<td>&quot;Mismatch in sequence spacings&quot;</td>
</tr>
<tr>
<td>MMLEN</td>
<td>4</td>
<td>&quot;Mismatch in sequence lengths&quot;</td>
</tr>
<tr>
<td>BV</td>
<td>5</td>
<td>&quot;Bad input or parameter&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants SIMULATEPOPCORNH_E<name>, and the
status descriptions in SIMULATEPOPCORNH_MSGE<name>. The source code with these messages is in SimulatePopcorn.h on line 1.91.

Structures

These constants define the cosmological model

*/
#define SIMULATEPOPCORN_ho 0.7
#define SIMULATEPOPCORN_OMEGAMATTER 0.3
#define SIMULATEPOPCORN_OMEAGAVACUUM 0.7
/*
*/

Structures

These are function pointers to functions that model burst waveforms.

*/
typedef void (REAL4LALWform) (REAL4 *output, REAL4 input);
/*
*/

The following structure contains the input of the simulation.

*/
typedef struct tagSimPopcornInputStruc {
    REAL4LALWform  *inputwform; /*waveform of a single burst*/
    REAL4  inputduration; /*mean duration of a single burst*/
    REAL4  inputlambda; /*mean time interval between successive bursts*/
    UINT4  inputNdataset; /*number of detector sites 1 for H1/H2, 2 for H/L*/
    INT2   inputsite0; /*first detector code*/
    INT2   inputsite1; /*second detector code*/
    COMPLEX8FrequencySeries  *wfilter0; /*response of the first detector*/
    COMPLEX8FrequencySeries  *wfilter1; /*response of the second detector*/
} SimPopcornInputStruc;
/*
*/

The following structure contains the parameters of the simulation.
typedef struct tagSimPopcornParamsStruc {
    UINT4 paramsstarttime; /*starting time*/
    UINT4 paramslength; /*length of the time serie in s*/
    UINT4 paramssrate; /*sampling rate of the time serie in Hz*/
    UINT4 paramsseed; /*random generator seed*/
    REAL8 paramsfref; /*reference frequency if normalization, -1 otherwise*/
} SimPopcornParamsStruc;

The following structure contains the simulated pair time series and Ω spectrum

typedef struct tagSimPopcornOutputStruc {
    REAL4TimeSeries *SimPopcorn0;
    REAL4TimeSeries *SimPopcorn1;
    REAL4FrequencySeries *omega0;
    REAL4FrequencySeries *omega1;
} SimPopcornOutputStruc;
15.6 Header GenerateInspiral.h

Header file for the inspiral injection interface code. The code contained in GenerateInspiral.c is an interface between the injection package and the inspiral package. More precisely, the function GenerateInspiral.c is used within the FindChirpSimulation.c file of the FindChirp package in order to inject waveforms into real data. The injection is done through the inject package in order to take into account the interferometer position, binary orientation ...  

GenerateInspiral has the capability of injecting both waveform designed within the inspiral package (TaylorT1, T2, T3, PadeT1, EOB, and spinning waveform) and the inject package (so-called PPN waveform). also a test code as well which allows to check the output of code. It is called InjectionInterfaceTest.c

Synopsis

```c
#include <lal/GenerateInspiral.h>
```

Error codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NORM</td>
<td>0</td>
<td>&quot;Normal exit&quot;</td>
</tr>
<tr>
<td>NULL</td>
<td>1</td>
<td>&quot;Null pointer&quot;</td>
</tr>
<tr>
<td>DFDT</td>
<td>2</td>
<td>&quot;Waveform sampling interval is too large&quot;</td>
</tr>
<tr>
<td>ZERO</td>
<td>3</td>
<td>&quot;inclination zero for SpinTaylor waveform&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants `GENERATEINSPIRALH_E<name>` and the status descriptions in `GENERATEINSPIRALH_MSGE<name>`. The source code with these messages is in `GenerateInspiral.h` on line 1.90.
15.6.1 Module **GenerateInspiral.c**

Generates a CoherentGW inspiral waveform for injection.

Prototypes

```c
void LALGenerateInspiral(
    LALStatus *status,
    CoherentGW *waveform,
    SimInspiralTable *thisEvent,
    PPNParamStruc *ppnParams
)

void LALGetApproximantFromString(
    LALStatus *status,
    CHAR *thisEvent,
    Approximant *approximant
)

void LALGetOrderFromString(
    LALStatus *status,
    CHAR *thisEvent,
    Order *order
)

void LALGenerateInspiralPopulatePPN(
    LALStatus *status,
    PPNParamStruc *ppnParams,
    SimInspiralTable *thisEvent
)

void LALGenerateInspiralPopulateInspiral(
    LALStatus *status,
    InspiralTemplate *inspiralParams,
    SimInspiralTable *thisEvent,
    PPNParamStruc *ppnParams
)
```

**LALGenerateInspiral()** create an inspiral binary waveform generated either by the **inspiral** package (EOB, PadeT1, TaylorT1, TaylorT2, TaylorT3, SpinTaylor) or the **inject** package (GeneratePPN). It is used in the module **FindChirpSimulation** in **findchirp** package.

There are three parsed arguments

- a **CoherentGW** structure which stores amplitude, frequency and phase of the waveform (output)
- a **thisEvent** structure which provides some waveform parameters (input)
- a **PPNParamStruc** which gives some input parameters needed by the GeneratePPN waveform generation. That arguments is also used as an output by all the different approximant (output/input).

The input must be composed of a valid thisEvent structure as well as the variable deltaT of the PPNParamStruc. All others variables of the PPNParamStruc are populated within that function.

**LALGetOrderFromString()** convert a string provided by the **CoherentGW** structure in order to retrieve the order of the waveform to generate.
LALGetApproximantFromString() convert a string provided by the CoherentGW structure in order to retrieve the approximant of the waveform to generate.

LALGenerateInspiralPopulatePPN() Populate the PPNParamsStruc with the input argument thisEvent. That structure is used by both inspiral waveforms inject waveforms.

LALGenerateInspiralPopulateInspiral() Populate the InspiralTemplate structure if the model chosen belongs to the inspiral package.

Algorithm
None.

Notes
Inject only time-domain waveforms for the time being such as GeneratePPN, TaylorT1, TaylorT2, TaylorT3, PadeT1 and EOB , Spintaylor..

Uses
None.
15.7 Header GeneratePPNInspiral.h

Provides routines to generate restricted parametrized post\(^{5/2}\)-Newtonian inspiral waveforms.

Synopsis

```
#include <lal/GeneratePPNInspiral.h>
```

This header covers routines to generate a "restricted" parametrized post\(^{5/2}\)-Newtonian binary inspiral waveform in the time domain. That is, the calculation of the wave phase is accurate to post\(^{5/2}\)-Newtonian order (including corrections up to order \(v^6/c^5\), where \(v\) is the orbital speed), but the wave amplitudes are accurate only to leading (post\(^0\)-Newtonian) order. Furthermore, at each order the post\(^{n}\)-Newtonian correction can be turned on, off, or set to an unphysical value, by adjusting a parameter \(p_n\).

The post-Newtonian expansion implicitly assumes an adiabatic inspiral, where one can represent the waveform by an "instantaneous" amplitude and frequency that vary over timescales longer than one wave period. The orbital frequency of the system to post\(^{5/2}\)-Newtonian order is given in Eqs. 6.4.1 and 6.9.1 of \cite{1}; here we work entirely in terms of the gravitational-wave frequency, which is twice the orbital frequency:

\[
f(t) = \frac{M_\odot}{8\pi T_\odot m_{\text{tot}}} \left\{ p_0 \Theta^{-3/8} + p_1 \Theta^{-1/2} + p_2 \left( \frac{743}{2688} + \frac{11}{32} \eta \right) \Theta^{-5/8} - p_3 \frac{3\pi}{10} \Theta^{-3/4} + p_4 \left( \frac{1855099}{1445068} + \frac{5697}{258048} \eta + \frac{371}{2048} \eta^2 \right) \Theta^{-7/8} - p_5 \left( \frac{7729}{21504} + \frac{3}{256} \eta \right) \pi \Theta^{-1} \right\},
\]

where \(M_\odot\) is the mass of the Sun, \(T_\odot = GM_\odot/c^3 = 4.925491 \times 10^{-6}\) is the "geometrized" solar mass in time units, \(m_{\text{tot}} = m_1 + m_2\) is the total mass of the binary, \(\eta = m_1 m_2 / m_{\text{tot}}^2\) is the (symmetric) mass ratio parameter, and \(\Theta\) is a dimensionless time parameter:

\[
\Theta(t) = \frac{\eta M_\odot}{5T_\odot m_{\text{tot}}}(t_c - t).
\]

Here \(t_c\) is the time of coalescence of the two masses in the point-mass approximation. The post-Newtonian parameters \(p_k\) are defined such that in a normal (physical) post\(^{n}\)-Newtonian expansion, one sets \(p_1 = 0\) and \(p_{k>n} = 0\), and \(p_k = 1\) for all other \(k\). However, changing this convention can be used to model in an approximate way things such as spin, eccentricity, or non-GR theories of gravity. We also note that while most terms are normalized to their normal post-Newtonian values, the normalization on the \(p_1\) term is completely arbitrary, since it is zero in a normal post-Newtonian expansion.

The wave phase as a function of time can be computed analytically from Eq. \eqref{eq:15.24} as \(\phi_{\text{orb}} = 2\pi \int f dt\):

\[
\phi(t) = \phi_c - \frac{2}{\eta} \left\{ p_0 \Theta^{5/8} + p_1 \frac{5}{4} \Theta^{1/2} + p_2 \left( \frac{3715}{8064} + \frac{55}{96} \eta \right) \Theta^{3/8} - p_3 \frac{3\pi}{4} \Theta^{1/4} + p_4 \left( \frac{9275495}{1445068} + \frac{284875}{258048} \eta + \frac{1855}{2048} \eta^2 \right) \Theta^{7/8} - p_5 \left( \frac{38645}{172032} + \frac{15}{2048} \eta \right) \pi \log \left( \frac{\Theta}{\Theta_0} \right) \right\}
\]

Here \(\Theta_0\) is an arbitrary constant; changing it is equivalent to changing \(\phi_c\). We note that the post\(^{5/2}\)-Newtonian term introduces a late-time divergence in phase which renders meaningless the interpretation of \(\phi_c\) as "phase at coalescence"; in our convention we define \(\phi_c\) to correspond to the case \(\Theta_0 = 1\).

We refer the interested reader to Sec. 6.6 of \cite{1} for a discussion of how propagation effects shift the phase of the waveform relative to the orbital phase. To summarize, though: A changing propagation delay does introduce a time-dependent phase shift in the waveform, but the dependence on \(t\) is weak except at very late times; although it looks like a post\(^{1/2}\)-Newtonian phase correction, it can in fact be represented as a post\(^3\)-Newtonian phase correction combined with a post\(^{3/2}\)-Newtonian amplitude correction. Since we are concerned with restricted post\(^{5/2}\)-Newtonian waveforms, which model the amplitude only to leading (post\(^{5}\)-Newtonian) order, we can ignore these propagation effects.

To leading order, then, the amplitude of the + and \(\times\) polarizations of the wave are given by Eqs. 6.6.1–6.6.4 of \cite{1} as:

\[
A_+ (t) = -\frac{2T_\odot c}{D} (1 + \cos^2 i) \left( \frac{m_{\text{tot}}}{M_\odot} \right) \left[ \pi T_\odot m_{\text{tot}} f(t) \right]^{2/3},
\]

\[
A_\times (t) = -\frac{2T_\odot c}{D} (2\cos i) \left( \frac{m_{\text{tot}}}{M_\odot} \right) \left[ \pi T_\odot m_{\text{tot}} f(t) \right]^{2/3},
\]
where $D$ is the distance to the source and $i$ is the inclination of the axis of the source to the line of sight. The normal polarization convention in [2] is used, where the reference $x$-coordinate axis for the + and $\times$ polarization tensors is the ascending node of the rotational plane as it crosses the plane transverse to the propagation direction. This convention implies that the + and $\times$ waveforms are elliptically polarized as follows:

\[
\begin{align*}
    h_+(t) &= A_+(t) \cos \phi(t), \\
    h_\times(t) &= A_\times(t) \sin \phi(t).
\end{align*}
\]

(15.29)  
(15.30)

Error conditions

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUL</td>
<td>1</td>
<td>&quot;Unexpected null pointer in arguments&quot;</td>
</tr>
<tr>
<td>OUT</td>
<td>2</td>
<td>&quot;output field a, f, phi, or shift already exists&quot;</td>
</tr>
<tr>
<td>TBAD</td>
<td>3</td>
<td>&quot;Bad sampling interval&quot;</td>
</tr>
<tr>
<td>FBAD</td>
<td>4</td>
<td>&quot;Bad starting frequency; could not get valid start time&quot;</td>
</tr>
<tr>
<td>PBAD</td>
<td>5</td>
<td>&quot;Bad post-Newtonian parameters&quot;</td>
</tr>
<tr>
<td>MBAD</td>
<td>6</td>
<td>&quot;Bad masses&quot;</td>
</tr>
<tr>
<td>DBAD</td>
<td>7</td>
<td>&quot;Bad distance&quot;</td>
</tr>
<tr>
<td>MEM</td>
<td>8</td>
<td>&quot;Out of memory&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants `GENERATEPPNINSPIRALH_E<name>`, and the status descriptions in `GENERATEPPNINSPIRALH_MSGE<name>`. The source code with these messages is in `GeneratePPNInspiral.h` on line 1.187.

Termination conditions

In addition to the error conditions above, there are a number of ways that the signal generation routine can terminate gracefully while still returning a valid waveform. In many cases one wants to continue generating a waveform “until things fall apart”; the following codes, returned in the `PPNParamStruc` below, allow the waveform generator to report exactly how things fell apart.

For the sake of LAL namespace conventions, these termination codes are `#defined` and autodocumented exactly like error codes.

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSTOP</td>
<td>0</td>
<td>&quot;Reached requested termination frequency&quot;</td>
</tr>
<tr>
<td>LENGTH</td>
<td>1</td>
<td>&quot;Reached maximum length, or end of provided time series vector&quot;</td>
</tr>
<tr>
<td>FNOTMON</td>
<td>2</td>
<td>&quot;Frequency no longer increasing monotonically&quot;</td>
</tr>
<tr>
<td>PNFAIL</td>
<td>3</td>
<td>&quot;Evolution dominated by higher-order PN terms&quot;</td>
</tr>
<tr>
<td>RTOOSMALL</td>
<td>4</td>
<td>&quot;Orbital radius too small for PN approximation&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants `GENERATEPPNINSPIRALH_E<name>`, and the status descriptions in `GENERATEPPNINSPIRALH_MSGE<name>`. The source code with these messages is in `GeneratePPNInspiral.h` on line 1.211.

Types

Structure `PPNParamStruc`

This structure stores the parameters for constructing a restricted post-Newtonian waveform. It is divided into three parts: parameters passed along to the output structure but not used by waveform generator, parameters used as input to the waveform generator, and parameters set by the generator to evaluate its success.

**Passed fields:**

`SkyPosition position` The location of the source on the sky, normally in equatorial coordinates.

`REAL4 psi` The polarization angle of the source, in radians.
LIGOTimeGPS epoch  The start time of the output series.

Input fields:

REAL4 mTot  The total mass $m_{tot} = m_1 + m_2$ of the binary system, in solar masses.

REAL4 eta  The mass ratio $\eta = m_1 m_2 / m_{tot}^2$ of the binary system. Physically this parameter must lie in the range $\eta \in (0, 1/4]$; values outside of this range may be permitted in order to represent “nonphysical” post-Newtonian expansions.

REAL4 d  The distance to the system, in metres.

REAL4 inc  The inclination of the system to the line of sight, in radians.

REAL4 phi  The phase at coalescence $\phi_c$ (or arbitrary reference phase for a post$^{5/2}$-Newtonian approximation), in radians.

REAL8 deltaT  The requested sampling interval of the waveform, in s.

REAL4 fStartIn  The requested starting frequency of the waveform, in Hz.

REAL4 fStopIn  The requested termination frequency of the waveform, in Hz. If set to 0, the waveform will be generated until a termination condition (above) is met. If set to a negative number, the generator will use its absolute value as the terminating frequency, but will ignore post-Newtonian breakdown; it will terminate only at the requested frequency $-f_{\text{StopIn}}$, a local maximum frequency, or the central singularity.

UINT4 lengthIn  The maximum number of samples in the generated waveform. If zero, the waveforms can be arbitrarily long.

REAL4Vector *ppn  The parameters $p_n$ selecting the type of post-Newtonian expansion. If ppn=NULL, a “normal” (physical) expansion is assumed.

Output fields:

REAL8 tc  The time $t_c - t$ from the start of the waveform to coalescence (in the point-mass approximation), in s.

REAL4 dfdt  The maximum value of $\Delta f \Delta t$ encountered over any timestep $\Delta t$ used in generating the waveform.

REAL4 fStart  The actual starting frequency of the waveform, in Hz (normally close but not identical to fStartIn).

REAL4 fStop  The frequency at the termination of the waveform, in Hz.

INT4 length  The length of the generated waveform.

INT4 termCode  The termination condition (above) that stopped computation of the waveform.

const CHAR *termDescription  The termination code description (above).

Structure GalacticInspiralParamStruc

This structure stores the position and mass parameters of a galactic inspiral event. The fields are:

REAL4 rho  The distance of the binary system from the Galactic axis, in kpc.

REAL4 z  The distance of the system from the Galactic plane, in kpc.

REAL4 lGal  The Galactocentric Galactic longitude of the system (i.e. the Galactic longitude of the direction from the Galactic centre through the system), in radians. See SkyCoordinates.h for the definition of this quantity.

REAL4 m1, m2  The masses of the binary components, in solar masses.

LIGOTimeGPS geocentEndTime  The geocentric end time of the inspiral event.
15.7.1 Module *GeneratePPNInspiral.c*

Computes a parametrized post-Newtonian inspiral waveform.

**Prototypes**

```c
void LALGeneratePPNInspiral( LALStatus *stat,
                           CoherentGW *output,
                           PPNParamStruct *params )
```

**Description**

This function computes an inspiral waveform using the parameters in *params*, storing the result in *output*.

In the *params* structure, the routine uses all the “input” fields specified in *GeneratePPNInspiral.h*, and sets all of the “output” fields. If *params->ppn=NULL*, a normal post-2-Newtonian waveform is generated; i.e. $p_0 = 1$, $p_1 = 0$, $p_2 = 1$, $p_3 = 1$, $p_4 = 1$, $p_5+ = 0$.

In the *output* structure, the field *output->h* is ignored, but all other pointer fields must be set to NULL. The function will create and allocate space for *output->a*, *output->f*, and *output->phi* as necessary. The *output->shift* field will remain set to NULL, as it is not required to describe a nonprecessing binary.

**Algorithm**

This function is a fairly straightforward calculation of Eqs. [15.24–15.28] in *GeneratePPNInspiral.h*. However, there are some nontrivial issues involved, which are discussed in some depth in Secs. 6.4, 6.6, and 6.9.2 of [1]. What follows is a brief discussion of these issues and how this routine deals with them.

**Computing the start time:**

When building a waveform for data analysis, one would generally like to start the waveform at some well-defined frequency where it first enters the band of interest; one then defines the start time of the integration by inverting Eq. [15.24] if *GeneratePPNInspiral.h*. The current algorithm follows this standard approach by requiring the calling routine to specify *params->fStartIn*, which is then inverted to find $\Theta_{\text{start}}$. This inversion is in fact the most algorithmically complicated part of the routine, so we will discuss it in depth.

To help clarify the problem, let us rewrite the equation in dimensionless parameters $y = 8\pi f T_\odot m_{\text{tot}}/M_\odot$ and $x = \Theta^{-1/8}$:

$$y = \sum_{k=0}^{5} C_k x^{k+3}, \quad \text{ (15.31)}$$

where:

- $C_0 = p_0$
- $C_1 = p_1$
- $C_2 = p_2 \left( \frac{743}{2688} + \frac{11}{32} \eta \right)$
- $C_3 = p_3 \frac{3\pi}{10}$
- $C_4 = p_4 \left( \frac{1855099}{14450688} + \frac{56975}{258048} \eta + \frac{371}{2048} \eta^2 \right)$
- $C_5 = p_5 \left( \frac{7729}{21504} + \frac{3}{256} \eta \right) \pi$

We note that $x$ is a time parameter mapping the range $t = -\infty \rightarrow t_c$ to $x = 0 \rightarrow \infty$.

In a normal post-Newtonian expansion it is possible to characterize the general behaviour of this equation quite accurately, since the values of $p_k$ are known and since $\eta$ varies only over the range [0,1/4]. In a parametrized post-Newtonian expansion, however, even the relative orders of magnitude of the coefficients can vary significantly, making a robust generic root finder impractical. However, we are saved by the fact that we can restrict our search to domains where the post-Newtonian expansion is a valid approximation. We define the post-Newtonian expansion not to be valid if any of the following conditions occur:
1. A higher-order term in the frequency expansion becomes larger in magnitude than the leading (lowest-order nonzero) term.

2. The inferred orbital radius, approximated by \( r \sim 4m_{\text{tot}} \Theta^{1/4} \), drops below \( 2m_{\text{tot}} \); i.e. \( \Theta < 1/16 \) or \( x > \sqrt{2} \).

3. The frequency evolution becomes non-monotonic.

We can further require as a matter of convention that the lowest-order nonzero coefficient in the frequency expansion be positive; this is simply a sign convention stating that the frequency of a system be positive at large radii.

The first two conditions above allow us to set firm limits on the range of the initial \( x_{\text{start}} \). Let \( C_j \) be the lowest-order nonzero coefficient; then for every nonzero \( C_{k>j} \) we can define a point \( x_k = |C_j/C_k|^{1/(k-j)} \) where that term exceeds the leading-order term in magnitude. We can therefore limit the range of \( x \) to values less than \( x_{\text{max}} \), which is the minimum of \( \sqrt{2} \) and all \( x_k \). We note that even if we were to extend the post-Newtonian expansion in Eq. [15.31] to an infinite number of terms, this definition of \( x_{\text{max}} \) implies that the frequency is guaranteed to be monotonic up to \( x_{\text{max}}(5 - \sqrt{7})/6 \), and positive up to \( x_{\text{max}}/2 \). Thus we can confidently begin our search for \( x_{\text{start}} \) in the domain \((0, 0.39x_{\text{max}})\), where the leading-order term dominates, and end it if we ever exceed \( x_{\text{max}} \).

We therefore bracket our value of \( x_{\text{start}} \) as follows: We start with an initial guess \( x_{\text{guess}} = (y_{\text{start}}/C_j)^{1/(1+j)} \), or \( 0.39x_{\text{max}} \), whichever is less. If \( y(x_{\text{guess}}) > y_{\text{start}} \), we iteratively decrease \( x \) by factors of 0.95 until \( y(x) < y_{\text{start}} \); this is guaranteed to occur within a few iterations, since we are moving into a regime where the leading-order behaviour dominates more and more. If \( y(x_{\text{guess}}) < y_{\text{start}} \), we iteratively increase \( x \) by factors of 1.05 until \( y(x) > y_{\text{start}} \), or until \( x > x_{\text{max}} \); this is also guaranteed to occur quickly because, in the worst case, it only takes about 20 iterations to step from \( 0.39x_{\text{max}} \) to \( x_{\text{max}} \), and if \( x_{\text{guess}} \) were much lower than \( 0.39x_{\text{max}} \) it would have been a pretty good guess to begin with. If at any point while increasing \( x \) we find that \( y \) is decreasing, we determine that the starting frequency is already in a regime where the post-Newtonian approximation is invalid, and we return an error. Otherwise, once we have bracketed the value of \( x_{\text{start}} \), we use \textit{LALSBisectionFindRoot()} to pin down the value to an accuracy of a part in \( 10^5 \).

**Computing the phase and amplitudes:**

Once we have \( x_{\text{start}} \), we can find \( \Theta_{\text{start}} \), and begin incrementing it; at each timestep we compute \( x \) and hence \( f \), \( \phi \), \( A_+ \), and \( A_\times \) according to Eqs. [15.24] [15.26] [15.27] and [15.28]. The routine progressively creates a list of length-1024 arrays and fills them. The process stops when any of the following occurs:

1. The frequency exceeds the requested termination frequency.
2. The number of steps reaches the suggested maximum length in \*params.
3. The frequency is no longer increasing.
4. The parameter \( x > x_{\text{max}} \).
5. We run out of memory.

In the last case an error is returned; otherwise the waveform is deemed “complete”. Output arrays are created of the appropriate length and are filled with the data.

Internally, the routine keeps a list of all coefficients, as well as a list of booleans indicating which terms are nonzero. The latter allows the code to avoid lengthy floating-point operations (especially the logarithm in the post-1/2-Newtonian phase term) when these are not required.

When generating the waveform, we note that the sign of \( \dot{f} \) is the same as the sign of \( y'/x^2 = \sum(k+3)C_kx^k \), and use this series to test whether the frequency has stopped increasing. (The reason is that for waveforms far from coalescence \( \dot{f} \) is nearly constant; numerical errors can cause positive and negative fluctuations \( \Delta f \) between timesteps. The analytic formulae for \( \dot{f} \) or \( y' \) are less susceptible to this.) The coefficients \( (k+3)C_k \) are also precomputed for added efficiency.

**Warnings and suggestions:**

If no post-Newtonian parameters are provided (i.e. \texttt{params->ppn=NULL}), we generate a post2-Newtonian waveform, \textit{not} a post5/2-Newtonian waveform. This is done not only for computationally efficiency, but also because the accuracy and reliability of the post5/2-Newtonian waveform is actually worse. You can of course specify a post5/2-Newtonian waveform with an appropriate assignment of \texttt{params->ppn}, but you do so at your own risk!
This routine also performs no sanity checking on the requested sampling interval $\Delta t = \texttt{params}\rightarrow\text{deltaT}$, because this depends very much on how one intends to use the generated waveform. If you plan to generate actual wave functions $h_+ (t)$ at the same sample rate, then you will generally want a sampling interval $\Delta t < 1/2 f_{\text{max}}$; you can enforce this by specifying a suitable $\texttt{params}\rightarrow f_{\text{StopIn}}$.

However, many routines (such as those in $\texttt{SimulateCoherentGW.h}$) generate actual wave functions by linear interpolation of the amplitude and phase data, which then need only be sampled on timescales $\sim f^{-1/2}$ rather than $\sim f^{-1}$. More precisely, we would like our interpolated phase to differ from the actual phase by no more than some specified amount, say $\pi/2$ radians. The largest deviation from linear phase evolution will typically be on the order of $\Delta \phi \approx \frac{1}{2} \dot{\phi}(\Delta t/2)^2 \approx \frac{\pi}{4} \Delta f \Delta t$, where $\Delta f$ is the frequency shift over the timestep. Thus in general we would like to have

$$\Delta f \Delta t \lesssim 2$$

for our linear interpolation to be valid. This routine helps out by setting the output parameter field $\texttt{params}\rightarrow dfdt$ equal to the maximum value of $\Delta f \Delta t$ encountered during the integration.

**Uses**

- LALMalloc()
- LALFree()
- LALSCreateVectorSequence() LALSDestroyVectorSequence()
- LALCreateVector() LALSDestroyVector()
- LALDCreateVector() LALDDestroyVector()
- LALSBisectionFindRoot() LALSnprintf()

**Notes**
15.7.2 Module GeneratePPNampCorInspiral.c

Computes a parametrized post-Newtonian inspiral waveform with amplitude corrections.

Prototypes

```c
void
LALGeneratePPNampCorInspiral( LALStatus *stat,
   CoherentGW *output,
   PPNParamStruc *params )
```

Description

See GeneratePPNInspiral.c

- Phase computed to 3.5PN
- Amplitude computed to 2.5PN

Algorithm

Uses

- LALMalloc()
- LALFree()
- LALSCreateVectorSequence()
- LALSDestroyVectorSequence()
- LALSCreateVector()
- LALSDestroyVector()
- LALDCreateVector()
- LALDDestroyVector()
- LALSBisectionFindRoot()
- LALSnprintf()

Notes

Author: Creighton, T. D., McKechnan David, Van Den Broeck Chris

$Id: GeneratePPNampCorInspiral.c,v 1.27 2008/02/28 13:36:04 mckechan Exp$
15.7.3 Module GeneratePPNAmpTruncInspiral.c

Computes a parametrized post-Newtonian inspiral waveform with amplitude corrections.

Prototypes

```c
void LALGeneratePPNAmpTruncInspiral( LALStatus *stat,
                                   CoherentGW *output,
                                   PPNParamStruc *params );
```

Description

See GeneratePPNAmpTruncInspiral.c

Truncated waveform

Algorithm

Uses

- LALMalloc()
- LALFree()
- LALSCreateVectorSequence()
- LALSDestroyVectorSequence()
- LALSCreateVector()
- LALSDestroyVector()
- LALDCreateVector()
- LALDDestroyVector()
- LALSBisectionFindRoot()
- LALSnprintf()

Notes
Module GetInspiralParams.c

Computes the input parameters for a PPN inspiral.

Prototypes

```c
void LALGetInspiralParams( LALStatus *stat,
    PPNParamStruc *output,
    GalacticInspiralParamStruc *input,
    RandomParams *params )
```

Description

This function takes a Galactic location and pair of masses from `*input` and uses them to set the PPNParamStruc fields `output->position`, `output->mTot`, `output->eta`, and `output->d`. The fields `output->psi`, `output->inc`, and `output->phi` are set randomly to reflect a uniform distribution in solid angle (that is, cosine of inclination is uniform between $-1$ and $1$, other angles are uniform between $0$ and $2\pi$). The routine uses the random sequence specified by `*params` when given, but if `*params=NULL` a new sequence is started internally using the current execution time as a seed. The field `input->geocentEndTime` is ignored by this routine.

The other PPNParamStruc input fields are not touched by this routine, and must be specified externally before generating a waveform with this structure.

Algorithm

Galactocentric Galactic axial coordinates $\rho$, $z$, and $l_G$ are transformed to geocentric Galactic Cartesian coordinates:

\[
\begin{align*}
x_e &= R_e + \rho \cos l_G , \\
y_e &= \rho \sin l_G , \\
z_e &= z ,
\end{align*}
\]

where

\[
R_e \approx 8.5 \text{kpc}
\]

is the distance to the Galactic core (this constant will probably migrate into LALConstants.h eventually). These are converted to geocentric Galactic spherical coordinates:

\[
\begin{align*}
d &= \sqrt{x_e^2 + y_e^2 + z_e^2} , \\
b &= \arcsin \left( \frac{z_e}{d_e} \right) , \\
l &= \arctan2(y_e, x_e) .
\end{align*}
\]

In the calculation of $d$ we factor out the leading order term from the square root to avoid inadvertent overflow, and check for underflow in case the location lies on top of the Earth. The angular coordinates are then transformed to equatorial celestial coordinates $\alpha$ and $\delta$ using the routines in SkyCoordinates.h.

Uses

- LALGalacticToEquatorial()
- LALUniformDeviate()
- LALCreateRandomParams()
- LALDestroyRandomParams()

Notes

Author: Creighton, T. D.

$Id: GetInspiralParams.c,v 1.11 2007/06/08 14:41:47 bema Exp$

$Id: GetInspiralParams.c,v 1.11 2007/06/08 14:41:47 bema Exp$
15.7.5 Module GenerateInspiralSmooth.c

Smooths the end of an inspiral waveform by adding an exponential ringdown at the end.

Prototypes

```c
void LALGenerateInspiralSmooth( LALStatus *stat,
                               CoherentGW **output,
                               PPNParamStruc *params,
                               REAL4 *qfactor )
```

Description

This function creates a smooth ending to an inspiral waveform from GeneratePPNInspiral. It works by reading in **output and the damping factor *qfactor.

It is assumed that the **output structure is the output from LAL_GeneratePPNInspiral so that it contains amplitude, frequency and phase information in (**output)->a, (**output)->f and (**output)->phi respectively. These data is then extended by keeping the frequency fixed and exponentially damping the amplitude with damping factor qfactor.

Note: The length of the injection stored in **waveform will be correct. However, the length and time of the inspiral are not updated in the PPNParamStruc params. Therefore, they will still contain the actual end time of the inspiral part of the waveform.

Algorithm

The function reads in \( f_{\text{final}} \) and \((a_{+},x)_{\text{final}}\) then it populates additional data entries by:

\[
\begin{align*}
    f &= f_{\text{final}} \\
    a_{+},x &= (a_{+},x)_{\text{final}} \exp(-\pi f_{\text{final}} t/qfactor) \\
    \phi &= \phi_{\text{final}} + (f_{\text{final}})t
\end{align*}
\]

Here, \( t \) is the elapsed time after the end of the inspiral. The waveform ends when its amplitude has been suppressed by a factor of \( \exp(-10) \).

Uses

- LALRealloc()
- LALFree()
- LALSResizeVector()
- LALDResizeVector()
- LALSDestroyVectorSequence()
- LALSDestroyVector()

Notes
15.7.6 Program GeneratePPNInspiralTest.c

Generates a parametrized post-Newtonian inspiral waveform.

Usage

GeneratePPNInspiralTest [-m m1 m2] [-r dist] [-i inc phi] [-f fmin fmax]
[-t dt] [-w deltat] [-p order] [-d debuglevel] [-o outfile]

Description

This program generates the amplitude, phase, and frequency of a post-Newtonian inspiral waveform as functions of time. The following option flags are accepted:

- **-m** Sets the binary masses to \( m_1 \) and \( m_2 \) solar masses (default values: \( 1.4 M_\odot \)).
- **-r** Sets the binary system distance to \( dist \) kpc (default value: 8.5kpc).
- **-i** Sets the inclination and initial phase angles to \( inc \) and \( phi \) degrees (default values: 0 degrees).
- **-f** Sets the initial and final wave frequencies to \( f_{\text{min}} \) and \( f_{\text{max}} \) Hz (default values: 40Hz and 500Hz).
- **-t** Sets the waveform sampling interval to \( dt \) seconds (default value: 0.01s).
- **-w** Generates actual waveforms rather than phase and amplitude functions, sampled at intervals of \( deltat \) seconds (no default).
- **-p** Sets the post\( n/2 \)-Newtonian order to \( n = \text{order} \) (default value: \( n = 4 \)).
- **-d** Sets the debug level to \( \text{debuglevel} \) (default value: 0).
- **-o** Sets the output filename to \( \text{outfile} \) (by default no output is produced).

Exit codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NORM</td>
<td>0</td>
<td>&quot;Normal exit&quot;</td>
</tr>
<tr>
<td>SUB</td>
<td>1</td>
<td>&quot;Subroutine failed&quot;</td>
</tr>
<tr>
<td>ARG</td>
<td>2</td>
<td>&quot;Error parsing arguments&quot;</td>
</tr>
<tr>
<td>VAL</td>
<td>3</td>
<td>&quot;Input argument out of valid range&quot;</td>
</tr>
<tr>
<td>FILE</td>
<td>4</td>
<td>&quot;Could not open file&quot;</td>
</tr>
<tr>
<td>PRINT</td>
<td>5</td>
<td>&quot;Wrote past end of message string&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants \text{GENERATEPPNINSPIRALTESTC\_E}<name>, and the status descriptions in \text{GENERATEPPNINSPIRALTESTC\_MSGE}<name>. The source code with these messages is in \text{GeneratePPNInspiralTest.c} on line 1.81.

Algorithm

This program simply parses the command line, sets the appropriate fields of a \text{PPNParamStruc}, and passes it in to \text{LALGeneratePPNInspiral()}. No maximum waveform length is specified; the function will allocate as much data as necessary.

If the **-w and -o** options are given, the amplitude, phase, and frequency are generated as above, but are then resampled at intervals \( deltat \) to generate actual wave output.

Uses

\text{lalDebugLevel}
\text{LALPrintError()}
\text{LALCheckMemoryLeaks()}
\text{LALSCreateVector()}
\text{LALSDestroyVector()}
\text{LALGeneratePPNInspiral()}
\text{LALSDestroyVectorSequence()}

15.8  Header GenerateTaylorCW.h

Provides routines to generate Taylor-parameterized continuous waveforms.

Synopsis

```c
#include <lal/GenerateTaylorCW.h>
```

This header covers routines to generate continuous quasiperiodic waveforms whose frequency varies slowly and smoothly with time. For such sources the frequency function is normally described by its Taylor “spin-down” (or spin-up) coefficients. This type of waveform may be typical of objects such as neutron stars that are gradually shedding angular momentum, or are accelerating in the gravitational potential of a star cluster. The Taylor expansion is likely not suitable for long-term modelling of the frequency of waves from glitching neutron stars, neutron stars in close binary orbits, or neutron stars that are accreting or shedding angular momentum in a stochastic manner.

The frequency and phase of such slowly-varying quasiperiodic sources are given by their Taylor series:

\[
\begin{align*}
    f(t) &= f_0 \left[ 1 + \sum_{k=1}^{n} f_k (t - t_0)^k \right], \\
    \phi(t) &= \phi_0 + 2\pi f_0 \left[ (t - t_0) + \sum_{k=1}^{n} \frac{f_k}{k+1} (t - t_0)^{k+1} \right],
\end{align*}
\]

where \( f_k \) are the spin-normalized Taylor coefficients. If the source’s spin is varying over some timescale \( \tau \), one typically expects that \( f_k \sim \tau^{-k} \). Note that in this and later discussions, \( f \) and \( \phi \) refer to the frequency and phase of the gravitational wave, which are typically some constant multiple of (often twice) the frequency and phase of the rotating source.

The `CoherentGW` structure allows for a very general description of waveforms with modulations in the amplitudes or relative phases of the wave polarizations, as described in `SimulateCoherentGW.h`. However, in this simplest model of quasiperiodic waveforms, we neglect such phenomena as precession that would produce these effects. Thus for any given source one can choose a polarization basis (described by some polarization angle \( \psi \)) in which the wave has a constant elliptical polarization of the form:

\[
\begin{align*}
    h_+(t) &= A_+ \cos \phi(t), \\
    h_\times(t) &= A_\times \sin \phi(t).
\end{align*}
\]

Error conditions

<table>
<thead>
<tr>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUL 1</td>
<td>&quot;Unexpected null pointer in arguments&quot;</td>
</tr>
<tr>
<td>OUT 2</td>
<td>&quot;Output field a, f, phi, or shift already exists&quot;</td>
</tr>
<tr>
<td>MEM 3</td>
<td>&quot;Out of memory&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants `GENERATETAYLORCW_E<name>`, and the status descriptions in `GENERATETAYLORCW_MSGE<name>`. The source code with these messages is in `GenerateTaylorCW.h` on line 1.107.

Types

Structure TaylorCWParamStruc

This structure stores the parameters for constructing a gravitational waveform with a Taylor-polynomial frequency and phase. As with the `PPNParamStruc` type in `GeneratePPNInspiral.h`, we divide the fields into passed fields (which are supplied to the final `CoherentGW` structure but not used in any calculations), input fields (that are used by the waveform generator), and output fields (that are set by the waveform generator). They are:

Passed fields:

`SkyPosition position` The location of the source on the sky, normally in equatorial coordinates.
REAL4 psi  The polarization angle of the source, in radians.
LIGOTimeGPS epoch  The start time $t_0$ of the output series.

*Input fields:*

REAL8 deltaT  The requested sampling interval of the waveform, in s.
UINT4 length  The number of samples in the generated waveform.
REAL4 aPlus, aCross  The polarization amplitudes $A_+, A_\times$, in dimensionless strain units.
REAL8 phi0  The wave phase at time $t_0$, in radians.
REAL8 f0  The wave frequency at time $t_0$, in Hz.
REAL8Vector *f  The spin-normalized Taylor parameters $f_k$, as defined in Eq. 15.37 above. If $f$=NULL, a monochromatic wave is generated.

*Output fields:*

REAL4 dfdt  The maximum value of $\Delta f \Delta t$ encountered over any timestep $\Delta t$ used in generating the waveform.
15.8.1 Module GenerateTaylorCW.c

Computes a Taylor-parametrized continuous waveform.

Prototypes

```c
void LALGenerateTaylorCW( LALStatus *stat,
                        CoherentGW *output,
                        TaylorCWParamStruc *params );
```

Description

This function computes a quaiperiodic waveform using the parameters in `*params`, storing the result in `*output`.

In the `*params` structure, the routine uses all the “input” fields specified in `GenerateTaylorCW.h`, and sets all of the “output” fields. If `params->f=NULL`, a precisely periodic (monochromatic) waveform is generated.

In the `*output` structure, the field `output->h` is ignored, but all other pointer fields must be set to `NULL`. The function will create and allocate space for `output->a`, `output->f`, and `output->phi` as necessary. The `output->shift` field will remain set to `NULL`.

Algorithm

This function is a fairly straightforward calculation of Eqs. 15.37 and 15.38 in `GenerateTaylorCW.h`. There are no real tricks involved, except to note that the phase $\phi$ and the time elapsed $t - t_0$ are computed and stored as `REAL8s` in order to provide waveforms that are accurate to small fractions of a cycle over many years.

Since the waveform does not include any effects such as precession, the amplitudes $A_+, A_\times$ and the shift angle $\Phi$, as defined in `SimulateCoherentGW.h`, are constant. This is dealt with by setting `output->a` to be a `REAL4TimeVectorSequence` of two identical vectors $(A_+, A_\times)$ spanning the requested time of the waveform, under the assumption that any routine using this output data (such as the routines in `SimulateCoherentGW.h`) will interpolate these two endpoints to get the instantaneous values of $A_+, A_\times$. The field `output->shift` is left as `NULL`, so the constant value of $\Phi$ must be subsumed into the polarization angle $\psi$.

The fields `output->f` and `output->phi` are created and filled at the requested sampling interval `params->deltaT`; it is up to the calling routine to ensure that this sampling interval is reasonable. As a guideline, we want to be able to determine the instantaneous wave phase accurately to within a fraction of a cycle. For functions that compute the phase by linear interpolation of `output->phi`, this means sampling on timescales $\Delta t \lesssim f^{-1/2} \sim \max\{\sqrt{\kappa f_0 k^2}T^k}\}^{-1}$, where $T$ is the duration of the waveform. More precisely, the largest deviation from linear phase evolution will typically be on the order of $\Delta \phi \approx (1/2)\Delta \phi (\Delta t/2)^2 \approx (\pi/4)\Delta f \Delta t$, where $\Delta f$ is the frequency shift over the timestep. So if we want our interpolated phase to agree with the true phase to within, say, $\pi/2$ radians, then we would like to have

$$\Delta f \Delta t \lesssim 2.$$  

This routine provides a check by setting the output parameter field `params->dfdt` equal to the maximum value of $\Delta f \Delta t$ encountered during the integration.

Uses

- `LALMalloc()`  
- `LALFree()`  
- `LALSCreateVectorSequence()`  
- `LALSDestroyVectorSequence()`  
- `LALSCreateVector()`  
- `LALSDestroyVector()`  
- `LALDCreateVector()`  
- `LALDDestroyVector()`  
- `LALSnprintf()`

Notes

Author: Creighton, T. D.

$Id: GenerateTaylorCW.c,v 1.6 2007/06/08 14:41:47 bema Exp$
15.8.2 Program SimulateTaylorCWTest.c

Generates a quasiperiodic waveform.

Usage

SimulateTaylorCWTest [-s sourcefile] [-r respfile] [-l site earthfile sunfile]
                   [-o outfile] [-t sec nsec npt dt] [-h hsec hnsec fh]
                   [-d debuglevel]

Description

This program generates a Taylor-parameterized continuous waveform as a function of time. The following option flags are accepted:

- **s** Reads source information from the file `sourcefile`, whose format is specified below. If absent, it injects a monochromatic wave with intrinsic frequency 100 Hz, strain amplitude 1000, and zero phase at the GPS epoch, arriving from the direction RA=00h00m, dec=0°.

- **r** Reads a detector response function from the file `respfile`, whose format is specified below. If absent, it generates raw dimensionless strain.

- **l** Sets the detector location and orientation. `site` must be one of the following character strings: `LHO`, `LLO`, `VIRGO`, `GEO600`, `TAMA300`, or `CIT40`. `earthfile` and `sunfile` are ephemeris files of the format expected by `LALInitBarycenter()`. If the `-l` option is not specified, a stationary (barycentric) detector aligned with the wave’s + polarization is assumed.

- **o** Writes the generated time series to the file `outfile`. If absent, the routines are exercised, but no output is written.

- **t** Sets the timing information for the generated waveform: `sec` and `nsec` are integers specifying the start time in GPS seconds and nanoseconds, `npt` is the number of time samples generated, and `dt` is the sampling interval in seconds. If absent, `-t 0 0 65536 9.765625e-4` is assumed.

- **h** Performs “ideal heterodyning” (phase subtraction) on the signal: `hsec` and `hnsec` are integers specifying an epoch of zero phase subtraction in GPS seconds and nanoseconds, and `fh` is the frequency of phase subtraction in Hz. If absent, no heterodyning is performed.

- **d** Sets the debug level to `debuglevel`. If absent, level 0 is assumed.

Format for `sourcefile`: The source file consists of any number of lines of data, each specifying a continuous waveform. Each line consists of a GPS epoch where the frequency, phase, and Taylor coefficients are defined (INT8 nanoseconds), followed by 7 or more whitespace-delimited REAL8 numbers: the + and × wave amplitudes (dimensionless strain) and polarization angle ψ (degrees), the right ascension and declination (degrees), the initial phase (degrees) and frequency (Hz), followed by zero or more Taylor coefficients $f_k$ (Hz$^k$). Note that the wave amplitudes and polarization angle are read as REAL8, but are later cast to REAL4.

Format for `respfile`: The response function $R(f)$ gives the real and imaginary components of the transformation from ADC output $o$ to tidal strain $h$ via $h(f) = R(f)\bar{o}(f)$. It is inverted internally to give the detector transfer function $T(f) = 1/R(f)$. The format `respfile` is a header specifying the GPS epoch $t_0$ at which the response was taken (INT8 nanoseconds), the lowest frequency $f_0$ at which the response is given (REAL8 Hz), and the frequency sampling interval $\Delta f$ (REAL8 Hz):

```plaintext
# epoch = t0
# f0 = f0
# deltaF = deltaF
```

followed by two columns of REAL4 data giving the real and imaginary components of $R(f_0 + k\Delta f)$.
Format for outfile: The output files generated by this program consist of a two-line header giving the GPS epoch $t_0$ of the first time sample (INT8 nanoseconds) and the sampling interval $\Delta t$ (REAL8 seconds):

\[
\# \text{ epoch } = t_0 \\
\# \text{ deltaT } = \Delta t
\]

followed by a single column of REAL4 data representing the undigitized output of the interferometer's gravitational-wave channel.

Exit codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NORM</td>
<td>0</td>
<td>&quot;Normal exit&quot;</td>
</tr>
<tr>
<td>SUB</td>
<td>1</td>
<td>&quot;Subroutine failed&quot;</td>
</tr>
<tr>
<td>ARG</td>
<td>2</td>
<td>&quot;Error parsing arguments&quot;</td>
</tr>
<tr>
<td>VAL</td>
<td>3</td>
<td>&quot;Input argument out of valid range&quot;</td>
</tr>
<tr>
<td>FILE</td>
<td>4</td>
<td>&quot;Could not open file&quot;</td>
</tr>
<tr>
<td>INPUT</td>
<td>5</td>
<td>&quot;Error reading file&quot;</td>
</tr>
<tr>
<td>MEM</td>
<td>6</td>
<td>&quot;Out of memory&quot;</td>
</tr>
<tr>
<td>PRINT</td>
<td>7</td>
<td>&quot;Wrote past end of message string&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants `SIMULATETAYLORCWTESTC_E<name>` and the status descriptions in `SIMULATETAYLORCWTESTC_MSGE<name>`. The source code with these messages is in `SimulateTaylorCWTest.c` on line 1.144.

Algorithm

If a `sourcefile` is specified, this program first reads the epoch field, and then calls `LALSReadVector()` to read the remaining fields. (If no `sourcefile` is specified, the parameters are taken from `#define` constants.) The arguments are passed to `LALGenerateTaylorCW()` to generate frequency and phase timeseries. The required sampling resolution for these timeseries is estimated at $0.1/\sum_k \sqrt{|k f_0 f_k T_k - 1|}$, where $T$ is the duration of the waveform, to ensure that later interpolation of the waveforms will give phases accurate to well within a radian.

The output from `LALGenerateTaylorCW()` is then passed to `LALSimulateCoherentGW()` to generate an output time series. If there are multiple lines in `sourcefile`, the procedure is repeated and the new waveforms added into the output. A warning is generated if the wave frequency exceeds the Nyquist rate for the output time series.

Uses

- `lalDebugLevel`
- `LALPrintError()`  `LALCheckMemoryLeaks()`
- `LALSReadVector()`  `LALSDestroyVector()`
- `LALGenerateTaylorCW()`  `LALSDestroyVectorSequence()`

Notes
15.9 Header `GenerateSpinOrbitCW.h`

Provides routines to generate continuous waveforms with spindown and orbital modulation.

**Synopsis**

```c
#include <lal/GenerateSpinOrbitCW.h>
```

This header covers routines to generate continuous quasiperiodic waveforms with a smoothly-varying intrinsic frequency modulated by orbital motions around a binary companion. The intrinsic frequency is modeled by Taylor series coefficients as in `GenerateTaylorCW.h`, and the orbital modulation is described by a reduced set of orbital parameters. Note that the routines do not account for spin precession, accretion processes, or other complicating factors; they simply Doppler-modulate a polynomial frequency function.

The frequency and phase of the wave in the source’s rest frame are given by Eqs. (15.37) and (15.38) of `GenerateTaylorCW.h`, where $t$ is the proper time in this rest frame. The frequency and phase of the wave fronts crossing a reference point in an inertial frame (e.g. the Solar system barycentre) are simply $f[t(t_r)]$ and $\phi[t(t_r)]$, where

$$t_r = t + R(t)/c$$

(15.41)

is the (retarded) time measured at the inertial reference point a distance $r$ from the source.

The generation of the waveform thus consists of computing the radial component $R(t)$ of the orbital motion of the source in the binary centre-of-mass frame, inverting Eq. (15.41) to find the “emission time” $t$, and plugging this into the Taylor expansions to generate the instantaneous frequency and phase. The received frequency is also multiplied by the instantaneous Doppler shift $[1 + \dot{R}(t)/c]^{-1}$ at the time of emission.

Since we do not include precession effects, the polarization state of the wave is constant: we simply specify the polarization amplitudes $A_+, A_\times$ and the polarization phase $\psi$ based on the (constant) orientation of the source’s rotation. The following discussion defines a set of parameters for the source’s orbital revolution, which we regard as completely independent from its rotation.

**Orbital motion**

Fig. 15.2 illustrates the notation conventions defining a binary orbit. We define a radial axis $R$ directed from the observer (Earth) to the source, as shown. The horizontal plane is thus the plane of the sky, and the direction marked $N$ is the direction along a meridian towards the North celestial pole. The tilted plane is the plane of the binary orbit, and the axis labeled $z$ is the normal to this plane directed such that the orbit is right-handed about this axis. The ascending node of the orbit, denoted by $\mathbf{\Omega}$, is the direction defined by $\mathbf{R} \times \mathbf{\hat{z}}$. The binary orbit itself is shown as an off-centred ellipse, with the barycentre at one of its foci; the wave-emitting source is also shown.

The inclination angle $i$ is the angle between the sky and orbital planes. The longitude of the ascending node $\Omega$ is the angle in the plane of the sky from the North direction to the ascending node, measured right-handed about $\mathbf{R}$. The argument of the periapsis $\omega$ is the angle in the orbital plane from the ascending node to the direction of periapsis (point where the source is closest to the system barycentre), and the true anomaly $v(t)$ of the source is the angle from the periapsis to the current location of the source; both angles are measured right-handed about $\mathbf{\hat{z}}$ (i.e. prograde). The periapsis separation $r_p$ is the distance from the periapsis to the barycentre, and we denote the eccentricity of the orbital ellipse as $e$, so that the separation between the source and the barycentre at any time is $r = r_p(1 + e)/(1 + e \cos v)$.

In this convention, $i \in [0, \pi]$ and $\Omega \in [0, 2\pi)$. Another convention common in astronomy is to restrict $\Omega$ to the range $[0, \pi)$, referring to whichever node (ascending or descending) lies in this range. The argument
of the periaps $\omega$ is then also measured from this node. In this case the range of $i$ must be extended to $(−\pi, \pi]$; it is negative if the reference node is descending, and positive if it is ascending. The formulae that follow are the same in either convention, though, since one can verify that adding $\pi$ to $\Omega$ and $\omega$ is equivalent to reversing the sign on $i$.

Some spherical trigonometry gives us $R = r \sin(\omega + v) \sin i$. We can differentiate $R$ with respect to $t$, and apply Kepler's second law $r^2 \dot{v} = r^2 \dot{v}_p = \text{constant}$, where $\dot{v}_p$ is the angular speed at periapsis, to get:

$$R = R_0 + \frac{(1 + e) r_p \sin i}{1 + e \cos v} \sin(\omega + v),$$

$$\dot{R} = \dot{R}_0 + \frac{\dot{v}_p r_p \sin i}{1 + e} [\cos(\omega + v) + e \cos \omega].$$

Without loss of generality, we will henceforth drop the offsets $R_0$ and (constant) $\dot{R}_0$ from these equations. This means that we ignore the overall propagation delay between the $R = R_0$ plane and the observer, and incorporate any (constant) Doppler shifts due to net centre-of-mass motions into the values of $f$ and $\dot{v}_p$. The resulting times and parameter values are referred to as being in the barycentric frame. The only time delays and Doppler shifts that we explicitly treat are those arising from the motion of the source relative to the $R = R_0$ sky plane passing through the system barycentre.

All we need now to determine the orbital motion is an equation for $v(t)$. Many basic astronomy textbooks give exact but transcendental expressions relating $v$ and $t$ for elliptical orbits with $0 \leq e < 1$, and/or series expansions of $v(t)$ for $e \ll 1$. However, for a generic binary system we cannot guarantee that $e \ll 1$, and for now we would like to retain the possibility of modeling open orbits with $e \geq 1$. For now we will simply present the exact formulae, and discuss the numerical solution methods in the modules under this header.

Let $t_p$ be the time of a periapsis passage (preferably a recent one in the case of closed orbits). We express both $t$ and $v$ in terms of an intermediate variable $E$ (called the eccentric anomaly for elliptic orbits, unnamed for open orbits). The formulae are:

$$t - t_p = \begin{cases} \frac{1}{v_p} \sqrt{\frac{1 + e}{1 - e^2}} (E - e \sin E) & 0 \leq e < 1 \\ \frac{1}{v_p} E \left(1 + \frac{e^2}{T^2} \right) & e = 1 \\ \frac{1}{v_p} \sqrt{\frac{e + 1}{e - 1}} (e \sinh E - E) & e > 1 \end{cases}$$

$$\tan \left(\frac{v}{2}\right) = \begin{cases} \frac{E}{\sqrt{1 - e}} \tan \left(\frac{E}{T}\right) & 0 \leq e < 1 \\ \frac{e + 1}{\sqrt{e - 1}} \tanh \left(\frac{E}{T}\right) & e > 1 \end{cases}$$

Thus to solve for $v(t)$ one typically inverts the equation for $t - t_p$ numerically or by series expansion, finds the corresponding $E$, and then plugs this into the expression for $v$. However, in our case we would then need to do another numerical inversion to find the retarded time $t_r$ from Eq. (15.44). A more efficient approach is thus to take an initial guess for $E$, compute both $t$, $v$, and hence $t_r$, and then refine directly on $E$.

Other notation conventions

Since we may deal with highly eccentric or open orbits, we will specify these orbits with parameters that are definable for all classes of orbit. Thus we specify the size of the orbit with the periaps separation $r_p$ rather than the semimajor axis $a$, and the speed of the orbit with the angular speed at periaps $\dot{v}_p$ rather than with the period $P$. These parameters are related by:

$$a = \frac{r_p}{1 - e},$$

$$P = \frac{2\pi}{\dot{v}_p} \sqrt{\frac{1 + e}{(1 - e)^3}}.$$

Furthermore, for improved numerical precision when dealing with near-parabolic orbits, we specify the value of $1 - e$ rather than the value of $e$. We note that $1 - e$ has a maximum value of 1 for a circular orbit, positive for closed elliptical orbits, zero for parabolic orbits, and negative (unbounded) for hyperbolic orbits.
### Error conditions

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUL</td>
<td>1</td>
<td>&quot;Unexpected null pointer in arguments&quot;</td>
</tr>
<tr>
<td>OUT</td>
<td>2</td>
<td>&quot;Output field a, f, phi, or shift already exists&quot;</td>
</tr>
<tr>
<td>MEM</td>
<td>3</td>
<td>&quot;Out of memory&quot;</td>
</tr>
<tr>
<td>ECC</td>
<td>4</td>
<td>&quot;Eccentricity out of range&quot;</td>
</tr>
<tr>
<td>FTL</td>
<td>5</td>
<td>&quot;Periapsis motion is faster than light&quot;</td>
</tr>
<tr>
<td>SGN</td>
<td>6</td>
<td>&quot;Sign error: positive parameter expected&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants `GENERATESPINORBITCW_E<name>`, and the status descriptions in `GENERATESPINORBITCW_MSGE<name>`. The source code with these messages is in `GenerateSpinOrbitCW.h` on line 1.252.

### Types

**Structure SpinOrbitCWParamStruc**

This structure stores the parameters for constructing a gravitational waveform with both a Taylor-polynomial intrinsic frequency and phase, and a binary-orbit modulation. As with the `PPNParamStruc` type in `GeneratePPNInspiral.h`, we divide the fields into passed fields (which are supplied to the final `CoherentGW` structure but not used in any calculations), input fields (that are used by the waveform generator), and output fields (that are set by the waveform generator). They are:

**Passed fields:**

- **SkyPosition position** The location of the source on the sky, normally in equatorial coordinates.
- **REAL4 psi** The polarization angle of the source, in radians.

**Input fields:**

- **LIGOTimeGPS epoch** The start time of the output series.
- **LIGOTimeGPS spinEpoch** A reference time \( t_{\text{ref}} \) (in the barycentric frame) at which the rotational properties of the source are specified.
- **LIGOTimeGPS orbitEpoch** A time \( t_{\text{peri}} \) (in the barycentric frame) at which the source passes through periapsis. Note that this is the proper or “true” time of passage; the observed periapsis passage occurs at time \( t_{\text{peri}} + r(t_{\text{peri}})/c \).
- **REAL8 deltaT** The requested sampling interval of the waveform, in s.
- **UINT4 length** The number of samples in the generated waveform.
- **REAL4 aPlus, aCross** The polarization amplitudes \( A_+, A_\times \), in dimensionless strain units.
- **REAL8 phi0** The phase of the wave emitted at time \( t_{\text{ref}} \), in radians.
- **REAL8 f0** The frequency of the wave emitted at time \( t_{\text{ref}} \) (and incorporating any Doppler shift due to \( \dot{R}_0 \)), in Hz.
- **REAL8Vector *f** The spin-normalized Taylor parameters \( f_k \), as defined in Eq. 15.37 of `GenerateTaylorCW.h`. If \( f = \text{NULL} \), the (proper) spin of the source is assumed to be constant.
- **REAL8 omega** The argument of the periapsis, \( \omega \), in radians.
- **REAL8 rPeriNorm** The projected, speed-of-light-normalized periapsis separation of the orbit, \( (r_p/c) \sin i \), in s.
- **REAL8 oneMinusEcc** The value of \( 1 - e \).
- **REAL8 angularSpeed** The angular speed at periapsis, \( \dot{\upsilon}_p \), in Hz.

**Output fields:**
**REAL4 dfdt** The maximum value of $\Delta f \Delta t$ encountered over any timestep $\Delta t$ used in generating the waveform.
15.9. Module `GenerateSpinOrbitCW.c`

Computes a spindown- and Doppler-modulated continuous waveform.

Prototypes

```c
void LALGenerateSpinOrbitCW( LALStatus *stat,
                            CoherentGW *output,
                            SpinOrbitCWParamStruc *params )
```

Description

This function computes a quaperiodic waveform using the spindown and orbital parameters in `*params`, storing the result in `*output`.

In the `*params` structure, the routine uses all the “input” fields specified in `GenerateSpinOrbitCW.h`, and sets all of the “output” fields. If `params->f=NULL`, no spindown modulation is performed. If `params->rPeriNorm=0`, no Doppler modulation is performed.

In the `*output` structure, the field `output->h` is ignored, but all other pointer fields must be set to `NULL`. The function will create and allocate space for `output->a`, `output->f`, and `output->phi` as necessary. The `output->shift` field will remain set to `NULL`.

Algorithm

This routine calls `LALGenerateCircularSpinOrbitCW()`, `LALGenerateCircularSpinOrbitCW()`, `LALGenerateCircularSpinOrbitCW()`, or `LALGenerateCircularSpinOrbitCW()`, depending on the value of `params->oneMinusEcc`. See the other modules under `GenerateSpinOrbitCW.h` for descriptions of these routines’ algorithms.

If `params->rPeriNorm=0`, this routine will call `LALGenerateTaylorCW()` to generate the waveform.

It creates a `TaylorCWParamStruc` from the values in `*params`, adjusting the values of `params->phi0`, `params->f0`, and `params->f` from the reference time `params->spinEpoch` to the time `params->epoch`, as follows: Let \( \Delta t = t^{(2)} - t^{(1)} \) be the time between the old epoch \( t^{(1)} \) and the new one \( t^{(2)} \). Then the phase, base frequency, and spindown parameters for the new epoch are:

\[
\phi^{(2)}_0 = \phi^{(1)}_0 + 2\pi f^{(1)}_0 t \left( 1 + \sum_{k=1}^{N} \frac{1}{k+1} f^{(1)}_k \Delta t^k \right)
\]
\[
f^{(2)}_0 = f^{(1)}_0 \left( 1 + \sum_{k=1}^{N} f^{(1)}_k \Delta t^k \right)
\]
\[
f^{(2)}_k = \frac{f^{(1)}_0}{f^{(2)}_0} \left( f^{(1)}_k + \sum_{j=k+1}^{N} \binom{j}{k} f^{(1)}_j \Delta t^{j-k} \right)
\]

The phase function \( \phi(t) = \phi^{(i)}_0 + 2\pi f^{(i)}_0 \left[ t - t^{(i)} + \sum_{k=1}^{N} \frac{f^{(i)}_k}{k+1} (t - t^{(i)})^{k+1} \right] \) then has the same functional dependence on \( t \) for either \( i = 1 \) or \( 2 \).

Uses

- `LALDCreateVector()`
- `LALDDestroyVector()`
- `LALGenerateCircularSpinOrbitCW()`
- `LALGenerateEllipticSpinOrbitCW()`
- `LALGenerateTaylorCW()`

Notes

Author: Creighton, T. D.

$Id: GenerateSpinOrbitCW.c,v 1.5 2007/06/08 14:41:47 bema Exp$
15.9.2 Module GenerateEllipticSpinOrbitCW.c

Computes a continuous waveform with frequency drift and Doppler modulation from an elliptical orbital trajectory.

Prototypes

```c
void LALGenerateEllipticSpinOrbitCW( LALStatus *stat,
                                      CoherentGW *output,
                                      SpinOrbitCWParamStruc *params );
```

Description

This function computes a quaiperiodic waveform using the spindown and orbital parameters in `*params`, storing the result in `*output`.

In the `*params` structure, the routine uses all the “input” fields specified in `GenerateSpinOrbitCW.h`, and sets all of the “output” fields. If `params->f=NULL`, no spindown modulation is performed. If `params->oneMinusEcc ∈ (0,1]` (an open orbit), or if `params->rPeriNorm × params->angularSpeed ≥ 1` (faster-than-light speed at periapsis), an error is returned.

In the `*output` structure, the field `output->h` is ignored, but all other pointer fields must be set to NULL. The function will create and allocate space for `output->a`, `output->f`, and `output->phi` as necessary. The `output->shift` field will remain set to NULL.

Algorithm

For elliptical orbits, we combine Eqs. (15.41), (15.44), and (15.45) to get \( t_r \) directly as a function of the eccentric anomaly \( E \):

\[
t_r = t_p + \left( \frac{r_p \sin i}{c} \right) \sin \omega \\
+ \left( \frac{P}{2\pi} \right) \left[ E + [v_p(1-e)\cos \omega - e] \sin E + \left[ v_p \sqrt{\frac{1-e}{1+e}} \sin \omega \right] \left[ \cos E - 1 \right] \right],
\]

where \( v_p \) is a normalized velocity at periapsis and \( P = 2\pi \sqrt{(1+e)/(1-e)}/\dot{v}_p \) is the period of the orbit. For simplicity we write this as:

\[
t_r = T_p + \frac{1}{n} \left( E + A \sin E + B [\cos E - 1] \right),
\]

where \( T_p \) is the observed time of periapsis passage and \( n = 2\pi/P \) is the mean angular speed around the orbit. Thus the key numerical procedure in this routine is to invert the expression \( x = E + A \sin E + B [\cos E - 1] \) to get \( E(x) \). We note that \( E(x+2n\pi) = E(x) + 2n\pi \), so we only need to solve this expression in the interval \([0,2\pi]\), sketched to the right.

We further note that \( A^2 + B^2 \leq 1 \), although it approaches 1 when \( e \to 1 \), or when \( v_p \to 1 \) and either \( e = 0 \) or \( \omega = \pi \). Except in this limit, Newton-Raphson methods will converge rapidly for any initial guess. In this limit, though, the slope \( dx/dE \) approaches zero at the point of inflection, and an initial guess or iteration landing near this point will send the next iteration off to unacceptably large or small values. However, by restricting all initial guesses and iterations to the domain \( E \in [0,2\pi] \), one will always end up on a trajectory branch that will converge uniformly. This should converge faster than the more generically robust technique of bisection.

In this algorithm, we start the computation with an arbitrary initial guess of \( E = 0 \), and refine it until the we get agreement to within 0.01 parts in part in \( N_{\text{cyc}} \) (where \( N_{\text{cyc}} \) is the larger of the number of wave cycles in an orbital period, or the number of wave cycles in the entire waveform being generated), or one part in \( 10^{15} \) (an order of magnitude off the best precision possible with REAL8 numbers). The latter case indicates that REAL8 precision may fail to give accurate phasing, and one should consider modeling the orbit as a set of Taylor frequency coefficients \( \text{a la LALGenerateTaylorCW()} \). On subsequent timesteps, we use the
previous timestep as an initial guess, which is good so long as the timesteps are much smaller than an orbital period. This sequence of guesses will have to readjust itself once every orbit (as \( E \) jumps from \( 2\pi \) down to 0), but this is relatively infrequent; we don’t bother trying to smooth this out because the additional tests would probably slow down the algorithm overall.

Once a value of \( E \) is found for a given timestep in the output series, we compute the system time \( t \) via Eq. (15.44), and use it to determine the wave phase and (non-Doppler-shifted) frequency via Eqs. (15.37) and (15.38). The Doppler shift on the frequency is then computed using Eqs. (15.45) and (15.43). We use \( v \) as an intermediate in the Doppler shift calculations, since expressing \( \dot{R} \) directly in terms of \( E \) results in expression of the form \((1 - e)/(1 - e \cos E)\), which are difficult to simplify and face precision losses when \( E \sim 0 \) and \( e \rightarrow 1 \). By contrast, solving for \( v \) is numerically stable provided that the system \texttt{atan2()} function is well-designed.

The routine does not account for variations in special relativistic or gravitational time dilation due to the elliptical orbit, nor does it deal with other gravitational effects such as Shapiro delay. To a very rough approximation, the amount of phase error induced by gravitational redshift goes something like \( \Delta \phi \sim fT(v/c)^2 \Delta (r_p/r) \), where \( f \) is the typical wave frequency, \( T \) is either the length of data or the orbital period (whichever is smaller), \( v \) is the true (unprojected) speed at periapsis, and \( \Delta (r_p/r) \) is the total range swept out by the quantity \( r_p/r \) over the course of the observation. Other relativistic effects such as special relativistic time dilation are comparable in magnitude. We make a crude estimate of when this is significant by noting that \( v/c \gtrsim v_p \) but \( \Delta (r_p/r) \gtrsim 2e/(1 + e) \); we take these approximations as equalities and require that \( \Delta \phi \lesssim \pi \), giving:

\[
f_0Tv_p^2 \frac{4e}{1 + e} \lesssim 1. \tag{15.50}
\]

When this criterion is violated, a warning is generated. Furthermore, as noted earlier, when \( v_p \geq 1 \) the routine will return an error, as faster-than-light speeds can cause the emission and reception times to be non-monotonic functions of one another.

Uses

\begin{verbatim}
LALMalloc() LALFree()
LALSCreateVectorSequence() LALSDestroyVectorSequence()
LALSCreateVector() LALSDestroyVector()
LALDCreateVector() LALDDestroyVector()
LALSnprintf() LALWarning()
\end{verbatim}

Notes
15.9.3 Module GenerateParabolicSpinOrbitCW.c

Computes a continuous waveform with frequency drift and Doppler modulation from a parabolic orbital trajectory.

Prototypes

```c
void LALGenerateParabolicSpinOrbitCW( LALStatus *stat, CoherentGW *output, SpinOrbitCWParamStruc *params )
```

Description

This function computes a quaiperiodic waveform using the spindown and orbital parameters in *params, storing the result in *output.

In the *params structure, the routine uses all the “input” fields specified in GenerateSpinOrbitCW.h, and sets all of the “output” fields. If params->f=NULL, no spindown modulation is performed. If params->oneMinusEcc≠0, or if params->rPeriNorm×params->angularSpeed≥1 (faster-than-light speed at periapsis), an error is returned.

In the *output structure, the field output->h is ignored, but all other pointer fields must be set to NULL. The function will create and allocate space for output->a, output->f, and output->phi as necessary. The output->shift field will remain set to NULL.

Algorithm

For parabolic orbits, we combine Eqs. (15.41), (15.44), and (15.45) to get \( t_r \) directly as a function of \( E \):

\[
    t_r = t_p + \frac{r_p \sin i}{c} \left[ \cos \omega + \left( \frac{1}{v_p} + \cos \omega \right) E - \frac{\sin \omega}{4} E^2 + \frac{1}{12v_p} E^3 \right],
\]

where \( v_p = r_p \dot{\nu}_p \sin i/c \) is a normalized velocity at periapsis. Following the prescription for the general analytic solution to the real cubic equation, we substitute \( E = x + 3v_p \sin \omega \) to obtain:

\[
    x^3 + px = q,
\]

where:

\[
    p = 12 + 12v_p \cos \omega - 3v_p^2 \sin^2 \omega,
\]

\[
    q = 12v_p^2 \sin \omega \cos \omega - 24v_p \sin \omega + 2v_p^3 \sin^3 \omega + 12v_p(t_r - t_p).
\]

We note that \( p > 0 \) is guaranteed as long as \( v_p < 1 \), so the right-hand side of Eq. 15.52 is monotonic in \( x \) and has exactly one root. However, \( p \to 0 \) in the limit \( v_p \to 1 \) and \( \omega = \pi \). This may cause some loss of precision in subsequent calculations. But \( v_p \sim 1 \) means that our solution will be inaccurate anyway because we ignore significant relativistic effects.

Since \( p > 0 \), we can substitute \( x = y \sqrt{3/4p} \) to obtain:

\[
    4y^3 + 3y = \frac{q}{2} \left( \frac{3}{p} \right)^{3/2} \equiv C.
\]

Using the triple-angle hyperbolic identity \( \sinh(3\theta) = 4 \sinh^3 \theta + 3 \sinh \theta \), we have \( y = \sinh \left( \frac{1}{4} \sinh^{-1} C \right) \). The solution to the original cubic equation is then:

\[
    E = 3v_p \sin \omega + 2 \sqrt{\frac{p}{3}} \sinh \left( \frac{1}{4} \sinh^{-1} C \right).
\]

To ease the calculation of \( E \), we precompute the constant part \( E_0 = 3v_p \sin \omega \) and the coefficient \( \Delta E = 2 \sqrt{p/3} \). Similarly for \( C \), we precompute a constant piece \( C_0 \) evaluated at the epoch of the output time series,
and a stepsize coefficient $\Delta C = 6(p/3)^{3/2} \dot{\nu}_p \Delta t$, where $\Delta t$ is the step size in the (output) time series in $t_r$. Thus at any timestep $i$, we obtain $C$ and hence $E$ via:

\[
C = C_0 + i \Delta C, \\
E = E_0 + \Delta E \times \begin{cases} 
\sinh \left[ \frac{1}{3} \ln \left( C + \sqrt{C^2 + 1} \right) \right], & C \geq 0, \\
\sinh \left[ -\frac{1}{3} \ln \left( -C + \sqrt{C^2 + 1} \right) \right], & C \leq 0,
\end{cases}
\]

where we have explicitly written $\sinh^{-1}$ in terms of functions in `math.h`. Once $E$ is found, we can compute $t = E(12 + E^2)/(12 \dot{\nu}_p)$ (where again $1/12 \dot{\nu}_p$ can be precomputed), and hence $f$ and $\phi$ via Eqs. (15.37) and (15.38). The frequency $f$ must then be divided by the Doppler factor:

\[
1 + \frac{\dot{R}}{c} = 1 + \frac{\dot{\nu}_p}{4 + E^2} (4 \cos \omega - 2E \sin \omega)
\]

(once again $4 \cos \omega$ and $2 \sin \omega$ can be precomputed).

This routine does not account for relativistic timing variations, and issues warnings or errors based on the criteria of Eq. (15.50) in `GenerateEllipticSpinOrbitCW.c`. The routine will also warn if it seems likely that `REALS` precision may not be sufficient to track the orbit accurately. We estimate that numerical errors could cause the number of computed wave cycles to vary by

\[
\Delta N \lesssim f_0 T \varepsilon \left[ 6 + \ln \left( |C| + \sqrt{|C|^2 + 1} \right) \right],
\]

where $|C|$ is the maximum magnitude of the variable $C$ over the course of the computation, $f_0 T$ is the approximate total number of wave cycles over the computation, and $\varepsilon \approx 2 \times 10^{-16}$ is the fractional precision of `REALS` arithmetic. If this estimate exceeds 0.01 cycles, a warning is issued.

Uses

- LALMalloc()
- LALFree()
- LALSCreateVectorSequence()
- LALSDestroyVectorSequence()
- LALSCreateVector()
- LALSDestroyVector()
- LALDCreateVector()
- LALDDestroyVector()
- LALSnprintf()
- LALWarning()

Notes
15.9.4 Module *GenerateHyperbolicSpinOrbitCW.c*

Computes a continuous waveform with frequency drift and Doppler modulation from a hyperbolic orbital trajectory.

**Prototypes**

```c
void LALGenerateHyperbolicSpinOrbitCW( LALStatus *stat,
                                     CoherentGW *output,
                                     SpinOrbitCWParamStruc *params )
```

**Description**

This function computes a quasiperiodic waveform using the spindown and orbital parameters in `*params`, storing the result in `*output`.

In the `*params` structure, the routine uses all the “input” fields specified in `GenerateSpinOrbitCW.h`, and sets all of the “output” fields. If `params->f=NULL`, no spindown modulation is performed. If `params->oneMinusEcc<0` (a non-hyperbolic orbit), or if `params->rPeriNorm×params->angularSpeed≥1` (faster-than-light speed at periapsis), an error is returned.

In the `*output` structure, the field `output->h` is ignored, but all other pointer fields must be set to `NULL`.

The function will create and allocate space for `output->a`, `output->f`, and `output->phi` as necessary. The `output->shift` field will remain set to `NULL`.

**Algorithm**

For hyperbolic orbits, we combine Eqs. (15.44), (15.44), and (15.45) to get \( t_r \) directly as a function of \( E \):

\[
\begin{align*}
t_r &= t_p + \frac{r_p \sin \frac{i}{c}}{\sin \omega} \\
&+ \frac{1}{n} \left( -E + [v_p(e - 1) \cos \omega + e] \sinh E - \left[ v_p \sqrt{\frac{e - 1}{e + 1}} \sin \omega \right] \cosh E - 1 \right),
\end{align*}
\]

where \( v_p = r_p \dot{r}_p \sin \frac{i}{c} \) is a normalized velocity at periapsis and \( n = v_p \sqrt{(1 - e)^3/(1 + e)} \) is a normalized angular speed for the orbit (the hyperbolic analogue of the mean angular speed for closed orbits). For simplicity we write this as:

\[
t_r = T_p + \frac{1}{n} \left( E + A \sinh E + B[\cosh E - 1] \right),
\]

where \( T_p \) is the observed time of periapsis passage. Thus the key numerical procedure in this routine is to invert the expression \( x = E + A \sinh E + B[\cosh E - 1] \) to get \( E(x) \). This function is sketched to the right (solid line), along with an approximation used for making an initial guess (dotted line), as described later.

We note that \( A^2 - B^2 < 1 \), although it approaches 1 when \( e \to 1 \), or when \( v_p \to 1 \) and either \( e = 0 \) or \( \omega = \pi \). Except in this limit, Newton-Raphson methods will converge rapidly for any initial guess. In this limit, though, the slope \( dx/dE \) approaches zero at \( E = 0 \), and an initial guess or iteration landing near this point will send the next iteration off to unacceptably large or small values. A hybrid root-finding strategy is used to deal with this, and with the exponential behaviour of \( x \) at large \( E \).

First, we compute \( x = x_{\pm 1} \) at \( E = \pm 1 \). If the desired \( x \) lies in this range, we use a straightforward Newton-Raphson root finder, with the constraint that all guesses of \( E \) are restricted to the domain \([-1,1]\). This guarantees that the scheme will eventually find itself on a uniformly-convergent trajectory.

Second, for \( E \) outside of this range, \( x \) is dominated by the exponential terms: \( x \approx \frac{1}{2}(A + B)\exp(E) \) for \( E \gg 1 \), and \( x \approx -\frac{1}{2}(A - B)\exp(-E) \) for \( E \ll -1 \). We therefore do an approximate Newton-Raphson iteration on the function \( \ln |x| \), where the approximation is that we take \( d\ln |x|/d|E| \approx 1 \). This involves
computing an extra logarithm inside the loop, but gives very rapid convergence to high precision, since \( \ln |x| \) is very nearly linear in these regions.

At the start of the algorithm, we use an initial guess of \( E = -\ln[-2(x - x_{-1})/(A - B) - \exp(1)] \) for \( x < x_{-1}, E = x/x_{-1} \) for \( x_{-1} \leq x \leq 0, E = x/x_{+1} \) for \( 0 \leq x \leq x_{+1} \), or \( E = \ln[2(x-x_{+1})/(A+B)-\exp(1)] \) for \( x > x_{+1} \). We refine this guess until we get agreement to within 0.01 parts in part in \( N_{\text{cyc}} \) (where \( N_{\text{cyc}} \) is the larger of the number of wave cycles in a time \( 2\pi/n \), or the number of wave cycles in the entire waveform being generated), or one part in \( 10^{15} \) (an order of magnitude off the best precision possible with \text{REAL8} numbers). The latter case indicates that \text{REAL8} precision may fail to give accurate phasing, and one should consider modeling the orbit as a set of Taylor frequency coefficients à la \text{LALGenerateTaylorCW}(). On subsequent timesteps, we use the previous timestep as an initial guess, which is good so long as the timesteps are much smaller than \( 1/n \).

Once a value of \( E \) is found for a given timestep in the output series, we compute the system time \( t \) via Eq. (15.44), and use it to determine the wave phase and (non-Doppler-shifted) frequency via Eqs. (15.37) and (15.38). The Doppler shift on the frequency is then computed using Eqs. (15.45) and (15.43). We use \( \nu \) as an intermediate in the Doppler shift calculations, since expressing \( \dot{R} \) directly in terms of \( E \) results in expression of the form \( (e - 1)/(e \cosh E - 1) \), which are difficult to simplify and face precision losses when \( E \sim 0 \) and \( e \to 1 \). By contrast, solving for \( \nu \) is numerically stable provided that the system \text{atan2()} function is well-designed.

This routine does not account for relativistic timing variations, and issues warnings or errors based on the criteria of Eq. (15.50) in \text{GenerateEllipticSpinOrbitCW.c}.

Uses

LALMalloc() LALFree()
LALSCreateVectorSequence() LALSDestroyVectorSequence()
LALSCreateVector() LALSDestroyVector()
LALDCreateVector() LALDDestroyVector()
LALSnprintf() LALWarning()
15.10 Header SkyCoordinates.h

Provides routines to convert among various sky coordinate systems.

Synopsis

#include <lal/SkyCoordinates.h>

This header covers routines to perform coordinate transformations among the various spherical coordinate systems used in astronomy. Most of these routines are discussed in Sec. 5.1 of [4]; we reproduce here some of the essential elements of this discussion.

A general spatial coordinate system is shown in Fig. 15.3. It is defined by six parameters: three positions specifying the location of the origin $O$, two angles specifying the direction of the pole or $z$-axis, and one further angle specifying a reference meridian or $x$-axis orthogonal to the $z$-axis. A $y$-axis can also be defined such that $x$, $y$, and $z$ form an orthogonal right-handed coordinate system; however, in astronomy it is more conventional to use spherical coordinates, defined as follows:

For any given point $P$, define a plane (called its meridian) containing both the $z$-axis and the point in question. The longitude $\lambda$ is the angle in the $x$-$y$ plane from the $x$-axis to the line where the object’s meridian crosses the $x$-$y$ plane. The latitude $\phi$ is the angle in the meridian plane between this line and the direction to the object. The distance $r$ is simply measured in a straight line from the origin to the point. Longitude is defined to increase in the right-handed direction about the $z$-axis (i.e. the $y$-axis lies at positive $\pi/2$ radians longitude), and is typically given in the range $[0, 2\pi]$ radians. Latitude is defined to increase towards the $z$-axis (i.e. the $z$-axis lies at positive $\pi/2$ radians latitude), and is typically given in the range $[-\pi/2, \pi/2]$ radians; a point with latitude $\pm\pi/2$ radians has arbitrary (undefined) longitude. Distance should always be positive. This convention is shown in Fig. 15.3.

In the routines in this module, we do not perform transformations between coordinate systems having different origins. By default, all coordinates are assumed to be centred on the observer; however, one may also consider coordinate systems that are geocentric (having their origin at the centre of the Earth), heliocentric (origin at the centre of the Sun), barycentric (origin at the centre of mass of the solar system), and Galactocentric (origin at the centre of our Galaxy). Since we ignore translations in the coordinate origin, distances remain unchanged, so these routines only consider transformations in latitude and longitude. To put it another way, these routines transform directions in space, not locations in space. These directions are generically stored in the SkyPosition structure, defined below.

The coordinate systems that we consider are defined as follows:
Horizon coordinates: This is a local coordinate system for a particular observation point $O$ on the Earth, as shown in Fig. 15.4. The $z$-axis is defined to be the direction opposite to the local acceleration due to gravity. The $x$-axis is defined to lie in the plane formed by the $z$-axis and the Earth’s rotational axis, and to be directed into the northern hemisphere. In this coordinate system, the latitude coordinate is called the altitude and the longitude coordinate is the negative of what astronomers call the azimuth; this sign reversal is due to the fact that astronomers define azimuth to increase clockwise, and our longitudinal coordinates uniformly increase counterclockwise about the $z$-axis.

This coordinate system is related to the geographic coordinate system (below) by the geographic latitude $\phi_z$ and longitude $\lambda_z$ of the observer’s $z$-axis direction.

Geographic coordinates: This is a planetwide Earth-fixed coordinate system, shown in Fig. 15.5. The $z$-axis is defined to be parallel to the Earth’s axis, in the direction of the Earth’s north pole. The $x$-axis is defined to be parallel to the direction perpendicular from the Earth’s rotation axis to a reference point in Greenwich, UK (the prime meridian). Note that we adopt a longitude convention that is consistent with the Astronomical Almanac, but opposite to that in [4], in that our geographic longitudes increase eastward (counterclockwise) like the rest of our longitudinal coordinates.

The terms “latitude” and “longitude” without qualification normally refer to geographic latitude and longitude. However, we emphasize once again that geographic latitude and longitude as defined above refer to directions in space, not to locations on the Earth’s surface. This can lead to some confusion. The geodetic latitude and longitude of a point on the Earth’s surface are the latitude and longitude of its vertical direction; this is the standard meaning used by cartographers, and relates directly to the horizon-based coordinate system above. However, one can also define a geocentric latitude and longitude for a point on the surface, which are the latitude and longitude of the direction from the geometric centre of the Earth through that point. These angles are not necessarily the same, due to the Earth’s ellipticity, as shown in Fig. 15.8 in TerrestrialCoordinates.h.

Geographic coordinates are related to sky-fixed equatorial coordinates by specifying the counterclockwise angle to the prime meridian from the reference meridian $\Upsilon$ of the sky-fixed coordinates, as defined below. This angle is called the Greenwich Mean Sidereal Time (GMST), and is often specified in hours, minutes, and seconds.
**Equatorial coordinates:** This is the standard sky-fixed coordinate system. The $z$-axis is defined as for geographic coordinates, above; the plane orthogonal to this passing through the Earth’s centre is called the *equator*. The $x$-axis is defined to be the direction, as viewed from the centre of the Earth, where the Sun appears to cross the equator moving north in spring. This is called the *vernal equinox* $\Upsilon$, and is shown in Fig. 15.6. In this coordinate system, the latitude coordinate is called the *declination* $\delta$ and the longitude coordinate is called the *right ascension* $\alpha$.

**Ecliptic coordinates:** This is another sky-fixed coordinate system, shown in Fig. 15.6. The $z$-axis is defined to be the direction orthogonal to the orbital plane of the Earth about the Sun, directed such that the Earth orbits in a right-handed sense. The $x$-axis is defined as for equatorial coordinates, above; we note that by definition it lies parallel to the intersection of the equatorial and orbital planes of the Earth.

The equatorial and ecliptic coordinate systems are related by a single angle $\epsilon$, called the *obliquity of the ecliptic* (that is, the inclination of the Earth’s rotation axis relative to its orbital axis). Ecliptic latitude is normally denoted as $\beta$ and ecliptic longitude as $\lambda$.

**Galactic coordinates:** This coordinate system is shown in Fig. 15.7. The $z$-axis is defined to be the direction orthogonal to the plane of our Galaxy and pointing into the northern hemisphere of the equatorial coordinate system. (Unfortunately this convention has the unintuitive result that the physical rotation of the Galaxy is left-handed about this axis.) The $x$-axis is defined to be the direction of the Galactic centre as viewed from the Earth. The Galactic latitude coordinate is normally denoted as $b$ and the Galactic longitude as $l$.

The definition of the Galactic coordinate system is completely unrelated to any of the other coordinate systems; thus, the relationship between Galactic and equatorial coordinates requires one to specify three arbitrary (but constant) angles. Two of these are the right ascension $\alpha_{\text{NGP}}$ and declination $\delta_{\text{NGP}}$ of the North Galactic Pole ($z$-axis) in the equatorial coordinate system, the third is the Galactic longitude $l_{\text{ascend}}$ of the point where the Galactic plane ascends through the equatorial plane; i.e. the $l$ value for the direction along the intersection of the Galactic and equatorial planes, such that right-handed rotation about the Galactic $z$-axis moves you from south to north through the equator.
Error conditions

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUL</td>
<td>1</td>
<td>&quot;Unexpected null pointer in arguments&quot;</td>
</tr>
<tr>
<td>SYS</td>
<td>2</td>
<td>&quot;Wrong coordinate system in input&quot;</td>
</tr>
<tr>
<td>ZERO</td>
<td>3</td>
<td>&quot;Angular coordinates undefined at origin&quot;</td>
</tr>
<tr>
<td>SING</td>
<td>4</td>
<td>&quot;Point is inside singular ellipsoid&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above did not obey the LAL naming convention, i.e. the code does not use the file name in all caps as the prefix for the error names. Consult the source code for the full names. Better yet, fix the code. The source code with these messages is in SkyCoordinatesLaldoc.h on line 1.249.

Types

Enumeration CoordinateSystem

This enumerated type is used to identify data as being in one of the coordinate systems discussed above. The allowed values are:

- **COORDINATESYSTEM_HORIZON** A horizon coordinate system.
- **COORDINATESYSTEM_GEOGRAPHIC** The Earth-fixed geographic coordinate system.
- **COORDINATESYSTEM_EQUATORIAL** The sky-fixed equatorial coordinate system.
- **COORDINATESYSTEM_ECLIPTIC** The ecliptic coordinate system.
- **COORDINATESYSTEM_GALACTIC** The galactic coordinate system.

Structure SkyPosition

This structure stores the two spherical coordinates of a sky position; i.e. a generic latitude and longitude. The structure is not defined specific to a particular coordinate system, but maintains a tag indicating which coordinate system it is expressed in. The fields are:

- **REAL8 longitude** The longitudinal coordinate (in radians), as defined above.
- **REAL8 latitude** The latitudinal coordinate (in radians), as defined above.
- **CoordinateSystem system** The coordinate system in which the latitude and longitude have been expressed.

Structure EarthPosition

This structure stores the location of a point on (or near) the surface of the Earth in both geodetic and geocentric coordinates, as described in TerrestrialCoordinates.c. The fields are:

- **SkyPosition geodetic** The geographic coordinates of the upward vertical direction from the point; that is, the point’s geodetic latitude and longitude.
- **REAL8 elevation** The vertical distance of the point above the reference ellipsoid, in metres.
- **REAL8 x, y, z** The Earth-fixed geocentric Cartesian coordinates of the point, in metres.
- **REAL8 radius** The distance of the point from the geocentre, in metres.
- **SkyPosition geocentric** The geographic coordinates of the direction from the centre of the Earth through the point; that is, the point’s geocentric latitude and longitude.

Structure ConvertSkyParams

This structure stores parameters for the function LALConvertSkyPosition(). The fields are:

- **CoordinateSystem system** The coordinate system to which one is transforming.
- **SkyPosition *zenith** The position of the zenith of the horizon coordinate system; may be NULL if one is neither converting to nor from a horizon system.
LIGOTimeGPS *gpsTime The GPS time for conversions between Earth-fixed and sky-fixed coordinates; may be NULL if no such conversion is required (or if one is transforming to or from horizon coordinates and *zenith is given in the sky-fixed equatorial system).
15.10.1 Module CelestialCoordinates.c

Converts among Galactic, ecliptic, and equatorial coordinates.

Prototypes

```c
void
LALGalacticToEquatorial( LALStatus *stat,
    SkyPosition *output,
    SkyPosition *input )
```

```c
void
LALEquatorialToGalactic( LALStatus *stat,
    SkyPosition *output,
    SkyPosition *input )
```

```c
void
LALEclipticToEquatorial( LALStatus *stat,
    SkyPosition *output,
    SkyPosition *input )
```

```c
void
LALEquatorialToEcliptic( LALStatus *stat,
    SkyPosition *output,
    SkyPosition *input )
```

Description

These functions perform the specified coordinate transformation on the contents of *input and store the result in *output. The two pointers may point to the same object, in which case the conversion is done in place. The functions will also check input->system and set output->system as appropriate.

These routines are collected together because they involve fixed, absolute coordinate systems, so the transformations require no additional parameters such as the time or site of observation. We also note that there are no direct conversions between Galactic and ecliptic coordinates. At the risk of additional computational overhead, it is simple to use the equatorial coordinate system as an intermediate step.

Algorithm

These routines follow the spherical angle relations on p. 13 of [1]. Note that the actual formulae for Galactic longitude and right ascension in this reference are wrong; we give corrected formulae below derived from the sine and cosine equations. (The Galactic to equatorial transformations can also be found in Sec. 12.3 of [1].) All positions are assumed to be in the J2000 epoch.

**Galactic coordinates:** The following formulae relate Galactic latitude \( b \) and longitude \( l \) to declination \( \delta \) and right ascension \( \alpha \):

\[
\begin{align*}
    b &= \arcsin[\cos \delta \cos \delta_{NGP} \cos(\alpha - \alpha_{NGP}) + \sin \delta \sin \delta_{NGP}], \\
    l &= \arctan2[\sin \delta \cos \delta_{NGP} - \cos \delta \cos(\alpha - \alpha_{NGP}) \sin \delta_{NGP}, \cos \delta \sin(\alpha - \alpha_{NGP})] \\
        &+ l_{ascend},
\end{align*}
\]

(15.59)

(15.60)

where \( \arctan2(y,x) \) can be thought of as the argument of the complex number \( x + iy \); unlike \( \arctan(y/x) \), it ranges over the full range \([0, 2\pi)\) instead of just half of it. The inverse transformations are:

\[
\begin{align*}
    \delta &= \arcsin[\cos b \cos \delta_{NGP} \sin(l - l_{ascend}) + \sin b \sin \delta_{NGP}], \\
    \alpha &= \arctan2[\cos b \cos(l - l_{ascend}), \sin b \cos \delta_{NGP} - \cos b \sin(l - l_{ascend}) \sin \delta_{NGP}] \\
        &+ \alpha_{NGP}.
\end{align*}
\]

(15.61)

(15.62)
In these equations we have defined the orientation of the Galaxy with the following parameters (which should eventually be placed in `LALConstants.h`):

\[
\begin{align*}
\alpha_{\text{NGP}} & = 192.8594813^\circ = \text{the right ascension (epoch J2000) of the north Galactic pole} \\
\delta_{\text{NGP}} & = 27.1282511^\circ = \text{the declination (epoch J2000) of the north Galactic pole} \\
l_{\text{ascend}} & = 33^\circ = \text{the longitude of the ascending node of the Galactic plane}
\end{align*}
\]

The ascending node of the Galactic plane is defined as the direction along the intersection of the Galactic and equatorial planes where rotation in the positive sense about the Galactic \(z\) axis carries a point from the southern to northern equatorial hemisphere. That is, if \(u\) points in the direction \(\delta = 90^\circ\) (celestial north), and \(v\) points in the direction \(b = 90^\circ\) (Galactic north), then \(u \times v\) points along the ascending node.

**Ecliptic coordinates:** The following formulae relate Ecliptic latitude \(\beta\) and longitude \(\lambda\) to declination \(\delta\) and right ascension \(\alpha\):

\[
\begin{align*}
\beta & = \arcsin(\sin \delta \cos \epsilon - \cos \delta \sin \alpha \sin \epsilon) , \\
\lambda & = \arctan2(\cos \delta \sin \alpha \cos \epsilon + \sin \delta \sin \epsilon, \cos \delta \cos \alpha) .
\end{align*}
\]

The inverse transformations are:

\[
\begin{align*}
\delta & = \arcsin(\cos \beta \sin \lambda \sin \epsilon + \sin \beta \cos \epsilon) , \\
\alpha & = \arctan2(\cos \beta \sin \lambda \cos \epsilon - \sin \beta \sin \epsilon, \cos \beta \cos \lambda) .
\end{align*}
\]

Here \(\epsilon\) is the obliquity (inclination) of the ecliptic plane, which varies over time; at epoch J200 it has a mean value of:

\[\epsilon = 23.4392911^\circ .\]

**Uses**

**Notes**
15.10.2 Module TerrestrialCoordinates.c

Converts among equatorial, geographic, and horizon coordinates.

Prototypes

```c
void LALEquatorialToGeographic( LALStatus *stat,
                                 SkyPosition *output,
                                 SkyPosition *input,
                                 LIGOTimeGPS *gpsTime )

void LALGeographicToEquatorial( LALStatus *stat,
                                 SkyPosition *output,
                                 SkyPosition *input,
                                 LIGOTimeGPS *gpsTime )

void LALSSystemToHorizon( LALStatus *stat,
                          SkyPosition *output,
                          SkyPosition *input,
                          const SkyPosition *zenith )

void LALHorizonToSystem( LALStatus *stat,
                        SkyPosition *output,
                        SkyPosition *input,
                        const SkyPosition *zenith )

void LALGeodeticToGeocentric( LALStatus *stat, EarthPosition *location )

void LALGeocentricToGeodetic( LALStatus *stat, EarthPosition *location )
```

Description

The functions LALEquatorialToGeographic() and LALGeographicToEquatorial() convert between equatorial and geographic coordinate systems, reading coordinates in the first system from *input and storing the new coordinates in *output. The two pointers may point to the same object, in which case the conversion is done in place. The functions will also check input->system and set output->system as appropriate. Because the geographic coordinate system is not fixed, one must also specify the time of the transformation in *gpsTime.

The function LALSSystemToHorizon() transforms coordinates from either celestial equatorial coordinates or geographic coordinates to a horizon coordinate system, reading coordinates in the first system from *input and storing the horizon coordinates in *output, as above. The parameter *zenith specifies the direction of the vertical axis in the original coordinate system; the routine checks to see that input->system and zenith->system agree. Normally this routine is used to convert from geographic latitude and longitude to a horizon system, in which case *zenith simply stores the geographic (geodetic) coordinates of the observer; if converting from equatorial coordinates, zenith->longitude should store the local mean sidereal time of the horizon system.

The function LALHorizonToSystem() does the reverse of the above, transforming coordinates from horizon coordinates to either equatorial or geographic coordinates as specified by zenith->system; the value of output->system is set to agree with zenith->system.

Although it is conventional to specify an observation location by its geodetic coordinates, some routines may provide or require geocentric coordinates. The routines LALGeocentricToGeodetic() and LALGeodeticToGeocentric() perform this computation, reading and writing to the variable parameter structure *location. The function LALGeocentricToGeodetic() reads the fields location->x, y, z, and computes location->zenith and location->altitude. The function LALGeodeticToGeocentric() does the reverse, and also sets the fields location->position and location->radius.
**Algorithm**

These routines follow the formulae in Sec. 5.1 of [4], which we reproduce below.

**Geographic coordinates:** Since geographic and equatorial coordinates share the same z-axis, the geographic latitude $\phi$ of a direction in space is the same as its declination $\delta$, and longitude $\lambda$ and right ascension $\alpha$ differ only through the rotation of the Earth:

$$\lambda = \alpha - \left( \frac{2\pi \text{ radians}}{24 \text{ hours}} \right) \times \text{GMST},$$

(15.67)

where GMST is Greenwich mean sidereal time. The conversion routines here simply use the functions in the date package to compute GMST for a given GPS time, and add it to the longitude. While this is simple enough, it does involve several function calls, so it is convenient to collect these into one routine.

**Horizon coordinates:** We correct a typographical error on the second line of Eq. 5.45 of [4] (it should have $\cos A$, not $\sin A$). We also note that while our latitudinal coordinate is just the altitude $a$ in this system, our longitudinal coordinate increases counterclockwise, and thus corresponds to the negative of the azimuth $A$ as defined by [4]. So we have:

$$a = \arcsin(\sin \delta \sin \phi + \cos \delta \cos \phi \cos h),$$

(15.68)

$$-A = \arctan2(\cos \delta \sin h, \sin \delta \cos \phi - \cos \delta \sin \phi \cos h),$$

(15.69)

where $\delta$ is the declination (geographic latitude) of the direction being transformed, $\phi$ is the geographic latitude of the observer’s zenith (i.e. the observer’s geodetic latitude), and $h$ is the hour angle of the direction being transformed. This is defined as:

$$h = \lambda_{\text{zenith}} - \lambda = \text{LMST} - \alpha$$

where LMST is the local mean sidereal time at the point of observation. The inverse transformation is:

$$\delta = \arcsin(\sin a \sin \phi + \cos a \cos A \cos \phi),$$

(15.70)

$$h = \arctan2(\cos a \sin(-A), \sin a \cos \phi - \cos a \cos A \sin \phi).$$

(15.71)

As explained in CelestialCoordinates.c, the function $\arctan2(y, x)$ returns the argument of the complex number $x + iy$. 


Geocentric coordinates: As shown in Fig. 15.8 the ellipticity of the Earth means that the vertical axis of a point on the Earth’s surface does not pass through the geometric centre of the Earth. This means that the geodetic latitude of a location (defined as the latitude angle \( \phi_{\text{geodetic}} \) of that location’s zenith direction) is typically some 10 arcminutes larger than its geocentric latitude (defined as the latitude angle \( \phi_{\text{geographic}} \) of the position vector from the geocentre through the location). Cartographers traditionally refer to locations by their geodetic coordinates, since these can be determined locally; however, geocentric coordinates are required if one wants to construct a uniform Cartesian system for the Earth as a whole.

To transform from geodetic to geocentric coordinates, one first defines a “reference ellipsoid”, the best-fit ellipsoid to the surface of the Earth. This is specified by the polar and equatorial radii of the Earth \( r_p \) and \( r_e \), or equivalently by \( r_e \) and a flattening factor:

\[
    f \equiv 1 - \frac{r_p}{r_e} = 0.00335281
\]

(This constant will eventually migrate into LALConstants.h.) The surface of the ellipsoid is then specified by the equation

\[
    r = r_e (1 - f \sin^2 \phi),
\]

where \( \phi = \phi_{\text{geodetic}} \) is the geodetic latitude. For points off of the reference ellipsoid, the transformation from geodetic coordinates \( \lambda, \phi \) to geocentric Cartesian coordinates is:

\[
    x = (r_e C + h) \cos \phi \cos \lambda, \quad (15.72) \\
    y = (r_e C + h) \cos \phi \sin \lambda, \quad (15.73) \\
    z = (r_e S + h) \sin \phi, \quad (15.74)
\]

where

\[
    C = \frac{1}{\sqrt{\cos^2 \phi + (1 - f)^2 \sin^2 \phi}}, \quad (15.75) \\
    S = (1 - f)^2 C, \quad (15.76)
\]

and \( h \) is the perpendicular elevation of the location above the reference ellipsoid. The geocentric spherical coordinates are given simply by:

\[
    r = \sqrt{x^2 + y^2 + z^2}, \quad (15.77) \\
    \lambda_{\text{geocentric}} = \lambda = \lambda_{\text{geodetic}}, \quad (15.78) \\
    \phi_{\text{geocentric}} = \arcsin(z/r). \quad (15.79)
\]

When computing \( r \) we are careful to factor out the largest component before computing the sum of squares, to avoid floating-point overflow; however this should be unnecessary for radii near the surface of the Earth.

The inverse transformation is somewhat trickier. Eq. 5.29 of [4] conveniently gives the transformation in terms of a sequence of intermediate variables, but unfortunately these variables are not particularly computer-friendly, in that they are prone to underflow or overflow errors. The following equations essentially reproduce this sequence using better-behaved methods of calculation.

Given geocentric Cartesian coordinates \( x = r \cos \phi_{\text{geocentric}} \cos \lambda, \ y = r \cos \phi_{\text{geocentric}} \sin \lambda, \) and \( z = r \sin \phi_{\text{geocentric}} \), one computes the following:

\[
    \omega = \sqrt{\left(\frac{x}{r_e}\right)^2 + \left(\frac{y}{r_e}\right)^2}, \\
    E = (1 - f) \left| \frac{z}{r_e} \right| - f(2 - f), \\
    F = (1 - f) \left| \frac{z}{r_e} \right| + f(2 - f),
\]
\[ P = \frac{4}{3} (EF + \varpi^2) = \frac{4}{3} \left[ \varpi^2 + (1 - f)^2 \left( \frac{z}{r_e} \right)^2 - f^2(2 - f)^2 \right], \]
\[ Q = 2(F^2 - E^2) = 8f(1 - f)(2 - f) \left| \frac{z}{r_e} \right|, \]
\[ D = P^3 + \varpi^2Q^2, \]
\[ v = \begin{cases} \left( \sqrt{D} + \varpi Q \right)^{1/3} - \left( \sqrt{D} - \varpi Q \right)^{1/3} & D \geq 0 \\ 2\sqrt{-P} \cos \left( \frac{1}{3} \arccos \left[ \frac{Q}{P^{1/3}} \right] \right) & D \leq 0 \end{cases} \]
\[ G = \frac{1}{2} \left( E + \sqrt{E^2 + \varpi^2} \right), \]
\[ H = \frac{\varpi^2 F - \varpi v G}{G^2(2G - E)}, \]
\[ t = G \left( \sqrt{1 + H} - 1 \right). \]

Once we have \( t \) and \( \varpi \), we can compute the geodetic longitude \( \lambda \), latitude \( \phi \), and elevation \( h \):
\[ \lambda = \arctan2(y, x), \quad (15.80) \]
\[ \phi = \text{sgn}(z) \arctan \left[ \frac{2}{1 - f} \left( \frac{(\varpi - t)(\varpi + t)}{\varpi t} \right) \right], \quad (15.81) \]
\[ h = r_e(\varpi - t/\varpi) \cos \phi + [z - \text{sgn}(z)r_e(1 - f)] \sin \phi. \quad (15.82) \]

These formulae, however, introduce certain concerns of numerical precision that have been only partially dealt with in this code. Specifically:

- There is a coordinate singularity at \( \varpi = 0 \), which we deal with by setting \( \phi = \pm 90^\circ \) and \( \lambda \) arbitrarily to \( 0^\circ \). When \( z = 0 \) as well, we arbitrarily choose the positive sign for \( \phi \). However, the computation of \( h \) in particular has tricky cancelations as \( \varpi \to 0 \), which may give rise to numerical errors. These have not yet been thoroughly explored.

- There is another coordinate singularity when \( D \to 0 \), which defines an ellipsoid with equatorial radius \( r_0 = r_ef(2 - f) = 42.6977 \) km and axial height \( z_0 = r_e/(1 - f) = 42.8413 \) km. Within this ellipsoid, lines of constant latitude begin to cross one another. The listed solution is an analytic continuation of the exterior solution which assigns these points a unique, if arbitrary, geodetic latitude. This solution has some peculiar behaviour, such as giving points in the equatorial plane a positive latitude. In practice, however, users will rarely be interested coordinate transformations deep within the Earth’s core.

- The equations for \( v \) and \( G \) have square and cube roots of expressions involving squares and cubes of numbers. For formal robustness one should factor out the leading-order dependence, so that one is assured of taking squares and cubes of numbers near unity. However, we are using \texttt{REAL8} precision, and have already normalized our distances by the Earth’s radius \( r_e \), so the point is almost certainly irrelevant.

- The expression for \( H \) may go to zero, leading to precision errors in \( t \); the number of digits of precision lost is on the order of the number of leading zeros after the decimal place in \( H \). I arbitrarily state that we should not lose more than 4 of our 16 decimal places of precision, meaning that we should series-expand the square root for \( H < 10^{-4} \). To get our 12 places of precision back, we need an expansion to \( H^3 \):
\[ t \approx G \left( \frac{1}{2}H - \frac{3}{8}H^2 + \frac{5}{16}H^3 \right). \]

- When computing \( \phi \), we first compute \( t - \varpi, t + \varpi, t^{-1}, \) and \( \varpi^{-1} \), sort them by order of magnitude, and alternately multiply large and small terms. We note that if the argument of the arctan function is large we have
\[ \arctan(x) = \text{sgn}(x) \frac{\pi}{2} - \arctan \left( \frac{1}{x} \right), \]
but the \texttt{atan()} function in the C math library should be smart enough to do this itself.
Ellipsoidal vs. orthometric elevation: In this module it is assumed that all elevations refer heights above the reference ellipsoid. This is the elevation computed by such techniques as GPS triangulation. However, the “true” orthometric elevation refers to the height above the mean sea level or geoid, a level surface in the Earth’s gravitational potential. Thus, even if two points have the same ellipsoidal elevation, water will still flow from the higher to the lower orthometric elevation.

The difference between the geoid and reference ellipsoid is called the “undulation of the geoid”, and can vary by over a hundred metres over the Earth’s surface. However, it can only be determined through painstaking measurements of local variations in the Earth’s gravitational field. For this reason we will ignore the undulation of the geoid.

Uses

LALGPStoUTC()
XLALGreenwichMeanSiderealTime()
LALDHeapSort()

Notes
15.10.3 Module SkyCoordinates.c

Automatically converts among sky coordinate systems.

Prototypes

```c
void LALConvertSkyCoordinates( LALStatus *stat,
    SkyPosition *output,
    SkyPosition *input,
    ConvertSkyParams *params )
```

Description

The function `LALConvertSkyCoordinates()` transforms the contents of `input` to the system specified in `params`, storing the result in `output` (which may point to the same object as `input` for an in-place transformation). The routine makes calls to the functions in `CelestialCoordinates.c` and `TerrestrialCoordinates.c` as required; the `params` object must store any data fields required by these functions, or an error will occur.

The function `LALNormalizeSkyPosition()` “normalizes” any given (spherical) sky-position (in radians), which means it projects the angles into $[0, 2\pi) \times [-\pi/2, \pi/2]$ if they lie outside.

Algorithm

`LALConvertSkyCoordinates()` is structured as a simple loop over transformations, each of which moves the output sky position one step closer to the desired final coordinates system. The usual “flow” of the algorithm is:

```
horizon ≜ geographic ≜ equatorial \updownarrow \leftrightarrow \downarrow Galactic
```

although one can also convert directly between equatorial and horizon coordinate systems if `params.zenith` is given in equatorial coordinates (i.e. if its longitudinal coordinate is the local mean sidereal time rather than the geographic longitude of the observer). This leads to the only error checking done within this function: when transforming to horizon coordinates, it checks that `params.zenith` is either in sky-fixed equatorial or Earth-fixed geographic coordinates. Other than this, error checking is left to the secondary function call; if a parameter is absent or poorly formatted, the called function will return an error.

Uses

```
LALHorizonToSystem()  LALSystemToHorizon()
LALGeographicToEquatorial()  LALEquatorialToGeographic()
LAEquatorialToEcliptic()  LALEclipticToEquatorial()
LAEquatorialToGalactic()  LALGalacticToEquatorial()
```

Notes

Author: Creighton, T. D.

$Id: SkyCoordinates.c,v 1.12 2007/06/08 14:41:47 bema Exp$
15.10.4 Program SkyCoordinatesTest.c

Transforms coordinates among various systems.

Usage

SkyCoordinatesTest [-i system lat lon] [-o system] [-z lat lon]
[-a altitude] [-c lat lon rad] [-t sec nsec] [-d debuglevel]

Description

This program converts between any two celestial coordinate systems, or between geocentric and geodetic terrestrial coordinates, using the routines in SkyCoordinates.h. The following option flags are accepted:

- **-i** Sets the input coordinate system and coordinate values for a celestial coordinate transformation: `system` may be one of `horizon`, `geographic`, `equatorial`, `ecliptic`, or `galactic`; `lat` and `lon` are the latitude and longitude coordinates in that system (in degrees). If the `-i` option is not given, then no celestial coordinate transformation will be performed (although a terrestrial coordinate transformation may still occur; see below).

- **-o** Sets the output coordinate system for a celestial coordinate transformation: `system` may be any of the above. If the `-o` option is not given, then no celestial coordinate transformation will be performed (although a terrestrial coordinate transformation may still occur; see below).

- **-z** Sets the `geodetic` latitude and longitude of the observer to `lat` and `lon`, respectively (in degrees). Either this or the `-c` option (below) is required for a celestial coordinate transformation involving the horizon system.

- **-a** Sets the elevation of the observer above the Earth’s reference ellipsoid to `altitude` (in metres). If given along with the `-z` option, above, the program will compute and print out the geocentric coordinates of the observer as well.

- **-c** Sets the `geocentric` latitude and longitude of the observer to `lat` and `lon`, respectively (in degrees), and the distance from the geocentre to `rad` (in metres). The program will convert and print out the geodetic coordinates of the observer. Either this or the `-z` option (below) is required for a celestial coordinate transformation involving the horizon system; if both are given, this option is ignored.

- **-t** Sets the GPS time of the conversion to `sec` seconds plus `nsec` nanoseconds. The time will be printed in various other formats. This option is required for any transformation between Earth-fixed and sky-fixed coordinate systems.

- **-d** Sets the debug level to `debuglevel`. If not specified, level 0 is assumed.

If no option flags are specified at all, then the routine will randomly generate a sky position in Galactic coordinates, convert it to ecliptic coordinates and back again, and return an error if the result disagrees by more than a milliradian.

Exit codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NORM</td>
<td>0</td>
<td>&quot;Normal exit&quot;</td>
</tr>
<tr>
<td>SUB</td>
<td>1</td>
<td>&quot;Subroutine failed&quot;</td>
</tr>
<tr>
<td>ARG</td>
<td>2</td>
<td>&quot;Error parsing arguments&quot;</td>
</tr>
<tr>
<td>MEM</td>
<td>3</td>
<td>&quot;Out of memory&quot;</td>
</tr>
<tr>
<td>TEST</td>
<td>4</td>
<td>&quot;Test case failed&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants `SKYCOORDINATESTESTC_E<name>`, and the status descriptions in `SKYCOORDINATESTESTC_MSGE<name>`. The source code with these messages is in SkyCoordinatesTest.c on line 1.99.
Algorithm

Uses

lalDebugLevel
LALPrintError() LALCheckMemoryLeaks()
LALMalloc() LALFree()
LALGeocentricToGeodetic() LALGeodeticToGeocentric()
LALGPSstoINT8() LALINT8stoGPS()
LALGPSstoUTC() LALDateString()
LALGPSstoGMST1() LALJulianDate()
LALCHARCreateVector() LALCHARDestroyVector()
LALCreateRandomParams() LALDestroyRandomParams()
LALUniformDeviate() LALConvertSkyCoordinates()
LALNormalizeSkyPosition()

Notes
References


Section 7

Burst Packages
Chapter 16

Package \texttt{block}
16.1 Header `BlockRho.h`

Provides structures and definitions global to the Rho algorithms, notably `BlockSearchParams`.

### Error conditions

<table>
<thead>
<tr>
<th><code>&lt;name&gt;</code></th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>1</td>
<td>&quot;Null pointer&quot;</td>
</tr>
<tr>
<td>NNUL</td>
<td>2</td>
<td>&quot;Non-null pointer&quot;</td>
</tr>
<tr>
<td>ALOC</td>
<td>3</td>
<td>&quot;Memory allocation error&quot;</td>
</tr>
<tr>
<td>NUMZ</td>
<td>4</td>
<td>&quot;Data segment length is zero&quot;</td>
</tr>
<tr>
<td>ARG</td>
<td>5</td>
<td>&quot;Error parsing command-line arguments&quot;</td>
</tr>
<tr>
<td>DATA</td>
<td>6</td>
<td>&quot;Too few input data points to define a Rho statistic&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants `BLOCKRHOH_<name>`, and the status descriptions in `BLOCKRHOH_MSGE<name>`. The source code with these messages is in `BlockRho.h` on line 1.76.

### Structures and Unions
16.1.1 Module BlockRho2.c

Prototypes

static REAL8 nextOrder( INT4 arg )

static REAL8 iFunct( INT4 arg )

static void lnIRatio(
    LALStatus *status,
    REAL8Vector **result,
    INT4 N,
    INT4 T
)

void LALBlockRho2 ( LALStatus *status,
    REAL8 *result,
    REAL8 *rpeak,
    INT4 *myindex,
    REAL8Sequence *data,
    UINT4 *marginOfExclusion
)

Description

To be written

Algorithms

Matlab Code:
    does the following:

    1. 
    2. 
    3. 
    4. 

Uses

    • LALDDestroyVector

Notes
Chapter 17

Package burstsearch

Functions for the implementation of the standard burst searches:

- A standard interface for burst event trigger generators.
- A function for the estimation of burst parameters.
- A wrapper for the TFCLUSTERS algorithm.
- A wrapper for the SLOPE algorithm.
- A set of functions to implement the excess power search technique which was suggested in Ref. [2] and later independently invented in Ref. [3]. The implementation here is described in detail in Ref. [4].
17.1 Header TFTransform.h

Provides routines to compute time-frequency planes from either time-domain or frequency-domain data, for use in the excess power search technique.

Synopsis

```
#include "TFTransform.h"
```

This package provides a suite for functions for computing time-frequency representations of data using stacked Fourier transforms.

The first few functions simply combine functionality from the packages fft and window in a convenient way. They are designed to streamline the task of setting up structures to prepare for taking many discrete Fourier transforms (DFTs), including windows and plans for FFTW.

A general description of the time-frequency (TF) transform provided by TFTransform is as follows. Suppose one starts with some data $h_j$, $0 \leq j < n$ in the time domain, with sampling time $\Delta t$, so that the data point $h_j$ corresponds to a time $t_j = t_{\text{start}} + j\Delta t$. Taking the standard DFT yields complex data

$$
\hat{h}_\gamma = \sum_{j=0}^{n-1} e^{-2\pi ij\gamma/n} h_j
$$

in the Fourier domain, for $0 \leq \gamma \leq [n/2] + 1$. Here the data point $\hat{h}_\gamma$ corresponds to a frequency $f_\gamma = \gamma\Delta f$, where $\Delta f = 1/(n\Delta t)$ is the frequency resolution.

Now suppose that we can factorize the number $n$ of data points as

$$
n = 2N_T N_F.
$$

Then, by a time-frequency plane we shall mean a set of $N_T N_F$ complex numbers $H_{IJ}$ with $0 \leq I < N_T$ and $0 \leq \Gamma < N_F$, obtained by an invertible linear transformation from the original data, such that the data point $H_{IJ}$ corresponds approximately to a time $t_I = t_{\text{start}} + I\Delta t$ and to a frequency $f_{\Gamma} = \Gamma\Delta f$. Here $N_F$ is the number of frequency bins in the TF plane, and $N_T$ is the number of time bins. The time resolution $\Delta t$ and frequency resolution $\Delta f$ are related by $\Delta t \Delta f = 1$, and are given by $\Delta t = 2N_F\Delta t$ and $\Delta f = N_T\Delta f$. Note that there are many other time-frequency representations of data that are not of this type; see [1].

There are many possible choices of linear transformations from the data $h_j$ to data $H_{IJ}$ satisfying the above properties. Here we have implemented two simple choices. The first choice consists of dividing the time-domain data $h_j$ into $N_T$ equal-sized chunks, each of length $n/N_T$, and then taking the forward DFT of each chunk. Then, $H_{IJ}$ is just the $I$th element of the $J$th chunk. In terms of formulae this corresponds to

$$
H_{JS} = \sum_{k=0}^{2N_F-1} \exp \left[ 2\pi ik\Sigma/(2N_F) \right] h_{2N_FJ+k},
$$

for $0 \leq J < N_T$ and $0 \leq \Sigma < N_F$. We call this first type of TF plane a vertical TF plane, since it corresponds to a series of vertical lines if the time axis is horizontal and the frequency axis vertical.

The second type of TF plane is obtained by first taking a DFT of all the time-domain data to obtain frequency-domain data, then dividing the frequency-domain data into $N_F$ equal-sized chunks, and then taking the inverse DFT of each chunk. We call the resulting TF plane a horizontal TF plane. In terms of formulae the TF plane elements are

$$
H_{JS} = \sum_{\gamma=0}^{N_T-1} \exp \left[ -2\pi iJ\gamma/N_T \right] \hat{h}_{N_T\Sigma+\gamma},
$$

for $0 \leq J < N_T$ and $0 \leq \Sigma < N_F$, where $\hat{h}_\gamma$ is given by Eq. (17.1).

Structures

```
struct TFPlaneParams
```

Parameters needed to describe a particular TF plane.

```
INT4 timeBins Number of time bins $N_T$ in TF plane.
```
INT4 freqBins Number of freq bins $N_F$ in TF plane.

REAL8 deltaT The time resolution $\Delta t$ of the TF plane in seconds, $\text{deltaT}$ will always be $1/\text{deltaT}$.

REAL8 flow The lowest frequency $f_{\text{low}}$ in the TF plane in Hertz [such that the data point $H_{J\Gamma}$ corresponds to a time $t_J = t_{\text{start}} + J\Delta t$ and to a frequency $f_{\Gamma} = f_{\text{low}} + \Gamma\Delta f$, in a slight generalization of the above correspondence].

struct RealDFTParams

WindowType windowType
REAL4Vector *window
REAL4 sumofsquares
RealFFTPlan *plan

struct ComplexDFTParams

WindowType windowType
REAL4Vector *window
REAL4 sumofsquares
ComplexFFTPlan *plan

struct COMPLEX8TimeFrequencyPlane
This structure has some fields that also appear in the structures REAL4TimeSeries and COMPLEX8FrequencySeries.

CHAR *name The name of the TF plane.

LIGOTimeGPS epoch The initial time $t_{\text{start}}$ of the data used to generate the TF plane.

CHARVector *sampleUnits The units of the quantities $H_{J\Gamma}$.

TFPlaneParams *params Parameters needed to generate the plane from input data. (See above.)

TFPlaneType planeType This is an enumerated type that can be either verticalPlane or horizontalPlane, corresponding to the two types of TF plane.

COMPLEX8 *data The $N_T \times N_F$ array of complex numbers $H_{J\Sigma}$.

struct VerticalTFTransformIn

RealDFTParams *dftParams
INT4 startT

struct HorizontalTFTransformIn

ComplexDFTParams *dftParams
INT4 startT
17.1.1 Module CreateRealDFTParams.c

Creates a structure of type RealDFTParams,

Prototypes

Description

The inputs to CreateDFTParams() consist of (i) a parameter winParams of type WindowParams* giving the length of the vectors to be Fourier transformed and the type of windowing to be used, (ii) a pointer dftParams to a pointer to a RealDFTParams structure, and (iii) an integer sign specifying the direction of the transform, with +1 indicating forward transform and −1 indicating inverse transform. On exit, *dftParams will point to the newly created structure.

Uses

LALCreateForwardRealFFTPlan
LALCreateReverseRealFFTPlan
LALSCreateVector
LALWindow

Notes
References


Chapter 18

Package ring
18.1 Header Ring.h

Black hole ringdown waveform generation.

Synopsis

```c
#include <lal/Ring.h>
```

Routines for generating waveforms for black hole ringdown. The ringdown waveform is an exponentially-damped sinusoid

\[ r(t) = \begin{cases} 
  e^{-\pi ft/Q} \cos(2\pi ft + \phi_0) & \text{for } t \geq 0 \\
  0 & \text{for } t < 0 
\end{cases} \]

(18.1)

where \( f \) is the central frequency of the ringdown waveform, \( Q \) is the quality factor, and \( \phi_0 \) is the initial phase of the waveform. Note that Ref. [4] adopted the normalization convention \( q(t) = (2\pi)^{1/2} r(t) \).

For a black hole ringdown, the gravitational waveform produced, averaged over the various angles, is

\[ h(t) = A_q q(t) \]

(18.2)

where the central frequency and quality of the ringdown are determined from the mass and spin of the black holes. An analytic approximation yields [1, 2]

\[ f \simeq 32 \text{ kHz} \times [1 - 0.63(1 - \hat{a})^{3/10}] (M_\odot / M) \]

(18.3)

and

\[ Q \simeq 2(1 - \hat{a})^{-9/20} \]

(18.4)

with the black hole mass given by \( M \) and its spin by \( S = \hat{a} GM^2 / c \) (where \( G \) is Newton’s constant and \( c \) is the speed of light). The dimensionless spin parameter \( \hat{a} \) lies between zero (for a Schwarzschild black hole) and unity (for an extreme Kerr black hole). The amplitude of the waveform depends on these quantities as well as the distance \( r \) to the source and the fractional mass loss \( \epsilon \) radiated in gravitational waves [4]:

\[ A_q = 2.415 \times 10^{-21} Q^{-1/2} [1 - 0.63(1 - \hat{a})^{3/10}]^{-1/2} \left( \frac{Mpc}{r} \right) \left( \frac{M}{M_\odot} \right) \left( \frac{\epsilon}{0.01} \right)^{1/2} \].

(18.5)

Note that this is the amplitude factor for the waveform \( q(t) \), whereas the amplitude factor for \( r(t) \) would be \((2\pi)^{1/2} A_q\).

The mismatch between two nearby templates is given by \( ds^2 \), which can be thought of as the line interval for a mismatch-based metric on the \((f, Q)\) parameter space [3, 4, 5]:

\[ ds^2 = \frac{1}{8} \left\{ 3 + 16Q^4 \left[ \frac{3 + 4Q^2}{fQ(1 + 4Q^2)} \right] dQ^2 - 2 \left[ \frac{3 + 4Q^2}{fQ(1 + 4Q^2)} \right] dQ df + \frac{3 + 8Q^2}{f^2} df^2 \right\}. \]

(18.6)

When expressed in terms of \( \log f \) rather than \( f \), the metric coefficients depend on \( Q \) alone. We can exploit this property for the task of template placement. The method is the following: First, choose a “surface” of constant \( Q = Q_{\text{min}} \), and on this surface place templates at intervals in \( \phi = \log f \) of \( d\phi = df / \sqrt{g_{\phi\phi}} \) for the entire range of \( \phi \). Here, \( df = \sqrt{2ds_{\text{threshold}}^2} \). Then choose the next surface of constant \( Q \) with \( dQ = df / \sqrt{g_{QQ}} \) and repeat the placement of templates on this surface. This can be iterated until the entire range of \( Q \) has been covered; the collection of templates should now cover the entire parameter region with no point in the region being farther than \( ds_{\text{threshold}}^2 \) from the nearest template.

Error conditions

<table>
<thead>
<tr>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL 01</td>
<td>&quot;Null pointer&quot;</td>
</tr>
<tr>
<td>NNULL 02</td>
<td>&quot;Non-null pointer&quot;</td>
</tr>
<tr>
<td>ALOC 04</td>
<td>&quot;Memory allocation error&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants RINGH_E<name>, and the status descriptions in RINGH_MSGE<name>. The source code with these messages is in Ring.h on line 1.128.
typedef struct
    tagRingTemplateBank
{
    UINT4    numTmplt;
    SnglRingdownTable *tmplt;
}
RingTemplateBank;

This structure contains a bank of ringdown waveforms. The fields are:

numTmplt  The number of templates in the bank.
tmplt    Array of ringdown templates.

Type RingTemplateBankInput

typedef struct
    tagRingTemplateBankInput
{
    REAL4 minQuality;
    REAL4 maxQuality;
    REAL4 minFrequency;
    REAL4 maxFrequency;
    REAL4 maxMismatch;
    REAL4 templatePhase;
    REAL4 templateDistance;
    REAL4 templateEpsilon;
}
RingTemplateBankInput;

This structure contains the parameters required for generating a ringdown template bank. The fields are:

minQuality  The minimum quality factor in the bank.
maxQuality  The maximum quality factor in the bank.
minFrequency  The minimum central frequency in the bank (in Hz).
maxFrequency  The minimum central frequency in the bank (in Hz).
maxMismatch  The maximum mismatch allowed between templates in the bank.

templatePhase  The phase of the ringdown templates, in radians. Zero is a cosine-phase template; \(-\pi/2\) is a sine-phase template.
18.1.1 Module Ring.c

Routines to generate ringdown waveforms and to make a ringdown template bank.

Prototypes

REAL4 XLALBlackHoleRingSpin( REAL4 Q )
REAL4 XLALBlackHoleRingMass( REAL4 f, REAL4 Q )
REAL4 XLALBlackHoleRingAmplitude( REAL4 f, REAL4 Q, REAL4 r, REAL4 epsilon )
REAL4 XLALBlackHoleRingEpsilon( REAL4 f, REAL4 Q, REAL4 r, REAL4 amplitude )
int XLALComputeRingTemplate( REAL4TimeSeries *output, SnglRingdownTable *input )
int XLALComputeBlackHoleRing( REAL4TimeSeries *output, SnglRingdownTable *input, REAL4 dynRange )
int XLALCreateRingTemplateBank( RingTemplateBankInput *input )
void XLALDestroyRingTemplateBank( RingTemplateBank *bank )

Description

The routine LALComputeRingTemplate() computes the ringdown waveform

\[ r(t) = \begin{cases} e^{-\pi ft/Q} \cos(2\pi ft) & \text{for } t \geq 0 \\ 0 & \text{for } t < 0 \end{cases} \]  

(18.7)

where the parameters \( f \) and \( Q \) are specified in the input structure. The output must have an appropriate amount of memory allocated, and must have the desired temporal spacing set. Note: Ref. [4] used a different convention for the ringdown normalization: there the ringdown waveform was taken to be \( q(t) = (2\pi)^{-1/2} r(t) \).

The routine LALComputeBlackHoleRing() computes a waveform for a black hole with the specified physical parameters (in the input structure). The parameters are the black hole mass \( M \) (in solar masses \( M_\odot \)), the spin \( S = \hat{a}GM^2/c \) expressed in terms of the dimensionless mass parameter \( \hat{a} \), the fractional mass lost \( \epsilon \) in ringdown radiation expressed as a percent, and the distance to the typical source (angle-averaged waveform) \( r \) given in megaparsecs (Mpc). The central frequency and quality of the ringdown are approximated as [1, 2]:

\[ f \approx 32 \text{kHz} \times [1 - 0.63(1 - \hat{a})^{3/10}](M_\odot/M) \]  

(18.8)

and

\[ Q \approx 2(1 - \hat{a})^{-9/20}. \]  

(18.9)

The strain waveform produced is \( h(t) = A_q q(t) \) where the amplitude factor is [4]

\[ A_q = 2.415 \times 10^{-21} Q^{-1/2}[1 - 0.63(1 - \hat{a})^{3/10}]^{-1/2} \left( \frac{\text{Mpc}}{r} \right) \left( \frac{M}{M_\odot} \right) \left( \frac{\epsilon}{0.01} \right)^{1/2}. \]  

(18.10)

Note that this is written \( A_q \) to emphasize that it is the amplitude factor for \( q(t) \) rather than \( r(t) \).

The routine LALCreateRingTemplateBank() creates a bank of ringdown templates that cover a set range in the parameters \( f \) and \( Q \). The bank is destroyed with LALDestroyRingTemplateBank().

Algorithm

The waveform generation routines use recurrence relations for both the exponentially-decaying envelope and for the co-sinusoid.

The template placement algorithm is described above.

Uses

Notes

* Author: Jolien Creighton
* $Id: Ring.c,v 1.12 2007/09/02 00:40:10 lgoggin Exp $
References


Chapter 19

Package slopefilters

A collection of filters to detect slopes in arrays of input data
### 19.1 Header SlopeDetectorFilter.h

Declares functions for slope detection. Defines the fields of a structure `SLOPEFilterParams` whose fields are enumerated and described below.

#### Synopsis

```c
#include <lal/SlopeDetectorFilter.h>
```

#### Error conditions

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUTNULLP</td>
<td>1</td>
<td>&quot;Null input pointer&quot;</td>
</tr>
<tr>
<td>OUTPUTNULLP</td>
<td>2</td>
<td>&quot;Null output pointer&quot;</td>
</tr>
<tr>
<td>TAPSNULLP</td>
<td>3</td>
<td>&quot;Inappropriate null taps pointer&quot;</td>
</tr>
<tr>
<td>HISTNULLP</td>
<td>4</td>
<td>&quot;Null history pointer&quot;</td>
</tr>
<tr>
<td>INVFLTLEN</td>
<td>5</td>
<td>&quot;Invalid filter length&quot;</td>
</tr>
<tr>
<td>DATATOOSHORT</td>
<td>6</td>
<td>&quot;Data segment too short&quot;</td>
</tr>
<tr>
<td>AMBHISETBIT</td>
<td>7</td>
<td>&quot;Ambiguous history buffer set bit&quot;</td>
</tr>
<tr>
<td>INVALIDACTION</td>
<td>8</td>
<td>&quot;Filter action invalid&quot;</td>
</tr>
<tr>
<td>BINOFFINVALID</td>
<td>9</td>
<td>&quot;Bin offset invalid&quot;</td>
</tr>
<tr>
<td>INVALIDTAPSBIT</td>
<td>10</td>
<td>&quot;Invalid taps bit&quot;</td>
</tr>
<tr>
<td>DIVBYZERO</td>
<td>11</td>
<td>&quot;Division by zero&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants `SLOPEDETECTORFILTERH_E<name>`, and the status descriptions in `SLOPEDETECTORFILTERH_MSGE<name>`. The source code with these messages is in `SlopeDetectorFilter.h` on line 1.119.

#### Structures

`SLOPEFilterParams` is a structure defined in this header which has the fields defined in table 19.1.

---

<table>
<thead>
<tr>
<th>datatype</th>
<th>parameter</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UINT4</td>
<td><code>forder</code></td>
<td>number of bins (N) used in filter</td>
</tr>
<tr>
<td>REAL4*</td>
<td><code>tap</code></td>
<td>filter taps</td>
</tr>
<tr>
<td>UINT4*</td>
<td><code>history_allocated</code></td>
<td>1 if set, 0 if not.</td>
</tr>
<tr>
<td>UINT4*</td>
<td><code>taps_set</code></td>
<td>1 if set, 0 if not.</td>
</tr>
<tr>
<td>REAL4*</td>
<td><code>history</code></td>
<td>history buffer</td>
</tr>
<tr>
<td>UINT4</td>
<td><code>function_select</code></td>
<td>see function descriptions</td>
</tr>
<tr>
<td>REAL4</td>
<td><code>waveform_offset</code></td>
<td>offset of waveform from bin, (0-1)</td>
</tr>
<tr>
<td>REAL4</td>
<td><code>sampling_period_s</code></td>
<td>sampling period</td>
</tr>
</tbody>
</table>

Table 19.1: Fields of the `SLOPEFilterParams` structure.
19.1.1 Module SlopeDetectorFilter.c

Functions in this code segment implement various time domain search algorithms for transients and bursts in the LIGO data.

Prototypes

```c
void LALSlopeDetectorFilter( LALStatus *status,
REAL4Vector* output_data,
const REAL4Vector* input_data,
const UINT4 ntaps )

void LALSlopeLineFitFilter( LALStatus *status,
REAL4Vector* output_data,
const REAL4Vector* input_data,
const SLOPEFilterParams fparams )

void LALSlopeConvolutionFilter( LALStatus *status,
REAL4Vector* output_data,
const REAL4Vector* input_data,
const SLOPEFilterParams fparams )
```

Description

The three functions prototyped above implement time domain linear and nonlinear filters on input data. The resulting output data is intended to enhance sensitivity to burst sources.

The first function is `void LALSlopeDetectorFilter()`, which takes input data through its third argument `const REAL4Vector* input_data`. For each element \( i \) of the input data ntuple \( x_i \), the best fit slope to the data points \( i \) to \( i + N - 1 \) is found, where \( N \) is the fourth argument `ntaps` passed to the function. This function provides no provision for history buffers. The output data is written to the address `REAL4Vector* output_data` passed as the second argument.

The second function is `LALSlopeLineFitFilter()`, which also takes a `const REAL4Vector* input_data` pointer to input data as its third argument. For each element \( i \) of the input data ntuple \( x_i \), the best fit slope, \( a_i \), or the best fit intercept to the data (vertical) axis, \( b_i \), or a particular nonlinear combination of \( a_i \) and \( b_i \) is found. Any of these three quantities can be returned to the output address `REAL4Vector* output_data`, depending on the argument passed to `fparams.function_select`. Acceptable values for this parameter are shown in table 19.2.

The third function is `LALSlopeConvolutionFilter()`. This function convolves input data with time domain finite impulse response (FIR) filters of four types. Before running the filter on data, the filter taps must be set. Allocate enough memory to hold the number of filter taps you want. Set `fparams.forder` to be equal to the number of taps. Set the `fparams.function_select` to one of the values given in table 19.3.

The `BOXCAR` option creates \( N \) taps that are all set to \( 1/N \). The `GAUSSIAN` option creates taps that represent part of a gaussian curve between \( \pm 3\sigma \). The `SINE` option sets the taps to be one period of a sine wave. Both the sine and Gaussian tap waveforms are of unity height. If the `USER` option is set, then the `input data->length` field is used to set the taps. In the latter case, the `input data->length` field must be the same as `fparams.forder`, and equal the number of elements in the input data array. In all cases, memory must be allocated by

<table>
<thead>
<tr>
<th>fparams.function_select</th>
<th>value</th>
<th>filter output</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILTER_OUTPUT_SLOPE</td>
<td>1</td>
<td>fit to slope ( a_i ) over ( N ) bins</td>
</tr>
<tr>
<td>FILTER_OUTPUT_OFFSET</td>
<td>2</td>
<td>fit to offset ( b_i ) over ( N ) bins</td>
</tr>
<tr>
<td>FILTER_OUTPUT_ALF</td>
<td>3</td>
<td>ALF filter (see algorithms)</td>
</tr>
</tbody>
</table>

Table 19.2: settings for `fparams.function_select` used in conjunction with function `LALSlopeLineFitFilter()`.
### Table 19.3: settings for \( \text{fparams.function_select} \) used in conjunction with function `LALSlopeConvolutionFilter()`.

<table>
<thead>
<tr>
<th>( \text{fparams.function_select} )</th>
<th>value</th>
<th>filter output</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>FILTER_OUTPUT_BOXCAR</code></td>
<td>4</td>
<td>set taps to uniform window over N bins</td>
</tr>
<tr>
<td><code>FILTER_OUTPUT_GAUSSIAN</code></td>
<td>5</td>
<td>set taps to Gaussian over N bins</td>
</tr>
<tr>
<td><code>FILTER_OUTPUT_SINE</code></td>
<td>6</td>
<td>set taps to sine wave period over N bins</td>
</tr>
<tr>
<td><code>FILTER_OUTPUT_USER</code></td>
<td>7</td>
<td>set taps to user defined</td>
</tr>
<tr>
<td><code>FILTER_OUTPUT_CONVOLVE</code></td>
<td>8</td>
<td>convolve data with a filter already set up</td>
</tr>
</tbody>
</table>

the calling function to the \( \text{fparams.taps_set} \) field, which must be initialized to zero. Once the function is called to set the taps, the \( \text{fparams.taps_set} \) field is changed to one.

Once the filter has been initialized, it can be convolved with data. History buffering is supported, and works in the same way as for the `LALSlopeLineFitFilter` function described above. Note that the \( \text{fparams.function_select} \) field must be set to `FILTER_OUTPUT_CONVOLVE`.

### Algorithm

The algorithms used for applying filters with the three functions described above are given below. For all formulae, \( x_i \) represents an array of input data. The filters embedded in the functions `LALSlopeDetectionFilter` and `LALSlopeLineFitFilter` work by fitting each overlapping set of N successive data points to a straight line \( a_i x_i^j + b_i \). The fit parameters \( a_i \) and \( b_i \) are used as the basis to form discriminants. The `LALSlopeConvolutionFilter` function convolves input data either with a standard set of waveforms, or with waveforms set by the user as described above. In all the equations below, \( N \) is the number of successive data samples used to compute the quantities, the filter ‘order’.

For `LALSlopeDetectionFilter` the output data array is \( y_i \):

\[
y_i = \frac{1}{N} \sum_{j=0}^{N-1} x_i^j t_{i+j} - \left( \frac{1}{N} \sum_{k=0}^{N-1} x_i^k \right) \left( \frac{1}{N} \sum_{m=0}^{N-1} t_{i+m} \right)
\]

\[
\frac{1}{N} \sum_{p=0}^{N-1} t^2_{i+p} - \left( \frac{1}{N} \sum_{q=0}^{N-1} t_{i+q} \right)^2
\]

(19.1)

For `LALSlopeLineFitFilter` with \( \text{fparams.function_select} \) set to `FILTER_OUTPUT_SLOPE`, the output data \( a_i \) is

\[
a_i = \frac{12}{\tau N (N^2 - 1)} \left( \sum_{j=0}^{N-1} j x_j - \frac{N - 1}{2} \sum_{k=0}^{N-1} x_k \right),
\]

(19.2)

where \( \tau \) is the sampling period. Apart from a factor of \( \tau \), equations [19.1] and [19.2] are equivalent, since we have the following identities:

\[
\sum_{j=0}^{N-1} j = \frac{N(N - 1)}{2}
\]

(19.3)

\[
\sum_{j=0}^{N-1} j^2 = \frac{N}{6} (2N - 1)(N - 1)
\]

Note that slope filters have a minimum order of 2, since it takes at least 2 points to estimate the slope.

For `LALSlopeLineFitFilter` with \( \text{fparams.function_select} \) set to `FILTER_OUTPUT_OFFSET`, \( a_i \) is calculated and used to estimate the output \( b_i \) given by

\[
b_i = \sum_{j=0}^{N-1} x_j - \frac{a_i \tau (N - 1)}{2}.
\]

(19.4)

For `LALSlopeLineFitFilter` with \( \text{fparams.function_select} \) set to `FILTER_OUTPUT_ALF`, a nonlinear combination of \( a_i \) and \( b_i \) is constructed. Define \( \sigma_a \) and \( \sigma_b \):
\begin{align}
\sigma_a &= \frac{12}{\tau N(N^2 - 1)} \\
\sigma_b &= \frac{4N + 2}{N(N - 1)}. \tag{19.5}
\end{align}

Let \( X_a = a_i/\sigma_a \) and \( X_b = b_i/\sigma_b \). Then the output \( c_i \) of the ALF filter is
\begin{equation}
c_i = \frac{X_a^2 + X_b^2 - 2\alpha X_a X_b}{1 - \alpha^2}, \tag{19.6}
\end{equation}
where
\begin{equation}
\alpha = -\sqrt{\frac{3(N + 1)}{2(2N + 1)}}. \tag{19.7}
\end{equation}

This nonlinear combination was found by the Orsay group to be particularly sensitive to bursts in the Zwerger Mueller catalog. See gr/qc-0010037 for a full description.

The remaining filters use a standard convolution algorithm to convolve input data with the waveforms described above (boxcar, gaussian, sinewave, user). The convolution of input data \( x_i \) with filter taps \( c_j \) is
\begin{equation}
y_i = \sum_{j=0}^{N-1} x_{i+j} c_j \tag{19.8}
\end{equation}

These functions do not use any other LAL routines.

Notes

All pointers passed to these functions must be preallocated by the calling routine to hold the objects written to them. In particular, the pointer to the output data must be of size \( \text{input\_data\_length} - \text{filter\_order} + 1 \), where \( \text{filter\_order} \) is \( \text{ntaps} \) in the function \text{LALSlopeDetectorFilter} and \( \text{fparams\_forder} \) in \text{LALSlopeLineFitFilter} and \text{LALSlopeConvolutionFilter}. Unused parameters that are pointers can be set to \text{NULL} (for example, the \( \text{fparams.tap} \) field can be null if calling the \text{LALSlopeLineFitFilter} function.

The functions \text{LALSlopeLineFitFilter} and \text{LALSlopeConvolutionFilter} support the passing of a history buffer between successive applications of a filter. The history buffer pointer needs to be allocated by the calling routine when using these functions. Enough memory must be allocated to it to hold \( \text{fparams\_forder} - 1 \) \text{REAL4} elements. After running the filter, the history buffer will contain the last \( \text{fparams\_forder} - 1 \) elements of the input data sent to the filter. By adding these elements to the beginning of the next set of input data sent to the filter, a continuous filtered data stream with a seamless boundary may be maintained between successive applications of the filter. Note that currently the passing of history buffers between successive MPI jobs is not supported under LDAS.

Both \text{LALSlopeLineFitFilter} and \text{LALSlopeConvolutionFilter} require the field \( \text{fparams\_sampling\_period\_s} \) to contain the reciprocal of the sampling frequency for the channel.

The \text{waveform\_offset} field in \text{SLOPEFilterParam} is there in case you need to offset the filter waveform by a fraction of a frequency bin (eg, if you want the maximum of a gaussian filter to fall equally between two bins, you would set this field to 0.5).
19.1.2 Program SlopeDetectorFilterTest.c

This test program ensures that all errors are flagged correctly.

Usage

SlopeDetectorFilterTest

Description

Exit codes

<table>
<thead>
<tr>
<th>name</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>0</td>
<td>&quot;Test finished OK&quot;</td>
</tr>
<tr>
<td>NOM</td>
<td>1</td>
<td>&quot;Error checking failed for bad input data&quot;</td>
</tr>
<tr>
<td>CHK</td>
<td>2</td>
<td>&quot;Error checking failed for bad output data pointer&quot;</td>
</tr>
<tr>
<td>FLS</td>
<td>3</td>
<td>&quot;Error checking failed for input data shorter than number of filter taps&quot;</td>
</tr>
<tr>
<td>COR</td>
<td>4</td>
<td>&quot;Returned unexpected error upon calling function with good data&quot;</td>
</tr>
<tr>
<td>BADVECTOR</td>
<td>5</td>
<td>&quot;Error making input vector&quot;</td>
</tr>
<tr>
<td>BADFREE</td>
<td>6</td>
<td>&quot;Error freeing allocated vector&quot;</td>
</tr>
<tr>
<td>INCOMPATIBLEOUTPUT</td>
<td>7</td>
<td>&quot;Filter output differs from expected&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants SLOPEDETECTORFILTERTESTC_E<name>, and the status descriptions in SLOPEDETECTORFILTERTESTC_MSGE<name>. The source code with these messages is in SlopeDetectorFilterTest.c on line 1.87.

Uses

LALSlopeDetectorFilter()

Notes

Author: Daw, E. J.

$Id: SlopeDetectorFilterTest.c,v 1.7 2007/06/08 14:41:52 bema Exp $
Chapter 20

Package: tfclusters

This package implements a nonlinear search algorithm for arbitrary transients, by (i) applying a threshold on a spectrogram, (ii) applying a set of clustering analysis algorithms, and (iii) applying a threshold on the total power in the cluster.

• (i) Thresholded Spectrogram: The spectrogram is computed from stacked periodograms, i.e., from the norm square of normalized Fourier transforms of adjacent, non-overlapping segments of data. No window is used to multiply the data before the transform. A threshold (which could be frequency dependent) is applied on the power in the spectrogram, to give a binary map representation of the time-frequency plane. By convention, pixels with power above the threshold will be called black pixels, and those below will be white pixels. In general, the threshold on the power should be chosen so that every pixel in the binary map has an equal probability of being black.

• (ii) Clustering Analysis: The pixels in the time-frequency plane are labelled by two positive integers \((i, j)\) referring respectively to their time and frequency indices. The distance between two pixels is defined by \(d_{12} = |i_1 - i_2| + |j_1 - j_2|\). Pixels are said to be nearest neighbours if \(d_{12} = 1\). A cluster is a set of one or more black pixels such that each black pixel has at least one black neighbour at a distance of 1, and such that no black pixel outside the set is at a distance of 1 of any pixel in the set. The size of a cluster is the cardinality of the set. The first threshold to be applied on clusters is that the size \(s \geq \sigma\), with \(\sigma \geq 1\). For \(\sigma > 1\), a second set of threshold is applied on clusters with \(s < \sigma\): All pairs of clusters are considered. For a pair with sizes \(s_1\) and \(s_2\), if \(d_{12} \leq \delta_{s_1, s_2}\), the two clusters are merged and are called a generalized cluster. \(\delta_{s_1, s_2}\) is an integer; there are \(\sigma(\sigma - 1)/2\) such numbers. When a given cluster can pair with more than one other cluster and satisfy the distance thresholds, all these clusters are merged together into a single generalized cluster. By an abuse of language, ‘generalized clusters’ will often be called simply ‘clusters’ below.

• (iii) Power Threshold: For a given (generalized) cluster of size \(s\), its total power \(P\) is defined as the sum of the power over all its pixels. If the data are Gaussian noise, and if the threshold on the spectrogram power in step (i) above was \(T\), then \(P - sT \sim \chi^2_{2s}\), i.e., is a chi-square variable with \(2s\) degrees of freedom. The third threshold is therefore \(Q(P - sT, 2s) < \alpha\), with \(0 < \alpha < 1\) and \(Q(x, a) = \int_x^{\infty} e^{-t^{a-1}} dt/\Gamma(a)\). Any cluster that has survived cuts (i) and (ii) and for which this inequality is true will be called a significant event.
20.1 Header \texttt{TFClusters.h}

Provides function prototypes for running a simple transient detection algorithm.

Synopsis

\texttt{#include <lal/TFClusters.h>}

This header provides the necessary structure definitions that are used by the code, together with function prototypes.

Error Conditions

\begin{tabular}{|c|c|p{10cm}|}
\hline
\texttt{name} & \texttt{code} & description	\
\hline
\texttt{NULLP} & 1 & "Null pointer"	\
\texttt{NNULLP} & 2 & "Non-null pointer"	\
\texttt{STRICTPOS} & 4 & "Argument must be strictly positive"	\
\texttt{POS} & 8 & "Argument must be positive"	\
\texttt{INCOMP} & 16 & "Time Series parameters incompatible with requested time-frequency parameters"	\
\texttt{MALLOC} & 32 & "Memory allocation error"	\
\texttt{NZERO} & 64 & "Non-zero parameter"	\
\texttt{01} & 128 & "Argument must be in [0,1]"	\
\texttt{IARG} & 256 & "Invalid Argument"	\
\texttt{MAXITE} & 512 & "Maximum number of iteration exceeded"	\
\hline
\end{tabular}

The status codes in the table above are stored in the constants \texttt{TFCLUSTERSH\_E<name>}, and the status descriptions in \texttt{TFCLUSTERSH\_MSGE<name>}. The source code with these messages is in \texttt{TFClusters.h} on line 1.96.
Structures

struct TFPlaneParams
A description of a time-frequency plane.

INT4 timeBins Number of time bins in TF plane.
INT4 freqBins Number of frequency bins in TF plane.
REAL8 deltaT Time resolution of the plane.
REAL8 deltaF Frequency resolution of the plane.
REAL8 flow Low-frequency boundary of the plane.

struct TFCSpectrogram
A container for the power in the spectrogram.

TFPlaneParams *params Parameters of the spectrogram.
REAL8* power A pointer to the power vector: power at time index i and frequency index j is given by
power[i*params->freqBins + j].

struct CList
A container for the clusters that are detected.

UINT4 nclusters The number of clusters.
UINT4* sizes Vector of cluster sizes.
UINT4** t Time coordinates: time index of jth pixel in ith cluster is t[i][j].
UINT4** f Frequency coordinates: frequency index of jth pixel in ith cluster is f[i][j].
REAL8** P Instantaneous power: power of jth pixel in ith cluster is P[i][j].

struct CListDir
A container for the various thresholds

UINT4 freqBins Number of frequency bins in spectrogram.
REAL8* rho Vector of size freqBins containing the threshold on the power for the first cut, as a function
of the frequency index.
REAL8 minf Minimum frequency in Hertz to be considered in the analysis.
REAL8 maxf Maximum frequency in Hertz to be considered in the analysis. All clusters with at least one
pixel outside of [minf, maxf] are rejected.
UINT4 sigma Threshold σ on cluster size.
UINT4* s1,s2 Size pairs for distance threshold.
UINT4* d Vector of distance thresholds δ_{s1,s2}. For a certain value of i in [0,σ(σ−1)/2−1], δ_{s1,s2} =d[i]
for s1 =s1[i] and s2 =s2[i].
UINT4 mdist Maximum values of |d|.
REAL8 alpha For white Gaussian noise at input, fraction of clusters that pass the last cut on the total
cluster power.
20.1.1 Module TFClusters.c

Compute the spectrogram from a time series.

Prototype

```c
void LALComputeTFCSpectrogram ( LALStatus *status, TFCSpectrogram *out, TFPlaneParams *tspec, REAL4TimeSeries *tseries )
```

Description

Computes the spectrogram \( *out \) for the time series \( *tseries \), using the parameters defined in \( *tspec \). FFTs can overlap if \( \text{deltaT} \times \text{timeBins} \) is larger than the time series duration; if they do overlap, a Welch window is applied. The power is the norm square of the (normalized) discrete Fourier transform.

Uses

- LALCreateForwardRealFFTPlan()
- LALCreateVector()
- LALForwardRealFFT()
- LALCDestroyVector()
- LALDestroyRealFFTPlan()

Author: Sylvestre, J

$Id: TFClusters.c,v 1.31 2007/06/08 14:41:56 bema Exp$

Compute the cross-spectrogram from a time series.

Prototype

```c
void LALComputeXTFCSpectrogram ( LALStatus *status, TFCSpectrogram *out, TFPlaneParams *tspec, REAL4VectorTimeSeries *tseries )
```

Description

Computes the cross-spectrogram \( *out \) for the time series \( *tseries \), using the parameters defined in \( *tspec \). FFTs can overlap if \( \text{deltaT} \times \text{timeBins} \) is larger than the time series duration; if they do overlap, a Welch window is applied. The power is the norm square of the (normalized) discrete Fourier transform.

Uses

- LALCreateForwardRealFFTPlan()
- LALCreateVector()
- LALForwardRealFFT()
- LALCDestroyVector()
- LALDestroyRealFFTPlan()

Author: Sylvestre, J

$Id: TFClusters.c,v 1.31 2007/06/08 14:41:56 bema Exp$
Apply the first two levels of thresholds: (i) cut on power of individual pixels in spectrogram and (ii) cut on cluster sizes in thresholded spectrogram.

Prototype

```c
void LALGetClusters ( LALStatus *status, CList *clist, TFCSpectrogram *tpower, CListDir *dir )
```

Description

First, this function transforms `tpower` into a binary map, by applying the frequency dependent thresholds `dir->rho` on the power in the spectrogram. Only frequencies up to `dir->maxf` are retained. A recursive function is then called to identify the clusters on a ‘nearest neighbours’ basis (i.e., pixels touching by one ‘edge’). Only clusters with power strictly between `dir->minf` and `dir->maxf` are kept (if `dir->maxf` is negative, only clusters with at least some power between `dir->minf` and `dir->maxf` are kept). Clusters larger or equal to `dir->sigma` are sent to `clist`. The remaining clusters are grouped in pairs. Whenever a pair pass the distance thresholds defined by `dir->s1`, `dir->s2` and `dir->d`, the two clusters are fused and are added as a single cluster to `clist`.

Uses

- `LALInitCList()`
- `LALFreeCList()`

Notes

- `clist` must be initialized by a proper call to `LALInitCList()` before calling this function.
- Calling this function destroys `tpower`.
- `dir->rho[0]` corresponds to `minf`, not DC.
Apply the final cut by thresholding on the total power in the clusters.

Prototype

```c
void LALClustersPowerThreshold ( LALStatus *status, CList *out, CList *in, CListDir *dir )
```

Description

This function loops over all clusters in `*in`; for each cluster it computes its total power by summing over the pixels of the cluster, and computes the probability for Gaussian noise to produce a cluster with this total power at this stage of the analysis. This probability is compared to `dir->alpha`; if smaller, the cluster from `*in` is appended to `*out`. Therefore, `dir->alpha` is the fraction of clusters that had survive the first cuts that will pass this one, assuming Gaussian noise as input of the algorithm. When `dir->alpha` ≤ 0, only clusters which have at least one pixel with power larger or equal to `-dir->alpha` times the first power threshold will survive.

Notes

- `*out` must be initialized by a proper call to `LALInitCList()` before calling this function.
Merge two cluster lists.

Prototype

```c
void LALMergeClusterLists ( LALStatus *status, CList *out, CList *A, CList *B )
```

Description

Merge *A* and *B* into cluster list *out*. The merging is done so that any two clusters that overlapp or that have black pixels that are nearest neighbors will be replaced by the union of the two clusters in *out*. The clusters that don’t satisfy these two conditions are just copied into *out*.

Uses

- `LALCopyCList()`
- `LALInitCList()`
- `LALFreeCList()`

Notes

- *out* must be initialized by a proper call to `LALInitCList()` before calling this function.
Make a copy of a cluster list.

Prototype

```c
void LALCopyCList ( LALStatus *status, CList *dest, CList *src );
```

Description

Make a copy of *src onto *dest.

Uses

`LALFreeCList()`

Notes

- *src, if not empty, will be overwritten.
- *dest must be initialized by a proper call to `LALInitCList()` before calling this function.
Initialize a spectrogram with default values.

Prototype

```c
void LALPlainTFCSpectrogram( LALStatus *status, TFPlaneParams *tspec, REAL4TimeSeries *tseries, REAL8 T )
```

Description

Initialize the spectrogram `*tspec` so that it has a time resolution `T` and frequency resolution `1/T`, with frequency ranging from `1/T` to the Nyquist frequency of the time series `*tseries`. Also set the length of `*tspec` so it matches `*tseries`. 
Initialize a spectrogram with default values.

Prototype

```c
void LALPlainTFCSpectrogramWin( LALStatus *status, TFPlaneParams *tspec, REAL4TimeSeries *tseries, REAL8 T )
```

Description

Initialize the spectrogram *tspec* so that it has a time resolution T/2 and frequency resolution 1/T, with frequency ranging from 1/T to the Nyquist frequency of the time series *tseries.*
Initialize a cluster list structure.

**Prototype**

```c
void LALInitCList ( LALStatus *status, CList *clist, TFPlaneParams *tspec )
```

**Description**

Initialize `clist` and set its parameters to `tspec`. 
Initialize a threshold structure.

Prototype

`void LALFillCListDir ( LALStatus *status, CListDir *cldir, REAL8 rho )`

Description

Initialize `cldir`. This means allocating memory for `cldir->s1, cldir->s2, cldir->d` according to the value of `cldir->sigma`, and for the threshold vector `cldir->rho`; all the values of `cldir->rho` are initialized to `rho`.

Notes

- Before calling this function, `cldir->sigma` and `cldir->freqBins` must be set to their desired value.
Functions to destroy the different structures.

Prototype

```c
void LALFreeCList( LALStatus *status, CList *clist )
void LALFreeSpecgram( LALStatus *status, TFCSpectrogram *power )
void LALFreeCListDir( LALStatus *status, CListDir *cdir )
```

Description

Release allocated memory.
Operating Instructions

```c
#include <TFClusters.h>
#include <Random.h>

static LALStatus status;
REAL4TimeSeries tseries;
CListDir dir;
CList clist, list;
TFPlaneParams tspec;
TFCSpectrogram spower;

RandomParams *params;
REAL4Vector *vect;

REAL8 T, P;
UINT4 i, j, N;
INT4 seed = 0;

// first generate a time series of white Gaussian noise of unit variance
N = 1024 * 128;
LALCreateVector(&status, &vect, N);
LALCreateRandomParams(&status, &params, seed);
LALNormalDeviates(&status, vect, params);

// set epoch and sample time
Tseries.epoch.gpsSeconds = 0;
tseries.epoch.gpsNanoSeconds = 0;
tseries.deltaT = 1.0/1024.0;
tseries.f0 = 0.0;
tseries.data = vect;

// Next compute a spectrogram for the time series
T = 1.0; // this is the resolution in seconds of the spectrogram
LALPlainTFCSpectrogram(&status, &tspec, &tseries, T); // this creates spectrogram parameters at
// the 'Heisenberg limit' from DC+1/T to the Nyquist frequency
LALComputeTFCSpectrogram(&status, &spower, &tspec, &tseries);

// Set thresholds
dir.freqBins = tspec.freqBins; // number of frequency bins in spectrogram
dir.sigma = 5; // threshold on cluster size
dir.minf = 0; // min frequency to consider (Hz)
dir.maxf = 512; // max frequency to consider (Hz)
LALFillCListDir(&status, &dir, -log(0.1)); // allocate memory and set the threshold on power so
// that 1 every 10 pixel is black

// set thresholds on distance for different size couples
dir.d[0] = 0; // 1,1
dir.d[1] = 0; // ...
dir.d[2] = 0;
dir.d[3] = 0; // 1,4
dir.d[4] = 0; // 2,2
dir.d[5] = 0;
dir.d[6] = 2; // 2,4
```
dir.d[7] = 3; 3,3
dir.d[8] = 4; 3,4
dir.d[9] = 4; 4,4

dir.mdist = 4; no need to worry about things that are more than 4 units away from each other

run cluster threshold algorithm
LALInitCList(&status, &clist, &tspec); initialize clist

LALGetClusters(&status, &clist, &spower, &dir); generate list of clusters

LALFreeSpecgram(&status, &spower); spectrogram no longer useful

run threshold on cluster total power
dir.alpha = 0.25; only 1/4 of all clusters from white noise will make it

LALInitCList(&status, &list, &tspec); initialize list

LALClustersPowerThreshold(&status, &list, &clist, &dir); generate new list

clean up a bit
LALFreeCList(&status, &clist);
LALFreeCListDir(&status, &dir);

display results to stdout
printf("Id/t/tSize/t/tPower/n");
for(i=0; i<list.nclusters; i++)
for(P=0, j=0; j<list.sizes[i]; j++) P += list.P[i][j];
printf("%i/t/t%i/t/t%g/n",i,list.sizes[i],P);
clean up
LALFreeCList(&status, &list);
20.1.2 Program **TFClustersTest1.c**

Sample usage of the routines in **TFClusters.h**.

**Usage**

**TFClustersTest1**

**Description**

Generates 128 seconds of white Gaussian noise at 2048 Hz, and apply the three levels of thresholding. Writes a list of clusters on stdout.

**Exit codes**

Returns 0 on success, otherwise returns 1.

**Uses**

**Notes**
20.2 Header **TFCThresholds.h**

Provides function prototypes for computing the first power thresholds for non-white noise.

**Synopsis**

```c
#include <lal/TFCThresholds.h>
```

This header provides the necessary structure definitions that are used by the code, together with function prototypes.

**Error Conditions**

<table>
<thead>
<tr>
<th>name</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULLP</td>
<td>0</td>
<td>&quot;Null pointer&quot;</td>
</tr>
<tr>
<td>01</td>
<td>128</td>
<td>&quot;Argument must be in [0,1]&quot;</td>
</tr>
<tr>
<td>EGOAL</td>
<td>129</td>
<td>&quot;Error goal smaller than numerical precision&quot;</td>
</tr>
<tr>
<td>NERR</td>
<td>130</td>
<td>&quot;A numerical error occurred&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants `TFCTHRESHOLDSH_E<name>`, and the status descriptions in `TFCTHRESHOLDSH_MSGE<name>`. The source code with these messages is in `TFCThresholds.h` on line 1.60.
Structures

struct RiceThreshold

A container for the parameters used in the computation of the thresholds.

nFreq Number of frequency bins.

meanRe Mean values of the real part of the Fourier transform of the data, as a function of frequency (size = nFreq).

varRe Variances of the real part of the Fourier transform of the data, as a function of frequency (size = nFreq).

meanIm, varIm As above for imaginary part.

bpp The target black pixel probability.

eGoal The absolute error goal on the power thresholds.
20.2.1 Module **TFCThresholds.c**

Computes thresholds on the power from a best fit to the Rice distribution.

**Prototype**

```c
void LALTFCRiceThreshold ( LALStatus *status, REAL4* rho, RiceThresholdParams* thr );
```

**Description**

Computes the thresholds on the power at every frequency, using a best fit to the Rice distribution.

**Notes**

- `rho` must be pointing to allocated memory.
Chapter 21

Package `waveburst`

A collection of routines needed for Waveburst DSO.
21.1 Header LALWavelet.h

Provides structures for wavelets, clusters, pixels. Declares prototypes of all the functions used by waveburst DSO.

Synopsis

```
#include <lal/LALWavelet.h>
```

This package provides all the structures and procedures needed to work with wavelets, do percentile transform of wavelets, perform coincidence between two channels, cluster the remaining pixels and select only those events that satisfy certain criteria.

Error conditions

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULLP</td>
<td>1</td>
<td>&quot;Null pointer&quot;</td>
</tr>
<tr>
<td>TYPEMISMATCH</td>
<td>2</td>
<td>&quot;Wavelet type mismatch&quot;</td>
</tr>
<tr>
<td>DIFF</td>
<td>3</td>
<td>&quot;Difference between computed and expected values exceeds the threshold&quot;</td>
</tr>
<tr>
<td>NONZEROREMAINDER</td>
<td>4</td>
<td>&quot;Non-zero remainder&quot;</td>
</tr>
<tr>
<td>OUTOFBOUNDS</td>
<td>5</td>
<td>&quot;Array out of bounds&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants LALWAVELETH_E<name>, and the status descriptions in LALWAVELETH_MSGE<name>. The source code with these messages is in LALWavelet.h on line 1.91.

Structures

```
struct Slice
```

Slice structure is used to extract each stepth element from an array starting from start.

- **UINT4 start**: Index to start with.
- **UINT4 size**: Number of elements in the slice.
- **UINT4 step**: Skip step − 1 elements when choosing the next one.

```
struct Wavelet
```

Wavelet structure contains all the wavelet data and metadata.

- **enum WAVETYPE type**: Possible values: HAAR, BIORTHOGONAL, DAUBECHIES, SYMLET, DMeyer.
- **enum BORDER border**: Possible values: B_PAD_ZERO, B_CYCLE, B_MIRROR, B_PAD_EDGE, B_POLYNOM.
- **enum TREETYPE treeType**: Possible values: DIADIC, BINARY.
- **UINT4 level**: Wavelet decomposition level.
- **HPFilterLength**: Highpass filter length.
- **LPFilterLength**: Lowpass filter length.
- **REAL4TimeSeries *data**: Wavelet coefficients.
struct **PixelWavelet**

PixelWavelet describes pixel in two dimensional time frequency spectrogram.

**UINT4 time** Time index.

**UINT4 frequency** Frequency index.

**UINT4 clusterID** Id of the cluster the pixel belongs to.

**BOOLEAN core** Defines if the pixel is in the core or in the halo of the cluster.

**UINT4 neighbors[8]** List of neighbors indexed the same way as in pMask.

**UINT4 neighborsCount** The number of neighbors.

**REAL4 amplitude** Percentile amplitude.

**REAL4 amplitudeOriginal** Original amplitude.

struct **ClusterBlobWavelet**

ClusterBlobWavelet describes blobs of percentile and original amplitudes written to the database.

**UINT4 start_time_indx** Time index where the cluster starts.

**UINT4 stop_time_indx** Time index where the cluster ends.

**UINT4 start_freq_indx** Minimum frequency index of the cluster.

**UINT4 stop_freq_indx** Maximum frequency index of the cluster.

**UINT4 time_width** \( stop\_time\_indx - start\_time\_indx + 1 \).

**UINT4 freq_width** \( stop\_freq\_indx - start\_freq\_indx + 1 \).

**REAL4 *pBlob** Array of percentile amplitudes for the smallest rectangle containing the cluster.

**REAL4 *oBlob** Array of original amplitudes for the smallest rectangle containing the cluster.

struct **ClusterWavelet**

ClusterWavelet organizes non-zero pixels into clusters and contains various characteristics of those clusters.

**Wavelet *wavelet**** The wavelet for which clustering is done.

**REAL4 *medians** Medians of the amplitudes on each layer.

**UINT4 pMaskCount** Number of non-zero pixels that survived after percentile transform and coincidence.

**UINT4 clusterCount** Number of clusters after percentile transform, coincidence and initial selection on cluster size.

**UINT4 clusterCountFinal** Final number of clusters that survived all the selection criteria.

**INT4 clusterType** Currently possible values are: ORIGINAL_CL, PIXELMIXING_CL, SWAP_CL, SIMULATION_CL. At the moment this field is not yet actively used

**REAL4 delta_t** Wavelet time step.

**REAL4 delta_f** Wavelet frequency step.

**PixelWavelet **pMask** An array of all non-zero pixels (after percentilie transform and coincidence).

**UINT4 *cScuts** An array elements of which can take two values: 0 (the cluster has not passed some selection criteria) and 1.

**UINT4 **cList** An array of arrays that list pixels (in terms of pMask index) belonging to the corresponding cluster.
UINT4 *volumes An array of volumes \((\text{coreSize} + \text{halo})\) of clusters.

UINT4 *coreSize An array of core sizes of clusters.

REAL4 *correlation An array of correlations (asymmetries) for each cluster computed as follows:
\[
\text{NumberOfPositivePercentileAmplitudes} - \text{NumberOfNegativePercentileAmplitudes}
\]
\[\text{coreSize}\]

REAL4 *likelihood Likelihood of each cluster.

REAL4 *power Power of each cluster.

REAL4 *maxAmplitude Absolute maximum value of a pixel amplitude for each cluster.

REAL8 *relativeStartTime Start time of the cluster relative to the beginning of time series given as an input.

REAL8 *relativeStopTime Stop time of the cluster relative to the beginning of time series given as an input.

REAL8 *duration \(\text{relativeStopTime} - \text{relativeStartTime}\).

LIGOTimeGPS *absoluteStartTime Absolute start time of the cluster expressed as GPS time.

LIGOTimeGPS *absoluteStopTime Absolute stop time of the cluster expressed as GPS time.

REAL4 *startFrequency Minimum frequency of the cluster.

REAL4 *stopFrequency Maximum frequency of the cluster.

REAL4 *bandwidth \(\text{stopFrequency} - \text{startFrequency}\).

ClusterBlobWavelet *blobs The actual percentile or original amplitudes describing the cluster.

REAL4 nonZeroFractionAfterPercentile Fraction of non-zero pixels after percentile transform.

REAL4 nonZeroFractionAfterCoincidence Fraction of non-zero pixels after coincidence.

REAL4 nonZeroFractionAfterSetMask Fraction of non-zero pixels after setMask was applied and small cluster were zeroed out.

BOOLEAN pixelSwapApplied The flag that indicates whether pixel swap was applied.

BOOLEAN pixelMixerApplied The flag that indicates whether pixel mixing was applied.

struct InputLayerWavelet
InputLayerWavelet contains input for LALGetLayerWavelet function.

Wavelet *wavelet Wavelet.

UINT4 index Index of a layer to extract.

struct OutputLayerWavelet
OutputLayerWavelet contains output for LALGetLayerWavelet function.

REAL4TimeSeries *layer The extracted layer.

INT4 status Status. Not used at the moment.

struct InputGetMaxLayerWavelet
InputGetMaxLayerWavelet contains input for LALGetMaxLayerWavelet function.

Wavelet *wavelet Wavelet.
struct OutputGetMaxLayerWavelet

OutputGetMaxLayerWavelet contains output for LALGetMaxLayerWavelet function.
UINT4 maxLayer Maximum layer.

struct InputPercentileWavelet

InputPercentileWavelet contains input for LALPercentileWavelet function.
Wavelet *in Original wavelet.
REAL4 nonZeroFraction Fraction of pixels that survive percentil transform.

struct OutputPercentileWavelet

OutputPercentileWavelet contains output for LALPercentileWavelet function.
ClusterWavelet *out Transformed wavelet embedded into ClusterWavelet structure.

struct InputPixelSwapWavelet

InputPixelSwapWavelet contains input for LALPixelSwapWavelet function.
ClusterWavelet *in Original wavelet.

struct OutputPixelSwapWavelet

OutputPixelSwapWavelet contains output for LALPixelSwapWavelet function.
ClusterWavelet *out Wavelet in which each layer is swapped in time. It is used to estimate the accidental coincidence of glitches in two channels.

struct InputPixelMixerWavelet

InputPixelMixerWavelet contains input for LALPixelMixerWavelet function.
ClusterWavelet *in Original wavelet.
UINT4 seed Seed for random number generator.

struct OutputPixelMixerWavelet

OutputPixelMixerWavelet contains output for LALPixelMixerWavelet function.
ClusterWavelet *out Wavelet in which pixels are randomly mixed. It is used to estimate the accidental coincidence of glitches in two channels.

struct InputCoincidenceWavelet

InputCoincidenceWavelet contains input for LALCoincidenceWavelet function.
ClusterWavelet *one Wavelet from channel one.
ClusterWavelet *two Wavelet from channel two.
UINT4 timeWindowNanoSec Maximum time shift in nanoseconds between two channels.

struct OutputCoincidenceWavelet

OutputCoincidenceWavelet contains output for LALCoincidenceWavelet function.
ClusterWavelet *one Wavelet from channel one filtered by coincidence with channel two.
ClusterWavelet *two Wavelet from channel two filtered by coincidence with channel one.
REAL4 occupancyOne The resulting fraction of non-zero pixels in channel one.
REAL4 occupancyTwo The resulting fraction of non-zero pixels in channel two.
struct InputClusterWavelet

InputClusterWavelet contains input for LALClusterWavelet function.

BOOLEAN aura  This flag defines if we should use halo when clustering.

UINT4 minClusterSize Minimum cluster size.

UINT4 maxClusterSize Maximum cluster size.

ClusterWavelet *w  Wavelet to cluster (with percentile amplitudes, after coincidence).

Wavelet *original Original wavelet.

struct OutputClusterWavelet

OutputClusterWavelet contains output for LALClusterWavelet function.

ClusterWavelet *w  The resulting set of clusters.
21.1.1 Module LALWavelet.c

Prototypes

```c
void LALGetLayerWavelet(LALStatus *status,
                         OutputLayerWavelet **output,
                         InputLayerWavelet *input)

void LALGetMaxLayerWavelet(LALStatus *status,
                           OutputGetMaxLayerWavelet **output,
                           InputGetMaxLayerWavelet *input)

void LALPercentileWavelet(LALStatus *status,
                          OutputPercentileWavelet **output,
                          InputPercentileWavelet *input)

void LALPixelSwapWavelet(LALStatus *status,
                         OutputPixelSwapWavelet **output,
                         InputPixelSwapWavelet *input)

void LALPixelMixerWavelet(LALStatus *status,
                         OutputPixelMixerWavelet **output,
                         InputPixelMixerWavelet *input)

void LALCoincidenceWavelet(LALStatus *status,
                           OutputCoincidenceWavelet **output,
                           InputCoincidenceWavelet *input)

void LALClusterWavelet(LALStatus *status,
                       OutputClusterWavelet **output,
                       InputClusterWavelet *input)

void LALReuseClusterWavelet(LALStatus *status,
                            OutputClusterWavelet **output,
                            InputReuseClusterWavelet *input)

void LALAllocateWavelet(LALStatus *status,
                        Wavelet **wavelet)

void LALFreeWavelet(LALStatus *status,
                    Wavelet **wavelet)

void LALFreeREAL4TimeSeries(LALStatus *status,
                           REAL4TimeSeries **t)

void LALFreeREAL8TimeSeries(LALStatus *status,
                           REAL8TimeSeries **t)
```
void LALFreeClusterWavelet(LALStatus *status,
                        ClusterWavelet **w)

void LALFreeOutPercentile(LALStatus *status,
                          OutputPercentileWavelet **p)

void LALFreeOutCoincidence(LALStatus *status,
                          OutputCoincidenceWavelet **co)

void LALFreeOutCluster(LALStatus *status,
                        OutputClusterWavelet **cl)

void LALSetAmplitudesWavelet(LALStatus *status,
                             ClusterWavelet *w)

void LALAssignREAL4TimeSeries(LALStatus *status,
                              REAL4TimeSeries **left,
                              REAL4TimeSeries *right)

Description

Uses

Notes
Section 8

Inspiral Packages
Chapter 22

Package bank

This is a module that generates a template bank to search for inspiralling and merging binaries consisting of neutron stars and/or black holes. Template placement is based on the geometrical formalism \[5, 3\]. A user calls either \texttt{LALInspiralCreateCoarseBank}, to create a coarse bank, or \texttt{LALInspiralCreateFineBank}, to create a fine bank, at specified minimal matches. \texttt{LALInspiralCreateCoarseBank} first chooses templates along the $\eta = 1/4$ curve and then systematically lays a rectangular grid in the $\tau_0-\tau_2$ plane, or $\tau_0-\tau_3$ plane depending on user’s choice.

If the metric on a (two-dimensional) signal manifold characterised by parameters $(x_0, x_1)$ is $g_{ij}$ then the spacing between templates is given by

$$dx_0 = \sqrt{\frac{2(1-MM)}{g_{00}}}, \quad dx_1 = \sqrt{\frac{2(1-MM)}{g_{11}}}.$$  \hspace{1cm} (22.1)

### 22.1 Template Placement for Binary Inspiral Searches

Template placement is a problem of populating the binary parameter space (masses, spins, etc.) with as small a number of templates as possible, subject to the constraint that every signal that lies within the space has an overlap greater than or equal to a certain minimal match with at least one template in the grid. Past studies \[1, 2, 3\] have shown that this is most easily achieved by using the space of chirp-times to lay templates rather than the space of component masses.

The number of chirp times that one can define is determined by the post-Newtonian order one is working with. At the second post-Newtonian order (i.e. an approximation accurate to order $v^4$) and for a binary consisting of two non-spinning compact objects in a quasi-circular orbit, there are four chirp-times $\tau_k$, $k = 0, 2, 3, 4$, of which we can choose any two to characterize the binary:

$$\tau_0 = \frac{5M}{256\eta v_0^5} \quad (22.2)$$
$$\tau_2 = \frac{5M}{192\eta v_0^5} \left( \frac{743}{336} + \frac{11}{4}\eta \right) \quad (22.3)$$
$$\tau_3 = \frac{\pi M}{8\eta v_0^5} \quad (22.4)$$
$$\tau_4 = \frac{5M}{128\eta v_0^5} \left( \frac{3058673}{1016064} + \frac{5429}{1008}\eta + \frac{617}{144}\eta^2 \right) \quad (22.5)$$

where $m$ is the total mass of the binary, $\eta = m_1m_2/m^2$ is the symmetric mass ratio and $v_0 = (\pi mf_0)^{1/3}$ is a fiducial velocity parameter corresponding to a fiducial frequency $f_0$, usually chosen to be the lower frequency cutoff of the detector sensitivity.

This algorithm allows one to choose a coarse grid of templates either in the $\tau_0-\tau_2$ or $\tau_0-\tau_3$ space depending on the value of the \texttt{enum CoordinateSpace}, which can take one of two values: \texttt{Tau0Tau2} or \texttt{Tau0Tau3}. The shape of the coordinate spaces for some interesting range of masses is shown in Fig. 22.1. The important point to note in these figures is that the $\eta = 1/4$ curve spans from the minimum to the maximum value of the Newtonian chirp-time $\tau_0$. This feature will be used in the construction of the grid. Note that the minimum

\footnote{We use units in which $c = G = 1$; thus $1M_\odot = 4.92549095 \times 10^{-6}$ s.}
Figure 22.1: The parameter space binary masses and corresponding chirp-times. Chirp times are computed using an $f_0$ of 40 Hz. Note that our parameter is specified by $m_1, m_2 > m_{\text{min}}$ and $M = m_1 + m_2 < M_{\text{max}}$ rather than by $m_{\text{min}} < m_1, m_2 < m_{\text{max}}$. (We use capital $M$ to denote the total mass and lower-case $m$ to denote the component masses.) In the above example $m_{\text{min}} = 0.2 M_{\odot}$ and $M_{\text{max}} = 100 M_{\odot}$.

(maximum) value of the Newtonian chirp-time occurs when the two masses are equal and the total mass is a maximum (minimum).

### 22.1.1 Coarse Grid Algorithm: the square placement

The coarse grid algorithm used in this module is most economically described by the following main steps in the algorithm:

1. Compute the minimum and maximum chirp-times corresponding to the search space: $(\tau_{0 \text{min}}, \tau_{0 \text{max}})$, $(\tau_{3 \text{min}}, \tau_{3 \text{max}})$ (or $(\tau_{2 \text{min}}, \tau_{2 \text{max}})$)

2. Choose a lattice of templates along the equal mass curve.

3. Lay a rectangular grid in the rectangular region defined by the minimum and maximum values of the chirp-times and retain only if (a) the point lies in the parameter space, OR (b) one of the vertices defined by the rectangle lies in the parameter space. If, instead of choosing the templates as specified in (b), we accept every template whose span has a non-zero overlap with the parameter space, then we would generate too many spurious templates, especially in the low-mass region. This is because of the following reason: We have chosen to lay templates along the equal-mass curve and the parameter space is very thin for large values of $\tau_0$ along this curve – the region where the ‘distance’ between the $m_1 = m_{\text{min}}$ curve and the equal-mass curve is smaller than one minimal-match. By laying templates in...

\[\text{In what follows we will only mention } \tau_3; \text{ however, the algorithm is itself valid, and has been implemented, in the case of } (\tau_0, \tau_2) \text{ too.}\]
a rectangular grid in the $\tau_0-\tau_3$ space we would be encountering templates in this region and just above the equal-mass curve none of whose vertices would be within the search space. By throwing away such templates there is no danger of creating any `holes' in the parameter space. Indeed, criterion (b) will help in filling the holes below the $m_1 = m_{\text{min}}$ curve that would be created by accepting templates that meet only criterion (a).

The algorithm begins by identifying the vertices at the boundary of the search space corresponding to the range of masses over which one intends to carry out a search. It first chooses a set of templates along the equal mass curve and then lays a rectangular lattice of templates in the rectangular region defined by $(\tau_0^{\min}, \tau_3^{\min}), (\tau_0^{\max}, \tau_3^{\min}), (\tau_0^{\max}, \tau_3^{\max}),$ and $(\tau_0^{\min}, \tau_3^{\max}).$

### Parameters specifying a coarse grid

The algorithm takes as input the structure `InspiralCoarseBankIn` and returns a pointer to an array of type `InspiralTemplateList` and the number of elements in the array. `InspiralCoarseBankIn` is designed to provide the coarse bank algorithm with information about the search space such as the minimum mass of component stars, maximum total mass, etc., as well as other parameters not directly used by the coarse grid algorithm but are needed by inspiral wave generation algorithms. This is so that the coarse grid algorithm can correctly set the members of the `InspiralTemplateList` structure. In particular, the members of the `InspiralCoarseBankIn` structure used by the coarse grid algorithm are:

- **REAL8 mMin**, minimum mass of the component stars
- **REAL8 MMax**, maximum total mass of the binary
- **REAL8 mmCoarse**, coarse grid minimum match
- **REAL8 fLower**, lower frequency cutoff to be used in the computation of the noise moments
- **REAL8 fUpper**, upper frequency cutoff to be used in the computation of the noise moments
- **void (*NoisePsd)(LALStatus *status, REAL8 *shf, REAL8 f)**, pointer to a function that returns the noise spectral density (in units of per Hz$^{-1}$); noismodels modules have built in noise power spectral densities `LALGEOpsd`, `LALLIGOIPsd`, `LALTAMAPsd` and `LALVIRGOPsd`
- **REAL8 order**, the post-Newtonian order to be used for wave generation the choices being `newtonian`, `onePN`, `onePointFivePN`, `twoPN`, `twoPointFivePN`, `threePN`, `threePointFivePN`
- **CoordinateSpace space**, the coordinate space in which templates are laid (the choices being `Tau0Tau2`, `Tau0Tau3`; at `onePN` order the only choice is `Tau0Tau2`).
- **etamin**, minimum value of $\eta$; must be pre-computed using the formula $\text{etamin} = m_{\text{Min}} * (M_{\text{Max}} - m_{\text{Min}}) / \text{pow}(M_{\text{Max}}, 2)$ before calling the `LALInspiralCreateCoarseBank`.

Additionally, the user may specify the fine bank minimal match `mmFine`, if `LALInspiralCreateFineBank` is called. A fine bank can be created only around grid points where the metric is known.

Other members of the `InspiralCoarseBankIn` structure that are not directly used by the coarse grid algorithm but are needed to correctly set all the members of the `InspiralTemplate` are:

- **REAL8 tSampling**, time-domain sampling rate in Hz.
- **REAL8 approximant**, the approximation method to be used for wave generation which can be anyone of `TaylorT1`, `TaylorT2`, `TaylorT3`, `TaylorF1`, `TaylorF2`, or `PadeT1`, `PadeF1`, `EOB`. Note that the placement algorithm is guaranteed to work with only TaylorF2 approximants. A detailed study of approximants that are well captured by this template placement algorithm is described elsewhere [4].
- **GridSpacing**, specify the type of placement to be used. For the physical template families placement, it must be `SquareNotOriented`, `Hexagonal`, `HybridHexagonal`. For the non-BCV case, it could be `Square`, `SquareNotOriented`, `Hexagonal`, `HexagonalNonOriented`. Concerning the physical case, the non oriented square placement, as described in this section, places the template on a square lattice but does not take into account the orientation of the ellipse, which might cause problem in some particular cases. The Hexagonal placement was then implemented taking into account the orientation of the ellipse. No square placement with the correct orientation has been implemented since the hexagonal is now used for in the LIGO analysis. Concerning the non-spinning BCV placement, all 4 cases are available. However, only the non oriented square placement and oriented hexagonal placement have been thoroughly tested as well.
22.1.2 Coarse Grid Algorithm: the hexagonal placement

This module has the ability to place the templates on an hexagonal lattice. This placement reduces by 30% (as expected) the number of templates used by the square lattice. Although the algorithm is different (and described in this documentation), it is based on the same structure and uses the same function to be called, namely `LALInspiralCreateCoarseBank`. The lattice is specified by the variable `gridSpacing` from the structure `InspiralCoarseBankIn`, where the user must set `gridSpacing` to `Hexagonal`.

22.1.3 Coarse Grid Algorithm: the hybrid hexagonal placement

A very similar placement place the template on an hexagonal lattice similarly to the previous hexagonal placement but has a slightly different placement when upper and lower boundaries of the parameter space are covered by a single template. In such a case, the hexagonal placement is replace by a placement along the bissectrice of the upper and lower boundaries. See the documentation for more details. This algorithm is called by setting `gridSpacing` to `HybridHexagonal`.

22.1.4 Coarse Grid algorithm: non physical placement (BCV)

There is also the ability to create a bank for the non-spinning BCV templates. See the documentation. If so, the user must set `approximant` to `BCV` instead of one of the physical approximant (EOB, TaylorT1, ...).
## 22.2 Header LALInspiralBank.h

Header file for the template placement codes.

### Synopsis

```c
#include <lal/LALInspiralBank.h>
```

This header file covers routines that are used in template placement.

### Error codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>1</td>
<td>&quot;Null pointer&quot;</td>
</tr>
<tr>
<td>MEM</td>
<td>2</td>
<td>&quot;Memory allocation failure&quot;</td>
</tr>
<tr>
<td>CHOICE</td>
<td>3</td>
<td>&quot;Invalid choice for an input parameter&quot;</td>
</tr>
<tr>
<td>DIVO</td>
<td>4</td>
<td>&quot;Division by zero&quot;</td>
</tr>
<tr>
<td>SIZE</td>
<td>8</td>
<td>&quot;Invalid input range&quot;</td>
</tr>
<tr>
<td>FRANGE</td>
<td>16</td>
<td>&quot;Limits outside range of frequency series&quot;</td>
</tr>
<tr>
<td>ORDER</td>
<td>32</td>
<td>&quot;Inappropriate PN order&quot;</td>
</tr>
<tr>
<td>GRIDSPACING</td>
<td>64</td>
<td>&quot;Inappropriate grid spacing parameter [SquareNotOriented or Hexagonal]&quot;</td>
</tr>
<tr>
<td>HEXAINIT</td>
<td>128</td>
<td>&quot;Empty bank. abnormal behaviour in HexaBank generation.&quot;</td>
</tr>
<tr>
<td>FCUT</td>
<td>5</td>
<td>&quot;Inappropriate cutoff frequency [SchwarzISCO, BKLISCO or ERD]&quot;</td>
</tr>
<tr>
<td>FHIGH</td>
<td>6</td>
<td>&quot;Final frequency is less than the low frequency cutoff.&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants `LALINSPIRALBANKH_E<name>`, and the status descriptions in `LALINSPIRALBANKH_MSGE<name>`. The source code with these messages is in `LALInspiralBank.h` on line 1.93.

### Enums

1. **CoordinateSpace:**

   ```c
typedef enum
   {
     Tau0Tau2,
     Tau0Tau3,
     Psi0Psi3,
     PTFIntrinsic,
     PTFFull
   }
   CoordinateSpace;
   ```

   Choose templates either in the \((\tau_0, \tau_2)\) or \((\tau_0, \tau_3)\) space. This is one of the members of the Inspiral-CoarseBankIn structure.

   This enum allows users to choose template bank either in the \((\tau_0, \tau_2)\) space of chirptimes (the choice made by `Tau0Tau2`) or in the \((\tau_0, \tau_3)\) space of chirptimes (the choice made by `Tau0Tau3`). This was implemented in releases before May 25, 2002. On May 25 we migrated to a new, slightly faster, computation of the metric in which, at present, only the choice `Tau0Tau3` can be made. Since October 2003 a new choice `Psi0Psi3` was added to handle BCV templates. In November 2007 two new choices were added: `PTFIntrincitc` is a PTF metric in only the intrinsic parameters (a \(4 \times 4\) matrix), and `PTFFull` is the PTF metric in the full parameter space (intrinsic and extrinsic parameters).

2. **InspiralBankMassRange:**
typedef enum
{
    MinComponentMassMaxTotalMass,
    MinMaxComponentMass,
    MinMaxComponentTotalMass
} InspiralBankMassRange;

An enum that appears in the InspiralCoarseBankIn structure which fixes the way templates are chosen: The choice MinComponentMassMaxTotalMass means the minimum of the component masses will be given by $m_{\text{min}}$ and maximum total mass is given by $M_{\text{max}}$ of the InspiralBankCoarseIn structure. The choice MinMaxComponentMass means the minimum of the components masses will be again fixed by $m_{\text{min}}$ and the maximum of the component masses is fixed by $m_{\text{max}}$ of the InspiralCoarseIn structure below.

3. FreqCut:

typedef enum
{
    SchwarzISCO,
    BKLISCO,
    ERD
} FreqCut;

An enum that lists all the formulas that can be used to specify an upper frequency cutoff. From lowest to highest, the choices are: SchwarzISCO, the innermost stable circular orbit (ISCO) for a test particle orbiting a Schwarzschild black hole. BKLISCO, a mass ratio dependent ISCO derived from estimates of the final spin of a merged black found in a paper by Buonanno, Kidder, and Lehner (arXiv:0709.3839). ERD, an effective ringdown frequency studied in Pan et al (arXiv:0704.1964) that was found to give good fit between stationary-phase templates and numerical relativity waveforms.

4. GridSpacing:

typedef enum
{
    SquareNotOriented,
    Square,
    HexagonalNotOriented,
    Hexagonal,
    HybridHexagonal,
    S2BCV
} GridSpacing;

This enum is set by the user to specify the type of placement requested. It can be Square, Hexagonal, SquareNotOriented, HexagonalNotOriented, S2BCV. The two first align the ellipse along the eigenvectors whereas the two next do not. The last is a square placement which was being used during S2 and is therefore obsolete and should not be used (feel free to remove it). Historically, we used the SquareNotOriented placement until S4. Then, in S5, we switched to the Hexagonal placement, which should be used for future searches.

5. Position:

typedef enum
{
    In,
    Above,
    Below,
    Out,
This enum can take the following values **In**, **Out**, **Below**, **Edge**, **Above** and is used *only* by the Hexagonal placement. It simply specifies the place of a point with respect to the parameter space. **Edge**, means that the ellipse covers two boundaries(upper and lower).

6. **InsidePolygon:**

```c
typedef enum
{
    False,
    True
} InsidePolygon;
```

This enum is set to true or false, it is just a boolean variable for the purpose of BCV placement but can be used in an other context..

7. **Generation**

```c
typedef enum
{
    Sterile,
    Fertile
} Generation;
```

This enum is either **fertile**, **sterile**, and is a boolean expression used *only* by the Hexagonal placement.

**Structures**

1. **InspiralMetric** Structure to store metric at various points the signal manifold.

```c
typedef struct
tagInspiralMetric
{
    REAL8 G00;
    REAL8 G11;
    REAL8 G01;
    REAL8 g00;
    REAL8 g11;
    REAL8 theta;

    REAL4 Gamma[10];
}
```

/* Gamma[] is a vector that stores the upper triangular part of the metric in
 * the space of parameters. For time domain searches, Gamma[0,...,5] stores
 * the following information :
 * Gamma[0] -> (tc,tc) metric component
 * Gamma[1] -> (tc,t0) metric component
 * Gamma[2] -> (tc,t3) metric component
 * Gamma[3] -> (t0,t0) metric component
 * Gamma[4] -> (t0,t3) metric component
 * Gamma[5] -> (t3,t3) metric component
 * For spinBCV searches, (in 4 dimensions) Gamma[0,...,9] would be required.
 */
REAL4 Gamma[10];
We store the diagonalized metric together with the angle theta between the $\tau_0$-axis and the semi-major axis of the ambiguity ellipse. The members of this structure are:

- **G00**: 00-component of the metric in $(\tau_0, \tau_2(3))$ coordinates.
- **G11**: 11-component of the metric in $(\tau_0, \tau_2(3))$ coordinates.
- **G01**: 01-component of the metric in $(\tau_0, \tau_2(3))$ coordinates.
- **g00**: 00-component of the diagonalised metric.
- **g11**: 11-component of the diagonalised metric.
- **theta**: Angle from tau0 to semi-major axis of the ellipse.
- **Gamma[6]**: 3d metric co-efficients in $(t_C, \tau_0, \tau_2(3))$ coordinates.
- **space**: The enum describing the coordinate space in which the metric is computed.

2. **InspiralCoarseBankIn**: Input for choosing a template bank. This is the structure that must be filled by a routine calling the code `InspiralCreateCoarseBank` or `InspiralCreateBCVBank`. Unless BCV template bank is needed (that is, `InspiralCreateBCVBank`) then one can ignore the parameters psi0Min, psi0Max, psi3Min, psi3Max, alpha, numFcutTemplates.

```c
typedef struct
tagInspiralCoarseBankIn
{
    InspiralBankMassRange massRange;
    CoordinateSpace space;

    REAL8 mMin;
    REAL8 mMax;
    REAL8 MMax;
    REAL8 MMin;
    REAL8 alpha;
    REAL8 psi0Min;
    REAL8 psi0Max;
    REAL8 psi3Min;
    REAL8 psi3Max;
    REAL8 mmCoarse;
    REAL8 mmFine;
    REAL8 fLower;
    REAL8 fUpper;
    REAL8 tSampling;
    REAL8 etamin;
    REAL8 betaMin;
    REAL8 betaMax;
    REAL8 chiMin;
    REAL8 chiMax;
    REAL8 kappaMin;
    REAL8 kappaMax;
    REAL8FrequencySeries shf;
    /* Maximum size of the power spectral density array for use in
     * the computation of the metric in SBBH; typical values that
     * assures that the code runs quickly are 1024–8192.
     */
    UINT4 ShMaxSz;

    /* See for random number generation in RandomBank algorithm */
    UINT4 iseed;
};
```
/* nTIni is an estimate for the number of templates that might 
* be required; this is used in the random bank generation 
* routine with a seed number of templates = nTIni*sqrt(nTIni) */
UINT4 nTIni;
/* iflso is an integer that tells whether to compute the moments 
* using an upper limit defined by flso; this is not used anywhere 
* at the moment */
INT4 iflso;
/* spinBank=0:use Owen+Hanna bank*/
/* spinBank=1:use extended bank by AEI/Cardiff/Osaka */
/* spinBank=2:use random bank algorithm */
INT4 spinBank;
/* Number of templates required in the fCut (upper cutoff) 
* dimension and the value of upper and lower cutoffs */
UINT4 numFcutTemplates;
REAL4 HighGM;
REAL4 LowGM;

/* Type of gridspacing required: 
  1=SquareNotOriented, 
  2=Square, 
  3=HexagonalNotOriented, 
  4=Hexagonal, 
  5=HybridHexagonal, 
  6=S2BCV */
GridSpacing gridSpacing;
/* post-Newtonian order and approximation */
Order order;
Approximant approximant;
/* parameters for different/multiple freq. cutoffs */
INT4 NumFreqCut;
FreqCut MaxFreqCut;
FreqCut MinFreqCut;

InsidePolygon insidePolygon;
ComputeMoments computeMoments;
/* ComputeMoments tells whether to re-compute the moments 
* using an upper limit defined by flso; This is done after 
* the template bank is generated */

} InspiralCoarseBankIn;

- **massRange**: enum that determines whether templates should be chosen using fixed ranges for component masses or to use minimum component mass and maximum total mass.
- **space**: enum that decides whether to use ($\tau_0, \tau_2$) or ($\tau_0, \tau_3$) in constructing the template bank
- **alpha**: the BCV amplitude correction parameter
- **psi0Min**: minimum value of the parameter $\psi_0$
- **psi0Max**: maximum value of the parameter $\psi_0$
- **psi3Min**: minimum value of the parameter $\psi_3$
- **psi3Max**: maximum value of the parameter $\psi_3$
• \( m_{\text{Min}} \): minimum mass of components to search for
• \( m_{\text{Max}} \): maximum mass of components to search for
• \( M_{\text{Max}} \): alternatively, maximum total mass of binary to search for
• \( m_{\text{mmCoarse}} \): coarse grid minimal match
• \( m_{\text{mmFine}} \): fine grid minimal match
• \( f_{\text{Lower}} \): lower frequency cutoff
• \( f_{\text{Upper}} \): upper frequency cutoff
• \( t_{\text{Sampling}} \): sampling rate
• \( \eta_{\text{min}} \): minimum value of \( \eta \) in our search
• \( \text{shf} \): frequency series containing the PSD
• \( \text{iflso} \): (currently not implemented) \( \text{flso} \) will be used as an
• \( \text{numFcutTemplates} \): number of templates in the \( f_{\text{cut}} \) direction

The next two members are used in setting up the \text{InspiralTemplate} parameter structure but not in creating the template bank.

• \( \text{order} \): post-Newtonian order of the waveform
• \( \text{approximant} \): approximant of the waveform
• \( \text{NumFreqCut} \): number of different upper frequency cutoffs (spaced evenly between \( \text{MinFreqCut} \) and \( \text{MaxFreqCut} \)) to use when creating a template bank.
• \( \text{MaxFreqCut} \): largest upper frequency cutoff to use
• \( \text{MinFreqCut} \): smallest upper frequency cutoff to use

3. \text{InspiralFineBankIn} Structure needed by the function \text{LALInspiralCreateFineBank}. which computes a finer mesh around a given lattice point using the value of the fine-mesh minimal match, coarse-mesh minimal match and the metric at the current lattice point.

```c
typedef struct
tagInspiralFineBankIn
{
    InspiralTemplateList templateList;
    InspiralCoarseBankIn coarseIn;
} InspiralFineBankIn;
```

• \( \text{templateList} \): a list containing all the fine-mesh templates
• \( \text{coarseIn} \): input structure that contains useful necessary parameters to construct a fine-mesh.

4. \text{InspiralTemplateList} A grid of inspiral templates (i.e., a template list).

```c
typedef struct
tagInspiralTemplateList
{
    INT4 ID;
    InspiralTemplate params;
    InspiralMetric metric;
    UINT4 nLayer;
    struct tagInspiralTemplateList *next;
} InspiralTemplateList;
```

Structure returned by the coarse and fine bank generation routines. Currently we generate an array of type \text{InspiralTemplateList} which contains the coordinate markers (the parameter structure \text{InspiralTemplate} defined in the \text{inspiral} package) and the metric at each of those points. There is a desire to make this a truly linked list at some time in the future. The member of this structure are:
22.2. Header LALInspiralBank.h

• **ID**: An unique integer ID of the template
• **params**: Value of the parameters at the lattice point
• **metric**: metric at the lattice point
• **next**: pointer to next lattice point; but this is currently not filled by the bank code.

5. **InspiralBankParams**: This is a structure needed in the inner workings of the LALInspiralCreateCoarseBank code.

```c
typedef struct tagInspiralBankParams {
    INT4 nparams;
    REAL8 minimalMatch;
    REAL8 x0;
    REAL8 x1;
    REAL8 dx0;
    REAL8 dx1;
    REAL8 x0Min;
    REAL8 x0Max;
    REAL8 x1Min;
    REAL8 x1Max;
    InspiralMetric *metric;
} InspiralBankParams;
```

- **nparams**: Number of parameters (currently fixed at 2, so this is as of now unused)
- **x0**: the first coordinate, chosen to be always $\tau_0$
- **x1**: the second coordinate, chosen to be either $\tau_2$ or $\tau_3$
- **dx0**: increment in the x0-direction
- **dx1**: increment in the x1-direction
- **x0Min**: minimum value of the first coordinate as defined by the search region
- **x0Max**: maximum value of the first coordinate as defined by the search region
- **x1Min**: minimum value of the second coordinate as defined by the search region
- **x1Max**: maximum value of the second coordinate as defined by the search region
- **metric**: pointer to the metric at the current location.

6. **InspiralMomentsIn** Inputs to the function that computes the moments of the PSD. The moment is defined as:

$$I(p) \equiv \int_{x_{\text{min}}}^{x_{\text{max}}} \frac{x^{-p}}{S_h(x)} \, dx,$$

where $x = f/f_0$ is a scaled frequency, $f_0$ being a fiducial frequency, taken in these routines as the user supplied lower cutoff of the detector response.

```c
typedef struct {
    REAL8 xmin;
    REAL8 xmax;
    REAL8 ndx;
    REAL8 norm;
    REAL8FrequencySeries *shf;
} InspiralMomentsIn;
```

- **xmin**: lower limit of the integral $x_{\text{min}}$
• $\text{xmax}$: upper limit of the integral $x_{\text{max}}$
• $\text{ndx}$: index $p$ (without the negative sign) in the moment integral as above
• $\text{norm}$: norm to be used in computing the moment, the returned value is the above integral divided by the norm.
• $\text{#shf}$: the frequency series containing the noise psd.

7. **InspiralMomentsEtc** Parameter structure that holds the moments of the PSD and other useful constants required in the computation of the metric.

```c
typedef struct
  tagInspiralMomentsEtc
{
  REAL8 a01, a21, a22, a31, a41, a42, a43;
  REAL8 j[18];
} InspiralMomentsEtc;
```

• $a01, a21, \ldots$: Coefficients in the expansion of the phase of the Fourier transform of an inspiral waveform computed in the stationary phase approximation. See documentation under the function LALInspiralComputeMetric later in this Section for a description of these coefficients.

• $j[18]$: The required moments are all computed once and stored in this array. The required moments are from $J(1)$ to $J(17)$ (except $J(2), J(3)$ and $J(16)$ that are not required at 2PN order, however, they are computed since future extensions, planned in the near future, will require them). However, in C we need an array size 18 to use an array that has an index 18. To ease the notation we have therefore defined an over sized (by one element) array.

8. **RectangleIn** and **RectangleOut**: Input and output structures to function LALRectangleVertices.

```c
typedef struct
  tagRectangleIn
{
  REAL8 x0, y0, dx, dy, theta;
} RectangleIn;
```

```c
typedef struct
  tagRectangleOut
{
  REAL8 x1;
  REAL8 y1;
  REAL8 x2;
  REAL8 y2;
  REAL8 x3;
  REAL8 y3;
  REAL8 x4;
  REAL8 y4;
  REAL8 x5;
  REAL8 y5;
} RectangleOut;
```

9. **HexaGridParam** This is a structure needed in the inner workings of the LALInspiralHexagonalBank code.

```c
typedef struct
  tagHexaGridParam
{
```
REAL4 x0Min;
REAL4 x1Min;
REAL4 x0Max;
REAL4 x1Max;
REAL4 mm;
REAL4 mMin;
REAL4 mMax;
REAL4 etaMin;
REAL4 MMin;
REAL4 MMax;
REAL4 fLower;
GridSpacing gridSpacing;
InspiralBankMassRange massRange;
CoordinateSpace space;
}
HexaGridParam;

It contains some part of CoarseBankIn and some other standard parameters. It provides the parameter
space boundaries with the minimum and maximum values of mass parameters, the minimal match, the
space, massRange and gridSpacing parameter.

10. **CellEvolution** This is a structure needed in the inner workings of the LALInspiralHexagonalBank
code.

```c
typedef struct
tagCellEvolution
{
    INT4 nTemplateMax;
    INT4 nTemplate;
    INT4 fertile;
}
CellEvolution;
```

This structure checks the status of the placement. **fertile** tells if the placement is still evolving or not.
**nTemplateMax** is the number of maximum templates allowed, which can be resized. And **nTemplate**
is the number of template set. nTemplate can not be higher than nTemplateMax.

11. **CellList** This is a structure needed in the inner workings of the LALInspiralHexagonalBank
code.

```c
typedef struct
tagCellList
{
    INT4 id;
    struct tagCellList *next;
}
CellList;
```

Similarly to the square placement, which uses InspiralList, we used a linked list for the hexagonal place-
ment. A different structure has been implemented so as to simplify the complexity of the algorithm.
It also set an id to each cell which has been created. This id is unique to each cell/template.

12. **InspiralCell** This is a structure needed in the inner workings of the LALInspiralHexagonalBank
code.

```c
typedef struct
tagInspiralCell
{
    INT4 ID;
    INT4 in;
    INT4 child[6];
```
REAL4 t0;
REAL4 t3;
REAL4 dx0;
REAL4 dx1;
Generation status;
Position position;
Position RectPosition[5];
InspiralMetric metric;
}
InspiralCell;

Each cell is defined by this structure, which contains the position of each cell in the \(\tau_0/\tau_3\) parameter space, the metric at that point, and various information such as the status of the cell. Is it still fertile? what is its position with respect to the parameter space and so on. child is a 6-length array with a link to the 6 templates (hexagonal) around the current template that we are dealing with.
22.2.1 Module LALInspiralCreateCoarseBank.c

Prototypes

```c
void LALInspiralCreateCoarseBank(
    LALStatus *status,
    InspiralTemplateList **list,
    INT4 *nlist,
    InspiralCoarseBankIn coarseIn
)
```

- `list`, Output, an array containing the template bank parameters
- `nlist`, Output, the number of templates found by the function
- `coarseIn`, Input, specifies the search space, range of masses, etc.

The coarse grid algorithm works in two stages: After computing the minimum and maximum chirp-times corresponding to the search space: \((\tau_{0}^{\text{min}}, \tau_{0}^{\text{max}}), (\tau_{2}^{\text{min}}, \tau_{2}^{\text{max}})\) (or \((\tau_{3}^{\text{min}}, \tau_{3}^{\text{max}})\)) the algorithm

1. chooses a lattice of templates along the equal mass curve and then
2. lays a rectangular grid in the rectangular region defined by the minimum and maximum values of the chirp-times and retain only if (a) the point lies in the parameter space, OR (b) one of the vertices defined by the rectangle lies in the parameter space.

Description

**Templates along the equal mass curve**  The algorithm works in two stages: In the first stage, templates are built along the equal mass (that is, \(\eta = 1/4\)) curve starting from the minimum value of the Newtonian chirp-time and stopping at its maximum value. Given the \(n\) th template at \(O\) with parameters \((\tau_{0}^{(n)}, \tau_{3}^{(n)})\), and given also the distance between templates in our preferred coordinates \((D\tau_{0}^{(n)}, D\tau_{3}^{(n)})\), consider lines \(\tau_{0} = \tau_{0}^{(n)} + D\tau_{0}^{(n)}\) (QA of Fig. 22.2) and \(\tau_{3} = \tau_{3}^{(n)} + D\tau_{3}^{(n)}\) (PB of Fig. 22.2). The template next to

![Algorithm sketching the placement of templates along \(\eta = 1/4\) curve.](image)

\(^3\)In what follows we will only mention \(\tau_{3}\); however, the algorithm is itself valid, and has been implemented, in the case of \((\tau_{0}, \tau_{2})\) too. However, we recommend that the space \(\tau_{0}-\tau_{3}\) be used.
on the equal mass curve, must lie either along PB or along QA (cf. Fig. 22.2) in order that all the signals that may lie on OAB are spanned by at least one of the two templates. Clearly, if we were to place the \((n+1)\)th template at \(B\), some of the signals won’t have the required minimal match. However, placing the \((n+1)\)th template at \(A\) suffices our requirement. (Note, however, that there is no guarantee that this will always work; it works only if the curve along which templates are being laid is a slowly varying function.) To locate the \((n+1)\)th template we compute the following pairs of coordinates:

\[
\tau_0^{(n+1)} = \tau_0^{(n)} + D\tau_0^{(n)}, \quad \tau_3^{(n+1)} = 4A_3 \left( \frac{\tau_0^{(n+1)}}{4A_0} \right)^{2/5},
\]

\[
\tau_3^{(n+1)} = \tau_3^{(n)} + D\tau_3^{(n)}, \quad \tau_0^{(n+1)} = 4A_0 \left( \frac{\tau_3^{(n+1)}}{4A_3} \right)^{5/2},
\]

where

\[
A_0 = \frac{5}{256(\pi f_0)^{8/3}}, \quad A_3 = \frac{\pi}{8(\pi f_0)^{5/3}}.
\]

Of the two pairs, the required pair is the one that is closer to the starting point \((\tau_0^{(n)}, \tau_3^{(n)})\).

Templates in the rest of the parameter space In the second stage, the algorithm begins again at the point \((\tau_0^{\text{min}}, \tau_3^{\text{min}})\), corresponding distance between templates \((D\tau_0^{\text{min}}, D\tau_3^{\text{min}})\), and chooses a rectangular lattice of templates in the rectangular region defined by \((\tau_0^{\text{min}}, \tau_3^{\text{min}})\) \((\tau_0^{\text{max}}, \tau_3^{\text{max}})\) \((\tau_0^{\text{max}}, \tau_3^{\text{max}})\). The implementation of the algorithm along the equal mass curve and in a rectangular lattice in the rest of the parameter space is shown plotted in Fig. 22.3 where the templates chosen are represented as points.

![Figure 22.3: Algorithm sketching the construction of a rectangular lattice of templates.](image)

Algorithm
The algorithm to lay templates along the equal-mass curve is as follows:

**Begin at** \(\tau_0 = \tau_0^{\text{min}}\)

**do while** \((\tau_0 < \tau_0^{\text{max}})\)

\{

\[
\begin{align*}
\tau_0^A &= \tau_0 + D\tau_0, \quad \tau_3^A = 4A_3 \left( \frac{\tau_0^A}{4A_0} \right)^{2/5}, \\
\tau_3^B &= \tau_3 + D\tau_3, \quad \tau_0^B = 4A_0 \left( \frac{\tau_3^B}{4A_3} \right)^{5/2},
\end{align*}
\]

\}
if \((\tau_0^A, \tau_3^A)\) is closer \((\tau_0, \tau_3)\) than \((\tau_0^B, \tau_3^B)\)
{
    \tau_0 = \tau_0^A, \tau_3 = \tau_3^A
}
else
{
    \tau_0 = \tau_0^B, \tau_3 = \tau_3^B
}
Add \((\tau_0, \tau_3)\) to InspiralTemplateList
numTemplates++
Compute metric at \((\tau_0, \tau_3)\)
Compute distance between templates at new point: \((D\tau_0, D\tau_3)\)

The algorithm to lay templates in the rest of the parameter space is as follows:
Begin at \(\tau_0 = \tau_0^{min}, \tau_3 = \tau_3^{min}\)
Compute metric at \((\tau_0, \tau_3)\)
Compute distance between templates at new point: \((D\tau_0, D\tau_3)\)
Add \((\tau_0, \tau_3)\) to InspiralTemplateList
numTemplates++
do while \((\tau_3 <= \tau_3^{max})\)
{
    do while \((\tau_0 <= \tau_0^{max})\)
    {
        if \(((\tau_0, \tau_3)\) is inside the parameter space)
        {
            Compute metric at \((\tau_0, \tau_3)\)
            Compute distance between templates at new point: \((D\tau_0, D\tau_3)\)
            Add \((\tau_0, \tau_3)\) to InspiralTemplateList
            numTemplates++
        }
        Increment \(\tau_0\): \(\tau_0 = \tau_0 + D\tau_0\)
    }
    Increment \(\tau_3\): \(\tau_3 = \tau_3 + D\tau_3\)
    Get next template along \(\tau_3 = \text{const.} : (\tau_0, \tau_3)\)
}

Uses
LALInspiralNextTemplate()
LALInspiralParameterCalc()
LALInspiralSetParams()
LALInspiralSetSearchLimits()
LALInspiralComputeParams()
LALInspiralComputeMetric()
LALInspiralUpdateParams()
LALInspiralValidTemplate()

Notes
void LALInspiralCreateFlatBank (
    LALStatus *status,
    REAL4VectorSequence *list,
    InspiralBankParams *bankParams
)

- *list*, Output, an array containing the template bank parameters
- *bankParams*, Input. It is necessary and sufficient to input the eigenvalues of the metric and the angle between the $x_0$ axis and the semi-major axis of the ambiguity ellipse, that is, bankParams.metric.g00, bankParams.metric.g11, bankParams.metric.theta, the minimal match, bankParams.minimalMatch and the range of the two coordinates over which templates must be chosen: (bankParams->x0Min, bankParams->x0Max) and (bankParams->x1Min, bankParams->x1Max).

The code expects list->vectorLength=2 and allocates just the requisite amount of memory to list and returns the number of grid points in list->length. The data points list->data[2j], j=1,2,..., list->length, contain the $x_0$-coordinates of the grid and data points list->data[2j+1], contain the $x_1$-coordinates of the grid.

Description

Given the metric and the minimalMatch this routine calls bank/LALInspiralUpdateParams to get the spacings in user coordinates (which are not necessarily the eigen-directions) and lays a uniform grid of templates in the range specified in (bankParams->x0Min, bankParams->x0Max) and (bankParams->x1Min, bankParams->x1Max).

Algorithm

The algorithm to lay templates is as follows: Given the increments $Dx_0$ and $Dx_1$ found from calling bank/LALInspiralUpdateParams lay a rectangular grid in the space of $(x_0,x_1)$.

\[
x_1 = x_1^{\text{min}}
\]
\[
do\ while\ (x_1 \leq x_1^{\text{max}})
\{
    x_0 = x_0^{\text{min}}
    do\ while\ (x_0 \leq x_0^{\text{max}})
    \{
        \text{Add}\ (x_0,x_1)\ \text{to}\ \text{list}\n        \text{numTemplates++}
        \text{Increment}\ x_0: \ x_0 = x_0 + Dx_0
    \}
    \text{Increment}\ x_1: \ x_1 = x_1 + Dx_1
\}

Uses

LALInspiralUpdateParams()
LALRalloc()

Notes

Author: Churches, D. K and Sathyaprakash, B.S.
$Id: LALInspiralCreateCoarseBank.c,v 1.73 2007/12/18 17:01:01 devanka Exp $
22.2.2 Module LALInspiralCreateBCVBank.c

Lay a flat grid of BCV templates in the user specified range of the parameters ($\psi_0, \psi_3$) in coarseIn structure (see below).

Prototypes

```c
void LALInspiralCreateBCVBank (  
    LALStatus *status,  
    InspiralTemplateList **list,  
    INT4 *nlist,  
    InspiralCoarseBankIn coarseIn
)
```

- `list`, Output, an array containing the template bank parameters.
- `nlist`, Output, the number of templates in bank.

Description

Given the range of the parameters ($\psi_0, \psi_3$), the number of templates in the fCut direction, minimalMatch, noise spectral density, upper and lower frequency cutoffs (all in the input structure coarseIn) this routine outputs the list of templates in the BCV bank for the parameters ($\psi_0, \psi_3, f_{cut}$).

Algorithm

A flat signal manifold is assumed and templates are laid uniform in the three dimensions. See below for an explanation of how templates are chosen in the fcut direction.

Uses

LALInspiralUpdateParams()
LALRalloc()

Notes
22.2.3 Module LALInspiralBCVFcutBank.c

Given a grid of templates with distinct values of \((\psi_0, \psi_3)\) this routine returns a new grid in which every template has `numFcutTemplates` partners differing from one another in the ending frequency `f_{\text{end}BCV}`. A call to this function should be preceded by a call to LALInspiralCreateFlatBank.c, or a similar function, that gives a grid in \((\psi_0, \psi_3)\) space.

Prototypes

```c
void LALInspiralBCVFcutBank (  
    LALStatus *status,  
    InspiralTemplateList **list,  
    INT4 *NList,  
    InspiralCoarseBankIn coarseIn
)
```

```c
void LALInspiralBCVFcutS3S4 (  
    LALStatus *status,  
    InspiralTemplateList **list,  
    INT4 *NList,  
    InspiralCoarseBankIn coarseIn
)
```

- `list`, Output/Input, an array initially containing the template bank with the values of `list[j]->psi0`, `list[j]->psi3`, `list[j]->fLower`, specified, is replaced on return with a re-sized array specifying also `list->fFinal`.
- `NList`, Output/Input, the number of templates in the Input bank is replaced by the number of templates in the output bank.
- `numFcutTemplates`, Input, the largest number of templates for the parameter `f_{cut}` of BCV.

Description

A lattice of templates for BCV models should include, in addition to the values of \((\psi_0, \psi_3)\) a range of \(f_{\text{cut}}\) – the cutoff frequency. The right approach would be to compute the metric in the three-dimensional space of \((\psi_0, \psi_3, f_{\text{cut}})\) and to choose templates as dictated by the metric. However, analytic computation of the metric has not been easy. Therefore, it has become necessary (at least for the time being) to make alternate choice of the cutoff frequencies.

In this routine we implement a simple choice based on physical grounds: The post-Newtonian models predict an ending frequency that is larger than, but close to, the Schwarzschild last-stable orbit frequency \(f_{\text{iso}} = (63/2\pi M)^{-1}\) where \(M\) is the total mass, while the effective one-body model has an ending frequency close to the light-ring, whose Schwarzschild value is \(f_{\text{lr}} = (33/2\pi M)^{-1}\). It is necessary to know the total mass of the system in both cases. However, not all pairs of \((\psi_0, \psi_3)\) can be inverted to get a positive \(M\) but only when \(\psi_0 > 0\) and \(\psi_3 < 0\). Even then it is not guaranteed that the symmetric mass ratio will be less than \(1/4\), a necessary condition so that the component masses are found to be real. However, we do not demand that the symmetric mass ratio is less than a quarter. If the total mass is non-negative then we compute the \((f_{\text{iso}}, f_{\text{lr}})\) and choose a user specified `numFcutTemplates` number of templates with their cutoff frequency `list->fFinal` defined uniformly spaced in the range \([f_{\text{iso}}, f_{\text{lr}}]\).

Furthermore, this routine discards all templates for which either the mass is not defined or, when defined, `list->fFinal` is smaller than the user defined lower frequency cutoff or larger than the Nyquist frequency of templates. Thus, the number of templates returned by this routine could be larger or fewer than the input number of templates.
Algorithm

Given \( (\psi_0, \psi_3) \) one can solve for \( (M, \eta) \) using:

\[
M = -\frac{\psi_3}{16\pi^2\psi_0}, \quad \eta = \frac{3}{128\psi_0(\pi M)^{3/3}}.
\] (22.8)

Given the total mass compute the last stable orbit and light-ring frequencies using

\[
f_{\text{iso}} = (6^{3/2}\pi M)^{-1}, \quad f_{\text{lr}} = (3^{3/2}\pi M)^{-1}.
\] (22.9)

Divide the range \( (f_{\text{iso}}, f_{\text{lr}}) \) so as to have \( n_{\text{cut}} = \text{numFcutTemplates} \) templates over this range:

\[
df = f_{\text{lr}} \left(1 - 2^{-3/2}\right) \frac{1}{(n_{\text{cut}} - 1)}.
\] (22.10)

Next, choose templates at \( f_k = f_{\text{lr}} - k \times df \), where \( k = 0, \ldots, n_{\text{cut}} - 1 \). Note that by definition \( f_0 = f_{\text{lr}} \) and \( f_{n_{\text{cut}}-1} = f_{\text{iso}} \); there are exactly \( n_{\text{cut}} \) templates in the range \( (f_{\text{iso}}, f_{\text{lr}}) \). We discard a template if either \( M \) is not defined or if \( f_{\text{cut}} \) is smaller than the lower frequency cutoff specified in \text{list[j]}->fLower.

Uses

LALRalloc()

Notes
### 22.2.4 Module `LALInspiralBankUtils.c`

#### Prototypes

```c
REAL4 XLALInspiralTau3FromTau0AndEqualMassLine(REAL4 tau0, REAL4 fL);
REAL4 XLALInspiralTau3FromNonEqualMass(REAL4 M, REAL4 eta, REAL4 fL);
REAL4 XLALInspiralTau0FromMEta(REAL4 M, REAL4 eta, REAL4 fL);
REAL8 XLALInspiralMFromTau0AndNonEqualMass(REAL8 tau0, REAL8 extremMass, REAL8 fL);
```

#### Description

In a parameter space defined by $m_1$ and $m_2$, or equivalently, $M = m_1 + m_2$ and $\eta = \frac{m_1 m_2}{M^2}$, the conversion to chirp-time parameter such as $\tau_0$ and $\tau_3$ is quite common. In particular, it is interesting to get the value of $\tau_3$ when only $\tau_0$ is known, and a constraint on the masses exists (e.g., $m_1 = m_2$ or one of the mass equals mMin or mMax. This modules contains a few functions to perform these conversion.

#### Algorithm

We know that

$$\tau_0 = \frac{A_0}{\eta} M^{-5/2},$$

and

$$\tau_3 = \frac{A_3}{\eta} M^{-2/3},$$

where

$$A_0 = \frac{5}{256(\pi \ast f_L)^{8/3}},$$

and

$$A_3 = \frac{\pi}{8(\pi \ast f_L)^{5/3}}.$$ 

Therefore, it is straightforward to express $\tau_3$ as a function of $\tau_0$ and $\eta$:

$$\tau_3 = \frac{A_3}{\eta} \left( \frac{\tau_0 \eta}{A_0} \right)^{2/5}$$

if $\eta = 0.25$ on the equal-mass line, then

$$\tau_3 = 4A_3 \left( \frac{\tau_0}{4A_0} \right)^{2/5}$$

(22.15)
Equation 22.15 returns $\tau_3$ given in $M, \eta$ and $f_L$ and is defined in XLALInspiralTau3FromNonEqualMassLine.

Equation 22.16 returns $\tau_3$ in the particular case $m_1 = m_2$, given $\tau_0$ only, and is defined in XLALInspiralTau3FromTau0AndEqualMassLine.

Equation 22.11 returns $\tau_0$ given $M, \eta$ and $f_L$, and is defined XLALInspiralTau0FromMEta.

Finally, XLALInspiralMFromTau0AndNonEqualMass returns $M$ when $\tau_0$ is known and a constraint exists on one of the individual mass (e.g., $m_1 = m_{\text{Max}}$ or $m_1 = m_{\text{Min}}$). This functions requires a little more algebra and is used in the HybridHexagonal placement. The function LALInspiralHybridHexagonal describes this algebra.
22.2.5 Module InspiralSpinBank.c

This module creates a bank of templates to search for precessing binaries.

Prototypes

```c
void LALInspiralSpinBank(
    LALStatus *status,
    SnglInspiralTable **tiles,
    INT4 *ntiles,
    InspiralCoarseBankIn *coarseIn
)
```

Description

This function creates a bank of BCVSpin templates to search for precessing binaries.

Algorithm

The code checks `coarseIn->mMin` to determine whether the limits on the target region are in terms of masses or phenomenological parameters. A positive value indicates that mass limits are being used.

If mass limits are used, the target region of parameter space is a distorted box in the coordinates \((x = \psi_0, y = \psi_3, z = \beta)\). The metric at high values of \(\beta\) is constant. It is convenient to rotate to coordinates \((x', y', z')\) which lie along eigenvectors of the metric.

The algorithm first draws a rectilinear box in the primed coordinates which includes the target region, then steps through along the directions of the primed coordinates. At each point it tests if the point lies within the target region. If the point is inside the target region, the algorithm adds a template to the linked list. If not, it continues through the box containing the target region.

The tiling is done with a body-centered cubic lattice. People usually solve the non-overlapping bcc problem rather than the overlapping one here, so it’s worth mentioning how we do it. I don’t have time to stick in the 3D figures you need to show it properly, but you can figure out the spacing by finding the smallest sphere that contains the Wigner-Seitz cell. When you do that you find that the lattice constant (spacing between templates in the plane, in proper distance) is \((4/3) \sqrt[3]{2}\mu\). So the coordinate spacing is that divided by the square root of the corresponding eigenvalue of the metric. (The vertical spacing in the bcc lattice is multiplied by a further 1/2.)

If \((\psi_0, \psi_3, \beta)\) limits are used, the tiling is done in the given box with a bcc lattice.

Uses

- `LALCalloc()`
- `LALFree()`
- `LALGetInspiralMoments()`
- `LALSSymmetricEigenVectors()`

Notes

Currently we use a static function for the metric based on an approximation that is good only for large \(\beta\). We should update it and put it out in the LAL namespace.

The metric relies on approximations that make it valid only for a binary system with a total mass \(< 15M_\odot\) where the larger body’s minimum mass is at least twice the smaller body’s maximum mass. If the parameter range is specified with physical parameters rather than the phenomenological parameters \((\psi_0, \psi_3, \beta)\) then using mass values that violate these conditions will result in an error message.

* Authors: Hanna, C. R. and Owen, B. J.

* $Id: InspiralSpinBank.c,v 1.26 2007/06/08 14:41:42 bema Exp $
22.2.6 Module LALInspiralCreateFineBank.c

Function to create a fine grid of templates.

Prototypes

```c
void LALInspiralCreateFineBank(LALStatus *status,
    InspiralTemplateList **outlist,
    INT4 *nlist,
    InspiralFineBankIn fineIn)
```

- `outlist`, Output, containing an array of template bank parameters
- `nlist`, Output, the number of fine bank templates around a given coarse-mesh point
- `fineIn`, Input, the parameters required to find the fine bank

Description

The fine grid algorithm is a very simple algorithm that computes a uniform grid of templates around a given coordinate point – which can in particular be a coarse grid point – from a knowledge of the metric at the coordinate point and the coarse and fine grid minimal matches, \(D_{\tau_0,3}\) and \(d_{\tau_0,3}\), respectively. Since \(D\tau\) is not necessarily an integral multiple of \(d\tau\) the rectangular fine grid about the point in question will be larger than required. The algorithm chooses templates symmetrically about the given coarse grid point. It does so by laying a rectangular lattice of templates with spacings \(d_{\tau_0}\) and \(d_{\tau_3}\), in the rectangular region defined by

\[
\begin{align*}
(\tau_0 - \Delta\tau_0, \tau_3 - \Delta\tau_3), \\
(\tau_0 + \Delta\tau_0, \tau_3 - \Delta\tau_3), \\
(\tau_0 + \Delta\tau_0, \tau_3 + \Delta\tau_3), \\
(\tau_0 - \Delta\tau_0, \tau_3 + \Delta\tau_3),
\end{align*}
\]

where

\[
\Delta\tau_0 = d\tau_0 \left[ \frac{D\tau_0}{2d\tau_0} \right],
\]

and for any \(x\), \([x]\) denotes the smallest integer greater than or equal to \(x\). The algorithm takes as input a structure of type `InspiralFineBankIn` and returns a pointer-to-a-pointer of type `InspiralTemplateList` as well as the number of fine grid templates `int` around the lattice point in question.

Figure 22.4: Algorithm sketching the construction of a rectangular fine grid around a given coordinate point.
The spacing between fine grid templates is chosen to be a constant determined by the metric at the coarse grid point; for example,

\[ d\tau_0 = \sqrt{\frac{2(1 - MM_{\text{Fine}})}{g_{00}}}. \]

Only those grid points that are within the parameter space boundary, or have the vertices of the ambiguity rectangle inside the parameter space, are kept and others are discarded.

Algorithm

The Fine grid algorithm works as follows:

- From input structure extract coordinates of the grid point \((\tau_0^G, \tau_3^G)\).
- Compute coarse and fine grid spacings \((D\tau_0, D\tau_3)\) and \((d\tau_0, d\tau_3)\).
- Compute half-sides of the smallest symmetric rectangle about \((\tau_0, \tau_3)\):
  \[ \Delta\tau_0 = d\tau_0 \text{ceil}[D\tau_0/(2d\tau_0)], \quad \Delta\tau_3 = d\tau_3 \text{ceil}[D\tau_3/(2d\tau_3)]. \]
- Begin at \(\tau_3 = \tau_3^G - \Delta\tau_3,\)
  do while \((\tau_3 <= \tau_3^G + \Delta\tau_3)\)
  \{ 
    Begin at \(\tau_0 = \tau_0^G - \Delta\tau_0,\)
    do while \((\tau_0 <= \tau_0^G + \Delta\tau_0)\)
    \{ 
      if \(((\tau_0, \tau_3)\text{ is inside the parameter space})
      \} 
      Add \((\tau_0, \tau_3)\) to InspiralTemplateList
      numTemplates++
    \}
    Increment \(\tau_0 : \tau_0 = \tau_0 + d\tau_0\)
  \}
  Increment \(\tau_3 : \tau_3 = \tau_3 + d\tau_3\)

Uses

LALInspiralComputeParams()
LALInspiralUpdateParams()
LALInspiralValidTemplate()

Notes

Author: Sathyaprakash, B.S. and Churches, D. K.

$Id: LALInspiralCreateFineBank.c,v 1.16 2007/06/08 14:41:42 bema Exp $
22.2.7 Module `LALInspiralComputeMetric.c`

Module to compute the components of the metric which is used to describe distances on the signal manifold.

Prototypes

```c
void
LALInspiralComputeMetric (  
    LALStatus *status,  
    InspiralMetric *metric,  
    InspiralTemplate *params,  
    InspiralMomentsEtc *moments  
);
```

```c
void
LALInspiralComputeMetricBCV (  
    LALStatus *status,  
    InspiralMetric *metric,  
    REAL8FrequencySeries *psd,  
    InspiralTemplate *params  
);
```

- **metric**, Output, the metric at the lattice point defined by **params**
- **params**, Input, the parameters where metric must be computed
- **moments**, Input, moments \(J(1), \ldots, J(17)\), of the PSD and other constants needed in the computation of the metric.

**Description**

We calculate the components of the metric using the procedure outlined in Owen [5]. This uses the moments of the noise curve,

\[
I(q) \equiv S_h(f_0) \int_{f_c/f_0}^{f_s/f_0} \frac{x^{-q/3}}{S_h(x)} \, dx
\]  
(22.17)

and

\[
J(q) \equiv \frac{I(q)}{I(7)}.
\]  
(22.18)

(Please note that the function `LALInspiralMoments` doesn't compute \(I(q)\) defined here; the index \(q\) is defined differently there and the normalisation is supplied by the user. For ease of writing, here we shall follow the standard notation.) Then the moment functional \(\mathcal{J}\) is defined such that, for a function \(a\),

\[
\mathcal{J}[a] \equiv \frac{1}{I(7)} \int_{f_c/f_0}^{f_s/f_0} \frac{x^{-7/3}}{S_h(x)} a(x) \, dx
\]  
(22.19)

which gives us

\[
\mathcal{J} = \left[ \sum_n a_n x^n \right] = \sum_n a_n J(7 - 3n).
\]  
(22.20)

The above equation is used to calculate the components of the metric using the following formula:

\[
\gamma_{\alpha\beta} = \frac{1}{2} (\mathcal{J}[\psi_{\alpha}\psi_{\beta}] - \mathcal{J}[\psi_{\alpha}]\mathcal{J}[\psi_{\beta}])
\]  
(22.21)

where \(\psi_{\alpha}\) is the derivative of the Fourier phase of the inspiral waveform with respect to the parameter \(\lambda^\alpha\), that is \(\psi_{\alpha} \equiv \Psi,_{\alpha}\). Writing the derivative index as \(\alpha = 0, j\), with \(j = 1, \ldots, n\) we have

\[
\psi_0 \equiv 2\pi f, \quad \psi_j \equiv \frac{\partial \Delta \Psi}{\partial \Delta \lambda^j}.
\]  
(22.22)
The phase $\Psi$ is that which appears in the usual stationary-phase formula for the Fourier transform:
\[ \tilde{h}(f) \propto f^{-7/6} e^{[-\pi/4-\Phi_0+2\pi f t_0+\Psi(f,\vec{\lambda})]} . \]  
(22.23)

If we take the usual chirp times and multiply each by $(\pi f_0)$ then we get dimensionless chirp times $\tau_k$, and then the phase $\Psi$ may be written in the form
\[ \Psi = 2\pi f t_c + \sum_k \Psi_k(f) \tau_k \]  
(22.24)

where, defining $v_0 = (\pi m f_0)^{1/3}$ ($m$ being total mass and $f_0$ a fiducial starting frequency), the chirptimes $\tau_k$, up to 2nd PN order, are given by
\[ \tau_0 = \frac{5}{256\nu v_0^2}, \tau_2 = \frac{5}{192\nu v_0^2} \left( \frac{743}{336} + \frac{11}{4} \eta \right) , \]
\[ \tau_3 = \frac{\pi}{8\nu v_0^2}, \tau_4 = \frac{5}{128\nu v_0^2} \left( \frac{3,058,673}{1,016,064} + \frac{542}{1008} \eta + \frac{617}{144} \eta^2 \right) . \]  
(22.25)

Up to second post-Newtonian approximation the $\Psi_k$ are given by
\[ \Psi_0 = \frac{6}{5 \nu^{3/2}}, \Psi_2 = \frac{2}{\nu}, \Psi_3 = -\frac{3}{\nu^{2/3}}, \Psi_4 = \frac{6}{\nu^{4/3}} . \]  
(22.26)

where $\nu = f/f_0$.

If we now make the substitution $f = v^3/\pi m$ then we find that the phase may be expressed in a simpler form
\[ \Psi(v) = 2\pi f t_c + \sum_k \theta_k v^{k-5} \]  
(22.27)

where the chirp parameters $\theta_k$ are given by
\[ \theta_0 = \frac{3}{128\eta}, \theta_2 = \frac{5}{96\eta} \left( \frac{743}{336} + \frac{11}{4} \eta \right) , \]
\[ \theta_3 = -\frac{3\pi}{8\eta}, \theta_4 = \frac{15}{64\eta} \left( \frac{3,058,673}{1,016,064} + \frac{542}{1008} \eta + \frac{617}{144} \eta^2 \right) . \]  
(22.28)

If we want to express $\Psi$ in terms of $f$ rather than $v$ we simply substitute $v = (\pi m f)^{1/3}$ to obtain
\[ \Psi(f) = 2\pi f t_c + \sum_k \theta'_k f^{(k-5)/3} \]  
(22.29)

where
\[ \theta'_k = (\pi m)^{(k-5)/3} \theta_k . \]  
(22.30)

We are now in a position to start calculating components of $\gamma_{\alpha\beta}$. We had
\[ \psi_j = \frac{\partial \Delta \Psi}{\partial \Delta \lambda^j} \]  
(22.31)

where $\Psi$ is given by Eq. (22.29). Therefore we may write
\[ \Delta \Psi = \Delta \theta'_0 f^{-5/3} + \Delta \theta'_2 f^{-1} + \Delta \theta'_3 f^{-2/3} + \Delta \theta'_4 f^{-1/3} \]  
(22.32)

All we need to do now is specify the coordinates $\lambda^j$ with respect to which the derivatives will be taken. In general, the template placement algorithm works in $(\eta, \tau)$ coordinates. It is simplest for us to calculate the components of $\gamma_{\alpha\beta}$ in the $(m, \eta)$ coordinate system and then perform a coordinate transformation to get the components in the $(\tau_0, \tau_3)$ system. So, first of all we calculate the components of $\gamma_{\alpha\beta}$ in the $(m, \eta)$ system.

This involves calculating the following:
\[ \frac{\partial \Delta \Psi}{\partial \Delta m} = \frac{\Delta \theta'_0}{\Delta m} f^{-5/3} + \frac{\Delta \theta'_2}{\Delta m} f^{-1} - \frac{\Delta \theta'_3}{\Delta m} f^{-2/3} + \frac{\Delta \theta'_4}{\Delta m} f^{-1/3} \]  
(22.33)

and
\[ \frac{\partial \Delta \Psi}{\partial \Delta \eta} = \frac{\Delta \theta'_0}{\Delta \eta} f^{-5/3} + \frac{\Delta \theta'_2}{\Delta \eta} f^{-1} - \frac{\Delta \theta'_3}{\Delta \eta} f^{-2/3} + \frac{\Delta \theta'_4}{\Delta \eta} f^{-1/3} \]  
(22.34)
where all of the derivatives are easily calculable. This gives us the terms \( \psi_j \) as a power series in \( f \). These are then used in the formula

\[
\gamma_{\alpha\beta} = \frac{1}{2} (J[\psi_\alpha \psi_\beta] - J[\psi_\alpha]J[\psi_\beta])
\]  

(22.35)

to calculate the components of \( \gamma_{\alpha\beta} \). The fact that each of the \( \psi_j \) is in the form of a power series in \( f \) allows us to calculate \( \gamma_{\alpha\beta} \) using

\[
J \left[ \sum_n a_n x^n \right] = \sum_n J(7 - 3n).
\]  

(22.36)

i.e., we can express all the \( J[\cdot] \) in terms of the integral \( J(q) \) which we calculate numerically at the outset for the required values of \( q \).

Once we have obtained \( \gamma_{\alpha\beta} \) in this way, we take the inverse of this matrix to give us \( \gamma_{\alpha'\beta'} \) in the \((m, \eta)\) system. Then we perform the following coordinate transformation to give us the components of \( \gamma_{\alpha'\beta'} \) in our chosen system,

\[
\gamma_{\alpha'\beta'} = \Delta_{\alpha}^{\alpha'} \Delta_{\beta}^{\beta'} \gamma_{\sigma\delta}
\]  

(22.37)

where the transformation matrix \( \Delta_{\alpha}^{\alpha'} \) is defined by

\[
\Delta_{\alpha}^{\alpha'} = \frac{\partial x^{\alpha'}}{\partial x^\alpha}.
\]  

(22.38)

Finally, we take the inverse of this matrix to obtain \( \gamma_{\alpha'\beta'} \) in the chosen system. Since the unprimed system corresponds to \((t_c, m, \eta)\) coordinates and the primed system to \((t_c, \tau, \tau_3)\) coordinates, the matrix \( \Delta_{\alpha'}^{\beta'} \) has element

\[
\Delta_{\alpha'}^{\beta'} = \begin{pmatrix}
1 & 0 & 0 \\
0 & \frac{\partial \theta_0}{\partial m} & \frac{\partial \tau}{\partial m} \\
0 & \frac{\partial \theta_0}{\partial \eta} & \frac{\partial \tau}{\partial \eta}
\end{pmatrix} = \begin{pmatrix}
1 & 0 & 0 \\
0 & \frac{\partial \tau}{\partial m} & -\frac{\tau}{m} \\
0 & -\frac{\tau}{\eta} & -\frac{\tau}{\eta}
\end{pmatrix}
\]  

(22.39)

Finally, what is needed in laying a lattice in the space of dynamical parameters (also referred to as intrinsic parameters) is the metric with the kinematical parameter (also called extrinsic parameter) projected out: In other words one defines the 2-dimensional metric \( g_{mn} \) by

\[
g_{mn} = \gamma_{mn} - \frac{\gamma_{0m} \gamma_{0n}}{\gamma_{00}}.
\]  

(22.40)

**Metric computation in the \( \tau_0 - \tau_3 \) space**

The metric cannot be directly computed in the \((\tau_0, \tau_2)\) space. Therefore, in the previous Section we first computed the metric in the \((m, \eta)\) space and then transformed to \((\tau_0, \tau_2)\) space. The same method can also be used to find the metric in the \((\tau_0, \tau_3)\) space. However, in \((\tau_0, \tau_3)\) space one can directly compute the metric without recourse to \((m, \eta)\) coordinates. It is of interest to see whether this yields the same results as the previous method.

The starting point of our derivation is Eq. (3.7) of Owen and Sathyaprakash (Phys. Rev. D 60, 022002, 1999) for the Fourier domain phase which we shall write as:

\[
\Psi(f; \theta_1, \theta_2) = a_{01} \theta_1 v^{-5} + \left[ a_{21} \frac{\theta_1}{\theta_2} + a_{22} \left( \theta_1 \theta_2^2 \right)^{1/3} \right] v^{-3} + a_{31} \theta_2 v^{-2} + \left[ a_{41} \frac{\theta_1}{\theta_2^2} + a_{42} \left( \frac{\theta_1}{\theta_2} \right)^{1/3} + a_{43} \left( \frac{\theta_2^3}{\theta_1} \right)^{1/3} \right] v^{-1},
\]  

(22.41)

to 2nd post-Newtonian order. Here \( v = (f/f_0)^{1/3}, \theta_1 \) and \( \theta_2 \) are identical to the \( \theta^1 \) and \( \theta^2 \) parameters of Owen and Sathyaprakash defined in Eq. (3.3) there and the \( a \) coefficients are given by:

\[
a_{01} = \frac{3}{5}, \quad a_{21} = \frac{11\pi}{12}, \quad a_{22} = \frac{743}{2016} \left( \frac{25}{2\pi^2} \right)^{1/3}, \quad a_{31} = \frac{3}{2},
\]

\[
a_{41} = \frac{617}{384} \pi^2, \quad a_{42} = \frac{5429}{5376} \left( \frac{25\pi}{2} \right)^{1/3}, \quad a_{43} = \frac{15293365}{10838016} \left( \frac{5}{4\pi^4} \right)^{1/3}.
\]  

(22.42)

Differentials of the phase with respect to the coordinates \( \theta_1 \) and \( \theta_2 \) appear in the metric which we write as:

\[
\psi_m = \frac{\partial \Psi}{\partial \theta_m} = \sum_0^N \Psi_{mk} v^{k-5}.
\]  

(22.43)
where $N$ is the post-Newtonian order up to which the phase is known, or the post-Newtonian order at which the metric is desired. Expansion coefficients $\Psi_{mn}$ can be considered be $(2 \times N)$ matrix which to second post-Newtonian order is given by:

$$\Psi = \begin{bmatrix}
a_{01} & 0 & a_{21}/\theta_2 + (a_{22}/3)(\theta_2/\theta_1)^{2/3} & 0 & a_{41}/\theta_2^2 + a_{42}/(3(\theta_2^2\theta_2)^{1/3}) - (a_{43}/3)(\theta_2/\theta_1)^{4/3} \\
0 & 0 & -a_{21}/\theta_2^2 + (2a_{22}/3)(\theta_1/\theta_2)^{1/3} & a_{31} & -2a_{41}/\theta_1\theta_2^2 - (a_{42}/3)(\theta_1/\theta_2)^{1/3} + (4a_{43}/3)(\theta_2/\theta_1)^{1/3}
\end{bmatrix}.$$  

(22.44)

Using the definition of the metric introduced earlier and projecting out the $t_c$ coordinate, one finds that

$$g_{mn} = \frac{1}{2} \sum_{k,l=0}^{N} \Psi_{mk} \Psi_{nl} \left[ J(17 - k - l) - J(12 - k)J(12 - l) - \frac{(J(9 - k) - J(4)J(12 - k))(J(9 - l) - J(4)J(12 - l))}{(J(1) - J(4)^2)} \right]$$  

(22.45)

where $J$'s are the moments introduced earlier.

Algorithm

Uses

LALMAlloc
LALInspiralMoments
LALInverse3
LALMatrixTransform
LALFree

Notes
22.2.8 Module LALInspiralLongestTemplateInBank.c

To find the longest template in a template bank.

Prototypes

```c
void LALInspiralLongestTemplateInBank
   (LALStatus *status,
    UINT4 *templateLength,
    InspiralCoarseBankIn *coarseIn)
```

Description

Given the parameters of a template bank find the longest template in the bank. This is done by looking at the duration for which a signal corresponding to smallest masses lasts. One simply calls the `LALInspiralWaveLength` code for a system consisting of two stars each of mass $m_{\text{Min}}$.

Algorithm

Uses

- `LALInspiralParameterCalc`
- `LALInspiralWaveLength`

Notes

Author: Sathyaprakash, B.S.
22.2.9 Module LALInspiralMoments.c

Module to calculate the moment of the noise power spectral density.

Prototypes

```c
void LALGetInspiralMoments (
    LALStatus *status,
    InspiralMomentsEtc *moments,
    REAL8FrequencySeries *psd,
    InspiralTemplate *params
)
```

```c
void LALInspiralMoments(
    LALStatus *status,
    REAL8 *moment,
    InspiralMomentsIn pars
)
```

- **moment**, Output, the value of the moment
- **pars**, Input

Description

The moments of the noise curve are defined as

\[ I(q) = S_h(f_0) \int_{f_s/f_0}^{f_c/f_0} \frac{x^{-q}}{S_h(x)} \, dx. \]  

(22.46)

Because in practice we will always divide one of these moments by another, we do not need to include the \( S_h(f_0) \) term, which always cancels. This function calculates the integral

\[ I = \int_{f_s/f_0}^{f_c/f_0} \frac{x^{-q}}{S_h(x)} \, dx. \]  

(22.47)

It then divides this quantity by a normalisation constant which has been passed to the function. In the case of calculating the components of the metric for the signal manifold for the purpose of generating a template bank, this constant is given by \( I(7) \), because of the definition of the quantity

\[ J(q) = \frac{I(q)}{I(7/3)}. \]  

(22.48)

Algorithm

Given the exponent **parsndx** and limits of integration **parsxmin** and **parsxmax** this function returns the moment of the power spectral density specified by the frequency series **pars.shf** according to

\[ \text{moment} = \int_{\text{xmin}}^{\text{xmax}} \frac{x^{-\text{ndx}}}{S_h(x)} \, dx. \]  

(22.49)

Uses

LALDRombergIntegrate

Notes

Authors: Brown, D. A., Cokelaer, T. and Sathyaprakash, B. S.

$Id: LALInspiralMoments.c,v 1.19 2007/06/08 14:41:42 bema Exp$
22.2.10 Module `LALInspiralMomentsIntegrand.c`

Prototypes

```c
void LALInspiralMomentsIntegrand(
    LALStatus *status,
    REAL8 *integrand,
    REAL8 x,
    void *params
);
```

- `integrand`, Output, the value of the integrand
- `x`, Input, the point where the integrand is required
- `params`, Input, of type `InspiralMomentsIn` containing the details required in moments calculation

Description

The moments of the noise curve are defined as

\[
I(q) \equiv \int_{f_s/f_0}^{f_c/f_0} \frac{x-q/3}{S_h(x)} dx.
\]  

(22.50)

This function calculates the integrand of this integral, i.e. for a given `x` it calculates

\[
\frac{x-q/3}{S_h(x)}.
\]

(22.51)

by interpolating the frequency series containing `S_h(f)`.

Algorithm

Uses

LALDPolynomialInterpolation

Notes

Authors: Brown, D. A., and Sathyaprakash, B. S.

$Id: LALInspiralMomentsIntegrand.c,v 1.10 2007/06/08 14:41:42 bema Exp $
22.2.11 Module LALInspiralSetSearchLimits.c

Function which calculates the minimum and maximum values of $\tau_0$ and $\tau_3$.

Prototypes

```c
void LALInspiralSetSearchLimits (
    LALStatus *status,
    InspiralBankParams *bankParams,
    InspiralCoarseBankIn coarseIn
)
```

- `bankParams`, Output containing the boundary of search, current lattice point, etc.
- `coarseIn`, Input, specifies the parameters of the search space.

This Function calculates the minimum and maximum values of $\tau_0$ and $\tau_3$ as determined by the total mass of the binary $m$ and the symmetric mass ratio $\eta$. The function also calculates the coordinates of the first template in the bank. These coordinates are $\tau_0 = \tau_{0\text{min}}$, $\tau_3 = \tau_{3\text{min}}$.

Description

We start with the definition of the chirp times $\tau_0$ and $\tau_3$,

$$\tau_0 = \frac{5}{256(\pi f_a)^{8/3}m^{5/3}\eta}$$  \hspace{1cm} (22.52)

and

$$\tau_3 = \frac{1}{8(\pi^2 f_a^2)^{1/3}m^{2/3}\eta}$$  \hspace{1cm} (22.53)

$\tau_0$ is minimised when $\eta = 1/4$ and $m = M\text{Max}$. $\tau_0$ is maximised when $\eta = 1/4$ and $m = 2m\text{Min}$. $\tau_3$ is minimised when $\eta = 1/4$ and $m = M\text{Max}$. $\tau_3$ is maximised when $\eta = m\text{Min}(M\text{Max} - m\text{Min})/M\text{Max}^2$.

Algorithm

Uses

Notes
22.2.12 Module LALInspiralNextTemplate.c

Routine to compute the parameters of the template next to the current template, but in the positive $\tau_{2(3)}$ axis.

Prototypes

```c
void LALInspiralNextTemplate(LALStatus *status,
                             InspiralBankParams *bankPars,
                             InspiralMetric metric)
```

- `bankPars`, Output, the parameters of the bank at the next grid point; the point may, indeed, lay outside the parameter space.
- `matrix` Input, the value of the metric which would allow computation of the next lattice point (in the $\tau_{2(3)}$ direction).

Description

The coarse grid algorithm works by starting at one corner of the parameter space, incrementing along positive $\tau_0$ direction, with increments determined by the local value of the metric, till the boundary of the parameter space is reached. It then gets back to the starting point and increments along positive $\tau_{2(3)}$ direction, with an increment defined by the metric defined locally; it starts at the first point inside the parameter space but consistent with a square lattice. This routine is called each time a translation along the $\tau_{2(3)}$ direction is required.

Algorithm

Uses

None.

Notes

Author: Sathyaprakash, B. S.

$Id: LALInspiralNextTemplate.c,v 1.8 2007/06/08 14:41:42 bema Exp $
22.2.13 Module LALInspiralComputeParams.c

This function takes as input $\tau_0$, $\tau_3$ and $f_a$ (the lower frequency of the detectors sensitivity). It then calculates $m$ (the total mass of the binary), $\eta$ (the symmetric mass ratio) and the individual mass of the compact objects.

Prototypes

```c
void LALInspiralComputeParams(LALStatus *status, 
InspiralTemplate *pars, 
InspiralBankParams bankParams, 
InspiralCoarseBankIn coarseIn)
```

- `pars`, Output, inspiral wave parameter structure
- `bankParams`, Input, the parameters of the template bank
- `coarseIn`, Input, input parameters specifying the coarse bank

Description

We start with the definition of the chirp times $\tau_0$ and $\tau_3$,

$$\tau_0 = \frac{5}{256(\pi f_a)^{8/3}m^{5/3}\eta}$$

and

$$\tau_3 = \frac{1}{8(\pi^2 f_a^3)^{1/3}m^{2/3}\eta}$$

These equations may be inverted to yield

$$m = \frac{5}{32\pi^2 f_a^2} \tau_3$$

and

$$\eta = \left(\frac{2\pi^2 \tau_0^2 \tau_3^2}{25 f_a^3}\right)^{1/3}$$

The individual masses may be calculated as follows. We have

$$m = m_1 + m_2$$

and

$$\eta = \frac{m_1 m_2}{(m_1 + m_2)^2}$$

From Eq.(22.58) we may eliminate either $m_1$ or $m_2$,

$$m_1 = m - m_2$$

This may be substituted into Eq.(22.59) to give

$$\eta = \frac{(m - m_2)m_2}{(m - m_2 + m_2)^2}$$

which may be re-arranged to give

$$m_2^2 - mm_2 + \eta m^2 = 0$$

i.e.

$$m_2 = \frac{m \pm \sqrt{m^2(1 - 4\eta)}}{2}$$

Therefore, since we know that $\eta \leq 1/4$, real roots are guaranteed. If we had eliminated $m_2$ rather than $m_1$ then we would have arrived at an identical expression for $m_1$, and so of one object has mass

$$m_1 = \frac{m + \sqrt{m^2(1 - 4\eta)}}{2}$$

then the other object must have mass

$$m_2 = \frac{m - \sqrt{m^2(1 - 4\eta)}}{2}$$
Algorithm

Uses
LALInspiralParameterCalc

Notes
Module \textbf{LALInspiralValidParams.c}

Module which checks whether or not a pair of parameter values are consistent with the search space.

Prototypes

\begin{verbatim}
void LALInspiralValidParams(
    LALStatus *status,
    INT4 *valid,
    InspiralBankParams bankParams,
    InspiralCoarseBankIn coarseIn
)
\end{verbatim}

- \texttt{valid}, Output, 0 means invalid template, 1 means valid
- \texttt{bankParams}, Input
- \texttt{coarseIn}, Input

Module which checks whether or not a pair of parameter values $\tau_0$ and $\tau_2(3)$ correspond to a user specified range of component masses $(m_{\text{Min}}, m_{\text{Max}})$ OR to a minimum value of the component masses $m_{\text{Min}}$ and maximum total mass $M_{\text{Max}}$. In the first case chirptimes satisfying the constraint $m_{\text{Min}} \leq m_1, m_2 \leq m_{\text{Max}}$ are accepted as valid systems. In the second cases chirptimes satisfying the constraint $m_{\text{Min}} \leq m_1, m_2$, and $M_{\text{Max}} \leq m = m_1 + m_2$ are treated as valid.

Description

We start with the definition of the chirp times $\tau_0$ and $\tau_3$,

$$
\tau_0 = \frac{5}{256(\pi f_a)^{8/3} m^{5/3} \eta}
$$

and

$$
\tau_3 = \frac{1}{8(\pi^2 f_a^2)^{1/3} m^{2/3} \eta}
$$

These equations may be inverted to yield

$$
m = \frac{5}{32\pi^2 f_a^2} \frac{\tau_3}{\tau_0}
$$

and

$$
\eta = \left( \frac{2\pi^2}{25f_a^3} \frac{\tau_0^2}{\tau_3^{1/3}} \right)^5
$$

The individual masses may be calculated as follows. We have

$$
m = m_1 + m_2
$$

and

$$
\eta = \frac{m_1 m_2}{(m_1 + m_2)^2}
$$

From Eq.\[(22.70)\] we may eliminate either $m_1$ or $m_2$,

$$
m_1 = m - m_2
$$

This may be substituted into Eq.\[(22.71)\] to give

$$
\eta = \frac{(m - m_2)m_2}{[(m - m_2) + m_2]^2} = \frac{(m - m_2)m_2}{m^2}
$$

which may be re-arranged to give

$$
m_2^2 - mm_2 + \eta m^2 = 0,
$$
i.e.

\[ m_2 = \frac{m \pm \sqrt{m^2(1 - 4\eta)}}{2} \]  

(22.75)

Therefore, since we know that \( \eta \leq \frac{1}{4} \), real roots are guaranteed. If we had eliminated \( m_2 \) rather than \( m_1 \) then we would have arrived at an identical expression for \( m_1 \), and so of one object has mass

\[ m_1 = \frac{m + \sqrt{m^2(1 - 4\eta)}}{2} \]  

(22.76)

then the other object must have mass

\[ m_2 = \frac{m - \sqrt{m^2(1 - 4\eta)}}{2} \]  

(22.77)

This function is also given \( m_{\text{Min}} \) and \( M_{\text{Max}} \) as inputs, which it may use to calculate the minimum value of \( \eta \) which is possible with those inputs,

\[ \eta_{\text{min}} = \frac{m_{\text{Min}}(M_{\text{Max}} - m_{\text{Min}})}{M_{\text{Max}}^2} \]  

(22.78)

To recap, the function calculates \( m, \eta, \eta_{\text{min}} \) and \( m_{1,2} \). It then checks that

\[ \eta_{\text{min}} \leq \eta \leq \frac{1}{4} \]  

(22.79)

and that

\[ m_1 \geq m_{\text{Min}} \]  

(22.80)

and

\[ m_2 \geq m_{\text{Min}} \]  

(22.81)
22.2.15 Module LALInspiralValidTemplate.c

Module which checks whether or not a given template should be kept in the template list.

Prototypes

```c
void LALInspiralValidTemplate(
    LALStatus *status,
    INT4 *valid,
    InspiralBankParams bankParams,
    InspiralCoarseBankIn coarseIn)
```

- **valid**, Output, 0 means invalid template, 1 means valid
- **bankParams**, Input
- **coarseIn**, Input

Description

Given the parameter values $\tau_0$ and $\tau_{2(3)}$ this code checks to see if they correspond to physical values of the masses of a binary and their symmetric mass ratio $\eta$. The parameter values will be accepted as valid parameters even though they may not lie within the search space but their span does, as described below. At the moment the code allows extra templates only in the positive-$\tau_{2(3)}$ direction only. We have found that placing templates in other directions is redundant.

Algorithm

Consider the point $(\tau_0, \tau_{2(3)})$ describing the template, and also a point at $(\tau_0, \tau_{2(3)} - \text{bankParams.dx1}/2)$, i.e. displaced in the negative $\tau_{2(3)}$ direction. $\text{bankParams.dx1}$ is calculated from the metric and corresponds to the vertical spacing between the horizontal rows of templates being considered. Accept the template if at least one of those points is within the search space.

Uses

Notes
22.2.16 Module LALInspiralUpdateParams.c

Module to update the parameters used in creating a coarse bank based on a square lattice.

Prototypes

```c
void LALInspiralUpdateParams(LALStatus *status,
                            InspiralBankParams *bankParams,
                            InspiralMetric metric,
                            REAL8 minimalmatch)
```

- **bankParams** Output, refreshed to get the next location
- **metric** Input, metric at the current location
- **minimalmatch** Input, the minimal match

Description

While scanning the \( \tau_0 \)-direction after reaching the boundary of the parameter space, we have to return to the starting point of the same line and use the metric there to increment one step upwards in the direction of \( \tau_2(3) \), to a *template list*.

The \( dx_i \) returned by this function gives the spacing for a square lattice (e.g., \( dx_i \) as given in Owen PRD 53).

Algorithm

Copy the parameters in the temporary parameter structure to the current parameter structure.

Uses

None.

Notes
22.2.17 Module LALMatrixTransform.c

A routine to transform a second rank tensor under a given transformation.

Prototypes

```c
void LALMatrixTransform (LALStatus *status,
    INT4 n,
    REAL8 **data1,
    REAL8 **data2,
    REAL8 **data3)
```

- **n**, Input, dimension of the matrix (currently, and possibly always, only 3)
- **data1**, Input, transformation matrix
- **data2**, Input, matrix whose transformation is required
- **data3**, Output, transformed matrix

Description

Given the matrix of transformation in **data1** and a second rank tensor **data2**, this routine computes the transformed tensor in **data3**.

Algorithm

\[ C_{ij} = A_{im}A_{jl}B_{ml}. \]

Uses

None.

Notes
22.2.18 Module LALDeterminant.c

Module to calculate the determinant of a 3-dimensional matrix $g_{ij}$.

Prototypes

```c
void LALDeterminant3(LALStatus *status,
    REAL8 *determinant,
    REAL8 **matrix)
```

- **determinant**, Output, determinant of the matrix.
- **matrix**, Input, the input ($3 \times 3$) matrix whose determinant is required.

Description

This code computes the determinant of a 3-dimensional matrix.

Algorithm

Given a matrix $g_{ij}$ its determinant is computed using the formula $g = \epsilon^{ijk} g_{i1} g_{j2} g_{k3}$, where $\epsilon$ is the totally anti-symmetric tensor in 3-dimensions.

Uses

None.

Notes

Don’t ever generalise this to higher dimensions since this would take many more operations than some of the standard routines.
22.2.19 Module `LALInverse3.c`
Uses $g^{ij} = 1/(2g)\epsilon^{ikl}\epsilon^{jab}g_{ka}g_{lb}$ to compute the inverse.

Prototypes

```c
void LALInverse3(LALStatus *status,
                 REAL8 **inverse,
                 REAL8 **matrix)
```

- `inverse`, Output, inverse of the $(3 \times 3)$ input matrix
- `*matrix`, Input, matrix whose inverse is required

Description
Uses $g^{ij} = 1/(2g)\epsilon^{ikl}\epsilon^{jab}g_{ka}g_{lb}$ to compute the inverse; though not efficient, it is good enough for the 3-d matrix that we have. Prevents the need for having a new library.

Algorithm
Uses

`LALDeterminant3`

Notes
Do not generalise to more than 3 dimensions.
22.2.20 Module **LALInspiralSetParams.c**

A routine that fills an `InspiralTemplate` structure based on the values in the `InspiralCoarseBankIn` structure.

**Prototypes**

```
void LALInspiralSetParams(LALStatus *status,
                          InspiralTemplate *tempPars,
                          InspiralCoarseBankIn coarseIn)
```

- `tempPars`, Output
- `coarseIn`, Input

**Description**

This function takes as an input a structure of type `InspiralCoarseBankIn` and it fills up the elements of a structure of type `InspiralTemplate`. The function sets the fields `massChoice`, `ieta`, `signalAmplitude`, `tSampling`, `fLower`, `fCutoff`, `order`, `approximant`, `nStartPad`, `nEndPad`.

**Algorithm**

Uses

None

**Notes**
22.2.21 Module **LALRectangleVertices.c**

Module to find the vertices of a rectangle given its centre, half side-lengths and orientation angle.

**Prototypes**

```c
void LALRectangleVertices(
    LALStatus *status,
    RectangleOut *out,
    RectangleIn *in
)
```

- **out**, Output.
- **in**, Input.

**Description**

This code computes the vertices of a rectangle for plotting a grid of templates with xmgr, useful when looking at the minimal-match-rectangles around mesh points in a template bank.

**Algorithm**

Given the centre \((x_0, y_0)\) and half-sides \((dx, dy)\), the vertices of a rectangle in a diagonal coordinate system are given by

\[
\begin{align*}
x_1 &= x_0 - dx, & y_1 &= y_0 - dy, \\
x_2 &= x_0 + dx, & y_2 &= y_0 - dy, \\
x_3 &= x_0 + dx, & y_3 &= y_0 + dy, \\
x_4 &= x_0 - dx, & y_4 &= y_0 + dy.
\end{align*}
\]

The coordinates of a rectangle oriented at an angle \(\theta\) is found by using the formulas

\[
\begin{align*}
x' &= x \cos(\theta) - y \sin(\theta), \\
y' &= y \cos(\theta) + x \sin(\theta).
\end{align*}
\]

The function returns five coordinate points \((1,2,3,4,1)\), and not just the four vertices, to help a plotting programme to complete the rectangle.

**Uses**

None.

**Notes**
22.2.22 Module LALInsidePolygon.c

Module to check whether a point with coordinates \((x_0, y_0)\) is inside a polygon defined by the vectors \((v_x, v_y)\), which size \(n\) must be provided. The functions returns 1 if the point is inside or 0 otherwise.

Prototypes

```c
void LALInsidePolygon( LALStatus *status,
REAL4 *inputx,
REAL4 *inputy,
INT4 n,
REAL4 x0,
REAL4 y0,
INT4 *valid)
```

- \(v_x, v_y\) Input, two arrays of floats defining the polygon.
- \(n\) Input, the size of the vectors.
- \(x_0, y_0\) Input, the coordinate of the point.
- \(valid\) Output, 0 if outside and 1 if inside.

Description/Algorithm

None

Uses

None.

Notes

Tested in matlab codes and some BCV tests within lal/lalapps.
## 22.2. Header LALInspiralBank.h

### 22.2.23 Module LALInspiralHybridHexagonalBank.c

**Prototypes**

```c
void
LALInspiralCreatePNCoarseBankHybridHexa(
   LALStatus *status,
   InspiralTemplateList **list,
   INT4 *nlist,
   InspiralCoarseBankIn coarseIn
);
```

**Description**

This code does almost the same as the standard Hexagonal Bank code. However, once the templates cover both the equal line and an other line ($m_1 = m_{\text{Min}}$ or $m_2 = m_{\text{Max}}$), then there is no need to carry on any square/hexagonal placement. One can simply populate templates along a bissectrice.

**Algorithm**

The algorithm is identical to the hexagonal placement. However, once a template covers both the equal mass line and the upper boundary, then the hexagonal placement stops. So, an additional placement is needed to finalise the bank. In principle the placement needs to be completed on both side of the template bank, at low mass and high mass. So, we should start from the two templates which covers the two boundaries and populate the parameter space along a bissectrice.

The coordinates of the bissectrice at a given $\tau_0$ coordinate is estimated by tracing a vertical line in the $\tau_0/\tau_3$ plane, estimate the value of $\tau_3$ on the upper boundary and low boundary ($\eta = 1/4$ line), and finally take the mean of the two values. Although, is an approximation since we should also take into account the orientation of the ellipse, we think this is good enough. The vertical line crosses the parameter space on the $\eta = 1/4$ line and the other parameter space boundary which is define either by (1) $m_1 = \text{variable}$ and $m_2 = m_{\text{Min}}$ or (2) $m_1 = \text{variable}$ and $m_2 = m_{\text{Max}}$. Concerning (1), $\eta = 1/4$, this is a trivial computation, since

$$
\tau_3 = \frac{A^3}{\eta} \left(\frac{\eta \tau_0}{A_0}\right)^{2/5},
$$

which in the case of $\eta = 1/4$ simply becomes:

$$
\tau_3 = 4.43 \left(\frac{\tau_0}{4.40}\right)^{2/5}.
$$

In the case (2), if $\tau_0$ is provided, if we can extract the total mass and $\eta$ parameter, then $\tau_3$ is given by

$$
\tau_3 = \frac{A^3}{\eta} M^{-2/3}.
$$

So, we need $M$ and $\eta$. Starting from

$$
\tau_0 = \frac{A_0}{\eta} (M)^{-5/3},
$$

we can extract a cubic equation

$$
x^3 - px + q = 0
$$

where $x = M^{1/3}$, $p = -\frac{A_0}{\tau_0 m_{\text{Extreme}}}$ and $q = -m_{\text{Extreme}} = 0$. $m_{\text{Extreme}}$ is either set to $m_{\text{Min}}$ or $m_{\text{Max}}$ depending on which side of the parameter space we are.

The solution for $x$ is standard and takes the expression :

$$
x = \left( -\frac{q}{2} - \frac{1}{2} \sqrt{\frac{27q^2 + 4p^3}{27}} \right)^{\frac{1}{3}} + \left( -\frac{q}{2} + \frac{1}{2} \sqrt{\frac{27q^2 + 4p^3}{27}} \right)^{\frac{1}{3}};
$$

(22.87)
Figure 22.5: Example of hybrid hexagonal placement. Once an ellipse covers the upper and lower boundary, then the hexagonal placement stops. This occurs necessarily at low and high mass range.

Figure 22.6: Example of hybrid hexagonal placement. Once the ellipses covers the upper and lower part of the parameter space (at $\tau_0=3.6$ and $\tau_0=0.4$), then the placement is switched from the hexagonal to a placement along the bissectric of the upper/lower boundaries as described in the text.
Uses

LALPopulateNarrowEdge()
XLALInspiralBissectionLine()

Notes
22.2.24 Program *CoarseTest.c*

Test code for the inspiral modules. A template bank can be created either using a full range for component masses of the binary \( m_1 \) and \( m_2 \), that is \((m_{\text{Min}}, m_{\text{Max}})\), OR minimum value of the component masses \( m_{\text{Min}} \) and maximum value of the total mass \( M_{\text{Max}} \). In the first case chirptimes satisfying the constraint \( m_{\text{Min}} \leq m_1, m_2 \leq m_{\text{Max}} \) are accepted as valid systems. In the second case chirptimes satisfying the constraint \( m_{\text{Min}} \leq m_1, m_2, \) and \( M_{\text{Max}} \leq m = m_1 + m_2 \), are treated as valid. Users are expected to provide both \( m_{\text{Max}} \) and \( M_{\text{Max}} \).

For LALLIGOIPsd the choice \( m_{\text{Min}} = 1M_\odot \) \( m_{\text{Max}} = 20M_\odot \) gives 2292 templates, while the same \( m_{\text{Min}} \) but choosing \( M_{\text{Max}} = 40M_\odot \) instead gives 2512 templates – about 10% increase. However, the extra templates are ALL short-lived templates and therefore potentially trigger a large number of false alarms, as noted by Brown in E7 analysis.

This test code creates a template bank and stores it into CoarseTest.out . Then, it creates a finer template bank around a sub-set of the original template bank and stores it in the same output file.

**Usage**

Input the following values of the InspiralCoarseBankIn structure to create a template bank:

- Minimum component mass in solar masses. \( m_{\text{Min}} = 1.0 \);
- Maximum component mass in solar masses. \( m_{\text{Max}} = 20.0 \);
- Maximum total mass. **This should be specified independently of whether or not \( m_{\text{Max}} \) is specified.** It is used in setting up a rectangular area in the space of chirptimes where the templates will be laid. \( M_{\text{Max}} = 40.0 \);
- The type of search space. \( \text{massRange} = \text{MinComponentMassMaxTotalMass}; \) OR \( \text{massRange} = \text{MinMaxComponentMasses}; \)
- Coarse bank minimal match \( \text{coarseIn}\rightarrow\text{mmCoarse} = 0.80 \);
- Fine bank minimal match \( \text{mmFine} = 0.97 \);
- Lower frequency cutoff \( f_{\text{Lower}} = 40. \);
- Upper frequency cutoff \( f_{\text{Upper}} = 1024L \);
- Whether or not iso should be used as an upper frequency cutoff (Currently not used; so please specify \( f_{\text{Upper}}. \) \( \text{coarseIn}\rightarrow\text{iflso} = 0 \));
- Sampling rate \( t_{\text{Sampling}} = 4000L \);
- Space in which templates should be created: whether \((\tau_0, \tau_2)\) or \((\tau_0, \tau_3)\). \( \text{coarseIn}\rightarrow\text{space} = \text{Tau0Tau2}; \) OR \( \text{coarseIn}\rightarrow\text{space} = \text{Tau0Tau3}; \) OR
- Order and type of the approximant to be used in template generation. These members will NOT be used in creating a template bank but in filling up the \( \text{InspiralTemplate} \) structure created by the bank. \( \text{order} = \text{twoPN}; \) \( \text{coarseIn}\rightarrow\text{approximant} = \text{TaylorT1}; \)
- \( \text{minimum value of } \eta \) \( \text{etamin} = m_{\text{Min}} \times (M_{\text{Max}} - m_{\text{Min}})/\text{pow}(M_{\text{Max}},2.); \)
- Finally, fill the psd structure (see test code for an example). This involves allocating memory to vector \( \text{shf.data} \) and filling it with noise PSD as also choosing the starting frequency and the frequency resolution.

**CoarseTest**

**Description**

This test code gives an example of how one calls \text{LALInspiralCreateCoarseBank} and \text{LALInspiralCreateFineBank} modules.
Exit codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
</table>

The status codes in the table above are stored in the constants _E<name>_, and the status descriptions in _MSGE<name>_. The source code with these messages is in CoarseTest.c on line 1.136.

Uses

lalDebugLevel
LALInspiralCreateCoarseBank
LALInspiralCreateFineBank
LALFree
LALCheckMemoryLeaks

Notes
22.2.25 Program **CoarseTest2.c**

Test code for the bank modules.

**Usage**

`CoarseTest2`

**Description**

This test code gives an example of how to generate a template bank and generates vertices of the ambiguity 'rectangle' around each lattice point suitable for plotting with xmgr or xgrace. This code generates the template bank for physical template families such as EOB, TaylorT1, ... save the coordinates into a file called `CoarseTest2.out` and then creates a rectangle for each coordinates which is inscribed into the ambiguity ellipse function. It can be extended to the BCV case easily by replacing approximant (BCV) and parameter space (Psi0andPsi3). The code has to be changed to use psi0/psi3 instead of tau0/tau3 though.

**Exit codes**

<table>
<thead>
<tr>
<th><code>&lt;name&gt;</code></th>
<th>code</th>
<th>description</th>
</tr>
</thead>
</table>

The status codes in the table above are stored in the constants `_E<name>`, and the status descriptions in `_MSGE<name>`. The source code with these messages is in `CoarseTest2.c` on line `1.64`.

**Uses**

- `lalDebugLevel`
- `LALRectangleVertices`
- `LALInspiralCreateCoarseBank`

**Notes**
22.2.26 Program ChirpSpace.c

Test code for LALInspiralParameterCalc module. If the variable type is set to 1 the code works out the boundary of the region enclosed by the parameter space specified by maximum total mass and minimum companion masses, as given in mmin and Mmax. If the variable type is set to 0 it computes the boundary of the region corresponding to the companion masses in the range defined by mmin and mmax.

Uses
LALInspiralParameterCalc

Notes
22.2.27 Program GetOrientationEllipse.c

Test code for the bank modules.

Usage

./getOrientationEllipse

Description

This code illustrates the use of several functions such as LALInspiralParameterCalc, LALGetInspiralMoments, and LALInspiralComputeMetric. It shows how to defined a suitable InspiralCoarseBankIn structure so as to extract the metric components for a set of binary parameters. In this example, we first declare all the relevant parameter needed (minimum and maximum mass, fLower, design sensitivity curve and so on), which can be changed by the user before compilation.

Then, a loop spans a square parameter space defined by tau0 in the range [.1,40] seconds and tau3 in [1, 2] seconds. For each set of parameter, the metric is computed and the code prints on stdout the value of the coordinate used (tau0, tau3) and the orientation of the metric in degrees. We do not check whether a template is valid or not in this code but one could have use a function such as LALInspiralValidtemplate to do so.

Notes
22.2.28 Program **InspiralSpinBankTest**

Tests InspiralSpinBank().

**Usage**

`InspiralSpinBankTest`

This program uses InspiralSpinBank() to generate a template bank from command line parameter input. It also has the option to make a *MATHEMATICA®* notebook using LALMath3DPlot() which will plot the 3D template bank. If the `-b` option is specified, the program will read the template bank from an XML file instead of generating it. (This only works if LAL is compiled with metaio.)

**Command line options**

- `-b` Specifies the XML file to read template bank from. (Only with metaio.)
- `-n` Specifies the minimum smaller mass between 0 and 5.0 $M_\odot$.
- `-x` Specifies the maximum smaller mass between 0 and 5.0 $M_\odot$.
- `-m` Specifies the minimum mismatch threshold (typically 0.03) but for the sake of testing it is best to pick a value $O[1]$ to save compiling time.
- `-p` Specifies that the program should generate a *MATHEMATICA®* notebook “Math3DNotebook.nb”.

**Exit codes**

<table>
<thead>
<tr>
<th><code>&lt;name&gt;</code></th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NORM</td>
<td>0</td>
<td>&quot;Normal exit&quot;</td>
</tr>
<tr>
<td>MEM</td>
<td>1</td>
<td>&quot;Memory allocation error&quot;</td>
</tr>
<tr>
<td>SUB</td>
<td>2</td>
<td>&quot;Subroutine error&quot;</td>
</tr>
<tr>
<td>FILE</td>
<td>4</td>
<td>&quot;File I/O error&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants `INSPIRALSPINBANKTESTC_E<name>`, and the status descriptions in `INSPIRALSPINBANKTESTC_MSGE<name>`. The source code with these messages is in `InspiralSpinBankTest.c` on line 1.118.

**Notes**

- The metric used in InspiralSpinBank() is only valid for binary systems with a total mass < $15M_\odot$ where the minimum larger mass is at least twice the maximum smaller mass. Choosing mass values that violate these conditions will cause an error message.
- Only up to 20,000 templates will be read from an XML file. Making an animated image will start bogging most systems down with more than a few thousand templates, so you can switch the option off by editing the notebook within Mathematica.
22.2.29 Program SpaceCovering.c

Test code for the bank modules.

Usage

SpaceCovering
   --template [TaylorT1, EOB ....]
   --grid-spacing [square, hexagonal, squareOriented, hexagonalOriented]

   --template BCV --grid-spacing Hexagonal gives 178 templates and --template EOB grid-spacing Hexagonal gives 532 templates

Description

This test code gives an example of how to generate a template bank and generates vertices of the ambiguity 'rectangle' around each lattice point suitable for plotting with xmgr or xgrace.

   This is valid for various template bank placement (Hexagonal, SquareNorOriented). This is valid for BCV and physical template families.

Uses

lalDebugLevel
LALRectangleVertices
LALInspiralCreateCoarseBank

Notes
22.2.30 Module LALInspiralComputePTFMetric.c

Module to compute the components of the metric which is used to describe distances on Physical Template Family signal manifold.

Prototypes

```c
INT4 XLALInspiralComputePTFIntrinsticMetric (  
    InspiralMetric   *metric,  
    REAL8FrequencySeries *psd,  
    InspiralTemplate   *params
)
```

- `metric`, Output, the metric at the lattice point defined by `params`
- `psd`, Input, the power spectral density of the data
- `params`, Input, the parameters where metric must be computed in the computation of the metric.

```c
INT4 XLALInspiralComputePTFFullMetric (  
    InspiralMetric   *metric,  
    REAL8FrequencySeries *psd,  
    InspiralTemplate   *params
)
```

- `metric`, Output, the metric at the lattice point defined by `params`
- `psd`, Input, the power spectral density of the data
- `params`, Input, the parameters where metric must be computed in the computation of the metric.

Description

We calculate the components of the metric using the procedure outlined by Yi.

Algorithm

Uses

LALMalloc
LALFree

Notes

Author: Yi Pan, Duncan Brown

$Id: LALInspiralComputePTFMetric.c,v 1.3 2007/11/08 19:20:56 duncan Exp $

References

Chapter 23

Package findchirp

This package contains LAL routines to search for binary inspiral chirps using templated matched filtering and the $\chi^2$ veto. The findchirp package is designed to allow the user to filter interferometer data and produce a list of candidate inspiral events. It also contains functionality to perform simulation and testing of the inspiral search. Conceptually the package is divided into the following parts:

- Processing the raw interferometer input data into a form that can be used for the filtering process.
- Processing an inspiral chirp template into a form that can be used by the filter, generating the chirp template internally, if necessary.
- Using the processed input data, construction of a statistic on which to search for chirps and searching for candidate events.
- Constructing a veto statistic to apply to candidate events to reduce the possibility of false alarms.

We introduce the conventions used in the package and then describe the theory and implementation of the code. An overview of the package is as follows:

1. The header FindChirpDatatypes.h provides the core data types definitions used by all the findchirp routines.
2. The header FindChirp.h and modules grouped therein provide the core functionality of the package. This includes code to perform matched filtering and search for chirps with a signal to noise ratio above a given threshold. This header also provides an interface to inject time domain waveforms so that various Monte Carlo simulations may be run using the same code that is used to actually search for the chirps.
3. The header FindChirpSP.h provides functionality needed to filter interferometer data using second order post-Newtonian inspiral chirps generated by the stationary phase approximation to the inspiral waveform.
4. The header FindChirpTD.h provides functionality to filter interferometer data using TaylorT1, TaylorT2 or TaylorT3 time domain waveforms generated by the inspiral package.
5. The header FindChirpChisq.h and associated module provide functionality to perform a $\chi^2$ veto on candidate events generated by the FindChirpFilter() function.
6. The header FindChirpBCV.h provides functionality to filter interferometer data using the BCV detection template family.
7. The header FindChirpBCVSpin.h provides functionality to filter interferometer data using the BCV detection template family.
23.1 Conventions

We follow the standards for LSC code in the LIGO technical note T010095. All the findchirp functions measure mass in units of $M_{\odot}$, time in units of seconds and distance in units of megaparsecs.

23.1.1 Conventions for Discrete Quantities

The raw output of the interferometer is the error signal from the length sensing and control servo, LSC-AS_Q. Although this signal is a dimensionless quantity, we say that it has units of “counts” and we denote it by $v(t)$. The calibrated detector output is related in the frequency domain to the raw detector output by the detector response function according to

$$\tilde{s}(f) = R(f;t)\tilde{v}(f)$$

where $\tilde{v}(f)$ is the forward Fourier transform of $v(t)$ given by

$$\tilde{v}(f) = \int_{-\infty}^{\infty} dt \ v(t) e^{-2\pi ift}.$$  

$R(f;t)$ is the (complex) response function of the detector at time $t$ and has units of strain/count. In practice, the interferometer output is a discretely sampled quantity with sampling interval $\Delta t$. Henceforth, we let $N$ be a power of 2 and follow the convention that the subscript $j$ refers to discretely sampled time domain quantities and the subscript $k$ to discretely sampled frequency domain quantities. The frequency domain quantity $\tilde{v}(f_k)$ denotes the value of the continuous function $\tilde{v}(f)$ at a particular frequency, labeled $f_k = k/(N\Delta t)$. If the units of $v_j$ are counts, then $\tilde{v}(f_k)$ has units of counts/Hz. We define the quantity $\tilde{v}_k$ by $\tilde{v}_k = \tilde{v}(f_k)/\Delta t$, which has units of counts. If $k$ is negative, this corresponds to negative frequencies.

23.1.2 The Discrete Fourier Transform

If $v(t_j)$ is sampled at intervals of $\Delta t$, then the sampling theorem tells us that $v(t_j)$ is bandwidth limited to the frequency range $-f_{Ny} \leq f \leq f_{Ny}$, where

$$f_{Ny} = \frac{1}{2\Delta t}$$

is the Nyquist critical frequency. Any power in $v(t)$ at frequencies above $f_{Ny}$ will be aliased into the range $-f_{Ny} \leq f \leq f_{Ny}$, corrupting the signal. To prevent this, signals of frequency higher than $f_{Ny}$ in the interferometer output are removed using analog low-pass filters before the signal is digitized. Therefore $v(t_j)$ completely determines the signal $v(t)$ in the band of interest. We may approximate the Fourier transform of this band limited signal $v(t_j)$ by

$$\tilde{v}(f_k) \rightarrow \sum_{j=0}^{N-1} \Delta t \ v(t_j)e^{-2\pi if_k t_j} = \Delta t \sum_{j=0}^{N-1} v_j e^{-2\pi ijk/N},$$

where $-(N/2 + 1) \leq k \leq N/2$ and the symbol $\rightarrow$ means equal to under discretization. Notice that the approximation to the Fourier transform is periodic in $k$ with period $N$ and so

$$\tilde{v}_{-k} = \tilde{v}_{N-k} \quad k = 1, \ldots, N - 1.$$  

Thus we let $k$ vary from 0 to $N - 1$ where zero frequency (DC) corresponds to $k = 0$, positive frequencies $0 < f < f_{Ny}$ to values in the range $0 < k < N/2$ and negative frequencies $-f_{Ny} < f < 0$ correspond to values in the range $N/2 < k < N$. The value $k = N/2$ approximates the value of the Fourier transform at both $-f_{Ny}$ and $f_{Ny}$; both these values are equal due to the periodicity of the discrete transform defined by

$$\tilde{v}_k = \sum_{j=0}^{N-1} v_j e^{-i2\pi jk/N}.$$  

We may estimate the discrete inverse Fourier transform in a similar way, using the relation

$$\Delta f = f_{k+1} - f_k = \frac{k + 1}{N\Delta t} - \frac{k}{N\Delta t} = \frac{1}{N\Delta t}$$

where $0 \leq k \leq N-1$.
to obtain
\[ v_j = \frac{1}{N} \sum_{k=0}^{N-1} \tilde{v}_k e^{2\pi i j k / N}. \] (23.8)

Note that the LAL inverse FFT routines omit the factor of $1/N$ as documented in the fft package.

### 23.1.3 Power Spectral Densities

The one sided power spectral density $S_n(|f|)$ of $n(t)$ to be
\[ \langle \tilde{n}(f)\tilde{n}^*(f') \rangle = \frac{1}{2} S_n(|f|) \delta(f - f') \] (23.9)
where angle brackets denote an average over different realizations of the noise. If $n(t)$ has units of $U$ then $\tilde{n}(f)$ has units of (time) $\times$ $U$. The units $\delta(f - f')$ are (time), since
\[ \int_{-\infty}^{\infty} \delta(f) \, df = 1 \] (23.10)
is a dimensionless quantity and $df$ has units (time)$^{-1}$. Therefore we see that $S_n(|f|)$ has units of (time) $\times$ $U^2$.

If we replace $\tilde{n}(f_k)$ with the discretely sampled quantities $\tilde{n}_k = \tilde{n}(f_k)$, we obtain
\[ \langle \tilde{n}_k \tilde{n}_{k'}^* \rangle = \frac{N}{2\Delta t} S(|f_k|) \delta_{kk'} \] (23.11)
where $\delta_{kk'}$ is the dimensionless Kronecker $\delta$-function, obtained by discretization of the continuous $\delta$-function:
\[ \delta(f - f') \to N\Delta t \delta_{kk'} \] (23.12)

Equation (23.11) defines $S(|f_k|)$ in terms of the discrete frequency domain quantities.
23.2 Header FindChirpDatatypes.h

Provides core prototypes for the core datatypes using in findchirp.

Synopsis

```
#include <lal/FindChirpDatatypes.h>
```

Error codes

None.

Types

Structure *FindChirpStandardCandle*

Structure used to contain a binary inspiral standard candle. distance is the distance in Mpc at which an optimally oriented binary with the mass parameters stored in tmplt would produce the signal-to-noise ratio squared rhosq.

```
typedef struct
tagFindChirpStandardCandle
{
    CHAR ifo[3];
    InspiralTemplate tmplt;
    REAL4 rhosq;
    REAL4 sigmasq;
    REAL4 distance;
} FindChirpStandardCandle;
```


InspectorTemplate tmplt Binary parameters used to compute the standard candle.

REAL4 rhosq The signal-to-noise ratio squared $\rho^2$ of the candle.

REAL4 sigmasq The variance of the matched filter $\sigma^2$ for the data used to calculate the standard candle.

REAL4 distance The distance at which an optimally oriented inspiral with the masses given by tmplt would give the signal-to-noise ratio squared rhosq.

Structure *DataSegmentVector*

Structure used to contain an array of DataSegments. DataSegments are defined in the header DataBuffer.h of the framedata package. Each DataSegment contains an INT4 number used to identify the data segment and pointers to a data channel (REAL4TimeSeries *chan), a power spectral estimate (REAL4FrequencySeries *spec) and a response function (COMPLEX8FrequencySeries *resp).

```
typedef struct
tagDataSegmentVector
{
    UINT4 length;
    DataSegment *data;
} DataSegmentVector;
```

UINT4 length Number of DataSegment structures in the vector.

DataSegment *data Pointer to an array of DataSegment structures.
Structure InspiralTemplateNode
This structure provides a method of constructing doubly linked lists of InspiralTemplate structures.

typedef struct
tagInspiralTemplateNode
{
    struct tagInspiralTemplateNode *next;
    struct tagInspiralTemplateNode *prev;
    InspiralTemplate *tmpltPtr;
}
InspiralTemplateNode;

struct tagInspiralTemplateNode *next The next structure in the linked list.

struct tagInspiralTemplateNode *prev The previous structure in the linked list.

InspiralTemplate *tmpltPtr A pointer to an InspiralTemplate structure containing the template parameters.

Structure InspiralTemplateNodeList
This structure provides a method of constructing linked lists of InspiralTemplateNode structures (as if it were not already complicated enough). Actually it is necessary to create a list of sub banks for the template bank veto so that roughly 20 or so templates can be filtered and stored in memory at one time.

typedef struct
tagInspiralTemplateNodeList
{
    struct tagInspiralTemplateNodeList *next;
    InspiralTemplateNode *nodePtr;
}
InspiralTemplateNodeList;

struct tagInspiralTemplateNodeList *next The next structure in the linked list.

InspiralTemplateNode *nodePtr A pointer to an InspiralTemplateNode structure.

Structure FindChirpSubBank
This structure provides contains subbanks for the template bank veto.

typedef struct
tagFindChirpSubBank
{
    UINT4 subBankSize;
    InspiralTemplate *bankHead;
    struct tagFindChirpSubBank *next;
}
FindChirpSubBank;

InspiralTemplate *bankHead A pointer to an InspiralTemplate structure which is the head of linked list of templates.

struct tagFindChirpSubBank *next The next structure in the linked list.
Structure `FindChirpSegment`

This structure contains the conditioned input data and its parameters and is one of the inputs to the `FindChirpFilter()` function.

```c
typedef struct {
    COMPLEX8FrequencySeries *data;
    COMPLEX8FrequencySeries *dataBCV;
    REAL4TimeSeries *dataPower;
    UINT4Vector *chisqBinVec;
    UINT4Vector *chisqBinVecBCV;
    REAL8 deltaT;
    REAL4Vector *segNorm;
    REAL4Vector *tmpltPowerVec;
    REAL4Vector *a1;
    REAL4Vector *b1;
    REAL4Vector *b2;
    REAL4Vector *tmpltPowerVecBCV;
    REAL4 fLow;
    UINT4 invSpecTrunc;
    UINT4 number;
    UINT4 analyzeSegment;
    INT4 level;
    Approximant approximant;
} FindChirpSegment;
```

COMPLEX8FrequencySeries *data The conditioned data used as part of the matched filter correlation. The exact content of this structure is determined by which data conditioning routine is called (stationary phase, time domain, BCV or spinning BCV). The data in this structure is denoted \( \tilde{F}_k \) and the vector is of length \( N/2 + 1 \). For frequency domain templates (FindChirpSP, BCV and BCVSpin) it contains:

\[
\tilde{F}_k = \frac{\mathcal{d} \tilde{v}_k \left( \frac{k}{N} \right)^{-\frac{3}{2}}}{d^2 |R|^2 S_v (|f_k|)}.
\]  

(23.13)

For time domain templates (GeneratePPN, TaylorT1, TaylorT2, TaylorT3, PadeT1, EOB) it contains

\[
\tilde{F}_k = \frac{\mathcal{d} \tilde{v}_k}{d^2 |R|^2 S_v (|f_k|)}.
\]  

(23.14)

COMPLEX8FrequencySeries *dataBCV Conditioned input data used only for the BCV templates. The conditioning performed is as described in the documentation for the module `FindChirpBCVData.c`

UINT4Vector *chisqBinVec A vector containing the indices of the boundaries of the bins of equal power for the \( \chi^2 \) veto for this segment. The vector is of length \( p + 1 \), where \( p \) is the number of \( \chi^2 \) bins. If no \( \chi^2 \) veto is performed, this may be NULL.

UINT4Vector *chisqBinVecBCV A vector containing the indices of the boundaries of the bins of equal power for the second contribution to the \( \chi^2 \) statistic for the BCV templates for this segment.

REAL8 deltaT The time step \( \Delta t \) of the input data channel used to create the `FindChirpSegment`.

REAL4Vector *segNorm The quantity segment dependent normalization quantity \( S_k \). The vector is of length \( N/2 + 1 \). For stationary phase templates the segment dependent normalization is

\[
S_k = \sum_{k'=1}^{k} \frac{\left( \frac{k'}{N} \right)^{-\frac{3}{2}}}{d^2 |R|^2 S_v (|f_{k'}|)} 
\quad 1 \leq k \leq N/2
\]  

(23.15)
and can be computed once per data segment and re-used for each template. For time domain templates, the segment dependent normalization is
\[ S_k = \sum_{k'=1}^{k} \frac{\tilde{h}_{k'}}{d^2|R|^2S_v(|f_{k'}|)} \quad 1 \leq k \leq N/2 \]  
(23.16)
and it must be recomputed for each template \( \tilde{h}_k \).

**REAL4Vector **tmpltPowerVec Vector of length \( N/2 + 1 \) containing the weighted power in the template. For frequency domain templates, this is the summand in equation (23.15)
\[ \text{tmpltPowerVec} - > \text{data}[k] = \left( \frac{k}{N} \right)^{-\frac{3}{2}} \frac{\tilde{h}_k}{d^2|R|^2S_v(|f_k|)}. \]  
(23.17)
and can be computed once then re-used for all templates. For time domain templates, this is the summand in equation (23.15)
\[ \text{tmpltPowerVec} - > \text{data}[k] = \frac{\tilde{h}_k}{d^2|R|^2S_v(|f_k|)}. \]  
(23.18)
which must be re-computed for each template \( \tilde{h}_k \). This quantity is used in the computation of the \( \chi^2 \) bin boundaries and the re-computation of \( S_k \) for time domain templates.

**REAL4 a1** BCV-template normalization parameter.
**REAL4 b1** BCV-template normalization parameter.
**REAL4 b2** BCV-template normalization parameter.

**REAL4Vector **tmpltPowerVecBCV Additional weighted template power for BCV templates.

**REAL4 fLow** The (frequency domain) low frequency cutoff for the matched filter, \( f_{\text{low}} \).

**UINT4 invSpecTrunc** The number of points to which the inverse power spectrum \( S(|f_k|) \) is truncated to in the time domain in order to truncate the impulse response time of the matched filter.

**UINT4 number** A unique identification number for the `FindChirpDataSegment`. This corresponds to the number in the `DataSegment` from which the conditioned data was computed.

**INT4 level** A search level, used by the hierarchical search engine to determine if a particular data segment should be filtered against a particular template.

**Structure** `FindChirpSegmentVector`

A vector of `FindChirpSegment` structures, defined above.

code
```
typedef struct
  tagFindChirpSegmentVector
{
  UINT4 length;
  FindChirpSegment *data;
} FindChirpSegmentVector;
```

**UINT4 length** Number of `FindChirpSegment` structures in the vector
**DataSegment **data Pointer to an array of `FindChirpSegment` structures.
Structure `FindChirpTemplate`

This structure contains a frequency domain template used as input to the `FindChirpFilter` routine. This may either be a template generated in the frequency domain or the Fourier transform of template generated in the time domain.

```c
typedef struct
  tagFindChirpTemplate
{
    InspiralTemplate tmplt;
    COMPLEX8Vector *data;
    COMPLEX8VectorSequence *PTFQtilde;
    REAL4Array *PTFBinverse;
    REAL4Array *PTFB;
    REAL4 tmpltNorm;
    REAL4 norm;
    REAL8 momentI;
    REAL8 momentJ;
    REAL8 momentK;
    REAL8 rootMomentI;
    REAL8 numFactor;
    REAL8 numFactor1;
    REAL8 numFactor2;
    REAL8 numFactor3;
    REAL8Vector *A1BCVSpin;
    REAL8Vector *A2BCVSpin;
    REAL8Vector *A3BCVSpin;
} FindChirpTemplate;
```

**InspiralTemplate tmplt** The template parameters of this `FindChirpTemplate`. In addition to the mass parameters the following fields of `tmplt` should populated by the template generation functions as these are used by `FindChirpFilterSegment`:

**approximant** Used to check that the findchirp data segment and the template have been created for the same type of waveform.

**tC** The length of the chirp in seconds. Used by the max over chirp event finding algorithm.

**fFinal** The highest frequency component of the chirp. Used to pick the appropriate value of the segment normalization constant $S_k$ for this template.

**COMPLEX8Vector *data** Vector of length $N/2+1$ containing the frequency template data $\tilde{T}_k$. For a template generated in the frequency domain template (FindChirpSP) this should contain

$$\tilde{T}_k = \exp \left[ i\Psi(f_k; M, \eta) \right] \Theta (k - k_{isco}). \quad (23.19)$$

For a template generated in the time domain this should contain the discrete Fourier transform of the cosine phase chirp

$$\tilde{T}_k = \tilde{h}_{ck} = \text{DFT} [h(t)] \quad (23.20)$$

where $h(t)$ is an inspiral waveform generated by the `LALInspiralWave` function if the approximant TaylorT1, TaylorT2, TaylorT3, PadéT1 or EOB. Alternatively $h(t)$ can be generated by the `LALGeneratePPNInspiral` function if the approximant is GeneratePPN. Findchirp always uses second order post-Newtonian templates.

**REAL4 tmpltNorm** The template dependent normalisation constant $\mathcal{T}$. For the stationary phase template FindChirpSP this is

$$\mathcal{T}(M, \mu) = \left[ \left( \frac{2dG M_\odot}{(1 \text{ Mpc})c^2} \right) \left( \frac{5\mu}{96M_\odot} \right)^{\frac{1}{2}} \left( \frac{M}{\pi^2 M_\odot} \right)^{\frac{1}{2}} \left( \frac{GM_\odot}{\Delta t c^2} \right)^{\frac{1}{2}} \right]^2 \quad (23.21)$$
where $d$ is the dynamic range parameter $\text{dynRange}$. For time domain templates generated by $\text{LALInspiralWave}()$ (TaylorT1, TaylorT2, TaylorT3, PadeT1 and EOB) this is

$$T(\mu) = \left[ \frac{4dGM}{c^2} \left( \frac{\mu}{M} \right) \right]^2. \quad (23.22)$$

For time domain templates generated by $\text{LALGeneratePPNInspiral}()$ (GeneratePPN) it is

$$T = \left( \frac{d}{1\,\text{Mpc}} \right)^2. \quad (23.23)$$

**REAL8 momentI** Undocumented BCV normalization constant.

**REAL8 momentJ** Undocumented BCV normalization constant.

**REAL8 momentK** Undocumented BCV normalization constant.

**REAL8 rootMomentI** Undocumented BCV normalization constant.

**REAL8 numFactor** Undocumented BCV normalization constant.

**REAL8 numFactor1** Undocumented BCV normalization constant.

**REAL8 numFactor2** Undocumented BCV normalization constant.

**REAL8 numFactor3** Undocumented BCV normalization constant.

**REAL8 A1BCVSpin** Undocumented spinning BCV template data.

**REAL8 A2BCVSpin** Undocumented spinning BCV template data.

**REAL8 A3BCVSpin** Undocumented spinning BCV template data.
23.3 Header `FindChirp.h`

This header provides core prototypes, structures and functions to filter interferometer data for binary inspiral chirps.

**Synopsis**

```c
#include <lal/FindChirp.h>
```

Each function in findchirp falls into one of four classes:

1. Generate management functions which are independent of the type of filtering implemented. The prototypes for these functions are provided by this header file.

2. Functions for filtering data for time domain and frequency domain templates with an unknown amplitude and phase. These are the functions that implement matched filtering for time domain templates (TaylorT1, TaylorT2, TaylorT2, PadeT1, EOB and GeneratePPN) and matched filtering for post-Newtonian frequency domain templates (FindChirpSP). The main filter function `FindChirpFilterSegment()` is prototyped in this header file. The template generation and data conditioning functions are prototyped in `FindChirpSP.h` and `FindChirpTD.h`. Full documentation of the filtering algorithm used can be found in the documentation of the module `FindChirpFilter.c`.

3. Functions to filter interferometer data for using the frequency domain non-spinning black hole detection template family known as BCV. These functions are prototyped by the header `FindChirpBCV.h` which contains documentation of the algorithms used.

4. Functions to filter interferometer data for using the frequency domain spinning black hole detection template family known as BCVSpin. These functions are prototyped by the header `FindChirpBCVSpin.h` which contains documentation of the algorithms used.

The goal of all the filtering functions is to determine if the (calibrated) output of the interferometer \( s(t) \) contains a gravitational wave \( h(t) \) in the presence of the detector noise \( n(t) \). When the interferometer is operating properly

\[
s(t) = \begin{cases} 
    n(t) + h(t) & \text{signal present}, \\
    n(t) & \text{signal absent}.
\end{cases}
\]  

(23.24)

The detection of signals of known form in noise is a classic problem of signal processing[?] and can be answered by the construction of a *detection statistic* and a test to see if the statistic is above some pre-assigned threshold. The construction of the various detection statistics used for each the three types of search are described in the modules that implement the search.
Error codes

<table>
<thead>
<tr>
<th>name</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>1</td>
<td>&quot;Null pointer&quot;</td>
</tr>
<tr>
<td>NNUL</td>
<td>2</td>
<td>&quot;Non-null pointer&quot;</td>
</tr>
<tr>
<td>ALOC</td>
<td>3</td>
<td>&quot;Memory allocation error&quot;</td>
</tr>
<tr>
<td>NUMZ</td>
<td>5</td>
<td>&quot;Invalid number of points in segment&quot;</td>
</tr>
<tr>
<td>SEGZ</td>
<td>6</td>
<td>&quot;Invalid number of segments&quot;</td>
</tr>
<tr>
<td>CHIZ</td>
<td>7</td>
<td>&quot;Invalid number of chi squared bins&quot;</td>
</tr>
<tr>
<td>DTZO</td>
<td>8</td>
<td>&quot;deltaT is zero or negative&quot;</td>
</tr>
<tr>
<td>TRNC</td>
<td>10</td>
<td>&quot;Duration of inverse spectrum in time domain is negative&quot;</td>
</tr>
<tr>
<td>FLOW</td>
<td>11</td>
<td>&quot;Inverse spectrum low frequency cutoff is negative&quot;</td>
</tr>
<tr>
<td>FREE</td>
<td>12</td>
<td>&quot;Error freeing memory&quot;</td>
</tr>
<tr>
<td>RHT</td>
<td>15</td>
<td>&quot;Rhosq threshold is negative&quot;</td>
</tr>
<tr>
<td>CHIT</td>
<td>16</td>
<td>&quot;Chisq threshold is negative&quot;</td>
</tr>
<tr>
<td>CRUP</td>
<td>17</td>
<td>&quot;Chirp length or invSpecTrunc too long for length of data segment&quot;</td>
</tr>
<tr>
<td>SMSM</td>
<td>18</td>
<td>&quot;Size mismatch between vectors&quot;</td>
</tr>
<tr>
<td>HETR</td>
<td>19</td>
<td>&quot;Attempting to simulate heterodyned GW&quot;</td>
</tr>
<tr>
<td>DFDT</td>
<td>20</td>
<td>&quot;Waveform sampling interval is too large&quot;</td>
</tr>
<tr>
<td>APRX</td>
<td>21</td>
<td>&quot;Incorrect waveform approximant&quot;</td>
</tr>
<tr>
<td>UAPX</td>
<td>22</td>
<td>&quot;Unknown waveform approximant&quot;</td>
</tr>
<tr>
<td>CHTZ</td>
<td>23</td>
<td>&quot;Length of chirp is zero or negative&quot;</td>
</tr>
<tr>
<td>MASS</td>
<td>24</td>
<td>&quot;Invalid mass parameters for template generation&quot;</td>
</tr>
<tr>
<td>WVFN</td>
<td>25</td>
<td>&quot;Unknown injection waveform&quot;</td>
</tr>
<tr>
<td>BCVC</td>
<td>26</td>
<td>&quot;BCVC code: thetav not in [-pi, pi].&quot;</td>
</tr>
<tr>
<td>MAPX</td>
<td>26</td>
<td>&quot;Mismatch in waveform approximant&quot;</td>
</tr>
<tr>
<td>PTFW</td>
<td>27</td>
<td>&quot;Error generating PTF waveform&quot;</td>
</tr>
<tr>
<td>IGEN</td>
<td>28</td>
<td>&quot;Error computing eigenvalues&quot;</td>
</tr>
<tr>
<td>IMRW</td>
<td>29</td>
<td>&quot;Error computing IMR waveform&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants FINDCHIRPH_E<name>, and the status descriptions in FINDCHIRPH_MSGE<name>. The source code with these messages is in FindChirp.h on line 1.180.

Types

Structure **FindChirpInitParams**

This structure provides the essential information for the filter initialisation and memory allocation functions used by findchirp.

```c
typedef struct tagFindChirpInitParams {
    UINT4 numSegments;
    UINT4 numPoints;
    UINT4 ovrlap;
    UINT4 numChisqBins;
    BOOLEAN createRhosqVec;
    BOOLEAN createCVec;
    Approximant approximant;
} FindChirpInitParams;
```

**UINT4 numSegments** The number of data segments in the input **DataSegmentVector** and a the **FindChirpSegmentVector**.
UINT4 numPoints  The number of discrete data points $N$ in each data segment.

UINT4 overlap  The number of sample points by which each data segment overlaps.

UINT4 numChisqBins  The number of bins $p$ used to contruct the $\chi^2$ veto.

BOOLEAN createRhosqVec  Flag that controls whether or not the filter function should store the output of the matched filter, $\rho^2(t)$, as well as the events. Memory is allocated for this vector if the flag is set to 1.

BOOLEAN createCVec  Flag that controls whether or not the filter function should store the complex filter output $x(t) + iy(t)$ needed by the coherent inspiral code. Memory is allocated for this vector if the flag is set to 1.

Approximant approximant  Initialize the findchirp routines to filter with templates of type approximant. Valid approximants are TaylorT1, TaylorT2, TaylorT3, PadeT1, EOB, FindChirpSP, BCV and BCVSpin.

Structure FindChirpDataParams

This structure contains the parameters needed to call the data conditioning functions FindChirpSPData(), FindChirpTDData(), FindChirpBCVData() or FindChirpBCVSpinData(). It should be initialized by FindChirpDataInit() and destroyed by FindChirpDataFinalize().

typedef struct
  tagFindChirpDataParams
  {
    REAL4Vector *ampVec;
    REAL4Vector *ampVecBCV;
    REAL8Vector *ampVecBCVSpin1;
    REAL8Vector *ampVecBCVSpin2;
    RealFFTPlan *fwdPlan;
    RealFFTPlan *invPlan;
    REAL4Vector *wVec;
    COMPLEX8Vector *wtildeVec;
    REAL4Vector *tmpltPowerVec;
    REAL4Vector *tmpltPowerVecBCV;
    REAL4 fLow;
    REAL4 dynRange;
    UINT4 invSpecTrunc;
    Approximant approximant;
  }

FindChirpDataParams;

REAL4Vector *ampVec  A vector containing the frequency domain quantity $(k/N)^{-7/6}$, where $k$ is the frequency series index and $N$ is the number of points in a data segment. NB: for time domain templates, this is set to unity by the function FindChirpTDData().

REAL4Vector *ampVecBCV  A vector containing the frequency domain quantity $(k/N)^{-1/2}$, where $k$ is the frequency series index and $N$ is the number of points in a data segment.

REAL4Vector *ampVecBCVSpin1  Undocumented spinning BCV amplitude vector.

REAL4Vector *ampVecBCVSpin2  Undocumented spinning BCV amplitude vector.

REAL4Vector *fwdPlan  An FFTW plan used to transform the time domain interferometer data $v(t_j)$ into its DFT $\tilde{v}_k$.

REAL4Vector *invPlan  An FFTW plan used to transform the dimensionless frequency domain interferometer strain $\tilde{w}_k$ into the quantity $N\tilde{w}(t_j)$ to allow time domain truncation of the inverse power spectrum.

REAL4Vector *wVec  A vector used as workspace when truncating the inverse power spectrum in the time domain.
COMPLEX8Vector *wtildeVec A vector which on exit from the data conditioning function contains the inverse of the strain one sided power spectral density, after truncation in the time domain, for the last data segment conditioned. Typically all the data segments are conditioned using the same power spectrum, so this quantity is identical for all data segments. It contains:

$$\tilde{w}_k = \frac{1}{S(|f_k|)}.$$  \hspace{1cm} (23.25)

REAL4Vector *tmpltPowerVec A vector which on exit from FindChirpSPData() or from FindChirpBCVData() contains the quantity

$$\text{tmpltPower}[k] = \frac{f^{-7/3}}{S(|f_k|)}$$ \hspace{1cm} (23.26)

REAL4Vector *tmpltPowerVecBCV A vector which on exit from FindChirpBCVData() contains the quantity

$$\text{tmpltPowerBCV}[k] = \frac{f^{-1}}{S(|f_k|)}$$ \hspace{1cm} (23.27)

REAL4 fLow The frequency domain low frequency cutoff $f_{\text{low}}$. All frequency domain data is set to zero below this frequency.

REAL4 dynRange A dynamic range factor $d$ which cancels from the filter output. This allows quantities to be stored in the range of REAL4 rather than REAL8. This must be set to the same value as dynRange in the FindChirpTmpltParams. For LIGO data a value of $d = 2^{69}$ is appropriate.

UINT4 invSpecTrunc The length to which to truncate the inverse power spectral density of the data in the time domain. If set to zero, no truncation is performed.

Approximant approximant Condition the data for templates of type approximant. Valid approximants are TaylorT1, TaylorT2, TaylorT3, PadeT1, EOB, FindChirpSP, BCV and BCVSpin.

Structure FindChirpTmpltParams

This structure contains the parameters for generation of templates by the various template generation functions provided in finchirp.

typedef struct

tagFindChirpTmpltParams
{
    REAL8 deltaT;
    REAL4 fLow;
    REAL4 dynRange;
    REAL4Vector *xfacVec;
    REAL4VectorSequence *PTFQ;
    REAL4Vector *PTFphi;
    REAL4Vector *PTFomega_2_3;
    REAL4VectorSequence *PTFe1;
    REAL4VectorSequence *PTFe2;
    RealFFTPlan *fwdPlan;
    Approximant approximant;
    Order order;
    INT4 reverseChirpBank;
}

FindChirpTmpltParams;

REAL8 deltaT The sampling interval $\Delta t$ of the input data channel.

REAL4 fLow The frequency domain low frequency cutoff $f_{\text{low}}$. All frequency domain data is zero below this frequency.

REAL4 dynRange A dynamic range factor $d$ which cancels from the filter output. This allows quantities to be stored in the range of REAL4 rather than REAL8. This must be set to the same value as dynRange in the FindChirpDataParams. For LIGO data a value of $d = 2^{69}$ is appropriate.
REAL4Vector *xfacVec For frequency domain templates, this is a vector of length $N/2 + 1$ which contains the quantity $k^{-1/3}$. For time domain templates, this is a workspace vector of length $N$ which contains the time domain template generated by the inspiral package, shifted so that the end of the template is at the end of the vector. This vector is Fourier transformed to obtain the quantity findchirp template $\tilde{T}_k$.

REAL4Vector *fwdPlan For time domain templates, an FFTW plan used to transform the time domain data stored in xfacVec into its DFT which is stored in the findchirp template.

Approximant approximant Generate templates of type approximant. Valid approximants are TaylorT1, TaylorT2, TaylorT3, PadeT1, EOB, FindChirpSP, BCV and BCVSpin. For time domain templates the post-Newtonian order is always two. For stationary phase templates, the post-Newtonian order is specified by order.

Order order Specifies the post-Newtonian order of the templates. Valid pN orders are twoPN, twoPoint-FivePN, threePN, threePointFivePN, pseudoFourPN. pseudoFourPN is not the true four PN correction, but may increase the fitting factor between stationary phase and numerical relativity waveforms.

INT4 reverseChirpBank Switches a FindChirpSP template bank to be a reverse chirp template bank if true.

Structure Clustering
This structure contains the possible methods by which to maximize over a chirp in a data segment.

typedef enum {
    noClustering,
    tmplt,
    window
} Clustering;

noClustering The decision to do no clustering of events.

tmplt Cluster over the length of the data segment.

window Cluster over a given number of seconds given by the argument to the flag --cluster-window (required to be less than the length of the data segment).

Structure FindChirpFilterOutputVetoParams
This structure provides the parameters for the filter output veto.

typedef struct tagFindChirpFilterOutputVetoParams {
    REAL4 rsqvetoWindow;
    REAL4 rsqvetoThresh;
    REAL4 rsqvetoTimeThresh;
    REAL4 rsqvetoMaxSNR;
    REAL4 rsqvetoCoeff;
    REAL4 rsqvetoPow;

} FindChirpFilterOutputVetoParams;

REAL4 rsqvetoWindow Width of the $r^2$ veto window in units of seconds.

REAL4 rsqvetoThresh Threshold of the $r^2$ veto test analogous to the $r^2$ threshold employed in the bus and macho inspiral searches.
Structure FindChirpFilterParams

This structure provides the parameters used by the FindChirpFilterSegment() function.

typedef struct
  tagFindChirpFilterParams
  {
    REAL8 deltaT;
    REAL4 clusterWindow;
    REAL4 rhosqThresh;
    REAL4 chisqThresh;
    REAL4 chisqDelta;
    UINT4 maximiseOverChirp;
    UINT4 ignoreIndex;
    Clustering clusterMethod;
    Approximant approximant;
    COMPLEX8Vector *qVec;
    COMPLEX8Vector *qVecBCV;
    COMPLEX8Vector *qVecBCVSpin1;
    COMPLEX8Vector *qVecBCVSpin2;
    COMPLEX8Vector *qtildeVec;
    COMPLEX8Vector *qtildeVecBCV;
    COMPLEX8Vector *qtildeVecBCVSpin1;
    COMPLEX8Vector *qtildeVecBCVSpin2;
    COMPLEX8VectorSequence *PTFqVec;
    COMPLEX8Vector *PTFsnrVec;
    REAL4Array *PTFA;
    REAL4Array *PTFMatrix;
    ComplexFFTPlan *invPlan;
    REAL4TimeSeries *rhosqVec;
    COMPLEX8TimeSeries *cVec;
    REAL4Vector *chisqVec;
    FindChirpChisqParams *chisqParams;
    FindChirpChisqInput *chisqInput;
    FindChirpChisqInput *chisqInputBCV;
    FindChirpFilterOutputVetoParams *filterOutputVetoParams;
  }
  FindChirpFilterParams;

REAL8 deltaT The sampling interval \( \Delta t \).

REAL4 rhosqThresh The signal-to-noise ratio squared threshold \( \rho^2 \). If the matched filter output exceeds this value, that is \( \rho^2(t_j) > \rho^2_0 \), the event processing algorithm is entered and triggers may be generated (subject to addition vetoes such as the \( \chi^2 \) veto). The value of \( \rho^2_0 \) must be greater than or equal to zero.

REAL4 chisqThresh The \( \chi^2 \) veto threshold on. This threshold is described in details in the documentation for the \( \chi^2 \) veto.

REAL4 norm On exit this contains the normalisation constant that relates the quantity \( |q_j|^2 \) with the signal to noise squared, \( \rho^2(t_j) \) by

\[
\rho^2(t_j) = \text{norm} \times |q_j|^2.
\]  

(23.28)

UINT4 maximiseOverChirp If not zero, use the maximise over chirp length algorithm to decide which time \( t_j \) should have an inspiral trigger generated. Otherwise record all points that pass the \( \rho^2 \) and \( \chi^2 \) threshold as triggers (this may generate many triggers).

Approximant approximant Filter the data using templates of type approximant. Valid approximants are TaylorT1, TaylorT2, TaylorT3, PadeT1, EOB, FindChirpSP, BCV and BCVSpin. The value of approximant here must match that in the findchirp data segment and findchirp template used as input.
COMPLEX8Vector *qVec Pointer to vector of length N allocated by FindChirpFilterInit() to store the quantity \( q_j \). The pointer must not be NULL on entry, but the vector may contain garbage which will be overwritten with the value of \( q_j \) for the segment filtered on exit.

COMPLEX8Vector *qVecBCV Pointer to the additional vector required for the BCV templates, allocated by FindChirpFilterInit().

COMPLEX8Vector *qVecBCVSpin1 Pointer to the additional vector required for filtering spinning BCV templates, allocated by FindChirpFilterInit().

COMPLEX8Vector *qVecBCVSpin2 Pointer to the additional vector required for filtering spinning BCV templates, allocated by FindChirpFilterInit().

COMPLEX8Vector *qtildeVec Pointer to vector of length N allocated by FindChirpFilterInit() to store the quantity \( \tilde{q}_k \), given by

\[
\tilde{q}_k = \begin{cases} 
\tilde{F}_k \tilde{T}_k^* & 0 < k < \frac{N}{2} \\
0 & \text{otherwise}
\end{cases}
\]

(23.29)

The pointer must not be NULL on entry, but the vector may contain garbage which will be overwritten with the value of \( \tilde{q}_k \) for the segment filtered on exit.

COMPLEX8Vector *qtildeVecBCV Pointer to the additional vector required for filtering BCV templates, allocated by FindChirpFilterInit().

COMPLEX8Vector *qtildeVecBCVSpin1 Pointer to the additional vector required for filtering spinning BCV templates, allocated by FindChirpFilterInit().

COMPLEX8Vector *qtildeVecBCVSpin2 Pointer to the additional vector required for filtering spinning BCV templates, allocated by FindChirpFilterInit().

ComplexFFTPlan *invPlan Pointer to FFTW plan created by FindChirpFilterInit() to transform the quantity \( \tilde{q}_k \) to \( q_j \) using the inverse DFT. Must not be NULL.

REAL4TimeSeries *rhosqVec Pointer to a time series which contains a vector of length N. If this is not NULL, the filter output \( \rho^2(t_j) \) is stored in the vector.

COMPLEX8Vector *rhosqVec Pointer to a time series which contains a vector of length N. If this is not NULL, the complex filter output \( \rho(t_j) = x(t_j) + iy(t_j) \) is stored in the vector. This quantity can be used by the coherent filtering code.

REAL4Vector *chisqVec Workspace vector of length N used to compute and store \( \chi^2(t_j) \). Must not be NULL if numChisqBins is greater than zero. Contains \( \chi^2(t_j) \) on exit.

FindChirpChisqParams *chisqParams Pointer to parameter structure for the \( \chi^2 \) veto. Must not be NULL if numChisqBins is greater than zero.

FindChirpChisqInput *chisqInput Pointer to input data structure for the \( \chi^2 \) veto. Must not be NULL if numChisqBins is greater than zero.

FindChirpChisqInput *chisqInputBCV Pointer to input data structure for the BCV \( \chi^2 \) veto. Must not be NULL if the approximant is BCV and numChisqBins is greater than zero.

FindChirpFilterOutputVetoParams *filterOutputVetoParams Pointer to the parameter structure for the additional signal based veto function.

Structure FindChirpFilterInput

This structure groups the input data required for the FindChirpFilterSegment() function into a single structure.

typedef struct
tagFindChirpFilterInput
{
    FindChirpTemplate *fcTmplt;
    FindChirpSegment *segment;
}
FindChirpFilterInput;
FindChirpTemplate *fcTmplt  Pointer to the input template in a form that can be used by FindChirpFilterSegment()

FindChirpSegment *segment  Pointer to the input data segment in a form that can be used by FindChirpFilterSegment()

Structure FindChirpSubBankData
This structure contains data needed for the bank veto.

typedef struct
  tagFindChirpBankVetoData
{
  UINT4 length;
  COMPLEX8Vector **qtildeVecArray;
  COMPLEX8Vector **qVecArray;
  FindChirpFilterInput **fcInputArray;
  REAL4Vector *ccMat;
  REAL4Vector *normMat;
  REAL4Vector *spec;
  COMPLEX8Vector *resp;
}
FindChirpBankVetoData;

struct tagFindChirpSubBankData *next  The next structure in the linked list.

typedef struct
  tagFindChirpBankSimParams
{
  Approximant approx;
  Order order;
  REAL4 minMass;
  REAL4 maxMass;
  RandomParams *randParams;
  INT4 maxMatch;
  CHAR *frameName;
  CHAR *frameChan;
}
FindChirpBankSimParams;

Approximant approx  Waveform approximant to use for injection.

Approximant order  Waveform order to use for injection.

REAL4 minMass  Minimum mass of injected signals.

REAL4 maxMass  Maximum mass of injected signals.
23.3.1 Module FindChirpLinkedList.c

It is often convenient to deal with the inspiral templates as a doubly linked list. This module provides memory management functions for creating and destroying linked lists of inspiral template nodes for flat and hierarchical search management.

Prototypes

```c
void LALFindChirpCreateTmpltNode (  
    LALStatus *status,  
    InspiralTemplate *tmplt,  
    InspiralTemplateNode **tmpltNode  
);

void LALFindChirpDestroyTmpltNode (  
    LALStatus *status,  
    InspiralTemplateNode **tmpltNode  
);
```

Description

The function `LALFindChirpCreateTmpltNode()` adds the inspiral template parameter structure pointed to by `tmplt` to the linked list of template nodes `tmpltNode`. On entry `tmpltNode` should be set to memory address of the last node of the current linked list (or NULL if it is a new linked list) and on exit `tmpltNode` is set to the memory address of the last node in the linked list.

The function `LALFindChirpDestroyTmpltNode()` removed the node pointed to by `tmpltNode` from the doubly linked list. On exit `tmpltNode` is set to the address of the previous node in the list for removal of a node in the middle or at the end of the list. If the first node is removed `tmpltNode` is set to the address of the new first node.

Algorithm

None.

Uses

LALCalloc()
LALFree()

Notes
23.3.2 Module FindChirpMemory.c

Memory management functions for creating and destroying input data and workspace memory for findchirp.

Prototypes

```c
void LALInitializeDataSegmentVector (  
    LALStatus *status,  
    DataSegmentVector **dataSegVecPtr,  
    REAL4TimeSeries *chan,  
    REAL4FrequencySeries *spec,  
    COMPLEX8FrequencySeries *resp,  
    FindChirpInitParams *params  
)
```

```c
void LALFinalizeDataSegmentVector (  
    LALStatus *status,  
    DataSegmentVector **vector  
)
```

```c
void LALCreateDataSegmentVector (  
    LALStatus *status,  
    DataSegmentVector **vector,  
    FindChirpInitParams *params  
)
```

```c
void LALDestroyDataSegmentVector (  
    LALStatus *status,  
    DataSegmentVector **vector  
)
```

```c
void LALCreateFindChirpSegmentVector (  
    LALStatus *status,  
    FindChirpSegmentVector **vector,  
    FindChirpInitParams *params  
)
```

```c
void LALDestroyFindChirpSegmentVector (  
    LALStatus *status,  
    FindChirpSegmentVector **vector  
)
```

Description

The function `LALInitializeDataSegmentVector()` creates a vector of `DataSegment` structures of length and dimension specified in the `FindChirpInitParams` structure. The data segments are created using the input data in `chan`, `spec` and `resp`. No storage is allocated for the actual data in the data segments, instead the data segment `chan`, `spec` and `resp` pointers are pointed at the input data appropriately. This means that the input `chan`, `spec` and `resp` time and frequency series must persist for the duration of the filtering process.

The function `LALFinalizeDataSegmentVector()` frees any memory allocated by the function `LALInitializeDataSegmentVector()`. 
The function `LALCreateDataSegmentVector()` creates a vector of `DataSegment` structures of length and dimension specified in the `FindChirpInitParams` structure. **vector** must point to NULL on entry and contains a handle to the address of the created `DataSegmentVector` on exit.

The function `LALDestroyDataSegmentVector()` frees the memory of the `DataSegmentVector` at address **vector**.

The function `LALCreateFindChirpSegmentVector()` creates a vector of `FindChirpSegment` structures of length and dimension specified in the `FindChirpInitParams` structure. **vector** must point to NULL on entry and contains a handle to the address of the created `DataSegmentVector` on exit.

The function `LALDestroyFindChirpSegmentVector()` frees the memory of the `FindChirpSegmentVector` at address **vector**.

**Algorithm**

None.

**Uses**

`LALCalloc()`
`LALCreateVector()`
`LALCCreateVector()`
`LALU4CreateVector()`
`LALFree()`
`LALDestroyVector()`
`LALCDestroyVector()`
`LALU4DestroyVector()`

**Notes**
23.3.3 Module `FindChirpData.c`

This module provides routines to initialize the various data conditioning functions.

Prototypes

```c
void LALFindChirpDataInit (LALStatus *status,
                           FindChirpDataParams **output,
                           FindChirpInitParams *params)

void LALFindChirpDataFinalize (LALStatus *status,
                                FindChirpDataParams **output)
```

Description

Placeholder.

Algorithm

Blah.

Uses

LALCalloc()
LALFree()

Notes

Author: Brown, D. A., BCV-Modifications: Messaritaki E.

$Id: FindChirpData.c,v 1.9 2007/09/29 00:45:47 dfazi Exp $
23.3.4 Module FindChirpTemplate.c

Provides functions to initialize template creation routines.

Prototypes

```c
void LALFindChirpTemplateInit (
   LALStatus *status,
   FindChirpTmpltParams **output,
   FindChirpInitParams *params
);

void LALFindChirpTemplateFinalize (
   LALStatus *status,
   FindChirpTmpltParams **output
);
```

The function `LALFindChirpTemplateInit()` takes as input the address of a structure of type `FindChirpInitParams` containing the correct values to initialize a search. It creates a structure of type `FindChirpTmpltParams` as described above and returns its address.

The function `LALFindChirpTemplateFinalize()` takes as the address of a structure of type `FindChirpTmpltParams` destroys this structure and sets the address to NULL.

Algorithm

Blah.

Uses

- `LALCalloc()`
- `LALFree()`
- `LALCreateVector()`
- `LALDestroyVector()`

Notes
23.3.5 Module FindChirpFilter.c
This module provides the core of the matched filter for binary inspiral chirps.

23.3.6 Matched Filtering Using Post-Newtonian Templates
The gravitational wave strain induced in an interferometer by a binary inspiral may be written as
\[
h(t) = \frac{A(t)}{D} \cos (2\phi(t) - \theta),
\]
where
\[
A(t) = -\frac{2G\mu}{c^4} [\pi GMf(t)]^\frac{3}{2}
\]
and \(D\) is the effective distance, given by
\[
D = \frac{r}{\sqrt{F_+^2(1 + \cos^2 \iota)^2 + F_4^2 4 \cos^2 \iota}}.
\]
The phase angle \(\theta\) is
\[
\tan \theta = \frac{F_4 2 \cos \iota}{F_+ (1 + \cos^2 \iota)}
\]
and \(\phi(t)\) is the phase evolution of the inspiral waveform.

Prototypes

```c
void LALFindChirpFilterSegment (LALStatus *status, SnglInspiralTable **eventList, FindChirpFilterInput *input, FindChirpFilterParams *params)
```

Description
Algorithm
Notes
23.3.7 Module FindChirpFilterOutputVeto.c

Memory management functions for creating and destroying input data and workspace memory for findchirp.

Prototypes

```c
void LALFindChirpFilterOutputVeto(
    LALStatus *status, 
    SnglInspiralTable **eventList, 
    FindChirpFilterInput *input, 
    FindChirpFilterParams *fcParams
)
```

Description

The function `LALFindChirpFilterOutputVeto()` implements a signal based veto, currently it is used primarily for testing. The function itself tests the consistency of the triggers that survive the bns and macho inspiral search pipeline by monitoring the behavior of the $r^2$ time series which is calculated for each segment of data (256 seconds).

Thresholds for Searches

Two thresholds are currently employed in the binary neutron star and primordial black hole inspiral searches: a signal to noise ratio threshold ($\rho^\ast(t_j)$, $\ast$ denoting the threshold), and a threshold on the consistency of the template chirp waveform with the data ($r^{2\ast}$). At a given instant in time, $t_j$. $r^2(t_j)$ is defined as:

$$r^2(t_j) = \frac{\chi^2(t_j)}{p}$$

(23.34)

where:

- $p =$ number of $\chi^2$ bins

The search code calculates $\rho(t_j)$ and $r^2(t_j)$ for a given segment of data and looks for:

$$\rho(t_j) > \rho^\ast(t_j)$$

(23.35)

and

$$r^2(t_j) < r^{2\ast}(t_j) \ast (1 + \frac{\rho^2(t_j)\delta^2}{p})$$

(23.36)

where:

- $\ast =$ threshold used in the search
- $\rho =$ signal to noise ratio
- $\delta =$ mismatch between your data and template waveform
- $p =$ number of $\chi^2$ bins

If both these criteria are met at a given $t_j$, an inspiral "trigger" is recorded.

Algorithm

The algorithm inputs the the vector `chisqVec` (which is actually $r^2$) for the whole data segment and searches a time window (`rsqvetoWindow`) prior to the inferred coalescence time of the trigger up to the trigger time and counts the number of time samples above a given $r^{2\ast\ast}$ threshold (`rsqvetoThresh`) different than the search pipeline employs. Note as well that the threshold we impose does not get multiplied by the factor: $(1 + \rho^2(t_j)\delta^2/p)$. The outputted value from this test is stored in the `rsqveto_duration` field in the `sngl_inspiral` xml table. Future implementation of this function will have it take the calculated value and decide whether or not to store the trigger for future analysis.
Uses

Notes

The same test described here could also be employed for monitoring the behavior of the signal to noise time series, $\rho(t_j)$, about a trigger, therefore the inclusion of $qVec$ and $qNorm$ as input to the function for future work.
23.3.8 Module FindChirpSimulation.c

Provides an interface between code build from findchirp and various simulation packages for injecting chirps into data.

Prototypes

```c
void LALFindChirpInjectSignals (  
    LALStatus   *status,  
    REAL4TimeSeries *chan,  
    SimInspiralTable *events,  
    COMPLEX8FrequencySeries *resp  
)

INT4 XLALFindChirpSetAnalyzeSegment (  
    DataSegmentVector   *dataSegVec,  
    SimInspiralTable    *injections  
)

INT4 XLALFindChirpTagTemplateAndSegment (  
    DataSegmentVector   *dataSegVec,  
    InspiralTemplate    *tmpltHead,  
    SnglInspiralTable   **events,  
    CHAR     *ifo,  
    REAL4    tdFast,  
    UINT4    *analyseThisTmplt  
)

INT4 XLALFindChirpSetFollowUpSegment (  
    DataSegmentVector   *dataSegVec,  
    SnglInspiralTable   **events  
)

void LALFindChirpSetAnalyseTemplate (  
    LALStatus   *status,  
    UINT4      *analyseThisTmplt,  
    REAL4      mmFast,  
    REAL8      deltaF,  
    INT4       sampleRate,  
    FindChirpDataParams *fcDataParams,  
    int        numTmplts,  
    InspiralTemplate    *tmpltHead,  
    int        numInjections,  
    SimInspiralTable   *injections  
)

UINT4 XLALCmprSgmntTmpltFlags (  
    UINT4 numInjections,  
    UINT4 TmpltFlag,  
    UINT4 SgmntFlag  
)
UINT4
XLALFindChirpBankSimInitialize (
    REAL4FrequencySeries *spec,
    COMPLEX8FrequencySeries *resp,
    REAL8 fLow
)

LALFindChirpInjectSignals() injects the signals described in the linked list of SimInspiralTable structures events into the data chan. The response function resp should contain the response function to use when injecting the signals into the data.

Algorithm
None.

Notes
Uses
LALCalloc()
LALFree()

Notes
23.4 Header *FindChirpSP.h*

Provides structures and functions to condition interferometer data and generate binary inspiral chirps using the stationary phase approximation.

Synopsis

`#include <lal/FindChirpSP.h>`

Error codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>1</td>
<td>&quot;Null pointer&quot;</td>
</tr>
<tr>
<td>NNUL</td>
<td>2</td>
<td>&quot;Non-null pointer&quot;</td>
</tr>
<tr>
<td>ALOC</td>
<td>3</td>
<td>&quot;Memory allocation error&quot;</td>
</tr>
<tr>
<td>NUMZ</td>
<td>4</td>
<td>&quot;Invalid number of segments&quot;</td>
</tr>
<tr>
<td>SEGZ</td>
<td>5</td>
<td>&quot;Invalid number of points in segments&quot;</td>
</tr>
<tr>
<td>MISM</td>
<td>6</td>
<td>&quot;Mismatch between number of points in segments&quot;</td>
</tr>
<tr>
<td>DELT</td>
<td>7</td>
<td>&quot;deltaT is zero or negative&quot;</td>
</tr>
<tr>
<td>FLOW</td>
<td>8</td>
<td>&quot;Low frequency cutoff is negative&quot;</td>
</tr>
<tr>
<td>DYNR</td>
<td>9</td>
<td>&quot;Dynamic range scaling is zero or negative&quot;</td>
</tr>
<tr>
<td>ISTN</td>
<td>10</td>
<td>&quot;Truncation of inverse power spectrum is negative&quot;</td>
</tr>
<tr>
<td>DIVZ</td>
<td>11</td>
<td>&quot;Attempting to divide by zero&quot;</td>
</tr>
<tr>
<td>MAPX</td>
<td>12</td>
<td>&quot;Mismatch in waveform approximant&quot;</td>
</tr>
<tr>
<td>UAPX</td>
<td>13</td>
<td>&quot;Unknown approximant&quot;</td>
</tr>
<tr>
<td>ORDR</td>
<td>14</td>
<td>&quot;Invalid post-Newtonian order&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants `FINDCHIRPSPH_E<name>`, and the status descriptions in `FINDCHIRPSPH_MSGE<name>`. The source code with these messages is in `FindChirpSP.h` on line 1.107.

Types

None.
23.4.1 Module `FindChirpSPData.c`

Provides functions to condition the input data from the interferometer to a form that can be used by the `FindChirpFilter()` function.

At the present time this also includes the template independent part of the stationary phase filter.

Prototypes

```c
void LALFindChirpSPData ( 
    LALStatus *status, 
    FindChirpSegmentVector *fcSegVec, 
    DataSegmentVector *dataSegVec, 
    FindChirpDataParams *params 
)
```

`LALFindChirpSPDataInit()` takes as input the address of a structure of type `FindChirpInitParams` containing the correct values to initialize a search. It creates a structure of type `FindChirpSPDataParams` as described above and returns its address.

`LALFindChirpSPDataFinalize()` takes as the address of a structure of type `FindChirpSPDataParams` destroys this structure and sets the address to NULL.

`LALFindChirpSPData()` conditions the interferometer data as described by the algorithm below.

`LALFindChirpBCVData()` conditions the interferometer data as described by the algorithm below.

Algorithm for SP templates

The `LALFindChirpSPData()` function takes as input three quantities

1. An uncalibrated input data channel \( v_j \) (typically LSC-AS_Q).
2. A one sided power spectral density of the input data \( S_v(|f_k|) \).
3. The frequency domain response function of the input data channel \( R(f_k) \) which is used to convert from an uncalibrated input data into strain.

The input parameters also contain a dynamic range scaling parameter `dynRange`. This is used to keep the quantities in the range of \( \text{REAL4} \) and cancels from the filter output. It is typically \( 2^{69} \) for the LIGO channel AS_Q.

The discrete low frequency cutoff is computed from \( f_{\text{Low}} \) by

\[
k_{\text{low}} = \frac{f_{\text{Low}}}{\Delta f}.
\]  

\( k_{\text{low}} \) is set to unity if this quantity is less than one.

**Computation of strain and inverse power spectrum** The uncalibrated input data channel \( v_j \) is Fourier transformed into the frequency domain to obtain \( \tilde{v}_k \). This is then converted into strain \( \tilde{h}_k \) by computing

\[
\tilde{h}_k = \text{dynRange} \times R(f_k)\tilde{v}_k.
\]  

The inverse power spectrum \( S_v^{-1}(|f_k|) \) is computed between the low frequency cutoff \( f_{\text{Low}} \) and the Nyquist frequency. Below \( f_{\text{Low}} \), the inverse power spectrum is set to zero. If the low frequency cutoff is set to 0, then the DC component of the spectrum is set to zero.
Truncation of the Inverse Power Spectrum in the Time Domain  
Recall that the FFT we use to compute the match filter treats the data as being periodic and that we had to ignore part of the filter output that was corrupted due to wraparound of the filter output from the chirp signal.

As well as the chirp, we are also filtering the data against the inverse power spectrum. The chirp has a duration that is typically much less than then length of the data segment. It only corrupts a region that is the length of the chirp at the start of the data segment. However, in the time domain the inverse power spectrum is the same length of the data segment. This will cause the filter output to be corrupted for all times, as it is non-zero over the entire data segment.

To prevent this, we truncate the inverse power spectrum to a specified length in the time domain. This has the effect of smoothing out the high $Q$ features which and restructing the corruption of the filter to the part of the data segment where the power spectrum is non-zero. These regions can then be ignored when searching for chirps in the filter output.

The parameter that controls the duration of the power spectrum in the time domain is $\text{invSpecTrunc}$ and the algorithm used to perform the truncation is as follows:

1. Compute the square root of the inverse power spectrum, $\sqrt{S_v^{-1}(|f_k|)}$.
2. Set the Nyquist, $k = N/2$ and DC $k = 0$ components of this to zero.
3. Inverse FFT to to obtain the time domain inverse PSD of length $N$ points.
4. Zero the spectrum between the points $j = \text{invSpecTrunc}/2$ and $j = N - \text{invSpecTrunc}/2$. This sets the length of the inverse spectrum in the time domain to be $\text{invSpecTrunc}$ points.
5. FFT the time domain quantity back to the frequency domain.
6. Divide by $N$ so that to recover the quantity before the inverse FFT.
7. Square this quantity to recover $S_v^{-1}(|f_k|)$.
8. Set the Nyquist and DC frequencies to zero and zero the inverse power spectrum below $f_{\text{low}}$.

The strain inverse power spectral density is then computed by

$$S_h^{-1}(|f_k|) = \frac{1}{|\text{dynRange} \times R(f_k)|^2} S_v^{-1}(|f_k|).$$  \hfill (23.39)

Output Data  
The quantity $\text{segNorm}$ is computed by

$$\text{segNorm} = \sum_{k=k_{\text{low}}}^{N/2} \frac{k^{-7/6}}{|\text{dynRange} \times R(f_k)|^2 S_v(|f_k|)}. \hfill (23.40)$$

The output data is given by

$$\text{outputData}[k] = \frac{k^{-7/6} \times \text{dynRange} \times R(f_k) \tilde{v}_k}{|\text{dynRange} \times R(f_k)|^2 S_v(|f_k|)} \hfill (23.41)$$

and is stored in the $\text{FindChirpSegmentVector}$ structure. Note the quantity $k^{-7/6}$ which is specific to the stationary phase chirps used.

Calculation of the $\chi^2$ Bins  
If a $\chi^2$ veto is requested, the bin boundaries for the veto are computed at this point. The indices $k$ in the frequency domain that divide the power in the quantity

$$k^{-7/6}$$

into equal intervals are stored in the array $\text{chisqBin}$.

Algorithm for BCV templates

The $\text{LALFindChirpBCVData()}$ function takes as input...
Uses

LALAlloc()
LALFree()
LALCreateVector()
LALDestroyVector()
LALCreateForwardRealFFTPlan()
LALDestroyRealFFTPlan()
LALCCreateVector()
LALCDestroyVector()
LALForwardRealFFT()
LALReverseRealFFT()

Notes

Author: Brown, D. A.

23.4.2 Module FindChirpSPTemplate.c

Provides functions to create stationary phase inspiral templates in a form that can be used by the FindChirpFilter() function.

Prototypes

```c
void LALFindChirpSPTemplate (LALStatus *status,
                             FindChirpTemplate *fcTmplt,
                             InspiralTemplate *tmplt,
                             FindChirpTmpltParams *params)
```

The function LALFindChirpSPTemplate() creates the stationary phase template as described by the algorithm below.

Algorithm

Blah.

Uses

- LALCalloc()
- LALFree()
- LALCreateVector()
- LALDestroyVector()

Notes

Author: Brown, D. A.

$Id: FindChirpSPTemplate.c,v 1.40 2008/02/22 15:40:49 evano Exp $
23.5  Header FindChirpTD.h

Provides structures and functions to condition interferometer data and generate binary inspiral chirps using time domain waveforms.

Synopsis

```
#include <lal/FindChirpTD.h>
```

Error codes

<table>
<thead>
<tr>
<th>name</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>1</td>
<td>&quot;Null pointer&quot;</td>
</tr>
<tr>
<td>NNUL</td>
<td>2</td>
<td>&quot;Non-null pointer&quot;</td>
</tr>
<tr>
<td>ALOC</td>
<td>3</td>
<td>&quot;Memory allocation error&quot;</td>
</tr>
<tr>
<td>NUMZ</td>
<td>4</td>
<td>&quot;Invalid number of segments&quot;</td>
</tr>
<tr>
<td>SEGZ</td>
<td>5</td>
<td>&quot;Invalid number of points in segments&quot;</td>
</tr>
<tr>
<td>MISM</td>
<td>6</td>
<td>&quot;Mismatch between number of points in segments&quot;</td>
</tr>
<tr>
<td>DELT</td>
<td>7</td>
<td>&quot;deltaT is zero or negative&quot;</td>
</tr>
<tr>
<td>FLOW</td>
<td>8</td>
<td>&quot;Low frequency cutoff is negative&quot;</td>
</tr>
<tr>
<td>DYNR</td>
<td>9</td>
<td>&quot;Dynamic range scaling is zero or negative&quot;</td>
</tr>
<tr>
<td>ISTN</td>
<td>10</td>
<td>&quot;Truncation of inverse power spectrum is negative&quot;</td>
</tr>
<tr>
<td>DIVZ</td>
<td>11</td>
<td>&quot;Attempting to divide by zero&quot;</td>
</tr>
<tr>
<td>MAPX</td>
<td>12</td>
<td>&quot;Mismatch in waveform approximant&quot;</td>
</tr>
<tr>
<td>LONG</td>
<td>13</td>
<td>&quot;Time domain template too long&quot;</td>
</tr>
<tr>
<td>EMTY</td>
<td>14</td>
<td>&quot;Could not find end of chirp in xfacVec&quot;</td>
</tr>
<tr>
<td>SMPL</td>
<td>15</td>
<td>&quot;Waveform sampling interval is too large&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants `FINDCHIRPTDH_E<name>`, and the status descriptions in `FINDCHIRPTDH_MSGE<name>`. The source code with these messages is in `FindChirpTD.h` on line 1.108.

Types

None.
23.5.1 Module FindChirpTDData.c

Time domain filtering code.

Prototypes

```c
void LALFindChirpTDData (  
    LALStatus *status,  
    FindChirpSegmentVector *fcSegVec,  
    DataSegmentVector *dataSegVec,  
    FindChirpDataParams *params  
)
```

Description

Algorithm

Uses

Notes
23.5.2 Module FindChirpTDTemplate.c

Provides functions to create time domain inspiral templates in a form that can be used by the FindChirpFilter() function.

Prototypes

```c
void LALFindChirpTDTemplate ( LALStatus *status,
                                FindChirpTemplate *fcTmplt,
                                InspiralTemplate *tmplt,
                                FindChirpTmpltParams *params
                              )

void LALFindChirpTDNormalize( LALStatus *status,
                                FindChirpTemplate *fcTmplt,
                                FindChirpSegment *fcSeg,
                                FindChirpDataParams *params
                              )
```

The function *LALFindChirpTDTemplate()* creates a time domain template template using the inspiral package.

Algorithm

Blah.

Uses

- LALAlloc()
- LALFree()
- LALCreateVector()
- LALDestroyVector()

Notes
23.6 Header FindChirpChisq.h

Provides prototypes and functions to perform a $\chi^2$ veto on binary inspiral chirps using data generated by the FindChirpFilter() function.

Synopsis

```
#include <lal/FindChirpChisq.h>
```

Chisq veto documentation.

Error codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>1</td>
<td>&quot;Null pointer&quot;</td>
</tr>
<tr>
<td>NNUL</td>
<td>2</td>
<td>&quot;Non-null pointer&quot;</td>
</tr>
<tr>
<td>NUMZ</td>
<td>3</td>
<td>&quot;Number of points is zero or negative&quot;</td>
</tr>
<tr>
<td>CHIZ</td>
<td>4</td>
<td>&quot;Number of chisq bins is zero or negative&quot;</td>
</tr>
<tr>
<td>ALOC</td>
<td>5</td>
<td>&quot;Memory allocation error&quot;</td>
</tr>
<tr>
<td>UAPX</td>
<td>6</td>
<td>&quot;Unknown waveform approximant&quot;</td>
</tr>
<tr>
<td>IAPX</td>
<td>7</td>
<td>&quot;Incorrect waveform approximant&quot;</td>
</tr>
<tr>
<td>BINS</td>
<td>8</td>
<td>&quot;Error computing chisq bin boundaries&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants FINDCHIRPCHISQH_E<name>, and the status descriptions in FINDCHIRPCHISQH_MSGE<name>. The source code with these messages is in FindChirpChisq.h on line l.91.

Types

Structure FindChirpChisqInput

```
typedef struct
  tagFindChirpChisqInput
  {
    COMPLEX8Vector *qtildeVec;
    COMPLEX8Vector *qVec;
  }
FindChirpChisqInput;
```

This structure contains the input to the $\chi^2$ veto function. The quantities should be populated by FindChirpFilter() on entry. The fields are:

COMPLEX8Vector *qtildeVec A vector containing the frequency domain quantity $\tilde{q}_k$ as defined in FindChirpFilter().

COMPLEX8Vector *qVec A vector containing the time domain quantity $q_j$ as defined in FindChirpFilter().

Structure FindChirpChisqParams

```
typedef struct
  tagFindChirpChisqParams
  {
    REAL4          norm;
    REAL4          a1;
    REAL4          b1;
    REAL4          b2;
    UINT4Vector    *chisqBinVec;
    UINT4Vector    *chisqBinVecBCV;
    ComplexFFTPlan *plan;
    COMPLEX8Vector *qtildeBinVec;
  }
```
This structure contains the parameters used by the $\chi^2$ veto function `FindChirpChisqVeto()`. It is created and destroyed by the `FindChirpChisqVetoInit()` and `FindChirpChisqVetoFinalize()` functions. The fields are:

- **REAL4 norm** The normalization factor for the SP templates. Equals $4\Delta t/(NsegNorm)$.
- **REAL4 a1** BCV-template normalization parameter.
- **REAL4 b1** BCV-template normalization parameter.
- **REAL4 b2** BCV-template normalization parameter.
- **REAL4 bankMatch** Template bank match...
- **UINT4Vector *chisqBinVec** A vector containing the boundaries of the bins for the chi-squared veto for the stationary phase chirps, or the boundaries of the bins for the first sum of the chi-squared veto for the BCV templates.
- **UINT4Vector *chisqBinVecBCV** A vector containing the boundaries of the bins for the second part of the chi-squared statistic, for the BCV templates.
- **ComplexFFTPlan *plan** The FFTW plan used by the inverse DFT.
- **COMPLEX8Vector *qtildeBinVec** ...
- **COMPLEX8Vector *qtildeBinVecBCV** ...
- **COMPLEX8Vector **qBinVecPtr** Pointer to an array of pointers. Corresponds to $q^{(1)}_l(t_j)$, which is the contribution of the $l$-th frequency bin to the signal-to-noise ratio at the time $t_j$ (up to the appropriate normalization). It is used for both the stationary phase chirps and the BCV templates.
- **COMPLEX8Vector **qBinVecPtrBCV** Pointer to an array of pointers. Corresponds to $q^{(2)}_l(t_j)$, which is the contribution of the $l$-th frequency bin to the signal-to-noise ratio at the time $t_j$ (up to the appropriate normalization). It is used only for the BCV templates.
Module **FindChirpChisqInit.c**

Module to initialize the $\chi^2$ veto for the various templates (SP, BCV, etc.)

**Prototypes**

```c
void LALFindChirpChisqVetoInit (LALStatus *status, FindChirpChisqParams *params, UINT4 numChisqBins, UINT4 numPoints)

void LALFindChirpChisqVetoFinalize(LALStatus *status, FindChirpChisqParams *params, UINT4 numChisqBins)
```

**Description**

The function `LALFindChirpChisqVetoInit()` takes as input the number of bins required to construct the $\chi^2$ veto and the number of points a data segment as a parameter. The pointer `*params` must contain the address of a structure of type `FindChirpChisqParams` for which storage has already been allocated. On exit this structure will be populated with the correct values for execution of the function `LALFindChirpChisqVeto()`. The workspace arrays and the inverse FFTW plan used by the veto will be created.

The function `LALFindChirpChisqVetoFinalize()` takes the address of a structure of type `FindChirpChisqParams` which has been populated by `LALFindChirpChisqVetoInit()` as input. It takes the number of bins required to construct the $\chi^2$ veto and as a parameter. On exit all memory allocated by the `LALFindChirpChisqVetoInit()` will be freed.

**Algorithm**

chisq algorithm here

**Uses**

- `LALCreateReverseComplexFFTPlan()`
- `LALDestroyComplexFFTPlan()`
- `LALCreateVector()`
- `LALDestroyVector()`
- `LALCOMPLEX8VectorFFT()`

**Notes**


$Id: FindChirpChisqInit.c,v 1.8 2007/09/29 00:45:47 dfazi Exp$
23.6.2 Module FindChirpChisq.c

Module to implement the $\chi^2$ veto for the stationary phase chirp.

Prototypes

```c
void LALFindChirpComputeChisqBins(
    LALStatus *status,
    UINT4Vector *chisqBinVec,
    FindChirpSegment *fcSeg,
    UINT4 kmax
);

void LALFindChirpChisqVeto(
    LALStatus *status,
    REAL4Vector *chisqVec,
    FindChirpChisqInput *input,
    FindChirpChisqParams *params
);
```

Description

The function `LALFindChirpChisqVeto()` performs a $\chi^2$ veto on an entire data segment using the algorithm described below. On exit the vector `chisqVec` contains the value $\chi^2(t_j)$ for the data segment.

Algorithm

chisq algorithm here

Uses

- LALCreateReverseComplexFFTPlan()
- LALDestroyComplexFFTPlan()
- LALCreateVector()
- LALDestroyVector()
- LALCOMPLEX8VectorFFT()

Notes

Author: Anderson, W. G., and Brown D. A.

$Id: FindChirpChisq.c,v 1.40 2007/06/08 14:41:44 bema Exp$
23.7 Header FindChirpBCV.h

Provides structures and functions to condition interferometer data and generate binary inspiral chirps using the BCV detection template family.

Synopsis

\[ \text{#include <lal/FindChirpBCV.h>} \]

Binary Black Holes

For the binary black hole inspiral we use the BCV templates.

BCV Templates

The signal-to-noise ratio (SNR) for a signal \( s \) and a template \( h \) is given by

\[ \rho(h) = \frac{<s,h>}{\sqrt{<h,h>}}, \]  

with the inner product \( <s,h> \) being defined as

\[ <s,h> = 2 \int_{-\infty}^{\infty} \tilde{s}(f) \tilde{h}(f) S_h(f) df \]  

and \( S_h(f) \) being the one-sided noise power spectral density. The last equality in Eq. (23.44) holds only if \( \tilde{s}(f) \) and \( \tilde{h}(f) \) are the Fourier-transforms of real time-functions.

The effective frequency-domain template given by Buonanno, Chen and Vallisneri is

\[ \tilde{h}(f) = A(f) e^{i\psi(f)} \]  

where

\[ A(f) = f^{-7/6}(1 - \alpha f^{2/3}) \theta(f_{\text{cut}} - f), \]  

\[ \psi(f) = \phi_0 + 2\pi ft_0 + f^{-5/3}(\psi_0 + \psi_1 f^{1/3} + \psi_2 f^{2/3} + \psi_3 f + \ldots). \]  

In these expressions, \( t_0 \) and \( \phi_0 \) are the time of arrival and the frequency-domain phase offset respectively, and \( \theta \) is the Heaviside step function. For most inspiral templates approximated by the template (23.45), it is sufficient to use the parameters \( \psi_0 \) and \( \psi_{3/2} \) and set all other \( \psi \) coefficients equal to 0. So in the following:

\[ \psi(f) = \phi_0 + 2\pi ft_0 + f^{-5/3}(\psi_0 + \psi_{3/2} f) \]  

\[ = \phi_0 + \psi'(f). \]  

To simplify the equations, the abbreviation

\[ I_k \equiv 4 \int_0^{f_{\text{cut}}} \frac{df}{f^k S_h(f)} \]  

is used in the following.

Notice that in the code, \( \psi_0 \) is psi0 and \( \psi_{3/2} \) is psi3.

Normalized Template

We begin by normalizing the template \( \tilde{h}(f) \). Specifically, it is assumed that the normalized template is

\[ \tilde{h}(f) = N \tilde{h}(f) \]  

where \( N \) is a real number. Then:

\[ <\tilde{h},\tilde{h}> = 1 \Rightarrow 4R \int_0^{\infty} \frac{\hat{h}^* \hat{h}}{S_h(f)} df = 1 \Rightarrow \]  

\[ 4N^2 \int_0^{\infty} \frac{[f^{-7/6}(1 - \alpha f^{2/3})]^{2} \theta(f_{\text{cut}} - f)}{S_h} df = 1 \Rightarrow \]  

\[ N = \sqrt{I_{7/3} - 2\alpha I_{5/3} + \alpha^2 I_1} \]
So the normalized template is
\[
\hat{h}(f) = \frac{1}{\sqrt{I_{7/3} - 2\alpha I_{5/3} + \alpha^2 I_1}} f^{-7/6} (1 - \alpha f^{2/3}) e^{i\phi_0} e^{i\omega f} \theta(f_{\text{cut}} - f), \quad f > 0
\] (23.56)
and \(\hat{h}(f) = \hat{h}^*(-f), \quad f < 0\).

Next we construct an orthonormal basis \(\{\hat{h}_j\}\) for the 4-dimensional linear subspace of templates, with \(\phi_0 \in [0, 2\pi]\) and \(\alpha \in (-\infty, \infty)\) and all other parameters fixed. Specifically, we want the basis vectors to satisfy
\[
<\hat{h}_i, \hat{h}_j> = \delta_{ij}.
\] (23.57)
For that we construct two real functions \(A_1(f)\) and \(A_2(f)\), linear combinations of \(f^{-7/6}\) and \(f^{-1/2}\), which are related to the 4 basis vectors via:
\[
\hat{h}_{1,2}(f) = A_{1,2}(f) e^{i\psi(f)}
\] (23.58)
\[
\hat{h}_{3,4}(f) = A_{1,2}(f) i e^{i\psi(f)}.
\] (23.59)
Then, Eq. (23.57) becomes:
\[
4R \int_0^\infty \frac{A_i(f)A_j(f)}{S_h} df = \delta_{ij}.
\] (23.60)
So we choose:
\[
A_1(f) = a_1 f^{-7/6}
\] (23.61)
\[
A_2(f) = b_1 f^{-7/6} + b_2 f^{-1/2}.
\] (23.62)
Imposing condition (23.60) gives:
\[
4 \int_0^\infty \frac{A_1(f)A_1(f)}{S_h} df = 1 \Rightarrow a_1 = I_{7/3}^{-1/2}
\] (23.63)
\[
4 \int_0^\infty \frac{A_2(f)A_2(f)}{S_h} df = 1 \Rightarrow b_1^2 I_{7/3} + 2 b_1 b_2 I_{5/3} + b_2^2 I_1 = 1
\] (23.64)
\[
4 \int_0^\infty \frac{A_1(f)A_2(f)}{S_h} df = 0 \Rightarrow b_1 = -b_2 I_{5/3}\ I_{7/3}^{-1}
\] (23.65)
Solving Eqs. (23.64) and (23.65) we get
\[
b_1 = \frac{-I_{5/3}}{I_{7/3}} \left( I_1 - \frac{I_{5/3}^2}{I_{7/3}} \right)^{-1/2}
\] (23.66)
\[
b_2 = \left( I_1 - \frac{I_{5/3}^2}{I_{7/3}} \right)^{-1/2}.
\] (23.67)

The next step is to write the normalized template in terms of the 4 basis vectors
\[
\hat{h}(f) = c_1 \hat{h}_1(f) + c_2 \hat{h}_2(f) + c_3 \hat{h}_3(f) + c_4 \hat{h}_4(f)
\Rightarrow
\]
\[
(f^{-7/6} - \alpha f^{-1/2}) e^{i\phi_0} \sqrt{I_{7/3} - 2\alpha I_{5/3} + \alpha^2 I_1} = (c_1 + ic_3) a_1 f^{-7/6} + (c_2 + ic_4) (b_1 f^{-7/6} + b_2 f^{-1/2})
\] (23.69)
and matching the terms gives
\[
c_1 = \cos \phi_0 \cos \omega
\] (23.70)
\[
c_2 = \cos \phi_0 \sin \omega
\] (23.71)
\[
c_3 = \sin \phi_0 \cos \omega
\] (23.72)
\[
c_4 = \sin \phi_0 \sin \omega
\] (23.73)
if the angle \(\omega\) is defined by
\[
\tan \omega = -\frac{a_1 \alpha}{b_2 + b_1 \alpha}.
\] (23.74)
So
\[
\hat{h}(f) = \cos \phi_0 \cos \omega \hat{h}_1(f) + \cos \phi_0 \sin \omega \hat{h}_2(f) + \sin \phi_0 \cos \omega \hat{h}_3(f) + \sin \phi_0 \sin \omega \hat{h}_4(f).
\] (23.75)
Maximization of the SNR

The Fourier-transformed data is $s(f)$. The SNR is

$$\rho = < s, \hat{h}(f) > = \cos \phi_0 \cos \omega K_1 + \cos \phi_0 \sin \omega K_2 + \sin \phi_0 \cos \omega K_3 + \sin \phi_0 \sin \omega K_4$$

where the 4 integrals are defined by

$$K_1 = \langle s, \hat{h}_1 \rangle = \Re \int_0^{f_{cut}} \frac{4s^* a_1 f^{-7/6} e^{i \psi'}}{S_h} df$$

$$K_2 = \langle s, \hat{h}_2 \rangle = \Re \int_0^{f_{cut}} \frac{4s^* (b_1 f^{-7/6} + b_2 f^{-1/2}) e^{i \psi'}}{S_h} df$$

$$K_3 = \langle s, \hat{h}_3 \rangle = \Re \int_0^{f_{cut}} \frac{4s^* a_1 f^{-7/6} e^{i \psi'}}{S_h} df$$

$$K_4 = \langle s, \hat{h}_4 \rangle = \Re \int_0^{f_{cut}} \frac{4s^* (b_1 f^{-7/6} + b_2 f^{-1/2}) e^{i \psi'}}{S_h} df$$

Now set

$$A = \omega - \phi_0$$

$$B = \omega + \phi_0$$

so that the expression for the SNR becomes

$$\rho = \frac{1}{2} K_1 [\cos(\omega + \phi_0) + \cos(\omega - \phi_0)] + \frac{1}{2} K_2 [\sin(\omega + \phi_0) + \sin(\omega - \phi_0)] + \frac{1}{2} K_3 [\sin(\omega + \phi_0) - \sin(\omega - \phi_0)] + \frac{1}{2} K_4 [\cos(\omega - \phi_0) - \cos(\omega + \phi_0)] \Rightarrow$$

$$2 \rho = (K_1 + K_4) \cos A + (K_2 - K_3) \sin A + (K_1 - K_4) \cos B + (K_2 + K_3) \sin B.$$  

To maximize with respect to $A$ we take the first derivative

$$\frac{\partial (2\rho)}{\partial A} = -(K_1 + K_4) \sin A + (K_2 - K_3) \cos A$$

and set that equal to 0, which gives

$$\frac{\partial (2\rho)}{\partial A} \bigg|_{A_0} = 0 \Rightarrow \tan A_0 = \frac{K_2 - K_3}{K_1 + K_4}.$$  

Then the sine and cosine of $A_0$ can be found:

$$\sin A_0 = \pm \frac{\tan A_0}{\sqrt{1 + \tan^2 A_0}} = \pm \frac{K_2 - K_3}{\sqrt{(K_1 + K_4)^2 + (K_2 - K_3)^2}},$$

$$\cos A_0 = \pm \frac{1}{\sqrt{1 + \tan^2 A_0}} = \pm \frac{K_1 + K_4}{\sqrt{(K_1 + K_4)^2 + (K_2 - K_3)^2}}.$$  

Notice that for Eq. (23.88) to be satisfied, the same sign must be kept in Eqs (23.89) and (23.90). To find the values that correspond to the maximum, we take the second derivative of $\rho$ with respect to $A$:

$$\frac{\partial^2 (2\rho)}{\partial A^2} \bigg|_{A_0} < 0 \Rightarrow \left[ -(K_1 + K_4) \cos A - (K_2 - K_3) \sin A \right] \bigg|_{A_0} < 0$$

which is satisfied if the $+$ sign is considered in Eqs (23.89) and (23.90).

To maximize with respect to $B$ we take the first derivative

$$\frac{\partial (2\rho)}{\partial B} = -(K_1 - K_4) \sin B + (K_2 + K_3) \cos B$$

and set that equal to 0, which gives

$$\frac{\partial (2\rho)}{\partial B} \bigg|_{B_0} = 0 \Rightarrow \tan B_0 = \frac{K_2 + K_3}{K_1 - K_4}.$$
Then the sine and cosine of $B_0$ can be found:
\[
\sin B_0 = \pm \frac{\tan B_0}{\sqrt{1 + \tan^2 B_0}} = \pm \frac{K_2 + K_4}{\sqrt{(K_1 - K_4)^2 + (K_2 + K_3)^2}}, \quad (23.94)
\]
\[
\cos B_0 = \pm \frac{1}{\sqrt{1 + \tan^2 B_0}} = \pm \frac{K_1 - K_4}{\sqrt{(K_1 - K_4)^2 + (K_2 + K_3)^2}}. \quad (23.95)
\]

Again, the same sign must be kept in Eqs (23.94) and (23.95). To find the values that correspond to the maximum, we take the second derivative of $\rho$ with respect to $B$:
\[
\frac{\partial^2(2\rho)}{\partial B^2} \bigg|_{B_0} < 0 \Rightarrow \left[ -(K_1 - K_4) \cos B - (K_2 + K_3) \sin B \right]_{B_0} < 0 \quad (23.96)
\]
which is satisfied if the + sign is considered in Eqs (23.94) and (23.95).

Substituting the expressions for the sines and cosines of $B_0$ into Eq. (23.86), the maximum SNR is:
\[
\rho_{\text{max}} = \frac{1}{2} \sqrt{(K_1 + K_4)^2 + (K_2 - K_3)^2} + \frac{1}{2} \sqrt{(K_1 - K_4)^2 + (K_2 + K_3)^2}, \quad (23.97)
\]
\[
2\rho_{\text{max}} = \sqrt{K_1^2 + K_2^2 + K_3^2 + K_4^2} + 2(K_1K_4 - K_2K_3) + \sqrt{K_1^2 + K_2^2 + K_3^2 + K_4^2 - 2(K_1K_4 - K_2K_3)^2} \quad (23.98)
\]

To achieve a simpler form for the SNR, we can use Eqs (23.78)-(23.81) to combine the integrals $K_1$, $K_2$, $K_3$ and $K_4$. Specifically:
\[
K_1^2 + K_2^2 + K_3^2 + K_4^2 = \left| \int_0^{f_{\text{cut}}} \frac{4s^*a_1f^{-7/6}e^{i\psi}}{s_h} df \right|^2 + \left| \int_0^{f_{\text{cut}}} \frac{4s^*(b_1f^{-7/6} + b_2f^{-1/2})e^{i\psi}}{s_h} df \right|^2 \quad (23.99)
\]
and
\[
2(K_1K_4 - K_2K_3) = 2\sqrt{3} \left\{ \int_0^{f_{\text{cut}}} \frac{4s^*a_1f^{-7/6}e^{i\psi}}{s_h} df \left( \int_0^{f_{\text{cut}}} \frac{4s^*(b_1f^{-7/6} + b_2f^{-1/2})e^{i\psi}}{s_h} df \right) \right\}. \quad (23.100)
\]

The $\chi^2$-veto

If we are working with $p$ bins the maximum SNR for the template must be divided into $p$ equal parts. In this case, since we have two different amplitude-parts of the template, we have to calculate two sets of bin boundaries. For the first set, the quantity
\[
\int_0^{f_{\text{cut}}} \frac{4a_1^2f^{-7/3}}{s_h(f)} df
\]
must be divided into $p$ equal pieces. For the second set, the quantity
\[
\int_0^{f_{\text{cut}}} \frac{4[b_1f^{-7/6} + b_2f^{-1/2}]^2}{s_h(f)} df
\]
must be divided into $p$ equal pieces.

To check if the total SNR is smoothly distributed over the bins, take:
\[
\chi^2 = \sum_{i=1}^{p} \left[ \int_{f_i}^{f_{i+1}} \frac{4a_1f^{-7/6}s^*e^{i\psi'}}{s_h(f)} df \right]^2 - \frac{1}{p} \int_0^{f_{\text{cut}}} \frac{4a_1f^{-7/6}s^*e^{i\psi'}}{s_h(f)} df \quad (23.101)
\]
\[
+ \sum_{i=1}^{p} \left[ \int_{f_i}^{f_{i+1}} \frac{4[b_1f^{-7/6} + b_2f^{-1/2}]s^*e^{i\psi'}}{s_h(f)} df \right]^2 - \frac{1}{p} \int_0^{\infty} \frac{4[b_1f^{-7/6} + b_2f^{-1/2}]s^*e^{i\psi'}}{s_h(f)} df \quad (23.102)
\]

Quantities calculated in the code for the case of BCV templates

1. The template normalization squared (in LALFindChirpBCVTemplate):
\[
tmpltnorm = d^2 \left( \frac{\mu}{96M_\odot} \right) \left( \frac{M}{\pi^2M_\odot} \right)^{2/3} T_\odot^{-1/3} \left( \frac{2T_\odot c}{1Mpc} \right)^2. \quad (23.104)
\]
2. The exponential $e^{i\psi'}$ (in LALFindChirpBCVTemplate).

3. The BCV Moments (in LALFindChirpBCVData):

$$I_{73} = \sum_{k=0}^{N/2} \frac{4k^{-7/3}}{|dR|^{2} S_v(|f_k|)}$$  \hspace{1cm} (23.105)

$$I_{53} = \sum_{k=0}^{N/2} \frac{4k^{-5/3}}{|dR|^{2} S_v(|f_k|)}$$  \hspace{1cm} (23.106)

$$I_{1} = \sum_{k=0}^{N/2} \frac{4k^{-1}}{|dR|^{2} S_v(|f_k|)}.$$  \hspace{1cm} (23.107)

These quantities should be multiplied by $(\Delta t/N) (b/c$ of the FT) and by $(\text{tmplNorm}) (b/c$ of the template), but that is taken care of in the calculation of the SNR.

4. The BCV normalization factors (in LALFindChirpBCVData):

$$a_1 = (I_{73})^{-1/2}$$  \hspace{1cm} (23.109)

$$b_2 = \left( I_{1} - \frac{I_{53}^2}{I_{73}} \right)^{-1/2}$$  \hspace{1cm} (23.110)

$$b_1 = -\frac{I_{53}}{I_{73}} b_2.$$  \hspace{1cm} (23.111)

Again, these should be multiplied by $[(\Delta t/N)^{-1/2}(\text{tmplNorm})^{-1/2}]$ but that is taken care of later.

5. The two FTs required for the calculation of the SNR (in LALFindChirpBCVFilter):

$$q_j = \sum_{k=0}^{N/2} e^{2\pi ijk/N} \frac{(dR_k^*) a_1 k^{-7/6} e^{i\psi'}}{|dR|^{2} S_v(|f_k|)}$$  \hspace{1cm} (23.113)

$$q_{BCV}^j = \sum_{k=0}^{N/2} e^{2\pi ijk/N} \frac{(dR_k^*) (b_1 k^{-7/6} + b_2 k^{-1/2}) e^{i\psi'}}{|dR|^{2} S_v(|f_k|)}$$  \hspace{1cm} (23.114)

up to the appropriate normalization factors, namely $(\text{tmplNorm})^{1/2}(\Delta t/N)$

6. The SNR (in LALFindChirpBCVFilter):

$$\rho^2(t_j) = \left( \frac{\Delta t}{N} \right) \left\{ \frac{1}{2} \sqrt{|q_j|^2 + |q_{BCV}^j|^2} + 2 \Im(q_j q_{BCV}^j) + \frac{1}{2} \sqrt{|q_j|^2 + |q_{BCV}^j|^2 - 2 \Im(q_j q_{BCV}^j)} \right\}.$$  \hspace{1cm} (23.116)
Error codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>1</td>
<td>&quot;Null pointer&quot;</td>
</tr>
<tr>
<td>NNUL</td>
<td>2</td>
<td>&quot;Non-null pointer&quot;</td>
</tr>
<tr>
<td>ALOC</td>
<td>3</td>
<td>&quot;Memory allocation error&quot;</td>
</tr>
<tr>
<td>NUMZ</td>
<td>4</td>
<td>&quot;Invalid number of segments&quot;</td>
</tr>
<tr>
<td>SEGZ</td>
<td>5</td>
<td>&quot;Invalid number of points in segments&quot;</td>
</tr>
<tr>
<td>MISM</td>
<td>6</td>
<td>&quot;Mismatch between number of points in segments&quot;</td>
</tr>
<tr>
<td>DELT</td>
<td>7</td>
<td>&quot;deltaT is zero or negative&quot;</td>
</tr>
<tr>
<td>FLOW</td>
<td>8</td>
<td>&quot;Low frequency cutoff is negative&quot;</td>
</tr>
<tr>
<td>DYNR</td>
<td>9</td>
<td>&quot;Dynamic range scaling is zero or negative&quot;</td>
</tr>
<tr>
<td>ISTN</td>
<td>10</td>
<td>&quot;Truncation of inverse power spectrum is negative&quot;</td>
</tr>
<tr>
<td>DIVZ</td>
<td>11</td>
<td>&quot;Attempting to divide by zero&quot;</td>
</tr>
<tr>
<td>MAPX</td>
<td>12</td>
<td>&quot;Mismatch in waveform approximant&quot;</td>
</tr>
<tr>
<td>UAPX</td>
<td>13</td>
<td>&quot;Unknown approximant&quot;</td>
</tr>
<tr>
<td>ZNRM</td>
<td>14</td>
<td>&quot;No non-zero value assigned to one of a1, b1, b2&quot;</td>
</tr>
<tr>
<td>QLEN</td>
<td>15</td>
<td>&quot;params-&gt;qVec-&gt;length not equal to params-&gt;qVecBCV-&gt;length&quot;</td>
</tr>
<tr>
<td>CLUW</td>
<td>16</td>
<td>&quot;Unacceptable max-over-chirp clustering method for BCV&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants FINDCHIRPBVCVH_<name>, and the status descriptions in FINDCHIRPBVCVH_MSGE_<name>. The source code with these messages is in FindChirpBCV.h on line 1.112.

Types

None.
23.7.1 Module **FindChirpBCVData.c**

Provides functions to condition the input data from the interferometer to a form that can be used by the `FindChirpBCVFilter()` function.

At the present time this also includes the template independent part of the BCV filter.

**Prototypes**

```c
void LALFindChirpBCVData (LALStatus *status,
             FindChirpSegmentVector *fcSegVec,
             DataSegmentVector *dataSegVec,
             FindChirpDataParams *params)
```

**Description**

Placeholder.

**Algorithm**

Blah.

**Uses**

`LALCalloc()`

`LALFree()`

**Notes**

Author: Brown, D. A. and Messaritaki E.

$Id: FindChirpBCVData.c,v 1.12 2007/06/08 14:41:44 bema Exp $
23.7.2 Module FindChirpBCVTemplate.c

Provides functions to create BCV detection templates in a form that can be used by the FindChirpBCVFilter() function.

Prototypes

```c
void LALFindChirpBCVTemplate (
    LALStatus *status,
    FindChirpTemplate *fcTmplt,
    InspiralTemplate *tmplt,
    FindChirpTmpltParams *params
);
```

The function `LALFindChirpBCVTemplate()` creates the BCV template as described by the algorithm below.

Algorithm

Blah.

Uses

- LALAlloc()
- LALFree()
- LALCreateVector()
- LALDestroyVector()

Notes
23.7.3 Module FindChirpBCVChisq.c

Module to implement the $\chi^2$ veto for the BCV templates.

Prototypes

```c
void LALFindChirpBCVChisqVeto (
    LALStatus *status,
    REAL4Vector *chisqVec,
    FindChirpChisqInput *input,
    FindChirpChisqInput *inputBCV,
    FindChirpChisqParams *params
);
```

Description

The function `LALFindChirpBCVChisqVeto()` performs a $\chi^2$ veto on an entire data segment using the corresponding algorithm for the BCV templates, described below. On exit the vector `chisqVec` contains the value $\chi^2(t_j)$ for the data segment.

Algorithm

chisq algorithm here

Uses

- `LALCreateReverseComplexFFTPlan()`
- `LALDestroyComplexFFTPlan()`
- `LALCreateVector()`
- `LALDestroyVector()`
- `LALCOMPLEX8VectorFFT()`

Notes
23.7.4 Module FindChirpBCVFilter.c

This module provides the core of the matched filter for binary inspiral chirps for BCV templates.

Prototypes

```c
void LALFindChirpBCVFilterSegment (
    LALStatus *status,
    SnglInspiralTable **eventList,
    FindChirpFilterInput *input,
    FindChirpFilterParams *params
)
```

Description

Algorithm

Blah.

Uses

LALCalloc()
LALFree()

Notes

Author: Brown D. A. and Messaritaki E.

$Id: FindChirpBCVFilter.c,v 1.21 2007/06/08 14:41:44 bema Exp $
23.8 Header FindChirpBCVSpin.h

Provides structures and functions to condition interferometer data and generate binary inspiral chirps using the spinning BCV detection template family.

Synopsis

```
#include <lal/FindChirpBCVSpin.h>
```

Error codes

<table>
<thead>
<tr>
<th>Name</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>1</td>
<td>&quot;Null pointer&quot;</td>
</tr>
<tr>
<td>NNUL</td>
<td>2</td>
<td>&quot;Non-null pointer&quot;</td>
</tr>
<tr>
<td>ALOC</td>
<td>3</td>
<td>&quot;Memory allocation error&quot;</td>
</tr>
<tr>
<td>NUMZ</td>
<td>4</td>
<td>&quot;Invalid number of segments&quot;</td>
</tr>
<tr>
<td>SEGZ</td>
<td>5</td>
<td>&quot;Invalid number of points in segments&quot;</td>
</tr>
<tr>
<td>MISM</td>
<td>6</td>
<td>&quot;Mismatch between number of points in segments&quot;</td>
</tr>
<tr>
<td>DELT</td>
<td>7</td>
<td>&quot;deltaT is zero or negative&quot;</td>
</tr>
<tr>
<td>FLOW</td>
<td>8</td>
<td>&quot;Low frequency cutoff is negative&quot;</td>
</tr>
<tr>
<td>DYNR</td>
<td>9</td>
<td>&quot;Dynamic range scaling is zero or negative&quot;</td>
</tr>
<tr>
<td>ISTN</td>
<td>10</td>
<td>&quot;Truncation of inverse power spectrum is negative&quot;</td>
</tr>
<tr>
<td>DIVZ</td>
<td>11</td>
<td>&quot;Attempting to divide by zero&quot;</td>
</tr>
<tr>
<td>MAPX</td>
<td>12</td>
<td>&quot;Mismatch in waveform approximant&quot;</td>
</tr>
<tr>
<td>UAPX</td>
<td>13</td>
<td>&quot;Unknown approximant&quot;</td>
</tr>
<tr>
<td>CLUW</td>
<td>14</td>
<td>&quot;Unacceptable max-over-chirp clustering method for BCVSpin&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants `FINDCHIRPBCVSPINH_E<name>`, and the status descriptions in `FINDCHIRPBCVSPINH_MSGE<name>`. The source code with these messages is in `FindChirpBCVSpin.h` on line 1.109.

Types

None.
23.8.1 Module FindChirpBCVSpinData.c

Provides functions to condition data prior to filtering with spinning BCV detection templates.

Prototypes

```c
void
LALFindChirpBCVData(  
    LALStatus *status,    
    FindChirpSegmentVector *fcSegVec,  
    DataSegmentVector *dataSegVec,    
    FindChirpDataParams *params
)
```

The function `LALFindChirpBCVSpinData()` conditions the data as described by the algorithm below.

Algorithm

Blah.

Uses

- `LALCalloc()`
- `LALFree()`
- `LALCreateVector()`
- `LALDestroyVector()`

Notes

Author: Brown, D. A., Spinning BCV-Modifications: Jones, G.

$Id: FindChirpBCVSpinData.c,v 1.26 2007/06/08 14:41:44 bema Exp$
23.8.2 Module FindChirpBCVSpinTemplate.c

Provides functions to create spinning BCV detection templates in a form that can be used by the FindChirpBCVSpinFilter() function.

Prototypes

```c
void LALFindChirpBCVSpinTemplate (  
  LALStatus *status,  
  FindChirpTemplate *fcTmplt,  
  InspiralTemplate *tmplt,  
  FindChirpTmpltParams *params,  
  FindChirpDataParams *fcDataParams  
);
```

The function LALFindChirpBCVSpinTemplate() creates the spinning BCV template as described by the algorithm below.

Algorithm

This code calculates a number of quantities required by the LALFindChirpBCVSpinFilterSegment() function. To improve efficiency we calculate every template dependent quantity that does not require detector data in this function. Since the template bank does not provide values for $f_{\text{final}}$ we calculate it here. For every combination of $\psi_0$ and $\psi_3$ values provided by the template bank we find the frequency at the last stable orbit using $f_{\text{final}} = \frac{-16\pi\psi_0}{(\psi_3)^{3/2}}$ where $r_{LSO} = 6M_\odot$ is the separation of the binaries components.

We then calculate the complex phase of the template $\psi_{NM}(f) = \psi_{\text{initial}} + f^{-5/3}(\psi_0 + f\psi_3)$ between $f_{\text{low}}$ and $f_{\text{final}}$. Next we calculate 5 moments which are required to construct the template:

\[
I = 4 \int_0^\infty f^{-7/3} \frac{df}{S_n(f)} \\
J = 4 \int_0^\infty f^{-7/3} \cos(\beta f^{-2/3}) \frac{df}{S_n(f)} \\
K = 4 \int_0^\infty f^{-7/3} \sin(\beta f^{-2/3}) \frac{df}{S_n(f)} \\
L = 2 \int_0^\infty f^{-7/3} \sin(2\beta f^{-2/3}) \frac{df}{S_n(f)} \\
M = 2 \int_0^\infty f^{-7/3} \cos(2\beta f^{-2/3}) \frac{df}{S_n(f)}
\]  

(23.117)

In practice we integrate between our lowest non-zero frequency sample point $k=1$ (a division-by-zero error would occur at 0Hz) and the Nyquist frequency $k=\text{numPoints}/2$. From these moments we then find the orthonormalised amplitude vectors:

\[
\hat{A}_1(f) = \frac{f^{-7/6}}{I^{1/2}} \\
\hat{A}_2(f) = \frac{f^{-7/6} \left[ \cos(\beta f^{-2/3}) - \frac{J}{I} \right]^{1/2}}{\left[ IM + I^2 - J^2 \right]^{1/2}} \\
\hat{A}_3(f) = \frac{f^{-7/6} \left[ \sin(\beta f^{-2/3}) - \frac{K}{I} - \frac{(I^2 - J^2)K}{IM + I^2 - J^2} \left[ \cos(\beta f^{-2/3}) - \frac{J}{I} \right]^{1/2} \right]}{\left[ I^2 - IM - K^2 - \frac{(I^2 - J^2)K}{IM + I^2 - J^2} \right]^{1/2}}
\]

(23.118)

where $\beta$ is provided by the template bank code and the $f^{-7/6}$ and $f^{-2/3}$ vectors were calculated previously in LALFindChirpDataInit(). To avoid division-by-zero errors we explicitly set $\hat{A}_2(f) = \hat{A}_3(f) = 0$ when $\beta = 0$. 

Uses

LALAlloc()
LALFree()
LALCreateVector()
LALDestroyVector()

Notes
23.8.3 Module FindChirpBCVSpinFilter.c

Provides functions to filter data for spinning BCV templates.

Prototypes

```c
void
LALFindChirpBCVSpinFilterSegment (  
  const LALStatus *status,  
  const SnglInspiralTable **eventList,  
  const FindChirpFilterInput *input,  
  const FindChirpFilterParams *params,  
  const FindChirpDataParams *fcDataParams  
)
```

The function `LALFindChirpBCVSpinFilterSegment()` filters data for spinning BCV templates as described by the algorithm below.

Algorithm

Using the quantities calculated in `LALFindChirpBCVSpinTemplate()` we construct a template and filter our data producing a clustered signal-to-noise ratio time series. We filter our data in 256 second data segments. We first calculate the following functions in the frequency domain:

\[
q_{\tilde{1}}(f) = \hat{A}_1(f)e^{i\psi_{NM}(f)}s^*(f) / S_h(f)  
\]

\[
q_{\tilde{2}}(f) = \hat{A}_2(f)e^{i\psi_{NM}(f)}s^*(f) / S_h(f)  
\]

\[
q_{\tilde{3}}(f) = \hat{A}_3(f)e^{i\psi_{NM}(f)}s^*(f) / S_h(f)  
\]

where \( \hat{A}_1(f) \), \( \hat{A}_2(f) \) and \( \hat{A}_3(f) \) are the orthonormal amplitude functions and \( \psi_{NM}(f) \) is the non-modulational phase of our template. These quantities were calculated in `LALFindChirpBCVSpinTemplate()`.

\( s^* \) is the complex conjugate of our (detector) data in the frequency domain and \( S_h(f) \) is our estimate of the power spectral density of the detector data estimated over a 2048 second “blocks”. Using inverse FFTs we construct the complex time domain quantities \( q \), \( q_{BCVSpin1} \) and \( q_{BCVSpin2} \). We then calculate signal-to-noise ratio as

\[
\rho(t)^2 = q.re^2 + q.im^2 + q_{BCVSpin1}.re^2 + q_{BCVSpin1}.im^2 + q_{BCVSpin2}.re^2 + q_{BCVSpin2}.im^2.
\]

We then look for values of \( \rho(t) \) above our threshold - note that the \( \beta = 0 \) threshold is currently hardcoded. We do not calculate signal-to-noise ratio for the 64 second stretch at the beginning and end of each data segment to avoid edge-effects. These times are picked up by overlapping our 256 second data segments. For times for which signal-to-noise ratio is calculated we have the option of clustering our output using the --cluster-method window option in `lalapps_inspiral` with an appropriate choice of cluster length. For events that pass the signal-to-noise ratio threshold and survive clustering we store the template parameters \( \psi_0 \), \( \psi_3 \), \( \beta \) and \( f_{final} \) as well as 6 \( \alpha \) values which encode the relative contribution of the \( q \), \( q_{BCVSpin1} \) and \( q_{BCVSpin2} \) functions to the overall signal-to-noise ratio. These are simply calculated as

\[
\alpha_1 = q.re / \rho  
\]

\[
\alpha_2 = q_{BCVSpin1}.re / \rho  
\]

\[
\alpha_3 = q_{BCVSpin2}.re / \rho  
\]

\[
\alpha_4 = q.im / \rho  
\]

\[
\alpha_5 = q_{BCVSpin1}.im / \rho  
\]

\[
\alpha_6 = q_{BCVSpin2}.im / \rho.
\]

These obey \( \sum_{i=1}^{6} \alpha_i = 1 \) and might prove useful in future signal based veto studies.
Uses

LALCalloc()
LALFree()
LALCreateVector()
LALDestroyVector()

Notes
Chapter 24

Package inspiral

This is a module which generates inspiral waveforms. The type of waveform generated is determined by specifying the value of the enums Approximant and Order. All Approximants generate the restricted post-Newtonian (PN) waveform given by (we use units in which \(c = G = 1\)):

\[
h(t) = Av^2 \cos(\phi(t)),
\]

where the coefficient \(A\) is set equal to 1 for non-spinning binaries and to the appropriate modulation function for spinning binaries (see below), \(v\) is the PN expansion parameter, and \(\phi(t)\) is the appropriately defined phasing formula which is either an explicit or an implicit function of time. In what follows we summarize the basic formulas used in generating the waveforms. As we shall see there are a number of ways in which these waveforms can be generated. In the inspiral package these are controlled by two enums: The first enum, called Order, controls the PN order to which the various quantities are defined and the second enum, called Approximant, controls the approximant used in generating the waveforms.

### 24.1 Taylor and Pade approximants

Theoretical calculations have given us PN expansions (i.e. a series in terms of ascending powers of \(v\)) of an energy function \(E(x = v^2)\) and a gravitational wave (GW) luminosity function \(F(v)\):

\[
E(x) = E_N \sum_n E_n x^n, \quad F(v) = F_N \sum_j F_j v^j.
\]

One can use kinematical equations

\[
dt = \left(\frac{dt}{dE}\right) \left(\frac{dE}{dv}\right) dv,
\]

\[
d\phi/dt = 2\pi F,
\]

and the energy balance equation relating the luminosity in gravitational waves to the rate of change of binding energy \(F = -dE/dt\), to obtain a phasing formula [1]:

\[
t(v) = t_{\text{ref}} + m \int_v^{v_{\text{ref}}} \frac{E'(v)}{F(v)} dv,
\]

\[
\phi(v) = \phi_{\text{ref}} + 2 \int_v^{v_{\text{ref}}} v^3 \frac{E'(v)}{F(v)} dv,
\]

where \(E'(v) = dE/dv, v = (\pi m F)^{1/3}\) is an invariantly defined velocity, \(F\) is the instantaneous GW frequency, and \(m\) is the total mass of the binary. There are basically three ways of solving the problem:

1. Leave \(E'(v)/F(v)\) as it is and integrate the equations numerically. Using standard PN expansions for the energy and flux functions one generates the Approximant called TaylorT1. If instead one uses the P-approximant for the energy and flux functions [1, 4] then one generates the approximant called PadeT1. In reality, it is computationally cheaper to use two ordinary differential equations instead of the integrals. These are:

\[
\frac{dv}{dt} = -\frac{F(v)}{m E'(v)}, \quad \frac{d\phi(t)}{dt} = \frac{2v^3}{m}.
\]

These are implemented in modules LALInspiralWave1.
2. Re-expand \( E'(v)/F(v) \) in a Taylor expansion in which case the integrals can be solved analytically to obtain a parametric representation of the phasing formula in terms of polynomial expressions in the auxiliary variable \( v \)

\[
\phi(v) = \phi_{\text{ref}} + \phi_N(v) \sum_{k=0}^{n} \phi_k^v v^k, \\
t(v) = t_{\text{ref}} + t_N(v) \sum_{k=0}^{n} t_k v^k, \tag{24.5}
\]

This corresponds to TaylorT2 in the enum Approximant \([1]\). These are implemented in modules LALInspiralWave2.

3. The second of the polynomials in Eq. (24.5) can be inverted and the resulting polynomial for \( v \) in terms of \( t \) can be substituted in \( \phi(v) \) to arrive at an explicit time-domain phasing formula

\[
\phi(t) = \phi_{\text{ref}} + \phi_N^t \sum_{k=0}^{n} \phi_k^t \theta^k, \quad F(t) = F_N \sum_{k=0}^{n} F_k \theta^k, \tag{24.6}
\]

where \( \theta = [\eta(t_{\text{ref}} - t)/(5m)]^{-1}/8 \), \( F \equiv d\phi/2\pi dt = v^3/(\pi m) \) is the instantaneous GW frequency and \( \eta = m_1 m_2/m^2 \) is the symmetric mass ratio. This corresponds to the Approximant TaylorT3 \([2,3]\) \([1]\). These are implemented in modules LALInspiralWave3.

The expansion coefficients in PN expansions of the various physical quantities are summarized in Tables 24.1 and 24.2.

Table 24.1: Taylor coefficients of the energy functions \( E_{T_e}(x) = E_N \sum_{k=0} E_k x^k \) and \( e_{T_e}(x) = e_N \sum_{k=0} e_k x^k \), \( e_{P_e}(x) = e_N/(1 + c_1 x/(1 + c_2 x/(1 + c_3 x/\ldots))) \) and the corresponding location of the iso and pole. As there are no terms of order \( v^{2k+1} \) we have exceptionally chosen (for this Table only) the expansion parameter to be \( x \equiv v^2 \) rather than \( v \). In all cases the \( k = 0 \) coefficient is equal to 1, the last stable orbit is defined only for \( k \geq 1 \) in the case of T-approximants and for \( k \geq 2 \) in the case of P-approximants and \( N \) denotes the “Newtonian value”.

<table>
<thead>
<tr>
<th>( k )</th>
<th>( N )</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E_k )</td>
<td>( -\eta^2/2 )</td>
<td>( 9 + \eta )</td>
<td>( 12 )</td>
</tr>
<tr>
<td>( e_k )</td>
<td>( -x(-v^2) )</td>
<td>( 3 + \eta )</td>
<td>( 4 )</td>
</tr>
<tr>
<td>( e_{P_e} )</td>
<td>( -x(-v^2) )</td>
<td>( 3 + \eta )</td>
<td>( 3 )</td>
</tr>
<tr>
<td>( x_{T_e}^{\text{iso}} )</td>
<td>( 6 )</td>
<td>( 9 + \eta )</td>
<td>( 12 )</td>
</tr>
<tr>
<td>( x_{T_e}^{\text{pole}} )</td>
<td>( 6 )</td>
<td>( 9 + \eta )</td>
<td>( 12 )</td>
</tr>
<tr>
<td>( x_{P_e}^{\text{iso}} )</td>
<td>( 6 )</td>
<td>( 9 + \eta )</td>
<td>( 12 )</td>
</tr>
<tr>
<td>( x_{P_e}^{\text{pole}} )</td>
<td>( 6 )</td>
<td>( 9 + \eta )</td>
<td>( 12 )</td>
</tr>
</tbody>
</table>

24.2 Stationary Phase Approximation

Consider a GW signal of the form,

\[
h(t) = 2a(t) \cos \phi(t) = a(t) \left[ e^{-i\phi(t)} + e^{i\phi(t)} \right], \tag{24.7}
\]

where \( \phi(t) \) is the phasing formula, either specified as an explicit function of time or given implicitly by a set of differential equations \([1]\). The quantity \( 2\pi F(t) = d\phi(t)/dt \) defines the instantaneous GW frequency \( F(t) \), and is assumed to be continuously increasing. (We assume \( F(t) > 0 \).) Now the Fourier transform \( \tilde{h}(f) \) of \( h(t) \) is defined as

\[
\tilde{h}(f) = \int_{-\infty}^{\infty} dt e^{2\pi if t} h(t) = \int_{-\infty}^{\infty} dt \ a(t) \left[ e^{2\pi if t - \phi(t)} + e^{2\pi if t + \phi(t)} \right]. \tag{24.8}
\]
Table 24.2: Taylor coefficients of the flux $F_{\tau_n}(x) = F_N(x) \sum_{k=0}^n F_k(\eta) t_k^k$, PN expansion of the GW phase $\phi_{\tau_n}(v) = \phi_{\text{ref}} + \phi_{N}(v) \sum_{k=0}^n \phi_k v^k$, explicit time-domain phasing formula $\phi_{\tau_n}(t) = \phi_{\text{ref}} + \phi_{N} \sum_{k=0}^n \phi_k t_k^k$, PN expansion of time $t_{\tau_n}(v) = t_{\text{ref}} + t_{N}(v) \sum_{k=0}^n t_k^k v^k$, explicit time-domain PN expansion of GW frequency, $F_{\tau_n}(t) = F_N \sum_{k=0}^n F_k t_k^k$, and frequency-domain phase function obtained in the stationary phase approximation to the chirp, $\psi(f) = 2\pi f t_{\text{ref}} - \phi_{\text{ref}} + \tau_N \sum_{k=0}^5 \tau_k (\pi mf)^{(k-5)/3}$. Here $N$ denotes the “Newtonian value” and $\theta = [(t_{\text{bao}} - t)/(5m)]^{-1/8}$. In all cases the $k = 0$ coefficient is 1 and the $k = 1$ coefficient is zero. In certain cases the 2.5 PN term involves $v^3 \log v$ or $\theta^3 \log \theta$ term rather than a $v^5$ or $\theta^5$ term. In those cases we conventionally include the log $v$ dependence in the listed coefficient. Chirp parameters $\tau_k, k \geq 1$, are related to the expansion parameters $t_k^k$ and $\phi_k^k$ via $\tau_k = (8\phi_k^k - 5t_k^k)/3$. We have given the simplified expressions for these in all cases, except $k = 5$ where no simplification occurs due to the presence of the log term in $\phi_5^k$.

<table>
<thead>
<tr>
<th>$k$</th>
<th>$N$</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F_k$</td>
<td>$\frac{32\eta^2 v^{10}}{5}$</td>
<td>$-\frac{1247}{336}$</td>
<td>$-\frac{357}{12}$</td>
<td>$4\pi$</td>
<td>$-\frac{4471}{9072}$</td>
</tr>
<tr>
<td>$t_k^v$</td>
<td>$-\frac{5m}{20768\eta}$</td>
<td>$\frac{743}{232}$</td>
<td>$+\frac{11\eta}{4}$</td>
<td>$-\frac{32\pi}{3}$</td>
<td>$-\frac{3058673}{504}$</td>
</tr>
<tr>
<td>$\phi_k^v$</td>
<td>$-\frac{1}{107\eta^3}$</td>
<td>$\frac{743}{1008} + \frac{55\eta}{12}$</td>
<td>$-10\pi$</td>
<td>$\frac{15923365}{1016064}$</td>
<td>$+\frac{271452}{1008}$</td>
</tr>
<tr>
<td>$\phi_k^4$</td>
<td>$-\frac{2}{9\theta^5}$</td>
<td>$\frac{8064}{768}$</td>
<td>$+\frac{55\eta}{96}$</td>
<td>$-\frac{3\pi}{4}$</td>
<td>$\frac{9275495}{1445608}$</td>
</tr>
<tr>
<td>$F_k$</td>
<td>$\frac{8\eta^3}{328\eta}$</td>
<td>$\frac{743}{768} + \frac{11\eta}{12}$</td>
<td>$-\frac{\pi}{4}$</td>
<td>$\frac{18590999}{256048}$</td>
<td>$+\frac{59675^2}{256048}$</td>
</tr>
<tr>
<td>$\tau_k$</td>
<td>$\frac{3}{128\eta}$</td>
<td>$\frac{5}{9} \left(\frac{743}{84} + 11\eta\right)$</td>
<td>$-16\pi$</td>
<td>$2\phi_4^v$</td>
<td>$\frac{1}{3} \left(8\phi_5^v - 5t_5^5\right)$</td>
</tr>
</tbody>
</table>

The above transform can be computed in the stationary phase approximation (SPA). For positive frequencies only the first term on the right contributes and yields the following usual SPA:

$$\tilde{h}^\text{uspa}(f) = \frac{a(t_f)}{\sqrt{\dot{F}(t_f)}} e^{i[\psi(f(t_f)) - \pi/4]}, \quad \psi_f(t) \equiv 2\pi f t - \phi(t),$$

and $t_f$ is the saddle point defined by solving for $t$, $dv_f(t)/dt = 0$, i.e., the time $t_f$ when the GW frequency $F(t)$ becomes equal to the Fourier variable $f$. In the adiabatic approximation where the value of $t_f$ is given by the following integral:

$$t_f = t_{\text{ref}} + m \int_{v_f}^{v_{\text{ref}}} \frac{E'(v)}{F(v)} dv, \phi(v) = \phi_{\text{ref}} + 2 \int_{v_{\text{ref}}}^{v} dv v^3 \frac{E'(v)}{F(v)},$$

where $v_{\text{ref}}$ is a fiducial reference point that sets the origin of time, $v_f \equiv (\pi mf)^{1/3}$, $E'(v) \equiv dE/dv$ is the derivative of the binding energy of the system and $F(v)$ is the gravitational wave flux. Using $t_f$ and $\phi(t_f)$ in the above equation and using it in the expression for $\psi_f(t)$ we find

$$\psi_f(t_f) = 2\pi f t_{\text{ref}} - \phi_{\text{ref}} + 2 \int_{v_f}^{v_{\text{ref}}} (v_f^3 - v^3) \frac{E'(v)}{F(v)} dv.$$

This is the general form of the stationary phase approximation which can be applied to all time-domain signals, including the P-approximant and effective one-body waveforms. In some cases the Fourier domain phasing can be worked out explicitly, which we now give:

Using PN expansions of energy and flux but re-expanding the ratio $E'(v)/F(v)$ in Eq. (24.11) one can solve the integral explicitly. This leads to the following explicit, Taylor-like, Fourier domain phasing formula:

$$\psi_f(t_f) = 2\pi f t_{\text{ref}} - \phi_{\text{ref}} + \psi_N \sum_{k=0}^5 \psi_k (\pi mf)^{(k-5)/3}$$

where the coefficients $\psi_k$ up to 2.5 post-Newtonian approximation are given by:

$$\psi_N = \frac{3}{128\eta}, \quad \psi_0 = 1, \quad \psi_1 = 0, \quad \psi_2 = \frac{5}{9} \left(\frac{743}{84} + 11\eta\right), \quad \psi_3 = -16\pi,$$

$$
\psi_4 = \frac{5}{72} \left(\frac{3058673}{7056} + \frac{5429}{7}\eta + 617\eta^2\right),
$$

and

$$\psi_5 = \frac{1}{3} \left(8\phi_5^v - 5t_5^5\right).$$
\[ \psi_5 = \frac{5}{3} \left( \frac{7729}{252} + \eta \right) \pi + \frac{8}{3} \left( \frac{38645}{672} + \frac{15}{8} \eta \right) \ln \left( \frac{v}{v_{\text{ref}}} \right) \pi. \]

Eq. (24.12) is (one of) the standardly used frequency-domain phasing formulas. This is what is implemented in \texttt{LALInspiralStationaryPhaseApproximation2} and corresponds to \texttt{TaylorF2} in the enum \texttt{Approximant}.

Alternatively, substituting (without doing any re-expansion or re-summation) for the energy and flux functions their PN expansions or the P-approximants of energy and flux functions and solving the integral in Eq. (24.11) numerically one obtains the T-approximant SPA or P-approximant SPA, respectively. However, just as in the time-domain, the frequency-domain phasing is most efficiently computed by a pair of coupled, non-linear, ODE’s:

\[ \frac{d\psi}{df} - 2\pi t = 0, \quad \frac{dt}{df} + \frac{\pi m^2}{3v^2} \frac{E'(f)}{F(f)} = 0, \quad (24.13) \]

rather than by numerically computing the integral in Eqs. (24.11). However, the current implementation in \texttt{LALInspiralStationaryPhaseApproximation1} solves the integral and corresponds to \texttt{TaylorF1} in the enum \texttt{Approximant}.

### 24.2.1 Amplitude in the Fourier domain

The derivative of the frequency that occurs in the amplitude of the Fourier transform, namely \( 1/\sqrt{F(t)} \), in Eq. (24.9), is computed using

\[ F(t) = \frac{dF}{dt} = \frac{dF}{dv} \frac{dv}{dE} \frac{dE}{dt} = \frac{3v^2}{\pi m} \left[ \frac{-F(v)}{E'(v)} \right], \quad (24.14) \]

where we have used the fact that the gravitational wave flux is related to the binding energy \( E \) via the energy balance equation \( F = -dE/dt \) and that \( F = v^3/(\pi m) \). At the Newtonian order \( E = -\eta v^3/2 \), and \( F = 32\eta^2 v^5/5 \), giving \( F(t(v)) = 96\eta v^{11}/(5\pi m^2) \). Taking \( 2a(t(v)) = v^2 \) (i.e., \( h(t) = v^2 \cos(\phi(t)) \)), this gives, the total amplitude of the Fourier transform to be

\[ \frac{a(t(v))}{\sqrt{F(t(v))}} = \sqrt{\frac{5 \pi m^2}{384 \eta}} v f^{-7/2}. \]

This is the amplitude used in most of literature. However, including the full PN expansion in \( \dot{F}(t) \), gives a better agreement between the time-domain and Fourier domains signals and this code therefore uses the full PN expansion \[5\].

### 24.3 Detection template family

The Fourier transform of a chirp waveform in the restricted post-Newtonian approximation in the stationary phase approximation is given, for positive frequencies \( f \), by [cf. Eq. (24.12)]

\[ \tilde{h}(f) = h_0 f^{-7/6} \exp \left[ \sum_k \psi_k f^{(k-5)/3} \right], \quad (24.15) \]

where \( h_0 \) is a constant for a given system and \( \psi_k \) are parameters that depend on the two masses of the binary. Since the time-domain waveform is terminated at when the instantaneous GW frequency reaches a certain value \( F_{\text{cut}} \) (which is either the last stable orbit or the light-ring defined by the model) and since the contribution to a Fourier component comes mainly from times when the GW instantaneous frequency reaches that value, it is customary to terminate the Fourier transform at the same frequency, namely \( f_{\text{cut}} = F_{\text{cut}} \). In otherwords, the Fourier transform is taken to be

\[ \tilde{h}(f) = h_0 f^{-7/6} \theta(f - f_{\text{cut}}) \exp \left[ \sum_k \psi_k f^{(k-5)/3} \right], \quad (24.16) \]

where \( \theta(x < 0) = 0 \) and \( \theta(x \geq 0) = 1 \). We have seen that there are different post-Newtonian models such as the standard post-Newtonian, P-approximants, effective one-body (see Sec. 24.4), and their overlaps with one another is not as good as we would like them to be. The main reason for this is that matched filtering
is sensitive to the phasing of the waves. It is not clear which model best describes the true GW signal from a compact binary inspiral although some, like the EOB, are more robust in their predictions than others. Thus, Buonanno, Chen and Vallisneri proposed \[ \text{[10, 11]} \] a phenomenological model as a detection template family (DTF) based on the above expression for the Fourier transform. Indeed, they proposed to use a DTF that depends on four parameters \((\psi_0, \psi_3, f_\text{cut}, \alpha)\) for non-spinning sources \[ \text{[10]} \] and on six parameters \((\psi_0, \psi_3, f_\text{cut}, \alpha_1, \alpha_2, \beta)\) for spinning sources \[ \text{[11]} \]. We have implemented both the non-spinning and spinning waveforms in \text{LALBCVWaveform} and \text{LALBCVSpinWaveform}, respectively.

### 24.3.1 DTF for non-spinning sources

The proposed waveform has structure similar to the one above:

\[
\hat{h}(f) = h_0 f^{-7/6} \left( 1 - \alpha f^{2/3} \right) \theta(f - f_\text{cut}) \exp \left[ \psi_0 f^{-5/3} + \psi_3 f^{-2/3} \right],
\]

where the motivation to include an amplitude correction term \(\alpha\) is based on the fact that the first post-Newtonian correction to the amplitude would induce a term like this. Note carefully that the phasing does not include the full post-Newtonian expansion but only the Newtonian and 1.5 post-Newtonian terms. It turns out this four-parameter family of waveforms has good overlap with the two-parameter family of templates corresponding to different post-Newtonian models and their improvements.

### 24.3.2 DTF for spinning sources

In the generic case of spinning black hole binaries there are a total of 17 parameters characterizing the waveform amplitude and shape (see Sec.24.5). However, the phasing of the waves is determined, in general, by about 10 parameters. Apostolatos et al. \[ \text{[12]} \] studied the time-evolution based on which Apostolatos found \[ \text{[13]} \] that far fewer parameters can be used to capture the full structure of the spinning black hole binary waveforms. BCV suggested \[ \text{[11]} \] that one could use a DTF of waveforms that resembles the non-spinning case. Their proposal has the following structure:

\[
\hat{h}(f) = h_0 f^{-7/6} \left[ 1 + \alpha_1 \cos(\beta f^{-2/3}) + \alpha_2 \sin(\beta f^{-2/3}) \right] \theta(f - f_\text{cut}) \exp \left[ \psi_0 f^{-5/3} + \psi_3 f^{-2/3} \right],
\]

where \(\alpha_1, \alpha_2\) and \(\beta\) are the parameters designed to capture the spin-induced modulation of the waveform.

### 24.4 Effective one-body approach

The entry \text{EOB} in the enum \text{Approximant} corresponds to the effective one-body (EOB) approach of Buonanno and Damour \[ \text{[6, 7, 8, 9]} \]. The EOB formalism allows one to evolve the dynamics of two black holes beyond the last stable orbit into the plunge phase thereby increasing the number of wave cycles, and the signal-to-noise ratio, that can be extracted. Here a set of four ordinary differential equations (ODEs) are solved by making an ansatz for the tangential radiation reaction force.

In practical terms, the time-domain waveform is obtained as the following function of the reduced time \(\hat{t} = t/m\):

\[
h(\hat{t}) = C v_\omega^2(\hat{t}) \cos(\phi(\hat{t})), \quad v_\omega \equiv \left( \frac{d\phi}{d\hat{t}} \right)^{1/3}, \quad \phi \equiv 2\varphi.
\]

(The amplitude \(C\) is chosen to be equal to 1 in our codes.) The four ODEs correspond to the evolution of the radial and angular coordinates and the corresponding momenta:

\[
\frac{dr}{d\hat{t}} = \frac{\partial \hat{H}}{\partial p_r}(r, p_r, p_\phi),
\]

\[
\frac{d\varphi}{d\hat{t}} = \hat{\omega} \equiv \frac{\partial \hat{H}}{\partial p_\varphi}(r, p_r, p_\phi),
\]

\[
\frac{dp_r}{d\hat{t}} + \frac{\partial \hat{H}}{\partial r}(r, p_r, p_\phi) = 0,
\]

\[
\frac{dp_\varphi}{d\hat{t}} = \hat{\Phi}(\hat{\omega}(r, p_r, p_\phi)).
\]
The reduced Hamiltonian \( \hat{H} \) (of the one-body problem) is given, at the 2PN approximation, by

\[
\hat{H}(r,p_r,p_\varphi) = \frac{1}{\eta} \left[ 1 + 2\eta \left( \sqrt{A(r)} \left( 1 + \frac{\dot{r}^2}{B(r)} + \frac{p_\varphi^2}{r^2} \right) - 1 \right) \right],
\]

where \( A(r) \equiv 1 - \frac{2}{r} + \frac{2\eta}{r^3} \), \( B(r) \equiv \frac{1}{A(r)} \left( 1 - \frac{6\eta}{r^2} \right) \).

(24.24)

The damping force \( \mathcal{F}_\varphi \) is approximated by

\[
\hat{\mathcal{F}}_\varphi = -\frac{1}{\eta v^2} \mathcal{F}_\nu(v_\omega),
\]

where \( \mathcal{F}_\nu(v_\omega) = \frac{32}{7} \eta v^{\frac{10}{3}} \mathcal{F}_\nu(v_\omega) \) is the P-approximant to the flux function.

The initial data \( (r_0, p_r^0, p_\varphi^0) \) are found using

\[
r_0^3 \left[ 1 + 2\eta \left( \frac{\sqrt{z(r_0)} - 1}{1 - 3\eta/r_0^2} \right) \right] - \omega_0^{-2} = 0, \quad p_\varphi^0 = \left( \frac{r_0^3 - 3\eta}{r_0^3 - 3\eta^2 + 5\eta} \right)^{1/2} r_0^3, \quad p_r^0 = \frac{\mathcal{F}_\nu(\omega)}{C(r_0, p_\varphi^0)} (dp_r^0/dr_0).
\]

(24.27)

where \( z(r) \) and \( C(r, p_\varphi) \) are given by

\[
z(r) = \frac{v^3 A^2(r)}{r^3 - 3r^2 + 5\eta}, \quad C(r, p_\varphi) = \frac{1}{\eta \hat{H}(r, 0, 0) \sqrt{z(r)}} \frac{A^2(r)}{1 - 6\eta/r^2}.
\]

(24.28)

The plunge waveform is terminated when the radial coordinate attains the value at the light ring \( r_\text{lr} \) given by the solution to the equation,

\[
r_\text{lr}^3 - 3r_\text{lr}^2 + 5\eta = 0.
\]

(24.29)

### 24.5 Spinning Modulated Chirps

Waveforms from spinning black hole binaries at 2PN order can be generated using the choice SpinTaylorT3 in the enum Approximant. Current implementation closely follows Ref. [12]. The orbital plane of a binary consisting of rapidly spinning compact objects precesses causing the polarization of the wave received at an antenna to continually change. This change depends on the source location on the sky and it might therefore be possible to resolve the direction to a source. It is therefore essential to specify the coordinate system employed in the description of the waveform. As in Ref. [12] the coordinate system is adapted to the detector with the x-y plane in the plane of the interferometer with the axes along the two arms.

In the restricted post-Newtonian approximation the evolution of a binary system comprising of two bodies of masses \( m_1 \) and \( m_2 \), spins \( S_1 = (s_1, \theta_1, \varphi_1) \) and \( S_2 = (s_2, \theta_2, \varphi_2) \), orbital angular momentum \( L = (L, \theta_0, \varphi_0) \), is governed by a set of differential equations given by:

\[
\dot{L} = \left[ \frac{4m_1 + 3m_2}{2m_1 m_3^3} - \frac{3S_2 \cdot L}{2 L^2 m_3^3} \right] S_1 + \left( \frac{4m_2 + 3m_1}{2m_2 m_3^3} - \frac{3S_1 \cdot L}{2 L^2 m_3^3} \right) S_2 \times L v^6 - \frac{32\eta^2 m}{5L} L v^7,
\]

(24.30)

\[
\dot{S}_1 = \left( \frac{4m_1 + 3m_2}{2m_1 m_3^3} - \frac{3S_2 \cdot L}{2 L^2 m_3^3} \right) L \times S_1 + \frac{S_2 \times S_1}{2m_3^3} v^6,
\]

(24.31)

\[
\dot{S}_2 = \left( \frac{4m_2 + 3m_1}{2m_2 m_3^3} - \frac{3S_1 \cdot L}{2 L^2 m_3^3} \right) L \times S_2 + \frac{S_1 \times S_2}{2m_3^3} v^6.
\]

(24.32)

where as before \( v = (\pi mf)^{1/3} \), \( f \) is the gravitational wave frequency, \( m = m_1 + m_2 \) is the total mass, and \( \eta = m_1 m_2 / m^2 \) is the (symmetric) mass ratio. An overdot denotes the time-derivative. In the evolution of the orbital angular momentum we have included the lowest order dissipative term [the second term containing \( v^7 \) in Equation (24.30)] while keeping the non-dissipative modulation effects caused by spin-orbit and spin-spin couplings [the first term within square brackets containing \( v^6 \) in Equation (24.30)]. The spins evolve non-dissipatively but their orientations change due to spin-orbit and spin-spin couplings. Though the non-dissipative terms are not responsible for gravitational wave emission, and therefore do not shrink the orbit, they cause to precess the orbit.

In the absence of spins the antenna observes the same polarization at all times; the amplitude and frequency of the signal both increase monotonically, giving rise to a chirping signal. Precession of the orbit
and spins cause modulations in the amplitude and phase of the signal and smear the signal’s energy spectrum over a wide band. For this reason we shall call the spin modulated (sm) chirp, a *smirch* (an anagram of **sm** and **chir**).

The strain $h(t)$ produced by a smirch at the antenna is given by

$$h(t) = -A(t) \cos[\phi(t) + \varphi(t)],$$  \hspace{1cm} (24.33)

where $A(t)$ is the precession-modulated amplitude of the signal, $\phi(t)$ is its post-Newtonian carrier phase that increases monotonically and $\varphi(t)$ is the polarization phase caused by the changing polarization of the wave relative to the antenna. (We have neglected the Thomas precession of the orbit which induces additional, but small, corrections in the phase.) For a source with position vector $\mathbf{N} = (D, \theta_S, \varphi_S)$ the amplitude is given by,

$$A(t) = \frac{2\eta mv^2}{D} \left[ 1 + (\hat{\mathbf{L}} \cdot \hat{\mathbf{N}})^2 \right]^{1/2}. \hspace{1cm} (24.34)$$

Here $\hat{\mathbf{L}} = \mathbf{L}/\mathbf{L}$, $\hat{\mathbf{N}} = \mathbf{N}/\mathbf{D}$ ($D$ is the distance to the source) and $\psi(t)$ is the precession-modulated polarization angle given by

$$\tan \psi(t) = \frac{\hat{\mathbf{L}}(t) \cdot \hat{\mathbf{z}} - (\hat{\mathbf{L}}(t) \cdot \hat{\mathbf{N}})(\hat{\mathbf{z}} \cdot \hat{\mathbf{N}})}{\mathbf{N} \cdot (\mathbf{L}(t) \times \hat{\mathbf{z}})}. \hspace{1cm} (24.35)$$

Also, $F_+$ and $F_\times$ are the antenna beam pattern functions are given by

$$F_+(\theta_S, \varphi_S, \psi) = \frac{1}{2} \left[ 1 + \cos^2 \theta_S \right] \cos 2\phi_S \cos 2\psi - \cos \theta_S \sin 2\phi_S \sin 2\psi, \hspace{1cm} (24.36)$$

$$F_\times(\theta_S, \phi_S, \psi) = \frac{1}{2} \left[ 1 + \cos^2 \theta_S \right] \cos 2\phi_S \sin 2\psi + \cos \theta_S \sin 2\phi_S \sin 2\psi. \hspace{1cm} (24.37)$$

Next, the polarization phase $\varphi(t)$ is

$$\tan \varphi(t) = \frac{2\hat{\mathbf{L}}(t) \cdot \hat{\mathbf{N}} F_\times(\theta_S, \varphi_S, \psi)}{\left[ 1 + (\mathbf{L}(t) \cdot \mathbf{N})^2 \right] F_+(\theta_S, \varphi_S, \psi)}. \hspace{1cm} (24.38)$$

And finally, for the carrier phase we use the post-Newtonian expression, but without the spin-orbit and spin-spin couplings. These spin couplings modify the carrier phase by amounts much smaller than the post-Newtonian effects. To second post-Newtonian order the carrier phase is given by

$$\phi(t) = -\frac{2}{\eta \theta^3} \left[ 1 + \left( \frac{3715}{8064} + \frac{55}{96} \eta \right) \theta^4 - \frac{3275495}{14450688} \left( \frac{284875}{258048} \eta + \frac{1855}{2048} \eta^2 \right) \theta^4 \right]. \hspace{1cm} (24.39)$$

where $\theta = [\eta(t_C - t)/(5m)]^{-1/8}$, $t_C$ being the time at which the two stars merge together and the gravitational wave frequency formally diverges.
24.6 Header LALInspiral.h

Header file for the template generation codes.

Synopsis

```c
#include <lal/LALInspiral.h>
```

This header file covers routines that are used in template generation.

Error codes

<table>
<thead>
<tr>
<th>name</th>
<th>code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>1</td>
<td>&quot;Arguments contained an unexpected null pointer&quot;</td>
</tr>
<tr>
<td>MEM</td>
<td>2</td>
<td>&quot;Memory allocation error&quot;</td>
</tr>
<tr>
<td>DIVO</td>
<td>3</td>
<td>&quot;Division by zero&quot;</td>
</tr>
<tr>
<td>SIZE</td>
<td>4</td>
<td>&quot;Invalid input range&quot;</td>
</tr>
<tr>
<td>CHOICE</td>
<td>5</td>
<td>&quot;Invalid choice for an input parameter&quot;</td>
</tr>
<tr>
<td>ORDER</td>
<td>6</td>
<td>&quot;unknown order specified&quot;</td>
</tr>
<tr>
<td>APPROXIMANT</td>
<td>7</td>
<td>&quot;Invalid model&quot;</td>
</tr>
<tr>
<td>PSI0</td>
<td>8</td>
<td>&quot;psi0 must be &gt; 0&quot;</td>
</tr>
<tr>
<td>PSI3</td>
<td>9</td>
<td>&quot;psi3 must be &lt; 0&quot;</td>
</tr>
<tr>
<td>ALPHA</td>
<td>10</td>
<td>&quot;alpha must be defined positive&quot;</td>
</tr>
<tr>
<td>FCUTOFF</td>
<td>11</td>
<td>&quot;fcutoff must be defined and &gt; 0&quot;</td>
</tr>
<tr>
<td>NOWAVEFORM</td>
<td>12</td>
<td>&quot;No Waveform generated&quot;</td>
</tr>
<tr>
<td>STOPPED</td>
<td>13</td>
<td>&quot;Waveform generation stopped&quot;</td>
</tr>
<tr>
<td>ROOTINIT</td>
<td>14</td>
<td>&quot;Can't find good bracket for BisectionFindRoot&quot;</td>
</tr>
<tr>
<td>FLOWER</td>
<td>15</td>
<td>&quot;flower too low in comparison to flso&quot;</td>
</tr>
<tr>
<td>VECTOR</td>
<td>16</td>
<td>&quot;Attempting to write beyond the end of vector&quot;</td>
</tr>
<tr>
<td>FLOWERINJ</td>
<td>17</td>
<td>&quot;flower for the injection must be greater than zero&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants `LALINSPIRALH_E<name>`, and the status descriptions in `LALINSPIRALH_MSGE<name>`. The source code with these messages is in `LALInspiral.h` on line 1.123.

Structures

1. **Order**: Enum that tells which post-Newtonian order is being used.

   ```c
typedef enum {
    newtonian,
    oneHalfPN,
    onePN,
    onePointFivePN,
    twoPN,
    twoPointFivePN,
    threePN,
    threePointFivePN,
    pseudoFourPN
  } Order;
```

- **newtonian**: Newtonain order, flux and energy both to the lowest order.
- **oneHalfPN**: same as before
- **onePN**: Both energy and flux to order $O(v^2)$ beyond the Newtonian order.
- **onePointFivePN**: Energy to order $O(v^2)$ and flux to order $O(v^3)$
- **twoPN**: Both energy and flux to order $O(v^4)$
- **twoPointFivePN**: Energy to order $O(v^4)$ and flux to order $O(v^5)$
- **threePN**: Both energy and flux to order $O(v^6)$
- **threePointFivePN**: Energy to order $O(v^6)$ and flux to order $O(v^7)$

In all cases, the gravitational wave phase (also frequency and time) as an expansion of the gauge invariant parameter $v$ is given up to the order specified by flux. Note that there are certain undetermined parameters at threePN and threePointFivePN. The waveform generation codes use a specific value of those parameters while generating the wave.

2. **Approximant**: Enum that specifies the PN approximant to be used in computing the waveform.

```c
typedef enum {
    TaylorT1,
    TaylorT2,
    TaylorT3,
    TaylorF1,
    TaylorF2,
    PadeT1,
    PadeF1,
    EOB,
    BCV,
    BCVSpin,
    SpinTaylorT3,
    SpinTaylor,
    FindChirpSP,
    FindChirpPTF,
    GeneratePPN,
    BCVC,
    FrameFile,
    AmpCorPPN,
    NumRel,
    Eccentricity,
    EOBNR
} Approximant;
```

- **TaylorT1**: Time domain Taylor approximant in which the energy and flux are both kept as Taylor expansions and a first order ordinary differential equation is solved for the GW phase as a function of $t$. Outputs a time-domain wave.
- **TaylorT2**: Time domain Taylor approximant in which the phase evolution $\varphi(t)$ is obtained by iteratively solving post-Newtonian expansions $\varphi(v)$ and $t(v)$. Outputs a time-domain wave.
- **TaylorT3**: Time domain Taylor approximant in which phase is explicitly given as a function of time. Outputs a time-domain wave.
- **TaylorF1**: The stationary phase approximation that correctly represents, in the Fourier domain, the waveform given by TaylorT1 approximant (see Ref. [5] for details). Outputs a frequency-domain wave.
- **TaylorF2**: The standard stationary phase approximation. Outputs a frequency-domain wave.
- **PadeT1**: Time-domain P-approximant. Outputs a time-domain wave.
- **PadeF1**: Frequency-domain P-approximant (not yet implemented).
- **EOB**: Effective one-body waveform Outputs a time-domain wave.
- **BCV**: Detection template family of Buonanno, Chen and Vallisneri [10]. Outputs a frequency-domain wave.
- **BCVSpin**: Detection template family of Buonanno, Chen and Vallisneri including spin effects[11]. Outputs a frequency-domain wave.
- **SpinTaylorT3** Spinning case T3 models
- **SpinTaylor** Spinning case PN models (should replace SpinTaylorT3 in the future)
- **FindChirpSP** The stationary phase templates implemented by FindChirpSPTemplate in the find-chirp package (equivalent to TaylorF2 at twoPN order).
- **GeneratePPN** The time domain templates generated by LALGeneratePPNInspiral() in the inject package (equivalent to TaylorT3 at twoPN order).
- **FrameFile** The waveform contains arbitrary data read from a frame file.

```c
typedef enum {
    m1Andm2,
    totalMassAndEta,
    totalMassUAndEta,
    totalMassAndMu,
    t01,
    t02,
    t03,
    t04,
    psi0Andpsi3,
    bhns,
    fixedMasses,
    fixedPsi,
    fixedTau,
    massesAndSpin,
    minmaxTotalMass,
    spinOnly
} InputMasses;
```

**InputMasses:** This structure is one of the members of the **InspiralTemplate** structure.

A user can specify the parameters of a binary using any of the following combination of **masses**:

- **m1Andm2**: component masses
- **totalMassAndEta**: total mass and symmetric mass ratio
- **totalMassUAndEta**: total mass and eta but uniform distribution in totalMass
- **totalMassAndMu**: total mass and reduced mass
- **t01**: unused; shouldn’t be used.
- **t02**: chirptimes $\tau_0$ and $\tau_2$
- **t03**: chirptimes $\tau_0$ and $\tau_3$, and
- **t04**: chirptimes $\tau_0$ and $\tau_4$
- **psi0Andpsi3**: BCV parameters $\psi_0$ and $\psi_3$

The LALRandomInspiralSignal uses that structure as an input. Since the injected waveform are not necessarily wanted to be random, we also provide the following options

- **bhns**: One of the mass is a Neutron star and the other a black hole. ($m1 \in [\minMass-3]$ and $m2 \in [3-\maxMass]$).
- **fixedMasses**: The two masses are given by the input parameter structure.
- **fixedPsi**: The two psi values are given by the input parameter structure.
- **fixedTau**: The two tau values are given by the input parameter structure.

3. **InspiralTemplate**: The inspiral waveform parameter structure containing information about the waveform to be generated.

```c
typedef struct
tagInspiralTemplate
{
    /* Parameters needed to generate Taylor/Pade waveforms */
```
Approximant approximant;
Order order;
Order ampOrder;
REAL8 mass1;
REAL8 mass2;
REAL8 fCutoff;
REAL8 fLower;
REAL8 tSampling;
REAL8 distance;
REAL8 signalAmplitude;
REAL8 startPhase;
REAL8 startTime;
INT4 ieta;

/* Additional parameters for EOB waveforms */

REAL8 Theta;
REAL8 Zeta2;

/* Parameters for BCV1 template */

REAL8 alpha;
REAL8 psi0;
REAL8 psi3;

/* Additional parameters for BCV2 template */

REAL8 beta;
REAL8 alpha1;
REAL8 alpha2;
REAL8 alpha3;
REAL8 alpha4;
REAL8 alpha5;
REAL8 alpha6;

/* Parameters for spinning BH waveform */

REAL8 inclination;
REAL8 orbitTheta0;
REAL8 orbitPhi0;
REAL8 spin1[3];
REAL8 spin2[3];
REAL8 sourceTheta;
REAL8 sourcePhi;
REAL8 polarisationAngle;

/* Spin parameters for the PTF template */
REAL8 chi; /* dimensionless spin of black hole (i.e. mass1) */
REAL8 kappa; /* cosine of angle between spin of mass1 and orb ang mom */

/* Parameters which are currently might be used */

REAL8 eccentricity;

/* Paramters which are computed using LALInspiralParameterCalc */

REAL8 chirpMass;
REAL8 eta;
REAL8 totalMass;
REAL8 fFinal;
REAL8 t0;
REAL8 t2;
REAL8 t3;
REAL8 t4;
REAL8 t5;
REAL8 tC;

/* Note that tc and fFinal are computed during waveform generation!!! */

/* useless parameters */
REAL4 minMatch;
REAL8 mu;
INT4 level;
INT4 number;
INT4 nStartPad;
INT4 nEndPad;
REAL8 OmegaS;
REAL8 vFinal;

InputMasses massChoice;
INT4Vector *segmentIdVec;
LIGOtimeGPS end_time;
EventIDColumn *event_id;
CHAR ifo[LIGOMETA_IFO_MAX];

/* Gamma[] is a vector that stores the upper triangular part of the metric in
the space of parameters. For time domain searches, Gamma[0,...,5] stores
the following information :
* Gamma[0] -> (tc,tc) metric component
* Gamma[1] -> (tc,t0) metric component
* Gamma[2] -> (tc,t3) metric component
* Gamma[3] -> (t0,t0) metric component
* Gamma[4] -> (t0,t3) metric component
* Gamma[5] -> (t3,t3) metric component
* For spinBCV searches, (in 4 dimensions) Gamma[0,...,9] would be required.
*/
REAL4 Gamma[10];

struct tagInspiralTemplate *next;
struct tagInspiralTemplate *fine;
}

• ieta: parameter that tells whether the symmetric mass ratio $\eta$ should be set to zero in the
PN expansions of GW flux and binding energy. If $\text{ieta}=0$ $\eta$ will be set to zero, otherwise the
appropriate value of $\eta$ from the given parameters will be used.

• level: Flag used in hierarical serached to indicate if this is a coarse or a fine template

• segmentIdVec: Vector of segment that have been filtered against this template needed for the
LDAS implementation of the inspiral search.

• number: Unique ID number for this template needed for the LDAS implementation of the inspiral
search.

• minMatch: The minimal match specified by the user when the bank that contains this template
was created.
• **nStartPad**: Number of leading elements in the signal generation to be set to zero (input). If template is requested, that value must be set to zero. In the injection routines related to inject package, that nStartPad is set to zero. However, for injection performed using the inspiral package, that value can be set to non zero.

• **nEndPad**: Number of trailing bins to be set to zero, the resulting waveform will have at least this many bins zero at the end, probably more since we always deal with an integer power of 2 array (input).

• **mass1**: Mass of the primary in solar mass (input/output).

• **mass2**: Mass of the secondary in solar mass (mass1 need not be larger than mass2 (input/output).

• **spin1[3]**: Spin vector of the primary (currently not in use)

• **spin2[3]**: Spin vector of the secondary (currently not in use)

• **sourceTheta**: Co-latitude in the direction to the source.

• **sourcePhi**: Azimuth angle in the direction to the source.

• **orbitTheta0**: Initial co-latitude of the orbit.

• **orbitPhi0**: Initial azimuth angle of the orbit.

• **inclination**: Inclination of the orbit (currently not in use)

• **distance**: Distance to the binary in seconds

• **psi0**: BCV parameter $\psi_0$.

• **psi3**: BCV parameter $\psi_3$.

• **alpha**: BCV amplitude correction factor $\alpha f_{cut}^{2/3}$

• **eccentricity**: initial eccentricity of the orbit (currently not in use)

• **totalMass**: total mass of the binary $m = m_1 + m_2$ in solar mass (input/output).

• **eta**: symmetric mass ratio $\eta = m_1 m_2 / m^2$. (input/output).

• **chirpMass**: chirp mass of the binary $= \eta^{5/3} m$ in solar mass (output).

• **t0**: Newtonain chirp time in seconds (input/output).

• **t2**: first post-Newtonian chirp time in seconds (input/output).

• **t3**: 1.5 post-Newtonian chirp time in seconds (input/output).

• **t4**: second post-Newtonian chirp time in seconds (output).

• **t5**: 2.5 post-Newtonian chirp time in seconds (output).

• **tC**: total chirp time seconds (output).

• **mu**: reduced mass (in solar mass) (input/output)

• **fLower**: lower frequency cutoff of the detector in Hz (input)

• **fCutoff**: upper frequency cutoff in Hz to be used in generating the waveform. If the last stable orbit frequency is smaller than the upper cutoff it will be used in terminating the waveform instead of fCutoff (input).

• **tSampling**: Sampling rate in Hz (input)

• **startPhase**: starting phase of the waveform in radians (input)

• **startTime**: starting time of the waveform (in sec); if different from zero, the waveform will start with an instantaneous frequency different from fLower and reach fLower at time (approximately) zero (input, not used in Stationary phase approximation)

• **signalAmplitude**: dimensionless amplitude of the signal (input, currently unused.)

• **rInitial**: initial radial separation of the two, in units of total mass bodies (used only in EOB waveforms) (output)

• **vInitial**: initial velocity parameter, in units of the speed of light (output)

• **rFinal**: final 'separation' between the bodies, in units of total mass (output)

• **vFinal**: final velocity parameter, in units of the speed of light (output)

• **fFinal**: final frequency reached, in units of Hz (output)
• **rLightRing**: radial coordinate at the light ring, in units of total mass (output)
• **OmegaS**: The 3PN (unknown) parameter; calculated to be equal to zero by Damour, Jaranowski and Schaffer (input).
• **Theta**: The 3PN unknown flux parameter; likely to be around unity; most waveform generation routines take theta to be zero. Robustness of the EOB waveform has been demonstrated for $-2 < \Theta < 2$. (input)
• **massChoice**: The pair of (mass) parameters given (see structure defining this member for more details) (input).
• **order**: Post-Newtonain order to be used in generating the wave (input).
• **approximant**: Post-Newtonain approximant to be used in generating the wave (input).
• **tagInspiralTemplate *next**: Linked list to the next coarse bank template (currently not filled by inspiral or bank codes)
• **tagInspiralTemplate *fine**: Linked list to the next fine bank template (currently not filled by inspiral or bank codes)

4. **InspiralACSTParams**: This structure is needed to solve the differential equation giving the evolution of the orbital angular momentum and the spin angular momenta in the case of spinning black hole binaries.

```c
typedef struct
tagInspiralACSTParams
{
    REAL8 v;
    REAL8 magS1;
    REAL8 magS2;
    REAL8 NCap[3];
    REAL8 spin1[3];
    REAL8 M;
    REAL8 fourM1Plus;
    REAL8 fourM2Plus;
    REAL8 oneBy2Mcube;
    REAL8 threeBy2Mcube;
    REAL8 thirtytwoBy5etc;
} InspiralACSTParams;
```

• **v**: parameter of 'integration': $v=\sqrt{M/r}$
• **magS1**: The constant spin magnitude of the primary.
• **magS2**: The constant spin magnitude of the secondary.
• **NCap[3]**: Source direction (unit vector) in detector coordinate system.
• **spin1[3]**: Spin of the larger body.
• **M**: Total mass of the binary (in seconds).
• **fourM1Plus**: $= (4m_1 + 3m_2)/(2m_1M^3)$ (all masses expressed in seconds).
• **fourM2Plus**: $= (4m_2 + 3m_1)/(2m_2M^3)$ (all masses expressed in seconds).
• **oneBy2Mcube**: $= 1/(2M^4)$
• **threeBy2Mcube**: $= 3/(2M^3)$
• **thirtytwoBy5etc**: $(32/5)\eta^2 M$

5. **EtaTau02In, EtaTau04In**: These are the input structures needed to solve for the mass ratio $\eta$ given the chirptimes $(\tau_0, \tau_2)$ or $(\tau_0, \tau_4)$.

```c
typedef struct
tagEtaTau02In
{
    REAL8 t2;
    REAL8 A2;
    REAL8 B2;
} EtaTau02In;
```
typedef struct
tagEtaTau04In
{
    REAL8 t4;
    REAL8 A4;
    REAL8 B4;
    REAL8 C4;
} EtaTau04In;

Here, \( t_2 = \tau_2, A_2 = A_2(\tau_0/A_0)^{3/5}, \) and \( B_2 = B_2, \) where \( A_0 = 5/[256(\pi f_s)^{8/3}], A_2 = 3715/[64512(\pi f_s)^2], B_2 = 4620/3715. \)

Similarly, \( t_4 = \tau_4, A_4 = A_4(\tau_0/A_0)^{1/5}, B_4 = B_4 \) and \( C_4 = C_4, \) where \( A_0 = 5/[256(\pi f_s)^{8/3}], A_4 = 5 \times 3058673/[128 \times 1016064(\pi f_s)^{4/3}], B_4 = 5429 \times 1016064/(1008 \times 3058673), \) and \( C_4 = 617 \times 1016064/(144 \times 3058673). \)

6. **InspiralToffInput**: This is a structure needed by the inner workings of the inspiral wave generation code.

    typedef struct
tagInspiralToffInput
{
    REAL8 tN;
    REAL8 t2;
    REAL8 t3;
    REAL8 t4;
    REAL8 t5;
    REAL8 t6;
    REAL8 t7;
    REAL8 t6;
    REAL8 t7;
    REAL8 piM;
    REAL8 tc;
    REAL8 t;
} InspiralToffInput;

7. **expnCoeffs**: This structure contains various post-Newtonian and P-approximant expansion coefficients; the meanings of the coefficients is indicated as comments before each list.

    typedef struct
tagexpnCoeffs {
    int ieta;
    /* coefficients in the Taylor expansion of new energy function*/
    REAL8 eTaN, eTa1, eTa2, eTa3;
    /* coefficients in the Pade expression of new energy function*/
    REAL8 ePaN, ePa1, ePa2, ePa3;
    /* coefficients in the Taylor expansion of usual energy function*/
    REAL8 ETaN, ETa1, ETa2, ETa3;
    /* coefficients in the Taylor expansion of the derivative of the usual energy function*/
    REAL8 dETaN, dETa1, dETa2, dETa3;
    /* Taylor expansion coefficients of energy flux*/
    REAL8 FTaN, FTa1, FTa2, FTa3, FTa4, FTa5, FTa6, FTa7, FT16;
    /* Taylor expansion coefficients of factored flux*/
    REAL8 fTaN, fTa1, fTa2, fTa3, fTa4, fTa5, fTa6, fTa7;
    /* Coefficients of the corresponding P-approximant*/
    REAL8 fPaN, fPa1, fPa2, fPa3, fPa4, fPa5, fPa6, fPa7, fPa8;
    /* Taylor expansion coefficients in t(v)*/
} expnCoeffs;
REAL8 tvaN, tva2, tva3, tva4, tva5, tva6, tva7, tv16;
/* Taylor expansion coefficients in phi(v)*/
REAL8 pvaN, pva2, pva3, pva4, pva5, pva6, pva7, pv16;
/* Taylor expansion coefficients in phi(t)*/
REAL8 ptaN, pta2, pta3, pta4, pta5, pta6, pta7, pt16;
/* Taylor expansion coefficients in f(t)*/
REAL8 ftaN, fta2, fta3, fta4, fta5, fta6, fta7, ft16;
/* Taylor expansion coefficients in psi(f) in the Fourier phase*/
REAL8 pfaN, pfa2, pfa3, pfa4, pfa5, pfa6, pfa7, pf15, pf16;
/* Taylor expansion for the spinning case */
REAL8 ST[9], thetahat;

/** sampling rate and interval*/
REAL8 samplingrate, samplinginterval;
/** symmetric mass ratio, total mass, component masses*/
REAL8 eta, totalmass, m1, m2;
/** unknown 3PN parameters, euler constant*/
REAL8 lambda, theta, EulerC, omegaS, zeta2;

/** initial and final values of frequency, time, velocity; lso values of velocity and frequency; final phase.*/
REAL8 f0, fn, t0, tn, v0, vn, vf, vlso, flso, phiC;

/** last stable orbit and pole defined by various Taylor and P-approximants*/
REAL8 vlsoT0, vlsoT2, vlsoT4, vlsoT6;
REAL8 vlsoP0, vlsoP2, vlsoP4, vlsoP6;
REAL8 vpoleP4, vpoleP6;
} expnCoeffs;

8. Energy, flux, phase, time and frequency functions: The following functions are generic function definitions that will be used in template generation. The function LALInspiralChooseModel, which is called by wave generation interface code, points these functions to the appropriate specific functions depending on the choices made by the user.

typedef REAL8 EnergyFunction(
    REAL8 v,
    expnCoeffs *ak);

typedef REAL8 FluxFunction(
    REAL8 v,
    expnCoeffs *ak);

typedef void (TestFunction)(
    REAL8Vector *vector1,
    REAL8Vector *vector2,
    void *params);

typedef void (InspiralPhasing2)(
    LALStatus *status,
    REAL8 *phase,
    REAL8 v,
    expnCoeffs *ak);

typedef void (InspiralPhasing3)(
    LALStatus *status,
    REAL8 *f,
REAL8 td,
expnCoeffs *ak);

typedef void (InspiralFrequency3)(
    LALStatus *status,
    REAL8 *f,
    REAL8 td,
    expnCoeffs *ak);

typedef void (InspiralTiming2) (
    LALStatus *status,
    REAL8 *toff,
    REAL8 f,
    void *params);

9. **expnFunc**: Structure to hold the pointers to the generic functions defined above.

```c
typedef struct
tagexpnFunc
{
    EnergyFunction *dEnergy;
    FluxFunction *flux;
    InspiralTiming2 *timing2;
    InspiralPhasing2 *phasing2;
    InspiralPhasing3 *phasing3;
    InspiralFrequency3 *frequency3;
} expnFunc;
```

10. **TofVIn** and **TofVIntegrandIn**: Structures needed to compute the time elapsed from/to the starting epoch of the waveform when the velocity parameter was $v_0$, to/from the current epoch when velocity parameter is $v$.

```c
typedef struct
tagTofVIn
{
    REAL8 t;
    REAL8 v0;
    REAL8 t0;
    REAL8 vlo;
    REAL8 totalmass;
    EnergyFunction *dEnergy;
    FluxFunction *flux;
    expnCoeffs *coeffs;
} TofVIn;
```

```c
typedef struct
tagTofVIntegrandIn
{
    EnergyFunction *dEnergy;
    FluxFunction *flux;
    expnCoeffs *coeffs;
} TofVIntegrandIn;
```

11. **InspiralPhaseIn** and **PhiofVIntegrandIn**: Structures used to compute the phase of the signal from the ‘beginning’, when the velocity parameter is $v_0$, to a time when the velocity parameter has evolved to a user input value $v$.

```c
typedef struct
tagInspiralPhaseIn
{
    EnergyFunction *dEnergy;
    FluxFunction *flux;
    expnCoeffs *coeffs;
} InspiralPhaseIn;
```
typedef struct
tagInspiralPhaseIn
{
    REAL8 v0;
    REAL8 phi0;
    EnergyFunction *dEnergy;
    FluxFunction *flux;
    expnCoeffs *coeffs;
} InspiralPhaseIn;

typedef struct
tagPhiofVIntegrandIn
{
    EnergyFunction *dEnergy;
    FluxFunction *flux;
    expnCoeffs *coeffs;
} PhiofVIntegrandIn;

12. **InspiralDerivativesIn**: Structure used as an input to compute the derivatives needed in solving the phasing formula when the approximant is TaylorT1, TaylorP1 or EOB.

typedef struct
tagInspiralDerivativesIn
{
    REAL8 totalmass;
    EnergyFunction *dEnergy;
    FluxFunction *flux;
    expnCoeffs *coeffs;
} InspiralDerivativesIn;

13. **rk4GSLIntegrator**: Structure containing steps and controls for the GSL Runge-Kutta solver

typedef struct
tagrk4GSLIntegrator
{
    const gsl_odeiv_step_type *type;
    gsl_odeiv_step *step;
    gsl_odeiv_control *control;
    gsl_odeiv_evolve *evolve;
    REAL8 *y;
    rk4In *input;
} rk4GSLIntegrator;

14. **rk4In**: Structure used as an input to Runge-Kutta solver.

typedef struct
tagrk4In
{
    TestFunction *function;
    REAL8 x;
    REAL8Vector *y;
    REAL8Vector *dydx;
    REAL8Vector *yt;
    REAL8Vector *dym;
    REAL8Vector *dyt;
    REAL8 h;
    INT4 n;
} rk4In;
24.6.1 Module LALInspiralParameterCalc.c

Given a pair of masses (or other equivalent parameters) compute related chirp parameters.

Prototypes

void LALInspiralParameterCalc (LALStatus *status, InspiralTemplate *params)

- **params**: Input/Output, given a pair of binary parameters and a lower frequency cutoff, other equivalent parameters are computed by this function.

Description

The code takes as its input `params->fLower` in Hz and a pair of masses (in units of $M$⊙) or chirptimes (in seconds measured from `params->fLower`) and computes all the other mass parameters in the `params` structure. Users choice of input pair of *masses* should be specified by appropriately setting the variable `params->massChoice` as described in the Table below:

<table>
<thead>
<tr>
<th><code>params-&gt;massChoice</code></th>
<th>User should set</th>
<th>in units</th>
<th>which means</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1Andm2</td>
<td>(mass1, mass2)</td>
<td>$(M, M)$</td>
<td>$(m_1, m_2)$</td>
</tr>
<tr>
<td>totalMassAndEta</td>
<td>(totalmass, eta)</td>
<td>$(M, 0 &lt; \eta \leq 1/4)$</td>
<td>$(m, \eta)$</td>
</tr>
<tr>
<td>totalMassAndMu</td>
<td>(totalmass, mu)</td>
<td>$(M, M)$</td>
<td>$(m, \mu)$</td>
</tr>
<tr>
<td>t02</td>
<td>(t0, t2)</td>
<td>(sec, sec)</td>
<td>$(\tau_0, \tau_2)$</td>
</tr>
<tr>
<td>t03</td>
<td>(t0, t3)</td>
<td>(sec, sec)</td>
<td>$(\tau_0, \tau_3)$</td>
</tr>
<tr>
<td>t04</td>
<td>(t0, t4)</td>
<td>(sec, sec)</td>
<td>$(\tau_0, \tau_4)$</td>
</tr>
</tbody>
</table>

If `massChoice` is not set properly an error condition will occur and the function is aborted with a return value 999. In the above list $m_1$ and $m_2$ are the masses of the two compact objects, $m = m_1 + m_2$ is the total mass, $\eta = m_1 m_2 / (m_1 + m_2)^2$ is the symmetric mass ratio, $\mu = m_1 m_2 / (m_1 + m_2)$ is the reduced mass and $\tau$’s are the chirptimes defined in terms of $f_a = f_{Lower}$ by:

$$
\tau_0 = \frac{5}{256 \eta m^{5/3}(\pi f_a)^{8/3}}, \quad \tau_2 = \frac{3715 + 4620 \eta}{64512 \eta m(\pi f_a)^2}, \quad \tau_3 = \frac{\pi}{8 \eta m^{2/3}(\pi f_a)^{5/3}}
$$

$$
\tau_4 = \frac{5}{128 \eta m^{1/3}(\pi f_a)^{4/3}} \left[ 3058673 + \frac{5429}{1016064} \eta \left( \frac{144}{617} \eta^2 \right) \right], \quad \tau_5 = \frac{5}{256 \eta f_a} \left( \frac{7729}{252} + \eta \right).
$$

(24.40)

Whichever pair of parameters is given to the function as an input, the function calculates the rest. Apart from the various masses and chirptimes the function also calculates the chirp mass $M = (\mu^3 m^2)^{1/5}$ and the total chirp time $\tau_C$ consistent with the approximation chosen:

<table>
<thead>
<tr>
<th>Approximation</th>
<th>$\tau_C$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Newtonian</td>
<td>$\tau_0$</td>
</tr>
<tr>
<td>onePN</td>
<td>$\tau_0 + \tau_2$</td>
</tr>
<tr>
<td>onePointFivePN</td>
<td>$\tau_0 + \tau_2 - \tau_3$</td>
</tr>
<tr>
<td>twoPN</td>
<td>$\tau_0 + \tau_2 - \tau_3 + \tau_4$</td>
</tr>
<tr>
<td>twoPointFivePN</td>
<td>$\tau_0 + \tau_2 - \tau_3 + \tau_4 - \tau_5$</td>
</tr>
</tbody>
</table>

Algorithm

Root finding by bisection method is used to solve for mass ratio $\eta$ when chirptimes ($\tau_0$, $\tau_2$) or ($\tau_0$, $\tau_4$) is input.
Uses

When appropriate this function calls:
LALDBisectionFindRoot
LALEtaTau02
LALEtaTau04

Notes
24.6.2 Module LALInspiralAmplitude.c

Given an inspiral template structure containing the binary distance and a set of mass parameters, that module provides functions to compute the related amplitude.

Prototypes

```c
void LALInspiralRestrictedAmplitude (LALStatus *status, 
    InspiralTemplate *params )
```

Description

The inspiral template structure can stored (1) the distance of the binary (2) a set of binary masses such as the two masses or the total mass and eta and (3) an amplitude which is arbitrary fixed to unity when templates are computed. However we might need to have a template with the physical amplitude (for instance to deal with injections). The function `LALInspiralRestrictedAmplitude` takes an `InspiralTemplate` structure as input/output to return the restricted Newtonian amplitude by using the following formula.

\[ A = \frac{4c}{d\eta} M^{-5/3}. \]  

(24.41)

where \( d \) is in Mpc and \( M \) in solar mass. The result is stored in the `signalAmplitude` variable of the inspiral template structure.

Uses

When appropriate this function calls:

`LALInspiralParameterCalc`

Notes
24.6.3 Module `LALInspiralWaveLength.c`

Module to calculate the number of data points (to the nearest power of 2) needed to store a waveform.

Prototypes

```c
void LALInspiralWaveLength(
    LALStatus  *status,
    UINT4       *length,
    InspiralTemplate params
)
```

- **length**: output, number of bins to the nearest power of 2 greater than the minimum length required to store a wave of parameters as in `params`.
- **params**: input, parameters of the binary system.

Description

This module first calls `LALInspiralChooseModel`, which gives the length of the waveform in seconds. That function returns an estimated waveform length. However, the length might not be appropriate in some extreme cases (large masses and large lower cut-off frequency). It is especially true in the EOB case. Therefore, we introduce two constants namely `LALINSPIRAL_LENGTHOVERESTIMATION` (in percentage) which overestimate the length of the waveform and `LALINSPIRAL_MINIMALWAVELENGTH` which is the minimal waveform length in seconds. Multiplying this by the sampling rate `params.tSampling` gives the minimum number of samples needed to hold the waveform. To this are added the number of bins of leading and trailing zeroes requested by the user in `params.nStartPad` and `params.nEndPad`. The resulting number is rounded to an upward power of 2 and returned in `length`.

Algorithm

Uses

This function calls:
- `LALInspiralSetup`
- `LALInspiralChooseModel`
24.6.4 Module LALInspiralChooseModel.c

Module to set the pointers to the required energy and flux functions. Normally, a user is not required to call this function to generate a waveform.

Prototypes

static REAL8 dEt0(REAL8 v, expnCoeffs *ak)
static REAL8 dEt2(REAL8 v, expnCoeffs *ak)
static REAL8 dEt4(REAL8 v, expnCoeffs *ak)
static REAL8 dEt6(REAL8 v, expnCoeffs *ak)
static REAL8 Ft0(REAL8 v, expnCoeffs *ak)
static REAL8 Ft2(REAL8 v, expnCoeffs *ak)
static REAL8 Ft3(REAL8 v, expnCoeffs *ak)
static REAL8 Ft4(REAL8 v, expnCoeffs *ak)
static REAL8 Ft5(REAL8 v, expnCoeffs *ak)
static REAL8 Ft6(REAL8 v, expnCoeffs *ak)
static REAL8 Ft7(REAL8 v, expnCoeffs *ak)
static REAL8 ep0(REAL8 v, expnCoeffs *ak)
static REAL8 ep2(REAL8 v, expnCoeffs *ak)
static REAL8 ep4(REAL8 v, expnCoeffs *ak)
static REAL8 ep6(REAL8 v, expnCoeffs *ak)
static REAL8 dEp0(REAL8 v, expnCoeffs *ak)
static REAL8 dEp2(REAL8 v, expnCoeffs *ak)
static REAL8 dEp4(REAL8 v, expnCoeffs *ak)
static REAL8 dEp6(REAL8 v, expnCoeffs *ak)
static REAL8 Fp0(REAL8 v, expnCoeffs *ak)
static REAL8 Fp1(REAL8 v, expnCoeffs *ak)
static REAL8 Fp2(REAL8 v, expnCoeffs *ak)
static REAL8 Fp3(REAL8 v, expnCoeffs *ak)
static REAL8 Fp4(REAL8 v, expnCoeffs *ak)
static REAL8 Fp5(REAL8 v, expnCoeffs *ak)
static REAL8 Fp6(REAL8 v, expnCoeffs *ak)
static REAL8 Fp7(REAL8 v, expnCoeffs *ak)
static REAL8 Fp8(REAL8 v, expnCoeffs *ak)
void LALInspiralChooseModel(
   LALStatus *status,
   expnFunc *f,
   expnCoeffs *ak,
   InspiralTemplate *params
)

• **f**: Output containing the pointers to the appropriate energy, flux, frequency, timing and phasing functions.
• **ak**: Output containing the PN expansion coefficients.
• **params**: Input containing binary chirp parameters.
Description

This module gives the post-Newtonian expansions and/or P-approximants to the energy, its derivative and gravitational-wave flux functions. More specifically, the static REAL8 functions below give Taylor expansions of $dE/dv$, and $F(v)$, P-approximants of $e(v)$, $dE/dv$ (derived from $e(v)$) and $F(v)$.

LALInspiralChooseModel is used to set pointers to the required energy and flux functions $E'_T(v)$, $F_T(v)$, $E'_P(v)$ and $F_P(v)$, in expnFunc, as also the GW phasing and frequency functions used in the various approximants to generate the waveform. More specifically pointers are set to the following functions in the structure expnFunc:

- EnergyFunction *dEnergy
- FluxFunction *flux
- InspiralTiming2 *timing2
- InspiralPhasing2 *phasing2
- InspiralPhasing3 *phasing3
- InspiralFrequency3 *frequency3

LALInspiralChooseModel also outputs in ak the last stable orbit (LSO) velocity $v_{LSO}$ (as ak->vn) defined by the equation $E'(v_{LSO}) = 0$, the values of the GW frequency $f_{LSO} = v_{LSO}^3/(\pi m)$ (as ak->fn) and time (as ak->tn) elapsed from params->fLower to smaller of fCutOff and ak->fn by evaluating the integral

$$ t_n = t_0 - m \int_{v_0}^{v_n} \frac{E'(v)}{F(v)} \, dv, \quad (24.42) $$

where $t_0$ (usually equal to zero) is the user specified starting time for the waveform when the wave frequency reaches params->fLower and $v_0 = (\pi mf)^{1/3}$ (with $f = \text{params->fLower}$) is the velocity at time $t_0$. Note that $E'(v)$ and $F(v)$ are defined in f->dEnergy and f->flux.

Algorithm

Numerical integration is used to compute ak->tn.

Uses

LALInspiralTofV

Notes

- See Damour, Iyer and Sathyaprakash, PRD 57, 885, 1998 for further details. Damour, Iyer and Sathyaprakash, PRD 63, 044023, 2001 is a resource paper that summarizes how to generate waveforms in different approximations to the dynamics of a compact binary under radiation reaction.

- The Pade Approximant for the 1PN expansion is undefined as also EOB at orders less than 2PN. BCV is independent of the PN order. Spinning waveforms are only defined at the highest PN order.
24.6.5 Module **LALInspiralSetup.c**

Module to generate all the Taylor and Pade coefficients needed in waveform generation.

**Prototypes**

```c
void LALInspiralSetup (  
    LALStatus *status,  
    expnCoeffs *ak,  
    InspiralTemplate *params  
)
```

- **ak**: Output containing PN expansion coefficients of various physical quantities such as energy, flux, frequency, phase and timing.
- **params**: Input containing binary chirp parameters.

**Description**

Module to generate all the coefficients needed in the Taylor and Pade expressions for the energy and flux functions $E'(v)$ and $F(v)$. These are used to solve the gravitational wave phasing formula. The coefficients are used by the function **LALInspiralChooseModel** to define the energy and flux functions by accessing the structure `ak` and are tabulated in the two Tables 24.1 and 24.2.

**Algorithm**

None.

**Uses**

None.

**Notes**

Author: Sathyaprakash, B. S.
24.6.6 Module LALInspiralInit.c

Module to initialize some parameters for waveform generation.

Prototypes

```c
void LALInspiralInit (LALStatus *status,
                     InspiralTemplate *params,
                     InspiralInit *paramsInit)
```

Description

The input parameters is an InspiralTemplate structure which provides the waveform parameters such as masses, lower frequency... The function `LALInspiralInit` calls the `LALInspiralParameterCalc` function in order to compute all the mass parameters. Then, `LALInspiralRestrictedAmplitude` function is called to get the restricted Newtonian amplitude. `LALInspiralWaveLength`, `LALInspiralSetup` and `LALInspiralChooseModel` are also called in order to estimate the waveform length which is stored in an output structure called `InspiralInit`. We also stored energy, flux and evolution function of flux and energy in that structure.

The `LALInspiralChooseModel` function might fail or send a non zero status code. That function force it to be zero therefore the codes which use `LALInspiralInit` (mainly injection code right now) won’t stopped. Of course, if status code is non zero, we have to keep trace of it. Thus, the length of the waveform is fixed to zero in case of problems such as negative length, cutoff frequency lower than the lower cutoff frequency ....

Uses

```c
LALInspiralParameterCalc
LALInspiralRestrictedAmplitude
LALInspiralWaveLength LALInspiralChooseModel LALInspiralSetup
```

Notes

There is only one assert on the InspiralTemplate variable since all relevant asserts are already included in the different functions which are called throughout the `LALInspiralInit` function.
24.6.7 Module LALInspiralWaveTaper.c

The code LALInspiralWaveTaper imposes a smooth time window on a chirp for waves in the time-domain. It takes in a Real4Vector and searches for the beginning and end points of the signal, in case there are null data points at either end.

It tapers the wave over n data points according to formula 3.35 of the Physical Review D 63 084036. If the user selects a wave to be tapered over n data points then the 1st and nth data points are 0 and 1. For instance if n = 5, then there are 3 points for which the data is scaled by the formula.

The bookends option allows the user to specify whether just the start, just the end or both the start and end of the signal are tapered. This corresponds to bookends = 1, 2 or 3 respectively. If bookends does not equal 1, 2 or 3 then option 3 is assumed.

Prototypes

```c
void LALInspiralWaveTaper(
    LALStatus *status,
    REAL4Vector *signal,
    UINT4 n,
    UINT4 bookends
);
```

Description

Uses

Notes

* Author: McKechan D J A
24.6.8 Module LALInspiralWave.c and LALInspiralWaveTemplates.c

Interface routine needed to generate all waveforms in the inspiral package.

To generate a waveform a user is normally required to (a) choose the binary parameters, starting frequency, number of bins of leading and trailing zero-padding, etc., in the structure InspiralTemplate params and (b) call the following three functions in the order given: LALInspiralParameterCalc, LALInspiralWaveLength, and LALInspiralWave. Either a time- or a frequency-domain signal is returned depending upon the approximant requested (see Notes below).

Prototypes

```c
void LALInspiralWave(
    LALStatus *status,
    REAL4Vector *signal,
    InspiralTemplate *params
);
```

- **signal**: Output containing the inspiral waveform.
- **params**: Input containing binary chirp parameters.

```c
void LALInspiralWaveTemplates(
    LALStatus *status,
    REAL4Vector *signal1,
    REAL4Vector *signal2,
    InspiralTemplate *params
);
```

- **signal1**: Output containing the 0-phase inspiral waveform.
- **signal2**: Output containing the $\pi/2$-phase inspiral waveform.
- **params**: Input containing binary chirp parameters.

Description

The code LALInspiralWave is the user interface to the inspiral codes. It takes from the user all the physical parameters which specify the binary, and calls the relevant wave generation function. Currently nine different approximants are fully implemented. These are TaylorT1, TaylorT2, TaylorT3, TaylorF1, TaylorF2, PadeT1, EOB, BCV, SpinTaylorT3. Taylor approximants can all be generated at seven different post-Newtonian orders, from Newtonian to 3.5 PN order, PadeT1 exists at order 1.5PN and higher, EOB at orders 2 and higher. SpinTaylorT3 is implemented only at 2PN order by solving the evolution equations for the spin and orbital angular momenta and a time-domain phasing formula. Finally, PN order is undefined for BCV. The approximant and the order are set up by the enums Approximant and Order, respectively.

The waveforms are all terminated one bin before the last stable orbit is reached. The last stable orbit corresponding to a given Approximant and Order is defined as follows: For all Taylor approximants at orders 0PN, 1PN and 1.5PN $v_{lso}^2 = 1/6$, and at 2PN, 2.5PN, 3PN and 3.5PN $v_{lso}^2 = x_{lso}^0$, where $x_{lso}$ is defined in Table 24.1. In the case of Pade approximant at 1.5PN order $v_{lso}^2 = 1/6$, and at orders 2PN, 2.5PN, 3PN and 3.5PN $v_{lso}^2 = x_{P}^{lso}$, where $x_{P}^{lso}$ is defined in Table 24.1. In the case of EOB approximant, defined only at orders greater than 2PN, the plunge waveform is terminated at the light-ring orbit defined by Equation 24.29.

In the case of LALInspiralWaveTemplates *signal1 contains the ‘0-phase’ inspiral template and *signal2 contains a signal that is $\pi/2$ out of phase with respect to *signal1. Currently, a template pair is generated only for the following approximants: TaylorT1, TaylorT2, TaylorT3, PadeT1, EOB.

See the test codes for examples of how to generate different approximations.

Algorithm

Simple use of switch statement to access different PN approximations.
Uses

Depending on the user inputs one of the following functions is called:
LALInspiralWave1
LALInspiralWave2
LALInspiralWave3
LALInspiralStationaryPhaseApprox1
LALInspiralStationaryPhaseApprox2
LALEOBWaveform
LALBCVWaveform
LALInspiralSpinModulatedWave

Notes

- A time-domain waveform is returned when the approximant is one of TaylorT1, TaylorT2, TaylorT3, PadeT1, EOB, SpinTaylorT3
- A frequency-domain waveform is returned when the approximant is one of TaylorF1, TaylorF2, BCV. In these cases the code returns the real and imaginary parts of the Fourier domain signal in the convention of fftw. For a signal vector of length $n=\text{signal}\rightarrow\text{length}$ ($n$ even):
  - $\text{signal}\rightarrow\text{data}[0]$ is the real 0th frequency component of the Fourier transform.
  - $\text{signal}\rightarrow\text{data}[n/2]$ is the real Nyquist frequency component of the Fourier transform.
  - $\text{signal}\rightarrow\text{data}[k]$ and $\text{signal}\rightarrow\text{data}[n-k]$, for $k=1,\ldots, n/2-1$, are the real and imaginary parts of the Fourier transform at a frequency $k\Delta f = k/T$, $T$ being the duration of the signal and $\Delta f = 1/T$ is the frequency resolution.
24.6.9 Module LALInspiralWave.c and LALInspiralWaveTemplates.c

Interface routine needed to generate all waveforms in the inspiral package.

To generate a waveform a user is normally required to (a) choose the binary parameters, starting frequency, number of bins of leading and trailing zero-padding, etc., in the structure InspiralTemplate params and (b) call the following three functions in the order given: LALInspiralParameterCalc, LALInspiralWaveLength, and LALInspiralWave. Either a time- or a frequency-domain signal is returned depending upon the approximant requested (see Notes below).

Prototypes

```c
void LALInspiralWave(  
    LALStatus *status,  
    REAL4Vector *signal,  
    InspiralTemplate *params
)

• signal: Output containing the inspiral waveform.
• params: Input containing binary chirp parameters.
```

```c
void LALInspiralWaveTemplates(  
    LALStatus *status,  
    REAL4Vector *signal1,  
    REAL4Vector *signal2,  
    InspiralTemplate *params
)

• signal1: Output containing the 0-phase inspiral waveform.
• signal2: Output containing the \( \pi/2 \)-phase inspiral waveform.
• params: Input containing binary chirp parameters.
```

Description

The code LALInspiralWave is the user interface to the inspiral codes. It takes from the user all the physical parameters which specify the binary, and calls the relevant wave generation function. Currently nine different approximants are fully implemented. These are TaylorT1, TaylorT2, TaylorT3, TaylorF1, TaylorF2, PadeT1, EOB, BCV, SpinTaylorT3. Taylor approximants can all be generated at seven different post-Newtonian orders, from Newtonian to 3.5 PN order, PadeT1 exists at order 1.5PN and higher, EOB at orders 2 and higher. SpinTaylorT3 is implemented only at 2PN order by solving the evolution equations for the spin and orbital angular momenta and a time-domain phasing formula. Finally, PN order is undefined for BCV. The approximant and the order are set up by the enums Approximant and Order, respectively.

The waveforms are all terminated one bin before the last stable orbit is reached. The last stable orbit corresponding to a given Approximant and Order is defined as follows: For all Taylor approximants at orders 0PN, 1PN and 1.5PN \( v_{2,iso}^2 = 1/6 \), and at 2PN, 2.5PN, 3PN and 3.5PN \( v_{2,iso}^2 = v_{3,iso}^2 \), where \( v_{3,iso} \) is defined in Table 24.1. In the case of Pade approximant at 1.5PN order \( v_{2,iso}^2 = 1/6 \), and at orders 2PN, 2.5PN, 3PN and 3.5PN \( v_{2,iso}^2 = x_{P4} \), where \( x_{P4} \) is defined in Table 24.1. In the case of EOB approximant, defined only at orders greater than 2PN, the plunge waveform is terminated at the light-ring orbit defined by Equation 24.29.

In the case of LALInspiralWaveTemplates *signal1 contains the ‘0-phase’ inspiral template and *signal2 contains a signal that is \( \pi/2 \) out of phase with respect to *signal1. Currently, a template pair is generated only for the following approximants: TaylorT1, TaylorT2, TaylorT3, PadeT1, EOB.

See the test codes for examples of how to generate different approximations.

Algorithm

Simple use of switch statement to access different PN approximations.
Uses

Depending on the user inputs one of the following functions is called:

LALInspiralWave1
LALInspiralWave2
LALInspiralWave3
LALInspiralStationaryPhaseApprox1
LALInspiralStationaryPhaseApprox2
LALEOBWaveform
LALBCVWaveform
LALInspiralSpinModulatedWave

Notes

• A time-domain waveform is returned when the approximant is one of TaylorT1, TaylorT2, TaylorT3, PadeT1, EOB, SpinTaylorT3

• A frequency-domain waveform is returned when the approximant is one of TaylorF1, TaylorF2, BCV. In these cases the code returns the real and imaginary parts of the Fourier domain signal in the convention of fftw. For a signal vector of length n=signal->length (n even):
  – signal->data[0] is the real 0th frequency component of the Fourier transform.
  – signal->data[n/2] is the real Nyquist frequency component of the Fourier transform.
  – signal->data[k] and signal->data[n-k], for k=1,..., n/2-1, are the real and imaginary parts of the Fourier transform at a frequency kΔf = k/T, T being the duration of the signal and Δf = 1/T is the frequency resolution.
24.6.10 Module LALInspiralWave1.c and LALInspiralWave1Templates.c

The code LALInspiralWave1 generates an time-domain inspiral waveform corresponding to the approximant TaylorT1 and PadeT1 as outlined in the documentation for the function LALInspiralWave.

Prototypes

```c
void LALInspiralWave1(   
    LALStatus   *status,
    REAL4Vector *signal,
    InspiralTemplate *params
 )
```

- **signal**: Output containing the inspiral waveform.
- **params**: Input containing binary chirp parameters.

```c
void LALInspiralWave1Templates(   
    LALStatus   *status,
    REAL4Vector *signal1,
    REAL4Vector *signal2,
    InspiralTemplate *params
 )
```

- **signal1**: Output containing the 0-phase inspiral waveform.
- **signal2**: Output containing the $\pi/2$-phase inspiral waveform.
- **params**: Input containing binary chirp parameters.

Description

LALInspiralWave1 is called if the user has specified the enum approximant to be either TaylorT1 or PadeT1. LALInspiralWave1Templates is exactly the same as LALInspiralWave1, except that it generates two templates one for which the starting phase is params.startPhase and the other for which the phase is params.startPhase + $\pi/2$.

Algorithm

This code uses a fourth-order Runge-Kutta algorithm to solve the ODEs in Equation (24.4).

Uses

LALInspiralSetup
LALInspiralChooseModel
LALInspiralVelocity
LALInspiralPhasing1
LALInspiralDerivatives
LALRungeKutta4.

Notes
24.6.11 Module LALInspiralWave2.c and LALInspiralWave2Templates.c

These modules generate a time-domain chirp waveform of type TaylorT2.

Prototypes

```c
void LALInspiralWave2(
    LALStatus *status,
    REAL4Vector *output,
    InspiralTemplate *params
)
```

- **output**: Output containing the inspiral waveform.
- **params**: Input containing binary chirp parameters.

```c
void LALInspiralWave2Templates(
    LALStatus *status,
    REAL4Vector *output1,
    REAL4Vector *output2,
    InspiralTemplate *params
)
```

- **output1**: Output containing the 0-phase inspiral waveform.
- **output2**: Output containing the π/2-phase inspiral waveform.
- **params**: Input containing binary chirp parameters.

Description

LALInspiralWave2 generates TaylorT2 approximant wherein the phase of the waveform is given as an implicit function of time as in Equation (24.5). A template is required to be sampled at equal intervals of time. Thus, first of the equations in Equation (24.5) is solved for \( v \) at equally spaced values of the time steps \( t_k \) and the resulting value of \( v_k \) is used in the second equation to obtain the phase \( \phi_k \).

LALInspiralWave2Templates is exactly the same as LALInspiralWave2 except that it generates two waveforms that differ in phase by \( \pi/2 \).

Uses

- LALInspiralParameterCalc
- LALDiseaseFindRoot
- LALInspiralPhasing2

Notes

Author: Sathyaprakash, B. S.

$Id: LALInspiralWave2.c,v 1.25 2008/03/06 13:04:36 sathya Exp$
24.6.12 Module LALInspiralWave3.c and LALInspiralWave3Templates.c

These modules generate a time-domain chirp waveform of type TaylorT3.

Prototypes

```c
void LALInspiralWave3 (  
    LALStatus *status,  
    REAL4Vector *output,  
    InspiralTemplate *params  
)
```

- **output**: Output containing the inspiral waveform.
- **params**: Input containing binary chirp parameters.

```c
void LALInspiralWave3Templates (  
    LALStatus *status,  
    REAL4Vector *output1,  
    REAL4Vector *output2,  
    InspiralTemplate *params  
)
```

- **output1**: Output containing the 0-phase inspiral waveform.
- **output2**: Output containing the $\pi/2$-phase inspiral waveform.
- **params**: Input containing binary chirp parameters.

Description

LALInspiralWave3 generates TaylorT3 approximant which corresponds to the case wherein the phase of the waveform is given as an explicit function of time as in Equation (24.6). LALInspiralWave3Templates simultaneously generates two inspiral waveforms and the two differ in phase by $\pi/2$.

Algorithm

Uses

- LALInspiralParameterCalc
- LALInspiralChooseModel
- LALInspiralSetup
- LALInspiralPhasing3 (via expnFunc)
- LALInspiralFrequency3. (via expnFunc)

Notes

Author: Sathyaprakash, B. S.

$Id: LALInspiralWave3.c,v 1.27 2008/03/06 13:04:51 sathya Exp$
24.6.13 Module `LALInspiralStationaryPhaseApprox1.c`

This module computes the stationary phase approximation to the Fourier transform of a chirp waveform by integrating Eq. [24.11]

Prototypes

```c
void LALInspiralStationaryPhaseApprox1 (  
    LALStatus *status,  
    REAL4Vector *signal,  
    InspiralTemplate *params
)
```

- `signal`: Output containing the inspiral waveform.
- `params`: Input containing binary chirp parameters.

Description

This module generates the Fourier domain waveform that is analogous of the time-domain approximant `TaylorT1`. Instead of re-expanding the the energy and flux functions they are kept in tact and the integral in Eq. [24.11] is solved numerically. The code returns the Fourier transform packed in the same way as `fftw` would for the Fourier transform of a real vector. For a signal vector of length \( n = \text{signal->length} \) (\( n \) even):

- `signal->data[0]` is the real 0th frequency component of the Fourier transform.
- `signal->data[n/2]` is the real Nyquist frequency component of the Fourier transform.
- `signal->data[k]` and `signal->data[n-k]`, for \( k=1,\ldots, n/2-1 \), are the real and imaginary parts of the Fourier transform at a frequency \( k\Delta f = k/T \), \( T \) being the duration of the signal and \( \Delta f = 1/T \) is the frequency resolution.

Algorithm

The lal code `LALDRombergIntegrate` is used to solve the integral in Eq. [24.11]. The reference points are chosen so that on inverse Fourier transforming the time-domain waveform will

- be padded with zeroes in the first `params->nStartPad` bins,
- begin with a phase shift of `params->nStartPhase` radians,
- have an amplitude of \( n v^2 \).

Uses

- `LALInspiralSetup`
- `LALInspiralChooseModel`
- `LALDRombergIntegrate`

Notes

If it is required to compare the output of this module with a time domain signal one should use an inverse Fourier transform routine that packs data in the same way as `fftw`. Moreover, one should divide the resulting inverse Fourier transform by a factor \( n/2 \) to be consistent with the amplitude used in time-domain signal models.

- Author: B.S. Sathyaprakash
24.6.14 Module **LALInspiralStationaryPhaseApprox2.c**

This module computes the usual stationary phase approximation to the Fourier transform of a chirp waveform given by Eq. (24.12).

### Prototypes

```c
void LALInspiralStationaryPhaseApprox2 (  
    LALStatus *status,  
    REAL4Vector *signal,  
    InspiralTemplate *params  
)
```

- **signal**: Output containing the inspiral waveform.
- **params**: Input containing binary chirp parameters.

### Description

Computes the Fourier transform of the chirp signal in the stationary phase approximation and returns the real and imaginary parts of the Fourier domain signal in the convention of fftw. For a signal vector of length \( n = \text{signal->length} \) (\( n \) even):

- \( \text{signal->data}[0] \) is the real 0th frequency component of the Fourier transform.
- \( \text{signal->data}[n/2] \) is the real Nyquist frequency component of the Fourier transform.
- \( \text{signal->data}[k] \) and \( \text{signal->data}[n-k] \), for \( k = 1, \ldots, n/2-1 \), are the real and imaginary parts of the Fourier transform at a frequency \( k \Delta f = k/T \), \( T \) being the duration of the signal and \( \Delta f = 1/T \) is the frequency resolution.

### Algorithm

The standard SPA is given by Eq. (24.12). We define a variable function pointer `LALInspiralTaylorF2Phasing` and point it to one of the static functions defined within this function that explicitly calculates the Fourier phase at the PN order chosen by the user. The reference points are chosen so that on inverse Fourier transforming the time-domain waveform will

- be padded with zeroes in the first `params->nStartPad` bins,
- begin with a phase shift of `params->nStartPhase` radians,
- have an amplitude of \( n^2 \).

### Uses

- `LALInspiralSetup`
- `LALInspiralChooseModel`
- `LALInspiralTaylorF2Phasing[0234567]PN`
- None

### Notes

If it is required to compare the output of this module with a time domain signal one should use an inverse Fourier transform routine that packs data in the same way as fftw. Moreover, one should divide the resulting inverse Fourier transform by a factor \( n/2 \) to be consistent with the amplitude used in time-domain signal models.

* **Author**: B.S. Sathyaprakash
Module LALEOBWaveform.c and LALEOBWaveformTemplates.c

Module to generate effective-one-body waveforms.

Prototypes

```c
void LALEOBWaveform (LALStatus *status, REAL4Vector *signal, InspiralTemplate *params)
```

```c
void LALSTPNWaveform (LALStatus *status, REAL4Vector *signal, InspiralTemplate *params)
```

```c
void LALSTPNWaveformTemplates (LALStatus *status, REAL4Vector *signal1, REAL4Vector *signal2, InspiralTemplate *params)
```

- `signal`: Output containing the inspiral waveform.
- `params`: Input containing binary chirp parameters.

```c
void LALEOBWaveformTemplates (LALStatus *status, REAL4Vector *signal1, REAL4Vector *signal2, InspiralTemplate *params)
```

- `signal1`: Output containing the 0-phase inspiral waveform.
- `signal2`: Output containing the $\pi/2$-phase inspiral waveform.
- `params`: Input containing binary chirp parameters.

```c
void LALEOBWaveformForInjection (LALStatus *status, CoherentGW *waveform, InspiralTemplate *params, PPNParamStruc *ppnParams)
```

- `injhc`: Output containing the 0-phase inspiral waveform.
- `injhp`: Output containing the $\pi/2$-phase inspiral waveform.
- `injphase`: Output containing the phase of inspiral waveform.
- `injfreq`: Output containing the frequency of inspiral waveform.
- `params`: Input containing binary chirp parameters.
Description

By solving four coupled ordinary differential equations in Eq. (24.20)-(24.23) this module computes the waveform in Eq. (24.19) (see discussion in Sec. 24.4 for details on how the initial conditions are chosen, when the waveform is terminated and so on). No quasi-normal mode oscillations are added to the plunge signal so the waveform is terminated around $2.8M$.

3PN vs 2PN

At 3PN, two additional parameters exist namely OmegaS and Zeta2. The first parameters should be set to zero. If the second parameter is also set to zero then the waveform corresponds to the standard waveforms.

Algorithm

A fourth order Runge-Kutta is used to solve the differential equations.

Uses

- LALInspiralSetup
- LALInspiralChooseModel
- LALInspiralVelocity
- LALInspiralPhasing1
- LALDBisectionFindRoot
- LALRungeKutta4
- LALHCapDerivatives
- LALHCapDerivatives3PN
- LALHCapDerivativesP4PN
- LALlightRingRadius
- LALlightRingRadius3PN
- LALlightRingRadiusP4PN
- LALpphiInit
- LALpphiInit3PN
- LALpphiInitP4PN
- LALprInit
- LALprInit3PN
- LALprInitP4PN
- LALrOfOmega
- LALrOfOmega3PN
- LALrOfOmegaP4PN

Notes

The length of the waveform returned by LALInspiralWaveLength is occasionally smaller than what is required to hold an EOB waveform. This is because EOB goes beyond the last stable orbit up to the light ring while LALInspiralWaveLength assumes that the waveform terminates at the last stable orbit. It is recommended that a rather generous params->nEndPad be used to prevent the code from crashing.
24.6.16 Module LALBCVWaveform.c

This module contains a single function LALBCVWaveform.

Prototypes

```c
void LALBCVWaveform(
    LALStatus  *status,
    REAL4Vector *signal,
    InspiralTemplate *params
);
```

- **signal**: Output containing the *Fourier transform* of the inspiral waveform.

- **params**: Input containing binary chirp parameters; it is necessary and sufficient to specify the following parameters of the `params` structure: `psi0`, `psi3`, `alpha`, `fEndBCV(fFinal)`, `nStartPad`, `fLower`, `tSampling`. All other parameters in `params` are ignored.

```c
void LALBCVSpinWaveform(
    LALStatus  *status,
    REAL4Vector *signal,
    InspiralTemplate *params
);
```

- **signal**: Output containing the *Fourier transform* of the inspiral waveform.

- **params**: Input containing binary chirp parameters; it is necessary and sufficient to specify the following parameters of the `params` structure: `psi0`, `psi3`, `alpha1`, `alpha2`, `beta`, `fEndBCV(fFinal)`, `nStartPad`, `fLower`, `tSampling`. All other parameters in `params` are ignored.

*Description*

This module can be used to generate *detection template family* of Buonanno, Chen and Vallisneri [10, 11]. There are two modules: LALBCVWaveform and LALBCVSpinWaveform. The former can be used to generate non-spinning waveforms and the DTF it implements is given in Sec. 24.3 and Eq. (24.17) and the latter to generate spinning waveforms (Eq. 24.18).

*Algorithm*

A straightforward implementation of the formula. Note that the routine returns *Fourier transform* of the signal as opposed to most other modules in this package which return time-domain signals. Also, the amplitude is quite arbitrary.

*Uses*

```c
ASSERT
ATTACHSTATUSPTR
DETACHSTATUSPTR
INITSTATUS
RETURN
```

*Notes*

- **Author**: B. S. Sathyaprakash
- **$Id$**: LALBCVWaveform.c,v 1.15 2007/09/27 16:48:33 gareth Exp $
24.6.17 Module LALInspiralSpinningBHBinary.c

This module generates the inspiral waveform from a binary consisting of two spinning compact stars.

Prototypes

```c
void LALInspiralSpinModulatedWave(
    LALStatus *status,
    REAL4Vector *signal,
    InspiralTemplate *in
)
```

```c
/* NOT DONE FOR THE MOMENT should remove the polarisation effects which are already
   taken into account in inject package */
void LALInspiralSpinModulatedWaveForInjection(
    LALStatus *status,
    CoherentGW *waveform,
    InspiralTemplate *params,
    PPNParamStruc *ppnParams
)
```

- `signal`: Output containing the spin modulated inspiral waveform.
- `in`: Input containing binary chirp parameters.

Description

Using the formalism described in Apostolatos et al [12] and Blanchet et al. [2] and formulas summarized in Sec. 24.5 this module computes the spin-modulated chirps from a pair of compact stars in orbit around each other.

Algorithm

This code uses a fourth-order Runge-Kutta algorithm to solve the nine first-order, coupled, ordinary differential equations in Eq. 24.30 Eq. 24.31 and Eq. 24.32. The solution is then used in Eq. ?? (and following equations) to get the waveform emitted by a spinning black hole binary.

Uses

LALInspiralSetup
LALInspiralChooseModel
LALInspiralVelocity
LALInspiralPhasing3
LALRungeKutta4.

Notes
24.6.18 Module LALSTPNWaveform.c

DOCUMENTATION IN PROGRESS

Module to generate STPN (spinning binaries) waveforms in agreement with the injection package (return a CoherentGW structure).

Prototypes

```c
void LALSTPNWaveformForInjection (
    LALStatus *status,
    CoherentGW *waveform,
    InspiralTemplate *params,
    PPNParamStruct *ppnParams
);
```

Description

Algorithm

Uses

LALSTPNderivatives

Notes

Author: Vallisneri, M. Cokelaer, T.

$Id: LALSTPNWaveform.c,v 1.17 2007/09/13 09:03:22 gareth Exp $
24.6.19 Module LALEtaTau02.c

Given \( \tau_0 \) and \( \tau_2 \) compute the mass ratio \( \eta \).

Prototypes

```c
void LALEtaTau02(
    LALStatus *status,
    REAL8 *x,
    REAL8 eta,
    void *p
);
```

Description

Given \( \tau_0 \) and \( \tau_2 \) one can determine \( \eta \) by solving

\[
- \eta^{2/5} \tau_2 + A_2 \left( \frac{\tau_0}{A_0} \right)^{3/5} \left( 1 + B_2 \eta \right) = 0,
\]

(24.43)

where \( A_0 = 5/[256(\pi f_s)^{8/3}] \), \( A_2 = 3715/[64512(\pi f_s)^2] \), \( B_2 = 4620/3715 \). This function returns the LHS of the above equation in \( x \) for a given eta.

Algorithm

None.

Uses

None.

Notes

The void pointer *p should point to a struct of type EtaTau02In:

```c
void *p;
EtaTau02In q;
```

```c
... p = (void *) &q;
```
Given $\tau_0$ and $\tau_4$ solve for the mass ratio $\eta$.

Prototypes

```c
void LALEtaTau04(
    LALStatus *status,
    REAL8 x,
    REAL8 eta,
    void *p
);
```

Description

Given $\tau_0$ and $\tau_4$ one can determine $\eta$ by solving

$$-\eta^{4/5}\tau_4 + A_4 \left( \frac{\tau_0}{A_0} \right)^{1/5} (1 + B_4 \eta + C_4 \eta^2) = 0,$$

(24.44)

where $A_0 = \frac{5}{[256(\pi f_s)^{8/3}]}$, $A_4 = \frac{5 \times 3058673}{[128 \times 1016064(\pi f_s)^{4/3}]}$, $B_4 = \frac{5429 \times 1016064}{1008 \times 3058673}$, and $C_4 = \frac{617 \times 1016064}{144 \times 3058673}$. This function returns the LHS of the above equation in $x$ for a given $\eta$.

Algorithm

None.

Uses

None.

Notes

The void pointer *p should point to a struct of type EtaTau04In

```c
void *p;
EtaTau04In q;
...
p = (void *) &q;
```
24.6.21 Module LALInspiralDerivatives.c

Module to calculate the RHS of the differential equations in Eq. (24.4).

Prototypes

```c
void
LALInspiralDerivatives (  
    REAL8Vector *values,  
    REAL8Vector *dvalues,  
    void         *params
)
```

- **values**: Input containing the values of the variables \( v \) and \( \phi \) at the current time.
- **dvalues**: Output containing the derivatives \( dv/dt \) and \( d\phi/dt \) at the current time.
- **params**: Input of type `InspiralDerivativesIn` that must be cast to a `void`.

Description

This module calculates the right-hand sides of the following two coupled first-order differential equations which are solved to obtain the gravitational wave phasing equation, as described in the documentation for the function `LALInspiralWave1`: The equations are

\[
\frac{dv}{dt} = -\frac{F(v)}{mE'(v)}, \quad \frac{d\phi}{dt} = \frac{2v^3}{m}.
\]  

Algorithm

Uses
None.

Notes

- This function has been intentionally made non-LAL compliant in the sense that it has no status structure. This is because this code outputs the RHS of the differential equations and is called repeatedly by a function that integrates the two differential equations and should therefore not suffer from undue overheads.

- The input `params` is of type `InspiralDerivativesIn` and must be cast to a `void` before calling this function. For example,

```c
InspiralDerivativesIn in3;
void *funcParams;

in3.totalmass = totalmass;
...
funcParams = (void *) &in3;
```
24.6.22 Module LALInspiralVelocity.c

The function LALInspiralVelocity calculates the velocity $v$ which corresponds to a time $t$ in the inspiralling binary system.

**Prototypes**

```c
void LALInspiralVelocity(
    LALStatus *status,
    REAL8 *v,
    TofVIn   *ak
)
```

**Description**

The function LALInspiralVelocity calculates the velocity $v$ corresponding to a time $t$ in the evolution of an inspiralling binary system. It does this by iteratively solving

$$ t(v) = t_0 - m \int_{v_0}^{v} \frac{E'(v)}{F(v)} dv . $$

(24.46)

LALInspiralVelocity calculates $v$, given $t(v)$, $t_0$, $m$, $v_0$, $E'(v)$ and $F(v)$.

**Algorithm**

**Uses**

LALDBisectionFindRoot

**Notes**
24.6.23 Module LALInspiralPhasing1.c

This module is used to set the phase of the waveform so that it is equal to the user specified phase $\phi_0$ when the ‘velocity’ of the system is equal to $v$.

Prototypes

```c
void LALInspiralPhasing1 (  
    LALStatus *status,  
    REAL8 *phiofv,  
    REAL8 v,  
    void *params
)
```

Description

The function LALInspiralPhasing1 calculates the phase $\phi(v)$ using the phasing formula,

$$
\phi(v) = \phi_0 - 2 \int_{v_0}^{v} \frac{E'(v)}{F(v)} \, dv .
$$

(24.47)

LALInspiralPhasing1 calculates $\phi(v)$, given $\phi_0$, $v_0$, $v$, $E'(v)$ and $F(v)$. The user can specify the phase to be of a particular value at an arbitrary point on the waveform when the post-Newtonian evolution variable $v$ reaches a specific value. Choosing $v = v_0$, the initial velocity, means that the initial phase of the wave is $\phi_0$; Choosing $v = v_{\text{iso}}$ means that the phase at the last stable orbit is $\phi_0$ and so on.

Algorithm

Numerical integration.

Uses

LALDRombergIntegrate

Notes
24.6.24 Module LALInspiralPhiofVIntegrand.c

The function LALInspiralPhiofVIntegrandIn calculates the quantity $v^3 E'(v)/F(v)$.

Prototypes

```
void LALInspiralPhiofVIntegrand (  
    LALStatus *status,  
    REAL8 *integrand,  
    REAL8 v,  
    void *params  
)
```

Description

The function LALInspiralPhiofVIntegrandIn calculates the quantity $v^3 E'(v)/F(v)$.

Algorithm

Uses

This function calls dEnergy and flux functions that are defined in the expnFunc structure and represent $E'(v)$ and $F(v)$, respectively, and pointed to the appropriate PN functions with a call to LALInspiralChooseModel.

Notes
24.6.25 Module LALInspiralPhasing2.c

The code LALInspiralPhasing2.c calculates the phase of an inspiral waveform as a function of the instantaneous frequency of the wave, up to 2\textsuperscript{nd} post-Newtonian order.

Prototypes

```c
void LALInspiralPhasing2_0PN (
    LALStatus *status,
    REAL8 *phase,
    REAL8 \nu,
    expnCoeffs *ak
)
```

```c
void LALInspiralPhasing2_2PN (
    LALStatus *status,
    REAL8 *phase,
    REAL8 \nu,
    expnCoeffs *ak
)
```

```c
void LALInspiralPhasing2_3PN (
    LALStatus *status,
    REAL8 *phase,
    REAL8 \nu,
    expnCoeffs *ak
)
```

```c
void LALInspiralPhasing2_4PN (
    LALStatus *status,
    REAL8 *phase,
    REAL8 \nu,
    expnCoeffs *ak
)
```

```c
void LALInspiralPhasing2_5PN (
    LALStatus *status,
    REAL8 *phase,
    REAL8 \nu,
    expnCoeffs *ak
)
```

```c
void LALInspiralPhasing2_6PN (
    LALStatus *status,
    REAL8 *phase,
    REAL8 \nu,
    expnCoeffs *ak
)
```
void LALInspiralPhasing2_7PN (  
    LALStatus *status,  
    REAL8 *phase,  
    REAL8 v,  
    expnCoeffs *ak  
)  

- **phase**: Output, the phase of the wave at the current epoch.
- **v**: Input, the PN expansion parameter at the current epoch.
- **ak**: Input containing PN expansion coefficients.

**Description**

The phase of the inspiral wave corresponding to the Approximant TaylorT2 as in Equation eq:InspiralPhasing2.

**Algorithm**

None.

**Uses**

None.

**Notes**

None.
24.6.26 Module LALInspiralPhasing3.c

The code LALInspiralPhasing3.c calculates the phase of the waveform from an inspiralling binary system as a function of time up to second post-Newtonian order.

Prototypes

```c
void LALInspiralPhasing3_0PN(
    LALStatus *status,
    REAL8 *phase,
    REAL8 td,
    expnCoeffs *ak
);
```

```c
void LALInspiralPhasing3_2PN(
    LALStatus *status,
    REAL8 *phase,
    REAL8 td,
    expnCoeffs *ak
);
```

```c
void LALInspiralPhasing3_3PN(
    LALStatus *status,
    REAL8 *phase,
    REAL8 td,
    expnCoeffs *ak
);
```

```c
void LALInspiralPhasing3_4PN(
    LALStatus *status,
    REAL8 *phase,
    REAL8 td,
    expnCoeffs *ak
);
```

```c
void LALInspiralPhasing3_5PN(
    LALStatus *status,
    REAL8 *phase,
    REAL8 td,
    expnCoeffs *ak
);
```

```c
void LALInspiralPhasing3_6PN(
    LALStatus *status,
    REAL8 *phase,
    REAL8 td,
    expnCoeffs *ak
);
```
void LALInspiralPhasing3_7PN (  
    LALStatus *status,  
    REAL8 *phase,  
    REAL8 td,  
    expnCoeffs *ak  
)  

- **phase**: Output, the phase of the wave at the current epoch.
- **td**: Input, the PN expansion coefficients of phase $\phi_k^j$ as a function of time (cf. Table 24.2).
- **ak**: Input containing PN expansion coefficients.

**Description**

The phase of the inspiral wave corresponding to the *Approximant TaylorT2* as in Equation 24.6.

**Algorithm**

None.

**Uses**

None.

**Notes**

None.
24.6.27 Module LALInspiralTofV.c

This module outputs

\[ tofv = t - t_0 + m \int_{v_0}^{v} \frac{E'(v)}{F(v)} \, dv. \]  \hspace{1cm} (24.48)

where the constants \( t, t_0, v_0 \), and functions in the integrand \( E'(v) \) and \( F(v) \) are defined in the \textit{void} structure \texttt{params}.

Prototypes

\begin{verbatim}
void LALInspiralTofV (    LALStatus *status,
    REAL8 *tofv,
    REAL8 v,
    void *params
 )
\end{verbatim}

Description

Algorithm

Uses

LALDRombergIntegrate

Notes
24.6.28 Module LALInspiralTofVIntegrand.c

The function LALInspiralTofVIntegrand calculates the quantity \( E'(v)/F(v) \). These are the energy and flux functions which are used in the gravitational wave phasing formula.

Prototypes

```c
void LALInspiralTofVIntegrand (  
    LALStatus *status,  
    REAL8 *integrand,  
    REAL8 v,  
    void *params
)
```

Description

The function LALInspiralTofVIntegrand calculates the quantity \( E'(v)/F(v) \). These are the energy and flux functions which are used in the gravitational wave phasing formula, which is defined as

\[
\begin{align*}
    t(v) &= t_{\text{ref}} + m \int_{v}^{v_{\text{ref}}} \frac{E'(v)}{F(v)} \, dv, \\
    \phi(v) &= \phi_{\text{ref}} + 2 \int_{v}^{v_{\text{ref}}} v^3 \frac{E'(v)}{F(v)} \, dv,
\end{align*}
\]

where \( v = (\pi m F)^{1/3} \) is an invariantly defined velocity, \( F \) is the instantaneous GW frequency, and \( m \) is the total mass of the binary.

Algorithm

Uses

This function calls the function which represents \( E'(v) \) and \( F(v) \). The pointer to each of these functions is set by a call to the function LALInspiralChooseModel.

Notes

Author: Sathyaprakash, B. S.

$Id: LALInspiralTofVIntegrand.c,v 1.4 2007/06/08 14:41:49 bema Exp$
24.6.29 Module LALInspiralTiming2.c

Module used in solving the timing and phasing functions in quadrature for the Approximant TaylorT2.

Prototypes

```c
void LALInspiralTiming2_0PN (LALStatus *status, REAL8 *toff, REAL8 f, void *params);

void LALInspiralTiming2_2PN (LALStatus *status, REAL8 *toff, REAL8 f, void *params);

void LALInspiralTiming2_3PN (LALStatus *status, REAL8 *toff, REAL8 f, void *params);

void LALInspiralTiming2_4PN (LALStatus *status, REAL8 *toff, REAL8 f, void *params);

void LALInspiralTiming2_5PN (LALStatus *status, REAL8 *toff, REAL8 f, void *params);

void LALInspiralTiming2_6PN (LALStatus *status, REAL8 *toff, REAL8 f, void *params);

void LALInspiralTiming2_7PN (LALStatus *status, REAL8 *toff, REAL8 f, void *params);
```
Description

Given $t$ and $v$ this module computes the quantity

$$tofv = t - t_G - t_N(v) \sum t_k v^k,$$

(24.50)

where the coefficients $t_k$ and the Newtonian value $t_N$ are all defined in Table 24.2.

Algorithm

None

Uses

None

Notes

None
24.6.30 Module LALInspiralFrequency3.c

The code LALInspiralFrequency3.c calculates the frequency the waveform from an inspiralling binary system as a function of time up to 3.5 post-Newtonian order.

Prototypes

```c
void LALInspiralFrequency3_0PN (  
    LALStatus  *status,  
    REAL8      *frequency,  
    REAL8      td,  
    expnCoeffs *ak  
)
```

```c
void LALInspiralFrequency3_2PN (  
    LALStatus  *status,  
    REAL8      *frequency,  
    REAL8      td,  
    expnCoeffs *ak  
)
```

```c
void LALInspiralFrequency3_3PN (  
    LALStatus  *status,  
    REAL8      *frequency,  
    REAL8      td,  
    expnCoeffs *ak  
)
```

```c
void LALInspiralFrequency3_4PN (  
    LALStatus  *status,  
    REAL8      *frequency,  
    REAL8      td,  
    expnCoeffs *ak  
)
```

```c
void LALInspiralFrequency3_5PN (  
    LALStatus  *status,  
    REAL8      *frequency,  
    REAL8      td,  
    expnCoeffs *ak  
)
```

```c
void LALInspiralFrequency3_6PN (  
    LALStatus  *status,  
    REAL8      *frequency,  
    REAL8      td,  
    expnCoeffs *ak  
)
```
void LALInspiralFrequency3_7PN (  
    LALStatus *status,  
    REAL8 *frequency,  
    REAL8 td,  
    expnCoeffs *ak  
)

- frequency: Output containing the inspiral waveform.
- td: Input containing PN expansion coefficients $F_k$ (cf. Table 24.2) of frequency as a function of time.
- ak: Input containing all PN expansion coefficients.

Description
This module computes the instantaneous frequency of an inspiral wave using

$$F(t) = F_N(\theta) \sum F_k \theta^k,$$

where the expansion coefficients $F_k$, Newtonian value $F_N$ and the time-variable $\theta$ are defined in Table 24.2.

Algorithm
Uses
None.

Notes
The frequency evolution defined by post-Newtonian expansion is not monotonic. Indeed, the equations become highly inaccurate close to the last stable orbit (lso) and breakdown at or slightly after lso, and the frequency begins to decrease at later times. It turns out that the evolution is monotonic at least up to lso.
24.6.31 Module LALRungeKutta4.c

Prototypes

\[
\text{rk4GSLIntegrator} \times \text{XLALRungeKutta4Init}( \text{INT4} \ n, \ \\
\hspace{1cm} \text{rk4In} * \text{input})
\]

\[
\text{void} \ \ \ \\
\text{LALRungeKutta4}( \ \\
\hspace{1cm} \text{LALStatus} * \text{status}, \ \\
\hspace{1cm} \text{REAL8Vector} * \text{yout}, \ \\
\hspace{1cm} \text{rk4GSLIntegrator} * \text{integrator}, \ \\
\hspace{1cm} \text{void} \ * \text{params})
\]

\[
\text{void} \ \text{XLALRungeKutta4Free}( \text{rk4GSLIntegrator} * \text{integrator})
\]

- \text{n}: The number of coupled equations being integrated.
- \text{yout}: The output values for the system after the time-step.
- \text{input}: The input for the system
- \text{integrator} Required for the GSL integrator. Created using XLALRungeKutta4Init().
- \text{params} Parameters to be passed to the derivative function

Description

The code LALRungeKutta4.c solves a system of \(n\) coupled first-order differential equations. Internally, it uses the gsl routines for performing adaptive step evolution of the system, but to the outside user, it returns results for a fixed step size.

Prior to evolving a system using LALRungeKutta4(), it is necessary to create the GSL integrator using XLALRungeKutta4Init(). Once the evolution of the system has finished, this integrator should then be freed using XLALRungeKutta4Free().

Algorithm

Uses

None.

Notes
Module LALInspiralParseParameters.c

DOC IN PROGRESS
Module to work with the inspiralTemplate Structure.

Prototypes

```c
void LALInspiralITStructureParseParameters(LALStatus *status,
                                          UINT4 argc,
                                          CHAR **argv,
                                          InspiralTemplate *params)
```

```c
void LALInspiralITStructurePrint(LALStatus *status,
                                 InspiralTemplate params)
```

```c
void LALInspiralITStructureSetDefault(LALStatus *status,
                                       InspiralTemplate *params)
```

```c
void LALInspiralITStructureHelp()
```

Description

This module is a set of functions to play with the inspiralTemplate structure of the inspiral package. It allows to set default values to the inspiral structure, to parse parameters from the inspiral structure and to print the inspiral structure.

Has to check and finalized...

- The `LALInspiralITStructureParseParameters` function allows the user to parse string with respect to that structure. Each variable in the inspiralTemplate structure might be parse with a string like “–(name of the variable)+(value)” i.e. `approximant TaylorT1`. Each argument starts with a double dash character followed by a key word which is exactly as written in the InspiralTemplate Structure such as –order, –mass1, –mass2, –fCutoff ...
  Once the string is parsed, the checking function is called.

- The `LALInspiralITStructurePrint` function will print on the stdout the value of the InspiralTemplate structure.

- The `LALInspiralITStructureSetDefault` set default values to the variables. Those values are written in the C-code.

- `LALInspiralITStructureHelp`

Algorithm

None
24.6.33 Test program LALGenerateInspiralWaveform.c

Test routine for wave generation codes.

To get some help just type the name of the executable and the option –h

Basically, you can provide all the arguments from the InspiralTemplate structure such as –approximant, –order ....

References


Chapter 25

Package *noisemodels*

This is a module that generates noisemodels of various interferometers and helps run simple filtering of data simulated with these noise models. The test codes can easily be generalized to filter data from real detectors.

There are basically three test codes:

1. **NoisePSDTest**
2. **RandomSignal**
3. **FilterTest**

**NoisePSDTest:** This test code makes four successive calls to the function `LALNoiseSpectralDensity` successively passing a pointer to the functions `LALGEOPsd`, `LALLIGOIPsd`, `LALTAMAPsd` and `LALVIRGOPsd`. The function `LALNoiseSpectralDensity` returns the power spectrum in units $\text{Hz}^{-1}$ while the test code `NoisePSDTest` outputs the amplitude spectrum in units $\text{Hz}^{-1/2}$. The figure below shows the output of the test code.

![NoisePSDTest output graph](image)

**RandomSignal:** This test code makes three successive calls to `LALRandomInspiralSignal` generating (1) only signal, (2) only noise and (3) signal + noise, each time filtering the generated data with a orthogonal set of templates whose parameters are the same as the generated signal in cases (1) and (3) but arbitrary in the case of (2). The resulting correlations are shown in the diagram below:

![RandomSignal correlation graph](image)
FilterTest: This test code generates a lattice of template coordinates at a given *minimal match* and for a given total maximum mass $M_{\text{max}}$ and minimum of components masses $m_{\text{min}}$, it then filters signals with random parameters with those templates that are 'close' to the signal and records the largest overlap obtained. The generated random signal could be (1) only signal, (2) only noise or (3) noise + signal at a specified SNR. For each of the signal generated the code outputs the maximum SNR obtained together with the parameters of the signal that was generated. A typical run may output the following on screen in addition to recording the output in a file named *FilterTest.outt*.

This test code does three things:
(a) Creates a filter bank at a certain minimal match
(b) Generates signals with random parameters
(c) Filters each of these signals with the a subset of templates close to the random signal and reports the best SNR achieved.

Results of the run are written in FilterTest.out.

#Number of Coarse Bank Templates=174

<table>
<thead>
<tr>
<th>t0</th>
<th>t2</th>
<th>Overlap/SNR</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.453413e+00</td>
<td>3.519999e-01</td>
<td>9.004098e+00</td>
</tr>
<tr>
<td>9.417101e-01</td>
<td>2.336294e-01</td>
<td>6.490586e+00</td>
</tr>
<tr>
<td>1.059791e+00</td>
<td>2.202322e-01</td>
<td>8.958589e+00</td>
</tr>
<tr>
<td>1.513863e+00</td>
<td>2.598545e-01</td>
<td>7.762945e+00</td>
</tr>
<tr>
<td>9.451709e-01</td>
<td>2.087785e-01</td>
<td>6.673258e+00</td>
</tr>
<tr>
<td>7.876498e-01</td>
<td>1.785656e-01</td>
<td>8.225487e+00</td>
</tr>
<tr>
<td>2.081274e+00</td>
<td>3.178223e-01</td>
<td>7.423584e+00</td>
</tr>
<tr>
<td>1.078093e+00</td>
<td>2.220369e-01</td>
<td>7.389996e+00</td>
</tr>
<tr>
<td>5.885222e-01</td>
<td>1.601810e-01</td>
<td>9.039020e+00</td>
</tr>
<tr>
<td>2.434745e+00</td>
<td>3.493113e-01</td>
<td>9.553841e+00</td>
</tr>
</tbody>
</table>
25.1 Header LALNoiseModels.h

Header file for model noise generation codes.

Synopsis

```c
#include <lal/LALNoiseModels.h>
```

This header file covers routines that are used in synthetic background noise expected in various detectors and signals with random parameters in background noise.

Error codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>1</td>
<td>&quot;Arguments contained an unexpected null pointer&quot;</td>
</tr>
<tr>
<td>MEM</td>
<td>2</td>
<td>&quot;Memory allocation error&quot;</td>
</tr>
<tr>
<td>CHOICE</td>
<td>4</td>
<td>&quot;Invalid choice for an input parameter&quot;</td>
</tr>
<tr>
<td>DIV0</td>
<td>8</td>
<td>&quot;Division by zero&quot;</td>
</tr>
<tr>
<td>SIZE</td>
<td>16</td>
<td>&quot;Invalid input size&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants `LALNOISEMODELSH_E<name>`, and the status descriptions in `LALNOISEMODELSH_MSGE<name>`. The source code with these messages is in `LALNoiseModels.h` on line 1.82.

Structures

```c
typedef enum {
    geo,
    ligo,
    tama,
    virgo
} Detector;
```

```c
typedef struct tagInspiralWaveCorrelateIn {
    REAL8 df;
    REAL8 fCutoff;
    REAL8 samplingRate;
    REAL4Vector signal1;
    REAL4Vector signal2;
    REAL8Vector psd;
    RealFFTPlan *revp;
} InspiralWaveCorrelateIn;
```

```c
typedef struct tagAddVectorsIn {
    REAL4Vector *v1;
    REAL4Vector *v2;
    REAL8 a1;
    REAL8 a2;
} AddVectorsIn;
```
typedef struct
tagRandomInspiralSignalIn
{
  INT4 used;  /* Seed for the random number generator */
  INT4 type;  /* Type of signal required to be generated */

  REAL8 mMin; /* smallest component mass allowed */
  REAL8 mMax; /* largest component mass allowed */
  /* OR */
  REAL8 MMax; /* largest total mass allowed */
  REAL8 MMin; /* largest total mass allowed */

  REAL8 SignalAmp; /* amplitude of the signal (relevant only when type=2) */
  REAL8 NoiseAmp; /* amplitude of noise (relevant only when type=2) */
  REAL8 etaMin; /* smallest value of the symmetric mass ratio */

  InspiralTemplate
    param; /* parameter struct; user to specify certain params. */
  REAL8Vector psd; /* power spectral density used for coloring the noise */
  RealFFTPlan *fwdp; /* pre-computed fftw plan for forward fftw */
  RealFFTPlan *revp;

  /* Chirp times are needed only if param.massChoice is t02 or t03 */
  REAL8 t0Min; /* smallest Newtonian chirp time */
  REAL8 t0Max; /* largest Newtonian chirp time */
  REAL8 tnMin; /* smallest 1, 1.5 PN chirp time */
  REAL8 tnMax; /* largest 1, 1.5 PN chirp time */
  /* min/max values of BCV parameters*/
  REAL8 psi0Min; /* smallest Newtonian psi-parameter */
  REAL8 psi0Max; /* largest Newtonian psi-parameter */
  REAL8 psi3Min; /* smallest 1.5 PN psi-parameter */
  REAL8 psi3Max; /* largest 1.5 PN psi-parameter */

  INT4 coalescenceTime ;/* bin in which is maximum of the chirp (coalescence time)*/

  /* These are for spin Taylor waveforms */
  REAL8 minDistance, maxDistance;
  REAL8 spin1min, spin1max, spin2min, spin2max;
  REAL8 theta0min, theta0max, phi0min, phi0max;
  REAL8 polarisationAngleMin, polarisationAngleMax;
  REAL8 sourceThetaMin, sourceThetaMax, sourcePhiMin, sourcePhiMax;

  REAL8 inclinationMin, inclinationMax;
}
RandomInspiralSignalIn;

typedef struct
tagInspiralWaveOverlapIn
{
  INT4 nBegin;
  INT4 nEnd;
  REAL4Vector signal;
  REAL8Vector psd;
  InspiralTemplate param;
  RealFFTPlan *fwdp;
  RealFFTPlan *revp;
  UINT2 ifExtOutput; /* A flag which takes values 0 or 1 to denote
                      if an extended output consisting of filter
                      and xcorr vectors need to be filled out in
                      the call to LALInspiralWaveOverlap ( ) */
}
InspiralWaveOverlapIn;
typedef struct
tagInspiralWaveOverlapOut
{
  REAL8 max, phase, alpha;
  INT4 bin; /* bin at which max occurs */
  REAL4Vector *filter1, *filter2; /* zero and pi/2 phase templates */
  REAL4Vector *xcorr1, *xcorr2; /* cross correlation against filter 1/2 */
}
InspiralWaveOverlapOut;

datatype struct
tagInspiralFindEventsIn
{
  UINT4 currentGPSTime;
  INT4 nBegin;
  INT4 nEnd;
  REAL8 Threshold;
  REAL8 ClusterThreshold;
  REAL4Vector signal;
  REAL8Vector psd;
  InspiralTemplate param;
  RealFFTPlan *fwdp;
  RealFFTPlan *revp;
  UINT2 displayData;
  UINT2 displayPSD;
  UINT2 displayTemplates;
  UINT2 displayCorrelation;
  UINT2 displayCorrelationStats;
}
InspiralFindEventsIn;

datatype struct
tagInspiralEventsList
{
  UINT4 bin;

  INT4 endTime;
  INT4 endTimeNS;
  INT4 impulseTime;
  INT4 impulseTimeNS;
  INT4 chisqDOF;

  REAL8 amplitude;
  REAL8 effDistance;
  REAL8 phase;
  REAL8 snr;
  REAL8 sigmasq;
  REAL8 chisq;

  InspiralTemplate param;
}
InspiralEventsList;

datatype struct
tagStatsREAL4VectorOut
{
  REAL8 mean;
  REAL8 var;
  REAL8 stddev;

typedef struct
    tagInspiralChisqDataVec
    {
        REAL4Vector *SNRIntegrand;
        REAL8Vector *psd;
    }
InspiralChisqDataVec;

typedef struct
    tagInspiralChisqParams
    {
        INT4 nBins; /* number of chi-squared bins to use */
        REAL8 totalMass;
        REAL8 fLower;
        REAL8 deltaT; /* sampling interval */
    }
InspiralChisqParams;

typedef struct
    tagInspiralSNRIntegrandParams
    {
        INT4 lag; /* the value of the lag which produced the largest correlation */
        REAL8 phase; /* phase of the correlation where the max occurs */
        REAL8 deltaT; /* sampling interval */
    }
InspiralSNRIntegrandParams;
25.1.1 Module LALNoiseSpectralDensity.c

This module generates an array of size specified in the vector `psd`, that is `psd.length`. The inputs are

1. The length of the psd array required: this must be given as a non-zero positive integer by setting the `length` of the `psd` vector to the desired value;

2. Frequency resolution `df` in Hz.

3. Pointer to a function that should be used in generating the power spectral density values in units of $\text{Hz}^{-1}$. This function must necessarily be of the type: `void (*NoisePsd)(LALStatus *status, REAL8 *shf, REAL8 f)`.

Presently, there are four such functions in the `noisemodels` package. These are `LALGEOPsd`, `LALLIGOIPsd`, `LALTAMAPsd`, `LALVIRGOPsd`. These four packages return a scaled PSD while this module returns the correctly scaled version. It is assumed that new PSD modules return unscaled versions. (Note, however, that it might be better to use the scaled versions of the PSD when computing the metric on the signal manifold; this is because computing the metric involves calculation of many moments of the noise PSD and one might encounter round-off errors if un-scaled version of PSD is used; I have not checked this to be the case but suspect that there might be some problems.)

Prototypes

```c
void
LALNoiseSpectralDensity
(
  LALStatus *status,
  REAL8Vector *psd,
  void (*NoisePsd)(LALStatus *status, REAL8 *shf, REAL8 f),
  REAL8 df
)
```

Description

Algorithm

Uses

Uses a user specified pointer to a function of type

```c
void (*NoisePsd)(LALStatus *status, REAL8 *shf, REAL8 f),
```

that returns PSD values in units of $\text{Hz}^{-1}$.

Notes
25.1.2 Module LALInspiralWaveCorrelate.c

Module to compute the correlation of two data sets. Suitable only when REAL4VectorFFT is used (i.e. rfftwi, one of fftw).

Notation: The input struct has two vectors: signal1 and signal2. This module computes the correlation by shifting signal2 with respect to positive time-direction relative to signal1. Thus, if signal1 denotes the detector output in which a signal, say signal2, is present at time $t_0$, then the correlation peaks at $t_0$.

Prototypes

```c
void LALInspiralWaveCorrelate(
    LALStatus *status,
    REAL4Vector *output,
    InspiralWaveCorrelateIn corrin
)
```

Description

The module expects two inputs signal1, signal2 in the Fourier-domain, computes their correlation weighted by the noise psd, and returns the correlated output in the time-domain. More precisely, given the Discrete Fourier transform (in the notation of fftw) $H_k$ and $Q_k$ of vectors $h_k$ and $q_k$, $k = 0, \ldots, n - 1$, this module computes the inverse Fourier transform of the weighted correlation $C_k$ defined as

$$ C_k = \frac{H_kQ_k + H_{n-k}Q_{n-k}}{S_k}, \quad C_{n-k} = \frac{H_kQ_{n-k} + H_{n-k}Q_k}{S_k}, \quad k = 1, \ldots, \frac{n}{2} - 1. $$

Algorithm

Uses

LALREAL4VectorFFT

Notes
25.1.3 Module **LALInspiralWaveNormalise.c**

Module to find the norm of a signal and to return a normalised array. The original signal is left untouched.

Prototypes

```c
void LALInspiralWaveNormalise(
    LALStatus *status,
    REAL4Vector *in,
    REAL8 *norm,
    REAL8Vector psd
)
```

Description

Given the positive frequency Fourier components $H_k$, $k = 0,\ldots,n-1$, of a vector and the noise PSD $S_m$, $m = 0,\ldots,n/2$, this module first computes the norm $H$ of the vector treating $S_m$ as the measure: (note that in FFTW notation, the zeroth frequency component is $H_0$, Nyquist is $H_{n/2}$, $H_k$, $k \neq 0,n/2$, $(H_{n-k})$ is the real (imaginary) part of the $k$th harmonic)

$$H = \sum_{k=1}^{n/2-1} \frac{H_k^2 + H_{n-k}^2}{S_k}. \quad (25.1)$$

(Note that the zeroth and Nyquist components are ignored in the computation of the norm.) It then replaces the original vector $H_k$ with normalized vector using:

$$\hat{H}_k = \frac{H_k}{\sqrt{H}}, \quad k = 0,\ldots,n-1. \quad (25.2)$$

Algorithm

Uses

none.

Notes

Author: Sathyaprakash, B. S.

$Id: LALInspiralWaveNormalise.c,v 1.7 2007/12/12 18:47:51 thomas Exp$
25.1.4 Module `LALInspiralWaveNormaliseLSO.c`

Module to find the norm of a signal and to return a normaliseLSOd array. The original signal is left untouched.

Prototypes

```c
void LALInspiralWaveNormaliseLSO(
   LALStatus *status,
   REAL4Vector *filter,
   REAL8 *norm,
   InspiralWaveNormaliseIn *in
);
```

Description

Given the positive frequency Fourier components $H_k$, $k = 0, \ldots, n - 1$, of a vector and the noise PSD $S_m$, $m = 0, \ldots, n/2$, this module first computes the norm $H$ of the vector treating $S_m$ as the measure: (note that in `fftw` notation, the zeroth frequency component is $H_0$, Nyquist is $H_{n/2}$, $H_k$, $k \neq 0, n/2$, $(H_{n-k})$ is the real (imaginary) part of the $k$th harmonic)

$$H = \sum_{k=1}^{n/2-1} \frac{H_k^2 + H_{n-k}^2}{S_k}. \quad (25.3)$$

The above sum is limited to frequency `in->fCutoff`. Also, note that the zeroth and Nyquist frequency components are ignored in the computation of the norm. Moreover, array elements of `filter` corresponding to frequencies greater than `in->fCutoff` are set to zero. That is, the code replaces the original vector $H_k$ with normalized vector using:

$$\hat{H}_k = \frac{H_k}{\sqrt{H}}, \quad k \times \text{in} \rightarrow \text{df} \leq \text{in} \rightarrow \text{fCutoff},$$

$$= 0, \quad k \times \text{in} \rightarrow \text{df} > \text{in} \rightarrow \text{fCutoff}. \quad (25.4)$$

In addition, the 0th and Nyquist frequency components are also set to zero.

Algorithm

Uses

none.

Notes
25.1.5 Module **LALEGOPsd.c**

Module to calculate the noise power spectral density for the EGO detector.

**Prototypes**

```c
void LALEGOPsd (LALStatus *status, REAL8 *psd, REAL8 f)
```

**Description**

The module takes as an input a frequency \( f \) in Hz, and it calculates the noise spectral density (per Hz) \( S_h(f) \) for that frequency. The noise PSD is based on data provided by grqc/0607092

\[
S_h(f) = S_0 \left\{ x^{p_1} + a_1 x^{p_2} + \frac{1 + b_1 x + b_2 x^2 + b_3 x^3 + b_4 x^4 + b_5 x^5 + b_6 x^6}{1 + c_1 x + c_2 x^2 + c_3 x^3 + c_4 x^4} \right\}
\]

(25.5)

where \( S_0 = 1.61 e^{-51} \)

\( p_1 = -4.05, p_2 = -0.69 \)

\( a_1 = 185.62, a_2 = 232.56 \)

\( b_1 = 31.18, b_2 = -64.72, b_3 = 52.24, b_4 = -42.16, b_5 = 10.17, b_6 = 11.53 \)

and \( c_1 = 13.58, c_2 = -36.46, c_3 = 18.56, c_4 = 27.43 \)

**Algorithm**

**Uses**

None.

**Notes**

**Author:** Cokelaer T.  
**$Id: LALEGOPsd.c,v 1.5 2007/06/08 14:41:50 bema Exp$**
25.1.6 Module **LALGEOPsd.c**

Module to calculate the expected noise power spectral density for the GEO600 detector.

Prototypes

```
void LALGEOPsd(LALStatus *status, REAL8 *psd, REAL8 f)
```

Description

The module takes as an input a frequency \( f \) in Hz, and it calculates the noise spectral density (per Hz) \( S_h(f) \) for that frequency. The noise PSD is based on data provided by J. Hough and G. Cagnoli (see T. Damour, B.R. Iyer and B.S. Sathyaprakash, Phys. Rev. D 63, 044023 (2001)) and is approximated by the following:

\[
S_h(f) = 10^{-16} \left( \frac{f}{f_0} \right)^{-30} + 34 \frac{f_0}{f} + \frac{20 \left[ 1 - (f/f_0)^2 + 0.5(f/f_0)^4 \right]}{1 + 0.5(f/f_0)^2}
\]  

(25.6)

The returned value is scaled up by \( s_0 = 10^{46} \). In otherwords, the expected noise PSD is a factor \( 10^{46} \) lower.

Algorithm

Uses

None.

Notes
25.1.7 Module LALAdvLIGOPsd.c

Module to calculate the noise power spectral density for the initial LIGO detector.

Prototypes

```c
void LALAdvLIGOPsd (LALStatus *status, REAL8 *psd, REAL8 f)
```

Description

The module takes as an input a frequency \( f \) in Hz, and it calculates the noise spectral density (per Hz) \( S_h(f) \) for that frequency. The noise PSD is based on data provided by Kip Thorne, and the fit by B.S.Sathyaprakash

\[
S_h(f) = S_0 \left\{ \left( \frac{f}{f_0} \right)^{-4.34} - 5 \left( \frac{f_0}{f} \right)^2 + 111 \left( \frac{1 - \left( \frac{f}{f_0} \right)^2 + 0.5 \left( \frac{f}{f_0} \right)^4}{1 + 0.5 \left( \frac{f}{f_0} \right)^2} \right) \right\};
\]

where, \( f_0 = 215 \text{Hz} \) The returned value is scaled up by \( S_0 = 10^{49} \).

Algorithm

Uses

None.

Notes
25.1.8 Module **LALLIGOIPsd.c**

Module to calculate the noise power spectral density for the initial LIGO detector.

**Prototypes**

```c
void LALLIGOIPsd (LALStatus *status, REAL8 *psd, REAL8 f)
```

**Description**

The module takes as an input a frequency \( f \) in Hz, and it calculates the noise spectral density (per Hz) \( S_h(f) \) for that frequency. The noise PSD is based on data provided by K. Blackburn (see T. Damour, B.R. Iyer and B.S. Sathyaprakash, Phys. Rev. D 63, 044023 (2001)) and is approximated by the following:

\[
S_h(f) = \left( \frac{4.49 f}{f_0} \right)^{-56} + 0.16 \left( \frac{f}{f_0} \right)^{-4.52} + 0.52 + 0.32 \left( \frac{f}{f_0} \right)^2
\]  

(25.8)

The returned value is scaled up by \( s_0 = 10^{46} / 9 \). In otherwords, the expected noise PSD is \( 9 \times 10^{-46} \) times the returned value.

**Algorithm**

**Uses**

None.

**Notes**
25.1.9 Module **LALTAMAPsd.c**

Module to calculate the noise power spectral density for the TAMA detector.

**Prototypes**

```c
void LALTAMAPsd(LALStatus *status, REAL8 *psd, REAL8 f)
```

**Description**

The module takes as an input a frequency \( f \) in Hz, and it calculates the noise spectral density (per Hz) \( S_h(f) \) for that frequency. The noise PSD is based on data provided by M.-K Fujimoto (see T. Damour, B.R. Iyer and B.S. Sathyaprakash, Phys. Rev. D 63, 044023 (2001)) and is approximated by the following:

\[
S_h(f) = \left( \frac{f}{f_0} \right)^{-5} + 13 \frac{f_0}{f} + 9 \left[ 1 + \left( \frac{f}{f_0} \right)^2 \right].
\]  

The returned value is scaled up by \( s_0 = 10^{46}/75 \). In other words, the expected noise PSD is \( 75 \times 10^{-46} \) times the returned value.

**Algorithm**

**Uses**
None.

**Notes**

Author: Sathyaprakash, B. S.

$Id: LALTAMAPsd.c,v 1.9 2007/06/08 14:41:50 bema Exp $
Module LALVIRGOPsd.c

Module to calculate the noise power spectral density for the VIRGO detector.

Prototypes

```c
void LALVIRGOPsd (LALStatus *status, REAL8 *psd, REAL8 f)
```

Description

The module takes as an input a frequency \( f \) in Hz, and it calculates the noise spectral density (per Hz) \( S_h(f) \) for that frequency. The noise PSD is based on data provided by J-Y. Vinet and is approximated by the following:

\[
S_h(f) = s_0 \left( \frac{7.87f}{f_0} \right)^{-4.8} + \frac{6}{17} \frac{f_0}{f} + \left[ 1 + \left( \frac{f}{f_0} \right)^2 \right],
\]

where \( s_0 = 10.2e^{-46} \)

Algorithm

Uses

None.

Notes
25.1.11 Module `LALRandomInspiralSignal.c`

Module to generate
(a) inspiral signals with random masses or chirp times that have values within the parameter space specified by an input struct,
(b) simulated Gaussian noise of PSD expected in a given interferometer
(c) inspiral signal as in (a) but of a specified amplitude added to simulated Gaussian noise as in (b).
In all cases the returned vector is the Fourier transform of the relevant signal.

Prototypes

```c
void LALRandomInspiralSignal
(  
  LALStatus *status,
  REAL4Vector *signal,
  RandomInspiralSignalIn *randIn
)
```

Description

The function receives input struct of type `RandomInspiralSignalIn` whose members are

```c
typedef struct
  tagRandomInspiralSignalIn
{
  INT4 useed;
  INT4 type;

  REAL8 mMin;
  REAL8 mMax;
  REAL8 MMax;
  REAL8 SignalAmp;
  REAL8 NoiseAmp;
  REAL8 etaMin;
  REAL8 t0Min;
  REAL8 t0Max;
  REAL8 tnMin;
  REAL8 tnMax;

  InspiralTemplate param;
 REAL8Vector psd;
  RealFFTPlan *fwdp;
} RandomInspiralSignalIn;
```

Depending on the value of the parameter (`randIn.type`) this code returns the Fourier transform of
(a) a pure inspiral signal of a given type (`randIn.type=0`),
(b) simulated noise expected in a chosen interferometer `randIn.type=1` or
(c) `SignalAmp`×s+`NoiseAmp`×n (`randIn.type=2`), where s is normalised signal and n random Gaussian noise whose PSD is that expected in a given interferometer with zero mean and unit rms.

User must specify the following quantities in the input structure When repeatedly called, the parameters of the signal will be uniformly distributed in the space of
(a) component masses in the range `[randIn.mMin, randIn.mMax]` if `param.massChoice=m1Andm2`,
(b) component masses greater than `randIn.mMin` and total mass less than `randIn.MMax` if `param.massChoice=totalMassAndEta`,
(c) component masses greater than `randIn.mMin` and uniform total mass less than `randIn.MMax` if `param.massChoice=totalMassUAndEta`,
(d) Newtonian and first post-Newtonian chirp times if `param.massChoice=t02`,
(e) Newtonian and 1.5 post-Newtonian chirp times if `param.massChoice=t03` and.
### Table 25.1: Input structure needed for the function LALRandomInspiralSignal

<table>
<thead>
<tr>
<th>Parameter</th>
<th>i/o</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT4 useed</td>
<td>input</td>
<td>Seed for the random number generator</td>
</tr>
<tr>
<td>INT4 type</td>
<td>input</td>
<td>Type of signal required to be generated</td>
</tr>
<tr>
<td>InspiralTemplate p</td>
<td>i/o</td>
<td>user must input certain params; others will be output</td>
</tr>
<tr>
<td>p.startTime</td>
<td></td>
<td>usually 0.</td>
</tr>
<tr>
<td>p.startPhase</td>
<td></td>
<td>[0, π/2]</td>
</tr>
<tr>
<td>p.nStartPad</td>
<td></td>
<td>number of zeros in the vector before the signal begins</td>
</tr>
<tr>
<td>p.nEndPad</td>
<td></td>
<td>number of zeros in the vector after the signal ends</td>
</tr>
<tr>
<td>p.signalAmplitude</td>
<td></td>
<td>1 for comparable mass systems 0 for test mass model</td>
</tr>
<tr>
<td>p.ieta</td>
<td></td>
<td>1 for comparable mass systems 0 for test mass model</td>
</tr>
<tr>
<td>p.fLower</td>
<td></td>
<td>lower frequency cutoff in Hz</td>
</tr>
<tr>
<td>p.fcutoff</td>
<td></td>
<td>upper frequency cutoff in Hz</td>
</tr>
<tr>
<td>p.tSampling</td>
<td></td>
<td>sampling rate in Hz</td>
</tr>
<tr>
<td>p.order</td>
<td></td>
<td>order of the PN approximant of the signal</td>
</tr>
<tr>
<td>p.approximant</td>
<td></td>
<td>PN approximation to be used for inspiral signal generation</td>
</tr>
<tr>
<td>InputMasses massChoice</td>
<td>input</td>
<td>space in which parameters are chosen: m1AndM2, totalMassAndEta, totalMassUAnd</td>
</tr>
<tr>
<td>REAL8Vector psd</td>
<td>input</td>
<td>pre-computed power spectral density used for coloring the noise</td>
</tr>
<tr>
<td>RealFFTPlan *fwdp</td>
<td>input</td>
<td>pre-computed fftw plan to compute forward Fourier transform</td>
</tr>
<tr>
<td>REAL8 mMin</td>
<td>input</td>
<td>smallest component mass allowed</td>
</tr>
<tr>
<td>REAL8 mMax</td>
<td>input</td>
<td>largest component mass allowed OR</td>
</tr>
<tr>
<td>REAL8 MMax</td>
<td>input</td>
<td>largest total mass allowed</td>
</tr>
<tr>
<td>REAL8 SignalAmp</td>
<td>input</td>
<td>amplitude of the signal (relevant only when type=2)</td>
</tr>
<tr>
<td>REAL8 NoiseAmp</td>
<td>input</td>
<td>amplitude of noise (relevant only when type=2)</td>
</tr>
<tr>
<td>REAL8 etaMin</td>
<td>input</td>
<td>smallest value of the symmetric mass ratio</td>
</tr>
</tbody>
</table>

Following chirp times are needed only if param.massChoice is t02 or t03

| REAL8 t0Min               | input | smallest Newtonian chirp time                                           |
| REAL8 t0Max               | input | largest Newtonian chirp time                                            |
| REAL8 tnMin               | input | smallest 1 chirp time if param.massChoice=t02                           |
|                            |     | smallest 1.5 chirp time if param.massChoice=t02                         |
| REAL8 tnMax               | input | largest 1 chirp time if param.massChoice=t02                            |
|                            |     | largest 1.5 chirp time if param.massChoice=t02                         |
(f) component masses in the range \([\text{randIn.mMin}, \text{randIn.mMax}]\) one of them being a neutron star and the other a black hole (one above 3 solar mass and one below) if \text{param.massChoice=bhns}. The function therefore checks the mass range validity i.e. \text{randIn.mMin} must be less than 3 and \text{randIn.mMax} greater than 3.

**Algorithm**

No special algorithm, only a series of calls to pre-existing functions.

**Uses**

- \text{random}
- \text{LALInspiralParameterCalc}
- \text{LALInspiralWave}
- \text{LALREAL4VectorFFT}
- \text{LALInspiralWaveNormaliseLSO}
- \text{LALCreateRandomParams}
- \text{LALNormalDeviates}
- \text{LALDestroyRandomParams}
- \text{LALREAL4VectorFFT}
- \text{LALColoredNoise}
- \text{LALAddVectors}

**Notes**
25.1.12 Module LALColoredNoise.c
This module colors a given white noise input into a colored noise of power spectral density \( \text{psd} \).

Prototypes

```c
void LALColoredNoise(
    LALStatus *status,
    REAL4Vector *noisy,
    REAL8Vector psd
);
```

Description

Given the Fourier transform \( N(f) \) of white noise, the Fourier transform of noise of power spectral density \( S(f) \) is given by \( N(f) = N(f) \times \sqrt{S(f)} \). In the discrete version there is an additional normalisation:

\[
N_k = N_k \times \sqrt{\frac{2S_k}{n}}, \quad N_{n-k} = N_{n-k} \times \sqrt{\frac{2S_k}{n}}, \quad k = 1, \ldots, \frac{n}{2}.
\]

Algorithm

Uses

none

Notes
25.1.13 Module LALAddVectors.c

Module to add two vectors with weights.

Prototypes

```c
void LALAddVectors
    (LALStatus *status,
     REAL4Vector *vector,
     AddVectorsIn in
    )
```

Description

Given weights $A_1$ and $A_2$ as in `AddVectorsIn` and vectors $v_1$ and $v_2$ this code returns vector $v$ given by

$$v[i] = A_1 \, v_1[i] + A_2 \, v_2[i];$$

Algorithm

Uses

none

Notes
25.1.14 Module LALInspiralWaveOverlap.c

Module to compute the overlap of a given data set with two orthogonal inspiral signals of specified parameters with a weight specified in a psd array. The code also returns in a parameter structure the maximum of the overlap, the bin where the maximum occurred and the phase at the maximum.

Prototypes

```c
void LALInspiralWaveOverlap(
    LALStatus *status,
    REAL4Vector *output,
    InspiralWaveOverlapOut *overlapout,
    InspiralWaveOverlapIn *overlapin
);
```

Description

Algorithm

Uses

LALInspiralWave
LALREAL4VectorFFT
LALInspiralWaveNormaliseLSO
LALInspiralWaveCorrelate

Notes
25.1.15 Module **LALInspiralFindEvents.c**

Module to find events in a given data set with an SNR larger than a pre-specified threshold. The module uses two orthogonal inspiral signals of specified parameters with a weight specified in a psd array. The code returns the number of events found, and for each event the snr, the bin number and the phase of the template at that bin.

Prototypes

```c
void LALInspiralFindEvents(
    LALStatus *status,
    INT4 *nEvents,
    InspiralEventsList **eventlist,
    InspiralFindEventsIn *findeventsin
)
```

Description

Algorithm

Uses

- LALInspiralWave
- LALREAL4VectorFFT
- LALInspiralWaveNormalise
- LALInspiralWaveCorrelate

Notes

Author: Sathyaprakash, B. S.

$Id: LALInspiralFindEvents.c,v 1.12 2007/06/08 14:41:50 bema Exp $
25.1.16 Module LALInspiralFindLoudestEvent.c

Module to find events in a given data set with an SNR larger than a pre-specified threshold. The module uses two orthogonal inspiral signals of specified parameters with a weight specified in a psd array. The code returns the number of events found, and for each event the snr, the bin number and the phase of the template at that bin.

Prototypes

```c
void LALInspiralFindLoudestEvent(
    LALStatus *status,
    INT4 *nEvents,
    InspiralEventsList *eventlist,
    InspiralFindEventsIn *findeventsin
);
```

Description

Algorithm

Uses

- LALInspiralWave
- LALREAL4VectorFFT
- LALInspiralWaveNormalise
- LALInspiralWaveCorrelate

Notes
25.1.17 Module LALInspiralFindEventsCluster.c

Module to find all ‘distinct’ events in a given data set with an SNR larger than a pre-specified threshold. To select ‘distinct’ events the code uses two thresholds; all events crossing a first \textit{lower} threshold are clustered. If at least one of the points in the clustered event crosses a second \textit{higher} threshold then the clustered event is a trigger and what is recorded is the loudest point in the clustered event and the number of points above the first threshold in the cluster.

The module uses two orthogonal inspiral signals of specified parameters with a weight specified in a psd array. The code returns the number of events found, and for each event the snr, the bin number and the phase of the template at that bin.

Prototypes

```c
void LALInspiralFindEventsCluster(
    LALStatus    *status,
    INT4         *nEvents,
    InspiralEventsList **eventlist,
    InspiralFindEventsIn *findeventsin
)
```

Description

Algorithm

Uses

- LALInspiralWave
- LALREAL4VectorFFT
- LALInspiralWaveNormaliseLSO
- LALInspiralWaveCorrelate

Notes

Author: Sathyaprakash, B. S.

$Id: LALInspiralFindEventsCluster.c,v 1.6 2007/06/08 14:41:50 bema Exp $
25.1.18 Module LALStatsREAL4Vector.c

Module to compute the mean, rms, minimum and maximum of a REAL4Vector.

Prototypes

```c
void LALStatsREAL4Vector
    (LALStatus *status,
     StatsREAL4VectorOut *out,
     REAL4Vector         *vector)
```

Description

Algorithm

Uses

none

Notes
25.1.19 Module LALInspiralComputeChisq.c

Prototypes

```c
void LALInspiralComputeChisq(
    LALStatus   *status,
    REAL4       *chisq,
    InspiralChisqDataVec  *input,
    InspiralChisqParams   *params
);
```

Description

Algorithm

Uses

LALREAL4VectorFFT

Notes

Author: Sathyaprakash, B. S.

$Id: LALInspiralComputeChisq.c,v 1.7 2007/06/08 14:41:50 bema Exp$
25.1.20 Module LALInspiralComputeSNRIntegrand.c

Prototypes

void LALInspiralComputeSNRIntegrand
   (LALStatus *status,
    REAL4Vector *output,
    InspiralWaveCorrelateIn corrin,
    InspiralSNRIntegrandParams *params)

Description

Algorithm

Uses

LALREAL4VectorFFT

Notes
25.1.21 Program BankEfficiency.c

Test code for the inspiral bank modules.

Usage

BankEfficiency [options]

The options are:
- \(-\alpha\) : BCV amplitude correction parameter
- \(-\text{approximant}\) : Post-Newtonian model such as TaylorT1, Pad\(\text{eT1}\), EOB, BCV ...
- \(-f_1\) : lower frequency cutoff
- \(-m_{\text{Min}}\) : minimal mass of component stars
- \(-m_{\text{Max}}\) : maximal mass of component stars
- \(-mm\) : minimal match for template bank
- \(-n\) : number of trials
- \(-\text{order}\) : order of PN model
- \(-\text{quiet}\) : if this flag is present, the output is restricted to the min
- \(-\text{seed}\) : seed for random generation
- \(-\text{sigAmp}\) : amplitude of the signal
- \(-\text{simType}\) : type of simulation, 0, 1 or 2
- \(-x_{\text{OMax}}\) : Max value of \(\psi_0\)
- \(-x_{\text{OMin}}\) : Min value of \(\psi\)

Description

This test code gives an example of how one might generate inspiral waveforms and use them to compute the overlap of a random signal (with or without simulated noise) of a certain strength. The parameter Note that one must calculate the length of the waveform and allocate memory for it before calling \texttt{InspiralWave}. The length of the waveform can be calculated by calling the function \texttt{InspiralWaveLength} beforehand, as shown.

There are only two functions which one can call to generate waveforms. These are \texttt{InspiralWave}, which will return a single waveform, and \texttt{InspiralWaveTemplates}, which returns a pair of waveforms which have phases which differ by \(\pi/2\).

Exit codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
</table>

The status codes in the table above are stored in the constants \_E<name>, and the status descriptions in \_MSGE<name>. The source code with these messages is in \texttt{FilterTest.c} on line 1.98.

Uses

This code directly uses the following functions (see those functions to find out what they call in turn):

\texttt{LALInspiralWaveLength}
\texttt{LALInspiralCreateCoarseBank}
\texttt{LALRandomInspiralSignal}
\texttt{LALInspiralParameterCalc}
\texttt{LALNoiseSpectralDensity}
\texttt{LALCreateForwardRealFFTPlan}
\texttt{LALCreateReverseRealFFTPlan}
\texttt{LALForwardRealFFT}
\texttt{LALReverseRealFFT}
\texttt{LALDestroyRealFFTPlan}
\texttt{LALInspiralWaveOverlap}
\texttt{LALInspiralParameterCalc}
Notes
25.1.22 Program RandomInspiralSignalTest.c

Test code for the inspiral wave generation and noisemodels modules.

Usage

RandomInspiralSignalTest

Description

This test code gives an example of how one might generate an inspiral waveform and compute its overlap with simulated detector noise (with or without the signal present). Recently, the code was modified so that users can output either the time-domain signal, noise, or signal+noise and not the correlated output. This is done via the variable TimeDomain which controls the type of output If TimeDomain=0 then the code filters the random signal with a template of the same parameters and outputs the results of the correlation. If TimeDomain=1 then the code outputs the time-domain signal/noise/signal+noise

The parameter randIn.type=0 generates only signal randIn.type=1 generates only noise randIn.type=2 generates randIn.SignalAmp * signal(t) + randIn.NoiseAmp * noise(t) Note that one must calculate the length of the waveform and allocate memory for it before calling InspiralWave. The length of the waveform can be calculated by calling the function InspiralWaveLength beforehand, as shown.

There are only two functions which one can call to generate waveforms. These are InspiralWave, which will return a single waveform, and InspiralWaveTemplates, which returns a pair of waveforms which have phases that differ by \( \pi/2 \).

Exit codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
</table>

The status codes in the table above are stored in the constants _E<name>_, and the status descriptions in _MSGE<name>_. The source code with these messages is in RandomInspiralSignalTest.c on line 1.89.

Uses

This code directly uses the following functions and macros (see those functions to find out what they call in turn):

lalDebugLevel
LALCreateForwardRealFFTPlan
LALCreateReverseRealFFTPlan
LALInspiralParameterCalc
LALInspiralWaveLength
LALInspiralWaveOverlap
LALInspiralFindEventsCluster
LALNoiseSpectralDensity
LALRandomInspiralSignal

Notes
25.1.23 Program **NoisePSDTest.c**

This program can be used generate expected noise NoiseSpectralDensity in various interferometers. See the beginning of the NoiseModels module to see details on how this test program works.

Prototypes

```c
int main ( void )
```

Description

Algorithm

Uses

- `LALDCreateVector`
- `LALNoiseSpectralDensity`
- `LALGEOPsd`
- `LALLIGOIPsd`
- `LALTAMAPsd`
- `LALVIRGOPsd`
- `LALAdvLIGOPsd`
- `LALDDestroyVector`
- `LALCheckMemoryLeaks`

Notes
Section 9

Pulsar Packages
Chapter 26

Package houghpulsar: The Hough transform

Alicia M. Sintes, M. Alessandra Papa., Krishnan, B.

The hierarchical Hough transform search strategy is an efficient and highly parallel computer algorithm [1] [2]. This package provides the necessary routines for the Hough incoherent search (the second stage of the search) to track the frequency evolution of peaks in the spectra, which is known in advance for some choice of the source parameters.

The Hough transform is a transformation between the data and the space of parameters that describe the signal.

Input: Set of points in time-frequency plane that have been obtained from the demodulated FT. The demodulation has been performed with respect to a certain sky location and certain spin-down parameters, in such a way that, if a source was located in the center of the patch and having the same spin-down parameters for which it has been demodulated, we will observe a set of points forming a horizontal line at the intrinsic frequency of the source $f_0$. Due to the mismatch between the source parameters and the demodulated parameters we will observe a certain pattern in the time frequency diagram following the Hough transform master equation.

Output: Histograms in the parameter space: for each $f_0$, residual spin-down and refined sky location.

The principal is the following: for every point in the time-frequency plane, we enhance the number count in the histogram in the pixels that are consistent. At the end of this procedure, significant clustering in a pixel in parameter space indicates likelihood of the data containing a signal having the parameters of that pixel.

The package is organized under the headers LUT.h PHMD.h HoughMap.h LALHough.h and the modules PatchGrid.c, Stereographic.c, ParamPLUT.c, NDParamPLUT.c,ConstructPLUT.c, Peak2PHMD.c, HoughMap.c, and DriveHough.c.

Acknowledgment

The authors thank S. Frasca and C. Palomba for helpful discussions, F. Massaioli for helping in the initial stages of code development, and B. Allen and J. Creighton for their valuable implementation advice.
26.1 Header \texttt{LUT.h}

Provides structures and functions required for the construction of look up tables (LUT) that are the core for building the Hough maps.

Synopsis

\texttt{#include <lal/LUT.h>}

Our goal is the construction of Hough maps. In order to produce them efficiently, the present implementation makes use of LUTs. Here we provide the necessary routines for their construction and use.

In principle, the subroutines provided are valid for any Hough master equation of the form:

$$
\nu - F_0 = \vec{\xi}(t) \cdot (\hat{n} - \hat{N}),
$$

where $\nu$ is the measured frequency of the signal at time $t$, $F_0$ intrinsic frequency of the signal at that time, $\hat{n}$ location of the source in the sky, $\hat{N}$ the center of the sky patch used in the demodulation procedure, and $\vec{\xi}(t)$ any vector.

The form of this vector $\vec{\xi}(t)$ depends on the demodulation procedure used in the previous step. In our case this corresponds to

$$
\vec{\xi}(t) = \left( F_0 + \sum_k F_k [\Delta T]^k \right) \frac{\vec{v}(t)}{c} + \left( \sum_k kF_k [\Delta T]^{k-1} \right) \frac{\vec{x}(t) - \vec{x}(t_0)}{c},
$$

and

$$
F_0 \equiv f_0 + \sum_k \Delta f_k [\Delta T]^k,
$$

where $\vec{v}(t)$ is the velocity of the detector, $\vec{x}(t)$ is the detector position, $T_N(t)$ is the time at the solar system barycenter (for a given sky location $\hat{N}$), $\Delta T \equiv T_N(t) - T_N(t_0)$, $\Delta f_k = f_k - F_k$ the residual spin-down parameter, $F_k$ the spin-down parameter used in the demodulation, and $f_0$, $f_k$ the intrinsic frequency and spin-down parameters of the source at time $t_0$.

Looking at the generic Hough master equation, one realizes that for a fixed time, a given value of $F_0$, and a measured frequency $\nu$ (from a selected peak), the source could be located anywhere on a circle (whose center points in the same direction of $\vec{\xi}(t)$ and is characterized by $\phi$, the angle between $\hat{n}$ and $\vec{\xi}$). Since the Hough transform is performed on a set of spectra with discrete frequencies, a peak on the spectrum appearing at $\nu$ could correspond to any source with a demodulated frequency in a certain interval. As a consequence, the location of the sources compatible with $F_0$ and $\nu$ is not a circle but an annulus with a certain width.

Our purpose is to map these annuli on a discrete space. An estimation of the average thickness of the annuli tells us that the vast majority of annuli will be very thin, and therefore our algorithm should not be optimized for drawing thick annuli but for thin ones. Also, the mapping implementation should be one with a uniform probability distribution in order to avoid discretization errors. In order to remove border effects, we use a biunivocal mapping, which requires that a pixel in a partial Hough map can belong only to one annulus, just touched by one peak of the spectrum. The criteria for the biunivocal mapping is that if and only if the center of the pixel is inside the annulus, then the pixel will be enhanced.

In order to simplify (reduce the computational cost of) this task we construct look up tables (LUT) where the borders of these annuli are marked for any possible $\nu - F_0$ value. Since we work on a discrete space these LUT are valid for many $F_0$ values.

At this point we have already chosen a sky tiling to produce the Hough map efficiently. It consists of changing coordinates so that the center of the patch is located at $(0, -\pi/2)$ in $(\alpha - \delta)$ (or in any other coordinate system), then we make use of the stereographic projection and we take horizontal and vertical lines on the projected plane at constant space separation. This projection is advantageous because it avoids distortions, i.e. the pixel size is almost constant independently of the sky location, and makes the algorithm simpler. The stereographic projection has the property to map circles on the celestial sphere to circles on the projected plane.
Figure 26.1: Stereographic projection. [Credit to: D.Hilbert, S.Cohn-Vossen, P. Nemenyi, “Geometry and Imagination”, Chelsea Publishing Company, New York 1952.]

Error conditions

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>1</td>
<td>&quot;Null pointer&quot;</td>
</tr>
<tr>
<td>SIZE</td>
<td>2</td>
<td>&quot;Invalid input size&quot;</td>
</tr>
<tr>
<td>SZMM</td>
<td>4</td>
<td>&quot;Size mismatch&quot;</td>
</tr>
<tr>
<td>INT</td>
<td>6</td>
<td>&quot;Invalid interval&quot;</td>
</tr>
<tr>
<td>SAME</td>
<td>8</td>
<td>&quot;Input/Output data vectors are the same&quot;</td>
</tr>
<tr>
<td>FREQ</td>
<td>10</td>
<td>&quot;Invalid frequency&quot;</td>
</tr>
<tr>
<td>VAL</td>
<td>12</td>
<td>&quot;Invalid value&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants LUTH_E<name>, and the status descriptions in LUTH_MSGE<name>. The source code with these messages is in LUT.h on line 1.628.

Constant declarations

The initial settings at compile time are the following:

**LINERR 0.001:** Maximum “error”, in terms of the width of the thinnest annulus, which allows one to represent/approximate a circle by a line.

**PIXERR 0.1:** Maximum “error”, in terms of the width of the thinnest annulus, which allows one to consider two border equivalents. It is only relevant for determining the LUT frequency range validity.

**PIXELFACTORX 2:** Width of the thinnest annulus in terms of pixels (x-direction).

**PIXELFACTORY 2:** Width of the thinnest annulus in terms of pixels (y-direction).
VEPI $1.0e-06$: Earth epicycle velocity divided by the light velocity $c$. TO BE CHANGED DEPENDING ON THE DETECTOR.

VTOT $1.0e-04$: Earth total velocity divided by $c$. TO BE CHANGED DEPENDING ON THE DETECTOR.

SIDEY ($100\times$ PIXELFACTORY): The maximum number of pixels in a given patch is $\text{SIDEY} \times \text{SIDEY}$. These numbers should be integers. They come from assuming that: the ‘longitudinal’ patch size is $dF \times 10^6/f_0$, where $10^6 = c/v_{epicycle}$, the thinnest possible annulus is $dF \times 10^4/f_0$, where $10^4 = c/v_{tot}$. Therefore, the ratio (longitudinal patch size)/(width thinnest annuli) = 100 (100 TO BE CHANGED DEPENDING ON THE DETECTOR).

MAX_N_BINS 150: Maximum number of frequency bins that can affect a patch.

MAX_N_BORDERS 208: Maximum number of borders in a patch.

Note several of these constants will be removed in a next release. They are used only for tiling the sky-patch or as default parameters in the test codes.

Structures and type definitions

```c
typedef UCHAR COORType
```

To be changed to `INT2 COORType` if the number of pixels in the x-direction exceeds 255.

```c
struct HOUGHBorder
```

This structure stores the border of a circle clipped on the projected plane. The fields are:

- INT4 yUpper: Upper y pixel index affected by this border.
- INT4 yLower: Lower y pixel index affected by this border. $y_{\text{Upper}} < y_{\text{Lower}}$ or $y_{\text{Upper}} < 0$ are possible.
- INT4 yCenter: y pixel value of the center of the circle.
- UINT2 ySide: Length of xPixel.
- COORType *xPixel: x pixel index of the border to be marked.

```c
struct HOUGHBin2Border
```

This structure stores the border indexes corresponding to one frequency bin plus the corrections to be added to the first column of the patch. The fields are:

- INT2 leftB1: Border index to be used (start-border ‘+1’).
- INT2 rightB1: Border index to be used (stop-border ‘-1’).
- INT2 leftB2: Border index to be used (start-border ‘+1’).
- INT2 rightB2: Border index to be used (stop-border ‘-1’).
- INT2 piece1max: Interval limits of the (first piece) correction to the first column.
- INT2 piece1min: If $\text{piece1min} > \text{piece1max}$ no corrections should be added.
- INT2 piece2max: Interval limits of the (second piece) correction to the first column.
- INT2 piece2min: If $\text{piece2min} > \text{piece2max}$ no corrections should be added.

```c
struct HOUGHptfLUT
```
This structure stores the patch-time-frequency *look up table*. The fields are:

- **INT2 timeIndex**  Time index of the LUT.
- **INT8 f0Bin**  Frequency bin for which it has been constructed.
- **REAL8 deltaF**  Frequency resolution \( df = 1/\text{TCOH} \), where \( 1/\text{TCOH} \) is the coherent integration time used in the demodulation procedure.
- **INT4 nFreqValid**  Number of frequencies where the LUT is valid.
- **INT4 iniBin**  First bin affecting the patch with respect to \( f_0 \).
- **INT4 nBin**  Exact number of bins affecting the patch.
- **INT4 offset**  Frequency bin corresponding to center of patch measured with respect to \( f_0\text{Bin} \) (zero in modulated case)
- **UINT2 maxNBins**  Maximum number of bins affecting the patch (for memory allocation purposes).
- **UINT2 maxNBorders**  Maximum number of borders affecting the patch (for memory allocation purposes).
- **HOUGHBorder *border**  The annulus borders.
- **HOUGHBin2Border *bin**  Bin to border correspondence.

**struct HOUGHPatchGrid**

This structure stores patch-frequency *grid* information. The fields are:

- **REAL8 f0**  Frequency to construct grid.
- **REAL8 deltaF**  Frequency resolution: \( df = 1/\text{TCOH} \).
- **REAL8 patchSizeX**  Patch size in radians along x-axis.
- **REAL8 patchSizeY**  Patch size in radians along y-axis.
- **REAL8 minWidthRatio**  Ratio between the minimum annulus width for this search, and the minimum annulus width for 1 year integration time. This value should be in the interval [1.0, 25.0].
- **REAL8 deltaX**  Longitudinal space resolution, x-direction.
- **REAL8 xMin**  Patch limit, as the coordinate of the center of the first pixel.
- **REAL8 xMax**  Patch limit, as the coordinate of the center of the last pixel.
- **UINT2 xSide**  Real number of pixels in the x direction (in the projected plane). It should be less than or equal to \( \text{xSideMax} \).
- **UINT2 xSideMax**  Maximum number of pixels in the x direction (for memory allocation), i.e. length of \( \text{xCoor} \).
- **REAL8 *xCoor**  Coordinates of the pixel centers.
- **REAL8 deltaY**  Longitudinal space resolution, y-direction.
- **REAL8 yMin**  Patch limit, as center of the first pixel.
- **REAL8 yMax**  Patch limit, as center of the last pixel.
- **UINT2 ySide**  Real number of pixels in the y-direction (in the projected plane). It should be less than or equal to \( \text{ySideMax} \).
- **UINT2 ySideMax**  Maximum number of pixels in the y direction (for memory allocation), i.e. length of \( \text{yCoor} \).
- **REAL8 *yCoor**  Coordinates of the pixel centers.
struct HOUGHResolutionPar

This structure holds the parameters needed for gridding the patch. The fields are:

REAL8 f0  Frequency at which construct the grid.
REAL8 deltaF  Frequency resolution: df=1/TCOH.
REAL8 patchSizeX  Patch size in radians along x-axis.
REAL8 patchSizeY  Patch size in radians along y-axis.
REAL8 minWidthRatio  Ratio between the minimum annulus width for this search and the minimum annulus width for one year integration time. This value should be in the interval [1.0, 25.0].

struct REAL8Cart3Coor

Three dimensional Cartesian coordinates. The fields are:

REAL8 x
REAL8 y
REAL8 z

struct REAL8Cart2Coor

Two dimensional Cartesian coordinates. The fields are:

REAL8 x
REAL8 y

struct REAL8Polar2Coor

Two dimensional polar coordinates. The fields are:

REAL8 alpha
REAL8 radius

struct REAL8UnitPolarCoor

Polar coordinates of a unit vector on the sphere. The fields are:

REAL8 alpha  Any value
REAL8 delta  In the interval $[-\pi/2, \pi/2]$

struct HOUGHDemodPar

Demodulation parameters needed for the Hough transform. All coordinates are assumed to be with respect to the same reference system. The fields are:

REAL8 deltaF  : Frequency resolution: df=1/TCOH.
REAL8UnitPolarCoor skyPatch  : $N_{center}$ (alpha, delta): position of the center of the patch.
REAL8 patchSizeX  Patch size in radians along x-axis.
REAL8 patchSizeY  Patch size in radians along y-axis.
REAL8Cart3Coor veloC  : $v(t)/c$ (x,y,z): Relative detector velocity
REAL8Cart3Coor positC : \( (x(t) - x(t_0))/c \): Position of the detector.

REAL8 timeDiff : \( T_N(t) - T_N(t_0) \): Time difference.

REAL8Vector spin : Spin down information. It includes the fields: length: maximum order of spin-down parameter, and *data: pointer to spin-down parameter set \( F_k \).

struct HOUGHParamPLUT

Parameters needed to construct the partial look up table. The fields are:

INT8 f0Bin Frequency bin for which it has been constructed

REAL8 deltaF Frequency resolution: \( df=1/TCOH \).

REAL8UnitPolarCoor xi Center of the circle on the celestial sphere, \( \xi(\alpha,\delta) \) in the rotated coordinates.

REAL8 cosDelta \( \Delta \cos(\phi) \) corresponding to one annulus.

INT4 offset Frequency bin corresponding to center of patch; measured w.r.t. f0Bin.

INT4 nFreqValid Number of frequency bins for which the LUT is valid.

REAL8 cosPhiMax0 \( \max(\cos(\phi)) \) of the f0Bin.

REAL8 cosPhiMin0 \( \min(\cos(\phi)) \) of the f0Bin.

REAL8 epsilon maximum angle (distance in radians) from the pole to consider a circle as a line in the projected plane.
26.1.1 Module Stereographic.c

Routines to perform rotations on the celestial sphere and stereographic projection.

Prototypes

```c
void LALRotatePolarU(LALStatus *status, 
                     REAL8UnitPolarCoor *out, 
                     REAL8UnitPolarCoor *in, 
                     REAL8UnitPolarCoor *par)

void LALInvRotatePolarU(LALStatus *status, 
                        REAL8UnitPolarCoor *out, 
                        REAL8UnitPolarCoor *in, 
                        REAL8UnitPolarCoor *par)

void LALStereoProjectPolar(LALStatus *status, 
                          REAL8Polar2Coor *out, 
                          REAL8UnitPolarCoor *in)

void LALStereoProjectCart(LALStatus *status, 
                        REAL8Cart2Coor *out, 
                        REAL8UnitPolarCoor *in)

void LALStereoInvProjectPolar(LALStatus *status, 
                               REAL8UnitPolarCoor *out, 
                               REAL8Polar2Coor *in)

void LALStereoInvProjectCart(LALStatus *status, 
                           REAL8UnitPolarCoor *out, 
                           REAL8Cart2Coor *in)
```

Description

The function `LALRotatePolarU()` rotates the celestial sphere so that a given point, in the rotated coordinates, corresponds to \((\alpha = 0, \delta = -\pi/2)\). The inputs are: `*par` the reference point (e.g., the center of the sky-patch) of type `REAL8UnitPolarCoor` and `*in` the point on the celestial sphere we want to rotate. The output is `*out` of type `REAL8UnitPolarCoor` containing the coordinates of the point in the rotated reference frame.

The function `LALInvRotatePolarU()` does the inverse rotation. Given the reference point `*par` (e.g., the center of the sky-patch) of type `REAL8UnitPolarCoor` and a point `*in` in the rotated reference frame, the output `*out` are the coordinates of the point is the same reference system as `*par`. All inputs and output being of type `REAL8UnitPolarCoor`.

Given a point on the celestial sphere `*in` of type `REAL8UnitPolarCoor`, the function `LALStereoProjectPolar()` returns `*out`, of type `REAL8Polar2Coor`, the stereographic projection of that point in polar coordinates, with the particularity that `out->radius` can be positive or negative. `in->delta=\pi/2` is an invalid argument and an error will output.

Given a point on the celestial sphere `*in` of type `REAL8UnitPolarCoor`, the function `LALStereoProjectCart()` returns `*out`, of type `REAL8Cart2Coor`, the stereographic projection of that point in Cartesian coordinates. `in->delta=\pi/2` is an invalid argument and an error will output.

Given a point on the projected plane `*in`, the functions `LALStereoInvProjectPolar()` and `LALStereoInvProjectCart()` provide the corresponding point on the sphere `*out` (corresponding to the inverse stereographic projection) of type `REAL8UnitPolarCoor`. 
26.1.2 Module PatchGrid.c
Function for tiling the sky-patch (on the projected plane).

Prototypes

```c
void LALHOUGHComputeSizePar (LALStatus *status, /* demodulated case */
   HOUGHSizePar *out,
   HOUGHResolutionPar *in1
 )

void LALHOUGHComputeNDSizePar (LALStatus *status, /* non-demod. case */
   HOUGHSizePar *out,
   HOUGHResolutionPar *in1
 )

void LALHOUGHFillPatchGrid (LALStatus *status,
   HOUGHPatchGrid *out, /* */
   HOUGHSizePar *in1)
```

Description

This is a provisional final now routine for tiling a sky-patch on the projected plane.

Patch size specified by user

---doc needs to be updated---

The reason to call it provisional is because the size of the patch depends on the grid used in the demodulation stage. Neighbour sky-patches should not be separated nor overlapping too much. Here for setting the patch size, we consider only $v_{epicycle}$, the frequency $f_0$ and $\delta F$ so that the 'longitudinal' size of the patch is given by $\text{side} \approx \frac{\delta F}{f_0} \cdot \frac{c}{v_{epi}}$. By taking $f_0$ to be the maximum frequency considered in that step, the patch-size is valid for a whole frequency range.

Given input parameters, the function LALHOUGHPatchGrid() provides patch information.

The input *in1 is a structure of type HOUGHResolutionPar containing some resolution parameters such as: in1->f0 a frequency, in1->deltaF the frequency resolution, and in1->minWidthRatio the ratio between the minimum annulus width for this search and the minimum annulus width for 1 year integration time. This value should be in the interval [1.0, 25.0].

The output structure *out of type HOUGHPatchGrid stores patch grid information. The fields are:

- out->f0 The frequency to construct grid
- out->deltaF The frequency resolution: $\text{df} = \frac{1}{\text{TCOH}}$
- out->minWidthRatio Same as in1->minWidthRatio
- out->deltaX Space discretization in the x-direction $= \frac{\text{deltaF} \cdot \text{minWidthRatio}}{(f_0 \cdot \text{VTOT} \cdot \text{PIXELFACTORX})}$.
- out->xMin Minimum x value allowed, given as the center of the first pixel.
- out->xMax Maximum x value allowed, given as the center of the last pixel.
- out->xSide Real number of pixels in the x direction (in the projected plane). It should be smaller or equal to xSideMax ($x_{\text{Side}} = \frac{\text{VTOT} \cdot \text{PIXELFACTORX}}{(\text{VEPI} \cdot \text{minWidthRatio})}$).
- out->xSideMax Length of xCoor.
- out->xCoor Coordinates of the pixel centers in the x-direction.
- out->deltaY Space resolution in the y-direction $= \frac{\text{deltaF} \cdot \text{minWidthRatio}}{(f_0 \cdot \text{VTOT} \cdot \text{PIXELFACTORY})}$.
- out->yMin Minimum y value allowed, given as the center of the first pixel.
- out->yMax Maximum y value allowed, given as the center of the last pixel.
out->ySide Real number of pixels in the y-direction. It should be smaller or equal to ySideMax
(ySide = VTOT * PIXELFACTORY / (VEPI * minWidthRatio)).

out->ySideMax Length of yCoor.

out->yCoor Coordinates of the pixel centers in the y-direction.

Uses

Notes
26.1.3 Module *ParamPLUT.c*

Function that calculates the parameters needed for generating the look-up-table.

**Prototypes**

```c
void LALHOUGHParamPLUT (LALStatus *status, HOUGHParamPLUT *out, /* parameters needed build LUT*/
HOUGHSizePar *size, HOUGHDemodPar *par) /* demodulation parameters */
```

**Description**

This routine calculates the parameters needed for generating the look-up-table. It is valid for all cases in which the Hough transform master equation is of the form:

\[ f(t) - f_0 = \vec{\xi} \cdot (\hat{n} - \hat{N}) \]

or equivalently,

\[ \cos(\phi) = \frac{(f(t) - f_0 + \vec{\xi} \cdot \hat{N})}{|\vec{\xi}|}. \]

\( \vec{\xi} \), hereafter \( \xi \), is calculated according to the demodulation procedure used in a first stage.

The inputs are:

- **INT8 f0Bin**: The frequency bin to construct the LUT.
- **HOUGHDemodPar *par** : The demodulation parameters:
  - `par->deltaF` : Frequency resolution: \( df = 1/TCOH \).
  - `par->skyPatch` : \( N_{center} \) (alpha, delta): position of the center of the patch.
  - `par->patchSizeX` : Size of sky patch along x-axis measured in radians.
  - `par->patchSizeY` : Size of sky patch along y-axis measured in radians.
  - `par->veloC` : \( v(t)/c \) (x,y,z): relative detector velocity.
  - `par->positC` : \( (x(t) - x(\hat{t}_0))/c \) (x,y,z). Position of the detector.
  - `par->timeDiff` : \( T_{\hat{N}}(t) - T_{\hat{N}}(\hat{t}_0) \): Time difference.
  - `par->spin` : length: Maximum order of spin-down parameter. *data*: Pointer to spin-down parameter set \( F_k \).

The output *out* of type **HOUGHParamPLUT** contains all the parameters needed to build the look-up-table for constructing the partial Hough maps. Those are:

- **out->f0Bin** : Frequency bin for which it has been constructed.
- **out->deltaF** : Frequency resolution: \( df = 1/TCOH \).
- **out->xi** : Center of the circle on the celestial sphere, \( xi(\alpha,\delta) \) in the rotated coordinates.
- **out->cosDelta** : \( \Delta \cos(\phi) \) corresponding to one annulus: \( \text{deltaF}/|\xi| \).
- **out->cosPhiMax0** : \( \max(\cos(\phi)) \) of the \( f0Bin : (xi \cdot N + \text{deltaF}/2)/|\xi| \).
- **out->cosPhiMin0** : \( \min(\cos(\phi)) \) of the \( f0Bin : \cos(\phi) \cdot \cos(\phi) \).
- **out->epsilon** : Maximum angle (distance in radians) from the pole to consider a circle as a line in the projected plane: \( 8 \cdot \text{LINERR} \cdot f0Bin \cdot \text{VEPI} / \text{VTOT} \). For explanations see Sintes’ notes.

**Uses**

LALRotatePolarU()

**Notes**

Author: Sintes, A. M., Krishnan, B.

$Id: ParamPLUT.c,v 1.5 2006/02/11 09:45:11 badri Exp$
Module **NDParamPLUT.c**

Function that calculates the parameters needed for generating the look-up-table.

**Prototypes**

```c
void LALNDHOUGHParamPLUT (LALStatus *status,
                          HOUGHParamPLUT *out, /* parameters needed build LUT*/
                          HOUGHSizePar  *size,
                          HOUGHDemodPar *par) /* demodulation parameters */
```

**Description**

This routine calculates the parameters needed for generating the look-up-table. It is valid for all cases in which the Hough transform master equation is of the form: \( f(t) - f_0 = \vec{\xi} \cdot \hat{n} \), or equivalently, \( \cos(\phi) = \frac{(f(t) - f_0)}{|\vec{\xi}|} \). \( \vec{\xi} \), hereafter \( \xi \), is calculated according to the demodulation procedure used in a first stage.

The inputs are:

- **INT8 f0Bin**: The frequency bin to construct the LUT.
- **HOUGHDemodPar *par**: The demodulation parameters:
  - `par->deltaF`: Frequency resolution: \( \Delta f = 1/\text{TCOH} \).
  - `par->skyPatch`: \( N_{center} \) (alpha, delta): position of the center of the patch.
  - `par->patchSizeX`: Size of sky patch along x-axis measured in radians.
  - `par->patchSizeY`: Size of sky patch along y-axis measured in radians.
  - `par->veloC`: \( v(t)/c \times (x,y,z) \): relative detector velocity.
  - `par->positC`: \( \vec{x}(t) - \vec{x}(t_0) \)/c \times (x,y,z). Position of the detector.
  - `par->timeDiff`: \( T_N(t) - T_N(t_0) \): Time difference.
  - `par->spin`: Length: Maximum order of spin-down parameter. *data*: Pointer to spin-down parameter set \( F_k \).

The output **out** of type **NDHOUGHParamPLUT** contains all the parameters needed to build the look-up-table for constructing the partial Hough maps. Those are:

- **out->f0Bin**: Frequency bin for which it has been constructed.
- **out->deltaF**: Frequency resolution: \( \Delta f = 1/\text{TCOH} \).
- **out->xi**: Center of the circle on the celestial sphere, \( \xi(\alpha,\delta) \) in the rotated coordinates.
- **out->cosDelta**: \( \Delta \cos(\phi) \) corresponding to one annulus: \( \Delta \Delta F/|\xi| \).
- **out->offset**: Frequency bin corresponding to center of patch (it is zero in the demodulated case).
- **out->nFreqValid**: Number of frequency bins for which LUT is valid.
- **out->cosPhiMax0**: \( \max(\cos(\phi)) \) of the \( f0Bin : (\xi \times N + \Delta \Delta F/2)/|\xi| \).
- **out->cosPhiMin0**: \( \min(\cos(\phi)) \) of the \( f0Bin : \cosPhiMax0 - \cosDelta \).
- **out->epsilon**: Maximum angle (distance in radians) from the pole to consider a circle as a line in the projected plane: \( 8.\times \text{LINERR} \times f0Bin \times \text{VEPI} \times \text{VEPI} / \text{VTOT} \). For explanations see Sintes’ notes.

**Uses**

- `LALRotatePolarU()`

**Notes**

Author: Sintes, A. M. and Krishnan, B.

\$Id: NDParamPLUT.c,v 1.3 2006/02/11 09:45:11 badri Exp $
26.1.5 Module ConstructPLUT.c

Construction of the look up table for generating partial Hough maps assuming the use of the stereographic projection.

Prototypes

```c
void LALHOUGHConstructPLUT(LALStatus *status,
   HOUGHptfLUT *lut,
   HOUGHPatchGrid *patch,
   HOUGHParamPLUT *par)
```

Macros (used only internally)

```c
#define MAX(A, B) (((A) < (B)) ? (B) : (A))
#define MIN(A, B) (((A) < (B)) ? (A) : (B))
#define cot(A) (1./tan(A))
#define rint(x) floor((x)+0.5)
```

Static function declarations

```c
static void PLUTInitialize(HOUGHptfLUT *);
static void FillPLUT(HOUGHParamPLUT *, HOUGHptfLUT *, HOUGHPatchGrid *);
static void CheckLeftCircle(REAL8, REAL8, REAL8, INT4 *, INT4 *, INT4 *,
   HOUGHPatchGrid *);
static void CheckRightCircle(REAL8, REAL8, REAL8, INT4 *, INT4 *, INT4 *,
   HOUGHPatchGrid *);
static void DrawRightCircle(REAL8, REAL8, REAL8, INT4, INT4, COORType *,
   HOUGHPatchGrid *);
static void DrawLeftCircle(REAL8, REAL8, REAL8, INT4, INT4, COORType *,
   HOUGHPatchGrid *);
static void CheckLineCase(REAL8, REAL8, REAL8, REAL8 *, INT4 *);
static void FindExactLine(REAL8, REAL8, REAL8, REAL8 *, REAL8 *);
static void FindLine(REAL8, REAL8, REAL8, REAL8, REAL8 *, REAL8 *);
static void CheckLineIntersection(REAL8, REAL8, REAL8, INT4 *, INT4 *, INT4 *,
   INT4 *, HOUGHPatchGrid *);
static void DrawLine(REAL8, REAL8, REAL8, INT4, INT4, INT4, INT4, INT4, INT4 *
   , HOUGHPatchGrid *);
static void Fill1Column(INT4, INT4 *, INT4 *, HOUGHptfLUT *, HOUGHPatchGrid *);
static void FillCaseN1(INT4, INT4, INT4, INT4, INT4, INT4, INT4, INT4 *
   , HOUGHptfLUT *, HOUGHPatchGrid *);
static void FillCaseN2(INT4, INT4, INT4, INT4, INT4, INT4, INT4, INT4 *
   , HOUGHptfLUT *, HOUGHPatchGrid *);
static void FillCaseN3(INT4, INT4, INT4, INT4, INT4, INT4, INT4, INT4 *
   , HOUGHptfLUT *, HOUGHPatchGrid *);
static void FillCaseN4(INT4, INT4, INT4, INT4, INT4, INT4, INT4, INT4 *
   , HOUGHptfLUT *, HOUGHPatchGrid *);
static void FillCaseN5(INT4, INT4, INT4, INT4, INT4, INT4, INT4, INT4 *
   , HOUGHptfLUT *, HOUGHPatchGrid *);
static void FillCaseN6(INT4, INT4, INT4, INT4, INT4, INT4, INT4, INT4 *
   , HOUGHptfLUT *, HOUGHPatchGrid *);
static void FillCaseN7(INT4, INT4, INT4, INT4, INT4, INT4, INT4, INT4 *
   , HOUGHptfLUT *, HOUGHPatchGrid *);
static void FillCaseN8(INT4, INT4, INT4, INT4, INT4, INT4, INT4, INT4 *
   , HOUGHptfLUT *, HOUGHPatchGrid *);
static void Fill1ColumnAnor(INT4, INT4, INT4, INT4, INT4, INT4, INT4, INT4 *
   , HOUGHptfLUT *, HOUGHPatchGrid *);
static void FillCaseA1(INT4, INT4, INT4, INT4, INT4, INT4, INT4, INT4 *
   , HOUGHptfLUT *, HOUGHPatchGrid *);
static void FillCaseA2(INT4, INT4, INT4, INT4, INT4, INT4, INT4, INT4 *
   , HOUGHptfLUT *, HOUGHPatchGrid *);
static void FillCaseA3(INT4, INT4, INT4, INT4, INT4, INT4, INT4, INT4 *
   , HOUGHptfLUT *, HOUGHPatchGrid *);
static void InitialCircleCase(INT4 *, REAL8, REAL8, REAL8, REAL8 *, INT4 *,
   INT4 *, INT4 *, HOUGHptfLUT *, HOUGHPatchGrid *);
static void SecondCircleCase(INT4, INT4, INT4, INT4, INT4, INT4, INT4, INT4 *
   , HOUGHptfLUT *, HOUGHPatchGrid *);
static void FollowCircleCase(INT4, INT4, INT4, INT4, INT4, INT4, INT4, INT4 *
   , HOUGHptfLUT *, HOUGHPatchGrid *);
static void InitialLineCase(INT4 *, REAL8, REAL8, REAL8, INT4 *, INT4 *
   , HOUGHptfLUT *, HOUGHPatchGrid *);
```
static void SecondLineCase(INT4, INT4 *, REAL8, REAL8, REAL8, INT4 *, HOUGHptfLUT *, HOUGHPatchGrid *);
static void FollowLineCase(INT4, INT4 *, REAL8, REAL8, REAL8, REAL8, INT4, INT4 *, HOUGHptfLUT *, HOUGHPatchGrid *);

Description

This module is the core of the Hough transform. The LAL function `LALHOUGHConstructPLUT()` constructs the look up tables that will be used to build the so-called partial-Hough maps. Each look up table is valid for a given sky-patch, time, and frequency range around a certain \( f_0 \) value. The look up table contains all the necessary information regarding the borders of the annuli clipped on the ‘projected’ two dimensional sky-patch.

The inputs are: `HOUGHPatchGrid *patch` containing the grid patch information. This is independent of the sky location of the patch. And `HOUGHParamPLUT *par` with all the other parameters needed.

The output is: `HOUGHptfLUT *lut`. The fields are:

- `lut-> timeIndex` Time index of the LUT.
- `lut->f0Bin` Frequency bin for which it has been constructed.
- `lut->deltaF` Frequency resolution: \( df=1/TCOH \)
- `lut->nFreqValid = PIXERR * f0Bin * VEPI / VTOT`: Number of frequencies where the LUT is valid.
- `lut->iniBin` First bin affecting the patch with respect to \( f_0 \).
- `lut->nBin` Exact number of bins affecting the patch.
- `lut->maxNBins` Maximum number of bins affecting the patch (for memory allocation purposes), i.e. length of `lut->bin`.
- `lut->maxNorders` Maximum number of borders affecting the patch (for memory allocation purposes), i.e. length of `lut->border`.
- `lut->border` The annulus borders.
- `lut->bin` Bin to border correspondence.

More detailed documentation can be found in the source code itself.

Uses

Notes
26.1.6 Program **TestConstructPLUT.c**

Tests the construction of the Look up Table (LUT)

**Usage**

`TestConstructPLUT [-d debuglevel] [-o outfile] [-f f0] [-p alpha delta] [-s patchSizeX patchSizeY]`

**Description**

This program generates a patch grid, calculates the parameters needed for building the LUT, builds the LUT and outputs a partial Hough map derivative into a file. The sky patch is set at the south pole, no spin-down parameters are assumed for the demodulation and every third peak in the spectrum is selected.

By default, running this program with no arguments simply tests the subroutines, producing an output file called `OutHough.asc`. All default parameters are set from `#defined` constants.

The `-d` option sets the debug level to the specified value `debuglevel`. The `-o` flag tells the program to print the partial Hough map derivative to the specified data file `outfile`. The `-f` option sets the intrinsic frequency `f0` at which build the LUT. The `-p` option sets the velocity orientation of the detector `alpha`, `delta` (in radians).

**Exit codes**

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NORM</td>
<td>0</td>
<td>&quot;Normal exit&quot;</td>
</tr>
<tr>
<td>SUB</td>
<td>1</td>
<td>&quot;Subroutine failed&quot;</td>
</tr>
<tr>
<td>ARG</td>
<td>2</td>
<td>&quot;Error parsing arguments&quot;</td>
</tr>
<tr>
<td>BAD</td>
<td>3</td>
<td>&quot;Bad argument values&quot;</td>
</tr>
<tr>
<td>FILE</td>
<td>4</td>
<td>&quot;Could not create output file&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants `TESTCONSTRUCTPLUTC_E<name>`, and the status descriptions in `TESTCONSTRUCTPLUTC_MSGE<name>`. The source code with these messages is in `TestConstructPLUT.c` on line 1.131.

**Uses**

- `LALHOUGHPatchGrid()`
- `LALHOUGHParamPLUT()`
- `LALHOUGHConstructPLUT()`
- `LALPrintError()`
- `LALMalloc()`
- `LALFree()`
- `LALCheckMemoryLeaks()`

**Notes**
26.1.7 Program TestNDConstructPLUT.c

Tests the construction of the Look up Table (LUT)

Usage

TestNDConstructPLUT [-d debuglevel] [-o outfile] [-f f0] [-p alpha delta] [-s patchSizeX patchSizeY]

Description

This program generates a patch grid, calculates the parameters needed for building the LUT, builds the LUT and outputs a partial Hough map derivative into a file. The sky patch is set at the south pole, no spin-down parameters are assumed for the demodulation and every third peak in the spectrum is selected.

By default, running this program with no arguments simply tests the subroutines, producing an output file called OutHough.asc. All default parameters are set from #defined constants.

The -d option sets the debug level to the specified value debuglevel. The -o flag tells the program to print the partial Hough map derivative to the specified data file outfile. The -f option sets the intrinsic frequency f0 at which build the LUT. The -p option sets the velocity orientation of the detector alpha, delta (in radians).

Exit codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NORM</td>
<td>0</td>
<td>&quot;Normal exit&quot;</td>
</tr>
<tr>
<td>SUB</td>
<td>1</td>
<td>&quot;Subroutine failed&quot;</td>
</tr>
<tr>
<td>ARG</td>
<td>2</td>
<td>&quot;Error parsing arguments&quot;</td>
</tr>
<tr>
<td>BAD</td>
<td>3</td>
<td>&quot;Bad argument values&quot;</td>
</tr>
<tr>
<td>FILE</td>
<td>4</td>
<td>&quot;Could not create output file&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants TESTNDCONSTRUCTPLUTC_E<name>, and the status descriptions in TESTNDCONSTRUCTPLUTC_MSGE<name>. The source code with these messages is in TestNDConstructPLUT.c on line l.132.

Uses

LALHOUGHPatchGrid()
LALNDHOUGHParamPLUT()
LALHOUGHConstructPLUT()
LALPrintError()
LALMalloc()
LALFree()
LALCheckMemoryLeaks()

Notes
26.2 Header PHMD.h

Conversion from peaks in a spectrum into a partial Hough map derivative.

Synopsis

#include <lal/PHMD.h>

The Hough map is an histogram, thus additive. It can be seen as the sum of several partial Hough maps constructed using just one periodogram, or equivalently, as the sum of partial Hough map derivatives (PHMD) and then integrating the result.

A PHMD can be represented by a set of borders, here called left and right. They indicate the beginning and the end of the annuli. The position of the so-called left borders should be marked with +1, and the position of the right borders should be marked with −1 in the PHMD. To obtain a partial Hough map, one needs to integrate each row of the PHMD from left to right.

The representation of a PHMD is simplified by considering pointers to the borders in a pre-calculated look-up-table, plus some extra information about their character and edge effects when clipping on a finite patch.

Error conditions

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>1</td>
<td>&quot;Null pointer&quot;</td>
</tr>
<tr>
<td>SIZE</td>
<td>2</td>
<td>&quot;Invalid input size&quot;</td>
</tr>
<tr>
<td>SZMM</td>
<td>4</td>
<td>&quot;Size mismatch&quot;</td>
</tr>
<tr>
<td>INT</td>
<td>6</td>
<td>&quot;Invalid interval&quot;</td>
</tr>
<tr>
<td>SAME</td>
<td>8</td>
<td>&quot;Input/Output data vectors are the same&quot;</td>
</tr>
<tr>
<td>FREQ</td>
<td>10</td>
<td>&quot;Invalid frequency&quot;</td>
</tr>
<tr>
<td>VAL</td>
<td>12</td>
<td>&quot;Invalid value&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants PHMDH_<name>, and the status descriptions in PHMDH_MSGE<name>. The source code with these messages is in PHMD.h on line 1.232.

Structures

struct HOUGHPeakGram

This structure stores the “peak-gram”. The fields are:

INT2 timeIndex The time index of the peak-gram.
REAL8 deltaF Frequency resolution: df=1/TCOH.
UINT8 fBinIni Frequency index of the first element of the spectrum covered by this peak-gram. It can be seen as an offset
UINT8 fBinFin Frequency index of the last element of the spectrum covered by this peak-gram.
UINT4 length Number of peaks present in the peak-gram.
INT4 *peak The peak indices relative to fBinIni, i.e., the zero peak corresponds to fBinIni.

struct HOUGHphmd

This structure stores a partial Hough map derivative. The fields are:

UINT8 fBin Frequency bin of this partial map derivative
UINT2 lengthLeft Exact number of Left borders.
UINT2 lengthRight  Exact number of Right borders.

UINT2 maxNBorders  Maximum number of borders of each type (for memory allocation purposes), i.e. length of *leftBorderP and *rightBorderP.

HOUGHBorder **leftBorderP  Pointers to borders.

HOUGHBorder **rightBorderP  Pointers to borders.

UINT2 ySide  Number of elements of firstColumn.

UCHAR *firstColumn  First column border, containing the edge effects when clipping on a finite patch.
26.2.1 Module **Peak2PHMD.c**

Construction of Partial-Hough-Map-Derivatives (PHMD) given a peak-gram and the look-up-table.

Prototypes

```c
void LALHOUGHPeak2PHMD (LALStatus *status,
                         HOUGHphmd *phmd, /* partial Hough map derivative */
                         HOUGHptfLUT *lut, /* Look up table */
                         HOUGHPeakGram *pg) /* peakgram */
```

Description

This routine produces a PHMD at a certain frequency for a given peak-gram and look-up-table.

The inputs are:
- `phmd->fBin`: The frequency bin of this PHMD.
- `*lut`: The look-up-table (of type HOUGHptfLUT)
- `*pg`: The peak-gram (of type HOUGHPeakGram)

The function `LALHOUGHPeak2PHMD` makes sure that the LUT, the peak-gram and also the frequency of the PHMD are compatible.

The output `HOUGHphmd *phmd` is a structure containing the frequency bin of this PHMD, the total number of borders of each type (*Left and Right*) to be marked, the pointers to the borders in the corresponding look-up-table, plus *border* effects of clipping on a finite patch.

Uses

Notes
26.2.2 Program TestPeak2PHMD.c

Tests the construction of Partial-Hough-Map-Derivatives (PHMD)

Usage

TestPeak2PHMD [-d debuglevel] [-o outfile] [-f f0] [-p alpha delta]

Description

This program generates a patch grid, calculates the parameters needed for building a LUT, builds the LUT, constructs a PHMD at a certain frequency (shifted from the frequency at which the LUT was built), and outputs the PHMD into a file. The sky patch is set at the south pole, no spin-down parameters are assumed for the demodulation and every third peak in the spectrum is selected. The peak-gram frequency interval is large enough to ensure compatibility with the LUT and the frequency of the PHMD.

By default, running this program with no arguments simply tests the subroutines, producing an output file called OutHough.asc. All default parameters are set from #defined constants.

The -d option sets the debug level to the specified value debuglevel. The -o flag tells the program to print the partial Hough map derivative to the specified data file outfile. The -f option sets the intrinsic frequency f0 at which build the LUT. The -p option sets the velocity orientation of the detector alpha, delta (in radians).

Exit codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NORM</td>
<td>0</td>
<td>&quot;Normal exit&quot;</td>
</tr>
<tr>
<td>SUB</td>
<td>1</td>
<td>&quot;Subroutine failed&quot;</td>
</tr>
<tr>
<td>ARG</td>
<td>2</td>
<td>&quot;Error parsing arguments&quot;</td>
</tr>
<tr>
<td>BAD</td>
<td>3</td>
<td>&quot;Bad argument values&quot;</td>
</tr>
<tr>
<td>FILE</td>
<td>4</td>
<td>&quot;Could not create output file&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants TESTPEAK2PHMDC_E<name>, and the status descriptions in TESTPEAK2PHMDC_MSGE<name>. The source code with these messages is in TestPeak2PHMD.c on line 1.137.

Uses

LALHOUGHPatchGrid()
LALHOUGHParamPLUT()
LALHOUGHConstructPLUT()
LALHOUGHPeak2PHMD()
LALPrintError()
LALMalloc()
LALFree()
LALCheckMemoryLeaks()

Notes

Author: Sintes, A. M., Krishnan, B.

$Id: TestPeak2PHMD.c,v 1.8 2007/06/08 14:41:46 bema Exp$
26.2.3 Program TestNDPeak2PHMD.c

Tests the construction of Partial-Hough-Map-Derivatives (PHMD)

Usage

TestNDPeak2PHMD [-d debuglevel] [-o outfile] [-f f0] [-p alpha delta] [-s patchSizeX patchSizeY]

Description

This program generates a patch grid, calculates the parameters needed for building a LUT, builds the LUT, constructs a PHMD at a certain frequency (shifted from the frequency at which the LUT was built), and outputs the PHMD into a file. The sky patch is set at the south pole, no spin-down parameters are assumed for the demodulation and every third peak in the spectrum is selected. The peak-gram frequency interval is large enough to ensure compatibility with the LUT and the frequency of the PHMD.

By default, running this program with no arguments simply tests the subroutines, producing an output file called OutHough.asc. All default parameters are set from #defined constants.

The -d option sets the debug level to the specified value debuglevel. The -o flag tells the program to print the partial Hough map derivative to the specified data file outfile. The -f option sets the intrinsic frequency f0 at which build the LUT. The -p option sets the velocity orientation of the detector alpha, delta (in radians).

Exit codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NORM</td>
<td>0</td>
<td>&quot;Normal exit&quot;</td>
</tr>
<tr>
<td>SUB</td>
<td>1</td>
<td>&quot;Subroutine failed&quot;</td>
</tr>
<tr>
<td>ARG</td>
<td>2</td>
<td>&quot;Error parsing arguments&quot;</td>
</tr>
<tr>
<td>BAD</td>
<td>3</td>
<td>&quot;Bad argument values&quot;</td>
</tr>
<tr>
<td>FILE</td>
<td>4</td>
<td>&quot;Could not create output file&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants TESTNDPEAK2PHMDC_E<name>, and the status descriptions in TESTNDPEAK2PHMDC_MSGE<name>. The source code with these messages is in TestNDPeak2PHMD.c on line l.135.

Uses

LALHOUGHPatchGrid()
LALNDHOUGHParamPLUT()
LALHOUGHConstructPLUT()
LALHOUGHPeak2PHMD()
LALPrintError()
LALMalloc()
LALFree()
LALCheckMemoryLeaks()

Notes
26.3 Header HoughMap.h

Provides subroutines for initialization and construction of Hough-map derivatives and total Hough-maps.

Synopsis

```c
#include <lal/HoughMap.h>
```

Error conditions

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>1</td>
<td>&quot;Null pointer&quot;</td>
</tr>
<tr>
<td>SIZE</td>
<td>2</td>
<td>&quot;Invalid input size&quot;</td>
</tr>
<tr>
<td>SZMM</td>
<td>4</td>
<td>&quot;Size mismatch&quot;</td>
</tr>
<tr>
<td>INT</td>
<td>6</td>
<td>&quot;Invalid interval&quot;</td>
</tr>
<tr>
<td>SAME</td>
<td>8</td>
<td>&quot;Input/Output data vectors are the same&quot;</td>
</tr>
<tr>
<td>FREQ</td>
<td>10</td>
<td>&quot;Invalid frequency&quot;</td>
</tr>
<tr>
<td>VAL</td>
<td>12</td>
<td>&quot;Invalid value&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants ` HOUGHMAPH_E <name>`, and the status descriptions in ` HOUGHMAPH_MSGE <name>`. The source code with these messages is in `HoughMap.h` on line 1.224.

Structures and type definitions

```c
typedef CHAR HoughDT
```

Hough Map derivative pixel type.

```c
typedef UCHAR HoughTT
```

Total Hough Map pixel type.

Depending of the number of maps to accumulate change both types `HoughDT` and `HoughTT` to `INT2` or `UINT2` respectively.

```c
struct HOUGHMapDeriv
```

This structure stores the Hough map derivative. The field is:

```c
UINT2 xSide  Number of physical pixels in the x direction.
UINT2 ySide  Number of physical pixels in the y direction.
HoughDT *map The pixel count derivatives. The number of elements to allocate is `ySide*(xSide+1)`.
```

```c
struct HOUGHMapTotal
```

This structure stores the Hough map. The fields are:

General information in case we want to save results:

```c
INT8 f0Bin Frequency bin for which it has been constructed
REAL8 deltaF Frequency resolution
UINT4 mObsCoh Ratio between the observation time and coherent timescale
UINT4 nPG Number of peakgrams used <= mObsCoh. There could be gaps during the observation time.
REAL8UnitPolarCoor skyPatch Coordinates of the versor $\hat{N}_{center}$ (alpha, delta) pointing to the center of the sky patch.
```
REAL8Vector spinDem  Spin parameters used in the demodulation stage.

REAL8Vector spinRes  Refined spin parameters used in the Hough transform.

       Here starts what is really needed:

UINT2 xSide  Number of physical pixels in the x direction.

UINT2 ySide  Number of physical pixels in the y direction.

HoughTT *map  The pixel counts. The number of elements to allocate is ySide*xSide.
26.3.1 Module HoughMap.c

Subroutines for initialization and construction of Hough-map derivatives and total Hough-maps.

Prototypes

```c
void LALHOUGHInitializeHD (LALStatus *status, HOUGHMapDeriv *hd) /* the Hough map derivative */
void LALHOUGHInitializeHT (LALStatus *status, HOUGHMapTotal *ht, /* the total Hough map */
                          HOUGHPatchGrid *patch) /* patch information */
void LALHOUGHAddPHMD2HD (LALStatus *status, /**< the status pointer */
                         HOUGHMapDeriv *hd, /**< the Hough map derivative */
                         HOUGHphmd *phmd) /**< info from a partial map */
void LALHOUGHAddPHMD2HD_W (LALStatus *status, /**< the status pointer */
                          HOUGHMapDeriv *hd, /**< the Hough map derivative */
                          HOUGHphmd *phmd) /**< info from a partial map */
void LALHOUGHIntegrHD2HT (LALStatus *status, /* the total Hough map */
                         HOUGHMapDeriv *hd) /* the Hough map derivative */
/** Find source sky location given stereographic coordinates indexes */
void LALStereo2SkyLocation (LALStatus *status, REAL8UnitPolarCoor *sourceLocation, /* output*/
                            UINT2 xPos,
                            UINT2 yPos,
                            HOUGHPatchGrid *patch,
                            HOUGHDemodPar *parDem)
```

Description

The function **LALHOUGHInitializeHD()** initializes the Hough map derivative space **HOUGHMapDeriv** *hd* to zero. Note that the length of the map *hd->map* should be *hd->ySide * (hd->xSide + 1).

The function **LALHOUGHInitializeHT()** initializes the total Hough map **HOUGHMapTotal** *ht* to zero and checks consistency between the number of physical pixels in the map and those given by the grid information structure **HOUGHPatchGrid** *patch*.

Given an initial Hough map derivative **HOUGHMapDeriv** *hd* and a representation of a PHMD **HOUGHphmd** *phmd*, the function **LALHOUGHAddPHMD2HD()** accumulates the partial Hough map derivative *phmd* to *hd* by adding +1 or −1 to the pixels corresponding to the left or right borders respectively. It takes into account corrections due to border effects as well.

The function **LALHOUGHIntegrHD2HT()** constructs a total Hough map **HOUGHMapTotal** *ht* from its derivative **HOUGHMapDeriv** *hd* by integrating each row (x-direction).

Uses

Notes
26.3.2 Program TestHoughMap.c

Tests the construction of Hough maps.

Usage

TestHoughMap [-d debuglevel] [-o outfile] [-f f0] [-p alpha delta]

Description

Similar to the previous ones, this program generates a patch grid, calculates the parameters needed for building a LUT, and builds the LUT. Then, given a peak-gram constructs a PHMD at a certain frequency (shifted from the frequency at which the LUT was built). The sky patch is set at the south pole, no spin-down parameters are assumed for the demodulation and every third peak in the spectrum is selected. The peak-gram frequency interval is large enough to ensure compatibility with the LUT and the frequency of the PHMD.

Moreover, this program initializes a Hough map HT and the Hough map derivative space HD, adds one PHMD into the Hough map derivative HD, constructs the total Hough map HT by integrating the HD, and outputs the HT into a file.

By default, running this program with no arguments simply tests the subroutines, producing an output file called OutHough.asc. All default parameters are set from #defined constants.

The -d option sets the debug level to the specified value debuglevel. The -o flag tells the program to print the partial Hough map derivative to the specified data file outfile. The -f option sets the intrinsic frequency f0 at which build the LUT. The -p option sets the velocity orientation of the detector alpha, delta (in radians).

Exit codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NORM</td>
<td>0</td>
<td>&quot;Normal exit&quot;</td>
</tr>
<tr>
<td>SUB</td>
<td>1</td>
<td>&quot;Subroutine failed&quot;</td>
</tr>
<tr>
<td>ARG</td>
<td>2</td>
<td>&quot;Error parsing arguments&quot;</td>
</tr>
<tr>
<td>BAD</td>
<td>3</td>
<td>&quot;Bad argument values&quot;</td>
</tr>
<tr>
<td>FILE</td>
<td>4</td>
<td>&quot;Could not create output file&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants TESTHOUGHMAPC_EX<name>, and the status descriptions in TESTHOUGHMAPC_MSGE<name>. The source code with these messages is in TestHoughMap.c on line 1.154.

Uses

LALHOUCHPatchGrid()
LALHOUCHParamPLUT()
LALHOUCHConstructPLUT()
LALHOUCHPeak2PHMD()
LALHOUCHInitializeHT()
LALHOUCHInitializeHD()
LALHOUCHAddPHMD2HD()
LALHOUCHIntegrHD2HT()
LALPrintError()
LALMalloc()
LALFree()
LALCheckMemoryLeaks()

Notes

Author: Sintes, A. M., Krishnan, B.

$Id: TestHoughMap.c,v 1.7 2007/06/08 14:41:46 bema Exp $
26.3.3 Program TestNDHoughMap.c

Tests the construction of Hough maps.

Usage

TestNDHoughMap [-d debuglevel] [-o outfile] [-f f0] [-p alpha delta] [-s patchSizeX patchSizeY]

Description

Similar to the previous ones, this program generates a patch grid, calculates the parameters needed for building a LUT, and builds the LUT. Then, given a peak-gram constructs a PHMD at a certain frequency (shifted from the frequency at which the LUT was built). The sky patch is set at the south pole, no spin-down parameters are assumed for the demodulation and every third peak in the spectrum is selected. The peak-gram frequency interval is large enough to ensure compatibility with the LUT and the frequency of the PHMD.

Moreover, this program initializes a Hough map HT and the Hough map derivative space HD, adds one PHMD into the Hough map derivative HD, constructs the total Hough map HT by integrating the HD, and outputs the HT into a file.

By default, running this program with no arguments simply tests the subroutines, producing an output file called OutHough.asc. All default parameters are set from defined constants.

The -d option sets the debug level to the specified value debuglevel. The -o flag tells the program to print the partial Hough map derivative to the specified data file outfile. The -f option sets the intrinsic frequency f0 at which build the LUT. The -p option sets the velocity orientation of the detector alpha, delta (in radians).

Exit codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NORM</td>
<td>0</td>
<td>&quot;Normal exit&quot;</td>
</tr>
<tr>
<td>SUB</td>
<td>1</td>
<td>&quot;Subroutine failed&quot;</td>
</tr>
<tr>
<td>ARG</td>
<td>2</td>
<td>&quot;Error parsing arguments&quot;</td>
</tr>
<tr>
<td>BAD</td>
<td>3</td>
<td>&quot;Bad argument values&quot;</td>
</tr>
<tr>
<td>FILE</td>
<td>4</td>
<td>&quot;Could not create output file&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants TESTNDHOUGHMAPC_E<name>, and the status descriptions in TESTNDHOUGHMAPC_MGE<name>. The source code with these messages is in TestNDHoughMap.c on line 1.152.

Uses

LALHOU G PatchGrid()
LALNDHOUGHParamPLUT()
LALHOU G ConstructPLUT()
LALHOU G Peak2PHMD()
LALHOU G InitializeHT()
LALHOU G InitializeHD()
LALHOU G AddPHMD2HD()
LALHOU G IntegrHD2HT()
LALPrintError()
LALMalloc()
LALFree()
LALCheckMemoryLeaks()

Notes

Author: Sintes, A. M., Krishnan, B.

$Id: TestNDHoughMap.c,v 1.5 2007/06/08 14:41:46 bema Exp $
Routines for building and updating the space of partial Hough map derivatives (PHMD), and related functions needed for the construction of total Hough maps at different frequencies and possible residual spin down parameters.

Synopsis

```c
#include <lal/LALHough.h>
```

As we mention before, the issue is to build histograms, the Hough map (HM), in the parameter space: for each intrinsic frequency \( f_0 \), each residual spin-down parameter, and each refined sky location inside the patch. Notice, from the master equation, that the effect of the residual spin-down parameter is just a change in \( F_0 \), and, at any given time, \( F_0 \) can be considered constant. Also, the Hough map is a histogram, thus additive. It can be seen as the sum of several partial Hough maps constructed using just one periodogram (or peak-gram).

Therefore, we can construct the HM for any \( f_0 \) and spin-down value by adding together, at different times, partial Hough maps (PHM) corresponding to different \( F_0 \) values (or equivalently, adding their derivatives PHMD and then integrating the result).

In practice this means that in order to obtain the HM for a given frequency and all possible residual spin-down parameters, we have to construct a CYLINDER of PHMD around the frequency \( f_0 \). All of the PHMD coming from data demodulated with the same parameters. The coordinates of the PHMD locate the position of the source in the sky, and by summing along different directions inside the cylinder we refine the spin-down value. To analyze another frequency, for all possible spin-down parameters, we just need to add a new line to the cylinder (and remove another one, in a circular buffer) and then proceed making all the possible sums again.

For the case of only 1 spin-down parameter we have to sum following straight lines whose slope is related to the grid in the residual spin-down parameter. We can distinguish (at most) as many lines as the number of the different periodograms used.

### Error conditions

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>1</td>
<td>&quot;Null pointer&quot;</td>
</tr>
<tr>
<td>SIZE</td>
<td>2</td>
<td>&quot;Invalid input size&quot;</td>
</tr>
<tr>
<td>SZMM</td>
<td>4</td>
<td>&quot;Size mismatch&quot;</td>
</tr>
<tr>
<td>INT</td>
<td>6</td>
<td>&quot;Invalid interval&quot;</td>
</tr>
<tr>
<td>SAME</td>
<td>8</td>
<td>&quot;Input/Output pointers are the same&quot;</td>
</tr>
<tr>
<td>FREQ</td>
<td>10</td>
<td>&quot;Invalid frequency&quot;</td>
</tr>
<tr>
<td>VAL</td>
<td>12</td>
<td>&quot;Invalid value&quot;</td>
</tr>
<tr>
<td>MEM</td>
<td>32</td>
<td>&quot;malloc() failed / Out of memory&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants `LALHOUGHH_E<name>`, and the status descriptions in `LALHOUGHH_MSGE<name>`. The source code with these messages is in `LALHough.h` on line 1339.

### Structures

```c
struct UINT8FrequencyIndexVector
```

This structure stores the frequency indexes of the partial-Hough map derivatives at different time stamps that have to be combined to form a Hough map for a given (null or) residual spin-down parameters. The fields are:

- `UINT4 length` Number of elements.
- `REAL8 deltaF` Frequency resolution.
- `UINT8 *data` The frequency indexes.
struct UINT8FrequencyIndexVectorSequence

This structure stores a set of frequency-index vectors. Every set corresponds to a different spin-down residual value. There will thus be as many sets as many spin-down residuals one wants to search over with the hough stage. The fields are:

UINT4 length  Number of elements.
UINT4 vectorLength  Frequency resolution.
UINT8FrequencyIndexVector *freqIndV  The frequency indexes.

struct HOUGHPeakGramVector

This structure contains a vector of peak-grams (for the different time stamps). The fields are:

UINT4 length  Number of elements.
HOUGHPeakGram *pg  The peak-grams.

struct HOUGHptfLUTVector

This structure contains a vector of partial look up tables (for the different time stamps). The fields are:

UINT4 length  Number of elements.
HOUGHptfLUT *lut  The partial look up tables.

struct HOUGHMapTotalVector

This structure contains a vector of Hough maps. The fields are:

UINT4 length  Number of elements.
HOUGHMapTotal *ht  The Hough maps.

struct PHMDVectorSequence

This structure contains a vector sequence of partial-Hough maps derivatives (for different time stamps and different frequencies) representing a circular buffer for the frequency indexes. The fields are:

UINT4 nfSize  Number of different frequencies.
UINT4 length  Number of elements for each frequency.
UINT8 fBinMin  Frequency index of the smallest intrisic freq.in buffer
REAL8 deltaF  Frequency resolution.
UINT4 breakLine  Mark $\in [0, \text{nfSize})$ (of the circular buffer) pointing to the starting of the fBinMin line.
HOUGHphmd *phmd  The partial Hough map derivatives.

struct HOUCHResidualSpinPar

This structure stores the residual spin-down parameters at a given time. The fields are:

REAL8 deltaF  Frequency resolution: $df=1/\text{TCOH}$.
REAL8 timeDiff  $T_N(t) - T_N(t_0)$: time difference

Author: Sintes, A.M., Krishnan, B.

26.4.1 Module DriveHough.c

Routines for building and updating the space of partial Hough map derivatives (PHMD), and related functions needed for the construction of total Hough maps at different frequencies and possible residual spin down parameters.

Prototypes

```c
void LALHOUGHConstructSpacePHMD (LALStatus *status, PHMDVectorSequence *phmdVS, /**< Cylindrical buffer of PHMDs */ HOUGHPeakGramVector *pgV, /**< Vector of peakgrams */ HOUGHptfLUTVector *lutV /**< vector of look up tables */)

void LALHOUGHupdateSpacePHMDup (LALStatus *status, PHMDVectorSequence *phmdVS, HOUGHPeakGramVector *pgV, HOUGHptfLUTVector *lutV)

void LALHOUGHupdateSpacePHMDDn (LALStatus *status, PHMDVectorSequence *phmdVS, HOUGHPeakGramVector *pgV, HOUGHptfLUTVector *lutV)

void LALHOUGHConstructHMT (LALStatus *status, HOUGHMapTotal *ht, /**< The output hough map */ UINT8FrequencyIndexVector *freqInd, /**< time-frequency trajectory */ PHMDVectorSequence *phmdVS /**< set of partial hough map derivatives */)

void LALHOUGHComputeFBinMap (LALSStatus *status, UINT8 *fBinMap, UINT8 *f0Bin, HOUGHResidualSpinPar *rs)

void LALHOUGHConstructHMT_W (LALSStatus *status, HOUGHMapTotal *ht, /**< The output hough map */ UINT8FrequencyIndexVector *freqInd, /**< time-frequency trajectory */ PHMDVectorSequence *phmdVS /**< set of partial hough map derivatives */)

void LALHOUGHWeighSpacePHMD (LALSStatus *status, PHMDVectorSequence *phmdVS, /**< partial hough map derivatives */ REAL8Vector *weightV /**< vector of weights */)

void LALHOUGHInitializeWeights (LALSStatus *status, REAL8Vector *weightV /**< vector of weights */)

void LALHOUGHNormalizeWeights (LALSStatus *status, REAL8Vector *weightV /**< vector of weights */)

void LALHOUGHComputeAMWeights (LALSStatus *status, REAL8Vector *weightV, LIGOTimeGPSVector *timeV, LALDetector *detector, EphemerisData *edat, REAL8 alpha, REAL8 delta)

void LALHOUGHComputeMultiIFOAMWeights (LALSStatus *status, REAL8Vector *weightV, SFTCatalog *catalog, EphemerisData *edat, REAL8 alpha, REAL8 delta)
```
Description

The function \texttt{LALHOUGHConstructSpacePHMD()} constructs the space of PHMD \texttt{PHMDVectorSequence *phmdVS}, given a \texttt{HOUGHPeakGramVector *pgV} and \texttt{HOUGHptfLUTVector *lutV}. The minimum frequency bin present corresponds to \texttt{phmdVS->fBinMin} and the total number of different frequencies is \texttt{phmdVS->nfSize}. At this moment the \texttt{fBinMin} line corresponds to the first row of the cylinder and \texttt{phmdVS->breakLine} is set to zero. \texttt{phmdVS->breakLine \in [0,nfSize]} is the pointer which identifies the position of the \texttt{fBinMin} row in the circular-cylinder buffer.

The function \texttt{LALHOUGHupdateSpacePHMDup()} updates the space of PHMD increasing the frequency \texttt{phmdVS->fBinMin} by one.

The function \texttt{LALHOUGHupdateSpacePHMDdn()} updates the space of PHMD decreasing the frequency \texttt{phmdVS->fBinMin} by one.

Given \texttt{PHMDVectorSequence *phmdVS}, the space of PHMD, and \texttt{UINT8FrequencyIndexVector *freqInd}, a structure containing the frequency indices of the PHMD at different time stamps that have to be combined to form a Hough map, the function \texttt{LALHOUGHConstrucHMT()} produces the total Hough map.

The function \texttt{LALHOUGHComputeFBinMap()} computes the corresponding frequency bin of a PHMD \texttt{UINT8 *fBinMap} for a given intrinsic frequency bin of a source \texttt{UINT8 *f0Bin}, and information regarding the time and the residual spin down parameters \texttt{HOUGHResidualSpinPar *rs}.

Uses

\texttt{LALHOUGHPeak2PHMD()}
\texttt{LALHOUGHAddPHMD2HD()}
\texttt{LALHOUGHIntegrHD2HT()}

Notes
26.4.2 Program TestDriveHough.c

Tests the construction

Usage

TestDriveHough [-d debuglevel] [-o outfile] [-f f0] [-p alpha delta]

Description

This program generates a patch grid, a vector of LUTs by changing the alpha component of the velocity orientation of the detector by a fixed amount in each of them, and a vector of peak-grams (all of them containing the same information). Similar to the previous test codes the patch is set at the south pole.

Then the program builds the set of PHMD, updates the cylinder and computes a Hough map at a given frequency using only one horizontal line set of PHMD, and outputs the result into a file.

By default, running this program with no arguments simply tests the subroutines, producing an output file called OutHough.asc. All default parameters are set from #defined constants.

The -d option sets the debug level to the specified value debuglevel. The -o flag tells the program to print the partial Hough map derivative to the specified data file outfile. The -f option sets the intrinsic frequency f0 at which build the LUT. The -p option sets the velocity orientation of the detector alpha, delta (in radians) for the first LUT (time-stamp).

Exit codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NORM</td>
<td>0</td>
<td>&quot;Normal exit&quot;</td>
</tr>
<tr>
<td>SUB</td>
<td>1</td>
<td>&quot;Subroutine failed&quot;</td>
</tr>
<tr>
<td>ARG</td>
<td>2</td>
<td>&quot;Error parsing arguments&quot;</td>
</tr>
<tr>
<td>BAD</td>
<td>3</td>
<td>&quot;Bad argument values&quot;</td>
</tr>
<tr>
<td>FILE</td>
<td>4</td>
<td>&quot;Could not create output file&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants TESTDRIVEHOUGH_E<name>, and the status descriptions in TESTDRIVEHOUGH_MSGE<name>. The source code with these messages is in TestDriveHough.c on line 1.143.

Uses

LALHOUHGPatchGrid() 
LALHOUGHParamPLUT() 
LALHOUGConstructPLUT() 
LALHOUGConstructSpacePHMD() 
LALHOUGHupdateSpacePHMDup() 
LALHOUGInitializeHT() 
LALHOUGConstructHMT() 
LALPrintError() 
LALMalloc() 
LALFree() 
LALCheckMemoryLeaks()

Notes

Author: Sintes, A. M., Krishnan, B.
$Id: TestDriveHough.c,v 1.7 2007/06/08 14:41:46 bema Exp$
26.4.3 Program TestDriveNDHough.c

Tests the construction

Usage

TestDriveNDHough [-d debuglevel] [-o outfile] [-f f0] [-p alpha delta] [-s patchSizeX patchSizeY]

Description

This program generates a patch grid, a vector of LUTs by changing the alpha component of the velocity orientation of the detector by a fixed amount in each of them, and a vector of peak-grams (all of them containing the same information). Similar to the previous test codes the patch is set at the south pole.

Then the program builds the set of PHMD, updates the cylinder and computes a Hough map at a given frequency using only one horizontal line set of PHMD, and outputs the result into a file.

By default, running this program with no arguments simply tests the subroutines, producing an output file called OutHough.asc. All default parameters are set from #defined constants.

The -d option sets the debug level to the specified value debuglevel. The -o flag tells the program to print the partial Hough map derivative to the specified data file outfile. The -f option sets the intrinsic frequency f0 at which build the LUT. The -p option sets the velocity orientation of the detector alpha, delta (in radians) for the first LUT (time-stamp).

Exit codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NORM</td>
<td>0</td>
<td>&quot;Normal exit&quot;</td>
</tr>
<tr>
<td>SUB</td>
<td>1</td>
<td>&quot;Subroutine failed&quot;</td>
</tr>
<tr>
<td>ARG</td>
<td>2</td>
<td>&quot;Error parsing arguments&quot;</td>
</tr>
<tr>
<td>BAD</td>
<td>3</td>
<td>&quot;Bad argument values&quot;</td>
</tr>
<tr>
<td>FILE</td>
<td>4</td>
<td>&quot;Could not create output file&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants TESTDRIVENDHOUGHC_E<name>, and the status descriptions in TESTDRIVENDHOUGHC_MSGE<name>. The source code with these messages is in TestDriveNDHough.c on line 1.144.

Uses

LALHOUGHPatchGrid()
LALNDHOUGHParamPLUT()
LALHOUGHConstructPLUT()
LALHOUGHConstructSpacePHMD()
LALHOUGHupdateSpacePHMDup()
LALHOUGHInitializeHT()
LALHOUGHConstructHMT()
LALPrintError()
LALMalloc()
LALFree()
LALCheckMemoryLeaks()

Notes
26.5 Header Statistics.h

Computes statistics of the Hough maps.

Synopsis

```
#include <lal/Statistics.h>
```

Given a total Hough map, this calculates the maximum number count, minimum number count, average and standard deviation and produces a histogram of the number counts.

Error conditions

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>1</td>
<td>&quot;Null Pointer&quot;</td>
</tr>
<tr>
<td>VAL</td>
<td>2</td>
<td>&quot;Invalid Value&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants STATISTICSH_E<name>, and the status descriptions in STATISTICSH_MSGE<name>. The source code with these messages is in Statistics.h on line 1.135.

Structures

```
struct HoughStats
```

This structure stores the statistics of a Hough map. The fields are:

- UINT4 maxCount
- UINT4 maxIndex[2]
- UINT4 minCount
- UINT4 minIndex[2]
- REAL8 avgCount
- REAL8 stdDev

Author: Krishnan, B., Sintes, A.M.

$Id: Statistics.h,v 1.8 2006/03/14 10:48:40 badri Exp $
26.5.1 Module Statistics.c

Calculation of statistical quantities of the Hough map and a histogram of the number counts.

Prototypes

```c
void LALHoughStatistics( LALStatus *status,
                         HoughStats *out,
                         HOUGHMapTotal *in)
```

```c
void LALHoughHistogram(LALStatus *status,
                        UINT8Vector *out,
                        HOUGHMapTotal *in)
```

Description

The function `LALHoughStatistics` calculates the maximum number count, minimum number count, average and standard deviation of a given total Hough map. The input `HOUGHMapTotal *in` is a total Hough map and the output is a structure `HoughStats *out`.

The `LALHoughHistogram` produces a histogram of the number counts in a total Hough map. The input is of type `HOUGHMapTotal *in` and the output `UINT4Vector *out`.

Uses

Notes
26.5.2 Program TestStatistics.c
Tests the statistics and the histogram number count of a given total Hough map.

Usage
TestStatistics [-d debuglevel] [-o outfile]

Description
This program creates a Hough map and ...

The -d option sets the debug level to the specified value debuglevel. The -o flag tells the program to print the histogram of the Hough number counts to the specified data file outfile.

Exit codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NORM</td>
<td>0</td>
<td>&quot;Normal exit&quot;</td>
</tr>
<tr>
<td>SUB</td>
<td>1</td>
<td>&quot;Subroutine failed&quot;</td>
</tr>
<tr>
<td>ARG</td>
<td>2</td>
<td>&quot;Error parsing arguments&quot;</td>
</tr>
<tr>
<td>BAD</td>
<td>3</td>
<td>&quot;Bad argument values&quot;</td>
</tr>
<tr>
<td>FILE</td>
<td>4</td>
<td>&quot;Could not create output file&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants TESTSTATISTICSC_E<name>, and the status descriptions in TESTSTATISTICSC_MSGE<name>. The source code with these messages is in TestStatistics.c on line 1.105.

Uses
LALHoughStatistics()
LALHoughHistogram()
LALPrintError()
LALMalloc()
LALFree()
LALCheckMemoryLeaks()

Notes

Author: Krishnan, B., Sintes, A.M.
$Id: TestStatistics.c,v 1.4 2007/06/08 14:41:46 bema Exp $
26.6 Header Velocity.h

Computation of instant and averaged velocities for a given detector and the like.

Synopsis

#include <lal/Velocity.h>

To find the velocity of a given detector at a given time, or the averaged velocity of a detector in a certain time interval.

Error conditions

<table>
<thead>
<tr>
<th>name</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>1</td>
<td>&quot;Null Pointer&quot;</td>
</tr>
<tr>
<td>VAL</td>
<td>2</td>
<td>&quot;Invalid Value&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants VELOCITYH_E<name>, and the status descriptions in VELOCITYH_MSGE<name>. The source code with these messages is in Velocity.h on line 1.135.

Structures

struct VelocityPar

This structure stores the parameters required by LALBarycenter to calculate Earth velocity at a given detector location.

LALDetector detector

EphemerisData *edat ephemeris data pointer from LALInitBarycenter

LIGOTimeGPS startTime start of time interval

REAL8 tBase duration of interval

REAL8 vTol fractional accuracy required for velocity

struct AvgVelPar

LALDetector detector

EphemerisData *edat
26.6.1 Module Velocity.c

Computation of instant and averaged velocities of a given detector and the like.

Prototypes

```c
void LALAvgDetectorVel( LALStatus *status,
                      REAL8 v[3], /**< [out] velocity vector */
                      VelocityPar *in /**< [in] input parameter structure */ )
```

```c
void LALAvgDetectorPos( LALStatus *status,
                        REAL8 x[3],
                        VelocityPar *in)
```

```c
void LALDetectorVel(LALStatus *status,
                    REAL8 v[3],
                    LIGOTimeGPS *time0,
                    LALDetector detector,
                    EphemerisData *edat)
```

```c
void LALDetectorPos(LALStatus *status,
                    REAL8 x[3],
                    LIGOTimeGPS *time0,
                    LALDetector detector,
                    EphemerisData *edat)
```

Description

The function `LALDetectorVel` finds the velocity of a given detector at a given time. It is basically a wrapper for LALBarycenter. The output is of the form `REAL8 v[3]`, and the input is a time `LIGOTimeGPS *time`, the detector `LALDetector detector`, and the ephemeris `EphemerisData *edat` from LALInitBarycenter.

The function `LALAvgDetectorVel` outputs the average velocity `REAL8 v[3]` of the detector during a time interval by using the trapezoidal rule. The input structure is of type `VelocityPar *in` containing all the required parameters.

Uses

- `LALFloatToGPS()`
- `LALMalloc()`
- `LALBarycenterEarth()`
- `LALBarycenter()`
- `LALFree()`

Notes
26.6.2 Program TestVelocity.c

Tests the calculation of the averaged velocity of a given detector.

Usage

TestVelocity [-d debuglevel] [-a accuracy]

Description

This program computes the averaged velocity of the GEO600 detector between the times 730000044 and 730003644 with a default accuracy of 0.01. The two ephemeris files (e.g., for data taken in 2003, sun03.dat and earth03.dat) are assumed to be in the directory lal/packages/pulsar/test/.

The -d option sets the debug level to the specified value debuglevel. The -a flag tells the program which accuracy to use.

Exit codes

<table>
<thead>
<tr>
<th>name</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NORM</td>
<td>0</td>
<td>&quot;Normal exit&quot;</td>
</tr>
<tr>
<td>SUB</td>
<td>1</td>
<td>&quot;Subroutine failed&quot;</td>
</tr>
<tr>
<td>ARG</td>
<td>2</td>
<td>&quot;Error parsing arguments&quot;</td>
</tr>
<tr>
<td>BAD</td>
<td>3</td>
<td>&quot;Bad argument values&quot;</td>
</tr>
<tr>
<td>FILE</td>
<td>4</td>
<td>&quot;Could not create output file&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants TESTVELOCITYC_E<name>, and the status descriptions in TESTVELOCITYC_MSGE<name>. The source code with these messages is in TestVelocity.c on line 1.110.

Uses

LALAvgDetectorVel()
LALPrintError()
LALMalloc()
LALFree()
LALCheckMemoryLeaks()

Notes
References


Chapter 27

Package pulsar: common routines

Teviet Creighton

This package provides routines for timing, metric calculation and mesh-generation relevant for pulsar searches.
27.1 Header PulsarTimes.h

Provides routines to transform among various time coordinates used in a pulsar search.

Synopsis

```c
#include <lal/PulsarTimes.h>
```

This header covers routines that computes time coordinate transformations, and derivatives of these transformations with respect to their parameters. The motivation is to provide a number of useful transformations for doing pulsar searches. For instance, one transformation might take you from the time measured at a point on Earth to the time measured in an inertial frame, synchronized according to a signal arriving from a particular direction in space. Another might transform from the inertial time coordinate to the proper time coordinate of a pulsar in a binary system. Another might gradually stretch the proper time coordinate of a pulsar that is spinning down, so that it appears to be at a constant frequency. Other transformations might be compositions of these, allowing you to go straight from detector time to the canonical rotation-defined time of the pulsar.

Mathematically, the transformation from one time \( t \) to another \( \tau \) is written as a function \( \tau(t) \). In general this function will depend on other parameters, such as the right ascension and declination of the source on the sky, or the latitude and longitude of the observer. Since in pulsar searches one is often concerned with how the transformation depends on these parameters, it is necessary to specify which parameters are allowed to vary and which will be treated as constants. We write the transformation as:

\[
\tau(t, \vec{\lambda}, \{p\}),
\]

where \( \vec{\lambda} = (\lambda_1, \ldots, \lambda_n) \) are the parameters that we will allow to vary, while \( \{p\} \) are the parameters that will be treated as constant. As the notation suggests, the variable parameters must be representable in a real vector space, while the constant parameters need not be real numbers; they may be integers, names, flags, or anything else required by the transformation function.

The modules under this header will typically provide function pairs of the form:

```c
void LALTau( LALStatus *,
             REAL8 *tau,
             REAL8Vector *variables,
             PulsarTimesParamStruc *constants );
```

```c
void LALDTau( LALStatus *,
              REAL8Vector *dTau,
              REAL8Vector *variables,
              PulsarTimesParamStruc *constants );
```

The actual function names will be different; these are just examples. The function \texttt{LALTau()} computes the transformation, while \texttt{LALDTau()} computes the transformation and its derivatives with respect to the parameters in \texttt{*variables}. The arguments are described below:

- \texttt{*stat} This is the universal status structure required by all LAL functions.
- \texttt{*tau} This stores the returned value of \( \tau(t, \vec{\lambda}, \{p\}) \).
- \texttt{*variables} This is a length \( n + 1 \) vector storing the arguments of \( \tau \) that are considered to be “variable”; that is, \( t \) and \( \vec{\lambda} \). They are stored as follows:
  ```c
  variables->data[0] = t,
  variables->data[i] = \lambda_i, \text{ where } i = 1, \ldots, n.
  ```
- \texttt{*constants} This stores the constant parameters \( \{p\} \), in a format described in the Structures section below.
- \texttt{*dTau} This is a length \( n + 2 \) vector storing the value of \( \tau(t, \vec{\lambda}, \{p\}) \) and its derivatives with respect to its variable arguments, in the following format:
  ```c
  dTau->data[0] = \tau
  dTau->data[1] = \partial \tau / \partial t
  ```
\[ d\tau \rightarrow d\text{data}[i+1] = \partial \tau / \partial \lambda^i, \quad \text{where} \quad i = 1, \ldots, n. \]

It may seem redundant that both \texttt{LALTau()} and \texttt{LALDTau()} compute and return the value of \( \tau \), especially since returning \( \tau \) in \texttt{*dTau} messes up an otherwise elegant indexing scheme. The reason is that many of the calculations involved in computing the derivatives of \( \tau \) are also used in calculating \( \tau \) itself, and it would be inefficient to have to repeat them by calling both functions. \texttt{LALTau()} is provided simply as a shortcut for those occasions when you do not need the derivatives.

It is also worth noting that different pulsar searches may involve the same transformations but vary different sets of parameters. There are two approaches to dealing with this. The simplest is usually to use a function that treats all of the parameters as variable, and then ignore (or set to zero) the derivatives that aren’t used. The more computationally efficient approach is to write separate pairs of routines that place different parameters in \texttt{*variables} and \texttt{*constants}, although this may require recompiling the library.

**Error conditions**

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUL</td>
<td>1</td>
<td>&quot;Null pointer&quot;</td>
</tr>
<tr>
<td>BAD</td>
<td>2</td>
<td>&quot;Bad parameter values&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants \texttt{PULSARTIMESH_E<name>}, and the status descriptions in \texttt{PULSARTIMESH_MSGE<name>}. The source code with these messages is in \texttt{PulsarTimes.h} on line 1.161.

**Types**

**Structure PulsarTimesParamStruc**

This structure stores a superset of all constant parameters required by the functions provided by this header \texttt{PulsarTimes.h}. Although the structure is quite large, it is supposed to store only constants, so only one structure need be allocated for each transformation type used in the overall algorithm. It is passed by pointer, so its size entails no overhead. A basic description of the current fields is given below, but a detailed discussion of those fields must be deferred to the documentation of the individual modules that use those particular fields. The fields are:

- \texttt{LIGOTimeGPS epoch} A reference detector time. All other times in the transformation are represented as \texttt{REAL8} numbers, giving the time in seconds since \texttt{epoch}.
- \texttt{REAL8 t0} A reference time for a particular transformation, normally defined such that \( \tau(t_0) = 0 \).
- \texttt{REAL8 tAutumn} Time of the first autumnal equinox following \texttt{epoch}.
- \texttt{REAL8 tMidnight} Time of the first sidereal midnight following \texttt{epoch}.
- \texttt{REAL8 latitude} Detector north latitude, in radians.
- \texttt{REAL8 longitude} Detector east longitude (i.e. counterclockwise about the north pole), in radians.
- \texttt{EphemerisData ephemeris} Ephemeris data containing positions, velocities, etc... of Earth and Sun for the year under consideration.
- \texttt{LALDetector site} The particular detector under consideration.

The following fields are used by the module \texttt{TComp.c}, which composes two transformations \( t_1(t) \) and \( t_2(t) \) into an overall transformation \( t_c(t) = t_2(t_1(t)) \).

- \texttt{void *t1( LALStatus *, REAL8 *, REAL8Vector *, PulsarTimesParamStruc * )} The first of the pair of transformations to be composed.
- \texttt{void *t2( LALStatus *, REAL8 *, REAL8Vector *, PulsarTimesParamStruc * )} The second of the pair of transformations to be composed.
- \texttt{void *dt1( LALStatus *, REAL8Vector *, REAL8Vector *, PulsarTimesParamStruc * )} The time derivative function corresponding to \texttt{*t1()}. 


void *dt2( LALStatus *, REAL8Vector *, REAL8Vector *, PulsarTimesParamStruc * ) The time derivative function corresponding to *t2().

PulsarTimesParamStruc *constants1 The constant parameters used by *t1().

PulsarTimesParamStruc *constants2 The constant parameters used by *t2().

UINT4 nArgs The number of variable parameters $\lambda^k$ to be sent to the function *t1().
27.1.1 Module GetEarthTimes.c

Computes the next sidereal midnight and autumnal equinox.

Prototypes

```c
void LALGetEarthTimes( LALStatus *stat, PulsarTimesParamStruc *times )
```

Description

This function takes a GPS time from the parameter field `times->epoch` and uses it to assign the fields `times->tAutumn` and `times->tMidnight`, which are REAL8 representations of the time in seconds from `times->epoch` to the next autumnal equinox or sidereal midnight, respectively. This routine was written under the PulsarTimes.h header because these quantities are vital for performing pulsar timing: they characterize the Earth's orbital and rotational phase, and hence the Doppler modulation on an incoming signal. See the PulsarTimes.h header for more information about the PulsarTimesParamStruc structure.

Algorithm

The routine first computes the Greenwich mean sidereal time at `times->epoch` using LALGPStoMST1(). The next sidereal midnight (at the Prime Meridian) is simply 86400 seconds minus that sidereal time.

Next the routine computes the time of the next autumnal equinox. The module contains an internal list of GPS times of autumnal equinoxes from 1992 to 2020, given to the nearest minute; this is certainly enough accuracy for use with the routines in TBaryPtolemaic(). If the specified time `times->epoch` is after the 2020 autumnal equinox, or more than a year before the 1992 equinox, then the next equinox is extrapolated assuming exact periods of length LAL_YRSID_SI.

When assigning the fields of `*times`, it is up to the user to choose a `times->epoch` that is close to the actual times that are being considered. This is important, since many computations use a REAL8 time variable whose origin is the time `times->epoch`. If this is too far from the times of interest, the REAL8 time variables may suffer loss of precision.

Uses

LALGPStoGMST1()

Notes
27.1.2 Module TBaryPtolemaic.c

Computes the barycentric arrival time of an incoming wavefront using a circular model of the Earth’s orbit.

Prototypes

```c
void LALTBaryPtolemaic( LALStatus *stat,
                        REAL8 *tBary,
                        REAL8Vector *variables,
                        PulsarTimesParamStruc *constants )

void LALDTBaryPtolemaic( LALStatus *stat,
                         REAL8Vector *dtBary,
                         REAL8Vector *variables,
                         PulsarTimesParamStruc *constants )
```

Description

These routines compute the barycentric time transformation and its derivatives. That is, if a signal originating from a right ascension \( \alpha \) and declination \( \delta \) on the sky and arrives at the detector at a time \( t \), then it will pass the centre of the solar system at a time \( t_b(t, \alpha, \delta) \).

The routines obey the calling convention presented in the header `PulsarTimes.h`, with \( n = 2 \) variable parameters \( \lambda_1 = \alpha, \lambda_2 = \delta \) (both measured in radians). The constant parameter fields used by these routines are `constants->tAutumn`, `constants->tMidnight`, `constants->latitude`, and `constants->longitude`.

Note that \(*variables\) must have a length of at least 3, and can be longer, but values beyond the third are ignored. \(*dtBary\) must be at least of length 1, and the number of derivatives computed is determined by \(*dtBary->length-1\). All elements beyond the fourth will be set to zero.

Algorithm

Let \( \hat{n}(\alpha, \delta) \) be the unit vector to the source on the sky, and \( x(t) \) be the position of the detector relative to the solar system barycentre. Then, ignoring relativistic effects and considering only the Roemer time delay, the barycentred time is:

\[
t_b(t, \alpha, \delta) = t + \frac{x(t)}{c} \cdot \hat{n}(\alpha, \delta).
\]

Choosing a right-handed coordinate system with \( \hat{e}_x \) toward the vernal equinox and \( \hat{e}_z \) toward celestial North, the components of \( \hat{n}(\alpha, \delta) \) can be written as:

\[
\begin{align*}
    n_x &= \cos \alpha \cos \delta, \\
    n_y &= \sin \alpha \cos \delta, \\
    n_z &= \sin \delta.
\end{align*}
\]  

(27.1)

In a first-order Ptolemaic model of the solar system, the rotation of the Earth and its orbit can be treated as simple circular motions, inclined to one another at an angle \( i = 23.5^\circ \). This approximation is insufficient for actual signal demodulation, since it can result in timing errors of several seconds due to the Earth’s orbital eccentricity. However, the derivatives of \( t_b \) will not be off by more than a percent or so, which is good enough for, say, computing a parameter-space metric. We define angles of rotation and revolution \( \theta_{\text{rot}}(t) = 2\pi(t - t_{\text{midnight}})/P_{\text{rot}} \), \( \theta_{\text{rev}}(t) = 2\pi(t - t_{\text{autumn}})/P_{\text{rev}} \), where \( P_{\text{rot}} \) is a sidereal day, \( P_{\text{rev}} \) a sidereal year, \( t_{\text{midnight}} \) the time of a sidereal midnight at the prime meridian, and \( t_{\text{autumn}} \) is the time of an autumnal equinox. In other words, \( \theta_{\text{rot}} = 0 \) corresponds to the Earth lying on the positive \( \hat{e}_x \) axis and \( \theta_{\text{rev}} = 0 \) corresponds to a point with zero longitude lying in the \( \hat{e}_z \) direction from the Earth’s axis. If the detector has a north latitude \( \lambda \) and an east longitude \( l \), then the detector position is:

\[
\begin{align*}
    x &= R \cos \theta_{\text{rev}} + r \cos(\theta_{\text{rot}} + l) \cos \lambda, \\
    y &= R \sin \theta_{\text{rev}} \cos i + r \sin(\theta_{\text{rot}} + l) \cos \lambda, \\
    z &= R \sin \theta_{\text{rev}} \sin i + r \sin \lambda,
\end{align*}
\]
where $R$ is the Earth-Sun distance and $r$ is the Earth’s radius. The time dependence of $x$ is implicit in the time dependence of $\theta_{\text{rot}}$ and $\theta_{\text{rev}}$.

Differentiating with respect to $t$, we obtain:

$$\frac{\partial b(t, \alpha, \delta)}{\partial t} = 1 + \frac{v(t)}{c} \cdot \hat{n}(\alpha, \delta),$$

where:

$$v_x = -V \sin \theta_{\text{rev}} - v \sin(\theta_{\text{rot}} + l) \cos \lambda,$$

$$v_y = V \cos \theta_{\text{rev}} \cos i + v \cos(\theta_{\text{rot}} + l) \cos \lambda,$$

$$v_z = V \cos \theta_{\text{rev}} \sin i,$$

$V = 2\pi R / P_{\text{rev}}$ is the Earth’s orbital velocity, and $v = 2\pi r / P_{\text{rot}}$ is the Earth’s equatorial rotational velocity.

Differentiating with respect to $\alpha$ gives:

$$\frac{\partial b(t, \alpha, \delta)}{\partial \alpha} = \frac{x(t)}{c} \cdot \frac{\partial \hat{n}(\alpha, \delta)}{\partial \alpha},$$

where $\partial \hat{n} / \partial \alpha$ is easily obtained from Eqs. 27.1. Similarly for $\delta$.

**Uses**

lalDebugLevel

**Notes**
27.1.3 Module TSpin.c

Computes the rotation-synchronized time coordinate for an object with smoothly-varying spin.

Prototypes

```c
void LALTSpin( LALStatus *stat,
    REAL8 *tSpin,
    REAL8Vector *variables,
    PulsarTimesParamStruc *constants )

void LALDTSpin( LALStatus *stat,
    REAL8Vector *dtSpin,
    REAL8Vector *variables,
    PulsarTimesParamStruc *constants )
```

Description

These routines compute the value and derivatives of a time transformation $t_s(t)$ from a physical (inertial) time coordinate to one that is synchronized with the rotation of an object (such as a pulsar). That is, the rotation period is constant in the new time coordinate, while the actual physical rotation rate may be changing gradually. The frequency drift $f(t)$ is characterized by frequency-normalized Taylor coefficients $f_k = (k!f)^{-1}\partial^k f/\partial t^k$.

The routines obey the calling convention presented in the header `PulsarTimes.h`, with $n$ variable parameters $\lambda^k = f_k$ (measured in Hz$^k$), where $k = 1, \ldots, n$. The only constant parameter field used by these routines is $\text{constants->t0}$, which is the time when the Taylor coefficients $f_k$ are evaluated. $t_s$ is defined to be zero at that time.

*variables can be a vector of arbitrary length; for consistency, *dtSpin must be one element longer.

Algorithm

If the frequency of a rotating body is varying in some suitably smooth manner, then it can be represented as a Taylor series:

$$f(t) = f_0 \left(1 + \sum_{k=1}^{n} f_k [t - t_0]^k \right).$$

It is worth pointing out that the parameters $f_k$ are frequency-normalized Taylor coefficients, so their magnitudes represent, not the frequency itself, but the timescale on which the frequency is varying. That is, if the frequency is changing on some timescale $\mathcal{T}$, then $f_k$ is typically of order $\mathcal{T}^{-k}$.

The spin-synchronized time $t_s(t)$ counts off intervals of constant rotational phase, so it must be proportional to $\int f(t) \, dt$. The integration constant is set by the convention that $t_s(t_0) = 0$. The constant of proportionality is set by demanding that, at $t = t_0$, both $t_s$ and $t$ measure time at the same rate. This gives us the transformation:

$$t_s(t) = t - t_0 + \sum_{k=1}^{n} \frac{f_k}{k+1} (t - t_0)^{k+1}.$$

The time derivative of this transformation is of course just the the factor in the equation for $f(t)$ that we started with:

$$\frac{\partial t_s(t)}{\partial t} = 1 + \sum_{k=1}^{n} f_k [t - t_0]^k.$$

The derivatives with respect to $f_k$ are similarly trivial:

$$\frac{\partial t_s(t)}{\partial f_k} = \frac{(t - t_0)^{k+1}}{k+1}. $$
Uses

lalDebugLevel

Notes
27.1.4 Module TComp.c

Computes the composition of two time transformations.

Prototypes

```c
void LALTComp( LALStatus *stat,
               REAL8 *tComp,
               REAL8Vector *variables,
               PulsarTimesParamStruc *constants )
```

```c
void LALDTComp( LALStatus *stat,
                REAL8Vector *dtComp,
                REAL8Vector *variables,
                PulsarTimesParamStruc *constants )
```

Description

These routines compute the value and derivatives of a time transformation $t_c(t)$ that is the composition of two other transformations $t_1$ and $t_2$; that is, $t_c(t) = t_2(t_1(t))$. More precisely, the transformation is $t_c(t, \vec{\lambda}(1) \oplus \vec{\lambda}(2)) = t_2[t_1(t, \vec{\lambda}(1)), \vec{\lambda}(2)]$. Note that $\vec{\lambda}(1)$ and $\vec{\lambda}(2)$ are assumed to represent independent sets of parameters. If there is any overlap between the parameter sets, DTComp() will not correctly compute the derivatives of $t_c(t)$ (although the value of $t_c$ will still be correct).

The routines obey the calling convention presented in the header PulsarTimes.h. The contents of *variables are, firstly, the time $t$ that will be sent to $t_1(t, \vec{\lambda}(1))$; next, the $n$ parameters $\lambda^1, \ldots, \lambda^n$ that will be sent to $t_1(t, \vec{\lambda}(1))$ as $\lambda^1_{(1)}, \ldots, \lambda^n_{(1)}$; last, the $m$ parameters $\lambda^{n+1}, \ldots, \lambda^{n+m}$ that will be sent to $t_2(t_1, \vec{\lambda}(2))$ as $\lambda^{1}_{(2)}, \ldots, \lambda^{m}_{(2)}$. Here $n$ and $m$ are the number of variable parameters expected by the transformations $t_1$ and $t_2$, so that $\text{variables->length} = n + m + 1$.

The constant parameter fields used by these routines are:

- **constants->t1** A function pointer to the function that evaluates $t_1(t)$.
- **constants->t2** A function pointer to the function that evaluates $t_2(t)$.
- **constants->dt1** A function pointer to the function that evaluates $t_1(t)$ and its derivatives.
- **constants->dt2** A function pointer to the function that evaluates $t_2(t)$ and its derivatives.
- **constants->constants1** A pointer to the constant parameters used by **constants->t1** and **constants->dt1**.
- **constants->constants2** A pointer to the constant parameters used by **constants->t2** and **constants->dt2**.
- **constants->nArgs** The number $n$ of variable parameters to be sent to $t_1(t)$.

Note that the number of variable parameters to be sent to $t_2(t)$ is not specified in **constants;** after sending the first **constants->nArgs** of them to $t_1(t)$, the remaining parameters (however many they are) are sent to $t_2(t)$. This is particularly useful for pulsar timing routines, where the last function in the composition chain is often a transformation that corrects for the pulsar spindown, using an arbitrary number of spindown parameters. The number of spindown parameters desired is then specified unambiguously by setting **variables->length.** Note however that **dtComp** must always have a length exactly one greater than **variables.
Algorithm

Computing the value of $t_c$ is trivial:

$$t_c(t) = t_2(t_1(t)).$$

The only trickiness is in handling the parameters, which is done using a local REAL8Vector variablesIn. This vector is not given its own memory; instead, its data field is pointed at either the first or the last block of parameters in variable->data.

Computing the derivatives of $t_c$ is not much trickier. For the time variable and the first $n$ parameters, the chain rule gives us:

$$\frac{\partial t_c(t)}{\partial t} = \frac{\partial t_2(t_1)}{\partial t_1} \frac{\partial t_1(t)}{\partial t},$$

$$\frac{\partial t_c(t)}{\partial \lambda^i} = \frac{\partial t_2(t_1)}{\partial t_1} \frac{\partial t_1(t)}{\partial \lambda_{(1)}^1}, \quad i = 1, \ldots, n.$$  

For the remaining parameters,

$$\frac{\partial t_c(t)}{\partial \lambda^j} = \frac{\partial t_2(t_1)}{\partial \lambda_{(2)}^1} \frac{\partial t_1(t)}{\partial \lambda^{n+j}}, \quad j = n + 1, \ldots.$$  

As noted in the module description, the derivatives will not be evaluated correctly if there is overlap between the two parameter sets $\vec{\lambda}_{(1)}$ and $\vec{\lambda}_{(2)}$. In particular, if some variable $\alpha$ is represented both by $\lambda^i = \lambda_{(1)}^i$ and $\lambda^{n+j} = \lambda_{(2)}^j$, the value of $\partial t_c/\partial \alpha$ is neither given by $\partial t_c/\partial \lambda^i$ nor by $\partial t_c/\partial \lambda^{n+j}$, but by:

$$\frac{\partial t_c}{\partial \alpha} = \frac{\partial t_c}{\partial \lambda^{n+j}} + \frac{\partial t_c}{\partial \lambda^i} \frac{\partial t_1}{\partial \lambda_{(1)}^i}.$$  

While this is not especially difficult to evaluate, it is impossible to code without giving DTComp() some way of knowing which parameters in $\vec{\lambda}_{(1)}$ and $\vec{\lambda}_{(2)}$ represent the same physical quantity. Such a scheme is not implemented at present.

Uses

lalDebugLevel

Notes
27.1.5 Module **DTEphemeris.c**

Computes the barycentric arrival time of an incoming wavefront using accurate ephemeris-based data files of the Sun and Earth’s motions.

**Prototypes**

```c
void LALDTEphemeris( LALStatus *status,
                     REAL8Vector *drv,
                     REAL8Vector *var,
                     PulsarTimesParamStruc *tev )
```

```c
void LALTEphemeris( LALStatus *status,
                    REAL8 *tBary,
                    REAL8Vector *var,
                    PulsarTimesParamStruc *tev )
```

**Description**

These routines compute the barycentric time transformation and its derivatives. That is, if a signal originating from a right ascension $\alpha$ and declination $\delta$ on the sky and arrives at the detector at a time $t$, then it will pass the centre of the solar system at a time $t_b(t, \alpha, \delta)$.

The input/output features of this function are nearly identical to those of **DTBaryPtolemaic()**, whose documentation should be consulted for the details. One important difference in calling this function is that the user has to supply the initialised ephemeris-data in the `PulsarTimesParamStruc->ephemeris` and the detector-data in `PulsarTimesParamStruc->site`. **DTBaryPtolemaic()** uses the Ptolemaic approximation to model the Earth/Sun system, while **DTEphemeris()** uses accurate ephemeris data read in from files in the calling function, and passed into **DTEphemeris()** using the `EphemerisData` structure, which is a member of the `PulsarTimesParamStruc`.

**Algorithm**

Uses

- `lalDebugLevel`  
- `LALBarycenterEarth()`  
- `LALBarycenter()`

**Notes**
27.2 Header LALBarycenter.h

Provides routines for transforming from arrival time at detector (GPS) to pulse emission time (TDB); i.e., for “barycentering” the measured astronomical time series.

Synopsis

#include <lal/LALBarycenter.h>

This header covers the routine LALBarycenter.c.

Error conditions

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>2</td>
<td>&quot;Null input to Barycenter routine.&quot;</td>
</tr>
<tr>
<td>OUTOFRANGE</td>
<td>4</td>
<td>&quot;tgps not in range of earth.dat file&quot;</td>
</tr>
<tr>
<td>OUTOFRANGES</td>
<td>8</td>
<td>&quot;tgps not in range of sun.dat file&quot;</td>
</tr>
<tr>
<td>BADSOURCEPOS</td>
<td>16</td>
<td>&quot;source position not in standard range&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants LALBARYCENTER_E<name>, and the status descriptions in LALBARYCENTERH_MSGE<name>. The source code with these messages is in LALBarycenter.h on line 1.84.

Structures

struct EphemerisFilenames

This structure contains two pointers to data files containing arrays of center-of-mass positions for the Earth and Sun, respectively. The tables are derived from the JPL ephemeris.

Files tabulate positions for one calendar year (actually, a little more than one year, to deal with overlaps). The first line of each table summarizes what is in it. Subsequent lines give the time (GPS) and the Earth’s position (x, y, z), velocity (v_x, v_y, v_z), and acceleration (a_x, a_y, a_z) at that instant. All in units of seconds; e.g. positions have units of seconds, and accelerations have units 1/sec.

CHAR *earthEphemeris File containing Earth’s position.

CHAR *sunEphemeris File containing Sun’s position.

struct EphemerisData

This structure contains all information about the center-of-mass positions of the Earth and Sun, listed at regular time intervals. The fields are

EphemerisFilenames ephiles Stucture giving names of the two files containing positions of Earth and Sun, resp., at evenly spaced times.

INT2 leap The number of leap seconds that have been inserted into UTC between Jan. 6, 1980 (= start of GPS calendar) and the current time tgps. But it’s perfectly OK to approximate this by number of leap sec inserted between Jan. 6, 1980 and Jan. 2 of year covered by this ephemeris file; e.g. leap = 13 for year 2000.

INT4 nentriesE The number of entries in Earth ephemeris table.

INT4 nentriesS The number of entries in Sun ephemeris table.

REAL8 dtable The spacing in sec. between consecutive intants in Earth ephemeris table.

REAL8 dstable The spacing in sec. between consecutive intants in Sun ephemeris table.

PosVelAcc *ephemE Array containing pos,vel,acc. of earth, as extracted from earth ephem file. Units are sec, 1, 1/sec.
LALBarycenter.h

PosVelAcc *ephemS Array containing pos,vel,acc. of sun, as extracted from sun ephem file. Units are sec, 1, 1/sec.

struct EarthState
Basic output structure of LALBarycenterEarth.c.

REAL8 einstein the einstein delay equiv TDB - TDT
REAL8 deinstein d(einstein)/d(tgps)
REAL8 posNow[3] Cartesian coords of Earth’s center at tgps, extrapolated from JPL DE405 ephemeris; units= sec.
REAL8 velNow[3] dimensionless velocity of Earth’s center at tgps, extrapolated from JPL DE405 ephemeris
REAL8 gastRad Greenwich Apparent Sidereal Time, in radians, at tgps. Its basically the angle thru which Earth has spun at given time. gast is like gmst, but has additional correction for short-term nutation.
REAL8 zA variable describing effect of lunisolar precession, at tgps
REAL8 thetaA variable describing effect of lunisolar precession, at tgps
REAL8 delpsi variable describing effect of Earth nutation, at tgps; see Explan. Supp. Astron. Almanac, pp.120.
REAL8 deleps variable describing effect of Earth nutation, at tgps
REAL8 se[3] vector that points from Sun to Earth at instant tgps, in DE405 coords; units = sec
REAL8 dse[3] d(se[3])/d(tgps). Dimensionless
REAL8 rse length of vector se[3]; units = sec
REAL8 drse d(rse)/d(tgps); dimensionless

struct BarycenterInput
Basic input structure to LALBarycenter.c.

LIGOTimeGPS tgps input GPS arrival time.
LALDetector site detector site info. NOTE: the site.location field must be modified to give the detector location in units of seconds (i.e. divide the values normally stored in site.location by LAL_C_SI).
REAL8 alpha Source right ascension in ICRS J2000 coords (radians).
REAL8 delta Source declination in ICRS J2000 coords (radians).
REAL8 dInv 1/(distance to source), in 1/sec.

struct EmissionTime
Basic output structure produced by LALBarycenter.c.

REAL8 deltaT t_e(TDB) - t_a(GPS) (+ constant = “light-travel-time from source to SSB”)
LIGOTimeGPS te pulse emission time t_e in TDB (plus constant = “light-travel-time from source to SSB”), in format of LIGOTimeGPS structure.
REAL8 tDot d(emission time in TDB)/d(arrival time in GPS)
REAL8 rDetector[3] Cartesian coords (0=x,1=y,2=z) of detector position at t_a (GPS), in ICRS J2000 coords. Units = sec.
REAL8 vDetector[3] Cartesian coords (0=x,1=y,2=z) of detector velocity at t_a (GPS), in ICRS J2000 coords. Dimensionless.

Author: Cutler, C.
$Id: LALBarycenter.h,v 1.12 2007/06/08 14:41:50 bema Exp $
27.2.1 Module LALBarycenter.c

Converts from detector arrival time (recorded by GPS clock) to pulse emission time, in TDB.

Prototypes

```c
void LALBarycenterEarth(LALStatus *stat,
    EarthState *earth, /**< [out] the earth's state at time tGPS */
    const LIGOTimeGPS *tGPS, /**< [in] GPS time tGPS */
    const EphemerisData *edat) /**< [in] ephemeris-files */
```

```c
void LALBarycenter(LALStatus *stat,
    EmissionTime *emit, /**< [out] emission-time information */
    const BarycenterInput *baryinput, /**< [in] info about detector and source-location */
    const EarthState *earth) /**< [in] earth-state (from LALBarycenterEarth()) */
```

Description

LALBarycenterEarth() computes the position and orientation of the Earth, at some arrival time $t_a$, specified LIGOTimeGPS input structure. The Einstein delay is also calculated. The results are stored in the EarthState output structure, which can then be fed as input to LALBarycenter(). The idea is that LALBarycenterEarth() calculates quantities that are independent of the source location and detector position on Earth. Thus this function is called ONCE for every desired arrival time; the results are then re-used as one steps around the sky (and/or changes detector) at that time.

LALBarycenter() transforms from detector arrival time $t_a$ in GPS (as specified in the LIGOTimeGPS structure) to pulse emission time $t_e$, in TDB. (Actually, the returned $t_e$ is the emission time plus the constant light-travel-time from source to SSB.) The inputs to LALBarycenter(), through the BarycenterInput structure, are the source location, detector site identifier, and GPS arrival time. The function returns the emission time $t_e(t_a)$, the derivative $dt_e/dt_a$, and the difference $t_e(t_a) − t_a$ through the EmissionTime structure. The emission time $t_e(t_a)$ is returned in the LIGOTimeGPS format, while the other two quantities are REAL8s.

Algorithm

The function "corrects" the pulse arrival time by removing the Roemer delay, (including effects of Moon, planets, and the Earths time-varying spin axis and spin rate), Einstein delay, and Shapiro delay. Accuracy is better than 3 $\mu$s. Full details will be in monograph by Cutler in Los Alamos preprint archive.

Notes

Author: Cutler, C.

$Id: LALBarycenter.c,v 1.10 2007/07/18 16:32:02 bema Exp $
27.3.1 Program LALBarycenterTest.c

Tests the routine LALBarycenter(). Exercises some of the error conditions and makes sure that they work.

Usage

LALBarycenterTest

Description

This program demonstrates the use of LALBarycenter.c. The two ephemeris files specified in the EphemerisFilenames structure (e.g., for data taken in 1998, sun98.dat and earth98.dat) are assumed to be in the same directory as the program as the test program.

Exit codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOM</td>
<td>0</td>
<td>&quot;Nominal exit&quot;</td>
</tr>
<tr>
<td>OPEN</td>
<td>1</td>
<td>&quot;Error checking failed to catch missing ephemeris file.&quot;</td>
</tr>
<tr>
<td>OUTOFRANGE</td>
<td>4</td>
<td>&quot;Failed to catch that tgps not in range of earth.dat file&quot;</td>
</tr>
<tr>
<td>OUTOFRANGES</td>
<td>8</td>
<td>&quot;Failed to catch that tgps not in range of sun.dat file&quot;</td>
</tr>
<tr>
<td>BADSOURCEPOS</td>
<td>16</td>
<td>&quot;Failed to catch bad source position&quot;</td>
</tr>
<tr>
<td>EPHFILE</td>
<td>32</td>
<td>&quot;Failed to catch error reading ephemeris file.&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants LALBARYCENTERTESTC_E<name>, and the status descriptions in LALBARYCENTERTESTC_MSGE<name>. The source code with these messages is in LALBarycenterTest.c on line 1.89.

Uses

lalDebugLevel
LALMalloc()
LALFree()
LALBarycenterInit()
LALBarycenterEarth()
LALBarycenter()
LALCheckMemoryLeaks()

Notes
# 27.4 Header FlatMesh.h

Provides routines to place search meshes for parameter spaces with constant parameter metrics.

**Synopsis**

```c
#include <lal/FlatMesh.h>
```

This header covers routines that lay out a mesh of points on an \( n \)-dimensional parameter space \( \{x^a\} \). The mesh points are placed in a unit-cube lattice in orthonormalized coordinates \( y^a = T^a_b x^b \), where \( T^a_b \) is an arbitrary but constant transformation matrix. This describes a fairly general procedure, but within LAL these routines are used for a specific purpose: to define a set of parameter values required to search a space of parameterized signals.

In most optimal signal-detection strategies, each signal that one might detect is parameterized by an \( n \)-tuple of parameters \( x^a = (x_1, \ldots, x_n) \). To detect a signal whose parameters are not known in advance, a typical search algorithm performs a series of targeted searches over a discrete set of parameter values \( x^a_1, x^a_2, \ldots \). If the unknown signal’s parameters happen to match one of the target points, it will be detected with maximum signal strength; otherwise, its effective signal strength is reduced according to the “distance” between its parameter values and the closest search point. The mismatch \( m(x^a, \Delta x^a) \) is defined as the fractional reduction in effective signal strength of a signal with parameters \( x^a + \Delta x^a \) in a search targeted at signal parameters \( x^a \). The mismatch can define a local distance measure on the parameter space, since for small values it approaches a quadratic form:

\[
m(x^a, \Delta x^a) = g_{bc}(x^a) \Delta x^b \Delta x^c + O(\Delta x^a)^3.
\]

The matrix \( g_{bc} \) is called the mismatch *metric*, and in general can vary as a function of the central target point \( x^a \).

One of the main goals in searching the signal parameter space is to choose a set of target points that is as small as possible, while still ensuring that an unknown signal will lie close enough to a target point that it will lose no more than some fraction \( m_{\text{thresh}} \) of its signal strength. If the mismatch metric \( g_{ab} \) is constant, this is relatively simple: the search points can be placed on a mesh that is rectilinear and uniformly-spaced in the eigenspace of \( g_{ab} \). This is described below.

Let \( \lambda^{(1)}, \ldots, \lambda^{(n)} \) be the \( n \) eigenvalues of \( g_{ab} \), and \( e^{a}_{(1)}, \ldots, e^{a}_{(n)} \) be the corresponding unit eigenvectors. Then the vectors \( \{e^{a}_{(i)}/\sqrt{\lambda^{(i)}}\} \) define a new coordinate basis in which coordinate distances correspond to metric distances. The simplest covering of the parameter space is to lay out a cubic mesh in this coordinate basis. If \( s \) is the side length of one of these cubes, then the maximum mismatch (in the quadratic approximation) between a point in the interior and a vertex is \( (s/2)\sqrt{n} \). We wish this to be no greater than some given threshold value \( m_{\text{thresh}} \). This means that the eigenvectors pointing from a mesh point to its immediate neighbours are of the form \( \pm 2m_{\text{thresh}}[n\lambda^{(i)}]^{-1/2} e^{a}_{(i)} \).

Let us define a transformation matrix \( M^a_b \) by:

\[
M^a_j = 2m_{\text{thresh}}[n\lambda^{(j)}]^{-1/2} e^{a}_{(j)}.
\]

Then the parameters \( x^a \) and the orthonormalized coordinates \( y^b \) are related by:

\[
x^a = M^a_b y^b, \quad y^b = (M^{-1})^b_a x^a.
\]

The search mesh can thus be placed as a unit-cube lattice in the \( y^b \) coordinate basis and then transformed back to find the mesh points in the \( x^a \) coordinates.

This header and its associated modules are placed in the pulsar package because they were originally intended for use in targeted pulsar searches, where the sky position is known but the frequency and spindown are not. In a search over the Taylor coefficients of the frequency function, the associated mismatch metric is nearly constant, with corrections on the order of the observation time over the spindown timescale.
Error conditions

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUL</td>
<td>1</td>
<td>&quot;Unexpected null pointer in arguments&quot;</td>
</tr>
<tr>
<td>OUT</td>
<td>2</td>
<td>&quot;Output handle points to a non-null pointer&quot;</td>
</tr>
<tr>
<td>MEM</td>
<td>3</td>
<td>&quot;Memory allocation error&quot;</td>
</tr>
<tr>
<td>DIM</td>
<td>4</td>
<td>&quot;Inconsistent parameter space dimension&quot;</td>
</tr>
<tr>
<td>LEN</td>
<td>5</td>
<td>&quot;Too few points specified&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants FLATMESHH_E<name>, and the status descriptions in FLATMESHH_MSGE<name>. The source code with these messages is in FlatMesh.h on line 1.145.

Types

Structure FlatMeshParamStruc

This structure stores and passes parameters for computing a search mesh by the routines in the module FlatMesh.c: it defines the transformation matrices between signal parameters and orthonormalized coordinates, a rectangular volume containing the region to be covered, and a function defining the covered region as a subset of the rectangular volume. The fields are:

REAL4VectorSequence *matrix The matrix $M^a_b$ (above) that transforms from orthonormalized to parameter coordinates.

REAL4VectorSequence *matrixInv The inverse of *matrix.

REAL4Vector *xMin A vector defining one corner of a rectangular region in parameter space that contains the region to be covered. By convention, *xMin is the corner where each parameter has its minimum value.

REAL4Vector *xMax A vector defining the opposite corner of the rectangular region above. By convention, *xMax is the corner where each parameter has its maximum value.

void (*intersection)( LALStatus *, REAL4VectorSequence , REAL4VectorSequence * ) The function that restricts the mesh to cover only the desired search region.

REAL4VectorSequence *controlPoints A set of "control points" to be passed as the second argument to *intersection (above), defining the size and shape of the desired search region.
27.4.1 Module FlatMesh.c

Places a mesh of templates on an $n$-dimensional rectilinear parameter space.

Prototypes

```c
void LALCreateFlatMesh( LALStatus *stat, 
                        REAL4VectorSequence **mesh, 
                        FlatMeshParamStruc *params );

void LALRectIntersect( LALStatus *stat, 
                        REAL4VectorSequence *mesh, 
                        REAL4VectorSequence *controlPoints );
```

Description

LALFlatMesh() lays out a mesh on an $n$-dimensional parameter space according to the method discussed in FlatMesh.h. It first creates a unit-cube lattice in $y^a$ over a rectilinear region large enough to cover the search area completely, and then calls the routine `params->intersection()` to restrict the list to those mesh points that lie inside the search region. (If this function pointer is NULL, then no restriction is done.) The list of mesh point locations is returned in **mesh; see FlatMesh.h for a description of the fields in *params.

LALRectIntersect() is a simple routine that restricts a parameter mesh *mesh to a rectilinear region defined by the first two vectors $x^a(1), x^a(2)$ in the sequence *controlPoints (other vectors in the sequence are ignored): the region is $[x_a^1(1), x_a^1(2)] \times \cdots \times [x_a^n(1), x_a^n(2)]$. In general the values of mesh->length and the pointer mesh->data will be changed when the dataset is reduced.

Algorithm

The algorithm in LALFlatMesh() initially lays a mesh over a region much larger than is ultimately required. First, in the $x^a$ coordinate system, the minimum and maximum parameter values params->xMin and params->xMax are used to define a rectilinear region $[x_a^1_{\text{min}}, x_a^1_{\text{max}}] \times \cdots \times [x_a^n_{\text{min}}, x_a^n_{\text{max}}]$ that is a superset of the desired search region. Upon transformation to the $y^a$ coordinate system, this superset is now a parallelogram; the algorithm then defines a super-superset $[y_a^1_{\text{min}}, y_a^1_{\text{max}}] \times \cdots \times [y_a^n_{\text{min}}, y_a^n_{\text{max}}]$ that completely encloses the parallelogram. A unit-cube mesh is placed on this super-superset, transformed back to $x^a$ coordinates, and then passed to params->intersect to restrict it to the region of interest.

Obviously if the desired region in $x^a$ coordinates is highly elongated along a non-principal axis, and if the transformation from $x^a$ to $y^a$ involves both rotation and elongation of principal axes, then the final intersection may eliminate all but a tiny fraction of the mesh points generated initially. However, laying out mesh points is rarely expected to be the dominant computational cost in any analysis, so some inefficiency can be tolerated. Furthermore, since the definition of signal parameters $x^a$ is somewhat arbitrary, one can cleverly choose an initial coordinate system that is aligned with the preferred axes of the desired search area, or with the preferred axes of the mismatch metric, whichever will improve performance the most.

LALRectIntersect() performs the dataset reduction “in place”, within the memory block allocated to mesh->data, and then uses LALRealloc() to reduce the memory storage accordingly. In most cases this will mean allocating a new block, copying the reduced dataset over, and then freeing the old block. However, when debugging is turned off, LALRealloc() reverts to the lower-level memory management function realloc(); this routine can often simply deallocate the excess memory without having to touch the reduced dataset at the start of the block.

Uses

```c
lalDebugLevel
LALInfo() LALPrintError()
LALRealloc()
LALMalloc() LALFree()
LALSCreateVectorSequence() LALSDestroyVectorSequence()
```
27.4.2 Program FlatMeshTest.c

Creates a template mesh for an arbitrary but constant $n$-dimensional mismatch metric.

Usage

FlatMeshTest [-o outfile] [-d debuglevel] [-m mismatch]
[eigenvectorfile inversefile rangefile]

Description

This test program creates a template mesh for a parameter space with a constant mismatch metric. The following option flags are accepted:

- -o Writes the output mesh to the file outfile.
- -d Sets the debug level to debuglevel.
- -m Sets the maximum allowed mismatch to mismatch, a positive number less than 1.

Once the above options are processed, any remaining command-line arguments must be the names of three files containing information about the eigenvectors of the metric and the desired search range; these files are described below. They are read using the function LALSReadVectorSequence(). If the -o option is not specified, results are written to stdout; if other options or arguments are not specified, the information is taken from #defined constants.

eigenvectorfile: This file contains the eigenvectors of the $n$-dimensional mismatch metric $g_{ab}$ described in FlatMesh.h. The file format is simply $n$ lines each containing $n$ whitespace-separated numbers in any standard floating-point format. Each line lists the components of a particular eigenvector; the eigenvector must be normalized so that its squared magnitude is 1 over the corresponding eigenvalue.

inversefile: This file also consists of $n$ lines each with $n$ floating-point numbers. It is simply the matrix inverse of the contents of eigenvectorfile taken as an $n \times n$ matrix.

rangefile: This file consists of two lines of $n$ floating-point numbers; these specify two opposite corners of a rectilinear region in parameter space to be covered by the mesh. Additional lines will be ignored.

Exit codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NORM</td>
<td>0</td>
<td>&quot;Normal exit&quot;</td>
</tr>
<tr>
<td>SUB</td>
<td>1</td>
<td>&quot;Subroutine failed&quot;</td>
</tr>
<tr>
<td>ARG</td>
<td>2</td>
<td>&quot;Error parsing arguments&quot;</td>
</tr>
<tr>
<td>MEM</td>
<td>3</td>
<td>&quot;Memory allocation error&quot;</td>
</tr>
<tr>
<td>DIM</td>
<td>4</td>
<td>&quot;Inconsistent parameter space dimension&quot;</td>
</tr>
<tr>
<td>LEN</td>
<td>5</td>
<td>&quot;Too few points specified&quot;</td>
</tr>
<tr>
<td>FILE</td>
<td>6</td>
<td>&quot;Could not open file&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants FLATMESHTESTC_E<name>, and the status descriptions in FLATMESHTESTC_MSGE<name>. The source code with these messages is in FlatMeshTest.c on line l.93.

Algorithm

For the most part this test program simply reads the input arguments and files, passes them to the function LALCreateFlatMesh() using LALRectIntersect() to define the parameter-space boundary, and prints the resulting mesh. However, there are two additional bits of processing that deserve comment.

The rows of the matrix in eigenvectorfile are already of the form $e_i^j / \sqrt{\lambda_j}$, as discussed in FlatMesh.h. To get the proper orthonormalized transformation matrix, one must simply multiply each
element by $2m_{\text{thresh}}/\sqrt{n}$. Similarly, the inverse transformation matrix elements should be divided by this number.

In order to ensure complete coverage of the desired parameter space, **FlatMeshTest** extends the boundaries of the rectilinear region specified in **rangefile** to include any mesh point whose patch volume touches on the desired search region. If $\mathbf{M}^0$ is the renormalized transformation matrix described above, then the sum of the magnitudes of the components along a column, $\Delta x_j = \sum_i |\mathbf{M}_{ij}|$ represents the maximum extent of a mesh point’s patch in the $j$th dimension. The algorithm in **FlatMeshTest** extends the rectangular search region by half this amount in each direction to ensure that any patch touching on the desired search volume is included. This assumes that the boundary of the search region is “soft”; i.e. that no harm will come of stepping slightly outside it.

**Uses**

- lalDebugLevel
- LALPrintError() LALCheckMemoryLeaks()
- LALCalloc() LALFree()
- LALCreateFlatMesh() LALSReadVectorSequence()
- LALSCreateVectorSequence() LALSDestroyVectorSequence()
- LALSCreateVector() LALSDestroyVector()

**Notes**

Author: Creighton, T. D.

$Id: FlatMeshTest.c,v 1.6 2007/06/08 14:41:52 bema Exp$
27.4.3 Program DirectedMeshTest.c

Computes the sky-position metric for a coherent or semicoherent pulsar search.

Usage

DirectedMeshTest [-o outfile] [-d debuglevel] [-p n dt t0 f0] [-l lat lon]
[-r ra dec] [-r dra ddec] [-t tau] [-m mismatch]

Description

This test program computes template meshes for directed pulsar searches with spindown, where it is assumed that the parameter metric is constant over the search space. The following option flags are accepted:

- o Prints the template mesh to the file outfile: each line consists of a sequence of whitespace-separated numbers representing the coordinates of the template. If absent, the routines are exercised, but no output is written.

- d Sets the debug level to debuglevel; if absent, -d 0 is assumed.

- p Sets the search parameters: the number of stacks n, the length of each stack dt (in seconds), and the start time of the first stack t0 (in seconds of GPS time), and the maximum source frequency f0 (in Hz). If absent, -t 1 86400 0 1000 is assumed.

- l Sets the detector latitude to lat (in degrees north from the equator) and longitude to lon (in degrees east of the prime meridian). If absent, -l 52.247 9.822 (GEO600) is assumed.

- s Sets the right ascension and declination of the target to ra and dec degrees, respectively. If absent, -r 192.8594813 27.1282511 is assumed (the Galactic core).

- r Sets the range of the sky search to ±dra degrees in right ascension and ±ddec degrees in declination about the target point. If absent, -s 0 0 is assumed (no search over sky position).

- t Sets the range of the spindown search according to the spindown timescale tau, in seconds: the spindown parameter f_k is constrained by |f_k| ≤ τ_k. If absent, -t 3.16e9 (century-long spindown) is assumed.

- m Sets the maximum mismatch threshold of the mesh to mismatch. If absent, -m 0.25 is assumed.

The program automatically determines how many spindown terms are required to cover the parameter space, by starting with none (or one if the search is only over spindown), computing the local template density from the parameter metric, and estimating the number of templates required to cover the search volume. The number of spindown parameters is then increased and the number of templates re-estimated. Eventually the estimated number of templates will start to decrease, as the proper width of the parameter space in the new dimensions is less than one template width. The last dimension before that happens is the correct dimension to use.

Exit codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NORM</td>
<td>0</td>
<td>&quot;Normal exit&quot;</td>
</tr>
<tr>
<td>SUB</td>
<td>1</td>
<td>&quot;Subroutine failed&quot;</td>
</tr>
<tr>
<td>ARG</td>
<td>2</td>
<td>&quot;Error parsing arguments&quot;</td>
</tr>
<tr>
<td>VAL</td>
<td>3</td>
<td>&quot;Input argument out of valid range&quot;</td>
</tr>
<tr>
<td>MEM</td>
<td>4</td>
<td>&quot;Memory allocation error&quot;</td>
</tr>
<tr>
<td>DET</td>
<td>5</td>
<td>&quot;Non-positive metric determinant&quot;</td>
</tr>
<tr>
<td>FILE</td>
<td>6</td>
<td>&quot;Could not open file&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants DIRECTEDMESHTESTC_E<name>, and the status descriptions in DIRECTEDMESHTESTC_MSGE<name>. The source code with these messages is in DirectedMeshTest.c on line 1.105.
Algorithm

The program is fairly straightforward. It uses \texttt{LALStackMetric()} to compute the parameter metric, passing it a canonical time function \texttt{LALDTComp()} that composites the \texttt{LALDTSpin()} and \texttt{LALDTBaryPtolemaic()} time transformations. It starts with a single spindown parameter. (If a \textit{search} over sky position is also indicated, it will start with no spindown parameters, using only the barycentric time transformation rather than the composite transformation.) The determinant of the metric is computed, and the number of patches estimated as:

\[
N_{\text{patches}} \approx \frac{\sqrt{|g_{ab}|}}{(\text{mismatch} / n)^{n/2}} \left(\Delta \alpha \Delta \delta \right)^{n-n_s(n_s+1)/2}
\]  

(27.2)

where \(g_{ab}\) is the parameter metric (spindown sector only if the sky search space is a single point), \(n\) is the number of dimensions in the search, \(n_s\) is the number of spindown terms, and \(\Delta \alpha\) and \(\Delta \delta\) are the half-ranges of the search in right ascension and declination, respectively (assuming these ranges are nonzero). For the first round we are considering a search \textit{only} over sky position \((n = 2, n_s = 0)\) or a single spindown search \((n = n_s = 1, \text{ignore the term in braces})\). The determinant is computed using \texttt{LALDMatrixDeterminantErr()}, repacking into \texttt{REAL8Arrays} the metric components and uncertainties returned by \texttt{LALStackMetric()}. An error is generated if the determinant is non-positive, or a warning if it is smaller than its estimated uncertainty.

In subsequent trials, we increase \(n_s\) successively by 1, and recompute \(g_{ab}\) and \(N_{\text{patches}}\). Eventually, when the width of the added dimension is less than one patch width, \(N_{\text{patches}}\) will decrease. When this happen, we back up to the value of \(n_s\) that gave the largest number of patches, and use that parameter metric.

The program then uses \texttt{LALDSymmetricEigenVectors()} to compute the eigenvalues and eigenvectors of the metric; these are combined and repacked into a \texttt{REAL4VectorSequence} used by \texttt{LALFlatMesh()}, as described in \texttt{FlatMesh.h}. The inverse transformation is computed using \texttt{LALDMatrixInverse()}, and again repacked into a \texttt{REAL4VectorSequence}. The search area is taken to be a rectangular space contoled by \texttt{LALRectIntersect()}, covering the sky area \(|\alpha - ra| \leq dra\) and \(|\alpha - ra| \leq dra\) (provided these ranges are nonzero), and the spindown volume \(|f_k| \leq \tau - k\) for \(k = 1,\ldots, n_s\). The volume boundaries are increased by half the maximum patch size in each direction, to ensure total coverage of the edges, as described in \texttt{FlatMeshTest.c}.

Uses

\begin{verbatim}
  lalDebugLevel  LALPrintError()  LALCheckMemoryLeaks()
  LALMalloc()     LALFree()     
  LALU4CreateVector()  LALU4DestroyVector()
  LALSCreateVector()  LALSDestroyVector()
  LALDCreateVector()  LALDDestroyVector()
  LALSCreateVectorSequence()  LALSDestroyVectorSequence()
  LALDCreateArray()  LALDDestroyArray()
  LALDTBaryPtolemaic()  LALTbaryPtolemaic()
  LALDTSpin()  LALTSpin()
  LALDTComp()  LALGetEarthTimes()
  LALCreateFlatMesh()  LALRectIntersect()
  LALStackMetric()  LALProjectMetric()
  LALDMatrixDeterminantErr()  LALMatrixInverse()
  LALDSymmetricEigenVectors()  LALSnprintf()
\end{verbatim}

Notes

Author: Creighton, T. D.

$Id: DirectedMeshTest.c,v 1.2 2007/06/08 14:41:52 bema Exp $
Provides routines to place search meshes for two-dimensional parameter spaces with varying metric.

Synopsis

#include <lal/TwoDMesh.h>

This header covers routines that lay out a mesh of points on an 2-dimensional parameter space \{(x, y)\}, placed such that no point in the space lies further than some maximum proper distance \(m_{\text{thresh}}\) from a mesh point.

The intended purpose of these routines is to place a set of “target” search points over a parameter space, in order to detect signals with unknown parameters. The formalism for defining a proper distance metric on the parameter space is defined in the header FlatMesh.h. However, whereas the routines under FlatMesh.h require the metric \(g_{ab}\) to be constant over the parameter space, the routines under this header only treat \(g_{ab}\) as constant over distances \(\sim m_{\text{thresh}}\).

![Figure 27.1: Mesh placement using parallelogram tiling. (a) The left and right sides of a tile are required to be vertical; the top and bottom sides can tilt to maximize the tile area. (b) Tiles can be stacked in fixed-width columns, even as the elliptical contours change. (c) Extra overlapping tiles are sometimes required at the corners of columns.](image)

Since the metric is treated as constant over distances \(\sim m_{\text{thresh}}\), this distance defines an elliptical contour around any mesh point. We define a “tile” as a parallelogram inscribed within the ellipse, with its left and right sides aligned with the \(y\) axis. This is shown in Fig. 27.1(a), above. A “column” is a set of tiles of constant horizontal width stacked one on top of the other, as shown in Fig. 27.1(b). As the metric changes over space, the vertical height and tilt of the tiles in a column may change, so long as their width remains fixed; we note that if the tilt changes, the tiles will overlap slightly to ensure complete coverage. Finally, the boundary of the parameter space may extend outside the “corners” of the column, crossing the end of a tile between its centre and its edge, as shown in Fig. 27.1(c). These triangular corners can be covered with one or more extra overlapping tiles of reduced width.

In a parameter space with constant metric, the tile area is maximized (and the number of covering tiles minimized) when the column width is \(\sqrt{2}\) times smaller than the projected horizontal width of the ellipses. When the ellipses vary, it is generally best to determine the column width from the narrowest ellipse in a column, to avoid singular effects when tile widths approach the ellipse widths and become infinitesimally high.

For the column-placement algorithm to work effectively, we require that the parameter space be representable as a range \(y \in [y_1(x), y_2(x)]\) between two single-valued functions defined on a domain \(x \in [x_{\text{min}}, x_{\text{max}}]\). If a desired search region is too complicated to express this way (e.g. it has disconnected regions, or “branches” where a vertical line intersects the boundary more than twice), then one should divide the region up into subregions with well-behaved boundary functions and tile these subregions separately.

This header and its associated modules are placed in the pulsar package because they were originally intended for use in searches over sky position, but they can be used generically for any two-dimensional parameter space search where the metric is not too poorly behaved.
Error conditions

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUL</td>
<td>1</td>
<td>&quot;Unexpected null pointer in arguments&quot;</td>
</tr>
<tr>
<td>OUT</td>
<td>2</td>
<td>&quot;Output handle points to a non-null pointer&quot;</td>
</tr>
<tr>
<td>MEM</td>
<td>3</td>
<td>&quot;Memory allocation error&quot;</td>
</tr>
<tr>
<td>METRIC</td>
<td>4</td>
<td>&quot;Non-positive metric&quot;</td>
</tr>
<tr>
<td>WIDTH</td>
<td>5</td>
<td>&quot;Column width too small&quot;</td>
</tr>
<tr>
<td>DIM</td>
<td>6</td>
<td>&quot;Incorrect dimensions&quot;</td>
</tr>
<tr>
<td>INT</td>
<td>7</td>
<td>&quot;Non-positive interval&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants `TWODMESH_E<name>`, and the status descriptions in `TWODMESH_MSGE<name>`. The source code with these messages is in `TwoDMesh.h` on line 1.137.

Types

Structure `TwoDMeshNode`

This structure represents a single node in a linked list of mesh points, specified in the coordinate system used to place it. The fields are:

- REAL4 x, y The coordinates of the mesh point.
- REAL4 dx The half-width of the tile centred on the mesh point.
- REAL4 dy[2] The heights of the two right-hand corners of the tile, relative to the mesh point.
- `TwoDMeshNode *next` The next mesh point in the linked list; NULL if this is the tail.
- `TwoDMeshNode *subMesh` The head of a linked list of fine mesh points within the rectangular area spanned by this mesh point list; NULL if there is no (further) refined mesh for this location.
- UINT4 nSub The number of fine mesh points in the above list. It is an error for `subNum` to be nonzero and `subMesh` to be NULL.

Structure `TwoDMeshParamStruc`

This structure stores the parameters required by the two-dimensional mesh placement functions. The fields are:

- REAL4 domain[2] The domain \([x_{\text{min}}, x_{\text{max}}]\) spanned by the desired parameter region.
- void (*getRange)( LALStatus *, REAL4[2], REAL4, void ) A function that returns in its second argument the range \([y_1(x), y_2(x)]\) spanned by the parameter region for a specified \(x\), which is passed in as the third argument. The fourth argument can be used to pass function-specific parameters.
- void *rangeParams The parameters to be passed as the fourth argument of `*getRange()`, above.
- void (*getMetric)( LALStatus *, REAL4[3], REAL4[2], void *) A function that returns in its second argument the components \(g_{xx}, g_{yy}\), and \(g_{xy}\) (in that order) of the metric evaluated at a point \((x, y)\), which is passed in as the third argument. The fourth argument can be used to pass function-specific parameters.
- void *metricParams The parameters to be passed as the fourth argument of `*getMetric()`, above.
- REAL4 mThresh The maximum mismatch \(m_{\text{thresh}}\) desired between any point in the region and the nearest mesh point; note that the maximum mismatch is equal to 1 minus the minimum match.
- REAL4 widthMaxFac The minimum ratio of mismatch ellipse width (projected onto the horizontal axis) to column width that must be maintained throughout the column: if an ellipse falls below this ratio due to shrinkage or rotation, as in Fig 29.1.b, the code will try a narrower column. If set to \(\leq 1\), the default value `TWODMESHINTERNALC_WMAXFAC = \sqrt{2}` will be used.
REAL4 widthRetryFac If the column is determined to be too wide (e.g. due to the value of widthMaxFac, above), the column width will be reduced by the factor widthRetryFac. If set to \( \leq 1 \), the default value \( \text{TWODMESHINTERNALC\_WRETRYFAC} = \sqrt{2} \) will be used.

UINT4 maxColumns The maximum number of columns the mesh placement routine will try before giving up. If zero, this number is ignored.

UINT4 nIn The maximum number of mesh points allowed, after which the placement routine will quit. If zero, this number is ignored.

UINT4 nOut The number of mesh points added by the placement routine. If an error occurs, this will store the number of mesh points completed before the error.

Structure TwoDColumnParamStruc

This structure stores additional parameters required when laying down a single column of a two-dimensional mesh. The area to be covered is specified by intersecting the area between two lines with the parameter space. If part of a column has already been covered, one can further restrict the area by specifying a pair of “clipping points” on each vertical line; the area to be covered is then restricted to lie above the line joining the bottom two corners and below the line joining the top two corners. The fields of the structure are:

REAL4 domain[2] The region in \( x \) spanned by the column. We require that \( \text{domain}[1] > \text{domain}[0] \).

REAL4 leftRange[2] The values \( y_1(x), y_2(x) \) (in that order) of the boundary functions at \( x = \text{domain}[0] \).

REAL4 rightRange[2] The values of \( y_1(x), y_2(x) \) (in that order) of the boundary functions at \( x = \text{domain}[1] \).

REAL4 leftClip[2] The \( y \) values of the bottom and top corners (in that order) of the clipping boundary at \( x = \text{domain}[0] \).

REAL4 rightClip[2] The \( y \) values of the bottom and top corners (in that order) of the clipping boundary at \( x = \text{domain}[1] \).

BOOLEAN tooWide This is set to 1 if the column-placement routine determines that the region is too wide to be covered with a single column of tiles.
27.5.1 Module TwoDMesh.c

Creates or destroys a hierarchical mesh of templates on an 2-dimensional parameter space.

Prototypes

```c
void LALCreateTwoDMesh( LALStatus *stat,
                        TwoDMeshNode **mesh,
                        TwoDMeshParamStruct *params )

void LALDestroyTwoDMesh( LALStatus *stat,
                          TwoDMeshNode **mesh,
                          UINT4 *nFree )

void LALRefineTwoDMesh( LALStatus *stat,
                        TwoDMeshNode *coarseMesh,
                        TwoDMeshNode *fineMesh )
```

Description

The routine `LALCreateTwoDMesh()` lays out an unevenly-spaced mesh on a 2-dimensional parameter space, according to the method presented in `TwoDMesh.h` and detailed in `TwoDMeshInternal.c`. The parameter `mesh` is a handle to the head of the newly-created linked list of mesh points, while `params` points to the parameter structure used to create the list. On completion, `params->nOut` is set to the number of mesh points created.

The routine `LALDestroyTwoDMesh()` destroys the list pointed to by `*mesh`, including all sub-meshes, and sets `*mesh=NULL`. If `*mesh` is already `NULL`, nothing is done (this is not an erroneous usage). If `nFree`≠`NULL`, then `*nFree` is set to the number of nodes freed.

The routine `LALRefineTwoDMesh()` creates a hierarchical search mesh by inserting copies of the nodes in the list pointed to by `fineMesh` into the `subMesh` fields of appropriate nodes in the list pointed to by `coarseMesh`. The contents of the `fineMesh` list are untouched. If a `fineMesh` tile does not overlap with any `coarseMesh` tile, a warning is generated, but this is not treated as an error. If an internal error does occur, the refinement will be left in a state of partial completion; there is just too much overhead involved in maintaining an uncorrupted copy of the `coarseMesh` list for it to be worthwhile.

Algorithm

`LALCreateTwoDMesh` simply creates a dummy node to serve as the head of the linked list, and calls `LALTwoDMesh()` in `TwoDMeshInternal.c` to attach a mesh to it. The details of the algorithm are given in `TwoDMeshInternal.c`.

`LALDestroyTwoDMesh()` navigates down the linked list of mesh points, destroying them as it goes. It calls itself recursively on any non-empty sub-meshes to destroy them too.

`LALRefineTwoDMesh()` moves along the `fineMesh` list; for each node in the list, it searches the `coarseMesh` list for the any tile that overlaps with the fine mesh tile. It then copies the fine mesh node (and its submesh, if any) into the coarse mesh node’s `subMesh` list, using `LALTwoDNodeCopy()` in `TwoDMeshInternal.c`. Although it uses more memory, this recursive copy routine is preferred over simple relinking, so as to avoid any possible memory leaks: destroying the coarse mesh list will leave the fine mesh list intact, and vice-versa.

To create a > 2 level hierarchical search mesh, build it from the bottom up: call `LALRefineTwoDMesh()` to add the finest mesh to the next finest, add that to the next finest, and so on up to the coarsest mesh.

Uses

- `lalDebugLevel`
- `LALPrintError()`
- `LALWarning()`
- `LALInfo()`
- `LALTwoDMesh()`
- `LALTwoDNodeCopy()`
- `LALFree()`
Notes
27.5.2 Module TwoDMeshInternal.c

Low-level routines to place a mesh of templates on an 2-dimensional parameter space.

Prototypes

```c
void LALTwoDMesh( LALStatus *stat,
                    TwoDMeshNode **tail,
                    TwoDMeshParamStruc *params )
```

```c
void LALTwoDColumn( LALStatus *stat,
                     TwoDMeshNode **tail,
                     TwoDColumnParamStruc *column,
                     TwoDMeshParamStruc *params )
```

```c
void LALTwoDNodeCopy( LALStatus *stat,
                       TwoDMeshNode **new,
                       TwoDMeshNode *old )
```

Description

These are low-level “internal” routines called by LALCreateTwoDMesh() to lay out an unevenly-spaced mesh on a 2-dimensional parameter space, according to the method discussed in TwoDMesh.h. They are made globally available to allow greater control to users attempting to tile complicated parameter spaces.

LALTwoDMesh() places a mesh on the parameter space specified by *params. On successful completion, the linked list of mesh points is attached to (*tail)->next (which must previously have been NULL), updates *tail to point to the new tail of the list, and increases params->nOut by the number of mesh points added. (This is useful for tiling several parameter regions independently with successive calls to LALTwoDMesh.) On an error, **tail is left unchanged, and params->nOut indicates where the error occurred.

LALTwoDColumn() places a single column of such a mesh, according to the additional column restrictions in *column. Again, on success, the mesh points are attached to (*tail)->next, *tail is updated, and params->nOut increased. If the column specified by *column is deemed to be too wide for a single column of templates, then column->tooWide is set to 1, *tail and params->nOut are not updated, and the function returns normally (i.e. not with an error code set). Other more fatal errors are treated as for LALTwoDMesh(), above.

LALTwoDNodeCopy() creates a copy of the node *old and points *new to the copy. If old->subMesh exists, each node in the submesh is copied into its corresponding place by recursive calls to LALTwoDNodeCopy(). On an error, the copy is destroyed and *new left unchanged.

Algorithm

LALTwoDMesh(): This routine starts placing mesh points at the left side of the parameter region, \( x = x_{\text{min}} \). It calls params->getRange() to get the bottom and top of the left edge of the first column. It also calls params->getMetric at these two corners, estimates the optimum width for the first column, and uses params->getRange() again to get the two corners of the right edge of the column. It then calls the subroutine LALTwoDColumn() (below) to place the mesh points in this column.

If LALTwoDColumn() reports that the estimated column width was too large, LALCreateTwoDMesh() tries again with the width reduced by the factor params->widthRetryFac. This continues until the estimated number of columns exceeds params->maxColumns; i.e. until the current column width is less than \((x_{\text{max}} - x_{\text{min}})/\text{params->maxColumns}\). If this occurs, the linked list is destroyed using LALDestroyTwoDMesh(), and an error is returned.

Otherwise, if LALTwoDColumn() returns success (and does not complain about the column width), LALCreateTwoDMesh() gets the width and heights of the next column, and calls LALTwoDColumn() again. This continues until eventually the right edge of a column lies beyond \( x_{\text{max}} \). This last column is squeezed so that its right edge lies exactly at \( x_{\text{max}} \); once it is filled, the mesh is deemed complete, and no further columns are generated.
LALTtwoDColumn(): This routine first locates the centreline of the column, and uses params->getRange() to see how much of this centreline is taken up by the requested parameter region, restricted by any clipping area specified in *column. If any region of this centreline remains uncovered, LALTtwoDColumn() places a tile at the bottom of the uncovered portion, and stacks more tiles upward until it reaches the top of the uncovered line. The sizes and orientations of each tile are determined from calls to params->getMetric.

While it is doing this, LALTtwoDColumn() keeps track of the bottom and top corners of the newly-covered region. If it finds that the top corners are not increasing monotonically, this is usually an indication that the metric is changing too rapidly, or that the tiles are getting too thin and tilted. Often this can be corrected by using narrower (and taller) tiles, so LALTtwoDColumn() reports this as a “column too wide” result: it sets column->tooWide, frees everythin attached to **tail and reduced params->nOut accordingly, then returns. This is also done if LALTtwoDColumn() ever determines that the maximum width of a mismatch ellipse is less than params->widthMaxFac times the current column width.

Having successfully stacked up the centreline of the current column, LALTtwoDColumn() then checks to see whether corners of the parameter region extend above or below the top and bottom of the newly-tiled region on either side of the centreline, by looking at the values in column->leftRange and column->rightRange. If a corner remains uncovered, LALTtwoDColumn() calls itself recursively on a half-width column on the appropriate side, setting the clipping area of the subroutine call to exclude the region already covered. In principle it could call itself up to four times (once for each column), and each recursive call could in turn call itself recursively in order to cover a particularly steep or complicated boundary. However, in most cases at most two additional tiles need to be placed (one on a top corner, one on a bottom corner). If you’re concerned about a runaway process, set params->nIn to stop generation after a given number of tiles. If a recursive call reports the column is too wide, this information is passed up the calling chain.

Once the centreline and any corners have been successfully covered, LALTtwoDColumn() updates *tail to the new tail of the list, and returns.

LALTtwoDNodeCopy(): This routine works by a simple recursive algorithm. First, a copy of the node is allocated and the output handle is pointed to it. Next, all non-pointer fields are copied over. Then, if new->subMesh exists, LALTtwoDNodeCopy() navigates its way along the list, calling itself recursively on each node, and attaching copies of each node to a corresponding list in (*new)->subMesh. If any errors are detected, *new is destroyed via LALDestroyTwoDMesh(), restoring it to NULL.

Computing tile sizes: Given a positive-definite 2-dimensional metric $g_{ab}$, the elliptical contour corresponding to a maximum mismatch $m_{\text{thresh}}$ is given by the set of points offset from the centre point by amounts $(\Delta x, \Delta y)$, where:

$m_{\text{thresh}} = g_{xx}(\Delta x)^2 + g_{yy}(\Delta y)^2 + 2g_{xy}\Delta x\Delta y.$

Thus for a tile of some half-width $\Delta x$, the heights of the two right-hand corners of the tile relative to its centre are:

$\Delta y = \frac{-g_{xy}\Delta x \pm \sqrt{g_{yy}m_{\text{thresh}} - (g_{xx}g_{yy} - g_{xy}^2)(\Delta x)^2}}{g_{yy}},$

and the maximum half-width of a tile is:

$\Delta x_{\text{max}} = \sqrt{\frac{g_{yy}m_{\text{thresh}}}{g_{xx}g_{yy} - g_{xy}^2}}.$

The positive-definiteness of the metric ensures that $g_{xx} > 0, g_{yy} > 0$, and $g_{xx}g_{yy} > g_{xy}^2$. Furthermore, if one maximizes the proper area of a tile with respect to $\Delta x$, one finds that the optimal tile half-width is:

$\Delta x_{\text{opt}} = \frac{\Delta x_{\text{max}}}{\sqrt{2}}.$

When estimating the width for the next column, LALTtwoDMesh() computes $\Delta x_{\text{opt}}$ at both the bottom and the top of the column and uses the smaller value (it is almost always better to underestimate $\Delta x$ than to overestimate it). In LALTtwoDColumn(), tile heights are computed using the column half-width $\Delta x$ and the value of the metric components at its particular location.

We also note that the width of a column is computed using the metric evaluated at the edge joining it to the previous column, and the height of a tile is computed using the metric evaluated at the edge joining it to the previous tile. In principle it might be more accurate to refine our estimate of the column width or
tile height by re-evaluating the metric at their centres, but this may be a significant excess computational burden. Furthermore, if the metric varies enough that the estimated width or height changes significantly over that distance, then the quadratic approximation to the match function is breaking down, and we shouldn’t be treating the constant-mismatch contour as an ellipse. The routines here do not do any sophisticated sanity-checking, though.

Uses

lalDebugLevel
LALInfo() LALPrintError()
LALMalloc() LALDestroyTwoDMesh()

Notes
27.5.3 Module TwoDMeshRanges.c

Some range computation routines suitable for use in LALCreateTwoDMesh().

Prototypes

```c
void LALInterpolateRangePolygon( LALStatus *stat, REAL4 range[2], REAL4 x, void *params )
void LALInterpolateRangeGrid( LALStatus *stat, REAL4 range[2], REAL4 x, void *params )
```

Description

This module contains range computation routines suitable for passing into LALCreateTwoDMesh() via the params->getRange function parameter.

The routine LALInterpolateRangePolygon() takes as its parameter a (void *) pointer to a REAL4VectorSequence structure, containing a list of 2-dimensional vectors (x,y) giving the locations of points on the boundary of a polygonal region. The function returns in range the points where a vertical line at the specified x-value crosses the edge of the polygon, in ascending order. If no intersections are found, then both range values are set equal to (one of) the nearest point(s) on the boundary.

The routine LALInterpolateRangeGrid() takes as its parameter a (void *) pointer to a REAL4Grid structure with physical dimension 1 and data dimension 2: for each point x along the physical dimension, the grid stores a vector of length 2, giving the lower and upper range values y_1(x) and y_2(x). The routine linearly interpolates these two sampled functions to compute the range interval for any specified x. If the specified x is outside the grid, then both range values are set equal to the average of the range points at the nearest endpoint of the grid.

Algorithm

The LALInterpolateRangePolygon() function is just a stub at present; it returns [0,1] as its range regardless of inputs.

Uses

lalDebugLevel

Notes

Author: Creighton, T. D.

$Id: TwoDMeshRanges.c,v 1.2 2007/06/08 14:41:51 bema Exp $
27.6 Header TwoDMeshPlot.h

Provides routines to plot two-dimensional parameter meshes.

Synopsis

```c
#include "TwoDMeshPlot.h"
```

This header covers routines that display 2-dimensional parameter meshes as generated by the routines in TwoDMesh.h. The output is written to a C FILE-type stream using `fprintf()` The routines do not provide a system-level interface to open or close file streams; they simply assume that they have been passed an open, writable stream. Nonetheless, their use of stream I/O means that they cannot be included in LAL (although executables, in particular the TwoDMesh.h test programs, may statically link against their object files).

Error conditions

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUL</td>
<td>1</td>
<td>&quot;Unexpected null pointer in arguments&quot;</td>
</tr>
<tr>
<td>OUT</td>
<td>2</td>
<td>&quot;Output handle points to a non-null pointer&quot;</td>
</tr>
<tr>
<td>MEM</td>
<td>3</td>
<td>&quot;Memory allocation error&quot;</td>
</tr>
<tr>
<td>METRIC</td>
<td>4</td>
<td>&quot;Non-positive metric&quot;</td>
</tr>
<tr>
<td>NOPLOT</td>
<td>5</td>
<td>&quot;Nothing to plot&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants `TWODMESHPLOTH_<name>`, and the status descriptions in `TWODMESHPLOTH_MSGE_<name>`. The source code with these messages is in `TwoDMeshPlot.h` on line 1.75.

Constants

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XSIZE</td>
<td>540</td>
<td>Horizontal size of plotting area (points = 1/72″)</td>
</tr>
<tr>
<td>YSIZE</td>
<td>720</td>
<td>Vertical size of plotting area (points)</td>
</tr>
<tr>
<td>XMARG</td>
<td>36</td>
<td>Distance from left of page to plotting area (points)</td>
</tr>
<tr>
<td>YMARG</td>
<td>36</td>
<td>Distance from bottom of page to plotting area (points)</td>
</tr>
</tbody>
</table>

The values in the table above are stored in the constants `TWODMESHPLOTH_<name>`.

Types

Structure TwoDMeshPlotStruc

This structure stores parameters specifying how to plot a PostScript diagram of the parameter mesh. The fields are:

- **REAL4 theta** Angle from the horizontal direction of the plot counterclockwise to the x-coordinate axis of the mesh, in degrees.
- **REAL4 xScale, yScale** Plotting scale of the mesh coordinate axes, in points per unit x or y (a point is 1/72 of an inch).
- **REAL4 bBox[4]** Bounding box surrounding the figure in plot coordinates, measured in points.
- **BOOLEAN autoscale** If true, xScale and yScale will be adjusted so that the drawn figure will lie within the bBox. If false, bBox will be adjusted to enclose the figure, given xScale and yScale.
- **REAL4 clipBox[4]** Four components x_{min}, y_{min}, x_{max}, y_{max} (in that order) specifying the corners of a box in the x-y coordinate system outside of which no marks will be made. If either max value is less than or equal to the corresponding min value, clipBox will be ignored.
- **UINT4 nLevels** The number of levels of recursive submeshes to plot. If zero, the mesh will not be plotted (although the boundary may be).
UINT4 nBoundary  *half* the number of points to plot along the boundary of the parameter region. At least 4 points are required; if plotBoundary < 2, none will be plotted.

INT2 *plotPoints* An array from [0] to [nLevels] indicating how to plot the mesh points at each recursive level: a value of 0 means don’t plot mesh points, a positive value means to plot filled circles of that diameter (in points), a negative value means to plot empty circles of that diameter (in points).

BOOLEAN *plotTiles* An array from [0] to [nLevels] indicating whether to plot the tiles around each mesh point, at each recursive level.

BOOLEAN *plotEllipses* An array from [0] to [nLevels] indicating whether to plot the mismatch ellipses around each mesh point, at each recursive level.

TwoDMeshParamStruc *params* An array from [0] to [nLevels] of parameter structures used to generate the meshes at each recursive level (making the assumption that all submeshes of the same level used the same parameters). In general only the getMetric() and mThresh fields are used from parameter structures after the first.
27.6.1 Module TwoDMeshPlot.c

Plots a hierarchical mesh of templates on an 2-dimensional parameter space.

Prototypes

```c
void LALPlotTwoDMesh( LALStatus *stat,
                       FILE *stream,
                       TwoDMeshNode *mesh,
                       TwoDMeshPlotStruc *params );
```

Description

This routine creates a PostScript plot of the parameter mesh list pointed to by `mesh`, using the plotting parameters given in `*params`. The PostScript output is printed to the writable output stream `*stream` using `fprintf()`.

Algorithm

The algorithm is set up so that it requires only one pass through the list. After defining PostScript macros to plot mesh points, mesh tiles, and mismatch ellipses, the routine then defines a macro to plot the boundary. Since some PostScript interpreters will fail if a macro contains too many objects, the boundary-plotting macro may be split into several macros.

`LALPlotTwoDMesh()` then calls a static (but LAL-compliant) subroutine `LALMakeMeshMacro()` to create one or more macros to plot the mesh points, tiles, or ellipses, as required by `*params`. This subroutine takes a pointer to the head of a list of mesh points as input, and traverses down the list, calling itself recursively on any submeshes it encounters (if `params->maxLevels` permits).

While plotting the boundary and other mesh objects, `LALPlotTwoDMesh()` and `LALMakeMeshMacro()` keep track of the bounding box surrounding all plotted objects. This is used either to set a bounding box for the overall plot, or to adjust the scale of the plot, depending on `params->autoscale`. If the resulting bounding box is larger than a single 8.5″ × 11″ page, `LALPlotTwoDMesh()` will divide the plot area up into pages of this side, calling the plotting macros on each page.

Uses

`LALMalloc()` `LALFree()`

Notes
27.6.2 Program TwoDMeshTest.c

Creates a 2-dimensional template mesh for linearly-changing mismatch ellipses.

Usage

TwoDMeshTest [-o outfile] [-p psfile flags] [-d debug] [-m mismatch nmax cmax]
            [-i metricfile rangefile] [-b x1 y1 x2 y2] [-e a b c]
            [-x dadx dbdx dcdx] [-y dady dbdy dcdy]

Description

This test program creates a template mesh for a parameter space with an arbitrary mismatch metric. The following option flags are accepted:

- **-o** Writes the output mesh list to the file `outfile`. If absent, no output is written.

- **-p** Plots the output mesh in a PostScript file `psfile`, using plot flags `flags` (see below). If absent, no plot is made.

- **-d** Sets the debug level to `debug`. If absent, a debug level of zero is used.

- **-m** Sets the maximum mismatch to `mismatch`, maximum number of mesh points to `nmax` and the maximum estimated number of columns to `cmax`. If `mismatch` is not in the range (0,1], it is taken to be 1. If `nmax` or `cmax` is non-positive, it is ignored (no maximum). If this option is not given, `-m 1 0 0` is assumed.

- **-i** Determines the metric and the parameter space boundary from REAL4Grid structures stored in the files `metricfile` and `rangefile`, read using the generic parser `LALSReadGrid()`. The formats for these grids are discussed below. If present, this option overrides the `-b`, `-e`, `-x`, and `-y` options, below. If absent, these options, or their defaults, will be used.

- **-b** Sets the parameter space boundary to be a parallelogram defined by the vectors \((x1,y1)\) and \((x2,y2)\) from the origin. If absent, the region is taken to be a unit square.

- **-e** Sets the parameters of the mismatch ellipse at the origin: its principal axis lengths are \(a\) and \(b\) units, and the angle from the \(x\)-axis to the first principal axis is \(c\) radians. If absent, `-e 0.1 0.05 1` is assumed.

- **-x** Sets the rates of change in the \(x\)-direction of \(a\), \(b\), and \(c\) (above) to \(dadx\), \(dbdx\), and \(dcdx\), respectively. If absent, the rates are taken to be zero.

- **-y** Sets the rates of change in the \(y\)-direction of \(a\), \(b\), and \(c\) (above) to \(dady\), \(dbdy\), and \(dcdy\), respectively. If absent, the rates are taken to be zero.

Exit codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NORM</td>
<td>0</td>
<td>&quot;Normal exit&quot;</td>
</tr>
<tr>
<td>SUB</td>
<td>1</td>
<td>&quot;Subroutine failed&quot;</td>
</tr>
<tr>
<td>ARG</td>
<td>2</td>
<td>&quot;Error parsing arguments&quot;</td>
</tr>
<tr>
<td>BAD</td>
<td>3</td>
<td>&quot;Bad argument value&quot;</td>
</tr>
<tr>
<td>MEM</td>
<td>4</td>
<td>&quot;Memory allocation error&quot;</td>
</tr>
<tr>
<td>FILE</td>
<td>5</td>
<td>&quot;Could not open file&quot;</td>
</tr>
<tr>
<td>METRIC</td>
<td>6</td>
<td>&quot;Axis length is zero or negative within specified region&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants `TWODMESHTESTC_E<name>`, and the status descriptions in `TWODMESHTESTC_MSGE<name>`. The source code with these messages is in `TwoDMeshTest.c` on line 1.102.
Algorithm

The test program reads the input arguments and creates a parameter structure *params to be passed to LALCreateTwoDMesh(). In particular, it computes the domain of the parameter space, and defines functions and parameter lists to compute the range in y at any x, and the metric at any point (x,y). If PostScript output is requested, it is generated using LALPlotTwoDMesh(), using the value of the command-line number flags to set the plotting parameters. Each of these functions is discussed below.

Metric and range grid files: If the -i option was given, the metric and parameter ranges are read from the files named by the metricfile and rangefile arguments. These two files must be in a format parseable by LALReadGrid(); see the documentation of that routine for more details.

The REAL4Grid extracted from metricfile must have grid dimension 2 and data dimension 3: the grid dimensions refer to the (x,y) coordinates of the points where the metric is evaluated, while the third dimension must have length 3, storing the metric components $g_{xx}$, $g_{yy}$, and $g_{xy}$ (in that order) at each point. Within the TwoDMesh.h routines, this metric grid is interpolated using LALInterpolateMetricGrid().

The REAL4Grid extracted from rangefile must have grid dimension 1 and data dimension 2: the grid dimension refers to an x coordinate, and the second dimension must have length 3, storing the lower and upper boundaries $y_1(x)$ and $y_2(x)$ at each sampled value of $x$. Within the TwoDMesh.h routines, this range grid is interpolated using LALInterpolateRangeGrid().

If the -i option is not given, then the parameter boundary and metric are determined by internal routines, with default settings that can be overridden using command-line options -p, -e, -x, and -y.

Parameter ranges: The parameter space boundary can be specified by input parameters $x_1=x_1$, $x_2=x_2$, $y_1=y_1$, and $y_2=y_2$. The parameter space is then defined to be a parallelogram with one corner on the origin, and two sides defined by vectors $(x_1,y_1)$ and $(x_2,y_2)$. Without loss of generality we assume that $x_1 < x_2$. The functions defining the boundaries are denoted $y_{a,b}(x)$, and we make no assumption about their signs or relative order. The algorithm used then depends on the signs of $x_1$ and $x_2$.

If $x_1 = x_2 = 0$, then the parameter space is singular, and no mesh need be generated.

If $x_1 = 0$ and $x_2 \neq 0$, then the domain is $[0,x_2]$, and the boundary functions are:

\[
\begin{align*}
y_a(x) &= y_2x/x_2 \\
y_b(x) &= y_1 + y_2x/x_2
\end{align*}
\]

If $x_2 = 0$ and $x_1 \neq 0$, then the domain is $[x_1,0]$, and the above equations for $y_{a,b}(x)$ simply have 1 and 2 reversed.

If $x_1$ and $x_2$ have the same sign, then the domain is $[0,x_1 + x_2]$ if $x_1$ and $x_2$ are positive, and $[x_1 + x_2,0]$ otherwise. The boundary functions are:

\[
\begin{align*}
y_a(x) &= \begin{cases} y_1x/x_1 & x \text{ between 0 and } x_1 \\ y_1 + y_2(x-x_1)/x_2 & x \text{ between } x_1 \text{ and } x_1 + x_2 \end{cases} \\
y_b(x) &= \begin{cases} y_2x/x_2 & x \text{ between 0 and } x_2 \\ y_2 + y_1(x-x_2)/x_1 & x \text{ between } x_2 \text{ and } x_1 + x_2 \end{cases}
\end{align*}
\]

If $x_1$ and $x_2$ have opposite sign, the domain is $[x_1,x_2]$ if $x_1 < 0$, and $[x_2,x_1]$ otherwise. The boundary functions are:

\[
\begin{align*}
y_a(x) &= \begin{cases} y_1x/x_1 & x \text{ between 0 and } x_1 \\ y_2x/x_2 & x \text{ between 0 and } x_2 \end{cases} \\
y_b(x) &= \begin{cases} y_1 + y_2(x-x_1)/x_2 & x \text{ between } x_1 \text{ and } x_1 + x_2 \\ y_2 + y_1(x-x_2)/x_1 & x \text{ between } x_2 \text{ and } x_1 + x_2 \end{cases}
\end{align*}
\]

The main program sorts the input parameters so that $x_1 \leq x_2$, stores them in a 4-dimensional array, and assigns a void pointer to that array. It also computes the domain. The routine LALTwoDRangeTest() takes a value of $x$ and the void pointer, computes the values of $y_a(x)$ and $y_b(x)$ according to the algorithm above, sorts them, and returns them ordered from lower to higher.
Metric values: The main program takes the input parameters $a$, $b$, $c$, $d_{adx}$, $d_{dbdx}$, and $d_{cdx}$, stores them in a 9-dimensional array, and assigns a `void` pointer to it. The routine `LALTwoDMetricTest()` takes a position $(x, y)$ and the `void` pointer, and computes the “local” value of the principal axis $a = a + x \times d_{adx} + y \times d_{ady}$, and similarly for $b$ and $c$. If that ellipse corresponds to the $m_{thresh}$ mismatch level contour, then the eigenvalues of the corresponding metric are $\lambda_1 = m_{thresh}/a^2$ and $\lambda_2 = m_{thresh}/b^2$. The metric components are thus:

$$
\begin{align*}
g_{xx} &= \lambda_1 \cos^2(c) + \lambda_2 \sin^2(c), \\
g_{yy} &= \lambda_1 \sin^2(c) + \lambda_2 \cos^2(c), \\
g_{xy} = g_{yx} &= (\lambda_1 - \lambda_2) \cos(c) \sin(c).
\end{align*}
$$

The routine assumes that the values of $a$, $b$, and $c$ refer to an $m_{thresh} = 1$ mismatch ellipse. It computes and returns $g_{xx}$, $g_{yy}$, and $g_{xy}$ in a 3-dimensional array.

PostScript flags: The parameter `flags` is an unsigned integer whose lowest-order bits contain parameters to be passed to `LALPlotTwoDMesh()`. The bits and their meanings are:

bit 0: 1 if mesh points will be plotted, 0 otherwise.
bit 1: 1 if mesh tiles will be plotted, 0 otherwise.
bit 2: 1 if mismatch ellipses will be plotted, 0 otherwise.
bit 3: 1 if the boundary will be plotted, 0 otherwise.

Thus a value of 15 will plot everything, while a value of 9 will just plot the mesh points and the boundary. A value of zero suppresses the plot.

If mesh points are to be plotted, they will be filled circles $1/72''$ (1 point) in diameter. The parameter space will be rotated so that the longer of the diagonals of the parallelogram will be vertical, and scaled to fit on one $8.5'' \times 11''$ page. That is, if $\|(x_1 + x_2, y_1 + y_2)\| \geq \|(x_1 - x_2, y_1 - y_2)\|$, the rotation angle of the coordinate axes will be $\theta = \pi/2 - \arctan2(y_1 + y_2, x_1 + x_2)$, or $\theta = \pi/2 - \arctan2(y_2 - y_1, x_2 - x_1)$ otherwise.

We note that the function $\arctan2(y, x)$ returns the argument of the complex number $x + iy$ in the range $[-\pi, \pi]$.

Uses

```
lalDebugLevel
LALPrintError() LALCheckMemoryLeaks()
LALCreateTwoDMesh() LALDestroyTwoDMesh()
LALSReadGrid() LALSDestroyGrid()
LALPlotTwoDMesh()
```

Notes

Author: Creighton, T. D.

$Id: TwoDMeshTest.c,v 1.8 2007/06/08 14:41:52 bema Exp$
27.7  Header Resample.h

Provides routines for resampling time series according to a new canonical time coordinate.

Synopsis

#include <lal/Resample.h>

One of the crucial problems in searching for constant-frequency astrophysical signals is removing the effects of Doppler modulation due to the Earth’s motion. This is normally accomplished by constructing a canonical time coordinate $\tau$ of an inertial frame (i.e. the barycentred time), and decimating/resampling the data at fixed intervals in $\tau$. The reconstructed $\tau$ depends on the direction to the source relative to the Earth’s motion; in addition, slow intrinsic parameterized modulations in the source frequency can also be corrected by this coordinate transformation.

Most of the routines in this module assume that $\tau$ can be piecewise expanded as a Taylor series in $t$. That is, one defines a set of fitting regions $T_i = [t_{\text{bound}(i-1)}, t_{\text{bound}(i)}]$, and a set of fitting points $t_{(i)} \in T_i$. In each region one then writes:

$$\tau(t) = \sum_{k=0}^{1} \frac{1}{k!} c_{k(i)} (t - t_{(i)})^k.$$

(27.3)

Since one is normally interested in tracking the difference $\tau(t) - t$, one can also write the expansion as:

$$\tau(t) - t = \sum_{k=0}^{a_{k(i)}} a_{k(i)} (t - t_{(i)})^k,$$

(27.4)

where

$$a_{0(i)} = c_{0(i)} - t_{(i)},$$

$$a_{1(i)} = c_{1(i)} - 1,$$

$$a_{k(i)} = \frac{c_{k(i)}}{k!}, k \geq 2.$$

These are the polynomial coefficients normally assumed in the modules under this header.

The procedure for resampling according to $\tau$ is normally combined with decimating the time series. That is, one takes a time series sampled at constant intervals $\Delta t$ in $t$, and samples it at constant intervals $d \Delta t$ in $\tau$, where the decimation factor $d$ is normally taken to be an integer $\geq 1$. When $\tau$ and $t$ are drifting out of phase relatively slowly, this means that most of the time every $d^{\text{th}}$ sample in the original time series becomes the next sample in the decimated time series. However, when $\tau$ and $t$ drift out of synch by an amount $\pm \Delta t$, one can force the decimated time series to track $\tau$ (rather than $t$) by sampling the $d^{\pm 1\text{th}}$ next datum (rather than the $d^{\text{th}}$). If the drift is sufficiently rapid or $d$ is sufficiently large, one may be forced to choose the point $d \pm 2$, $d \pm 3$, etc.; the size of this adjustment is called the correction shift. The number of (resampled) time intervals between one correction point and the next is called the correction interval.

Unless otherwise specified, all time variables and parameters in the functions under this header can be assumed to measure the detector time coordinate $t$. Canonical times are specified by giving the difference $\tau - t$.

Caveat emptor: The inclusion of this header and its associated modules into LAL is provisional at this time. The routines and the test code appear to work, but a later standalone code, operating on much larger datasets, appeared to encounter a memory leak. I have not yet determined whether this leak was in the standalone code or in these LAL routines.

Error conditions

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUL</td>
<td>1</td>
<td>&quot;Unexpected null pointer in arguments&quot;</td>
</tr>
<tr>
<td>OUT</td>
<td>2</td>
<td>&quot;Output handle points to a non-null pointer&quot;</td>
</tr>
<tr>
<td>MEM</td>
<td>3</td>
<td>&quot;Memory allocation error&quot;</td>
</tr>
<tr>
<td>DTPOS</td>
<td>4</td>
<td>&quot;Sampling interval is not positive&quot;</td>
</tr>
<tr>
<td>LENGTH</td>
<td>5</td>
<td>&quot;Vector lengths in polyco structure don’t agree&quot;</td>
</tr>
<tr>
<td>TIME</td>
<td>6</td>
<td>&quot;Requested output time span extends beyond range of validity of input&quot;</td>
</tr>
</tbody>
</table>
The status codes in the table above are stored in the constants `RESAMPLEH_E<name>`, and the status descriptions in `RESAMPLEH_MSGE<name>`. The source code with these messages is in `Resample.h` on line 1.133.

### Types

**Structure ResampleRules**

This structure stores the rules for taking a time series \( t \), sampled at constant intervals \( \Delta t \), and resampling it at constant intervals \( d\Delta t \) in the canonical time coordinate \( \tau \), as described above. The fields in this structure are as follows:

- `LIGOTimeGPS start`: The initial time for which the rules apply.
- `LIGOTimeGPS stop`: The final time for which the rules apply.
- `INT4 length`: The number of correction points, i.e. points where the resampling interval is adjusted from \( d\Delta t \) to \((d \pm n)\Delta t\).
- `INT4 *interval`: An array giving the number of resampled time intervals between correction points.
- `INT2 *shift`: An array giving the size of the correction shift (i.e. the number \( n \) above) at each correction point.
- `INT4 decimate`: The decimation factor \( d \).
- `REAL8 deltaT`: The sampling interval before decimation, in seconds.
- `REAL8 startDiff`: The difference \( \tau - t \) at the time \( \text{start} \), in seconds.
- `REAL8 stopDiff`: The difference \( \tau - t \) at the time \( \text{stop} \), in seconds.

**Structure PolycoStruc**

This structure stores the parameters of the piecewise polynomial fit of \( \tau - t \) as a function of \( t \). See Eq. (27.4) for notation. The fields of this structure are:

- `REAL4 ra`: The right ascension angle of the source, in radians in the range \([0, 2\pi]\).
- `REAL4 dec`: The declination angle of the source, in radians in the range \([-\pi/2, \pi/2]\).
- `REAL4Vector *spindown`: A vector \( \vec{\lambda} = (\lambda_0, \ldots, \lambda_{n-1}) \) of parameters describing a slow intrinsic frequency drift \( f = f(t) \) of the source: \( \lambda_k = f^{-1}d^{k+1}f/dt^{k+1} \) at the time given by \( \text{start} \) (below).
- `LIGOTimeGPS start`: The initial time over which the polynomial fit applies.
- `REAL4Sequence *tBound`: The sequence of times \( t\text{bound}(i) \) defining the endpoints of the fitting regions, given in seconds after the time \( \text{start} \). The first fitting region \( i = 0 \) runs from \( \text{start} \) to \( \text{start} + t\text{bound}(0) \), the next from there to \( \text{start} + t\text{bound}(1) \), and so on.
- `REAL4Sequence *t0`: The sequence of times \( t(i) \) in each fitting region at which the polynomial fits are computed, given in seconds after the time \( \text{start} \).
- `REAL4VectorSequence *polyco`: A sequence of vectors \( \vec{a}(i) = (a_0(i), a_1(i), \ldots) \) giving the coefficients of the polynomial fit at each time \( t(i) \). Each element \( a_k(i) \) has units of \( s^{1-k} \).

**Structure ResampleParamStruc**

This structure stores extra parameters required to construct a `ResampleRules` object from a `PolycoStruc` object. The fields of this structure are:

- `LIGOTimeGPS start`: The initial time for which the resample rules will apply.
- `LIGOTimeGPS stop`: The final time for which the resample rules will apply.
- `REAL8 deltaT`: The sampling interval before decimation, in seconds.
- `INT4 decimate`: The decimation factor.
27.7.1 Module CreateResampleRules.c

Creates an object of type ResampleRules according to a piecewise polynomial fit to the canonical time $\tau(t)$.

Prototypes

```c
void LALCreateResampleRules( LALStatus *stat, ResampleRules **rules, PolycoStruc *polyco, ResampleParamStruc *params )
```

Description

This function creates an object **rules of type ResampleRules, according to the polynomial fit to the canonical time function found in *polyco and the sampling parameters found in *params. See the header Resample.h for a description of these datatypes. Initially the output handle must be a valid handle (rules=NULL) but should not point to an existing object (*rules=NULL).

Algorithm

**Formulae for computing ResampleRules:** Since the resampling rules can be expected to be applied to datasets with a huge number of sample times, the primary concern when designing the algorithm was to maintain a low operation count per sample. For instance, a simple and easy approach would be to step through the time domain at the rate of one resampled time interval $d\Delta t$ per step, and record how many steps it takes for $\tau - t$ to change by one (unresampled) interval $\Delta t$. However, this approach requires us to recalculate $\tau - t$ at each sample time, when it is only likely to change significantly every thousand or so sample times. A much better approach is to invert the polynomial fit to give $t$ as a function of $\tau - t$; that is, we change $\tau - t$ by a discrete amount $\Delta t$, and find the corresponding change in $t$. This function need only be evaluated once for each distinct value of $\tau - t$, rather than once for each distinct value of $t$.

At present, CreateResampleRules() considers polynomial fits to $\tau(t)$ only up to quadratic order. This is almost certainly sufficient for most conventional pulsar searches, where the dominant high-order variation in $\tau$ over short periods (under one hour) arises from the Earth’s motion: $\tau - t \sim (R_{\text{Earth}}/c) \sin(2\pi t/P_{\text{Earth}})$. Since one normally applies polynomial fits to $\tau(t)$ over hour-long time stretches, $t$ is never more than half an hour ($1/48P_{\text{Earth}}$) from a fitting point, and the maximum error in $\tau - t$ due to cubic and higher terms is $\sim (R_{\text{Earth}}/c)(2\pi/48)^3/3! \sim 8 \mu s$. This is insignificant at the 32 kHz sampling rate expected for LIGO.

We do most of the computational work in dimensionless quantities normalized by the resampled time interval $d\Delta t$ (where $\Delta t$ is the unresampled time interval and $d$ is the decimation factor). In this way, dimensionless times can be converted into numbers of samples simply by rounding. In any given fitting region $[t_{\text{bound}(i-1)}, t_{\text{bound}(i)})$ we define dimensionless parameters:

$$
\begin{align*}
n &= \frac{\tau - t}{\Delta t}, \\
T &= \frac{t - t_{(i)}}{d\Delta t}, \\
A_k &= a_{k(i)}(d\Delta t)^{k-1}.
\end{align*}
$$

Eq. (27.4) in the header Resample.h therefore gives us, for a quadratic fit:

$$
\frac{n}{d} = A_0 + A_1 T + A_2 T^2.
$$

If one wants to know whether $n$ is increasing or decreasing with $T$, that is given by the sign of $A_1 + 2A_2 T$.

To compute the resample rules, we want to find the (integral) intervals in $T$ that correspond to (integral) shifts in $n$. Inverting the quadratic formula, we have the obvious solutions:

$$
T = \left(-\frac{A_1}{2A_2}\right) + s \times \sqrt{\left[-\frac{A_1}{2A_2}\right]^2 - \frac{A_0}{A_2} + \left(\frac{1}{dA_2}\right) \times n},
$$

(27.6)
where $s = \pm 1$, depending on the sign of $A_2$ and whether $n$ is increasing or decreasing with $T$ ($s$ has the same sign as $A_2$ if $n$ is increasing, and the opposite if $n$ is decreasing). Clearly most of these coefficients need only be computed once for any given polynomial fit, reducing the operation cost per evaluation of $T$.

The procedure for computing a resample rule is to keep track of the current values of $n$ and $T$, and whether $n$ is increasing or decreasing with $T$. One then increments/decrements $n$ by 1 (the shift), computes a new value of $T$, and finds the difference from the old value (the interval). If the interval is less than a whole number, then increase the size of the shift by 1 and try again. If the argument of the square root goes negative, then $\tau - t$ has reached a turning point: reverse the direction of the shift and the value of $s$. If $T$ moves outside the current fitting region, then get the new polynomial fit and recompute $A_k$ and $T$ (making sure to translate the old value of $T$, since the origin $T = 0$ has moved), and determine anew whether $n$ is increasing or decreasing. Continue until $T$ moves out of the timespan for which we want to compute resample rules.

If the *polyco structure contains only two polynomial coefficients per fitting interval, or if $A_2$ is dangerously small in a given fitting interval, then the CreateResampleRules() routine reverts to a linear fit:

$$ T = \left( -\frac{A_0}{A_1} \right) + \left( \frac{1}{dA_1} \right) \times n. \quad (27.7) $$

If the *polyco structure contains only one polynomial coefficients per fitting interval, or if $A_1$ is also dangerously small in a given fitting interval, then the CreateResampleRules() routine reverts to a constant fit: a shift is computed from the value of $n = dA_0$ at middle of the fitting region ($T = 0$).

Computational details: The basic structure of the algorithm is an inner loop and an outer loop. The inner loop is iterated once each time that $n$ is incremented or decremented and a new $T$ is computed; it terminates when $T$ moves out of the current region of fit, or past the end of the desired timespan. The outer loop is iterated once for each set of polynomial fitting parameters that cover the desired timespan, and normally terminates only when $T$ leaves that timespan. Loop termination is done with break and return commands, so as to avoid unnecessary repetition of tests.

Once some initial setup is done by CreateResampleRules(), this function calls distinct subroutines to perform quadratic, linear, or constant fits to $\tau - t$, depending on the number of polynomial coefficients per fitting region. This avoids repeatedly querying the number of coefficients.

Since the length of the arrays in the ResampleRules object are determined in the process of computing their contents, one must allocate temporary storage with care. On the one hand, one doesn’t want to allocate vastly more memory than will be required; on the other hand, allocating nodes to a linked list every time one evaluates $T$ would needlessly slow execution. Unfortunately, the only hard upper limit on the size of the ResampleRules arrays is the total number $N_{\text{max}}$ of resampled data in the timespan covered by the rules, and this probably overestimates the actual requirements by a factor of a thousand or so.

The solution taken here is to allocate a two-dimensional array of integers, with $2\sqrt{N_{\text{max}}}$ rows of $\sqrt{N_{\text{max}}}$ elements. Each pair of rows represents a continued sequence of intervals and shifts, and a new pair is allocated only when the previous one is full. This requires the data to be copied into the output array at the end of execution, and adds an extra test to the inner loop to determine when new space is needed, but the actual memory allocation is done infrequently and in reasonably-sized blocks. For instance, a week-long chunk of data sampled at 2 kHz would have $N_{\text{max}} = 1.2 \times 10^9$, but might actually require only $\sim 2$ million integers. These would end up being stored in $\sim 60$ arrays of length 34780.

Uses

LALMalloc()
LALCalloc()
LALFree()

Notes
27.7.2 Module DestroyResampleRules.c

Destroys an object of type ResampleRules.

Prototypes

```c
void LALDestroyResampleRules( LALStatus *stat, 
                              ResampleRules **rules )
```

Description

This function destroys an object **rules of type ResampleRules, and sets *rules to NULL.

Algorithm

Uses

```c
void LALFree()
```

Notes

Author: Creighton, T. D.
Revision: $Id: DestroyResampleRules.c,v 1.3 2007/06/08 14:41:51 bema Exp$
27.7.3 Module ApplyResampleRules.c

Resamples a time series according to a set of resampling rules.

Prototypes

```c
void LALApplyResampleRules( LALStatus *stat,
    REAL4TimeSeries *output,
    REAL4TimeSeries *input,
    ResampleRules *rules )
```

Description

This function sets `output->deltaT` and fills `output->data` with data from `*input`, using the resampling rules specified in `*rules`. If the timespan required to fill `output->data` is not a subset of the timespan covered by `*input` or `*rules`, the data at the nonintersecting times are set to zero.

Algorithm

At present this routine is just a stub. It does not apply or even check `*rules`, and instead simply makes `*output` equivalent to (a subset of) `*input`.

Uses

Notes
27.7.4 Module PolycoToTimingDifference.c

Computes values of the timing difference \((\tau - t)/\Delta t\) from a polynomial fit.

Prototypes

```c
void LALPolycoToTimingDifference( LALStatus *stat,
                       REAL4TimeSeries *difference,
                       PolycoStruc *polyco );
```

Description

This function fills a time series \(*difference\) with the values of the normalized timing difference \((\tau - t)/\Delta t\) between the detector time \(t\) and some canonical time \(\tau(t)\), where \(\Delta t\) is the sampling interval in \(t\). The timing difference function is computed from the piecewise-polynomial fit stored in \(*polyco\), via Eq. (27.4).

Algorithm

By storing the timing difference as a dimensionless quantity, it is relatively easy to determine rules for resampling the datastream at equal intervals in \(\tau\), since it gives the number of samples difference between the two time coordinates. When resampling a time series in \(t\), simply track the value of \((\tau - t)/\Delta t\): When this value increases by +1, choose the next sample after the one that would otherwise have been chosen; when the value decreases by −1, choose (or repeat) the sample immediately preceding the one that would otherwise have been chosen.

However, this is not a particularly efficient routine for computing the resampling method, as it requires several floating-point operations per sample, which is an unacceptable computational burden for any optimized pulsar search. It is primarily used to visualize and check the pulsar phase modulation model. See the routine in CreateResampleRules.c for a more efficient algorithm.

Uses

Notes
27.7.5 Module RulesToTimingDifference.c

Computes values of the timing difference $\frac{\tau - t}{\Delta t}$ from a set of resampling rules.

Prototypes

```c
void LALRulesToTimingDifference( LALStatus *stat,
                                  REAL4TimeSeries *difference,
                                  ResampleRules *rules )
```

Description

These function fills a time series `difference` with the values of the normalized timing difference $\frac{\tau - t}{\Delta t}$ between the detector time $t$ and some canonical time $\tau(t)$, sampled as a function of $t$. This is computed using the resampling rules `rules`, which specify how one resamples a function of $t$ as a function of $\tau$. Note that $\Delta t$ in the formula above is the raw (unresampled) sampling interval in $t$ used to compute the resampling rules, not necessarily the sampling interval of the time series. Thus a shift of $\pm 1$ in the resampling rules corresponds to a change of $\pm 1$ in the normalized timing difference.

Algorithm

This routine is quite simple: it increments a time counter `tNext` by resampling rule intervals until it finds the next correction after the current time $t$, accumulating the shifts to the timing function given by `rules->shift`. It then increments $t$ until it steps past `tNext`, each time filling `difference->data` with the current value of the cumulative timing error $\frac{\tau - t}{\Delta t}$. These are iterated until the time series is completely filled. Internally, all times are all converted to units of `rules->deltaT` and are measured from `rules->start`.

It is worth noting that a shift in the resampling, as given by `rules->shift`, adjusts the value of $\tau - t$ by shifting the current value of $t$. Thus the difference in detector time between the $n^{th}$ and the $(n+1)^{th}$ resampling corrections is $(\text{interval}[n] \times \text{decimate} + \text{shift}[n]) \times \Delta t$, while the difference in canonical time is simply $\text{interval}[n] \times \text{decimate} \times \Delta t$. Since `difference` is assumed to be sampled in detector time $t$, we use the first of these formulae.

Uses

Notes
### 27.7.6 Program ResampleTest.c

Tests the routines in Resample.h.

**Usage**

```bash
ResampleTest [-d debuglevel] [-p psfile] [-t tfile] [-c n a f] [-m dec df fm]
```

This program generates a quasiperiodic time series having a sinusoidal phase modulation, and generates a piecewise-polynomial fit to the phase function. It then generates and applies stroboscopic resampling rules to the time series to produce a monochromatic signal. The following option flags are accepted:

- **-d** Sets the global `lalDebugLevel` to the specified `debuglevel`.
- **-p** Power spectra of the time series before and after demodulation will be written to the file `psfile`.
- **-t** The timing difference function \((\tau - t)/\Delta t\) computed in three ways (analytically, from a polynomial fit, and from the resampling rules), will be written to the file `tfile`. See below for notation.
- **-c** Sets parameters for the “carrier” signal: the number of points `n`, the amplitude `a`, and the frequency `f`.
- **-m** Sets parameters for the signal modulation and resampling: the decimation factor `dec`, the maximum change in signal frequency `df`, and the frequency of the modulation `fm`.

All frequencies are in units of the sampling rate, which is an arbitrary scale. With no options, the program runs with `lalDebugLevel`=0, produces no output, and uses internally `#defined` signal parameters.

**Exit codes**

<table>
<thead>
<tr>
<th><code>&lt;name&gt;</code></th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NORM</td>
<td>0</td>
<td>&quot;Success, normal exit&quot;</td>
</tr>
<tr>
<td>SUB</td>
<td>1</td>
<td>&quot;Recursive error&quot;</td>
</tr>
<tr>
<td>ARG</td>
<td>2</td>
<td>&quot;Error parsing arguments&quot;</td>
</tr>
<tr>
<td>BAD</td>
<td>3</td>
<td>&quot;Bad argument value&quot;</td>
</tr>
<tr>
<td>FILE</td>
<td>4</td>
<td>&quot;Error opening or writing to output file&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants `RESAMPLETESTC_E<name>`, and the status descriptions in `RESAMPLETESTC_MSGE<name>`. The source code with these messages is in ResampleTest.c on line 1.79.

**Algorithm**

The modulated signal is of the form \(s(t) = A\sin[\phi(t)]\), where \(A\) is a rather arbitrary amplitude, and \(\phi(t)\) is a phase function of the form:

\[
\phi(t) = 2\pi f_c t + \frac{\Delta f}{f_m} \sin(2\pi f_m t) .
\]

Here \(f_c\) is the average “carrier” frequency, \(f_m\) is the frequency of the modulation, and \(\Delta f\) is the maximum change in the “instantaneous” frequency of the signal (or, equivalently, the separation between the carrier and sidebands in the power spectral density). The canonical (demodulated) time coordinate for this phase function is \(\tau = \phi/2\pi f_c\). The demodulation routines require quadratic fits to the function \(\tau - t\) at various times \(t_0\):

\[
\tau - t = \frac{(\Delta f/f_c)}{2\pi f_m} \sin(2\pi f_m t_0) + \frac{(\Delta f/f_c)}{2\pi f_m} \cos(2\pi f_m t_0)(t - t_0) - \pi f_m (\Delta f/f_c) \sin(2\pi f_m t_0)(t - t_0)^2 ,
\]  \hspace{1cm} (27.8)

with residuals less than \((2/3)\pi^2 f_m^2 (\Delta f/f_c)(t - t_0)^3\). We require this residual to be always less than one sample interval \(\Delta t\). This means that a piecewise-quadratic fit to the phase function must be evaluated at times \(t_0\) separated by no more than:

\[
\Delta t_0 \leq \sqrt[3]{\frac{12 f_c \Delta t}{\pi^2 f_m^2 \Delta f}} ,
\]  \hspace{1cm} (27.9)
noting that each piecewise fit is good for a time interval \( t_0 \pm \Delta t_0/2 \) about each central time \( t_0 \).

Thus to create a piecewise-polynomial fit defined by \texttt{PolycoStruc}, this program simply define a set of fitting times \( t_0[k] = (2k + 1)\Delta t_0/2 \), and computes the appropriate components of \texttt{polyco} from Eq. (27.8), above.

Uses

\begin{verbatim}
lalDebugLevel\nLALPrintError()\nLALSCreateVector()\nLALSDestroyVector()\nLALCreateResampleRules()\nLALApplyResampleRules()\nLALDestroyResampleRules()\end{verbatim}

Notes
References

Chapter 28

Package **pulsar**: amplitude folding routines

Greg Mendell

Contains function LALFoldAmplitudes: folds amplitudes into phase bins.

Files:

- **FoldAmplitudes.h** header file
- **FoldAmplitudes.c** source code
- **FoldAmplitudesTest.c** test code
- **foldamplitudes.tex** overview

Periodic sources of gravitational radiation will produce measured strains of the following form:

\[ c[i] = A(t_i, \vec{\lambda}) \sin[\Phi(t_i, \vec{\lambda})] + n(t_i) \]

In this equation \( c[i] \) is the discrete time series output of the detector (perhaps after some data conditioning, such as being resampled, narrow banded, or with instrument line noise removed). The amplitude, \( A(t_i, \vec{\lambda}) \), is assumed roughly constant at the gravity wave source, but is modulated by variation in the detector’s response due to the Earth’s motion. The phase, \( \Phi(t_i, \vec{\lambda}) \), is modulated by both the intrinsic spin down of the source, and the changes in relative motion between the source and the detector. This can be calculated for known pulsars. The vector \( \vec{\lambda} \) is a vector of parameters that describe the sky position, etc., of the source and location, etc., of the detector. Finally, \( n(t_i) \) is the noise, which also includes any other signals that are not coherent with the phase \( \Phi(t_i, \vec{\lambda}) \).

The folded amplitude is given by

\[ c_F[j] = \sum_{i'} \left\{ A(t_i, \vec{\lambda}) \sin[\Phi(t_i, \vec{\lambda})] + n(t_i) \right\}, \]

where the sum over \( i' \) means sum over all \( i' \)'s with \( \Phi \) in phase bin \( j \). If the bin sizes are sufficiently small, then \( c_F[j] \) can be approximated as

\[ c_F[j] = \sin \Phi_j \sum_{i'} A(t_i, \vec{\lambda}) \sum_{i'} n(t_i), \]

where \( \Phi_j \) is representative of the phase for bin \( j \) (e.g., the phase corresponding to the midpoint of the bin). However, because of amplitude modulation, the amplitudes that are added to a phase bin are not guaranteed to enter with the same sign. Thus, some sort of amplitude demodulation should be done.

If we demodulate \( A(t_i, \vec{\lambda}) \) (for example, in a minimum way such as multiplying by the sign of the response function) we multiply each element of the vector \( c[i] \) by an amplitude demodulation factor \( D(t_i) \)

\[ c_{D,F}[j] = \sin \Phi_j \sum_{i'} D(t_i) A(t_i, \vec{\lambda}) \sum_{i'} D(t_i) n(t_i), \]

815
If the average value of $D(t_i)$ is zero, and is not correlated with the noise, then

$$\sum_{i'} D(t_i)n(t_i) \approx 0$$

However, the average value of $D(t_i)$ is probably not zero. The following is a very preliminary suggestion of how to further reduce the noise. Consider folding the measured strains, $c[i]$, again, but this time shifting the phase bins by $\pi$. Define this phase shifted folded amplitude as:

$$c_{\pi,D,F}[j] = \sin(\Phi_j + \pi) \sum_{i''} D(t_i)A(t_i, \bar{\lambda}) + \sum_{i''} D(t_i)n(t_i),$$

where the sum over $i''$ means sum over all $i$’s with $\Phi + \pi$ in phase bin $j$. This will reverse the sign of the sum of the amplitudes that enter into each phase bin, but the sum of the noise contributions into each bin should be roughly the same. If the signal we are searching for is present, then amplitudes, $A(t_i, \bar{\lambda})$ are correlated with $D(t_i)$ such that

$$\sum_{i'} D(t_i)A(t_i, \bar{\lambda}) \approx \bar{A} = \text{constant}$$

Thus,

$$c_{D,F}[j] - c_{\pi,D,F}[j] \approx 2\bar{A} \sin \Phi_j,$$

plus residual noise. In practice, one needs to fold the amplitudes only once, and then make the replacement

$$c_{D,F}[j] \rightarrow c_{D,F}[j] - c_{D,F}[(j + N/2) \% j],$$

where $N$ is the number of phase bins. We can then statistically analyze the hypothesis that the demodulated folded amplitudes correspond to a sinusoid.


28.1 Header FoldAmplitudes.h

Synopsis

#include <lal/FoldAmplitudes.h>

Contains prototypes for:
struct FoldAmplitudesInput
struct FoldAmplitudesParams
function LALFoldAmplitudes

Error conditions

<table>
<thead>
<tr>
<th>name</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULLP</td>
<td>1</td>
<td>&quot;Null pointer!&quot;</td>
</tr>
<tr>
<td>VECSIZE</td>
<td>2</td>
<td>&quot;Input vectors were not the same length!&quot;</td>
</tr>
<tr>
<td>NUMBINS</td>
<td>3</td>
<td>&quot;Number of bins was less than 1!&quot;</td>
</tr>
<tr>
<td>BINSIZE</td>
<td>4</td>
<td>&quot;Bin max was less than bin min!&quot;</td>
</tr>
<tr>
<td>BIMIN</td>
<td>5</td>
<td>&quot;Bin min was not zero; nonzero bin min has not yet been implemented!&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants FOLDAMPLITUDESH_E{name}, and the status descriptions in FOLDAMPLITUDESH_MSGE{name}. The source code with these messages is in FoldAmplitudes.h on line l.109.

Structures

typedef struct tagFoldAmplitudesInput
{
    REAL4Vector *amplitudeVec; input vector of amplitudes
    REAL4Vector *phaseVec; input vector of phases
} FoldAmplitudesInput;

typedef struct tagFoldAmplitudesParams
{
    INT4 numBins; number of bins
    REAL4 binMin; minimum phase to bin
    REAL4 binMax; maximum phase to bin
} FoldAmplitudesParams;
28.1.1 Module FoldAmplitudes.c

Prototypes

\begin{verbatim}
void LALFoldAmplitudes( LALStatus *status, REAL4Vector *output, 
    const FoldAmplitudesInput *input, const FoldAmplitudesParams *params )
\end{verbatim}

Description

Contains source for function LALFoldAmplitudes:

inputs: a vector of amplitudes and a vector of phases.

params: number of phase bins, the minimum phase to bin, and the maximum phase to bin.

action: for each phase, the phase is first reduced by modulo arithmetic to a value between \( \text{binMin} \) and \( \text{binMax} \). The corresponding amplitudes is then added to the corresponding phase bins. The width of each bin is \( (\text{binMax} - \text{binMin})/\text{numBins} \).

output: a vector of folded amplitude; component \( i \) is the folded amplitude for phase bin \( i \).

Algorithm

Algorithm for folding amplitudes into phase bins:

1) Reduce the phase to a value between \( \text{binMax} \) and \( \text{binMin} \). Note that \( \text{binMax} - \text{binMin} = \text{binRange} \).
2) Find the bin index corresponding to this phase.
3) Add the amplitude from the input data vector to this phase bin.

Notes:

(i) The function is more efficient if \( \text{binRange} == 1 \).
(ii) The results are stored in the REAL4Vector output structure data vector.
(iii) Only \( \text{binMin} = 0.0 \) is currently supported.

Code:

\begin{verbatim}
if (binRange == 1.0) {
  for ( i = 0 ; i < amplitudeVec->length ; ++i )
  {
    phase = phaseVec->data[i] - floor( phaseVec->data[i] );
    binIndex = (INT4) floor( phase/binSize );
    output->data[binIndex] += amplitudeVec->data[i];
  }
} else {
  for ( i = 0 ; i < amplitudeVec->length ; ++i )
  {
    phase = phaseVec->data[i] - floor( phaseVec->data[i]/binRange ) * binRange;
    binIndex = (INT4) floor( phase/binSize );
    output->data[binIndex] += amplitudeVec->data[i];
  }
}
\end{verbatim}
Uses

For use in known pulsar search.

Notes
28.1.2 Program FoldAmplitudesTest.c

The test program test each of the error conditions, and then test the output of known input with the expected output.

Usage

FoldAmplitudesTest

Description

Exit codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOM</td>
<td>0</td>
<td>&quot;Nominal exit&quot;</td>
</tr>
<tr>
<td>CHK</td>
<td>1</td>
<td>&quot;Error checking failed to catch bad data&quot;</td>
</tr>
<tr>
<td>FLS</td>
<td>2</td>
<td>&quot;Incorrect answer for valid data&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants FOLDAMPLITUDESTESTC_E<name>, and the status descriptions in FOLDAMPLITUDESTESTC_MSGE<name>. The source code with these messages is in FoldAmplitudesTest.c on line l.79.

Uses

LALFoldAmplitudes()

Notes
Chapter 29

Package pulsar: Coherent search routines

Steven Berukoff, M. Alessandra Papa

This package provides a routine to perform a demodulation on a set of data. In particular, this routine works with frequency domain data by combining short timescale Fourier Transforms (SFTs) into longer time baseline demodulated Fourier Transforms (DeFTs). If the assumptions under which the method was developed are met ([1]), then the demodulation procedure concentrates the total power (within 5% – 10%) in a single frequency bin. In practice, due to the discretization of frequency space, this power may be shared between two neighbouring bins.

The procedure follows that outlined in [1] and is part of the continuous-wave search algorithm outlined in [2]. Briefly, the routine takes input SFTs, corrects for modulation effects due to intrinsic frequency spindown and Earth’s motion, and outputs a DeFT of long time baseline. In general the routine can be easily adapted to correct for an arbitrary modulation effect simply by the use of a suitable timing routine, here \texttt{tdb()}. The package is organized under the headers \texttt{LALDemod.h}, \texttt{LALComputeAM.h}, and \texttt{ComputeSky.h} and the modules \texttt{LALDemod.c} and \texttt{ComputeSky.c}. 

29.1 Header LALDemod.h

Computes a demodulated transform.

Synopsis

```
#include <lal/LALDemod.h>
```

The following is a brief synopsis of the demodulation, or 'Coherent Transform', procedure.

In order to remove frequency and amplitude modulation of a time series \( x_a \), we need two basic components:

**Frequency modulation information** This is given through a phase model \( \Phi \).

**Amplitude modulation information** This is given through two functions \( \hat{a} \) and \( \hat{b} \), which are derived from the beam-pattern functions \( F_+ \) and \( F_\times \).

Given these, the F statistic in the \( b^{th} \) frequency bin is

\[
F_b = \frac{4}{S_h(f_0)T_0} \frac{B|\hat{F}_a|^2 + A|\hat{F}_b|^2 - 2C\Re(F_aF_b^*)}{D}
\]  (29.1)

where

\begin{align*}
\hat{F}_a &= \sum_{a=0}^{NM-1} x_a \hat{a} e^{-2\pi i \Phi_{ab}(\vec{\lambda})} \quad (29.2) \\
\hat{F}_b &= \sum_{a=0}^{NM-1} x_a \hat{b} e^{-2\pi i \Phi_{ab}(\vec{\lambda})} \quad (29.3)
\end{align*}

\( T_0 \) is the observation time, \( S_h \) is the noise power spectral density, and \( A, B, C, \) and \( D \) are constants.

In writing the previous equation we have assumed that there is a total of \( M \cdot N \) data samples and \( 0 \leq a < MN \). \( \Phi_{ab} \) is the expected phase at time \( a \) for an intrinsic emission frequency \( T_{\text{DeFT}} \) (where the denominator is the DeFT time baseline). \( \Phi \) depends on \( \vec{\lambda} \), a vector of parameters that defines the phase model. Typically these are the source location and the spin-down parameter values of the template source for which one is demodulating. For simplicity, we will focus only on \( F_a \); the analysis for \( F_b \) is identical. Let us now suppose that the time series \( x_a \) is composed of \( M \) chunks, each of \( N \) samples. If we introduce a short-time index \( 0 \leq j < N - 1 \) and a short time-series index \( 0 \leq \alpha < M - 1 \), so that \( a = N\alpha + j \), we can rewrite the above sum as

\[
\hat{F}_a(\vec{\lambda}) = \sum_{\alpha=0}^{M-1} \sum_{j=0}^{N-1} x_{\alpha j} \hat{a}_{\alpha j} e^{-2\pi i \Phi_{ab}(\vec{\lambda})} \quad (29.4)
\]

Note that \( \hat{a}(t) \) is a periodic function with period equal to one sidereal day. Since the sum over \( N \) is on a timescale much shorter than that (say, 1 hour), then \( \hat{a}(t) \) won’t change significantly, and thus can be taken outside of that summation, and then is evaluated at the midpoint of each SFT time. Now, If \( \tilde{x}_{\alpha k} \) is the matrix of FTs formed along the short time index \( j \)

\[
x_{\alpha j} = \frac{1}{N} \sum_{k=0}^{N-1} \tilde{x}_{\alpha k} e^{2\pi i \frac{jk}{N}} \quad (29.5)
\]

making the appropriate substitutions, Eq\(29.2\) becomes

\[
\hat{F}_a(\vec{\lambda}) = \sum_{\alpha=0}^{M-1} \hat{a}_\alpha \sum_{k=0}^{N-1} \tilde{x}_{\alpha k} \left\{ \frac{1}{N} \sum_{j=0}^{N-1} e^{-2\pi i (\Phi_{\alpha,j}(\vec{\lambda}) - \frac{j}{N})} \right\} \quad (29.6)
\]

We assume that the phase evolution can be described as linear in \( t \) during the time duration \( T_{SFT} \); thus we can Taylor-expand \( \Phi \) around the temporal midpoint of every SFT time chunk. For large values of \( N \), the summation over \( j \) in eq. (29.12) can be expressed in closed form, thus saving computations, and eq. (29.12) can be rewritten as

\[
\hat{F}_a = \sum_{\alpha=0}^{M-1} \hat{a}_\alpha e^{-i\omega_\alpha} \sum_{k=0}^{N-1} \tilde{x}_{\alpha k} P_{\alpha k}(\vec{b}, \vec{\lambda}), \quad (29.7)
\]
with

\[ P_{\alpha k}(b, \lambda) = \frac{\sin x' - i - \cos x'}{x'} \]  \hspace{1cm} (29.8)

\[ x' = \sum f_s B_{sa} - k \]  \hspace{1cm} (29.9)

\[ y_{\alpha} = \sum f_s A_{sa}. \]  \hspace{1cm} (29.10)

In the previous expressions \( f_s \) indicate the spin-down parameters of different orders (labeled by the index \( s \)), and \( A_{sa} \) and \( B_{sa} \) are functions that depend on the phase evolution, whose values depend on \( \alpha \) and on \( \lambda \). The values of these functions are calculated by the \texttt{ComputeSky()} routine, also in this package. Incidentally, in the code, these are the values contained in the variable \texttt{skyConst}. Note that the function \( P_{\alpha k} \) is peaked around \( x' = 0 \). Thus in the summation over \( k \) in eq. (29.7) one only needs to consider a few values (\texttt{NTERMS}) of \( k \) around \( k^{*} \) such that \( x'(k^{*}) \approx 0 \). This approximation again saves computations. Eq. (29.7) can then be rewritten as

\[ \hat{F}_a = \sum_{\alpha=0}^{M-1} \hat{a}_{\alpha} e^{iy_{\alpha}} \sum_{k=k^{*} \pm \text{\texttt{NTERMS}}}^{k^{*} \pm \text{\texttt{NTERMS}}} \hat{x}_{\alpha \beta} P_{\alpha k}(b, \lambda). \]  \hspace{1cm} (29.11)

If \texttt{NTERMS} is 8 the power loss due to this approximation is less than \( \sim 5\% \).

Now, computing \( \hat{F}_a \) and \( \hat{F}_b \) can be done in parallel; given the approximations we have made, for each iteration of the \( \alpha \) loop, one computes first \( P_{\alpha k} \) (through the \( k \)-loop), multiplies by \( \hat{x}_{\alpha k} \), and then forms the statistics of (29.2) at the same time. After all the iterations of the \( \alpha \) loop are complete, that is, when all SFTs have been exhausted, the final statistic is computed.

### Error codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
</table>
| NULL   | 1    | "Arguments contained an unexpected null pointer"

The status codes in the table above are stored in the constants \texttt{LALDEMODH_E<name>}, and the status descriptions in \texttt{LALDEMODH_MSGE<name>}. The source code with these messages is in \texttt{LALDemod.h} on line 1.127.

### Types

#### Structure DemodPar

This structure contains the parameters for the demodulation routine. The parameters are:

- **INT4 spinDwnOrder** Maximum order of spdwn parameter
- **REAL8 *skyConst** The array of sky constants.
- **REAL8 *spinDwn** The set of template spin down parameters.
- **AMCoeffs *amcoe** The values of the function \( a \) and \( b \), plus their scalar products.
- **REAL8 f0** The minimum search frequency
- **REAL8 df** The search frequency spacing
- **INT4 SFTno** The number of SFTs in coherent timescale
- **INT4 Dterms** Terms used in the computation of the dirichlet kernel
- **INT4 ifMin** The index of the minimum frequency of the SFT frequency band.
- **INT4 imax** How many frequencies are searched.
- **BOOLEAN returnFaFb** Wether or not to include the values \( F_a/F_b \) in the return-structure \( Fstat \).

Note that when \( s = 0 \) the values computed are coefficients of the intrinsic frequency and thus must be computed for the value corresponding to the index \( b \).
Structure **LALFstat**

This structure contains the results from LALDemod: either only the value of the $\mathcal{F}$-statistic $F$, or also the values of the individual "filters" $F_a$ and $F_b$, depending on the DemodPar->returnFaFb.

*NOTE:* the memory has to be allocated before calling LALDemod().

**REAL8** $*F$ Array of values of the $\mathcal{F}$ statistic.

**COMPLEX16** $*F_a$ Results of match filter with $a(t)$.

**COMPLEX16** $*F_b$ Results of match filter with $b(t)$.
29.1.1 Module LALDemod.c

Computes a demodulated Fourier transform (DeFT) given a set of input short Fourier transforms (SFT).

Prototypes

```c
void LALDemod(LALStatus *status, LALFstat *Fstat, FFT **input, DemodPar *params)
void LALDemodFAST(LALStatus *status, LALFstat *Fstat, FFT **input, DemodPar *params)
```

Description

This routine computes the $F$ statistic for a set of templates that are defined by: one sky position, a set of spin-down parameters and a band of possible signal frequencies. The $F$ statistic is described in JKS, Phys Rev D 58, 063001 (1998). Here, it has been adapted to a single emission frequency model.

The parameter structure defines the search frequency band ($f_0$ and $i_{\text{max}}$), the search frequency resolution ($df$) the first frequency of the input SFTs ($i_{\text{min}}$), how many SFTs have to be combined ($\text{SFTno}$) and template parameters ($\ast \text{spinDwnOrder}$, $\ast \text{spinDwn}$ and $\ast \text{skyConst}$). $\text{amcoe}$ contains the values of the amplitude modulation functions $a$ and $b$. $\text{Dterms}$ represents the numbers of terms to be summed to compute the Dirichlet kernel on each side of the instantaneous frequency.

The input is: **input, an array of structures of type FFT. This data type will soon disappear as it is just a complex8frequencyseries.

The output is a pointer a structure of type LALFstat containing an array of the values of $F$. In addition, if DemodPar->returnFaFb == TRUE, the values of $F_a$ and $F_b$ will be returned in addition. (Memory has to be allocated correctly beforehand!)

Algorithm

The routine implements the analytical result of eq. 29.1.11. It thus uses a nested-loop structure, which computes $F$ for all the template frequencies.

The outer most loop is over the search frequencies. The next loop is over $\alpha$, which identifies the SFTs. The value of $k^*$ is then computed using the second of Eq. 29.8 and thus the summation over $k$ of Eq 29.1.11 is carried out, with a loop over Dterms. In this loop the product $\tilde{x}_{\alpha k}P_{\alpha k}$ is calculated. Once this loop completes, $e^{ik^*}$ is computed, the summation over $\alpha$ performed and, finally, the code yields the DeFT $\hat{x}_b$. It can be seen that the code closely follows the analytical development of the formalism.

Finally, note that in order to avoid repeated trigonometric function computations, a look-up-table (LUT) for sine and cosine is constructed at the beginning of the routine.

Uses

None

Notes
29.1.2 Program LALDemodTest.c

Performs required tests of \texttt{LALDemod()}.

\textbf{Usage}

\texttt{LALDemodTest -i <input data file> [-d <gap>] [-n] [-o]}

\textbf{Description}

This routine performs tests on the routine \texttt{LALDemod()}. Options:

- \texttt{-i} – the input data file (default is 'in.data'; an example is included, format below)
- \texttt{-n} – add zero-mean Gaussian noise to the signal
- \texttt{-d <gap>} – simulate gaps in the data. The number \texttt{<gaps>} refers to the integral number of SFT timescales between adjacent timestamps.
- \texttt{-o} – print out result data files

Structure: In more detail, let us begin with a discussion of the structure of the test code, which is composed of several modules.

- The first module reads in data from an input parameter data file. The parameters must be listed in the input file in the following order, with the corresponding format:

  \begin{verbatim}
  total observation time -- float
cohrent search time -- float
factor by which to modify SFT timescale -- float
Cross amplitude -- float
Plus amplitude -- float
DeFT frequency band (centered by default around \texttt{f0}) -- float
\texttt{f0}, intrinsic frequency of the signal at the beginning of the observation -- float
maximum order of signal spindown parameters -- int
signal spindown parameter 1 -- scientific notation
signal spindown parameter 2 -- scientific notation
signal spindown parameter 3 -- scientific notation
signal spindown parameter 4 -- scientific notation
signal spindown parameter 5 -- scientific notation
signal source right ascension (\alpha) -- float (value in DEGREES)
signal source declination (\delta) -- float (value in DEGREES)
maximum order of template spindown parameters -- int
template spindown parameter 1 -- scientific notation (NOT scaled by \texttt{f0})
template spindown parameter 2 -- scientific notation (NOT scaled by \texttt{f0})
template spindown parameter 3 -- scientific notation (NOT scaled by \texttt{f0})
template spindown parameter 4 -- scientific notation (NOT scaled by \texttt{f0})
template spindown parameter 5 -- scientific notation (NOT scaled by \texttt{f0})
template source right ascension (\alpha) -- float (value in DEGREES)
template source declination (\delta) -- float (value in DEGREES)
  \end{verbatim}

Note: Above, the *signal* spindown parameters are scaled by the intrinsic frequency, while the *template* spindown parameters are not. This is due to the difference in definitions between the \texttt{SimulateCoherentGW()} package, which generates the signal, and this package.

- The next module in the test code, which is optionally executed with the \texttt{'-n'} switch, creates noise using \texttt{LALNormalDeviates()} routine. By design, the noise is created in single precision, and is zero-mean and Gaussian. This noise is added, datum-by-datum, to the time series created in the next module, after the amplitude of the time series has been changed by a factor of \texttt{SNR}, which is specified in the input data file.
• The next module to be invoked creates a time series, according to the standard model for pulsars
  with spindown. This is done by using the `LALGenerateTaylorCW()` and `LALSimulateCoherentCW()`
  functions. This time series undergoes an FFT, and this transformed data then constitutes the SFT data
  to be input to the demodulation code. The fake signal data is characterized by an intrinsic frequency
  at the beginning of the observation plus some other source parameters. The DeFT is produced in a
  band `f0Band` (as specified as an input parameter) and centered at this frequency. The width of the
  band (plus some extra width of 2 × 10^{-4} f_0 Hz) determines the sampling frequency of the time series
  (Nyquist theorem). In practice this would be the inverse FFT of a data set that has been band-passed
  around `f_0` and then appropriately down-sampled (e.g. with a lock-in). The normalization rule for FFT
  data is the following: if sinusoidal data over a time T and with amplitude A is FFT-ed, the sum of
  the square amplitude of the output of the FFT (power) is equal to A^2 T. Thus, the power peak at the
  sinusoids frequency should be expected to be \( \sim \frac{A^2}{2T} \), within a factor of 2. The same normalization
  rule applies to the DeFT data. Thus by piecing together N SFTs we expect a DeFT power peak \( \sim N \)
  higher than that of the SFTs - at least in the case of perfect signal-template match.

  Let us now spend a few words on the choice of the SFT time baseline. Given an intrinsic search
  frequency one can compute the longest time baseline which is still compatible with the requirement
  that the instantaneous signal frequency during such time baseline does not shift by more than a
  frequency bin. This is the default choice for the SFT length, having assumed that the modulation is
  due to the spin of the Earth and having taken a simple epicyclic model to evaluate the magnitude of
  this effect. It is possible to choose a different time baseline by specifying a value for the variable `gap`
  other than 1. Note that the SFT time baseline is approximated to the nearest value such that the
  number of SFT samples is a power of two. This is also well documented in the code.

  The set of SFTs does not necessarily come from contiguous data sets: a set of time stamps is created
  that defines the time of the first sample of each SFT data chunk. The timestamps which are required
  in many parts of the code are generated in a small subroutine `times2()` . This routine takes as input
  the SFT timescale `tSFT`, the number of SFTs which will be created, `mObsSFT`, and a switch which lets
  the code know whether to make even timestamps, or timestamps with gaps (see below for more on
  this). The subroutine then writes the times to the `LIGOTimeGPS` vector containing the timestamps
  for the entire test code, and returns this vector. Note that each datum of the `LIGOTimeGPS` vector
  is comprised of two fields; if accessing the `i`th datum, the seconds part of the timestamp vector `ts`
  is `ts[i].gpsSeconds` and the nanoseconds part is `ts[i].gpsNanoSeconds`. These are the fields which
  are written in this `times()` .

  As an important side note, let us discuss the effect that a vector of timestamps with gaps has on the
  resulting transformed data. Since each of the timestamps refers to the first datum of each SFT, the
  existence of the gaps means that instead of transforming a continuous set of data, we are reduced to
  transforming a piecewise continuous set. Since we can envision these gaps as simply replacing real
  data with zeros, we correspondingly should see a power loss in the resulting FFTs signal bins and a
  broadening of the power spectrum. Since real detectors will clearly have gaps in the data, this effect
  is obviously something we seek to minimize or eliminate if possible. This work continues to be under
  development.

  The total observation time determines how many SFTs and how many DeFTs are created. The actual
  time baseline for both the DeFTs and the total observation time might differ from the ones defined in
  the input file, the reason being that they are rounded to the nearest multiple of the SFT time baseline.
  Note that use is made of the `LALBarycenter()` routine (see section 29.2) which (among other things)
  provides, at any given time, the actual instantaneous position and velocity of a detector at any specified
  location of the Earth with respect to the SSB.

  • Following the creation of a short chunk of time series data, an FFT is performed with the internal
    FFTW routines. This outputs a frequency domain chunk which is placed into the `SFTData` array of
    structures. This will contain all of the SFT data we need to demodulate, and in the future, will be the
    storage area for the real data.

  • The next module begins the demodulation process. First, the parameters for the demodulation rou-
    tine are assigned from values previously calculated in the test code. Similarly, parameters for the
    `LALComputeSky()` routine are assigned. This routine computes the coefficients \( A_{\alpha\alpha} \) and \( B_{\alpha\alpha} \)
    (see section 29.2) of the spin-down parameters for the phase model we have assumed. These coefficients are used
    within the `LALDemod()` routine itself. Since they only depend on the template sky position, in a search
    over many different spin-down parameters they are reused, thus one needs compute them only once.
Then, the `LALComputeAM()` routine is called, to calculate the amplitude modulation filter information. Finally, at last, the demodulation routine itself is called, and, if the command line option `-o` is used, output are several data files containing demodulated data (these are by default named `xhat_#`). These output files have two columns, one for the value of the periodogram and one for the frequency.

**Exit codes**

**Uses**

- `lalDebugLevel`
- `LALMalloc()`
- `LALFopen()`
- `LALFclose()`
- `LALSCreateVector()`
- `LALCreateRandomParams()`
- `LALNormalDeviates()`
- `LALDestroyRandomParams()`
- `LALSDestroyVector()`
- `LALCCreateVector()`
- `LALCreateForwardRealFFTPlan()`
- `LALREAL4VectorFFT()`
- `LALDestroyVector()`
- `LALDestroyRealFFTPlan()`
- `LALGenerateTaylorCW()`
- `LALSImulateCoherentGW()`
- `LALComputeSky()`
- `LALFree()`
- `LALDemod()`
- `LABarycenter()`
- `LALComputeAM()`

**Notes**

The implementation of the code here is intended to give a general outline of what the demodulation code needs to work. Most of this test function performs steps (e.g., noise, time- and frequency-series generation) that will be already present in the data.
29.2 **Header** **ComputeSky.h**

Computes phase coefficients necessary for a correct demodulation.

**Synopsis**

```c
#include <lal/ComputeSky.h>
```

This is a short summary of the analytical calculations which form the basis for the code in this routine.

Recall that a demodulated Fourier Transform (DeFT) is given by

$$
\hat{x}_b(\bar{\lambda}) = \sum_{\alpha=0}^{M-1} \sum_{k=0}^{N-1} \hat{x}_{ak} \left[ \frac{1}{N} \sum_{j=0}^{N-1} e^{-2\pi i (\Phi_{\alpha, k}(\bar{\lambda}) - \frac{2\pi j}{N})} \right]
$$

(29.12)

The index $b$ defines the DeFT frequency bin, the index $\alpha$ loops through the SFTs that build the DeFT, $k$ runs on all the SFT frequency bins, and $j$ is a time index that runs on each SFT. As shown in section 29.1, the next step in the development of the demodulation technique involves Taylor expanding the phase model about the temporal midpoint of each short segment of data, while retaining only first order terms. The Taylor expansion of $\Phi(t)$ about the temporal midpoint $t_{\alpha,1/2}$ is

$$
\Phi_\alpha(t) = \Phi(t_{\alpha,1/2}) + \left[ t - t_{\alpha,1/2} \right] \frac{d\Phi}{dt}(t_{\alpha,1/2})
$$

(29.13)

For each value of $\alpha$, this expression consist of either constant or linear terms in time. With the particular time discretization chosen in this code, $t = t_0 + (N\alpha + j) T_{obs}/NM$, we have

$$
\left[ t - t_{\alpha,1/2} \right] = \frac{T_{obs}}{M} \left( \frac{j}{N} - \frac{1}{2} \right) = T_s \left( \frac{j}{N} - \frac{1}{2} \right),
$$

(29.14)

where $T_s$ is the short time baseline of the $M$ short FTs. On the other hand, the phase can also be expressed as a function of SSB time $T$ (i.e. the time at the solar system barycenter). We will assume the source to be at rest in this reference frame. Now, if one adopts the notation $\Delta T_\alpha \equiv [T(t_{\alpha,1/2}) - T(t_0)]$ and $\dot{T}_\alpha \equiv dt/dt(t_{\alpha,1/2})$ the phase terms in the above equation are (neglecting constants)

$$
\Phi(t_{\alpha,1/2}) = f_0 \Delta T_\alpha + \frac{1}{2} f_1 \Delta T_\alpha^2 + \frac{1}{3} f_2 \Delta T_\alpha^3 + \frac{1}{4} f_3 \Delta T_\alpha^4 + \frac{1}{5} f_4 \Delta T_\alpha^5 + \frac{1}{6} f_5 \Delta T_\alpha^6
$$

(29.15)

$$
\frac{d\Phi}{dt}(t_{\alpha,1/2}) = \dot{T}_\alpha \left( f_0 + f_1 \Delta T_\alpha + f_2 \Delta T_\alpha^2 + f_3 \Delta T_\alpha^3 + f_4 \Delta T_\alpha^4 + f_5 \Delta T_\alpha^5 \right).
$$

(29.16)

These constants, for each value of $\alpha$, require $\dot{T}_\alpha$ and $\Delta T_\alpha$, which are calculated by a suitable timing routine. For this demodulation package, this timing routine is provided by `tdb()`. Thus, for a given sky position, the timing routine will be called once for each short time chunk, each call returning a specific $\dot{T}_\alpha$ and $\Delta T_\alpha$. By substituting Eqs. 29.22, 29.23 and 29.24 in Eq. 29.21 and grouping together the terms in $j$ (linear in $t$) in order to save computations, we have

$$
\Phi_\alpha(t) = \sum_{s=0}^{n_{\text{spin}}} f_s A_{s\alpha} + \frac{j}{N} \sum_{s=0}^{n_{\text{spin}}} f_s B_{s\alpha},
$$

(29.17)

where $n_{\text{spin}}$ is the maximum order of spindown parameter. Rather than store the values of $\dot{T}_\alpha$ and $\Delta T_\alpha$ for each value of $\alpha$, it is more efficient to calculate the constants $A_{s\alpha}$ and $B_{s\alpha}$ only once, and then use these values for every spindown parameter set used when searching in a given sky position. Analytical formulae for these constants are easily derived:

$$
A_{s\alpha} = \frac{1}{s+1} \Delta T_{\alpha}^{s+1} - \frac{1}{2} T_{\text{SFT}} \dot{T}_\alpha \Delta T_\alpha^s
$$

(29.18)

$$
B_{s\alpha} = T_{\text{SFT}} \dot{T}_\alpha \Delta T_\alpha^s
$$

(29.19)
Error conditions

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>1</td>
<td>&quot;Null Pointer&quot;</td>
</tr>
<tr>
<td>NNUL</td>
<td>2</td>
<td>&quot;Non-Null Pointer&quot;</td>
</tr>
<tr>
<td>NEGA</td>
<td>4</td>
<td>&quot;Bad Negative Value&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants COMPUTESKYH_E<name>, and the status descriptions in COMPUTESKYH_MSGE<name>. The source code with these messages is in ComputeSky.h on line 1.133.

Structures

struct CParams

This structure contains the parameters for the ComputeSky() routine. The parameters are:

INT8 spinDwnOrder The maximal number of spindown parameters per spindown parameter set.

INT8 mObsSFT The number of SFTs in the observation time.

REAL8 tSFT The timescale of one SFT.

LIGOTimeGPS *tGPS An array containing the GPS times of the first datum from each SFT.

REAL8 *skyPos The array containing the sky patch coordinates.

CHAR *sw A switch which turns modulation on/off.

void (*funcName)(REAL8 , REAL8 , REAL8 , REAL8 *, REAL8 *, const CHAR *sw) A function pointer, to make the use of different timing routines easy.
Author: Berukoff, S.J., Papa, M.A.
$Id: ComputeSky.h,v 1.10 2007/06/08 14:41:50 bema Exp $
29.2.1 Module ComputeSky.c

Computes the phase model coefficients necessary for a successful demodulation.

Prototypes

```c
void LALComputeSky (LALStatus *status, REAL8 *skyConst, INT8 iSkyCoh, CSParams *params)
```

Description

Given an input index which refers to the sky patch under consideration, this routine returns the phase model coefficients $A_s$ and $B_s$ which are needed to correctly account for the phase variance of a signal over time. The CSParams parameter structure contains relevant information for this routine to properly run. In particular, it contains an array of timestamps in LIGOTimeGPS format, which are the GPS times of the first data from each SFT. The input is an INT4 variable iSkyCoh, which is the index of the sky patch under consideration. For each sky patch, this code needs to be run once; the necessary phase model coefficients are calculated, and can then be applied to the relevant spindown parameter sets one is using in their search.

Algorithm

The routine uses a simplistic nested for-loop structure. The outer loop is over the number of SFTs in the observation timescale; this accounts for the temporal variability of the phase model coefficients. The inner loop is over the number of spindown parameters in one set. Inside the inner loop, the values are calculated using the analytical formulae given in the ComputeSky.h documentation.

Uses

Notes

The reference-time, at which the pulsar spin-parameters are defined, is taken to be the start-time *INTERPRETED* as an SSB time (i.e. no translation is done, the times are numerically equal!).

Author: Berukoff, S.J., Papa, M.A.

$Id: ComputeSky.c,v 1.12 2007/06/08 14:41:51 bema Exp$
29.3 Header 

Computes phase coefficients necessary for a correct demodulation for a source in a binary system.

Synopsis

```c
#include <lal/ComputeSkyBinary.h>
```

The methods employed here follow very closely those used within `ComputeSky()`. Note that at present this code simply corrects for the Doppler modulation present in a polynomial frequency function for signals from sources in elliptical orbits. It does not account for general relativistic effects.

At the risk of repeating existing documentation, but in the interests of clarity much of the following can also be found in the `ComputeSky()` documentation. Recall that a demodulated Fourier Transform (DeFT) is given by

\[ \hat{x}_b(\tilde{\lambda}) = \sum_{\alpha=0}^{M-1} \sum_{k=0}^{N-1} \tilde{x}_{\alpha k} e^{-2\pi i (\Phi_{\alpha j}(\tilde{\lambda}) - \frac{\pi}{2})} \]  

(29.20)

The index \( b \) defines the DeFT frequency bin, the index \( \alpha \) loops through the SFTs that build the DeFT, \( k \) runs on all the SFT frequency bins, and \( j \) is a time index that runs on each SFT. As shown in section 29.1 the next step in the development of the demodulation technique involves Taylor expanding the phase model about the temporal midpoint of each short segment of data, while retaining only first order terms. At this point it is necessary to clearly define some quantities. Times as defined at the chosen detector are denoted about the temporal midpoint of each short segment of data, while retaining only first order terms. At this point it is necessary to clearly define some quantities. Times as defined at the chosen detector are denoted by \( t \), times defined at the solar system barycenter (SSB) are denoted by \( T \), and the retarded time measured at an inertial reference point (chosen as the SSB) at a distance from the source are denote by \( t' \).

The Taylor expansion of \( \Phi(t) \) about the temporal midpoint \( t_{\alpha,1/2} \) is

\[ \Phi_\alpha(t) = \Phi(t_{\alpha,1/2}) + [t - t_{\alpha,1/2}] \frac{d\Phi}{dt}(t_{\alpha,1/2}) \]  

(29.21)

For each value of \( \alpha \), this expression consists of either constant or linear terms in time. With the particular time discretization chosen in this code, \( t = t_0 + (N\alpha + j) T_{obs}/NM \), we have

\[ [t - t_{\alpha,1/2}] = \frac{T_{obs}}{M} \left( \frac{j}{N} - \frac{1}{2} \right) = T_s \left( \frac{j}{N} - \frac{1}{2} \right), \]  

(29.22)

where \( T_s \) is the short time baseline of the \( M \) short FTs. On the other hand, the phase can also be expressed as a function of SSB time \( T \) (i.e. the time at the solar system barycenter). We will assume the source to be at rest in this reference frame. If we now adopt the notation

\[ \Delta t'_\alpha \equiv [t'(T(t_{\alpha,1/2}))/T(t_0))] \]  

and \( t'_\alpha \equiv dt'/dt(\alpha,1/2) \), the phase terms described in Eq (29.21) become (neglecting constants)

\[ \Phi(t_{\alpha,1/2}) = f_0(\Delta t'_\alpha) + \frac{1}{2} f_1(\Delta t'_\alpha)^2 + \frac{1}{3} f_2(\Delta t'_\alpha)^3 + \frac{1}{4} f_3(\Delta t'_\alpha)^4 + \frac{1}{5} f_4(\Delta t'_\alpha)^5 + \frac{1}{6} f_5(\Delta t'_\alpha)^6, \]  

(29.23)

\[ \frac{d\Phi}{dt}(t_{\alpha,1/2}) = t'_\alpha \left( f_0 + f_1(\Delta t'_\alpha) + f_2(\Delta t'_\alpha)^2 + f_3(\Delta t'_\alpha)^3 + f_4(\Delta t'_\alpha)^4 + f_5(\Delta t'_\alpha)^5 \right). \]  

(29.24)

Note that the polynomial phase function is expressed as a function of the retarded time \( t' \) and subsequently the intrinsic frequency and it’s derivitives \( (f_i) \) are defined at the chosen inertial reference frame (SSB).

In order to calculate, for each value of \( \alpha \), the quantities \( t'_\alpha \) and \( \Delta t'_\alpha \), we must now look at the binary system in more detail. At present we reference section [13.9] where the definition of all the following orbital variables can be found. For a given set of orbital input parameters we obtain the eccentric anomaly \( E \) by numerically solving

\[ T(t_\alpha) - T_p = \frac{P}{2\pi} \left[ E + (p \sin E + q (\cos E - 1))] \right). \]  

(29.25)

where the quantities \( p \) and \( q \), dependent only on the orbital parameters of the source system, are given by
\[
p = \frac{2\pi \, a \sin i \sqrt{1 - e^2}}{P_c} \cos \omega - e
\]
\[
q = \frac{2\pi \, a}{P_c} \sin i \sin \omega.
\] (29.26)

\[T(t_\alpha)\] is returned via a call to \texttt{LALBarycenter} and \(a \sin i, P, T_p, \omega, e\) are the projected semi-major axis (projected along the line of sight), the orbital period, the time of observed periapse passage as measured in the SSB, the argument of periapse, and the orbital eccentricity respectively. Having defined \(E\) (where the source is in it’s orbit) at a given detector time \(t_\alpha\) we can calculate the derivative of the retarded source time \(n\) with respect to the SSB time \(T\). This is given by

\[
\frac{dt'}{dT} = \frac{[1 - e \cos E]}{[1 + p \cos E - q \sin E]}.
\] (29.27)

The quantity \(\dot{t}_\alpha\) can now be expressed as

\[
\dot{t}_\alpha = \frac{dT}{dt} \frac{dt'}{dT},
\] (29.28)

where \(dT/dt\) is returned via a call to \texttt{LALBarycenter()}

We can now rewrite Eq. 29.21 and by grouping together the terms in \(j\) (linear in \(t\)) in order to save computations, we have

\[
\Phi_\alpha(t) = \sum_{s=0}^{n_{\text{spin}}} f_s A_{s,\alpha} + \frac{j}{N} \sum_{s=0}^{n_{\text{spin}}} f_s B_{s,\alpha},
\] (29.29)

where \(n_{\text{spin}}\) is the maximum order of spindown parameter.

Thus, for a given sky position and set of orbital parameters, the quantities \(\dot{t}_\alpha\) and \(\Delta t'_\alpha\) are calculated only once, just as in \texttt{ComputeSky()}. The analytical constants defined in Eq 29.29 now become

\[
A_{s,\alpha} = \frac{1}{s + 1} \Delta (t'_\alpha)^{s+1} - \frac{1}{2} T_{\text{SFT}} \dot{t}_\alpha \Delta (t'_\alpha)^s
\] (29.30)

\[
B_{s,\alpha} = T_{\text{SFT}} \dot{t}_\alpha \Delta (t'_\alpha)^s.
\] (29.31)

It is these constants that form the input to the function \texttt{LALDemod()}. 

Error conditions

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>1</td>
<td>&quot;Null Pointer&quot;</td>
</tr>
<tr>
<td>NNULL</td>
<td>2</td>
<td>&quot;Non-Null Pointer&quot;</td>
</tr>
<tr>
<td>RANG</td>
<td>3</td>
<td>&quot;Input parameter out of range&quot;</td>
</tr>
<tr>
<td>NEGA</td>
<td>4</td>
<td>&quot;Bad Negative Value&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants \texttt{COMPUTESKYBINARHY_E<name>}, and the status descriptions in \texttt{COMPUTESKYBINARHY_MSGE<name>}. The source code with these messages is in \texttt{ComputeSkyBinary.h} on line 1.190.

Structures

\texttt{struct CSBParams}

This structure contains the parameters for the \texttt{ComputeSkyBinary()} routine. The parameters are:

\texttt{INT8 spinDwnOrder} The maximal number of spindown parameters per spindown parameter set.

\texttt{INT8 mObsSFT} The number of SFTs in the observation time.

\texttt{REAL8 tSFT} The timescale of one SFT.

\texttt{LIGOTimeGPS *tGPS} An array containing the GPS times of the first datum from each SFT.
REAL8 *skyPos The array containing the sky patch coordinates.

REAL8 SemimajorAxis The projected speed-of-light-normalised semi-major axis of the orbit (in seconds).

REAL8 OrbitalPeriod The orbital period (in seconds).

REAL8 ArgPeriapse The argument of the periapse (in radians).

REAL8 OrbitalEccentricity The orbital eccentricity.

LIGOTimeGPS TperiapseSSB A time of observed periapse passage as defined in the SSB.
Author: Messenger, C.J., Berukoff, S.J., Papa, M.A.
$Id: ComputeSkyBinary.h,v 1.5 2007/06/08 14:41:50 bema Exp $
29.3.1 Module ComputeSkyBinary.c

Computes the phase model coefficients necessary for a successful demodulation for the case of a continuous wave source in a binary system.

Prototypes

```c
void LALComputeSkyBinary (LALStatus *status,
                         REAL8 *skyConst,
                         INT8 iskyCoh,
                         CSBParams *params)
```

Description

Given a set of input parameters defining a source location in the sky and the binary system in which the source resides, this routine returns the phase model coefficients $A_{s\alpha}$ and $B_{s\alpha}$ which are needed to correctly account for the phase variance of a signal over time. The `CSBParams` parameter structure contains relevant information for this routine to properly run. In particular, it contains an array of timestamps in LIGOTimeGPS format, which are the GPS times of the first data from each SFT. The input is an `INT4` variable `iskeyCoh`, which is the index of the sky location under consideration. For each sky location and set of orbital parameters, this code needs to be run once; the necessary phase model coefficients are calculated, and can then be applied to the relevant spindown parameter sets one is using in their search.

Algorithm

The routine uses a simplistic nested for-loop structure. The outer loop is over the number of SFTs in the observation timescale; this accounts for the temporal variability of the phase model coefficients. The inner loop is over the number of spindown parameters in one set. Inside the inner loop, the values are calculated using the analytical formulae given in the `ComputeSkyBinary.h` documentation.

Uses

- `LALBarycenter()`
- `LALBarycenterEarth()`
- `LALDBracketRoot()`
- `LALDBisectionFindRoot()`

Notes

Author: Messenger, C.J., Berukoff, S.J., Papa, M.A.

$Id: ComputeSkyBinary.c,v 1.8 2007/06/08 14:41:51 bema Exp$
29.4 Header LALComputeAM.h

Computes filter components for amplitude demodulation.

Synopsis

```c
#include <lal/LALComputeAM.h>
```

In order to compute the optimal statistic for pulsar searches, one must take account of the various modulations that change the emitted, (fairly) simple sinusoid into a non-trivial function of parameters. The frequency evolution of the signal (spindown effects, Doppler modulation, etc.) have already been accounted for; this routine filters the amplitude modulation effects.

Error conditions

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOTS</td>
<td>1</td>
<td>&quot;Input LIGOTimeGPS Vector is wrong size or NULL&quot;</td>
</tr>
<tr>
<td>BCERR</td>
<td>2</td>
<td>&quot;Baryinput pointer is invalid&quot;</td>
</tr>
<tr>
<td>ESERR</td>
<td>3</td>
<td>&quot;EarthState structure invalid, or pointer NULL&quot;</td>
</tr>
<tr>
<td>EPH</td>
<td>4</td>
<td>&quot;Ephemeris Table invalid, or pointer NULL&quot;</td>
</tr>
<tr>
<td>DAS</td>
<td>5</td>
<td>&quot;Detector and source information invalid, or pointer NULL&quot;</td>
</tr>
<tr>
<td>FRD</td>
<td>6</td>
<td>&quot;Detector geometry information invalid, or pointer NULL&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants `LALCOMPUTEAMH_E<name>`, and the status descriptions in `LALCOMPUTEAMH_MSGE<name>`. The source code with these messages is in `LALComputeAM.h` on line 1.75.

Structures

**struct AMCoeffs**

This structure contains the output of the routine: \(a(t), b(t)\), and the scalar products therein. That is:

- `REAL4Vector *a` The function \(a(t)\)
- `REAL4Vector *b` The function \(b(t)\)
- `REAL4 A` The scalar product \((a||a)\)
- `REAL4 B` The scalar product \((b||b)\)
- `REAL4 C` The scalar product \((a||b)\)
- `REAL4 D` The quantity \(AB - C^2\)

**struct CmplxAMCoeffs**

This structure contains the AM coefficients \(a\) and \(b\) in the case of a complex detector tensor, and some relevant scalar products. That is:

- `COMPLEX8Vector *a` The \(a\) coefficient evaluated at the relevant times
- `COMPLEX8Vector *b` The \(b\) coefficient evaluated at the relevant times
- `REAL4 A` The scalar product \((a||a)\)
- `REAL4 B` The scalar product \((b||b)\)
- `REAL4 C` The scalar product \((a||b)\)
- `REAL4 E` The scalar product \((a||ib)\)
- `REAL4 D` The quantity \(AB - C^2 - E^2\)
struct AMCoeffsParams

This structure contains the parameters for the routine. They include:

BarycenterInput *baryinput  Parameters from LALBarycenter()
EarthState *earth  The state of the earth at time t
EphemerisDate *edat  Pointer to the ephemerides
LALDetAndSource *das  Detector and source information
LALFrDetector  Detector geometry information
REAL4 polAngle  Polarization angle
LALLeapSecAccuracy leapAcc  Leap sec accuracy
Author: Berukoff, S.J.
$Id: LALComputeAM.h,v 1.8 2007/12/19 19:44:29 reinhard Exp $
29.4.1 Module LALComputeAM.c

Computes quantities for amplitude demodulation.

Prototypes

```c
void LALComputeAM (LALStatus *status, AMCoeffs *coe, LIGOTimeGPS *ts, AMCoeffsParams *params)
```

Description

This routine computes the quantities \( a(t) \) and \( b(t) \) as defined in Jaranowski, Krolak, and Schutz (gr-qc/9804014), hereafter JKS. These functions quantify the dependence of the detector output on the beam-pattern functions \( F_+ \) and \( F_\times \); in fact, \( a(t) \) and \( b(t) \) are the beam-pattern functions, without the dependence on polarization angle and detector arm angle. Since the \texttt{LALDemod()} suite is an attempt to compute an optimal statistic, it is necessary to include these quantities in the computation. Otherwise, the motion of the Earth as it revolves about its axis will smear the signal into several neighboring bins centered about the search frequency, consequently losing valuable SNR.

Algorithm

The routine is really simple. From JKS,

\[
F_+ = \sin \zeta [a(t) \cos 2\psi + b(t) \sin 2\psi] \\
F_\times = \sin \zeta [b(t) \cos 2\psi - a(t) \sin 2\psi]
\]  
(29.32)
(29.33)

We use the routine \texttt{LALComputeDetAMResponse()} to calculate \( F_+ \) and \( F_\times \) for a given polarization angle, and then extract \( a(t) \) and \( b(t) \), once for each timestamp \( t \). Additionally, computation of the optimal statistic requires that we compute inner products of these two quantities for later use. See ?? for more on the optimal statistic.

Uses

- \texttt{LALBarycenter()}
- \texttt{LALBarycenterEarth()}
- \texttt{LALComputeDetAMResponse()}

Notes
References


Chapter 30

Package pulsar: known pulsar time-domain search routines

This package provides routines for a time domain search of gravitational wave signals from known pulsars. The documentation and functionality of this package is incomplete.
30.1 Header **HeterodynePulsar.h**

Provides routines to heterodyne, average, and resample the data as required for time domain known pulsar search.

**Synopsis**

```cpp
#include <lal/HeterodynePulsar.h>
```

The gravitational wave signal from a non-precessing pulsar at twice its frequency can be modeled as

\[ h(t) = F_+(t; \psi)h_0(1 + \cos^2 \iota) \cos 2\Psi(t) + 2F_\times h_0 \cos \iota \sin 2\Psi(t) \]  \hspace{1cm} (30.1)

where \( F_+ \) and \( F_\times \) are the amplitude responses of the detectors, \( \psi \) is the polarization angle, \( \iota \) describes the inclination of the pulsar with respect to the line of sight, and \( \Psi(t) = \phi_0 + \phi(t) \) describes the phase of the pulsar.

The phase \( \Psi(t) \) of the pulsar is calculated as

\[ \Psi(t) = \phi_0 + 2\pi \left( f_0(T - T_0) + \frac{1}{2} f_0(T - T_0)^2 + \frac{1}{6} f_0(T - T_0)^3 \right) \]  \hspace{1cm} (30.2)

where \( T = t + \delta t = t + \frac{\vec{r} \cdot \vec{n}}{c} + \Delta E_\odot \) \hspace{1cm} (30.3)

where \( T \) is the time in a frame inertial with respect to the pulsar and \( \phi_0 \) is the phase of at time \( T_0 \). The time difference \( \delta t \) due to the motion of the earth in the solar system is calculated using \textit{LALBarycenter()}.

The function \texttt{LALCoarseHeterodyne()} heterodynes, averages, and resamples the data at a fixed frequency near the signal.

Let the calibrated data from the interferometer be \( d(t) = h(t) + n(t) \) where \( n(t) \) is the noise. The first step is to mix the time series with \( e^{-2\pi if_h} \) where \( f_h \) is a fixed frequency near the signal.

\[ V_h(t) = d(t)e^{-2\pi if_h} \]  \hspace{1cm} (30.4)

The function \texttt{LALFineHeterodyneToPulsar()} applies a second heterodyne to the data which removes the spindown and the Doppler shifts.

After applying \texttt{LALCoarseHeterodyne()} and \texttt{LALFineHeterodyneToPulsar()} to the data we have

\[ d(t)' = n(t)' + F_+(t; \psi)h_0(1 + \cos^2 \iota)e^{i2\phi_0} + 2F_\times h_0 \cos \iota e^{i2\phi_0}. \]  \hspace{1cm} (30.5)

More documentation soon.

```cpp
void LALCoarseHeterodyne( LALStatus *status, CoarseHeterodyneOutput *output, CoarseHeterodyneInput *input, CoarseHeterodyneParams *params );

void LALFineHeterodyneToPulsar( LALStatus *status, FineHeterodyneOutput *output, FineHeterodyneInput *input, FineHeterodyneParams *params );
```
### Error conditions

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULLINPUT</td>
<td>1</td>
<td>&quot;Input was Null&quot;</td>
</tr>
<tr>
<td>NULLOUTPUT</td>
<td>2</td>
<td>&quot;Output was Null&quot;</td>
</tr>
<tr>
<td>NULLPARAMS</td>
<td>3</td>
<td>&quot;Params was Null&quot;</td>
</tr>
<tr>
<td>RFACTOR</td>
<td>4</td>
<td>&quot;The decimation factor supplied was invalid&quot;</td>
</tr>
<tr>
<td>INVALIDF0</td>
<td>5</td>
<td>&quot;Invalid input f0&quot;</td>
</tr>
<tr>
<td>LENGTH</td>
<td>6</td>
<td>&quot;Input vectors were not the same length&quot;</td>
</tr>
<tr>
<td>BINARY</td>
<td>7</td>
<td>&quot;Binary model not yet implemented&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants HETERODYNEPULSARH_E<name>, and the status descriptions in HETERODYNEPULSARH_MSGE<name>. The source code with these messages is in HeterodynePulsar.h on line 1.246.

### Types

**Structure CoarseHeterodyneInput**

This structure stores the original calibrated gw data.

- REAL4TimeSeries V calibrated strain data from detector
- REAL4 f0 heterodyning base frequency

**Structure CoarseHeterodyneOutput**

This structure stores the output of the heterodyned data.

- COMPLEX8TimeSeries Vh heterodyned data
- COMPLEX16 varh variance of Vh
- REAL4 phase phase of the reference signal f0 at t0
- COMPLEX16 avg average of Vh
- COMPLEX16 kurt kurtosis of Vh
- COMPLEX16 skew skewness of Vh
- COMPLEX16 covar first term of covariance of Vh

**Structure CoarseHeterodyneParams**

This structure stores parameters for the coarse heterodyne.

- UINT4 boxM first decimation factor (and order of boxcar)
- REAL4IIRFilter *iirFilter1Re first IIR filter to be applied to real part of complex heterodyned data
- REAL4IIRFilter *iirFilter1Im first IIR filter to be applied to imaginary part of complex heterodyned data
- UINT4 iirM second decimation factor
- REAL4IIRFilter *iirFilter2Re second IIR filter to be applied to real part of complex heterodyned data
- REAL4IIRFilter *iirFilter2Im second IIR filter to be applied to imaginary part of complex heterodyned data
- UINT4 stats set to 1 to calculate only Vh and variance; 2 for Vh, var, kurt, skew, covar; 0 else
Structure **FineHeterodyneInput**

This structure stores the input for the fine heterodyne.

- **COMPLEX8TimeSeries Vh** heterodyned, averaged and resampled data
- **COMPLEX8TimeSeries varh** variance of corresponding Vh
- **REAL4 f0** frequency of the signal
- **REAL4 f1** first time derivative of frequency
- **REAL4 f2** second time derivative of frequency
- **REAL8 fEpochGPS** epoch of the frequency

**SkyPosition source** location of pulsar in sky - equatorial coordinate system
- **REAL4 pmRA** proper motion RA (radians / year)
- **REAL4 pmDEC** proper motion DEC (radians / year)
- **REAL8 posEpochGPS** epoch of RA and DEC
- **UINT4 model** 0 for isolated pulsar, 1 for binary
- **REAL8 e** eccentricity of orbit
- **REAL8 w**
- **REAL8 T0**
- **REAL8 Pb**
- **REAL8 x**
- **REAL8 lg**

Structure **FineHeterodyneOutput**

This structure stores the output of the fine heterodyne.

- **COMPLEX8TimeSeries B** bin value
- **COMPLEX8TimeSeries var** variance
- **REAL4 phase** phase

Structure **FineHeterodyneParams**

This structure stores the params of the fine heterodyne.

- **EphemerisData *edat**
- **LALDetector detector**
- **REAL4IIRFilter *iirFilterRe** IIR filter to be applied to real part of complex heterodyned data
- **REAL4IIRFilter *iirFilterIm** IIR filter to be applied to imaginary part of complex heterodyned data
- **UINT4 M** decimation factor
- **UINT2 iirFlag** 1 to apply iir filter, 0 for no iir filter
30.1.1 Module **HeterodynePulsar.c**

The routines in this module heterodyne, average and resample a time series for known pulsars.

**Prototypes**

```c
void LALCoarseHeterodyne ( LALStatus *status,
                           CoarseHeterodyneOutput *output,
                           CoarseHeterodyneInput *input,
                           CoarseHeterodyneParams *params )
```

```c
void LALFineHeterodyneToPulsar ( LALStatus *status,
                                 FineHeterodyneOutput *output,
                                 FineHeterodyneInput *input,
                                 FineHeterodyneParams *params )
```

**Description**

The function `LALCoarseHeterodyne()` ...

The function `LALFineHeterodyneToPulsar()` ...

**Algorithm**

To be completed.

**Uses**

- LALSIIRFilter()
- LALCreateVector()
- LALDestroyVector()
- LALBarycenterEarth()
- LALBarycenter()

**Notes**

Author: Dupuis, R. J.

$Id: HeterodynePulsar.c,v 1.5 2007/06/08 14:41:51 bema Exp$
30.1.2 Program HeterodynePulsarTest.c

This program demonstrates the usage of the functions LALCoarseHeterodyne() and LALFineHeterodyneToPulsar().

Usage

HeterodynePulsarTest

Description

This test program heterodynes, averages, and resamples an artificial signal using the functions LALCoarseHeterodyne() and LALFineHeterodyneToPulsar().

Exit codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOM</td>
<td>0</td>
<td>&quot;Nominal exit&quot;</td>
</tr>
<tr>
<td>CHK</td>
<td>1</td>
<td>&quot;Error checking failed to catch bad data&quot;</td>
</tr>
<tr>
<td>FLS</td>
<td>2</td>
<td>&quot;Incorrect answer for valid data&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants HETERODYNEPULSARTESTC_E<name>, and the status descriptions in HETERODYNEPULSARTESTC_MSGE<name>. The source code with these messages is in HeterodynePulsarTest.c on line 1.84.

Uses

LALCoarseHeterodyne()
LALFineHeterodyneToPulsar()

Notes
References


# 30.2 Header `FitToPulsar.h`

Provides routines for finding the best fit of the measured data to the strain expected from non-precessing pulsar.

## Synopsis

```c
#include <lal/FitToPulsar.h>
```

The model to be fitted to the data after `LALFineHeterodyneToPulsar` has been applied is

\[
y(t; a) = F_+(t; \psi)h_0(1 + \cos^2 \iota)e^{2\phi_0} - 2iF_-(t; \psi)h_0 \cos \iota e^{2\phi_0}
\]  
(30.6)

The reduced set of data points is fitted to this model by minimizing \(\chi^2\) over \(h_0, \phi_0, \iota,\) and \(\psi\).

\[
\chi^2(a) = \sum_k \left| \frac{B_k - y(t; a)}{\sigma_k^2} \right|^2
\]  
(30.7)

The minimization of \(\chi^2\) is done in two steps `LALCoarseFitToPulsar()` and `LALFineFitToPulsar()`.

More documentation soon.

## Error conditions

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULLINPUT</td>
<td>1</td>
<td>&quot;Input was Null&quot;</td>
</tr>
<tr>
<td>NULLOUTPUT</td>
<td>2</td>
<td>&quot;Output was Null&quot;</td>
</tr>
<tr>
<td>NULLPARAMS</td>
<td>3</td>
<td>&quot;Params was Null&quot;</td>
</tr>
<tr>
<td>VECSIZE</td>
<td>4</td>
<td>&quot;Input vectors were not the same length&quot;</td>
</tr>
<tr>
<td>MESH</td>
<td>5</td>
<td>&quot;Mesh parameters supplied were invalid&quot;</td>
</tr>
<tr>
<td>VAR</td>
<td>6</td>
<td>&quot;Variance vector in Input had invalid values&quot;</td>
</tr>
<tr>
<td>MAXCHI</td>
<td>7</td>
<td>&quot;The minimum value of chiSquare was greater than INICHISQU&quot;</td>
</tr>
<tr>
<td>DIVZERO</td>
<td>8</td>
<td>&quot;Attempted to divide by zero&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants `FITTOPULSARH_EX<name>`, and the status descriptions in `FITTOPULSARH_MSGE<name>`. The source code with these messages is in `FitToPulsar.h` on line 1.143.

## Structures

### Structure `CoarseFitInput`

This structure stores locked data to be fitted by model.

- `COMPLEX16Vector *B` heterodyned, averaged and resampled data
- `COMPLEX16Vector *var` variance of the rFactor points that were averaged
- `LIGOTimeGPS *t` time stamp for each data point (not necessarily with equal time steps)

### Structure `CoarseFitOutput`

This structure stores the results from the coarse fit of parameters.

- `REAL8 h0` best fit h0
- `REAL8 eh0[3]` standard error for h0, min standard error, max standard error
- `REAL8 cosIota` best fit cosIota
- `REAL8 phase` best fit phase
- `REAL8 psi` best fit psi (polarization angle)
- `REAL8 chiSquare` min value of chi square
- `REAL8Vector *mChiSquare` matrix with chi square values
Structure `CoarseFitParams`

This structure stores the parameters for the coarse fit.

- `REAL8 meshH0[3]` min h0, delta h0, number of steps
- `REAL8 meshCosIota[3]` min cosIota, delta cosIota, number of steps
- `REAL8 meshPhase[3]` min phase, delta phase, number of steps
- `REAL8 meshPsi[3]` min psi, delta psi, number of steps

`LALSource pulsarSrc` describes sky position of pulsar

`LALDetector detector` detector
30.2.1 Module **FitToPulsar.c**

Calculates the best fit parameters for a GW signal originating from a non-precessing pulsar.

**Prototypes**

```c
void LALCoarseFitToPulsar ( LALStatus *status,
                          CoarseFitOutput *output,
                          CoarseFitInput *input,
                          CoarseFitParams *params )

void LALFineFitToPulsar ( LALStatus *status,
                          FineFitOutput *output,
                          FineFitInput *input,
                          FineFitParams *params )
```

**Description**

This routine calculates the best fit of parameters by minimizing $\chi^2$ by going through fixed grid for $\iota, \psi, \phi_0$ and $h_0$. The best fit parameters returned by `LALCoarseFitToPulsar()` are then used as initial parameters for `LALFineFitToPulsar()`.

The function `LALFineFitToPulsar()` refines the fit using the Levenberg-Marquardt method for nonlinear fitting. This is done by calculating the Hessian and the gradient of $\chi^2$ ...

**Algorithm**

To be completed.

**Uses**

`LALSCreateVector()`
`LALSDestroyVector()`
`LALComputeDetAMResponse()`

**Notes**
30.2.2 Module **FitToPulsar.c**

Calculates the best fit parameters for a GW signal originating from a non-precessing pulsar.

**Prototypes**

```c
void LALCoarseFitToPulsar ( LALStatus *status,
                           CoarseFitOutput *output,
                           CoarseFitInput  *input,
                           CoarseFitParams  *params )

void LALFineFitToPulsar ( LALStatus *status,
                           FineFitOutput   *output,
                           FineFitInput    *input,
                           FineFitParams   *params )
```

**Description**

This routine calculates the best fit of parameters by minimizing $\chi^2$ by going through fixed grid for $\iota, \psi, \phi_0$ and $h_0$. The best fit parameters returned by `LALCoarseFitToPulsar()` are then used as initial parameters for `LALFineFitToPulsar()`.

The function `LALFineFitToPulsar()` refines the fit using the Levenberg-Marquardt method for nonlinear fitting. This is done by calculating the Hessian and the gradient of $\chi^2$ ...

**Algorithm**

To be completed.

**Uses**

- `LALSCreateVector()`
- `LALSDestroyVector()`
- `LALComputeDetAMResponse()`

**Notes**

**Program **FitToPulsarTest.c**

This test program demonstrates the correct usage of the functions `LALCoarseFitToPulsar` and `LALFineFitToPulsar`.

**Usage**

`FitToPulsarTest`

**Description**

To be completed.
Exit codes

<table>
<thead>
<tr>
<th>name</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOM</td>
<td>0</td>
<td>&quot;Nominal exit&quot;</td>
</tr>
<tr>
<td>CHK</td>
<td>1</td>
<td>&quot;Error checking failed to catch bad data&quot;</td>
</tr>
<tr>
<td>FLS</td>
<td>2</td>
<td>&quot;Incorrect answer for valid data&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants FITTOPULSARTESTC_E<name>, and the status descriptions in FITTOPULSARTESTC_MSGE<name>. The source code with these messages is in FitToPulsarTest.c on line 1.355.

Uses

LALCoarseFitToPulsar()
LALFineFitToPulsar()

Notes
30.3 Header PulsarCat.h

Provides structures and routines to store and manipulate pulsar properties.

Synopsis

\#include <lal/PulsarCat.h>

This header covers structures to store pulsar properties in a standard format, and routines to manipulate and update these properties. The set of properties stored in the catalogue is based on radio pulsar catalogues, with some additions and subtractions specific to gravitational wave observations. The list of properties can be expanded in future by adding more fields to the structure.

All properties are those that would be measured by an observer at the solar system barycentre. For properties that depend on time (e.g., time-varying position, periods, etc.), an epoch is specified. The properties are then those that would be observed at the specified instant of time at the solar system barycentre; i.e. when the wave fronts carrying that information pass the solar system barycentre.

A note on companion orbits: Several known pulsars exist in multiple systems, and radio-pulsar catalogues include detailed models of the companion orbits, as determined from the pulsar timing. See GenerateSpinOrbitCW.h in the inject package for a discussion of the parameters defining the orientation of a companion orbit.

Radio-pulsar observations rarely determine the inclination $i$ of the orbit to the sky plane, and thus cannot resolve the longitude of the ascending node $\Omega$ and the argument of the periapsis $\omega$ as independent parameters. Instead, they list the longitude of the periapsis $w$, which is the angle in the plane of the sky from the North direction towards the West direction, to the ray from the system barycentre to the periapsis projected onto the plane of the sky. If any three of $i$, $\Omega$, $\omega$, and $w$ are known, the fourth can be determined from the relation:

$$w - \Omega = \arctan 2(\sin \omega \cos i, \cos \omega),$$

or equivalently:

$$\omega = \arctan 2(\cos[w - \Omega], \sin[w - \Omega]/\cos i).$$

In addition to these Keplerian orbital parameters, some radio-pulsar systems have measured post-Keplerian relativistic orbital parameters. Some of these are obvious: $\dot{w}$ and $\dot{P}$ are the rate of change in $w$ (periapsis precession) and the orbital period $P$ (due to gravitational radiation reaction). The catalogue also lists post-Keplerian parameters “sin” and “r”, whose meanings I don’t know.

Error conditions

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUL</td>
<td>1</td>
<td>&quot;Unexpected null pointer in arguments&quot;</td>
</tr>
<tr>
<td>OUT</td>
<td>2</td>
<td>&quot;Output handle points to a non-null pointer&quot;</td>
</tr>
<tr>
<td>MEM</td>
<td>3</td>
<td>&quot;Memory allocation error&quot;</td>
</tr>
<tr>
<td>PARSE</td>
<td>4</td>
<td>&quot;Error parsing input file&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants PULSARCATH_E<name>, and the status descriptions in PULSARCATH_MSGE<name>. The source code with these messages is in PulsarCat.h on line 1.115.

Types

Structure CompanionNode

This structure stores the orbital parameters of a companion to a pulsar in a multiple system. If there is more than one companion, these structures form a linked list.

LIGOTimeGPS epoch Epoch of companion periapsis.

REAL8 x Projected orbital semimajor axis $(a/c)\sin i$, in seconds.

REAL8 period Orbital period, in seconds, measured at epoch.
REAL8 periodDot First time derivative of orbital period (dimensionless).
REAL8 omega Longitude of periapsis, in radians, measured at epoch.
REAL8 omegaDot Rate of advance of periapsis, in radians/s.
REAL8 ecc Orbital eccentricity.
REAL8 gamma Post-Keplerian “gamma” term, in seconds.
REAL8 sin Post-Keplerian “s” term.
REAL8 r Post-Keplerian “r” term.

CompanionNode *next Pointer to next companion’s data; NULL if there are no further companions in the system.

Structure PulsarCatNode

This structure represents a single node in a linked list of pulsar data, storing data for a single pulsar. The fields are:
CHAR bname[10] The B1950 pulsar name (e.g. B0021-72C), terminated by a ‘\0’ character.
CHAR jname[12] The J2000 pulsar name (e.g. J0024-7203U), terminated by a ‘\0’ character.
SkyPosition dpos Uncertainty in pos, in radians.
SkyPosition pm The pulsar proper motion, in radians per second.
SkyPosition dpm Uncertainty in pm, in radians per second.
LIGOTimeGPS posepoch The epoch of the postion measurement.
REAL8Vector *f The pulsar spin frequency \( f->data[0] \), and its time derivatives \( f->data[1] \ldots f->data[k] \ldots \), in units of Hz\( k+1 \).
REAL8Vector *df The uncertainty in the frequency and its time derivatives, in the same units.
LIGOTimeGPS fepoch The epoch of the spin and phase measurements.
REAL4 dist Distance to pulsar, in m. If negative, only a lower or upper limit has been established.
REAL4 dmin Lower-limit distance to pulsar, in m. If negative, no lower limit has been specified.
REAL4 dmax Upper-limit distance to pulsar, in m. If negative, no upper limit has been specified.
CHAR lcode Reliability of distance measurement on low side, from ‘a’ (best) to ‘d’ (worst).
CHAR ucode Reliability of distance measurement on high side, from ‘a’ (best) to ‘d’ (worst).
CompanionNode *companion Pointer to head of linked list of orbital parameters for other components of a multiple system; NULL if the pulsar has no known companion. See below for the contents of these data nodes.
UINT2 typecode Binary code for additional pulsar properties. The typecode is the logical “or” (i.e. the numerical sum) of the following property codes:
1 Globular cluster association
2 Supernova remnant association
4 Glitches in period
8 Binary or multiple pulsar
16 Millisecond pulsar
32 Recycled pulsar
64 Radio interpulse
128 Optical, xray, or gamma-ray pulsed emission

PulsarCatNode *next Next pulsar in the catalogue’s linked list; NULL if this is the last (or only) pulsar in the list.
**Enumeration PulsarCatIndex**

This enumerated type is used to give a default ordering to the fields in the pulsar catalogue. This is used, for instance, when reading pulsar catalogue data from a file. The values are of the form `PULSARCATINDEX_⟨label⟩`, where the (currently) allowed values of ⟨label⟩ are:

<table>
<thead>
<tr>
<th>label</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>pulsar name</td>
</tr>
<tr>
<td>RAJ</td>
<td>J2000 right ascension</td>
</tr>
<tr>
<td>DECJ</td>
<td>J2000 declination</td>
</tr>
<tr>
<td>PMRA</td>
<td>right ascension proper motion</td>
</tr>
<tr>
<td>PMDEC</td>
<td>declination proper motion</td>
</tr>
<tr>
<td>POSEPOCH</td>
<td>position measurement epoch</td>
</tr>
<tr>
<td>F</td>
<td>spin frequency</td>
</tr>
<tr>
<td>F1</td>
<td>spin frequency derivative</td>
</tr>
<tr>
<td>F1</td>
<td>spin frequency second derivative</td>
</tr>
<tr>
<td>PEPOCH</td>
<td>spin measurement epoch</td>
</tr>
<tr>
<td>Dist</td>
<td>distance</td>
</tr>
<tr>
<td>NUM</td>
<td>number of enum values</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>label</th>
<th>uncertainty</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAJERR</td>
<td></td>
</tr>
<tr>
<td>DECJERR</td>
<td></td>
</tr>
<tr>
<td>PMRAERR</td>
<td></td>
</tr>
<tr>
<td>PMDECERR</td>
<td></td>
</tr>
<tr>
<td>FERR</td>
<td></td>
</tr>
<tr>
<td>F1ERR</td>
<td></td>
</tr>
<tr>
<td>F2ERR</td>
<td></td>
</tr>
</tbody>
</table>

Author: Creighton, T. D.

$Id: PulsarCat.h,v 1.6 2007/06/08 14:41:50 bema Exp $
30.3.1 Module **PulsarCat.c**

Manipulates a catalogue of pulsar data.

**Prototypes**

```c
void LALUpdatePulsarCatNode( LALStatus *stat,
PulsarCatNode *node,
LALPlaceAndGPS *detectorTime,
EphemerisData *edat )

void LALUpdatePulsarCat( LALStatus *stat,
PulsarCatNode *head,
LALPlaceAndGPS *detectorTime,
EphemerisData *edat )

void LALDestroyPulsarCat( LALStatus *stat,
PulsarCatNode **head )
```

**Description**

The routine `LALUpdatePulsarCatNode()` updates all time-varying properties of the pulsar system to a new epoch specified by the input `*time`. The interpretation of this input is specified below, and can involve the time-varying position of the Earth, as specified in `*edat`. Right ascension and declination are updated based on the specified proper motion, and the pulsar frequency and its derivatives are updated based on the higher-order derivatives. For companion objects, a new periapsis epoch is chosen that is as close as possible (within half an orbit) of the desired epoch, and all other time-dependent orbital parameters are updated to this epoch. All updates are done “in place”, to eliminate memory usage and computation that are, in most cases, unnecessary.

The routine `LALUpdatePulsarCat()` does the same thing as above, but to all nodes in the list pointed to by `head`.

The routine `LALDestroyPulsarCat()` iteratively frees all memory allocated to the list pointed to by `*head`, and then sets `*head` to NULL.

**Interpretation of `*time`:** The epoch of a catalogue update is specified by a `LALPlaceAndGPS` structure `*time`, which contains both a GPS time and a detector site. The interpretation is as follows: The desired properties are those properties of the system, as measured by an observer at the solar system barycentre, that the system had when it emitted the waves that arrive at the specified detector at the specified GPS time. By contrast, the properties listed in the catalogue are those properties of the system, as measured by an observer at the solar system barycentre, that the system had when it emitted the waves that arrive at the solar system barycentre at the GPS time given in the catalogue. Having specified (in the `*time` structure) the instant that the wave fronts reach the detector, `LALUpdatePulsarCatNode()` first computes the time when those waves pass the solar system barycentre, and uses that time as the new catalogue epoch. Thus, after calling `LALUpdatePulsarCatNode()`, the GPS times in `*node` will in general not be the same as the GPS time in `*time`, but will differ by a light propagation time.

If the `time->p_detector` field is NULL, then `time->p_gps` is assumed to be the time when the waves reach the solar system barycentre, and the complication described above does not arise. If the `time->p_gps` field is NULL, then the routine will return an error. If `edat` is NULL, then `time->p` must also be NULL, or an error is returned.

**Algorithm**

The function `LALUpdatePulsarCatNode()` first computes the correct epoch for the pulsar data, taking into account the difference between the detector time specified in `*time` and the barycentric time specified in `*node`: a propagation time delay id computed using `LALTimeDelayFromEarthCenter()`, `LALBarycenterEarth()`, and `LALBarycenter()`, with the pulsar position taken from `*node` and the Earth
ephemeris given in *edat*. This is done in a loop (since updating the epoch can conceivably change the pulsar location), until the correct epoch is determined to within 3\(\mu\)s, the precision of \texttt{LALBarycenter()}

Next, the pulsar location \(\vec{\lambda}\) is updated using its proper motions \(\dot{\vec{\lambda}}\), and an uncertainty is computed using linear error propagation (assuming uncorrelated position and proper motion errors):

\[
\lambda_i|_{t=t_2} = \lambda_i|_{t=t_1} + \dot{\lambda}_i(t_2 - t_1) ,
\]

\[
\sigma_{\lambda_i}|_{t=t_2} = \sqrt{[\sigma_{\lambda_i}|_{t=t_2}]^2 + [\sigma_{\dot{\lambda}_i}(t_2 - t_1)]^2} ,
\]

where \(t_1\) is the old position epoch and \(t_2\) is the new epoch. Similarly, the \(k\)th frequency derivative \(f^{(k)} = d^k f/dt^k\) is updated and its uncertainty computed using the following formulae:

\[
f^{(k)}|_{t=t_2} = \sum_{j=k}^N \frac{(t_2 - t_1)^{j-k}}{(j-k)!} f^{(j)}|_{t=t_1} ,
\]

\[
\sigma_{f^{(k)}}|_{t=t_2} = \sqrt{\sum_{j=k}^N \left[ \frac{(t_2 - t_1)^{j-k}}{(j-k)!} \sigma_{f^{(j)}}|_{t=t_1} \right]^2} ,
\]

where \(t_1\) is the old spin epoch and \(t_2\) is the new epoch. An additional spin uncertainty is assessed based on the assumption that there may be unmeasured higher-order derivatives of the spin. The inverse spin timescale 

\[
\tau^{-1} = \max_{k=1}^N \left\{ (f^{(k)}/f)^{1/k} \right\}
\]

is roughly the time that the spin frequency will change by an amount comparable to its initial value. Naïvely, the next higher frequency derivative will be of order 

\[
f^{(N+1)} \sim f^{(\tau^{-1})^{N+1}} ,
\]

and will introduce a further error in \(f^{(k)}\) equal to:

\[
\delta f^{(k)} \approx f \left[ \tau^{-1}(t_2 - t_1) \right]^{N+1} (t_2 - t_1)^{-k} .
\]

This uncertainty is added in quadrature to the other errors.

In both position and spin uncertainty calculations, the uncertainty in the time shift, \(\sigma_{\Delta t} \sim 3\mu\)s or \(\sim 10^{-15}\Delta t\) (whichever is larger), is assumed to be negligible. This is a good assumption as long as spin frequencies are much less than 300kHz and observation times will not be much greater than spindown timescales.

Finally, the properties of any companion orbits are updated using their first time derivatives, under the implicit assumption that higher-order derivatives have negligible effects over the interval of the update. In the catalogue, all orbital properties are referred to an epoch of periapsis passage \(t_0\). Keeping only first-order time derivative corrections to the period, the number of orbits at some later time \(t_0 + \Delta t\) is:

\[
n = \frac{\Delta t}{P_0} \left( 1 - \frac{\dot{P} t}{2P_0} \right) ,
\]

where \(P_0 = P(t = t_0)\). This number is rounded to an integer to get a periapsis passage near the desired time, and the epoch of this passage is determined by inverting the formula (and again expanding only to first order):

\[
t = nP_0 \left( 1 + \frac{1}{2} n \dot{P} \right) .
\]

Once the new epoch is determined, the period and longitude of periapsis will be updated using their first derivatives.

\textit{Note:} I am assuming that the orbital period given in the pulsar catalogue is from periapsis to periapsis. If it is not, then a \(\dot{w}\) correction will have to be included when computing the updated epoch.

\textbf{Uses}

\begin{itemize}
  \item \texttt{lalDebugLevel}
  \item \texttt{LALConvertSkyCoordinates()}
  \item \texttt{LALINT8toGPS()}
  \item \texttt{LALGPstoINT8()}
  \item \texttt{LALBarycenterEarth()}
  \item \texttt{LALBarycenter()}
  \item \texttt{LALDestroyVector()}
  \item \texttt{LALFree()}
\end{itemize}

\textbf{Notes}

\begin{itemize}
  \item \textit{Author:} Creighton, T. D.
\end{itemize}
30.3.2 Module PulsarCatInput.c

Parses a catalogue of pulsar data.

Prototypes

```c
void LALReadPulsarCatHead( LALStatus *stat,
                           INT4    indx[PULSARCATINDEX_NUM],
                           TokenList *list )

void LALReadPulsarCatLine( LALStatus    *stat,
                           PulsarCatNode  *node,
                           TokenList     *list,
                           INT4          indx[PULSARCATINDEX_NUM] )
```

Description

The routine `LALReadPulsarCatHead()` takes a set of tokens `*list` (as generated from an input line by `LALCreateTokenList()`, and determines which tokens correspond to the pulsar catalogue fields enumerated by `indx[]`: each element of `indx[]` stores the number of the corresponding token. Each token should be a field name corresponding to one of the enumeration constants in `PulsarCatIndex` (e.g. "RAJ", "POSEPOCH", "F"), or "e" to represent an uncertainty in the preceding field. Unrecognized tokens are ignored. If any pulsar catalogue field does not have a corresponding token in `indx[]` is set to $-1$.

The routine `LALReadPulsarCatLine()` takes the input `*line`, splits it into whitespace-separated tokens, and parses each token into the corresponding field of `*node`, using `indx[]` to determine which tokens correspond to which fields. In general, each field has a distinct parsing algorithm, as specified below:

- **NAME** A standard B1950 or J2000 pulsar name (e.g. "B0021-72C", "J0024-7203U"), copied directly into `node->bname` or `node->jname`.

- **RAJ** J2000 right ascencion in the form "hours:minutes:seconds", where hours is a signed integer, minutes an unsigned integer, and seconds is an unsigned floating-point number in normal place-index notation (i.e. integral part, optional decimal place, and optional fractional part; no exponential notation).

- **DECJ** J2000 declination in the form "degrees:minutes:seconds", where degrees is a signed integer, minutes an unsigned integer, and seconds is an unsigned floating-point number in normal place-index notation (i.e. integral part, optional decimal place, and optional fractional part; no exponential notation).

- **PMRA** Right ascension component of proper motion in milliarcseconds per year, as a floating-point number (any notation).

- **PMDEC** Declination component of proper motion in milliarcseconds per year, as a floating-point number (any notation).

- **POSEPOCH** Epoch of position/proper motion measurements in Julian days, as a floating-point number (any notation). If the number is less than 2 million, then it is assumed that the actual Julian day is 2 million plus the number given.

- **F** The pulsar spin frequency in Hz, as a floating-point number (any notation).

- **F1** The first derivative of the pulsar spin frequency in Hz$^2$, as a floating-point number (any notation).

- **F2** The pulsar spin frequency in Hz$^3$, as a floating-point number (any notation).

- **PEPOCH** Epoch of frequency and frequency-derivative measurements in Julian days, as a floating-point number (any notation). If the number is less than 2 million, then it is assumed that the actual Julian day is 2 million plus the number given.
Uncertainty in any of the preceding quantities. This is given as an unsigned integer corresponding to the uncertainty in the last 1 or 2 significant digits of the corresponding quantity. Thus, the parsing routine for that quantity is also responsible for reading its uncertainty, accounting for the number of significant digits.

An asterisk * in any field means that the quantity is not measured. In most cases this means it will be treated as zero.

Uses

- lalDebugLevel
- LALWarning()
- LALStringToU2()
- LALLeapSecs()
- LALINT8toGPS()
- LALDCreateVector()
- LALDDestroyVector()

Notes

At present, LALLeapSecs() fails for times prior to the GPS epoch (1980-01-06 00:00:00, or JD2444244.5000), which prevents earlier Julian dates from being converted into LIGOTimeGPS structures. Pulsar catalogue entries with epochs earlier than this date will cause LALReadPulsarCatLine() to fail.
30.3.3 Program PulsarCatTest.c

Tests routines to manipulate pulsar data.

Usage

PulsarCatTest [-p posepoch ra dec pmra pmdec] [-l site earthfile sunfile] [-h]
[-t newepoch] [-i infile] [-o outfile] [-d debuglevel]
[fepoch f0 [f1 ...]]

Description

This program reads in or randomly generates pulsar parameters, stores them in a pulsar catalogue structure, and manipulates them based on command-line arguments. The following option flags are accepted (in any order):

- **p** The pulsar position at time `posepoch` is set to `ra` radians right ascension and `dec` radians declination, with proper motions of `pmra` and `pmdec` radians per second, respectively. See below for parsing formats for `posepoch`. If the `-p` option is not specified, a random source position is generated.

- **l** Sets the detector location. `site` must be one of the following character strings: `LHO`, `LLO`, `VIRGO`, `GEO600`, `TAMA300`, or `CIT40`. `earthfile` and `sunfile` are ephemeris files of the format expected by `LALInitBarycenter()`. If the `-l` option is not specified, the detector is placed at the solar system barycentre.

- **h** Prints usage information and then exits.

- **t** Sets the new epoch to which the pulsar data will be updated. See below for parsing formats for `newepoch`. If the `-t` option is not given, `-t J2000.0` is assumed.

- **i** Reads the pulsar data from the file `infile`, whose format is described below. This overrides the position and spin information read from the command line. If the name `stdin` is given, it will read from standard input, not from a file named `stdin`.

- **o** Writes the pulsar data to the file `outfile`. If the name `stdout` or `stderr` is given, it will write to standard output or standard error (respectively), not to a file named `stdout` or `stderr`. If the `-o` option is not given, the routines are exercised, but no output is written.

- **d** Sets the debug level to `debuglevel`. If absent, level 0 is assumed.

Once all valid options are read, the remaining command-line arguments are taken to be the epoch `fepoch` at which the pulsar spin frequency `f` (in Hz) was measured, plus zero or more frequency derivatives `f1`... (in Hz\(^{k+1}\)) for the `k`th derivative. If no additional arguments are given, spin timing information will be omitted.

Measurement epoch `posepoch`, `newepoch`, and `fepoch` may be specified either as a REAL8 Julian epoch preceded by a `J` character (e.g. `JD2000.0`), a REAL8 number of Julian days preceded by `JD` (e.g. `JD2451545.0`), or as an INT8 number of GPS nanoseconds with no prefix (e.g. `630763213000000000`). Note that the preceding examples all refer to noon UTC, January 1, 2000. Also, note that each Julian epoch is assumed to be exactly 365.25 Julian days, so `J2001.0` corresponds to 18:00 UTC, January 1, 2001.

If an input file is specified, it should consist of a header line that, when tokenized, can be parsed by `LALReadPulsarCatHead()`, followed by one or more lines of pulsar data parseable (when tokenized) by `LALReadPulsarCatLine()`. Blank lines (with no tokens) or divider lines (with only one token) will be skipped.

Exit codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NORM</td>
<td>0</td>
<td>&quot;Normal exit&quot;</td>
</tr>
<tr>
<td>SUB</td>
<td>1</td>
<td>&quot;Subroutine failed&quot;</td>
</tr>
<tr>
<td>ARG</td>
<td>2</td>
<td>&quot;Error parsing arguments&quot;</td>
</tr>
<tr>
<td>VAL</td>
<td>3</td>
<td>&quot;Input argument out of valid range&quot;</td>
</tr>
<tr>
<td>MEM</td>
<td>4</td>
<td>&quot;Out of memory&quot;</td>
</tr>
<tr>
<td>FILE</td>
<td>5</td>
<td>&quot;Could not open file&quot;</td>
</tr>
</tbody>
</table>
The status codes in the table above are stored in the constants `PULSARCATTESTC_E<name>`, and the status descriptions in `PULSARCATTESTC_MSGE<name>`. The source code with these messages is in `PulsarCatTest.c` on line 1.114.

**Algorithm**

This routine simply parses the input arguments, stuffs the data into a `PulsarCatNode` structure, and then calls `LAUpdatePulsarCatNode()` to update it to the new epoch.

If the `-i` option is given, the corresponding file is opened and read by `LALCHARReadVectorSequence()`, then each line is tokenized by `LALCreateTokenList()`.

Output via the `-o` option is in a custom human-readable format, which should be easy to figure out.

**Uses**

- `lalDebugLevel`
- `LALPrintError()`
- `LALmalloc()`
- `LALDCreateVector()`
- `LALCreateRandomParams()`
- `LALUniformDeviate()`
- `LALGPSstoINT8()`
- `LALCHARReadVectorSequence()`
- `LALCreateTokenList()`
- `LALReadPulsarCatHead()`
- `LALStringToD()`
- `LALLeapSec()`
- `LALSprintf()`
- `LALCheckMemoryLeaks()`
- `LALFree()`
- `LALDestroyVector()`
- `LALDestroyRandomParams()`
- `LALInitBarycenter()`
- `LALINT8toGPS()`
- `LALCHARDestroyVectorSequence()`
- `LALDestroyTokenList()`
- `LALReadPulsarCatLine()`
- `LALStringToI8()`
- `LALUpdatePulsarCat()`

**Notes**

At present the routine is kluged up to ignore pulsar position and frequency information from the command line, using hardwired parameters for PSR J0034-0534 instead. It can still override this with the `-i` option.
Section 10

Stochastic Packages
Chapter 31

Package \texttt{stochastic}

This package contains routines for stochastic background searches.
### 31.1 Header StochasticCrossCorrelation.h

Provides prototype and error code information for the modules needed to calculate the standard optimally-filtered cross-correlation statistic for stochastic background searches, given a pair of data segments, along with appropriate representations of the detector transfer function and the (uncalibrated) power spectral density of the noise in each detector. The relationship among these modules is illustrated in Fig. 31.1.

![Diagram](image)

**Figure 31.1:** Relationship among the modules dependent on StochasticCrossCorrelation.h, which are used to calculate the cross-correlation statistic \(Y\) and its theoretical variance per unit time \(\sigma^2/T\) from (uncalibrated) stretches of data \(h_1(t), h_2(t)\), from two detectors, using metadata on the power spectral densities \(P_1(f), P_2(f)\) and transfer functions \(\tilde{R}_1(f), \tilde{R}_2(f)\) for each detector. CrossCorr represents the module StochasticCrossCorrelation.c (Sec. 31.1.1) containing the functions LALStochasticCrossCorrelationStatistic(), LALStochasticHeterodynedCrossCorrelationStatistic(), and LALStochasticCrossCorrelationSpectrum(); ZeroPadAndFFT represents the module ZeroPadAndFFT.c (Sec. 31.1.5) containing the functions LALSZeroPadAndFFT() and LALCZeroPadAndFFT(); OptimalFilter represents the module StochasticOptimalFilter.c (Sec. 31.1.8) containing the function LALStochasticOptimalFilter(); Normalization represents the module StochasticOptimalFilterNormalization.c (Sec. 31.1.10) containing the function LALStochasticOptimalFilterNormalization(); InverseNoise represents the module StochasticInverseNoise.c (Sec. 31.1.12) containing the function LALStochasticInverseNoise(); OmegaGW represents the module StochasticOmegaGW.c (Sec. 31.1.14) containing the function LALStochasticOmegaGW(); Overlap represents the module OverlapReductionFunction.c (Sec. 31.1.16) containing the function OverlapReductionFunction().

### Synopsis

```c
#include <lal/StochasticCrossCorrelation.h>
```
Error conditions

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULLPTR</td>
<td>1</td>
<td>&quot;Null pointer&quot;</td>
</tr>
<tr>
<td>SAMEPTR</td>
<td>2</td>
<td>&quot;Input and Output pointers the same&quot;</td>
</tr>
<tr>
<td>ZEROLEN</td>
<td>3</td>
<td>&quot;Zero length for data member of series&quot;</td>
</tr>
<tr>
<td>NONPOSDELTAFT</td>
<td>4</td>
<td>&quot;Negative or zero frequency spacing&quot;</td>
</tr>
<tr>
<td>NONPOSDELTAFT</td>
<td>5</td>
<td>&quot;Negative or zero time spacing&quot;</td>
</tr>
<tr>
<td>NEGFMIN</td>
<td>6</td>
<td>&quot;Negative start frequency&quot;</td>
</tr>
<tr>
<td>MMTIME</td>
<td>7</td>
<td>&quot;Mismatch in epochs&quot;</td>
</tr>
<tr>
<td>MMHETERO</td>
<td>8</td>
<td>&quot;Mismatch in heterodyning frequencies&quot;</td>
</tr>
<tr>
<td>MMFMIN</td>
<td>9</td>
<td>&quot;Mismatch in start frequencies&quot;</td>
</tr>
<tr>
<td>MMDELTAFT</td>
<td>10</td>
<td>&quot;Mismatch in frequency spacings&quot;</td>
</tr>
<tr>
<td>MMLEN</td>
<td>11</td>
<td>&quot;Mismatch in sequence lengths&quot;</td>
</tr>
<tr>
<td>OORFREF</td>
<td>12</td>
<td>&quot;Out of range reference frequency&quot;</td>
</tr>
<tr>
<td>NONPOSOMEGA</td>
<td>13</td>
<td>&quot;Negative stochastic background strength&quot;</td>
</tr>
<tr>
<td>NONSYMDIJ</td>
<td>14</td>
<td>&quot;Non-symmetric response tensor&quot;</td>
</tr>
<tr>
<td>NONZEROHETERO</td>
<td>15</td>
<td>&quot;Non-zero heterodyning frequency specified for real time series&quot;</td>
</tr>
<tr>
<td>WRONGUNITS</td>
<td>16</td>
<td>&quot;Inconsistent input units&quot;</td>
</tr>
<tr>
<td>NONPOSWIN</td>
<td>17</td>
<td>&quot;Zero or negative total for window functions&quot;</td>
</tr>
<tr>
<td>MEMORY</td>
<td>18</td>
<td>&quot;Memory error&quot;</td>
</tr>
<tr>
<td>NOTYETHETERO</td>
<td>255</td>
<td>&quot;Non-zero heterodyning frequency not yet implemented&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants `STOCHASTICCROSSCORRELATIONH_E<name>`, and the status descriptions in `STOCHASTICCROSSCORRELATIONH_MSGE<name>`. The source code with these messages is in `StochasticCrossCorrelation.h` on line 1.258.

Structures

Structures and prototypes associated with `StochasticCrossCorrelation.c` (Sec. [31.1])

Prototypes

```c
void LALStochasticCrossCorrelationStatistic(
    LALStatus *status,
    REAL4WithUnits *output,
    const StochasticCrossCorrelationInput *input,
    BOOLEAN epochsMatch);

void LALStochasticHeterodynedCrossCorrelationStatistic(
    LALStatus *status,
    COMPLEX8WithUnits *output,
    const StochasticCrossCorrelationInput *input,
    BOOLEAN epochsMatch);

void LALStochasticCrossCorrelationSpectrum(
    LALStatus *status,
    COMPLEX8FrequencySeries *output,
    const StochasticCrossCorrelationInput *input,
    BOOLEAN epochsMatch);
```
LALStochasticCrossCorrelationStatisticStrain(
    LALStatus *status,
    REAL4WithUnits *output,
    const StochasticCrossCorrelationStrainInput *input,
    BOOLEAN epochsMatch);

void
LALStochasticHeterodynedCrossCorrelationStatisticStrain(
    LALStatus *status,
    COMPLEX8WithUnits *output,
    const StochasticCrossCorrelationStrainInput *input,
    BOOLEAN epochsMatch);

void
LALStochasticCrossCorrelationSpectrumStrain(
    LALStatus *status,
    COMPLEX8FrequencySeries *output,
    const StochasticCrossCorrelationStrainInput *input,
    BOOLEAN epochsMatch);

void
LALStochasticCrossCorrelationStatisticCal(
    LALStatus *status,
    REAL4WithUnits *output,
    const StochasticCrossCorrelationCalInput *input,
    BOOLEAN epochsMatch);

void
LALStochasticHeterodynedCrossCorrelationStatisticCal(
    LALStatus *status,
    COMPLEX8WithUnits *output,
    const StochasticCrossCorrelationCalInput *input,
    BOOLEAN epochsMatch);

void
LALStochasticCrossCorrelationSpectrumCal(
    LALStatus *status,
    COMPLEX8FrequencySeries *output,
    const StochasticCrossCorrelationCalInput *input,
    BOOLEAN epochsMatch);

struct REAL4WithUnits
Represents a dimensionful number as a 4-byte float with an associated units structure, which is the output of LALStochasticCrossCorrelationStatistic(). The fields are:

REAL4 value The numerical value.
LALUnit units The units.

struct COMPLEX8WithUnits
Represents a dimensionful number as a single-precision (8-byte) complex number with an associated units structure, which is the output of LALStochasticHeterodynedCrossCorrelationStatistic(). The fields are:

COMPLEX8 value The numerical value.
LALUnit units The units.
struct StochasticCrossCorrelationInput
Contains the input data needed by LALStochasticCrossCorrelationStatistic() to calculate the value of the standard optimally-filtered cross-correlation statistic. The fields are:

COMPLEX8FrequencySeries *hBarTildeOne Fourier transform of the first zero-padded data stream.
COMPLEX8FrequencySeries *hBarTildeTwo Fourier transform of the second zero-padded data stream.
COMPLEX8FrequencySeries *optimalFilter Optimal filter function in the frequency domain.

struct SZeroPadAndFFTParameters
Contains the parameters of LALSZeroPadAndFFT(). The fields are:

RealFFTPlan *fftPlan The FFT plan to be used by FFTW
REAL4Vector *window The window which is to be applied to the data
UINT4 length The length of the data after zero-padding

struct CZeroPadAndFFTParameters
Contains the parameters of LALCZeroPadAndFFT(). The fields are:

ComplexFFTPlan *fftPlan The FFT plan to be used by FFTW
REAL4Vector *window The window which is to be applied to the data
UINT4 length The length of the data after zero-padding

Prototypes associated with ZeroPadAndFFT.c (Sec. 31.1.5)

void
LALSZeroPadAndFFT(LALStatus *status, COMPLEX8FrequencySeries *output, const REAL4TimeSeries *input, SZeroPadAndFFTParameters *parameters);

void
LALCZeroPadAndFFT(LALStatus *status, COMPLEX8FrequencySeries *output, const COMPLEX8TimeSeries *input, CZeroPadAndFFTParameters *parameters);

Structures and prototypes associated with StochasticOptimalFilter.c (Sec. 31.1.8)

Prototypes

void
LALStochasticOptimalFilter(LALStatus *status, COMPLEX8FrequencySeries *optimalFilter, const StochasticOptimalFilterInput *input, const REAL4WithUnits *lambda);

void
LALStochasticOptimalFilterCal(LALStatus *status,
struct StochasticOptimalFilterInput
Contains the inputs of LALStochasticOptimalFilter(). The fields are:

REAL4FrequencySeries *overlapReductionFunction The overlap reduction function $\gamma(f)$ describing the pair of detector sites.
REAL4FrequencySeries *omegaGW The spectrum $\Omega_{GW}(f)$ of the stochastic gravitational-wave background.
COMPLEX8FrequencySeries *halfCalibratedInverseNoisePSD1 The reciprocal $1/P_{1HC}(f) = 1/(\tilde{R}_1(f)P_1(f)) = \tilde{R}_1(f)^*/P_1(f)$ of the half-calibrated noise power spectral density for the first detector.
COMPLEX8FrequencySeries *halfCalibratedInverseNoisePSD2 The reciprocal $1/P_{2HC}(f) = 1/(\tilde{R}_2(f)P_2(f)) = \tilde{R}_2(f)^*/P_2(f)$ of the half-calibrated noise power spectral density for the second detector.

Structures and prototypes associated with StochasticOptimalFilterNormalization.c (Sec. 31.1.10)

Prototypes

void
LALStochasticOptimalFilterNormalization(
    LALStatus *status,
    StochasticOptimalFilterNormalizationOutput *output,
    const StochasticOptimalFilterNormalizationInput *input,
    const StochasticOptimalFilterNormalizationParameters *parameters);

struct StochasticOptimalFilterNormalizationOutput
Contains the outputs of LALStochasticOptimalFilterNormalization(). The fields are:

REAL4WithUnits *normalization The normalization parameter $\lambda$.
REAL4WithUnits *variance The variance per unit time $\sigma^2/T$ of the cross-correlation statistic.

struct StochasticOptimalFilterNormalizationInput
Contains the inputs of LALStochasticOptimalFilterNormalization(). The fields are:

REAL4FrequencySeries *overlapReductionFunction The overlap reduction function $\gamma(f)$ describing the pair of detector sites.
REAL4FrequencySeries *omegaGW The spectrum $\Omega_{GW}(f)$ of the stochastic gravitational-wave background.
REAL4FrequencySeries *inverseNoisePSD1 The reciprocal $1/P_1(f) = |\tilde{R}_1(f)|^2/P_1(f)$ of the uncalibrated noise power spectral density for the first detector.
REAL4FrequencySeries *inverseNoisePSD2 The reciprocal $1/P_2(f) = |\tilde{R}_2(f)|^2/P_2(f)$ of the uncalibrated noise power spectral density for the second detector.
struct StochasticOptimalFilterNormalizationParameters
Contains the parameters of LALStochasticOptimalFilterNormalization(). The fields are:

REAL8 fRef The reference frequency used in defining the normalization.

BOOLEAN heterodyned Indicates whether the filter is to be used on heterodyned data or not.

REAL4Vector window1 The windowing function with which the first data stream was windowed.

REAL4Vector window2 The windowing function with which the second data stream was windowed.

Structures and prototypes associated with StochasticInverseNoise.c (Sec. 31.1.12)
Prototypes

void
LALStochasticInverseNoise(
    LALStatus *status,
    StochasticInverseNoiseOutput *output,
    const StochasticInverseNoiseInput *input);

void
LALStochasticInverseNoiseCal(
    LALStatus *status,
    StochasticInverseNoiseCalOutput *output,
    const StochasticInverseNoiseInput *input);

struct StochasticInverseNoiseOutput
Contains the outputs of LALStochasticInverseNoise(). The fields are:

REAL4FrequencySeries *calibratedInverseNoisePSD The reciprocal $1/P_C(f) = |\tilde{R}(f)|^2/P(f)$ of the un-calibrated noise power spectral density.

COMPLEX8FrequencySeries *halfCalibratedInverseNoisePSD The reciprocal $1/P_{HC}(f) = \tilde{R}(f)^*/P(f)$ of the half-calibrated noise power spectral density.

struct StochasticInverseNoiseInput
Contains the inputs to LALStochasticInverseNoise(). The fields are:

REAL4FrequencySeries *unCalibratedNoisePSD The power spectral density $P(f)$ of the noise contribution to the detector output.

COMPLEX8FrequencySeries *responseFunction The frequency-domain response function $\tilde{R}(f)$.

Structures and prototypes associated with StochasticOmegaGW.c (Sec. 31.1.14)
Prototypes

void
LALStochasticOmegaGW (  
    LALStatus *status,
    REAL4FrequencySeries *output,
    const StochasticOmegaGWParameters *parameters);
struct StochasticOmegaGWParameters

Contains the parameters used by \texttt{LALStochasticOmegaGW()} to define a power law: \( \Omega_{GW}(f) = \Omega_{R}(f/f_{R})^{\alpha} \). The fields are:

- **REAL4 alpha** The power-law exponent.
- **REAL8 fRef** The reference frequency \( f_{R} \) used to define the normalization.
- **REAL4 omegaRef** The amplitude \( \Omega_{R} = \Omega_{GW}(f_{R}) \) at reference frequency.
- **UINT4 length** The number of points in the output frequency series.
- **REAL8 f0** The start frequency of the output frequency series.
- **REAL8 deltaF** The frequency spacing of the output frequency series.

Structures and prototypes associated with \texttt{OverlapReductionFunction.c} (Sec. 31.1.16)

Prototypes:

```c
void LALOverlapReductionFunction(
    LALStatus *status,
    REAL4FrequencySeries *output,
    const LALDetectorPair *detectors,
    const OverlapReductionFunctionParameters *parameters);
```

struct OverlapReductionFunctionParameters

Contains the parameters used by \texttt{LALOverlapReductionFunction()} to determine the format of its output for the overlap reduction function. The fields are:

- **UINT4 length** The number of points in the output frequency series.
- **REAL8 f0** The start frequency of the output frequency series.
- **REAL8 deltaF** The frequency spacing of the output frequency series.

struct LALDetectorPair

Holds structures defining the location and orientation of a pair of gravitational wave detectors. This is the input to \texttt{LALOverlapReductionFunction()}. The fields are:

- **LALDetector detectorOne** The first interferometer.
- **LALDetector detectorTwo** The second interferometer.
31.1.1 Module StochasticCrossCorrelation.c

Calculates the value of the standard optimally-filtered cross-correlation statistic for stochastic background searches.

Prototypes

```c
void LALStochasticCrossCorrelationStatistic(
    LALStatus *status, 
    REAL4WithUnits *output, 
    const StochasticCrossCorrelationInput *input, 
    BOOLEAN epochsMatch)
```

```c
void LALStochasticHeterodynedCrossCorrelationStatistic(
    LALStatus *status, 
    COMPLEX8WithUnits *output, 
    const StochasticCrossCorrelationInput *input, 
    BOOLEAN epochsMatch)
```

```c
void LALStochasticCrossCorrelationSpectrum(
    LALStatus *status, 
    COMPLEX8FrequencySeries *output, 
    const StochasticCrossCorrelationInput *input, 
    BOOLEAN epochsMatch)
```

Description

**LALStochasticCrossCorrelationStatistic()**

The default version of the function, for handling non-heterodyned data, calculates the value of the standard optimally-filtered cross-correlation statistic

\[
Y := \int_{t_0}^{t_0+T} dt_1 \int_{t_0}^{t_0+T} dt_2 w_1(t_1) h_1(t_1) Q(t_1-t_2) w_2(t_2) h_2(t_2)
\]

\[
\approx \sum_{j=0}^{N-1} \delta t \sum_{k=0}^{N-1} \delta t w_1[j] h_1[j] Q[j-k] w_2[j] h_2[k]
\]

\[
= \sum_{\ell=0}^{M-1} \delta f \tilde{h}_1[\ell] \ast \tilde{Q}[\ell] \tilde{h}_2[\ell],
\]

where the sampling period is \(\delta t = T/N\), the frequency spacing is \(\delta f = [M\delta t]^{-1}\), the tilde indicates a discrete Fourier transform normalized to approximate the continuous Fourier transform:

\[
\tilde{Q}[\ell] := \sum_{k=0}^{N-1} \delta t Q[k] e^{-i2\pi k\ell/M}
\]

the asterisk indicates complex conjugation, and the overbar indicates windowing and zero-padding:

\[
\tilde{h}[k] = \begin{cases} 
    w[k] h[k] & k = 0, \ldots, N-1 \\
    0 & k = N, \ldots, M-1 
\end{cases}
\]

which is needed because the range of indices for \(h[k]\) and \(Q[k]\) do not match. \(M\) should be at least \(2N - 1\), but may be chosen to be, e.g., \(2M\) for convenience. The inputs to **LALStochasticCrossCorrelationStatistic()** are the (windowed) zero-padded, FFTed data streams \(\tilde{h}_1[\ell]\) and \(\tilde{h}_2[\ell]\), along with the optimal filter \(\tilde{Q}[\ell]\). Since the underlying time series are real, the input series only need to include the values for \(\ell = 0, \ldots, P-1\) (where \(P = \lceil \frac{M+1}{2} \rceil\) is the number of independent elements...
in the frequency series) with the elements corresponding to negative frequencies determined by complex conjugation. This allows $Y$ to be computed as

$$Y = \delta f \left( \bar{h}_1[0] \tilde{Q}[0] \bar{h}_2[0] + 2 \sum_{\ell=1}^{P-1} \text{Re} \left\{ \bar{h}_1[\ell]^* \tilde{Q}[\ell] \bar{h}_2[\ell] \right\} \right). \quad (31.4)$$

The routine \texttt{LALStochasticCrossCorrelationStatistic()} is designed for analyzing non-heterodyned data, so if the input FFTed datasets have a positive start frequency, and thus represent a range of frequencies $f_0 \leq f < f_0 + (P-1)\delta f$, it is assumed that they were produced by discarding frequencies below $f_0$ from a longer frequency series, which was still the Fourier transform of a real time series. In this case the cross-correlation statistic is calculated as

$$Y = \delta f \sum_{\ell=0}^{P-1} \text{Re} \left\{ \bar{h}_1[\ell]^* \tilde{Q}[\ell] \bar{h}_2[\ell] \right\}$$

$$\approx \int_{-f_0}^{-f_0-\delta f} df \bar{h}_1(f)^* \tilde{Q}(f) \bar{h}_2(f) + \int_{f_0}^{f_0+P\delta f} df \bar{h}_1(f)^* \tilde{Q}(f) \bar{h}_2(f) \quad (31.5)$$

The frequency sampling parameters (start frequency, frequency spacing, and number of points) must be the same for both data streams, but if the optimal filter is more coarsely sampled (for instance, if it varies in frequency too slowly to warrant the finer resolution), the data streams will be multiplied in the frequency domain and their product coarse-grained (cf. Sec. ??) to the optimal filter resolution before calculating (31.5).

If the \texttt{epochsMatch} boolean variable is set to a true value, the function will confirm that the start times for both time series agree. It can be set to false to allow for cross-correlation of time-shifted data as a control case.

\texttt{LALStochasticHeterodynedCrossCorrelationStatistic()}

In the case of heterodyned data, one wishes to calculate

$$Y := \int_{t_0}^{t_0+T} dt_1 \int_{t_0}^{t_0+T} dt_2 w_1(t_1) h_1(t_1)^* Q(t_1 - t_2) w_2(t_2) h_2(t_2)$$

$$\approx \sum_{j=0}^{N-1} \delta t \sum_{k=0}^{N-1} \delta t w_1[k] h_1[j]^* Q[j-k] w_2[k] h_2[k]$$

$$= \sum_{\ell=0}^{M-1} \delta f \bar{h}_1[\ell]^* \tilde{Q}[\ell] \bar{h}_2[\ell], \quad (31.6)$$

In this case, the Fourier transforms of the zero-padded data streams have $M$ independent elements, which must all be included in the sum, which is calculated as

$$Y = \sum_{\ell=0}^{M-1} \bar{h}_1[\ell]^* \tilde{Q}[\ell] \bar{h}_2[\ell]. \quad (31.7)$$

While the mean value of the cross-correlation statistic for heterodyned data should be real (assuming both series were heterodyned with the same phase), the value for an individual stretch of data will be complex, so the output is returned as \texttt{COMPLEX8WithUnits}.

\texttt{LALStochasticCrossCorrelationSpectrum()}

For diagnostic purposes, this function calculates the integrand of (31.1) or (31.6), i.e.

$$Y(f) = \bar{h}_1(f)^* \tilde{Q}(f) \bar{h}_2(f) = Y[\ell] = \bar{h}_1[\ell]^* \tilde{Q}[\ell] \bar{h}_2[\ell]$$

and returns it as a frequency series.

\footnote{Note that the $P$th frequency bin is not treated specially, as would be expected for the Nyquist frequency. This is the appropriate behavior if $M$ is an odd number (so that there is no Nyquist bin) or if, as a result of coarse-graining, the Nyquist bin has been removed from $\tilde{Q}$. At any rate, if there’s a significant contribution to the cross-correlation statistic from the Nyquist frequency, something is wrong.}
Algorithm

The function `LALStochasticCrossCorrelationSpectrum()` calculates the integrand \(31.8\) as follows:

First it calculates \(\bar{h}_1[\ell]^* \bar{h}_2[\ell]\) with `LALCCVectorMultiplyConjugate()` and matches the resolution of the result to that of `input->optimalFilter` with `LALCCoarseGrainFrequencySeries()`. Then it uses `LALCCVectorMultiply()` to calculate \(31.8\) from the input \(Q[\ell]\) and the coarse-grained \(\bar{h}_1[\ell]^* \bar{h}_2[\ell]\).

The functions `LALStochasticCrossCorrelationStatistic()` and `LALStochasticHeterodynedCrossCorrelationStatistic()` call `LALStochasticCrossCorrelationSpectrum()` and then integrate over all frequencies to calculate \(31.4\) or \(31.7\), respectively.

Uses

`LALStochasticCrossCorrelationSpectrum()` calls

- `LALCreateVector()`
- `LALCCVectorMultiplyConjugate()`
- `LALDestroyVector()`
- `LALCoarseGrainFrequencySeries()`
- `LALUnitMultiply()`

`LALStochasticCrossCorrelationStatistic()` and `LALStochasticHeterodynedCrossCorrelationStatistic()` call

- `LALCreateVector()`
- `LALStochasticCrossCorrelationSpectrum()`
- `LALDestroyVector()`
- `LALUnitMultiply()`

Notes

- When \(f_0 = 0\), \(\bar{h}_1[0], Q[0]\), and \(\bar{h}_2[0]\) are assumed to be real, but this is not checked.

- The optimal filter \(\tilde{Q}(f)\) is represented by a complex frequency series because it will in general be applied to whitened data include the different complex whitening filters for the two streams. (cf. Sec. 31.1.8)

- The coarse-graining technique produces the same cross-correlation statistic as fine-graining the optimal filter by assuming it is zero outside the coarse-grained frequency range and constant across each coarse-grained frequency bin.

- The output units are constructed by combining the input units, but under normal circumstances the units will be as follows:

  \[
  [\tilde{Q}] = \text{count}^{-2} \quad (31.9) \\
  [\bar{h}_{1,2}] = \text{count Hz}^{-1} \quad (31.10) \\
  [Y(f)] := [\tilde{h}_1][\tilde{Q}][\tilde{h}_2] = s^2 \quad (31.11) \\
  [Y] := [Y(f)] \text{Hz}^{-1} = s \quad (31.12)
  \]
31.1.2 Program StochasticCrossCorrelationStatisticTest.c

A program to test LALStochasticCrossCorrelationStatistic().

Usage

./StochasticCrossCorrelationStatisticTest [options]
Options:
  -h  print usage message
  -q  quiet: run silently
  -v  verbose: print extra information
  -d level set lalDebugLevel to level
  -i filename read first data stream from file filename
  -j filename read second data stream from file filename
  -k filename read optimal filter from file filename
  -n length frequency series contain length points
  -t  epochs need not match

This program tests the function LALStochasticCrossCorrelationStatistic(), which calculates the cross-correlation statistic given two zero-padded and Fourier-transformed data streams and a (frequency domain) optimal filter.

First, it tests that the correct error codes (cf. Sec. 31.1) are generated for the following error conditions (tests in italics are not performed if LAL_NDEBUG is set, as the corresponding checks in the code are made using the ASSERT macro):

- null pointer to output structure
- null pointer to input structure
- null pointer to first data stream
- null pointer to second data stream
- null pointer to optimal filter
- null pointer to data member of first data stream
- null pointer to data member of second data stream
- null pointer to data member of optimal filter
- null pointer to data member of data member of first data stream
- null pointer to data member of data member of second data stream
- null pointer to data member of data member of optimal filter
- zero length
- negative frequency spacing
- zero frequency spacing
- negative start frequency
- length mismatch between optimal filter and first data stream
- length mismatch between optimal filter and second data stream
- frequency spacing mismatch between optimal filter and first data stream
- frequency spacing mismatch between optimal filter and second data stream
- start frequency mismatch between optimal filter and first data stream
- start frequency mismatch between optimal filter and second data stream
mismatch between epochs of data streams

It then verifies that the correct cross-correlation statistic (value and units) is generated for each of the following simple test cases:

1. \( \tilde{Q}(f) = x(1-x); \tilde{h}_1(f) = x^2 + ix, \tilde{h}_2(f) = x^{-2} - ix^{-1}, \) with \( x = f/400 \text{ Hz} \). The expected result in this case is zero.

2. \( \tilde{Q}(f) = 1 \) for \( 300 \text{ Hz} < f < 500 \text{ Hz} \), \( 0 \) otherwise; \( \tilde{h}_1(f) = 1 - \tilde{h}_2(f) = f/800 \text{ Hz} \). With \( f_0 = \delta f = 80 \text{ Hz} \) and \( N = 9 \), the expected value is 116.8.

For each successful test (both of these valid data and the invalid ones described above), it prints “PASS” to standard output; if a test fails, it prints “FAIL”.

If the filename arguments are present, it also reads in the optimal filter and the two data streams from the specified files and use the specified parameters to calculate the cross-correlation statistic. The result is printed to standard output along with the resulting units in terms of the basic SI units.

Exit codes

<table>
<thead>
<tr>
<th>name</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOM</td>
<td>0</td>
<td>&quot;Nominal exit&quot;</td>
</tr>
<tr>
<td>ARG</td>
<td>1</td>
<td>&quot;Error parsing command-line arguments&quot;</td>
</tr>
<tr>
<td>CHK</td>
<td>2</td>
<td>&quot;Error checking failed to catch bad data&quot;</td>
</tr>
<tr>
<td>FLS</td>
<td>3</td>
<td>&quot;Incorrect answer for valid data&quot;</td>
</tr>
<tr>
<td>USE</td>
<td>4</td>
<td>&quot;Bad user-entered data&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants STOCHASTICCROSSCORRELATIONSTATISTICTESTC_E<name>., and the status descriptions in STOCHASTICCROSSCORRELATIONSTATISTICTESTC_MSGE<name>. The source code with these messages is in StochasticCrossCorrelationStatisticTest.c on line 1.213.

Uses

LALStochasticCrossCorrelationStatistic()  
LALCheckMemoryLeaks()  
LALCReadFrequencySeries()  
LALCCreateVector()  
LALCDestroyVector()  
LALCHARCCreateVector()  
LALCHARDestroyVector()  
LALUnitAsString()  
LALUnitCompare()  
getopt()  
printf()  
fprintf()  
freopen()  
fabs()

Notes

- No specific error checking is done on user-specified data. If length is missing, the resulting default will cause a bad data error.

- The length of the user-provided series must be specified, even though it could in principle be deduced from the input file, because the data sequences must be allocated before the LALCReadFrequencySeries() function is called.

- If some, but not all, of the filename arguments are present, the user-specified data will be silently ignored.
31.1.3 Program StochasticHeterodynedCrossCorrelationStatisticTest.c

A program to test LALStochasticHeterodynedCrossCorrelationStatistic().

Usage

./StochasticHeterodynedCrossCorrelationStatisticTest [options]
Options:

- h print usage message
- q quiet: run silently
- v verbose: print extra information
- d level set lalDebugLevel to level
- i filename read first data stream from file filename
- j filename read second data stream from file filename
- k filename read optimal filter from file filename
- n length frequency series contain length points
- t epochs need not match

This program tests the function LALStochasticHeterodynedCrossCorrelationStatistic(), which calculates the cross-correlation statistic given two zero-padded and Fourier-transformed data streams and a (frequency domain) optimal filter.

First, it tests that the correct error codes (cf. Sec. 31.1) are generated for the following error conditions (tests in italics are not performed if LAL_NDEBUG is set, as the corresponding checks in the code are made using the ASSERT macro):

- null pointer to output structure
- null pointer to input structure
- null pointer to first data stream
- null pointer to second data stream
- null pointer to optimal filter
- null pointer to data member of first data stream
- null pointer to data member of second data stream
- null pointer to data member of optimal filter
- null pointer to data member of data member of first data stream
- null pointer to data member of data member of second data stream
- null pointer to data member of data member of optimal filter
- zero length
- negative frequency spacing
- zero frequency spacing
- negative start frequency
- length mismatch between optimal filter and first data stream
- length mismatch between optimal filter and second data stream
- frequency spacing mismatch between optimal filter and first data stream
- frequency spacing mismatch between optimal filter and second data stream
- start frequency mismatch between optimal filter and first data stream
- start frequency mismatch between optimal filter and second data stream
mismatch between epochs of data streams

It then verifies that the correct cross-correlation statistic (value and units) is generated for each of the following simple test cases:

1. \( \bar{Q}(f) = \frac{f(N\delta f-f)}{(N\delta f/2)^2}; \bar{h}(f) = f^2 + if; \tilde{h}_2(f) = f^{-2} - if. \) With \( f_0 = \delta f = 80 \text{ Hz} \) and \( N = 9 \), the expected value is \(-1248\text{i}\).

2. \( \bar{Q}(f) = 1 \) for \( 300 \text{ Hz} < f < 500 \text{ Hz} \), 0 otherwise; \( \bar{h}_1(f) = 1 - \bar{h}_2(f) = f/800 \text{ Hz} \). With \( f_0 = \delta f = 80 \text{ Hz} \) and \( N = 9 \), the expected value is 58.4.

For each successful test (both of these valid data and the invalid ones described above), it prints “PASS” to standard output; if a test fails, it prints “FAIL”.

If the filename arguments are present, it also reads in the optimal filter and the two data streams from the specified files and use the specified parameters to calculate the cross-correlation statistic. The result is printed to standard output along with the resulting units in terms of the basic SI units.

Exit codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOM</td>
<td>0</td>
<td>&quot;Nominal exit&quot;</td>
</tr>
<tr>
<td>ARG</td>
<td>1</td>
<td>&quot;Error parsing command-line arguments&quot;</td>
</tr>
<tr>
<td>CHK</td>
<td>2</td>
<td>&quot;Error checking failed to catch bad data&quot;</td>
</tr>
<tr>
<td>FLS</td>
<td>3</td>
<td>&quot;Incorrect answer for valid data&quot;</td>
</tr>
<tr>
<td>USE</td>
<td>4</td>
<td>&quot;Bad user-entered data&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants STOCHASTICCROSSCORRELATIONTESTC_E<name>, and the status descriptions in STOCHASTICCROSSCORRELATIONTESTC_MSE<name>. The source code with these messages is in StochasticCrossCorrelationStatisticTest.c on line 1.216.

Uses

- LALStochasticCrossCorrelationStatistic()
- LALCheckMemoryLeaks()
- LALReadFrequencySeries()
- LALCreateVector()
- LALDestroyVector()
- LALCreateVector()
- LALDestroyVector()
- LALUnitAsString()
- LALUnitCompare()
- getopt()
- printf()
- fprintf()
- freopen()
- fabs()

Notes

- No specific error checking is done on user-specified data. If length is missing, the resulting default will cause a bad data error.

- The length of the user-provided series must be specified, even though it could in principle be deduced from the input file, because the data sequences must be allocated before the LALReadFrequencySeries() function is called.

- If some, but not all, of the filename arguments are present, the user-specified data will be silently ignored.
31.1.4 Program StochasticCrossCorrelationSpectrumTest.c

A program to test LALStochasticCrossCorrelationSpectrum().

Usage

./StochasticCrossCorrelationSpectrumTest [options]

Options:

- h    print usage message
- q    quiet: run silently
- v    verbose: print extra information
- d level set lalDebugLevel to level
- o filename write spectrum to file filename
- i filename read first data stream from file filename
- j filename read second data stream from file filename
- k filename read optimal filter from file filename
- m length optimal filter contains length points
- n length data streams contain length points
- t epochs need not match

This program tests the function LALStochasticCrossCorrelationSpectrum(), which calculates the cross-correlation spectrum given two zero-padded and Fourier-transformed data streams and a (frequency domain) optimal filter.

First, it tests that the correct error codes (cf. Sec. 31.1) are generated for the following error conditions (tests in italics are not performed if LAL_NDEBUG is set, as the corresponding checks in the code are made using the ASSERT macro):

- null pointer to output series
- null pointer to input structure
- null pointer to first data stream
- null pointer to second data stream
- null pointer to optimal filter
- null pointer to data member of first data stream
- null pointer to data member of second data stream
- null pointer to data member of optimal filter
- null pointer to data member of data member of first data stream
- null pointer to data member of data member of second data stream
- null pointer to data member of data member of optimal filter
- zero length
- negative frequency spacing
- zero frequency spacing
- negative start frequency
- length mismatch between data streams
- frequency spacing mismatch between data streams
- start frequency mismatch between data streams
- mismatch between epochs of data streams
It then verifies that the correct cross-correlation statistic (value and units) is generated for the following simple test case: \(\tilde{Q}(f) = x(1 - x); \tilde{h}_1(f) = x^2 + ix; \tilde{h}_2(f) = x^{-2} - ix^{-1}\), with \(x = f/400\) Hz. The expected result in this case is \(Y(f) = -i(2 - x)(x^2 + 2)\). For each successful test (this valid data test and the invalid ones described above), it prints "PASS" to standard output; if a test fails, it prints "FAIL".

If the filename arguments are present, it also reads in the optimal filter and the two data streams from the specified files and use the specified parameters to calculate the cross-correlation statistic. The result is printed to the specified output file.

### Exit codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOM</td>
<td>0</td>
<td>&quot;Nominal exit&quot;</td>
</tr>
<tr>
<td>ARG</td>
<td>1</td>
<td>&quot;Error parsing command-line arguments&quot;</td>
</tr>
<tr>
<td>CHK</td>
<td>2</td>
<td>&quot;Error checking failed to catch bad data&quot;</td>
</tr>
<tr>
<td>FLS</td>
<td>3</td>
<td>&quot;Incorrect answer for valid data&quot;</td>
</tr>
<tr>
<td>USE</td>
<td>4</td>
<td>&quot;Bad user-entered data&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants \(\text{STOCHASTICCROSSCORRELATIONSPECTRUMTESTC}\_<\text{name}>\), and the status descriptions in \(\text{STOCHASTICCROSSCORRELATIONSPECTRUMTESTC\_MSGE}\_<\text{name}>\). The source code with these messages is in \text{StochasticCrossCorrelationSpectrumTest.c} on line 1.207.

### Uses

- LALStochasticCrossCorrelationSpectrum()
- LALCheckMemoryLeaks()
- LALCReadFrequencySeries()
- LALCCreateVector()
- LALCDestroyVector()
- LALCHARCreateVector()
- LALCHARDestroyVector()
- LALUnitAsString()
- LALUnitCompare()
- getopt()
- printf()
- fprintf()
- freopen()
- fabs()

### Notes

- No specific error checking is done on user-specified data. If length is missing, the resulting default will cause a bad data error.

- The length of the user-provided series must be specified, even though it could in principle be deduced from the input file, because the data sequences must be allocated before the LALCReadFrequencySeries() function is called.

- If some, but not all, of the filename arguments are present, the user-specified data will be silently ignored.
31.1.5 Module ZeroPadAndFFT.c

Routines for zero-padding and Fourier transforming a time series.

Prototypes

```c
void LALSZeroPadAndFFT(
    LALStatus *status,
    COMPLEX8FrequencySeries *output,
    const REAL4TimeSeries *input,
    SZeroPadAndFFTParameters *parameters)

void LALCZeroPadAndFFT(
    LALStatus *status,
    COMPLEX8FrequencySeries *output,
    const COMPLEX8TimeSeries *input,
    CZeroPadAndFFTParameters *parameters)
```

Description

As described in 31.1.1, data streams to be cross-correlated need to be zero-padded to the same length as the optimal filter via

\[
\tilde{h}[k] = \begin{cases} 
    w[k]h[k] & k = 0, \ldots, N - 1 \\
    0 & k = N, \ldots, M - 1
\end{cases}
\]  

(31.13)

(where \(w[k]\) is a windowing function) before being Fourier transformed via

\[
\tilde{h}[\ell] := \sum_{k=0}^{M-1} \delta t \ h[k] e^{-i2\pi k\ell/M}.
\]  

(31.14)

LALSZeroPadAndFFT() performs this operation on a REAL4TimeSeries of length \(N\), zero-padding it to length \(M\) and Fourier-transforming it into a COMPLEX8FrequencySeries of length \([M/2] + 1\).

LALCZeroPadAndFFT() performs this operation on a COMPLEX8TimeSeries of length \(N\), zero-padding it to length \(M\) and Fourier-transforming it into a COMPLEX8FrequencySeries of length \(M\).

Algorithm

LALSZeroPadAndFFT() constructs the sequence \(\tilde{h}[k]\), and then applies a real-to-complex time-to-frequency discrete Fourier transform from the fft package.

LALCZeroPadAndFFT() constructs the sequence \(\tilde{h}[k]\), and then applies a complex-to-complex time-to-frequency discrete Fourier transform from the fft package.

Uses

LALSZeroPadAndFFT() calls:

LALSCreateVector()
LALSDestroyVector()
LALTImeFreqRealFFT()
memset()
strncpy()

LALSZeroPadAndFFT() calls:

LALCCreateVector()
LALCDestroyVector()
LALTImeFreqComplexFFT()
memset()
strncpy()
Notes

- The Fourier transform is defined to be the discrete approximation of a continuous Fourier transform, which makes it $\delta t$ times the discrete Fourier transform.

- The Fourier transform of a series of $M$ points is calculated with the FFTW [1] (via the interfaces in the fft package), which is efficient for products of small primes, so $M$ should be chosen to have this property. The minimum value, $2N - 1$, is odd and can thus be at best a power of 3. Additionally, if $2N - 1$ is a convenient number, $N$ will likely not be, which is one reason it might be convenient to work with $M = 2N$ instead.

- LALCZeroPadAndFFT() inherits its behavior from LALTimeFreqComplexFFT(), which currently does not use the initial phase of the reference oscillator. The calling routine must therefore remove the effects of this phase explicitly in order to obtain the band-limited FFT of the unheterodyned data.

- The output units are determined from the input units, but under normal circumstances in the context of a stochastic background search, we will have

\[
\begin{align*}
[h(t)] & = \text{count} \\
[\tilde{h}(f)] & := [h(t)] \text{Hz}^{-1} = \text{count Hz}^{-1}
\end{align*}
\]
31.1.6 Program **SZeroPadAndFFTTest.c**

Test suite for **LALSZeroPadAndFFT()**.

**Usage**

```
./SZeroPadAndFFTTest
Options:
  -h  print usage message
  -q  quiet: run silently
  -v  verbose: print extra information
  -d level  set lalDebugLevel to level
  -i filename  read time series from file filename
  -o filename  print frequency series to file filename
  -n length  input & output series contain length points
  -m measure plan
```

**Description**

This program tests the routine **LALSZeroPadAndFFT()**, which zero-pads and Fourier transforms a real time series of length \( N \) to produce a complex frequency series of length \( \lceil M/2 \rceil + 1 \).

First, it tests that the correct error codes (cf. Sec. 31.1) are generated for the following error conditions (tests in *italics* are not performed if **LAL_NDEBUG** is set, as the corresponding checks in the code are made using the ASSERT macro):

- null pointer to output series
- null pointer to input series
- null pointer to parameter structure
- null pointer to FFT plan
- null pointer to data member of window function
- null pointer to data member of output series
- null pointer to data member of input series
- null pointer to data member of data member of input series
- null pointer to data member of data member of output series
- zero length
- negative time spacing
- zero time spacing
- negative heterodyning frequency
- positive heterodyning frequency
- length mismatch between input series and window function
- length mismatch between output series and zero-padded data
- output series shorter than input series

It then verifies that the correct frequency series is generated for the simple test case \( \{h[k] = 1 + k|k = 0, \ldots, 7\} \) with \( \delta t = 0.5 \) s and rectangular windowing. For each successful test (both of these valid data and the invalid ones described above), it prints “**PASS**” to standard output; if a test fails, it prints “**FAIL**”.

If the **filename** arguments are present, it also reads a time series from a file, calls **LALSZeroPadAndFFT()**, and writes the results to the specified output file.

---

2The values are compared to a series generated by Matlab, correcting for the differing sign convention.
Exit codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOM</td>
<td>0</td>
<td>&quot;Nominal exit&quot;</td>
</tr>
<tr>
<td>ARG</td>
<td>1</td>
<td>&quot;Error parsing command-line arguments&quot;</td>
</tr>
<tr>
<td>CHK</td>
<td>2</td>
<td>&quot;Error checking failed to catch bad data&quot;</td>
</tr>
<tr>
<td>FLS</td>
<td>3</td>
<td>&quot;Incorrect answer for valid data&quot;</td>
</tr>
<tr>
<td>USE</td>
<td>4</td>
<td>&quot;Bad user-entered data&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants `SZEROPADANDFFTTESTC_E<name>`, and the status descriptions in `SZEROPADANDFFTTESTC_MSGE<name>`. The source code with these messages is in `SZeroPadAndFFTTest.c` on line 1.202.

Uses

- LALSZeroPadAndFFT()
- LALCheckMemoryLeaks()
- LALSReadTimeSeries()
- LALCPrintFrequencySeries()
- LALCreateVector()
- LALDestroyVector()
- LALSZeroPadAndFFT()
- LALCHARCreateVector()
- LALCHARDestroyVector()
- LALCreateForwardRealFFTPlan()
- LALDestroyRealFFTPlan()
- LALUnitAsString()
- LALUnitCompare()
- getopt()
- printf()
- fprintf()
- freopen()
- fabs()

Notes

- No specific error checking is done on user-specified data. If `length` is missing, the resulting default will cause a bad data error.
- The length of the user-provided series must be specified, even though it could in principle be deduced from the input file, because the data sequences must be allocated before the `LALSReadTimeSeries()` function is called.
- If one `filename` argument, but not both, is present, the user-specified data will be silently ignored.
31.1.7 Program CZeroPadAndFFTTest.c

Test suite for LALCZeroPadAndFFT().

Usage

```bash
./CZeroPadAndFFTTest
```

Options:
- `-h` print usage message
- `-q` quiet: run silently
- `-v` verbose: print extra information
- `-d level` set lalDebugLevel to level
- `-i filename` read time series from file filename
- `-o filename` print frequency series to file filename
- `-n length` input series contains length points
- `-m` measure plan

Description

This program tests the routine LALCZeroPadAndFFT(), which zero-pads and Fourier transforms a complex time series of length $N$ to produce a complex frequency series of length $2N - 1$.

First, it tests that the correct error codes (cf. Sec. 31.1) are generated for the following error conditions (tests in *italics* are not performed if LAL_NDEBUG is set, as the corresponding checks in the code are made using the ASSERT macro):

- null pointer to output series
- null pointer to input series
- null pointer to parameter structure
- null pointer to FFT plan
- null pointer to data member of window function
- null pointer to data member of output series
- null pointer to data member of input series
- null pointer to data member of optimal filter
- null pointer to data member of data member of input series
- null pointer to data member of data member of output series
- zero length
- negative time spacing
- zero time spacing
- length mismatch between input series and window function
- length mismatch between output series and zero-padded data
- output series shorter than input series

It then verifies that the correct frequency series\(^3\) is generated for the simple test case \(h[k] = 1 + k|k = 0, \ldots, 7\) with \(\delta t = 0.5\) s and rectangular windowing. For each successful test (both of these valid data and the invalid ones described above), it prints “PASS” to standard output; if a test fails, it prints “FAIL”.

If the `filename` arguments are present, it also reads a time series from a file, calls LALCZeroPadAndFFT(), and writes the results to the specified output file.

\(^3\)The values are compared to a series generated by Matlab, correcting for the differing sign convention.
Exit codes

<table>
<thead>
<tr>
<th>name</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOM</td>
<td>0</td>
<td>&quot;Nominal exit&quot;</td>
</tr>
<tr>
<td>ARG</td>
<td>1</td>
<td>&quot;Error parsing command-line arguments&quot;</td>
</tr>
<tr>
<td>CHK</td>
<td>2</td>
<td>&quot;Error checking failed to catch bad data&quot;</td>
</tr>
<tr>
<td>FLS</td>
<td>3</td>
<td>&quot;Incorrect answer for valid data&quot;</td>
</tr>
<tr>
<td>USE</td>
<td>4</td>
<td>&quot;Bad user-entered data&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants `CZEROPADANDFFTTESTC_E<name>` and the status descriptions in `CZEROPADANDFFTTESTC_MSGE<name>`. The source code with these messages is in `CZeroPadAndFFTTest.c` on line 1.201.

Uses

LALCZeroPadAndFFT()
LALCheckMemoryLeaks()
LALCReadTimeSeries()
LALCPrintFrequencySeries()
LALCCreateVector()
LALCDestroyVector()
LALCHARCreateVector()
LALCHARDestroyVector()
LALCreateForwardRealFFTPlan()
LALDestroyRealFFTPlan()
LALUnitAsString()
LALUnitCompare()
getopt()
printf()
fprintf()
freopen()
fabs()

Notes

- No specific error checking is done on user-specified data. If `length` is missing, the resulting default will cause a bad data error.

- The length of the user-provided series must be specified, even though it could in principle be deduced from the input file, because the data sequences must be allocated before the `LALCReadTimeSeries()` function is called.

- If one `filename` argument, but not both, is present, the user-specified data will be silently ignored.
31.1.8 Module StochasticOptimalFilter.c

Calculates the values of the optimal filter function for the standard cross-correlation statistic.

Prototypes

```c
void LALStochasticOptimalFilter(
    LALStatus *status,
    COMPLEX8FrequencySeries *optimalFilter,
    const StochasticOptimalFilterInput *input,
    const REAL4WithUnits *lambda)
```

Description

As described in [2, 3, 4], the optimal filter \( \tilde{Q}^C(f) \) which maximizes the ratio of the mean \( \mu = \langle Y \rangle \) to the standard deviation \( \sigma = \sqrt{\langle (Y - \mu)^2 \rangle} \) of the cross-correlation statistic (31.1) is

\[
\tilde{Q}^C(f) = \lambda \frac{\gamma(f) \Omega_{GW}(f)}{|f|^3 P_i^C(f) P_2^C(f)}
\]

(31.17)

where \( \lambda \) is a normalization constant, \( \gamma(f) \) is the overlap reduction function (cf. Sec. 31.1.16) for the two detectors, \( \Omega_{GW}(f) \) is the stochastic gravitational wave background strength (cf. Sec. 31.1.16), and \( P_i^C(f) \) is the power spectral density \( \langle h_i^C(f) h_i^C(f') \rangle = \delta(f - f') P_i^C(f) \) for the \( i \)th detector.

However, in practice, the data stream coming out of the \( i \)th detector is not the strain \( h_i^C(t) = h_{ab}(t, \vec{x}_i) d^{ab} \), but that convolved with an instrumental response function \( R_i(\tau) \) to produce an “uncalibrated” data stream

\[
h_i(t) = \int_0^\infty d\tau R_i(\tau) h_i^C(t - \tau)
\]

(31.18)

which has the simpler frequency-domain representation

\[
\tilde{h}_i(f) = \tilde{R}_i(f) \tilde{h}_i(f)
\]

(31.19)

If we want to calculate the cross-correlation statistic \( Y \) using the uncalibrated detector output, the expression is

\[
Y = \int_{-\infty}^\infty df \left( \frac{\tilde{h}_1(f)}{\tilde{R}_1(f)} \right)^* \tilde{Q}^C(f) \left( \frac{\tilde{h}_2(f)}{\tilde{R}_2(f)} \right) = \int_{-\infty}^\infty df \tilde{h}_1(f)^* \tilde{Q}(f) \tilde{h}_2(f)
\]

(31.20)

where the “uncalibrated optimal filter” is

\[
\tilde{Q}(f) = \frac{\tilde{Q}^C(f)}{\tilde{R}_1(f)^* \tilde{R}_2(f)} = \lambda \frac{1}{\tilde{R}_1(f)^* P_i^C(f)} \frac{\gamma(f) \Omega_{GW}(f)}{|f|^3} \frac{1}{\tilde{R}_2(f) P_2^C(f)}
\]

\[
\tilde{Q}(f) = \lambda \frac{\gamma(f) \Omega_{GW}(f)}{|f|^3 P_i^HC(f) P_2^HC(f)}
\]

(31.21)

where \( P_i^HC(f) = \tilde{R}_i(f) P_i^C(f) \) is the “half-calibrated” PSD. (The uncalibrated PSD is \( P_i(f) = |\tilde{R}_i(f)|^2 P_i^C(f) \).)

\[
\text{LALStochasticOptimalFilter}()
\]

generates a complex frequency series containing the uncalibrated optimal filter \( \tilde{Q}(f) \), taking as inputs real frequency series representing the overlap reduction function \( \gamma(f) \) and the stochastic gravitational wave background spectrum \( h_{100}^2 \Omega_{GW}(f) \), as well as complex frequency series representing the half-calibrated (inverse) PSDs \( \{1/P_i^HC(f)\} |i = 1, 2\} \), and as a real parameter the normalization constant \( \lambda \).

Algorithm

The routine \( \text{LALStochasticOptimalFilter}() \) fills its output series is filled with the values corresponding to the definition (31.21).
Uses

LALUnitMultiply()
LALUnitRaise()
LALUnitCompare()

Notes

- If $f_0 = 0$, the DC element $Q(0)$ is set to zero, regardless of the values of the inputs, because the $f^3$ term would make it diverge otherwise, and because any conceivable realistic noise spectrum will end up blowing up at zero frequency fast enough to kill the optimal filter.

- The implementation of the optimal filter function given here assumes a large observation time continuum-limit approximation. In this limit, the Dirichlet kernels (which appear in an exact expression for the standard cross-correlation statistic, when evaluated in discrete time [4]; see also the documentation for the module Dirichlet.c in the utilities package) may be replaced by Dirac delta functions.

- Although $Q_C(f)$ is real by construction, the uncalibrated optimal filter $\tilde{Q}(f)$ will in general be complex because the response functions $\tilde{R}_i(f)$ for the two sites will be different.

- The expected units for the inputs and output of this function are as follows (although the actual output units will be constructed from the input units):

$$[\lambda] = 10^{-36} \text{s}^{-1} \quad (31.22)$$

$$[\gamma] = \text{strain}^2 \quad (31.23)$$

$$[\Omega_{GW}] = 1 \quad (31.24)$$

$$[1/P_{1,2}^{HC}] = 10^{18} \text{Hz} \text{strain}^{-1} \text{count}^{-1} \quad (31.25)$$

$$[\tilde{Q}] := [\lambda][\gamma][\Omega_{GW}][\frac{1}{P_1^{HC}}][\frac{1}{P_2^{HC}}] s^3 = \text{count}^{-2} \quad (31.26)$$
31.1.9 Program StochasticOptimalFilterTest.c

Test suite for LALStochasticOptimalFilter().

Usage

./StochasticOptimalFilterTest [options]

Options:
- h  print usage message
- q  quiet: run silently
- v  verbose: print extra information
- d level  set lalDebugLevel to level
- n length  frequency series contain length points
- f fRef  set normalization reference frequency to fRef
- w filename  read gravitational-wave spectrum from file filename
- g filename  read overlap reduction function from file filename
- i filename  read first calibrated inverse noise PSD from file filename
- j filename  read second calibrated inverse noise PSD from file filename
- s filename  read first half-calibrated inverse noise PSD from file filename
- t filename  read second half-calibrated inverse noise PSD from file filename
- o filename  print optimal filter to file filename
- y  use normalization appropriate to heterodyned data

Description

This program tests the function LALStochasticOptimalFilter(), which generates a normalized optimal filter from a stochastic gravitational-wave background spectrum $h_0^2 \Omega_{GW}(f)$, an overlap reduction function $\gamma(f)$, and calibrated and half-calibrated noise power spectral densities $\{P_i^C(f), P_i^{HC}(f)\}$ for a pair of detectors.

First, it tests that the correct error codes (cf. Sec. 31.1) are generated for the following error conditions (tests in italics are not performed if LAL_NDEBUG is set, as the corresponding checks in the code are made using the ASSERT macro):

- null pointer to input structure
- null pointer to output series
- null pointer to overlap reduction function
- null pointer to gravitational-wave spectrum
- null pointer to first half-calibrated inverse noise PSD
- null pointer to second half-calibrated inverse noise PSD
- null pointer to data member of output series
- null pointer to data member of overlap reduction function
- null pointer to data member of gravitational-wave spectrum
- null pointer to data member of first half-calibrated inverse noise PSD
- null pointer to data member of second half-calibrated inverse noise PSD
- null pointer to data member of output series
- null pointer to data member of overlap reduction function
- null pointer to data member of gravitational-wave spectrum
- null pointer to data member of first half-calibrated inverse noise PSD
- null pointer to data member of second half-calibrated inverse noise PSD
- zero length
- negative frequency spacing
- zero frequency spacing
- negative start frequency
- length mismatch between overlap reduction function and output series
- length mismatch between overlap reduction function and gravitational-wave spectrum
- length mismatch between overlap reduction function and first half-calibrated inverse noise PSD
- length mismatch between overlap reduction function and second half-calibrated inverse noise PSD
- frequency spacing mismatch between overlap reduction function and gravitational-wave spectrum
- frequency spacing mismatch between overlap reduction function and first half-calibrated inverse noise PSD
- frequency spacing mismatch between overlap reduction function and second half-calibrated inverse noise PSD
- start frequency mismatch between overlap reduction function and gravitational-wave spectrum
- start frequency mismatch between overlap reduction function and first half-calibrated inverse noise PSD
- start frequency mismatch between overlap reduction function and second half-calibrated inverse noise PSD
- reference frequency less than frequency spacing
- reference frequency greater than maximum frequency

It then verifies that the correct optimal filter is generated [calculating the normalization with `LALStochasticOptimalFilterNormalization()` as described in Sec. ??, and checking the normalization by verifying that \((31.28)\) is satisfied] for each of the following simple test cases:

1. \(\gamma(f) = h_{100}^2 \Omega_{GW}(f) = P_1^C(f) = P_2^C(f) = P_1^{HC}(f) = P_2^{HC}(f) = 1;\) The expected optimal filter in this case is \(Q(f) \propto f^{-3}\).

2. \(\gamma(f) = P_1^C(f) = P_2^C(f) = P_1^{HC}(f) = P_2^{HC}(f) = 1; h_{100}^2 \Omega_{GW}(f) = f^3.\) The expected optimal filter in this case is \(Q(f) = \text{constant}\).

**Exit codes**

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOM</td>
<td>0</td>
<td>&quot;Nominal exit&quot;</td>
</tr>
<tr>
<td>ARG</td>
<td>1</td>
<td>&quot;Error parsing command-line arguments&quot;</td>
</tr>
<tr>
<td>CHK</td>
<td>2</td>
<td>&quot;Error checking failed to catch bad data&quot;</td>
</tr>
<tr>
<td>FLS</td>
<td>3</td>
<td>&quot;Incorrect answer for valid data&quot;</td>
</tr>
<tr>
<td>USE</td>
<td>4</td>
<td>&quot;Bad user-entered data&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants `STOCHASTICOPTIMALFILTERTESTC_EX<name>`, and the status descriptions in `STOCHASTICOPTIMALFILTERTESTC_MSGE<name>`. The source code with these messages is in `StochasticOptimalFilterTest.c` on line 1.231.
Uses

LALStochasticOptimalFilter()
LALCheckMemoryLeaks()
LALCReadFrequencySeries()
LALSCreateVector()
LALSDestroyVector()
LALCCreateVector()
LALCDestroyVector()
LALCHARCreateVector()
LALCHARDestroyVector()
LALUnitAsString()
LALUnitCompare()
getopt()
printf()
fprintf()
freopen()
fabs()

Notes

- No specific error checking is done on user-specified data. If `length` is missing, the resulting default will cause a bad data error. If `fRef` is unspecified, a default value of 1 Hz is used.

- The length of the user-provided series must be specified, even though it could in principle be deduced from the input file, because the data sequences must be allocated before the `LALCReadFrequencySeries()` function is called.

- If some, but not all, of the `filename` arguments are present, the user-specified data will be silently ignored.
### 31.1.10 Module StochasticOptimalFilterNormalization.c

Calculates the normalization factor for the optimal filter and the expected variance per unit time of the standard cross-correlation statistic.

#### Prototypes

```c
void LALStochasticOptimalFilterNormalization(
    LALStatus *status, 
    StochasticOptimalFilterNormalizationOutput *output, 
    const StochasticOptimalFilterNormalizationInput *input, 
    const StochasticOptimalFilterNormalizationParameters *parameters)
```

#### Description

As described in Section 31.1.8, the optimal filter for stochastic searches is defined as

\[
\tilde{Q}^C(f) = \lambda \frac{\gamma(f) \Omega_{GW}(f)}{|f|^3 P_1^C(f) P_2^C(f)}
\]

(31.27)

The normalization constant \( \lambda \) is chosen so that the expected mean value of the cross-correlation statistic is

\[
\mu = \frac{3H_0^2}{20\pi^2} T \frac{w_1w_2}{|f|^3} \int_{-\infty}^{\infty} df \frac{\gamma(f) \Omega_{GW}(f)}{|f|^3} \tilde{Q}^C(f) = \Omega_T
\]

(31.28)

where \( T \) is the integration time (cf. (31.1)), \( w_1 \) and \( w_2 \) are the functions used to window the data, and \( \Omega_T = \Omega_{GW}(f_R) \) is the overall strength of the stochastic background (see Sec. 31.1.16). This sets the value at

\[
\lambda = \frac{20\pi^2 \Omega_R}{3H_0^2 \frac{w_1w_2}{|f|^3}} \left( \int_{-\infty}^{\infty} df \frac{\gamma(f) \Omega_{GW}(f)}{|f|^3} \tilde{Q}^C(f) \right)^{-1}
\]

(31.29)

The same integral used to calculate \( \lambda \) also allows one to calculate the expected variance per unit integration time of the cross-correlation statistic, since

\[
\sigma^2 = \frac{4T}{3H_0^2 \frac{w_1w_2}{|f|^3}} \Omega_R \lambda \left( \int_{-\infty}^{\infty} df \frac{\gamma(f) \Omega_{GW}(f)}{|f|^3} \tilde{Q}^C(f) \right)^{-1}
\]

(31.30)

where we have used (31.27) to replace one of the two factors of \( \tilde{Q}^C(f) \) and (31.28) to replace the integral.

LALStochasticOptimalFilterNormalization() uses (31.29) to calculate the normalization constant \( \lambda \) and (31.30) to calculate the expected variance per unit time \( \sigma^2/T \) of the cross-correlation statistic.

#### Algorithm

The routine LALStochasticOptimalFilterNormalization() first uses (31.29) to find the normalization constant \( \lambda \) (the amplitude \( h_{100}^2 \Omega_R \) is found by logarithmic interpolation using the reference frequency \( f_R \) specified in the parameter structure and the input series representing \( h_{100}^2 \Omega_{GW}(f) \)).

The precise behavior of the normalization depends on the boolean parameter parameters->heterodyned, which indicates whether the filter is to be used on heterodyned data or not. In the case of heterodyned data, the integral is approximated by the sum

\[
\lambda \approx \frac{20\pi^2 \Omega_R}{3H_0^2 \frac{w_1w_2}{|f|^3}} \left( \delta f \sum_{k=0}^{N-1} (f_0 + k \delta f)^{-6} \frac{\gamma(k) \Omega_{GW}(k)^2}{P_1^C(k)P_2^C(k)} \right)^{-1}
\]

(31.31)
(Leaving out frequencies outside the band is equivalent to assuming one or both of the noise PSDs $P_{1,2}^C(f)$ blows up outside that range.)

In the case of non-heterodyned data with $f_0 = 0$, we calculate

$$\lambda \approx \frac{20\pi^2 \Omega_R}{3 H_0^2 \omega_1 \omega_2} \left( \delta f \frac{2 \Re \sum_{k=0}^{N-1} (k \delta f)^{-6} (\gamma[k] \Omega_{GW}[k])^2}{P_1^C[k] P_2^C[k]} \right)^{-1}$$  (31.32)

which includes negative frequencies as well. The difference between the two is because the cross-correlation statistic appearing in the definition (31.28) is the one calculated by StochasticHeterodynedCrossCorrelationStatistic() in the case of heterodyned and StochasticCrossCorrelationStatistic() in the case of non-heterodyned data.

Uses

LALUnitMultiply()
LALUnitRaise()
LALUnitCompare()

Notes

- The reference frequency $f_R$ must lie safely enough in the frequency range of the inputs to allow the value of $h_{100}^2 \Omega_R$ to be determined by interpolation.

- The implementation of the optimal filter function given here assumes a large observation time continuum-limit approximation. In this limit, the Dirichlet kernels (which appear in an exact expression for the standard cross-correlation statistic, when evaluated in discrete time [4]; see also the documentation for the module Dirichlet.c in the utilities package) may be replaced by Dirac delta functions.

- The units of the input series are checked for consistency; since $\mathbb{E}$

$$\langle \tilde{h}_1^C(f) \tilde{h}_2^C(f') \rangle = \frac{3H_0^2}{20\pi^2} \delta(f - f') \gamma(|f|) \Omega_{GW}(|f|)$$  (31.33)

and

$$\langle \tilde{h}_1(f) \tilde{h}_2(f') \rangle = \delta(f - f') P_1^C(f)$$  (31.34)

we demand that, up to powers of ten,

$$([\gamma][\Omega_{GW}])^2 = ([\tilde{h}_1][\tilde{h}_2][T]^{-2})^2 = [P_1^C][P_2^C][T]^{-2}$$  (31.35)

- This routine, like all those in the stochastic package, uses with single precision arithmetic, consistent with the accuracy of the continuous approximation for the intended number of data points. However, the limited dynamic range can pose a problem, especially in this routine, which uses numbers like $H_0^2$ which are far from unity when expressed in the standard SI units. To avoid this problem, the application of (31.29) takes the units $H_0$ of to be $10^{-18}$ s$^{-1}$ rather than s$^{-1}$. In these units $h_{100}H_0$ has a numerical value of 3.2407792903, and inclusion of $(h_{100}H_0)^2$ in an expression doesn’t risk single-precision overflow. When constructing the unit structures of its outputs, LALStochasticOptimalFilterNormalization() uses the power-of-ten feature of the LALUnit structure to account for the units of $H_0$.

- The expected units for the inputs and outputs of this function are as follows (although the actual output units will be constructed from the input units):

$$([\gamma][\Omega_{GW}])^2 = ([\tilde{h}_1][\tilde{h}_2][T]^{-2})^2 = [P_1^C][P_2^C][T]^{-2}$$  (31.36)

$$[\gamma] = \text{strain}^2$$  (31.37)

$$[\Omega_{GW}] = 1$$  (31.38)

$$[1/P_{1,2}^C] = 10^{36} \text{Hz\:strain}^{-2}$$  (31.39)

$$[\lambda] = 10^{36} [P_1^C][P_2^C][\gamma]^{-2} [\Omega_{GW}]^{-1} \text{s}^{-3} = 10^{-36} \text{s}^{-1}$$  (31.40)

$$[\sigma^2/T] = 10^{36} [\lambda][\Omega_{GW}] \text{s}^2 = s$$  (31.41)
31.1.11 Program StochasticOptimalFilterNormalizationTest.c

Test suite for LALStochasticOptimalFilterNormalization().

Usage

./StochasticOptimalFilterNormalizationTest [options]

Options:
- h: print usage message
- q: quiet: run silently
- v: verbose: print extra information
- d level: set lalDebugLevel to level
- n length: frequency series contain length points
- f fRef: set normalization reference frequency to fRef
- w filename: read gravitational-wave spectrum from file filename
- g filename: read overlap reduction function from file filename
- i filename: read first inverse noise PSD from file filename
- j filename: read second inverse noise PSD from file filename
- y: use normalization appropriate to heterodyned data

Description

This program tests the function LALStochasticOptimalFilterNormalization(), which calculates the normalization factor for an optimal filter and the expected variance per unit time of the cross-correlation statistic, given a stochastic gravitational-wave background spectrum $h_{100}^2 \Omega_{GW}(f)$, an overlap reduction function $\gamma(f)$, and unwhitened noise power spectral densities $\{P_i(f)\}$ for a pair of detectors.

First, it tests that the correct error codes (cf. Sec. 31.1) are generated for the following error conditions (tests in italics are not performed if LAL_NDEBUG is set, as the corresponding checks in the code are made using the ASSERT macro):

- null pointer to input structure
- null pointer to output structure
- null pointer to overlap reduction function
- null pointer to gravitational-wave spectrum
- null pointer to first inverse noise PSD
- null pointer to second inverse noise PSD
- null pointer to data member of overlap reduction function
- null pointer to data member of gravitational-wave spectrum
- null pointer to data member of first inverse noise PSD
- null pointer to data member of second inverse noise PSD
- null pointer to data member of data member of overlap reduction function
- null pointer to data member of data member of gravitational-wave spectrum
- null pointer to data member of data member of first inverse noise PSD
- null pointer to data member of data member of second inverse noise PSD
- zero length
- negative frequency spacing
- zero frequency spacing
- negative start frequency
31.1. Header StochasticCrossCorrelation.h

- length mismatch between overlap reduction function and gravitational-wave spectrum
- length mismatch between overlap reduction function and first inverse noise PSD
- length mismatch between overlap reduction function and second inverse noise PSD
- frequency spacing mismatch between overlap reduction function and gravitational-wave spectrum
- frequency spacing mismatch between overlap reduction function and first inverse noise PSD
- frequency spacing mismatch between overlap reduction function and second inverse noise PSD
- start frequency mismatch between overlap reduction function and gravitational-wave spectrum
- start frequency mismatch between overlap reduction function and first inverse noise PSD
- start frequency mismatch between overlap reduction function and second inverse noise PSD
- reference frequency less than frequency spacing
- reference frequency greater than maximum frequency

It then verifies that the correct optimal filter is generated (checking the normalization by verifying that (31.28) is satisfied) for each of the following simple test cases (with \( f_0 = 0, f_R = 1, \delta f = 1 \) and \( N = 8 \)):

1. \( \gamma(f) = h_{100}^2 \Omega_{GW}(f) = P_1^C(f) = P_2^C(f) = 1 \); The expected results in this case are \( \lambda = 3.079042427975, \sigma^2 T^{-1} = 4.822422518205 \)

2. \( \gamma(f) = P_1^C(f) = P_2^C(f) = 1; h_{100}^2 \Omega_{GW}(f) = f^3 \). The expected results in this case are \( \lambda = .4474881517327, \sigma^2 T^{-1} = .700859760794 \)

Exit codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOM</td>
<td>0</td>
<td>&quot;Nominal exit&quot;</td>
</tr>
<tr>
<td>ARG</td>
<td>1</td>
<td>&quot;Error parsing command-line arguments&quot;</td>
</tr>
<tr>
<td>CHK</td>
<td>2</td>
<td>&quot;Error checking failed to catch bad data&quot;</td>
</tr>
<tr>
<td>FLS</td>
<td>3</td>
<td>&quot;Incorrect answer for valid data&quot;</td>
</tr>
<tr>
<td>USE</td>
<td>4</td>
<td>&quot;Bad user-entered data&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants STOCHASTICOPTIMALFILTERNORMALIZATIONTESTC_E<name>, and the status descriptions in STOCHASTICOPTIMALFILTERNORMALIZATIONTESTC_MSGE<name>. The source code with these messages is in StochasticOptimalFilterNormalizationTest.c on line 1.226.

Uses

LALStochasticOptimalFilterNormalization()
LALCheckMemoryLeaks()
LALReadFrequencySeries()
LALSCreateVector()
LALSDestroyVector()
LALCCreateVector()
LALCDestroyVector()
LALCHARCreateVector()
LALCHARDestroyVector()
LALUnitAsString()
LALUnitCompare()
getopt()
printf()
fprintf()
freopen()
fabs()
Notes

• No specific error checking is done on user-specified data. If length is missing, the resulting default will cause a bad data error. If fRef is unspecified, a default value of 1 Hz is used.

• The length of the user-provided series must be specified, even though it could in principle be deduced from the input file, because the data sequences must be allocated before the LALCReadFrequencySeries() function is called.

• If some, but not all, of the filename arguments are present, the user-specified data will be silently ignored.
31.1.12 Module StochasticInverseNoise.c

Calculates the values of the calibrated and half-calibrated inverse noise power spectra from the uncalibrated noise power spectrum and the frequency-domain instrument response function.

Prototypes

```c
void LALStochasticInverseNoise(
    LALStatus *status,
    StochasticInverseNoiseOutput *output,
    const StochasticInverseNoiseInput *input )
```

Description

As described in Sec. 31.1.8, the most convenient combinations of the noise $P(f)$ (defined by $\langle h(f)h(f')^* \rangle = \delta(f - f')P(f)$) and instrument response $\tilde{R}(f) = h(f)/h(f)$ to use in constructing an optimal filter are the inverse half-calibrated power spectral density

$$\frac{1}{P_{HC}(f)} = \frac{1}{\tilde{R}(f)P_C(f)} = \frac{\tilde{R}(f)^*}{P(f)}$$  \hspace{1cm} (31.42)

and the inverse calibrated PSD

$$\frac{1}{P_C(f)} = \frac{|\tilde{R}(f)|^2}{P(f)}$$  \hspace{1cm} (31.43)

The function `LALStochasticInverseNoise()` takes in a `REAL4FrequencySeries` describing the uncalibrated PSD $P(f)$ along with a `COMPLEX8FrequencySeries` describing the frequency-domain response $\tilde{R}(f)$, and outputs a `REAL4FrequencySeries` describing the calibrated inverse PSD $1/P_C(f)$ along with a `COMPLEX8FrequencySeries` describing the half-calibrated inverse PSD $1/P_{HC}(f)$.

Algorithm

The output series are filled according to a straightforward implementation of (31.42)-(31.43). The DC components, if included in the series, are set to zero.

Uses

`LALUnitRaise()`
`LALUnitMultiply()`
`strncpy()`

Notes

- Note that although $P_C(f)$ and $P(f)$ are real, $P_{HC}(f)$ is complex.
- The output units are constructed by combining the input units, but under normal circumstances the units will be as follows:

$$[P] = \text{count}^2 \text{Hz}^{-1}$$  \hspace{1cm} (31.44)
$$[\tilde{R}] = 10^{18} \text{strain}^{-1} \text{count}$$  \hspace{1cm} (31.45)
$$[1/P_C] := [\tilde{R}]^2[P] = 10^{36} \text{Hz} \text{strain}^{-2}$$  \hspace{1cm} (31.46)
$$[1/P_{HC}] := [\tilde{R}][P] = 10^{18} \text{Hz} \text{strain}^{-1} \text{count}^{-1}$$  \hspace{1cm} (31.47)
31.1.13 Program StochasticInverseNoiseTest.c

Test suite for LALStochasticInverseNoise().

Usage

./StochasticInverseNoiseTest [options]
Options:
- h print usage message
- q quiet: run silently
- v verbose: print extra information
- d level set lalDebugLevel to level
- n length frequency series contain length points
- w filename read uncalibrated noise PSD from file filename
- f filename read response function from file filename
- u filename print calibrated inverse noise PSD to file filename
- m filename print half-calibrated inverse noise PSD to file filename

Description

This program tests the function LALStochasticInverseNoise(), which outputs an uncalibrated and "half-calibrated" inverse noise spectra from a uncalibrated data stream and a response function.

First, it tests that the correct error codes (cf. Sec. 31.1) are generated for the following error conditions (tests in italics are not performed if LAL_NDEBUG is set, as the corresponding checks in the code are made using the ASSERT macro):

- null pointer to output structure
- null pointer to input structure
- null pointer to uncalibrated noise
- null pointer to response function
- null pointer to calibrated inverse noise
- null pointer to half-calibrated inverse noise
- null pointer to data member of uncalibrated noise
- null pointer to data member of response function
- null pointer to data member of calibrated inverse noise
- null pointer to data member of half-calibrated inverse noise
- null pointer to data member of data member of uncalibrated noise
- null pointer to data member of data member of response function
- null pointer to data member of data member of calibrated inverse noise
- null pointer to data member of data member of half-calibrated inverse noise
- zero length
- negative frequency spacing
- zero frequency spacing
- negative start frequency
- length mismatch between uncalibrated noise and response function
- length mismatch between uncalibrated noise and calibrated inverse noise
• length mismatch between uncalibrated noise and half-calibrated inverse noise
• frequency spacing mismatch between uncalibrated noise and response function
• start frequency mismatch between uncalibrated noise and response function

It then verifies that the correct uncalibrated and half-calibrated inverse noise are generated for a simple test case:

1. $\tilde{R}(f) = (1 + i)f^2$, $P(f) = f^3$. The expected results are $1/P_C(f) = 2f$, $1/P_{HC}(f) = (1 - i)f^{-1}$.

For each successful test (both of these valid data and the invalid ones described above), it prints “PASS” to standard output; if a test fails, it prints “FAIL”.

If the four `filename` arguments are present, it also calculates a spectrum based on user-specified data and it prints the noise spectra to the files specified by the user.

Exit codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOM</td>
<td>0</td>
<td>&quot;Nominal exit&quot;</td>
</tr>
<tr>
<td>ARG</td>
<td>1</td>
<td>&quot;Error parsing command-line arguments&quot;</td>
</tr>
<tr>
<td>CHK</td>
<td>2</td>
<td>&quot;Error checking failed to catch bad data&quot;</td>
</tr>
<tr>
<td>FLS</td>
<td>3</td>
<td>&quot;Incorrect answer for valid data&quot;</td>
</tr>
<tr>
<td>USE</td>
<td>4</td>
<td>&quot;Bad user-entered data&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants `STOCHASTICINVERSENOISETESTC_EX<name>`, and the status descriptions in `STOCHASTICINVERSENOISETESTC_MSG<name>`. The source code with these messages is in `StochasticInverseNoiseTest.c` on line l.203.

Uses

- `getopt()`
- `LALStochasticInverseNoise()`
- `LALSCreateVector()`
- `LALCCreateVector()`
- `LALSDestroyVector()`
- `LALCDestroyVector()`
- `LALSReadFrequencySeries()`
- `LALCReadFrequencySeries()`
- `LALSPrintFrequencySeries()`
- `LALCPrintFrequencySeries()`
- `LALUnitAsString()`
- `LALUnitCompare()`
- `LALCheckMemoryLeaks()`

Notes

• No specific error checking is done on user-specified data. If the `length` argument missing, the resulting defaults will cause a bad data error.

• The length of the user-provided series must be specified, even though it could in principle be deduced from the input file, because the data sequences must be allocated before the `LALCReadFrequencySeries()` function is called.

• If one `filename` argument, but not both, is present, the user-specified data will be silently ignored.
31.1.14 Module StochasticOmegaGW.c

Generates a frequency series containing a simple power law spectrum.

Prototypes

```c
void LALStochasticOmegaGW(
    LALStatus *status,
    REAL4FrequencySeries *output,
    const StochasticOmegaGWParameters *parameters)
```

Description

The strength of a stochastic gravitational wave background is defined as

\[
\Omega_{GW}(f) := \frac{f}{\rho_{\text{crit}}} \frac{d\rho_{GW}}{df} ,
\]

where

\[
\rho_{\text{crit}} = \frac{3H_0^2c^2}{8\pi G}
\]

is the critical density needed to close the universe. Since the value of \( \rho_{\text{crit}} \) depends on the observed value of the Hubble constant \( H_0 \), it is traditional to remove this experimental uncertainty from the definition by working with \( h_{100}^2 \Omega_{GW} \), where \( h_{100} \) is the Hubble constant divided by 100 km s\(^{-1}\) Mpc\(^{-1}\).

\( \text{LALStochasticOmegaGW()} \) generates a simple power law spectrum

\[
h_{100}^2\Omega_{GW}(f) = \Omega_R \left( \frac{f}{f_R} \right)^\alpha
\]

The parameter `parameters.omegaRef` specifies the amplitude \( h_{100}^2 \Omega_R \) of the spectrum. This is simply defined as the value of \( h_{100}^2 \Omega_{GW}(f) \) at the reference frequency \( f_R \) which is specified in `parameters.omegaRef`.

Algorithm

\( \text{LALStochasticOmegaGW()} \) treats the constant spectrum \( \alpha = 0 \) as a special case, and simply sets every element of the output series to \( \Omega_R \).

If \( \alpha \neq 0 \), the DC value `output->data->data[0]` is set to 0 or `LAL_REAL4_MAX`, depending on whether \( \alpha \) is positive or negative, respectively.

The output units are set to be dimensionless.

Uses

`LAL_REAL4_MAX`

Notes

- This routine will eventually be generalized to include “broken” power law spectra

\[
h_{100}^2\Omega_{GW} = \begin{cases} h_{100}^2\Omega_1 f^{\alpha_1} & f \leq f_c \\ h_{100}^2\Omega_2 f^{\alpha_2} & f \geq f_c \end{cases}
\]
31.1.15 Program StochasticOmegaGWTest.c

A program to test LALStochasticOmegaGW().

Usage

```bash
./StochasticOmegaGWTest [options]
```

Options:

- `h` print usage message
- `q` quiet: run silently
- `v` verbose: print extra information
- `d level` set lalDebugLevel to level
- `a alpha` set power law exponent to alpha
- `O omegaRef` set amplitude to omegaRef
- `F fRef` set normalization reference frequency to fRef
- `f f0` set start frequency to f0
- `e deltaF` set frequency spacing to deltaF
- `n length` set number of points in frequency series to length
- `o filename` print gravitational-wave spectrum to file filename

Description

This program tests the function LALStochasticOmegaGW(), which outputs a power law spectrum

\[ h_{100}^{2} \Omega_{GW}(f) = \Omega_{R} \left( \frac{f}{f_{R}} \right)^{\alpha} \]  \tag{31.52} \]

First, it tests that the correct error codes (cf. Sec. 31.1) are generated for the following error conditions (tests in *italics* are not performed if LAL_NDEBUG is set, as the corresponding checks in the code are made using the ASSERT macro):

- null pointer to output series
- null pointer to parameter structure
- null pointer to data member of output series
- null pointer to data member of data member of output series
- zero length parameter
- negative frequency spacing
- zero frequency spacing
- mismatch between length of output series and length parameter
- zero reference frequency \( f_{R} \)
- negative amplitude parameter \( \Omega_{R} \)
- zero amplitude parameter \( \Omega_{R} \)

It then verifies that the correct frequency series are generated for two simple test cases: \( \alpha = 2.5 \) and \( \alpha = 0 \). For each successful test (both of these valid data and the invalid ones described above), it prints “PASS” to standard output; if a test fails, it prints “FAIL”.

If the `filename` argument is present, it also calculates a spectrum based on user-specified data. Figure 31.2 illustrates the output of the command with the following arguments:

```
StochasticOmegaGWTest -e 1 -n 1000 -F 100 -O 1e-6 -a 2 -o OmegaGW.dat
```
Stochastic Gravitational Wave Background Strength $\Omega_{GW}(f)$

![Graph of Stochastic Gravitational Wave Background Strength](image)

Figure 31.2: A quadratic stochastic gravitational-wave background spectrum.

Exit codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOM</td>
<td>0</td>
<td>&quot;Nominal exit&quot;</td>
</tr>
<tr>
<td>ARG</td>
<td>1</td>
<td>&quot;Error parsing command-line arguments&quot;</td>
</tr>
<tr>
<td>CHK</td>
<td>2</td>
<td>&quot;Error checking failed to catch bad data&quot;</td>
</tr>
<tr>
<td>FLS</td>
<td>3</td>
<td>&quot;Incorrect answer for valid data&quot;</td>
</tr>
<tr>
<td>USE</td>
<td>4</td>
<td>&quot;Bad user-entered data&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants STOCHASTICOMEGAGWTESTC_E<name>, and the status descriptions in STOCHASTICOMEGAGWTESTC_MSGE<name>. The source code with these messages is in StochasticOmegaGWTest.c on line 1.216.

Uses

- lalDebugLevel
- getopt()
- LALSCreateVector()
- LALStochasticOmegaGW()
- LALSPrintFrequencySeries
- LALSDestroyVector()
- LALCheckMemoryLeaks()

Notes

- No specific error checking is done on user-specified data. If deltaF or length are missing, the resulting defaults will cause a bad data error. If other arguments are unspecified, the following defaults are used:
  
  $\alpha = 0$
  
  $f_0 = 0$
fRef 1 Hz
omegaRef 1

- The routine `LALStochasticOmegaGW()` will eventually be generalized to include “broken” power law spectra

\[
\begin{align*}
 h_{100}^2 \Omega_{GW} &= \begin{cases} 
 \Omega_1 f^{\alpha_1} & f \leq f_c \\
 \Omega_2 f^{\alpha_2} & f \geq f_c 
\end{cases} 
\end{align*}
\]

(31.53)
31.1.16 Module `OverlapReductionFunction.c`

Calculates the values of the overlap reduction function for a pair of gravitational wave detectors.

Prototypes

```c
void LALOverlapReductionFunction(
    LALStatus *status,
    REAL4FrequencySeries *output,
    const LALDetectorPair *detectors,
    const OverlapReductionFunctionParameters *parameters)
```

Description

`LALOverlapReductionFunction()` calculates the values of the overlap reduction:

\[
\gamma(f) := \frac{5}{8\pi} \sum_A \int_{S^2} d\hat{\Omega} \, e^{i2\pi f\hat{\Omega} \cdot \Delta \vec{x}/c} \, F^A_1(\hat{\Omega}) F^A_2(\hat{\Omega}),
\]

where \(\hat{\Omega}\) is a unit vector specifying a direction on the two-sphere, \(\Delta \vec{x} := \vec{x}_1 - \vec{x}_2\) is the separation vector between the two detectors, and

\[
F^A_i(\hat{\Omega}) := e^A_{ab}(\hat{\Omega}) \, d^{ab}_i
\]

is the response of the \(i\)th detector \((i = 1, 2)\) to the \(A = +, \times\) polarization. Here \(d^{ab}_i\) is the response tensor for the \(i\)th detector, which relates the “strain” \(h\) measured by the detector to the metric perturbation \(h_{ab}\) due to gravitational waves by

\[
h = d^{ab}_{i}h_{ab}
\]

The Cartesian components of \(d^{ab}_{i}\) are constant in an earth-fixed rotating coordinate system. \(|e^A_{ab}(\hat{\Omega})|A = +, \times\) are the spin-2 polarization tensors for the “plus” and “cross” polarization states, normalized so that \(e^A_{ab}e^{Bab} = 2\delta^{AB}\). With this definition,

\[
\gamma(f) = d^{ab}_{1}d^{cd}_{2} \frac{5}{4\pi} \int_{S^2} d\hat{\Omega} \, e^{i2\pi f\hat{\Omega} \cdot \Delta \vec{x}/c} \, P^{ab}_{cd}(\hat{\Omega})
\]

where \(P^{ab}_{cd}(\hat{\Omega})\) is a projection operator onto the space of symmetric, traceless second-rank tensors orthogonal to \(\hat{\Omega}\).

The overlap reduction function for a pair of identical detectors is a maximum when they are coincident and coaligned; it decreases when the detectors are shifted apart (so there is a phase shift between the signals in the two detectors), or rotated out of coalignment (so the detectors are sensitive to different polarizations). The overlap reduction function arises naturally when calculating the cross-correlated signal due to an isotropic and unpolarized stochastic gravitational-wave background.

Given a choice of two detector sites, a frequency spacing \(\delta f\), a start frequency \(f_0\), and the number of desired values \(N\), `LALOverlapReductionFunction()` calculates the values of \(\gamma(f)\) at the discrete frequencies \(f_i = f_0 + i\delta f, i = 0, 1, \ldots, N - 1\).

Algorithm

As shown in Appendix B of [5] and Sec. III.B of [3], the overlap reduction function can be written in closed form in terms of the traceless response tensor \(D_{ab} = d_{ab} - \delta_{ab}d^c_c/3\) as sum of three spherical Bessel functions:

\[
\gamma(f) = \rho_1(\alpha) \, D_1^{ab} D_2^{ab} + \rho_2(\alpha) \, D_1^{ab} D_2^{c} \, s_b s_c + \rho_3(\alpha) \, D_1^{ab} D_2^{cd} \, s_a s_b s_c s_d,
\]

where

\[
\begin{bmatrix}
\rho_1 \\
\rho_2 \\
\rho_3
\end{bmatrix} = \frac{1}{2\alpha^2} \begin{bmatrix}
10\alpha^2 & -20\alpha & 10 \\
-20\alpha^2 & 80\alpha & -100 \\
5\alpha^2 & -50\alpha & 175
\end{bmatrix} \begin{bmatrix}
j_0 \\
j_1 \\
j_2
\end{bmatrix},
\]

(31.54)
$j_0$, $j_1$, and $j_2$ are the standard spherical Bessel functions:

\[
\begin{align*}
    j_0(\alpha) &= \frac{\sin \alpha}{\alpha}, \\
    j_1(\alpha) &= \frac{\sin \alpha}{\alpha^2} - \frac{\cos \alpha}{\alpha}, \\
    j_2(\alpha) &= 3 \frac{\sin \alpha}{\alpha^3} - 3 \frac{\cos \alpha}{\alpha^2} - \frac{\sin \alpha}{\alpha}.
\end{align*}
\]

(31.60)

$\vec{s}$ is a unit vector pointing in the direction of $\Delta \vec{x} := \vec{x}_1 - \vec{x}_2$, and $\alpha := 2\pi f |\Delta \vec{x}| / c$.

LALOverlapReductionFunction() calculates the values of $\gamma(f)$ as follows:

1. Gets the locations and response tensors for the two detectors from the LALDetector structures in the input.
2. Constructs the traceless parts $D_{iab}$ of the two detector response tensors and finds the distance $|\Delta \vec{x}|$ and direction $\vec{s}$ between the sites.
3. Calculates the frequency-independent coefficients $D_{1}^{ab}D_{2a}, D_{1}^{ab}D_{2b} c_{sb}c_{sc},$ and $D_{1}^{ab}D_{2c}^{cd} s_{a}s_{b}s_{c}s_{d}$ that appear in Eq. (31.58).
4. Calculates $\gamma(f)$ at each discrete frequency $f_i := f_0 + i\Delta f, i = 0, 1, \ldots, N - 1$, using the power series expansion

\[
\begin{align*}
    j_0(\alpha) &= 1 - \frac{\alpha^2}{6} + \frac{\alpha^4}{120} + O(\alpha^6) \\
    j_1(\alpha) &= 1 - \frac{\alpha^2}{3} + \frac{\alpha^4}{840} + O(\alpha^6) \\
    j_2(\alpha) &= 1 - \frac{\alpha^2}{15} + \frac{\alpha^4}{7560} + O(\alpha^6).
\end{align*}
\]

(31.61)

for the spherical Bessel functions $j_0(\alpha_i), j_1(\alpha_i), j_2(\alpha_i)$ when $\alpha_i = 2\pi f_i |\Delta \vec{x}| / c < 0.01$.

Uses

LALUnitRaise()

\(\sin()\)

\(\cos()\)

\(\sqrt{()}\)

\(\text{strncpy()}\)

Notes

- The $\gamma(f)$ here is related to the unnormalized $\Gamma(f)$ defined by Maggiore [6, 7] by $\gamma(f) = \frac{5}{2} \Gamma(f)$. This normalization, which agrees with the literature [5] on interferometers, is chosen so that $\gamma(f) \equiv 1$ for a pair of coincident, collimated interferometers with perpendicular arms. It means that, for combinations other than a pair of interferometers, our $\gamma(f)$ is not equal to the generalization of $\gamma(f)$ defined by Maggiore, whose relationship to $\Gamma(f)$ depends on the type of detector. Defining $\gamma(f)$ as we do allows us to use the formulæ from, e.g., [3], irrespective of the detector type in question.

- While $\gamma(f)$ is usually considered to be dimensionless, this routine attaches to it units of strain$^2$. This is because it contains two powers of the response tensor $d_{iab}$, which converts the dimensionless metric perturbation $h_{ab}$ to $h = h_{ab}d^{ab}$, which has units of strain.
31.1.17 Program OverlapReductionFunctionTest.c

A program to test LALOverlapReductionFunction().

Usage

```
./OverlapReductionFunctionTest [options]
```

Options:

- `-h` print usage message
- `-q` quiet: run silently
- `-v` verbose: print extra information
- `-d level` set lalDebugLevel to level
- `-s siteID1` calculate overlap red fcn for site siteID1
- `-t siteID2` with site siteID2
- `-f f0` set start frequency to f0
- `-e deltaF` set frequency spacing to deltaF
- `-n length` set number of points in frequency series to length
- `-o filename` print overlap reduction function to file filename

Description

This program tests the function LALOverlapReductionFunction(), which calculates the overlap reduction \((31.54)\) for a pair of gravitational wave detectors.

First, it tests that the correct error codes (cf. Sec. 31.1) are generated for the following error conditions (tests in italics are not performed if LAL_NDEBUG is set, as the corresponding checks in the code are made using the ASSERT macro):

- null pointer to parameter structure
- zero length parameter
- negative frequency spacing
- zero frequency spacing
- null pointer to output series
- null pointer to data member of output series
- mismatch between length of output series and length parameter
- null pointer to data member of data member of output series
- negative start frequency
- non-symmetric response tensor

It then verifies that the correct frequency series are generated for some simple test cases:

1. coincident, co-aligned interferometers: \(\gamma(f) = 1\)

2. co-aligned interferometers lying parallel to the \(x-y\) plane separated only in \(z\): \(\gamma(f) = \frac{5}{2} \alpha^{-1} \sin \alpha(1 - 3\alpha^{-2} + 3\alpha^{-5}) + \frac{5}{2} \alpha^{-1} \cos \alpha(2 - 3\alpha^{-2})\), where \(\alpha = 2\pi f |\Delta \vec{x}| / c\)

3. completely misaligned interferometers lying parallel to the \(x-y\) plane separated only in \(z\): \(\gamma(f) = 0\).

If the `filename` argument is present, it also calculates a spectrum based on user-specified data. Figure \[31.3\]-\[31.12\] illustrates the output of the overlap reduction function with the corresponding command arguments of each detector.
Figure 31.3: Overlap reduction function for the LHO–LLO pair. This was generated by the command

```
OverlapReductionFunctionTest -e 1 -n 1000 -s 0 -t 1 -o LHOLLLOOverlap.dat
```

Figure 31.4: Overlap reduction function for the LHO–VIRGO pair. This was generated by the command

```
OverlapReductionFunctionTest -e 1 -n 1000 -s 0 -t 2 -o LHOVIRGOOverlap.dat
```

Exit codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOM</td>
<td>0</td>
<td>&quot;Nominal exit&quot;</td>
</tr>
<tr>
<td>ARG</td>
<td>1</td>
<td>&quot;Error parsing command-line arguments&quot;</td>
</tr>
<tr>
<td>CHK</td>
<td>2</td>
<td>&quot;Error checking failed to catch bad data&quot;</td>
</tr>
<tr>
<td>FLS</td>
<td>3</td>
<td>&quot;Incorrect answer for valid data&quot;</td>
</tr>
<tr>
<td>USE</td>
<td>4</td>
<td>&quot;Bad user-entered data&quot;</td>
</tr>
</tbody>
</table>
Figure 31.5: Overlap reduction function for the LHO–GEO600 pair. This was generated by the command
OverlapReductionFunctionTest -e 1 -n 1000 -s 0 -t 3 -o LHOGEO6000Overlap.dat

Figure 31.6: Overlap reduction function for the LHO–TAMA300 pair. This was generated by the command
OverlapReductionFunctionTest -e 1 -n 1000 -s 0 -t 4 -o LHOTAMA300Overlap.dat

The status codes in the table above are stored in the constants OVERLAPREDUCTIONFUNCTIONTESTC_E<name>,
and the status descriptions in OVERLAPREDUCTIONFUNCTIONTESTC_MSGE<name>. The source code with these
messages is in OverlapReductionFunctionTest.c on line 1.315.

Uses
lalDebugLevel
Figure 31.7: Overlap reduction function for the LLO–VIRGO pair. This was generated by the command
OverlapReductionFunctionTest -e 1 -n 1000 -s 1 -t 2 -o LLOVIRGOOverlap.dat

Figure 31.8: Overlap reduction function for the LLO–GEO600 pair. This was generated by the command
OverlapReductionFunctionTest -e 1 -n 1000 -s 1 -t 3 -o LLOGEO600Overlap.dat

getopt()
LALSCreateVector()
LALOverlapReductionFunction()
LALSPrintFrequencySeries()
LALSDestroyVector()
LALCheckMemoryLeaks()
compute the energy density spectrum of stochastic backgrounds produced by cosmological population of astrophysical sources.

**Synopsis**
Figure 31.11: Overlap reduction function for the VIRGO–TAMA300 pair. This was generated by the command

```
OverlapReductionFunctionTest -e 1 -n 1000 -s 2 -t 4 -o VIRGOTAMA300Overlap.dat
```

Figure 31.12: Overlap reduction function for the GEO600–TAMA300 pair. This was generated by the command

```
OverlapReductionFunctionTest -e 1 -n 1000 -s 3 -t 4 -o GEO600TAMA300Overlap.dat
```

```c
#include <lal/AstroOmega.h>
```

### Error conditions

The errors that may occur in this module are integration errors already defined in Integrate.h
Structures

These are function pointers corresponding to the spectral energy density of a single source.

typedef void (REAL8LALSDensity) (REAL8 *output, REAL8 input);

These are input structures corresponding to the model parameters (the cosmological model parameters and the source model parameters)

```c
typedef struct
    tagAstroOmegaCosmoParams
{
    REAL8 ho; Hubble parameter
    REAL8 density_matter; density parameter of matter
    REAL8 density_vacuum; density parameter of vacuum
    REAL8 density_k; density parameter of curvature
}
AstroOmegaCosmoParams;
```

cosmological model parameters:

typedef struct
tagAstroOmegaSourceParams
{
    REAL8LALSDensity *SDensitySource; single spectral energy density
    REAL8 numax; frequency cutoff in the source frame
    REAL8 lambda; mass fraction of source progenitors expressed in inverse solar masses.
}
AstroOmegaSourceParams;

source parameters:

model parameters (cosmological + source)

typedef struct
    tagAstroOmegaParams
{
    AstroOmegaCosmoParams cosmoparams;
    AstroOmegaSourceParams sourceparams;
    void *extraparams;
}
AstroOmegaParams;
Module AstroOmega.c

[compute the energy density spectrum of stochastic backgrounds produced by cosmological population of astrophysical sources]

Prototypes

Description

The function of this module computes the energy density parameter $\Omega_{gw}(\nu_o)$ for a given source and a given cosmological model. The spectral properties of the stochastic background are characterized by the dimensionless parameter

$$\Omega_{gw} = \frac{1}{\rho_c} \frac{d\rho_{gw}}{d\log \nu_o}$$

(31.64)

where $\rho_{gw}$ is the gravitational energy density, $\nu_o$ the frequency in the observer frame and $\rho_c = \frac{3H_o^2}{8\pi G}$

(31.65)

the critical energy density to close the Universe today.

For a given cosmological model, a stochastic background of astrophysical origin is fully determined by the source formation rate as a function of redshift, $dR(z)$ and the gravitational spectral density of a single source, $\frac{dE_{gw}}{d\nu}$. Following Ferrari et al. (1999):

$$\Omega_{gw}(\nu_o) = \frac{1}{c^3 \rho_c} \nu_o F_{\nu_o}$$

(31.66)

where

$$F_{\nu_o} = \int_0^{z_{\text{max}}} \nu_o dR(z)$$

(31.67)

is the gravitational wave flux at the frequency $\nu_o$ integrated over all the sources. The gravitational flux of a single source located between $z, z+dz$ is:

$$f_{\nu_o} = \frac{1}{4\pi d_L^2} \frac{dE_{gw}}{d\nu} (1 + z)$$

(31.68)

where $d_L = (1 + z)r$ is the distance luminosity and $\nu = (1 + z)\nu_o$ the frequency in the source frame. The event rate between $z, z+dz$ as observed in our frame is given by:

$$dR(z) = \lambda_p \frac{R_c(z)}{1 + z} \frac{dV}{dz} dz$$

(31.69)

where $R_c(z)$ is the cosmic star formation rate and $\lambda_p$ is the mass fraction of the source progenitors. The term $(1 + z)$ in the denominator accounts for the time dilatation by cosmic expansion of the observed rate. The element of the comoving volume is

$$dV = 4\pi r^2 \frac{c}{H_o E(\Omega_i, z)} \frac{dz}{dz}$$

(31.70)

where the function $E(\Omega_i, z)$ is defined by the equation:

$$E(\Omega_i, z) = \left[ \Omega_m (1 + z)^3 + \Omega_v \right]^{1/2}$$

(31.71)

where $\Omega_m$ and $\Omega_v$ are respectively the density parameters due to matter (baryonic and non-baryonic) and the vacuum. The cosmic star formation rate is computed as:

$$R_c(z) = R_{SF2}(z) \frac{h_0}{0.65} \frac{E(\Omega_i, z)}{(1 + z)^{3/2}} M_\odot \text{yr}^{-1} \text{Mpc}^{-3}$$

(31.72)

where

$$R_{SF2}(z) = \frac{0.15 \exp^{3.4z}}{(22 + \exp^{3.4z})} M_\odot \text{yr}^{-1} \text{Mpc}^{-3}$$

(31.73)
is the cosmic star formation rate (Madau & Porciani, 2001) in a matter-dominated universe ($\Omega_m = 1$) with $H_0 = 65\, \text{km s}^{-1}\text{Mpc}^{-1}$. Combining the previous equations one obtains:

$$\Omega_{gw}(\nu_0) = \frac{8\pi G}{3c^2 H_0^2} \lambda_p \nu_o \int_0^{z_{sup}} \frac{dE_{gw}}{d\nu} \frac{R_c(z)}{E(z)(1+z)^2} dz$$  \hspace{1cm} (31.74)

The upper limit of the integral is determined by the cutoff frequency in the source frame, $\nu_{sup}$, as:

$$z_{sup} = \frac{\nu_{sup}}{\nu_o} - 1$$  \hspace{1cm} (31.75)

Note that we are limited to $z < 5$ since the cosmic star formation rate is not modeled above. This restriction is without significant consequence since objects with $z > 5$ contribute very little to the integrated signal.

**Operating Instructions**

the following program shows how to use the function LALAstroOmega

```c
#include <stdio.h>
#include <math.h>
#include <lal/LALConfig.h>
#include <lal/LALStdlib.h>
#include <lal/Integrate.h>
#include <lal/AstroOmega.h>

NRCSID (ASTROOMEGATESTC, "$Id: AstroOmega.c,v 1.6 2007/06/08 14:41:54 bema Exp $");

//define here the spectral energy density of a single source
static void SDensity (REAL8 *dEgw, REAL8 nu)
{
    *dEgw=pow(nu,3.);
    return;
}

int lalDebugLevel = 0;
int main ()
{
    static LALStatus status;
    DIntegrateIn zint;
    AstroOmegaGeneralParams params;
    AstroOmegaSourceParams sourcep;
    AstroOmegaCosmoParams cosmop;
    REAL8 omegaz, nu, test;

    //define here the model parameters
    //cosmological parameters
    cosmop.ho=0.65;
    cosmop.density_matter=0.3;
    cosmop.density_vacuum=0.7;
    cosmop.density_k=0.;

    //source parameters for rotating pulsars
    sourcep.fact = 2.88E-22;
    sourcep.numax = 4000.;
    sourcep.SDensitySource = SDensity;
    params.cosmoparams = cosmop;
    params.sourceparams = sourcep;
    for (nu = 0.; nu < sourcep.numax ; nu = nu + 10.)
    {
        
```
params.extraparams = &nu;
LALAstroOmegaSource (&status, &test, nu, &params);
printf("omega(%f) = %.2e\n", nu, test);
return 0;
}

Notes
References


Section 11

LAL Support Interface
Chapter 32

Package **support**

This package covers LAL support routines.

These routines do not conform to LAL requirements, and many of them should be used only for debugging and in test code, not in production code. These are compiled and installed as a separate library `lalsupport`.
32.1 Header **FileIO.h**

Provides standard LAL support IO functions.

**Synopsis**

```c
#include <lal/LALStdio.h>
#include <lal/FileIO.h>
```

Only use **FileIO.h** in test code that links to the **lalsupport** library.
32.1.1 Module FileIO.c

File IO routines for LAL. These should not be used in routines within the LAL library—only within test programs.

Prototypes

```c
FILE *
LALOpenDataFile( const char *fname )

FILE *LALFopen( const char *path, const char *mode );
int LALFclose( FILE *stream );
```

Description

The routines `LALFopen()` and `LALFclose()` are macro defined to be the same as the standard C routines `fopen()` and `fclose()`. These should only be used in test programs.

The routine `LALOpenDataFile()` is used to open a data file for reading. This routine is also to be used in test programs only. Unless the data file is specified with an absolute path (beginning with a `/`), or a specific path (beginning with a `./` or a `../`), the directory that the file is in is obtained from the environment variable `LAL_DATA_PATH`, which must be set at run-time. (If the environment variable is not set, the default path is . — i.e., the current directory.)

`LAL_DATA_PATH` should typically be set to `/usr/local/share/lal`, or wherever LAL data is installed in your system (which may be different if you used a `--prefix` argument when configuring LAL), but when the test suite is run with `make check`, the variable `LAL_DATA_PATH` is set to the current source directory. If the filename (including the directory path) is too long (more than 256 characters), `LALOpenDataFile()` returns `NULL` and sets `errno` to `ENAMETOOLONG`.

`LAL_DATA_PATH` can be any colon-delimeted list of directories, which are searched in order (just like the `PATH` environment variable). An extra colon inserts the default data directory (`⟨prefix⟩/share/lal`) into the search path at that point. E.g., a leading/trailing colon will look for the default data directory at the start/end of the list of directories.

It is strongly recommended that `LALOpenDataFile()` be used when writing test code.
32.2  Header **PrintVector.h**

This is a simple utility to print vectors into a file.

**Synopsis**

```c
#include <lal/PrintVector.h>
```

Contains the prototypes for the LAL(DT)PrintVector functions
32.2.1 Module PrintVector.c

Print a (datatype)Vector object into a file. For use in non-production and test code only.

Prototypes

```c
void LALZPrintVector ( COMPLEX16Vector *vector )
void LALCPrintVector ( COMPLEX8Vector *vector )
void LALDPrintVector ( REAL8Vector *vector )
void LALSPrintVector ( REAL4Vector *vector )
void LALI2PrintVector ( INT2Vector *vector )
void LALI4PrintVector ( INT4Vector *vector )
void LALI8PrintVector ( INT8Vector *vector )
void LALU2PrintVector ( UINT2Vector *vector )
void LALU4PrintVector ( UINT4Vector *vector )
void LALU8PrintVector ( UINT8Vector *vector )
void LALCHARPrintVector ( CHARVector *vector )
void LALPrintVector ( REAL4Vector *vector )
```

Description

Each member of this family of functions prints the elements of (datatype)Vector into a file. Note: the file names are (DT)PrintVector.000, (DT)PrintVector.001, and so on. ((DT) is the abbreviation for the datatype, included in the function names above.) The file numbers are incremented with each additional call. This function is for debugging use only: it uses a static internal variable to keep track of the file number so it should not be used in any real analysis codes.

Algorithm

Uses

LALFopen()
LALFclose()

Notes

This function uses an internal static variable to keep track of file numbers. For this reason it should only be used for debugging purposes in test functions, not in any production code.

Additionally, since printf cannot handle INT8 as integers, the functions LALI8PrintVector() and LALU8PrintVector() use a typecast to REAL8 and are thus only valid for numbers between around $-10^{15}$ and $10^{15}$.

The output is two or three space-separated columns: the first column is the index of the element; the second is the element itself for real and integer vectors and the real part of the element for complex vectors; complex vectors have a third column containing the imaginary part of the element.
32.2.2 Program PrintVectorTest.c

Tests the routines in PrintVector.c.

Usage

PrintVectorTest

Description

This program generates and prints a sequence of ⟨datatype⟩Vectors; the program itself always returns success, so the testing function is actually served by examination of the output files.

Exit codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Always returned.</td>
</tr>
</tbody>
</table>

Uses

LALZCreateVector()
LALCCreateVector()
LALDCreateVector()
LALSCreateVector()
LALI2CreateVector()
LALI4CreateVector()
LALI8CreateVector()
LALU2CreateVector()
LALU4CreateVector()
LALU8CreateVector()
LALCHARCreateVector()
LALZDestroyVector()
LALCDestroyVector()
LALDDestroyVector()
LALSDestroyVector()
LALI2DestroyVector()
LALI4DestroyVector()
LALI8DestroyVector()
LALU2DestroyVector()
LALU4DestroyVector()
LALU8DestroyVector()
LALCHARDestroyVector()
32.3  Header PrintFTSeries.h

This is a simple utility to print time and frequency series into a file.

Synopsis

#include <lal/PrintFTSeries.h>

Provides prototype information for the routines in PrintTimeSeries.c and PrintFrequencySeries.c.
32.3.1 Module PrintTimeSeries.c

Print a (datatype)TimeSeries object into a file. For use in non-production and test code only.

Prototypes

```c
void LALZPrintTimeSeries ( COMPLEX16TimeSeries *series, const CHAR *filename )
void LALCPrintTimeSeries ( COMPLEX8TimeSeries *series, const CHAR *filename )
void LALDPrintTimeSeries ( REAL8TimeSeries *series, const CHAR *filename )
void LALSPrintTimeSeries ( REAL4TimeSeries *series, const CHAR *filename )
void LALI2PrintTimeSeries ( INT2TimeSeries *series, const CHAR *filename )
void LALI4PrintTimeSeries ( INT4TimeSeries *series, const CHAR *filename )
void LALI8PrintTimeSeries ( INT8TimeSeries *series, const CHAR *filename )
void LALU2PrintTimeSeries ( UINT2TimeSeries *series, const CHAR *filename )
void LALU4PrintTimeSeries ( UINT4TimeSeries *series, const CHAR *filename )
void LALU8PrintTimeSeries ( UINT8TimeSeries *series, const CHAR *filename )
void LALPrintTimeSeries ( REAL4TimeSeries *series, const CHAR *filename )
```

Description

Each member of this family of functions prints the elements of (datatype)TimeSeries into a file. Note: the file name is specified using a character string. This function is for debugging use only: its arguments do not conform to LAL standards so it should not be used in any real analysis codes.

Algorithm

Uses

```c
LALFopen()
LALFclose()
LALCHARCreateVector()
LALCHARDestroyVector()
LALUnitAsString()
```

Notes

This function’s arguments do not conform to the LAL spec. For this reason it should only be used for debugging purposes in test functions, not in any production code.

Additionally, since printf cannot handle INT8 as integers, the functions LALI8PrintTimeSeries() and LALU8PrintTimeSeries() use a typecast to REAL8 and are thus only valid for numbers between around $-10^{15}$ and $10^{15}$.

The first five lines of the file are a header containing:

1. the name of the series
2. the starting epoch
3. the units expressed in terms of the basic SI units
4. column labels

after which come the data, one per line.

The output is two or three tab-separated columns: the first column is the time corresponding to the row in question, in seconds after the series’ starting epoch; for real and integer time series, the second column is the value of the series; for complex time series, the second column is the real part and the third the imaginary part of the value.
32.3.2 Program PrintFTSeriesTest.c

Tests the routines in PrintTimeSeries.c and PrintFrequencySeries.c.

Usage

PrintFTSeriesTest

Description

This program generates and prints a sequence of frequency and time series; the program only detects errors coming from other LAL functions, so more in-depth testing requires examination of the output files. (The program ReadFTSeriesTest also tests the routines in PrintFrequencySeries.c and ReadFrequencySeries.c.)

Exit codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOM</td>
<td>0</td>
<td>&quot;Nominal exit&quot;</td>
</tr>
<tr>
<td>FUN</td>
<td>1</td>
<td>&quot;Error from LAL function&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants PRINTFTSERIESTESTC_E<name>, and the status descriptions in PRINTFTSERIESTESTC_MSGE<name>. The source code with these messages is in PrintFTSeriesTest.c on line 1.93.

Uses

LALSCreateVector()
LALDCreateVector()
LALCCreateVector()
LALZCreateVector()
LALSDestroyVector()
LALDDestroyVector()
LALCDestroyVector()
LALZDestroyVector()
LALSPrintFrequencySeries()
LALDPrintFrequencySeries()
LALCPrintFrequencySeries()
LALZPrintFrequencySeries()
LALSPrintTimeSeries()
LALDPrintTimeSeries()
LALCPrintTimeSeries()
LALZPrintTimeSeries()
LALCheckMemoryLeaks()
LALNameLength
lalDebugLevel
lalDimensionlessUnit
LALStatus

Notes

The program as written generates and prints single and double precision real and complex time and frequency series. The routines for integers are not tested.
32.3.3 Module **PrintFrequencySeries.c**

Print a (datatype)FrequencySeries object into a file. For use in non-production and test code only.

**Prototypes**

```c
void LALZPrintFrequencySeries ( COMPLEX16FrequencySeries *series, const CHAR *filename )
void LALCPrintFrequencySeries ( COMPLEX8FrequencySeries *series, const CHAR *filename )
void LALDPrintFrequencySeries ( REAL8FrequencySeries *series, const CHAR *filename )
void LALSPrintFrequencySeries ( REAL4FrequencySeries *series, const CHAR *filename )
void LALI2PrintFrequencySeries ( INT2FrequencySeries *series, const CHAR *filename )
void LALI4PrintFrequencySeries ( INT4FrequencySeries *series, const CHAR *filename )
void LALI8PrintFrequencySeries ( INT8FrequencySeries *series, const CHAR *filename )
void LALU2PrintFrequencySeries ( UINT2FrequencySeries *series, const CHAR *filename )
void LALU4PrintFrequencySeries ( UINT4FrequencySeries *series, const CHAR *filename )
void LALU8PrintFrequencySeries ( UINT8FrequencySeries *series, const CHAR *filename )
void LALPrintFrequencySeries ( REAL4FrequencySeries *series, const CHAR *filename )
```

**Description**

Each member of this family of functions prints the elements of ⟨datatype⟩FrequencySeries into a file. Note: the file name is specified using a character string. This function is for debugging use only: its arguments do not conform to LAL standards so it should not be used in any real analysis codes.

**Algorithm**

Uses

- LALFopen()
- LALFclose()
- LALCHARCreateVector()
- LALCHARDestroyVector()
- LALUnitAsString()

**Notes**

This function’s arguments do not conform to the LAL spec. For this reason it should only be used for debugging purposes in test functions, not in any production code.

Additionally, since printf cannot handle INT8 as integers, the functions LALI8PrintFrequencySeries() and LALU8PrintFrequencySeries() use a typecast to REAL8 and are thus only valid for numbers between around $-10^{15}$ and $10^{15}$.

The first four lines of the file are a header containing:

1. the name of the series
2. heterodyning information, if any
3. the starting epoch, relative to the GPS reference epoch (1980 January 6)
4. the units expressed in terms of the basic SI units
5. column labels
after which come the data, one per line.

The output is two or three tab-separated columns: the first column is the frequency in hertz corresponding to the row in question; for real and integer frequency series, the second column is the value of the series; for complex frequency series, the second column is the real part and the third the imaginary part of the value.

Note that the frequency given is the physical frequency. In the case of a heterodyned frequency series, this is the heterodyning frequency plus the frequency offset. A frequency series of length \[N\] is assumed to be packed so that the 0th element corresponds to zero frequency offset, elements 1 through \([N/2]\) to positive frequency offsets (in ascending order), and elements \(N − [N/2]\) to \(N − 1\) to negative frequency offsets (also in ascending order, so that the frequency corresponding to the \(N − 1\)st element is just below that of the 0th element). If \(N\) is even, the element in position \(N/2\) is assumed to correspond both the maximum positive and negative frequency offset.
32.4 Header `ReadFTSeries.h`

This is a simple utility to Read time and frequency series into a file.

Synopsis

```c
#include <lal/ReadFTSeries.h>
```

Provides prototype information for the routines in `ReadTimeSeries.c` and `ReadFrequencySeries.c`.
32.4.1 Module ReadFrequencySeries.c

Prototypes

```c
void LALZReadFrequencySeries ( LALStatus* status,  
    COMPLEX16FrequencySeries *series,  
    const CHAR *filename )

void LALCReadFrequencySeries ( LALStatus* status,  
    COMPLEX8FrequencySeries *series,  
    const CHAR *filename )

void LALDReadFrequencySeries ( LALStatus* status,  
    REAL8FrequencySeries *series,  
    const CHAR *filename )

void LALSReadFrequencySeries ( LALStatus* status,  
    REAL4FrequencySeries *series,  
    const CHAR *filename )

void LALReadFrequencySeries ( LALStatus* status,  
    REAL4FrequencySeries *series,  
    const CHAR *filename )
```

Description

Each member of this family of functions reads from a file the output of the corresponding PrintFrequencySeries routine.

Algorithm

Uses

- LALOpenDataFile()
- LALParseUnitString()
- LALCHARCreateVector()
- LALCHARDestroyVector()
- LALDCreateVector()
- LALDDestroyVector()

Notes

These functions perform I/O operations, which are not a part of LAL proper. They should only be used for debugging purposes in test functions, not in any production code.
32.4.2 Module ReadTimeSeries.c

Prototypes

```c
void LALZReadTimeSeries ( LALStatus* status,
                           COMPLEX16TimeSeries *series,
                           const CHAR *filename )

void LALCReadTimeSeries ( LALStatus* status,
                           COMPLEX8TimeSeries *series,
                           const CHAR *filename )

void LALDReadTimeSeries ( LALStatus* status,
                           REAL8TimeSeries *series,
                           const CHAR *filename )

void LALSReadTimeSeries ( LALStatus* status,
                           REAL4TimeSeries *series,
                           const CHAR *filename )

void LALReadTimeSeries ( LALStatus* status,
                         REAL4TimeSeries *series,
                         const CHAR *filename )
```

Description

Each member of this family of functions reads from a file the output of the corresponding `PrintTimeSeries` routine.

Algorithm

Uses

- LALOpenDataFile()
- LALParseUnitString()
- LALCHARCreateVector()
- LALCHARDestroyVector()
- LALCreateVector()
- LALDestroyVector()

Notes

These functions perform I/O operations, which are not a part of LAL proper. They should only be used for debugging purposes in test functions, not in any production code.
32.4.3 Program ReadFTSeriesTest.c

Tests the routines in ReadTimeSeries.c and ReadFrequencySeries.c.

Usage

ReadFTSeriesTest

Description

For each of the real and complex datatypes (single and double precision), this program fills a Time- and FrequencySeries, prints them to disk with the appropriate LALPrint(datatype)FrequencySeries() and LALPrint(datatype)TimeSeries() routines, then reads them back in with the appropriate LALRead(datatype)FrequencySeries() and LALRead(datatype)TimeSeries() routines and checks to make sure the resulting series agree, printing the results to standard error.

Exit codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOM</td>
<td>0</td>
<td>&quot;Nominal exit&quot;</td>
</tr>
<tr>
<td>CHK</td>
<td>1</td>
<td>&quot;Error checking failed to catch bad data&quot;</td>
</tr>
<tr>
<td>FUN</td>
<td>2</td>
<td>&quot;Subroutine returned error for valid data&quot;</td>
</tr>
<tr>
<td>FLS</td>
<td>3</td>
<td>&quot;Subroutine returned unexpected results&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants READFTSERIESTESTC_E<name>, and the status descriptions in READFTSERIESTESTC_MSGE<name>. The source code with these messages is in ReadFTSeriesTest.c on line 1.110.

Uses

LALSCreateVector()
LALDCreateVector()
LALCCreateVector()
LALZCreateVector()
LALSDestroyVector()
LALDDestroyVector()
LALCDestroyVector()
LALZDestroyVector()
LALSPrintFrequencySeries()
LALDPrintFrequencySeries()
LALCPrintFrequencySeries()
LALZPrintFrequencySeries()
LALSReadFrequencySeries()
LALDReadFrequencySeries()
LALCReadFrequencySeries()
LALZReadFrequencySeries()
LALSPrintTimeSeries()
LALDPrintTimeSeries()
LALCPrintTimeSeries()
LALZPrintTimeSeries()
LALSReadTimeSeries()
LALDReadTimeSeries()
LALCReadTimeSeries()
LALZReadTimeSeries()
LALUnitRaise()
LALUnitCompare()
LALUnitMultiply()
LALCheckMemoryLeaks()
LALNameLength
lalDebugLevel
lalHertzUnit
lalStrainUnit
lalADCCountUnit
LALStatus
LALUnit
LALUnitPair

Notes
32.5 Header `ReadNoiseSpectrum.h`

Provides function to read in a file containing a possibly unequally sampled noise amplitude spectrum (strain/√(Hz)) and return as a frequency series.

Synopsis

```c
#include <lal/ReadNoiseSpectrum.h>
```
Error codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>1</td>
<td>&quot;Null pointer&quot;</td>
</tr>
<tr>
<td>NNUL</td>
<td>2</td>
<td>&quot;Non-null pointer&quot;</td>
</tr>
<tr>
<td>ALOC</td>
<td>3</td>
<td>&quot;Memory allocation error&quot;</td>
</tr>
<tr>
<td>OPEN</td>
<td>4</td>
<td>&quot;Error opening file&quot;</td>
</tr>
<tr>
<td>FCLO</td>
<td>5</td>
<td>&quot;Error closing file&quot;</td>
</tr>
<tr>
<td>PARS</td>
<td>8</td>
<td>&quot;Error parsing spectrum file&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above did not obey the LAL naming convention, i.e. the code does not use the file name in all caps as the prefix for the error names. Consult the source code for the full names. Better yet, fix the code. The source code with these messages is in `ReadNoiseSpectrum.h` on line 1.84.
32.5.1 Module ReadNoiseSpectrum.c

Function to read in noise spectrum from a formatted ascii file and return the amplitude noise spectrum in \( \text{strain}/\sqrt{\text{Hz}} \).

Prototypes

\[
\text{void LALReadNoiseSpectrum(LALStatus *stat, REAL4FrequencySeries *spectrum, CHAR *fname)}
\]

Description

\texttt{LALReadNoiseSpectrum()} fills the contents of the REAL4FrequencySeries \texttt{spectrum} from data read from formatted ascii file with name \texttt{fname}. The ascii file should have a header (greater than or equal to one line) which is indicated by a \# at the start of the line. The first line of the file must contain the number of points at which the spectrum is sampled. If the spectrum is sampled at 500 different points, then the first line would be

\[
\# \text{npoints=500}
\]

Replace 500 by the number of sample points in your particular data.

The REAL4FrequencySeries \texttt{spectrum} should be allocated before calling the routine which uses the length and metadata information to determine the evenly sampled output that is required. The function does nearest neighbor interpolation to get the points in the output frequency series.

Uses

\texttt{LALOpenDataFile()}

\texttt{LALMalloc()}

\texttt{LALFree()}

Notes

Author: Patrick Brady

$Id: ReadNoiseSpectrum.c,v 1.4 2007/06/08 14:41:55 bema Exp$
32.6  Header StreamInput.h

Provides routines to read data from an open stream and store it in LAL data structures.

Synopsis

#include "StreamInput.h"

This header provides prototypes for routines that construct LAL data structures using the data from a file (or other I/O) stream. The routines do not provide a system-level interface to create files and open or close file streams; they simply assume that they have been passed an open, readable stream. Nonetheless, because they involve I/O stream manipulation, these routines are placed in the lalsupport library rather than in lal proper.

The routines in StreamVectorInput.c and StreamVectorSequenceInput.c are compartmentalized in such a way that they can easily be converted if the LAL specification later changes the way in which I/O streams are handled. In particular, the only file I/O commands used are fgets() and feof(). Thus the upgrade would involve only the following global changes:

1. Replace all occurrences of FILE * with the name of the LAL I/O stream pointer type.
2. Replace all occurrences of fgets() and feof() with equivalent LAL functions.

In particular, there is no need to translate routines such as fscanf(); one should simply read data into a LAL CHARVector and then use sscanf() to format the input. This is the approach used in the numerical input routines in StreamVectorInput.c and StreamVectorSequenceInput.c.

The routines in StreamSequenceInput.c are less robust but much more efficient: they use fscanf() to parse the input stream directly. They are intended primarily for test programs that may need to read large datafiles of undetermined length. The routines in StreamSeriesInput.c and StreamGridInput.c also parse the input stream directly using fscanf(), to avoid potentially crippling computational overhead.

Error conditions

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUL</td>
<td>1</td>
<td>&quot;Unexpected null pointer in arguments&quot;</td>
</tr>
<tr>
<td>OUT</td>
<td>2</td>
<td>&quot;Output handle points to a non-null pointer&quot;</td>
</tr>
<tr>
<td>MEM</td>
<td>3</td>
<td>&quot;Memory allocation error&quot;</td>
</tr>
<tr>
<td>LEN</td>
<td>4</td>
<td>&quot;No numbers were read&quot;</td>
</tr>
<tr>
<td>SLEN</td>
<td>5</td>
<td>&quot;Not enough numbers read to fill sequence&quot;</td>
</tr>
<tr>
<td>VLEN</td>
<td>6</td>
<td>&quot;Could not determine complex vectorLength&quot;</td>
</tr>
<tr>
<td>DLEN</td>
<td>7</td>
<td>&quot;Dimension lengths inconsistent or not given&quot;</td>
</tr>
<tr>
<td>DIM</td>
<td>8</td>
<td>&quot;Inconsistent or non-positive arrayDim value&quot;</td>
</tr>
<tr>
<td>FMT</td>
<td>9</td>
<td>&quot;Badly formatted number&quot;</td>
</tr>
<tr>
<td>BUF</td>
<td>10</td>
<td>&quot;BUFFSIZE not a multiple of largest complex type size&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants STREAMINPUT_E<name>, and the status descriptions in STREAMINPUT_MSGE<name>. The source code with these messages is in StreamInput.h on line 1.110.

Types
32.6.1 Module StreamVectorInput.c

Reads data from a single line in an input stream.

Prototypes

```c
void LALCHARReadVector( LALStatus *stat, CHARVector **vector, FILE *stream )
void LALI2ReadVector ( LALStatus *stat, INT2Vector **vector, FILE *stream, BOOLEAN strict )
void LALI4ReadVector ( LALStatus *stat, INT4Vector **vector, FILE *stream, BOOLEAN strict )
void LALI8ReadVector ( LALStatus *stat, INT8Vector **vector, FILE *stream, BOOLEAN strict )
void LALU2ReadVector ( LALStatus *stat, UINT2Vector **vector, FILE *stream, BOOLEAN strict )
void LALU4ReadVector ( LALStatus *stat, UINT4Vector **vector, FILE *stream, BOOLEAN strict )
void LALU8ReadVector ( LALStatus *stat, UINT8Vector **vector, FILE *stream, BOOLEAN strict )
void LALSReadVector ( LALStatus *stat, REAL4Vector **vector, FILE *stream, BOOLEAN strict )
void LALDReadVector ( LALStatus *stat, REAL8Vector **vector, FILE *stream, BOOLEAN strict )
```

Description

These routines read ASCII data from the I/O stream `*stream` until a newline or the end-of-input is reached. (The line can be of arbitrary length; the data is temporarily stored in a linked list of buffers.) Once read, a LAL vector structure `**vector` is created and the data stored in it. The routine passes back a pointer to the new structure. For the numerical routines, the `strict` parameter determines whether the routine will do strict error checking based on the contents of the input stream (see below).

The basic routine in this module is `LALCHARReadVector()`, which simply stores bytes read from `*stream` until the next newline character `\n`, null character `\0`, or the end of the input as determined by the `feof()` function. The vector includes the newline (if present), and also an explicit `\0` at the end, if one was not already present. This routine should not be used to read a binary data stream, which are not logically divided into ‘lines’. Unless it aborts due to invalid arguments or failed memory allocation, `LALCHARReadVector()` will always return successfully regardless of the contents of the input stream; `*vector` will created containing at least a single `\0` terminator, if nothing else.

The other routines in this module use `LALCHARReadVector()` to read a line, and then parse it into numerical datatypes using the corresponding routine in the `StringConvert.c` module. Conversion stops when the routine encounters a character that cannot be parsed as part of a number. If `strict` is 0, the routine will fail only due to invalid arguments or memory allocation failure, not from a poorly-formatted input stream; if no numbers are read, `*vector` will remain `NULL`, but no error will be reported. (In this mode, the calling routine should always test the output before trying to dereference it, in order to avoid segmentation violations.) If `strict` is nonzero, the routine will report an error if the input stream was poorly formatted, either an `ELEN` error if no numbers were read, or `EFMT` if a character was encountered that was neither part of a parsable number nor whitespace.

Note that `strict`=0 allows an input stream to contain blank lines or comments. A comment begins with any character that cannot occur in a valid number, which will cause the numerical parser to skip the rest of the line. The usual comment delimiters are `#` and `/`, but any character except `+` `-` `e` `E` ``, none of digits, and whitespace will work.
Algorithm

Uses

LALMalloc() LALFree()
LALCHARCreateVector() LALCHARDestroyVector()
LALI2CreateVector() LALU2CreateVector()
LALI4CreateVector() LALU4CreateVector()
LALI8CreateVector() LALU8CreateVector()
LALSCreateVector() LALDCreateVector()
LALWarning()

Notes
32.6.2 Module StreamVectorSequenceInput.c

Reads the entire contents of an input stream into a vector sequence.

Prototypes

```c
void LALCHARReadVectorSequence( LALStatus *stat,
                                 CHARVectorSequence **sequence,
                                 FILE *stream )

void LALI2ReadVectorSequence ( LALStatus *stat, INT2VectorSequence **sequence, FILE *stream )

void LALI4ReadVectorSequence ( LALStatus *stat, INT4VectorSequence **sequence, FILE *stream )

void LALI8ReadVectorSequence ( LALStatus *stat, INT8VectorSequence **sequence, FILE *stream )

void LALU2ReadVectorSequence ( LALStatus *stat, UINT2VectorSequence **sequence, FILE *stream )

void LALU4ReadVectorSequence ( LALStatus *stat, UINT4VectorSequence **sequence, FILE *stream )

void LALU8ReadVectorSequence ( LALStatus *stat, UINT8VectorSequence **sequence, FILE *stream )

void LALSReadVectorSequence ( LALStatus *stat, REAL4VectorSequence **sequence, FILE *stream )

void LALDReadVectorSequence ( LALStatus *stat, REAL8VectorSequence **sequence, FILE *stream )
```

Description

These routines read data from the I/O stream *stream until the end-of-input is reached. Each line is stored as a data vector, and the vectors are combined into a LAL vector sequence structure **sequence. Each line vector is padded with zeros to match the length of the longest line. The routine passes back a pointer to the new structure.

The routine LALCHARReadVectorSequence() essentially stores an image of the I/O stream as a sequence of lines padded with \0 characters. However, it will skip over any empty lines, which occur, for instance, when the end-of-input or a null character \0 occurs immediately following a newline character \n. The numeric routines will additionally skip blank lines, comment lines, or other input lines that have no parseable numbers in them. (As with the routines in StreamVectorInput.c, comment in indicated by a # sign at the beginning of a line or a % sign anywhere in the line, signifying that the remainder of the line is to be ignored.) However, if an input line contains any parseable data, then the corresponding vector in the vector sequence will be allocated (and padded with zeros, if it is shorter than the longest line).

Algorithm

These functions first create a linked list of vectors, using the routines in StreamVectorInput.c to read them in. Once the list is complete, the longest vector length is determined, and the vector sequence is created and filled.

The numeric routines skip over blank, comment, or otherwise unparsable lines by catching and handling the LEN error code generated by the vector input routine. However, it is worth pointing out that the vector input routine will have generated an error message if the error reporting bit in lalDebugLevel was set. The vector sequence input routines will therefore generate a followup messages indicating that the preceding error was successfully dealt with. So you may see pairs of ABORT: and CONTINUE: error messages when reading files containing blank or comment lines.
Uses

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>LALMalloc()</td>
</tr>
<tr>
<td>LALFree()</td>
</tr>
<tr>
<td>LALCHARReadVector()</td>
</tr>
<tr>
<td>LALCHARDestroyVector()</td>
</tr>
<tr>
<td>LALCHARCreateVectorSequence()</td>
</tr>
<tr>
<td>LALI2ReadVector()</td>
</tr>
<tr>
<td>LALI2DestroyVector()</td>
</tr>
<tr>
<td>LALI2CreateVectorSequence()</td>
</tr>
<tr>
<td>LALI4ReadVector()</td>
</tr>
<tr>
<td>LALI4DestroyVector()</td>
</tr>
<tr>
<td>LALI4CreateVectorSequence()</td>
</tr>
<tr>
<td>LALI8ReadVector()</td>
</tr>
<tr>
<td>LALI8DestroyVector()</td>
</tr>
<tr>
<td>LALI8CreateVectorSequence()</td>
</tr>
<tr>
<td>LALU2ReadVector()</td>
</tr>
<tr>
<td>LALU2DestroyVector()</td>
</tr>
<tr>
<td>LALU2CreateVectorSequence()</td>
</tr>
<tr>
<td>LALU4ReadVector()</td>
</tr>
<tr>
<td>LALU4DestroyVector()</td>
</tr>
<tr>
<td>LALU4CreateVectorSequence()</td>
</tr>
<tr>
<td>LALU8ReadVector()</td>
</tr>
<tr>
<td>LALU8DestroyVector()</td>
</tr>
<tr>
<td>LALU8CreateVectorSequence()</td>
</tr>
<tr>
<td>LALSReadVector()</td>
</tr>
<tr>
<td>LALSDestroyVector()</td>
</tr>
<tr>
<td>LALSCreateVectorSequence()</td>
</tr>
<tr>
<td>LALDReadVector()</td>
</tr>
<tr>
<td>LALDDestroyVector()</td>
</tr>
<tr>
<td>LALDCreateVectorSequence()</td>
</tr>
</tbody>
</table>

Notes
32.6.3 Module StreamSequenceInput.c

Converts an input stream into a data sequence.

Prototypes

```c
void LALCHARReadSequence( LALStatus *stat, CHARSequence **sequence, FILE *stream )
void LALI2ReadSequence ( LALStatus *stat, INT2Sequence **sequence, FILE *stream )
void LALI4ReadSequence ( LALStatus *stat, INT4Sequence **sequence, FILE *stream )
void LALI8ReadSequence ( LALStatus *stat, INT8Sequence **sequence, FILE *stream )
void LALU2ReadSequence ( LALStatus *stat, UINT2Sequence **sequence, FILE *stream )
void LALU4ReadSequence ( LALStatus *stat, UINT4Sequence **sequence, FILE *stream )
void LALU8ReadSequence ( LALStatus *stat, UINT8Sequence **sequence, FILE *stream )
void LALSReadSequence ( LALStatus *stat, REAL4Sequence **sequence, FILE *stream )
void LALDReadSequence ( LALStatus *stat, REAL8Sequence **sequence, FILE *stream )
void LALCReadSequence ( LALStatus *stat, COMPLEX8Sequence **sequence, FILE *stream )
void LALZReadSequence ( LALStatus *stat, COMPLEX16Sequence **sequence, FILE *stream )
```

Description

These routines read data from the I/O stream *stream until the end-of-input is reached. (The input can be of arbitrary length; the data is temporarily stored in a linked list of buffers.) Once read, a LAL sequence structure **sequence is created and the data stored in it. The routine passes back a pointer to the new structure.

The routine LALCHARReadSequence() simply stores the entire remaining contents of the I/O stream in a CHARSequence, including whitespace, newline '\n', null '\0', or other special characters. (It can in principle be used to read and store binary data as a sequence of bytes. Note that the end-of-transmission byte '\004' does not necessarily mark the end-of-input, which is instead determined using the feof() function.)

The other routines in this module interpret the input as a sequence of whitespace-separated numbers, which are parsed directly from the I/O stream using fscanf(). The sequence is terminated at the end-of-input or at any point where fscanf() is unable to parse the input.

For the complex input routines LALCReadSequence() and LALZReadSequence(), each pair of numbers read are interpreted as the real and imaginary parts of a complex number. The usual input is for each line to contain a pair of numbers, but fscanf() does not distinguish between newline and other whitespace characters, so neither do these routines.

Unlike the numerical routines in other StreamInput.h modules, these routines have no mechanism to deal with comments; every whitespace-delimited substring will be treated as a number.
Algorithm

These routines read data into a linked list of buffers, to allow memory allocation to occur in batches for improved efficiency. The numerical routines also use `fscanf()` directly on the I/O stream to avoid the inefficiency of storing and parsing intermediate character strings, as is done by the corresponding vector sequence input routines. This reduces robustness and versatility (as indicated, for instance, by the inability of dealing with comments), and increases the number of potential points-of-failure (by requiring a consistent implementation across platforms of `getc()` and `fscanf()`, rather than the single function `fgets()` used by other stream input routines). However, these sacrifices are necessary to allow LAL applications to ingest large quantities of numerical data efficiently.

Uses

```
LALMalloc()       LALFree()
LALWarning()      LALCHARCreateVector()
LALI2CreateVector() LALU2CreateVector()
LALI4CreateVector() LALU4CreateVector()
LALI8CreateVector() LALU8CreateVector()
LALSCreateVector() LALDCreateVector()
LALCCreateVector() LALZCreateVector()
```

Notes

Author: Creighton, T. D.

$Id: StreamSequenceInput.m4,v 1.4 2005/02/18 03:09:47 jolien Exp$
32.6.4 Module StreamSeriesInput.c

Converts an input stream into a time or frequency series.

Prototypes

```c
void LAL<typecode>ReadTSeries( LALStatus *stat,
   <datatype>TimeSeries *series,
   FILE *stream )

void LAL<typecode>ReadTVectorSeries( LALStatus *stat,
   <datatype>TimeVectorSeries *series,
   FILE *stream )

void LAL<typecode>ReadTArraySeries( LALStatus *stat,
   <datatype>TimeArraySeries *series,
   FILE *stream )

void LAL<typecode>ReadFSeries( LALStatus *stat,
   <datatype>FrequencySeries *series,
   FILE *stream )
```

Description

These routines parse an input stream *stream to fill in the data and metadata fields of a time or frequency series *series. The field *series->data must be NULL, so that it can be created and filled by the routine. The other fields may be initialized or not; they will be overwritten by metadata read from *stream. If an error occurs, *series will be left unchanged, but *stream will have been read up to the point where the error occurred.

For each of these prototype templates there are in fact 10 separate routines corresponding to all the atomic datatypes <datatype> (except CHAR) referred to by <typecode>:

<table>
<thead>
<tr>
<th>&lt;typecode&gt;</th>
<th>&lt;datatype&gt;</th>
<th>&lt;typecode&gt;</th>
<th>&lt;datatype&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>I2</td>
<td>INT2</td>
<td>U2</td>
<td>UINT2</td>
</tr>
<tr>
<td>I4</td>
<td>INT4</td>
<td>U4</td>
<td>UINT4</td>
</tr>
<tr>
<td>I8</td>
<td>INT8</td>
<td>U8</td>
<td>UINT8</td>
</tr>
<tr>
<td>S</td>
<td>REAL4</td>
<td>C</td>
<td>COMPLEX8</td>
</tr>
<tr>
<td>D</td>
<td>REAL8</td>
<td>Z</td>
<td>COMPLEX16</td>
</tr>
</tbody>
</table>

Format for *stream: The input stream is assumed to be a text stream (ASCII) consisting of a header containing metadata followed by numerical data in standard integer or floating-point format, as recognized by the routines in StringConvert.c. The header consists of zero or more lines beginning with a ‘#’ character, followed by a metadata field name and value in the format:

```
# fieldname=value
```

The = sign in this format is standard but optional; it may be replaced or surrounded with any amount of any whitespace except a newline ‘\n’. If fieldname is unrecognized, it is ignored; if it is recognized, then value must be in a suitable format for the field type, as described below. Blank lines, or lines containing just a # character, are skipped. Once a line is encountered that contains non-whitespace characters and does not start with ‘#’, that line is assumed to be the beginning of the numerical data. From that point on, all non-whitespace characters must be part of parseable numbers; no more comments are permitted (although blank lines will still be skipped).

If a metadata field appears twice in the header, the later one takes precedence. At present these routines do not track which fields have been previously assigned, so no warnings or errors are generated.
How the data is packed into the `series->data` structure depends on what metadata has been provided, as described below.

**Required, conditional, and optional metadata:** The input stream need not contain a complete set of metadata, allowing some metadata to be read from `*stream` and others to be set elsewhere. For each type of series, some metadata will be *required*, and the routine will abort if the metadata is not found. Other metadata are *conditional*, meaning that the routine will operate differently depending on whether or not these metadata were found. The remaining metadata are *optional*; if they are not found in `*stream`, they will be left unchanged. The recognized metadata fields are listed below.

<datatype>TimeSeries:

**Required fields:** none

**Conditional fields:** length

**Optional fields:** name, epoch, deltaT, f0, sampleUnits, datatype

<datatype>TimeVectorSeries:

**Required fields:** none

**Conditional fields:** length, vectorLength

**Optional fields:** name, epoch, deltaT, f0, sampleUnits, datatype

<datatype>TimeArraySeries:

**Required fields:** dimLength

**Conditional fields:** length, arrayDim

**Optional fields:** name, epoch, deltaT, f0, sampleUnits, datatype

<datatype>FrequencySeries:

**Required fields:** none

**Conditional fields:** length

**Optional fields:** name, epoch, deltaT, f0, deltaF, sampleUnits, datatype

Below we describe the required format for the field values, as well as what occurs if a conditional field is or isn’t present.

**Required fields:**

*dimLength* (TimeArraySeries only): *value* consists of a sequence of UINT4s separated by whitespace (but *not* a newline `\n`). These data are stored in `series->data->dimLength`: the number of integers gives the number of array indices, while the value of each integer gives the dimension of the corresponding array index.

**Conditional fields:**

*arrayDim* (TimeArraySeries only): *value* is a single UINT4, to be stored in `series->data->arrayDim`. This must equal the product of the index ranges in `dimLength`, above, or an error is returned. If not given, the `arrayDim` field will be set equal to the product of the index ranges in `dimLength`. (The `arrayDim` and `dimLength` fields can appear in any order in `*stream`; checking is done only after all header lines have been read.)

*vectorLength* (TimeVectorSeries only): *value* is a single UINT4, to be stored in `series->data->vectorLength`. If not specified in the header portion of `*stream`, it will be taken to be the number of data on the first line of the data portion of `*stream`, or half the number of real data for a complex-valued TimeVectorSeries; if an odd number of real data are found on the first line of a complex TimeVectorSeries, then an error is returned.
length: value is a single UINT4, to be stored in series->data->length. If it is specified in the header portion of *stream, data will be read until length is reached. Otherwise, *stream will be read to its end or until an unparsable character is read, and length will then be set accordingly. (If parsing stops in the middle of filling a complex, vector, or array valued element, the partly-read element is discarded.)

Optional fields:

name: value is a string surrounded by quotes "", which is parsed in the manner of a string literal in C: it may contain ordinary printable characters (except "" and \\), escape sequences (such as \t for tab, \n for newline, or \\ and \" for literal backslash and quote characters), and octal or hexadecimal codes (\ooo or \x hh respectively) for arbitrary bytes. Unlike in C, literals cannot be split between lines, adjacent literals are not concatenated, and converted strings longer than LALNameLength – 1 will be truncated. The resulting string is stored in series->name, and will always contain a \0 terminator, beyond which the contents are unspecified.

epoch: value is a single INT8 number of GPS nanoseconds, or a pair of INT4s representing GPS seconds and nanoseconds separately, separated by non-newline whitespace.

deltaT (any time series): value is a single REAL8 number.

f0: value is a single REAL8 number.

deltaF (FrequencySeries only): value is a single REAL8 number.

sampleUnits: value is string surrounded by quotes "": the quotes are stripped and the string passed to LALParseUnitString() to determine series->sampleUnits. Since LALParseUnitString() is not very robust, it is recommended to use only unit strings that have been generated by LALUnitAsString(), or to remove this metadata field and set series->sampleUnits within the code.

datatype: value is string identifying the series type; e.g. REAL4TimeSeries (not surrounded by quotes). This should correspond to the type of *series, not to any field in *series. If there is a type mismatch, a warning is generated (and errors may occur later while parsing the data).

Data format: The data portion of *stream consists of whitespace-separated integer or real numbers. For complex input routines, the real data are parsed as alternately the real and imaginary parts of successive complex numbers. By convention, each line should correspond to a single base, complex, vector, or array valued element of the series->data sequence. However, this is required only in the case of a TimeVectorSeries where the vectorLength metadata was not set in the header, since in this case the value of vectorLength will be taken from the number of elements read on the first data line. After this, and in all other cases, newlines are treated as any other whitespace.

If a length value is specified in the header, then data are read until the required length is achieved; if fscanf() returns zero or negative before this (representing either the end-of-input or a character that cannot be interpreted as part of the numerical data), an error is returned. If a length value was not specified, data are read until fscanf() returns zero or negative: at this point any partially-completed complex, vector, or array valued element is discarded, and series->data->length set to the number of elements read.

Algorithm

These routines use LALCHARReadVector() to read the header lines and the first line of data. After this, data are parsed directly from *stream using fscanf(). This is done for efficiency: repeated calling of the LAL string parsing routines in StringConvert.c involves far too much computational overhead.

After the first data line has been read, the length of each sequence element will be known from the atomic type, as well as the specified dimLength (for arrays), vectorLength (for vectors), or number of elements on the first data line (for vectors without an explicitly specified vectorLength). If length is also specified, a sequence of the appropriate size is allocated, and all the data is copied or read directly into it. If length was not specified, the data read with fscanf() are stored in a linked list of buffers of size BUFFSIZE (a local #defined constant) until parsing stops. Then a sequence of the appropriate size is allocated and the data copied into it.
Uses

lalDebugLevel
LALPrintError() LALWarning()
LALMalloc() LALFree()
LALCHARReadVector() LALCHARDestroyVector()
LAL<typecode>CreateVector() LAL<typecode>DestroyVector()
LAL<typecode>CreateVectorSequence() LAL<typecode>DestroyVectorSequence()
LAL<typecode>CreateArraySequence() LAL<typecode>DestroyArraySequence()
LALStringTo<typecode>() LALParseUnitString()

where <typecode> is any of I2, I4, I8, U2, U4, U8, S, D, C, or Z.

Notes
### 32.6.5 Module StreamGridInput.c

Converts an input stream into a LAL grid structure.

#### Prototypes

```c
void LAL<typecode>ReadGrid( LALStatus *stat, <datatype>Grid **grid, FILE *stream )
```

#### Description

These routines parse an input stream `*stream` to create a grid structure `**grid`, and fill in its data and metadata fields. The output parameter `grid` must be a non-`NULL` handle to a `NULL`-valued pointer `*grid`.

This prototype template in fact refers to 10 separate routines corresponding to all the numerical atomic datatypes `<datatype>` referred to by `<typecode>`:

<table>
<thead>
<tr>
<th>&lt;typecode&gt;</th>
<th>&lt;datatype&gt;</th>
<th>&lt;typecode&gt;</th>
<th>&lt;datatype&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>I2</td>
<td>INT2</td>
<td>U2</td>
<td>UINT2</td>
</tr>
<tr>
<td>I4</td>
<td>INT4</td>
<td>U4</td>
<td>UINT4</td>
</tr>
<tr>
<td>I8</td>
<td>INT8</td>
<td>U8</td>
<td>UINT8</td>
</tr>
<tr>
<td>S</td>
<td>REAL4</td>
<td>C</td>
<td>COMPLEX8</td>
</tr>
<tr>
<td>D</td>
<td>REAL8</td>
<td>Z</td>
<td>COMPLEX16</td>
</tr>
</tbody>
</table>

**Format for `*stream`:** The input stream is assumed to be a text stream (ASCII) consisting of a header containing metadata followed by numerical data in standard integer or floating-point format, as recognized by the routines in `StringConvert.c`. The header consists of zero or more lines beginning with a '#' character, followed by a metadata field name and value in the format:

```
# fieldname=value
```

The `=` sign in this format is standard but optional; it may be replaced or surrounded with any amount of any whitespace except a newline '\n'. If `fieldname` is unrecognized, it is ignored; if it is recognized, then `value` must be in a suitable format for the field type, as described below. Blank lines, or lines containing just a '#' character, are skipped. Once a line is encountered that contains non-whitespace characters and does not start with '#', that line is assumed to be the beginning of the numerical data. From that point on, all non-whitespace characters must be part of parseable numbers; no more comments are permitted (although blank lines will still be skipped).

If a metadata field appears twice in the header, the later one takes precedence. At present these routines do not track which fields have been previously assigned, so no warnings or errors are generated. Some metadata fields are **required**, and the routine will abort if it doesn’t find them. Others are **optional**: if they are not found, the routine will assign some default value. The various fields and their required formats are given below:

**Required fields:**

- **dimLength**: `value` consists of a sequence of UINT4s separated by whitespace (but *not* a newline '\n'). These are used to create `(*grid)–>data` with the appropriate dimensions, and are stored in `(*grid)–>data–>dimLength–>data`: the number of integers \( M \) gives the data dimension number (the number of array indecies), while the value of each integer gives the length of each dimension (the range of values of the corresponding array index).

- **offset**: `value` consists of a sequence of REAL8s separated by whitespace (but *not* a newline '\n'). These values are stored in `(*grid)–>offset–>data`. The number of data \( m \) gives the grid dimension, which must be less than or equal to the data dimension \( M \) of the array defined above, and must be consistent among the `offset`, `interval`, and `dimUnits` fields.

- **interval**: `value` consists of a sequence of REAL8s separated by whitespace (but *not* a newline '\n'). These values are stored in `(*grid)–>interval–>data`. The number of data \( m \) gives the grid dimension, which must be less than or equal to the data dimension \( M \) of the array defined above, and must be consistent among the `offset`, `interval`, and `dimUnits` fields.
Optional fields:

**name**: value is a string surrounded by quotes "", which is parsed in the manner of a string literal in C: it may contain ordinary printable characters (except " and \), escape sequences (such as \t for tab, \n for newline, or \ \ and \" for literal backslash and quote characters), and octal or hexadecimal codes (\oo0 or \xhh respectively) for arbitrary bytes. Unlike in C, literals cannot be split between lines, adjacent literals are not concatenated, and converted strings longer than LALNameLength−1 will be truncated. The resulting string is stored in (*grid)->name, and will always contain a \0 terminator, beyond which the contents are unspecified. If this field is not given in stream, then the routine will simply assign (*grid)->name[0]=\0'.

**sampleUnits**: value is string surrounded by quotes "; the quotes are stripped and the string passed to LALParseUnitString() to determine (*grid)->sampleUnits. Since LALParseUnitString() is not very robust, it is recommended to use only unit strings that have been generated by LALUnitAsString(), or to remove this metadata field and set (*grid)->sampleUnits within the code. If this field is not given in stream, then lalDimensionlessUnit is assumed.

**dimUnits**: value is a sequence of strings, each surrounded by quotes ", and optionally separated by whitespace (but not a newline \n'); the quotes are stripped and the string passed to LALParseUnitString() to determine (*grid)->dimUnits. Since LALParseUnitString() is not very robust, it is recommended to use only unit strings that have been generated by LALUnitAsString(), or to remove this metadata field and reset (*grid)->dimUnits within the code. If this field is not given in stream, then (*grid)->dimUnits will be allocated as an array containing m units assigned the value lalDimensionlessUnit, where m is the length of the interval and offset vectors. If this field is given, then the number of unit strings must be consistent with the lengths of the interval and offset vectors, or the routine will abort.

**datatype**: value is string identifying the grid type; e.g. REAL4Grid (not surrounded by quotes). This should correspond to the type of **grid, not to any field in **grid. If there is a type mismatch, a warning is generated (and errors may occur later while parsing the data).

Data format: The first line that is neither blank nor beginning with a ‘#' character is assumed to be the start of the grid data, and is parsed as a sequence of whitespace-separated integers or real numbers. For complex datatypes, the numbers read are interpreted as alternately the real and imaginary parts of the data. Since the number of data required is known from the grid metadata, the routine will read from stream only until the structure is filled (except that the first line of data will be read in its entirety, even if it contains more numbers than required). If the routine encounters an unparsable number or the end-of-input before filling the structure, an error is returned.

Algorithm

These routines use LALCHARReadVector() to read the header lines and the first line of data. The metadata are stored in temporary variables or vectors. After the first data line has been read, the size of the grid should be known from the metadata: the structure is created, and the metadata and first line of data are copied in. After this, any additional data are parsed directly from stream using fscanf(). This is done for efficiency: repeated calling of the LAL string parsing routines in StringConvert.c involves far too much computational overhead.

Uses

lalDebugLevel  
LALPrintError() LALWarning()  
LALMalloc() LALFree()  
LALCHARReadVector() LALCHARDestroyVector()  
LALCreateVector() LALDestroyVector()  
LAL<typecode>CreateGrid() LAL<typecode>DestroyGrid()  
LALStringTo<typecode>() LALParseUnitString()

where <typecode> is any of I2, I4, I8, U2, U4, U8, S, D, C, or Z.
Notes
32.6.6 Program StreamInputTest.c

Reads a sequence or vector sequence from a file.

Usage

StreamInputTest [-o outfile] [-d debuglevel] [-t]
[-v {ch | i2 | i4 | i8 | u2 | u4 | u8 | s | d} infile]
[-s {ch | i2 | i4 | i8 | u2 | u4 | u8 | s | d | c | z} infile]

Description

This test program parses data from an input file or from stdin. The following option flags are accepted:

- **-o** Writes the output to **outfile**. If **outfile** is given as **stdout**, the data is written to standard output (not to a file named **stdout**). If the **-o** flag is not given, the routines are exercised, but no output is written.

- **-d** Sets the debug level to **debuglevel**; if absent, **-d 0** is assumed.

- **-t** Writes to **stderr** the system time required to read the file.

- **-v** Reads the contents of **infile** as a sequence of vectors to be parsed by the routines **LAL<datatype>ReadVectorSequence()**, where **<datatype>** is determined by the argument immediately following the **-v** option flag. If **infile** is given as **stdin**, the data is read from standard input (not from a file named **stdin**).

- **-s** As **-v**, above, except that the file contents are parsed by the routines **LAL<datatype>ReadSequence()**.

If neither **-v** nor **-s** is specified, **-v s StreamInput.dat** is assumed (this file is provided with the distribution so that running the code with no arguments, à la **make check**, will perform a nontrivial test of the algorithm).

For data read in as a character vector sequences, the output will consist of a number of lines equal to the length of the sequence, with each line being the length of the vector; all non-graphical characters in the vector (including the various types of whitespace) will be replaced with single spaces. For character sequences, the output will essentially be a copy of the input. For numerical vector sequences, the output will consist of separate lines for each vector of the sequence, with each line printing the components of the vector in some type-dependent format. For numerical sequences, each line of output contains a single number, or, in the case of complex datatypes, two numbers representing the real and imaginary components, again in some type-dependent format.

Exit codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NORM</td>
<td>0</td>
<td>&quot;Normal exit&quot;</td>
</tr>
<tr>
<td>SUB</td>
<td>1</td>
<td>&quot;Subroutine failed&quot;</td>
</tr>
<tr>
<td>ARG</td>
<td>2</td>
<td>&quot;Error parsing arguments&quot;</td>
</tr>
<tr>
<td>FILE</td>
<td>3</td>
<td>&quot;Could not open file&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants **STREAMINPUTTESTC_E<name>**, and the status descriptions in **STREAMINPUTTESTC_MSGE<name>**. The source code with these messages is in **StreamInputTest.c** on line 1.91.

Uses

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>lalDebugLevel</td>
<td>LALPrintError()</td>
</tr>
<tr>
<td>LALOpenDataFile()</td>
<td>LALCheckMemoryLeaks()</td>
</tr>
<tr>
<td>LALCHARReadVectorSequence()</td>
<td>LALCHARDestroyVectorSequence()</td>
</tr>
<tr>
<td>LALI2ReadVectorSequence()</td>
<td>LALI2DestroyVectorSequence()</td>
</tr>
<tr>
<td>LALI4ReadVectorSequence()</td>
<td>LALI4DestroyVectorSequence()</td>
</tr>
<tr>
<td>LALI8ReadVectorSequence()</td>
<td>LALI8DestroyVectorSequence()</td>
</tr>
<tr>
<td>LALU2ReadVectorSequence()</td>
<td>LALU2DestroyVectorSequence()</td>
</tr>
</tbody>
</table>
LALU4ReadVectorSequence() LALU4DestroyVectorSequence()
LALU8ReadVectorSequence() LALU8DestroyVectorSequence()
LALSReadVectorSequence() LALSDestroyVectorSequence()
LALDReadVectorSequence() LALDDestroyVectorSequence()
LALCHARReadSequence() LALCHARDestroyVector()
LALI2ReadSequence() LALI2DestroyVector()
LALI4ReadSequence() LALI4DestroyVector()
LALI8ReadSequence() LALI8DestroyVector()
LALU2ReadSequence() LALU2DestroyVector()
LALU4ReadSequence() LALU4DestroyVector()
LALU8ReadSequence() LALU8DestroyVector()
LALSReadSequence() LALSDestroyVector()
LALDReadSequence() LALDDestroyVector()
LALCReadSequence() LALCDestroyVector()
LALZReadSequence() LALZDestroyVector()

Notes
32.6.7 Program StreamSeriesInputTest.c

Reads a time or frequency series from a file, and writes it to another file.

Usage

StreamSeriesInputTest [-o outfile] [-i infile stype dtype] [-d debuglevel]

Description

This test program parses data from an input file or from stdin, using the routines in StreamSeriesInput.c, and possibly generating output using the routines in StreamSeriesOutput.c. The following option flags are accepted:

- **-o** Writes the output to *outfile*. If *outfile* is given as stdout, the data is written to standard output (not to a file named stdout). If the -o flag is not given, the input routines are exercised, but no output is written.

- **-i** Specifies the input file name *infile*, series type *stype*, and base datatype *dtype*. Series type is a single character: either t (time series), v (time vector series), a (time array series), or f (frequency series). Base datatype may be i2 (INT2), i4 (INT4), i8 (INT8), u2 (UINT2), u4 (UINT4), u8 (UINT8), s (REAL4), d (REAL8), c (COMPLEX8), or z (COMPLEX16). If the -i flag is not given, -i StreamSeriesInput.dat f s is assumed (this file is provided with the distribution so that running the code with no arguments, à la make check, will perform a nontrivial test of the algorithm).

- **-d** Sets the debug level to *debuglevel*; if absent, -d 0 is assumed.

See the documentation in StreamSeriesInput.c and StreamSeriesOutput.c for discussion of the input and output data file formats.

Exit codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NORM</td>
<td>0</td>
<td>&quot;Normal exit&quot;</td>
</tr>
<tr>
<td>SUB</td>
<td>1</td>
<td>&quot;Subroutine failed&quot;</td>
</tr>
<tr>
<td>ARG</td>
<td>2</td>
<td>&quot;Error parsing arguments&quot;</td>
</tr>
<tr>
<td>FILE</td>
<td>3</td>
<td>&quot;Could not open file&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants STREAMSERIESINPUTTESTC_<name>, and the status descriptions in STREAMSERIESINPUTTESTC_MSGE_<name>. The source code with these messages is in StreamSeriesInputTest.c on line 1.82.

Uses

```c
lalDebugLevel
LALOpenDataFile()
LALCheckMemoryLeaks()
LAL<typecode>ReadTSeries()
LAL<typecode>WriteTSeries()
LAL<typecode>ReadTVectorSeries()
LAL<typecode>WriteTVectorSeries()
LAL<typecode>ReadTArraySeries()
LAL<typecode>WriteTArraySeries()
LAL<typecode>ReadFSeries()
LAL<typecode>WriteFSeries()
LAL<typecode>DestroyVector()
LAL<typecode>DestroyVectorSequence()
LAL<typecode>DestroyArraySequence()
```

where <typecode> is any of I2, I4, I8, U2, U4, U8, S, D, C, Z.

Notes
32.7 Header `StreamOutput.h`

Provides routines to write data from LAL data structures to an open stream.

**Synopsis**

```c
#include "StreamOutput.h"
```

This header provides prototypes for routines that write the contents of LAL time/frequency series structures or grid structures to a file (or other I/O) stream, in a standard format. The routines do not provide a system-level interface to create files and open or close file streams; they simply assume that they have been passed an open, writeable stream. Nonetheless, because they involve I/O stream manipulation, these routines are placed in the `lalsupport` library rather than in `lal ` proper.

**Error conditions**

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUL</td>
<td>1</td>
<td>&quot;Unexpected null pointer in arguments&quot;</td>
</tr>
<tr>
<td>PRN</td>
<td>2</td>
<td>&quot;Print statement failed&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants `STREAMOUTPUT_E<name>`, and the status descriptions in `STREAMOUTPUT_MSGE<name>`. The source code with these messages is in `StreamOutput.h` on line 1.69.

**Types**
32.7.1 Module StreamSeriesOutput.c

Writes a time or frequency series to an output stream.

Prototypes

```c
void
LAL<typecode>WriteTSeries( LALStatus *stat,
FILE *stream,
<datatype>TimeSeries *series )

void
LAL<typecode>WriteTVectorSeries( LALStatus *stat,
FILE *stream,
<datatype>TimeVectorSeries *series )

void
LAL<typecode>WriteTArraySeries( LALStatus *stat,
FILE *stream,
<datatype>TimeArraySeries *series )

void
LAL<typecode>WriteFSeries( LALStatus *stat,
FILE *stream,
<datatype>FrequencySeries *series )
```

Description

These routines write the data and metadata in a time or frequency series *series to an output stream *stream in a standard format, described below. It returns an error if any attempt to write to the stream failed; *stream may then be left in a partially-written state.

For each of these prototype templates there are in fact 10 separate routines corresponding to all the atomic datatypes <datatype> (except CHAR) referred to by <typecode>:

<table>
<thead>
<tr>
<th>&lt;typecode&gt;</th>
<th>&lt;datatype&gt;</th>
<th>&lt;typecode&gt;</th>
<th>&lt;datatype&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>I2</td>
<td>INT2</td>
<td>U2</td>
<td>UINT2</td>
</tr>
<tr>
<td>I4</td>
<td>INT4</td>
<td>U4</td>
<td>UINT4</td>
</tr>
<tr>
<td>I8</td>
<td>INT8</td>
<td>U8</td>
<td>UINT8</td>
</tr>
<tr>
<td>S</td>
<td>REAL4</td>
<td>C</td>
<td>COMPLEX8</td>
</tr>
<tr>
<td>D</td>
<td>REAL8</td>
<td>Z</td>
<td>COMPLEX16</td>
</tr>
</tbody>
</table>

Format for *stream:  The data written to the output stream will be formatted in a manner consistent with the input routines in StreamSeriesInput.c. That is, it will begin with a metadata header, consisting of multiple lines of the form:

```
# fieldname = value
```

where fieldname is the name of a field in *series and value is the value of that metadata field, in some standard format (below). The following metadata fields will be written, one per line, based on the type of *series:

* <datatype>TimeSeries: datatype, name, epoch, deltaT, f0, sampleUnits, length
* <datatype>TimeVectorSeries: datatype, name, epoch, deltaT, f0, sampleUnits, length, vectorLength
* <datatype>TimeArraySeries: datatype, name, epoch, deltaT, f0, sampleUnits, length, dimLength, arrayDim
* <datatype>FrequencySeries: datatype, name, epoch, deltaT, f0, deltaF, sampleUnits, length
After all metadata have been written, the contents of \texttt{series->data->data} will be written in standard integer or floating-point notation, according to \texttt{<datatype>}: integers will be written to full precision, while floating-point numbers will be written in exponential notation with sufficient digits to ensure that they represent a unique binary floating-point number under the IEEE Standard 754 (this means 9 digits for \texttt{REAL4}s and 17 digits for \texttt{REAL8}s). Complex datatypes are represented by pairs of floating-point numbers representing alternately the real and imaginary parts.

The body of the file will be formatted with newlines '\n' separating individual base, complex, vector, or array valued elements of the sequence \texttt{series->data}. Within each element, integer or floating-point components will be separated by single ' ' characters. Thus the value of \texttt{series->data->length} will always equal the number of lines following the metadata header.

**Format for metadata fields:** Here we summarize briefly the format for the individual field values in the metadata header.

- \texttt{datatype}: \textit{value} is a string (\textit{not} surrounded by quotes) corresponding to the type of \texttt{*series}; e.g. \texttt{COMPLEX8FrequencySeries}.

- \texttt{name}: \textit{value} is a string surrounded by quotes " representing \texttt{series->name}. Standard C-language string literal notation is used: printable characters are written directly except for " and \ (rendered as " and \ \, respectively), characters with special C escape sequences are written as those sequences (e.g. \texttt{\t} for tab and \texttt{\n} for newline), and all other character bytes are written as three-digit octal codes \texttt{\ooo}. Writing stops at the first null byte \texttt{\0}.

- \texttt{epoch}: \textit{value} is a single \texttt{INT8} number representing \texttt{series->epoch} in GPS nanoseconds.

- \texttt{deltaT} (any time series): \textit{value} is a single \texttt{REAL8} number representing \texttt{series->deltaT}.

- \texttt{f0}: \textit{value} is a single \texttt{REAL8} number representing \texttt{series->f0}.

- \texttt{deltaF} (\texttt{FrequencySeries} only): \textit{value} is a single \texttt{REAL8} number representing \texttt{series->deltaF}.

- \texttt{sampleUnits}: \textit{value} is string surrounded by quotes "; inside the quotes is a unit string corresponding to \texttt{series->sampleUnits} as converted by the routine \texttt{LALUnitAsString()}.  

- \texttt{length}: \textit{value} is a single \texttt{UINT4} representing \texttt{series->data->length}.

- \texttt{vectorLength} (\texttt{TimeVectorSeries} only): \textit{value} is a single \texttt{UINT4} representing \texttt{series->data->vectorLength}.

- \texttt{dimLength} (\texttt{TimeArraySeries} only): \textit{value} consists of a sequence of \texttt{UINT4}s separated by single ' ' characters, representing the components of \texttt{series->data->dimLength->data}. The value of \texttt{series->data->dimLength->length} must be inferred from the number of components; it is not given as separate metadata.

- \texttt{arrayDim} (\texttt{TimeArraySeries} only): \textit{value} is a single \texttt{UINT4} representing \texttt{series->data->arrayDim}. If the array sequence was properly constructed, this will equal the product of the components of \texttt{dimLength}, above.

**Algorithm**

**Uses**

- \texttt{lalDebugLevel}
- \texttt{LALPrintError()}
- \texttt{LALCHARCreateVector()}
- \texttt{LALCHARDestroyVector()}
- \texttt{LALUnitAsString()}

**Notes**
32.7.2 Module StreamGridOutput.c

Writes a LAL grid structure to an output stream.

Prototypes

```c
void LAL<typecode>WriteGrid( LALStatus *stat, FILE *stream, <datatype>Grid *grid )
```

Description

These routines write the data and metadata in a grid structure `*grid` to an output stream `*stream` in a standard format, described below. It returns an error if any attempt to write to the stream failed; `*grid` may then be left in a partially-written state.

For each of these prototype templates there are in fact 10 separate routines corresponding to all the numeric atomic datatypes `<datatype>` referred to by `<typecode>`:

<table>
<thead>
<tr>
<th>&lt;typecode&gt;</th>
<th>&lt;datatype&gt;</th>
<th>&lt;typecode&gt;</th>
<th>&lt;datatype&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>I2</td>
<td>INT2</td>
<td>U2</td>
<td>UINT2</td>
</tr>
<tr>
<td>I4</td>
<td>INT4</td>
<td>U4</td>
<td>UINT4</td>
</tr>
<tr>
<td>I8</td>
<td>INT8</td>
<td>U8</td>
<td>UINT8</td>
</tr>
<tr>
<td>S</td>
<td>REAL4</td>
<td>C</td>
<td>COMPLEX8</td>
</tr>
<tr>
<td>D</td>
<td>REAL8</td>
<td>Z</td>
<td>COMPLEX16</td>
</tr>
</tbody>
</table>

Format for `*stream`: The data written to the output stream will be formatted in a manner consistent with the input routines in StreamGridInput.c. That is, it will begin with a metadata header, consisting of multiple lines of the form:

```
# fieldname = value
```

where `fieldname` is the name of a field in `*series` and `value` is the value of that metadata field, in some standard format (below). The following metadata fields will be written, one per line:

- **datatype**: `value` is a string (not surrounded by quotes) corresponding to the type of `*grid`; e.g. `COMPLEX8Grid`.
- **name**: `value` is a string surrounded by quotes " representing `grid->name`. Standard C-language string literal notation is used: printable characters are written directly except for " and \ (rendered as \" and \\, respectively), characters with special C escape sequences are written as those sequences (e.g. \t for tab and \n for newline), and all other character bytes are written as three-digit octal codes \ooo. Writing stops at the first null byte \0.
- **sampleUnits**: `value` is string surrounded by quotes "; inside the quotes is a unit string corresponding to `grid->sampleUnits` as converted by the routine LALUnitAsString().
- **dimUnits**: `value` is a sequence of `m` strings, surrounded by quotes " and separated by a space, where `m` is the grid dimension (number of grid axes); inside the quotes is a unit string corresponding to the elements of the `grid->dimUnits` array as converted by the routine LALUnitAsString().
- **offset**: `value` is a sequence of `m` REAL8 numbers separated by single spaces, representing the elements of the `grid->offset->data`; the number of data `m` is the grid dimension and corresponds to the value of `grid->offset->length`.
- **interval**: `value` is a sequence of `m` REAL8 numbers separated by single spaces, representing the elements of the `grid->interval->data`; the number of data `m` is the grid dimension and corresponds to the value of `grid->interval->length`.
- **dimLength**: `value` is a sequence of `M` REAL8 numbers separated by single spaces, representing the elements of the `grid->data->dimLength->data`; the number of data `M` is the data dimension and corresponds to the value of `grid->data->dimLength->length`, which must be greater than or equal to the grid dimension `m`, above.
After all metadata have been written, the contents of grid->data->data will be written in standard integer or floating-point notation, according to <datatype>: integers will be written to full precision, while floating-point numbers will be written in exponential notation with sufficient digits to ensure that they represent a unique binary floating-point number under the IEEE Standard 754 (this means 9 digits for REAL4s and 17 digits for REAL8s).

The input format in StreamGridInput.c does not specify how the numerical data is to be arranged, other than that the numbers be separated by whitespace, and that complex datatypes be represented by alternating real and imaginary components. These routines adopt the following conventions to improve human-readability: If the data dimension is equal to the grid dimension, then each line consists of a single datum (either a single number, or, for complex datatypes, a pair of numbers separated by whitespace), followed by a newline '\n'. If the data dimension is greater than the grid dimension, then each line will consist of a number of data equal to the length of the last dimension in grid->data->dimLength. If the data dimension is at least two greater than the grid dimension, and the dimension lengths are such that a single grid point comprises multiple lines of data, then an additional blank line '\n' is inserted to separate subsequent grid points.

Algorithm

Uses

<table>
<thead>
<tr>
<th>1alDebugLevel</th>
<th>LALPrintError()</th>
</tr>
</thead>
<tbody>
<tr>
<td>LALCHARCreatetVector()</td>
<td>LALCHARDestroyVector()</td>
</tr>
<tr>
<td>LALUnitAsString()</td>
<td></td>
</tr>
</tbody>
</table>

Notes
32.8 Header LALInitBarycenter.h

Provides a routine for reading Earth and Sun position information from data files.

Synopsis

#include "LALInitBarycenter.h"

This header covers the routine LALInitBarycenter.c. Since it involves file I/O, it is placed in the support package, and included in the lalsupport library.

Error conditions

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPEN</td>
<td>1</td>
<td>&quot;Could not open ephemeris file&quot;</td>
</tr>
<tr>
<td>MEM</td>
<td>2</td>
<td>&quot;Out of memory&quot;</td>
</tr>
<tr>
<td>EPHFILE</td>
<td>32</td>
<td>&quot;Error in reading an ephemeris file&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants LALINITBARYCENTERH_E<name>, and the status descriptions in LALINITBARYCENTERH_MSGE<name>. The source code with these messages is in LALInitBarycenter.h on line 1.66.
32.8.1 Module LALInitBarycenter.c

Reads Earth and Sun position information from data files.

Prototypes

```c
void LALInitBarycenter(LALStatus *stat, EphemerisData *edat)
```

Description

LALInitBarycenter() fills the contents of edat from data read from data files. See LALBarycenter.h in the pulsar package for the definition of the EphemerisData structure.

The function reads in two data files (specified in the edat->ephiles structure) that contain the position, velocity, and acceleration of the Earth and Sun, respectively, at regular intervals throughout the specified year. E.g., for 1998, the two files are earth98.dat and sun98.dat. These files are derived from the JPL DE405 ephemeris and are provided by Cutler. The first line of these files specifies the start time, sampling interval, and number of datapoints stored in each file, which are used to allocate data arrays edat->ephemE and edat->ephemS of appropriate length. LALInitBarycenter() should be called once near the beginning of the larger analysis package, and the fields edat->ephemE and edat->ephemS should be freed with LALFree() near the end. See the LALBarycenterTest program in the pulsar package for an illustration of how this routine is used.

Uses

LALOpenDataFile()
LALMalloc()
LALFree()

Notes

Author: Cutler, C.

$Id: LALInitBarycenter.c,v 1.11 2007/10/04 16:07:11 reinhard Exp $
32.9 Header LALXMGRInterface.h

Provides prototypes, structures and functions to allow visualisation of the events generated findchirp and the inspiral shared object.

Synopsis

#include <lal/LALXMGRInterface.h>
### Error codes

<table>
<thead>
<tr>
<th>name</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>1</td>
<td>&quot;Null pointer&quot;</td>
</tr>
<tr>
<td>NNUL</td>
<td>2</td>
<td>&quot;Non-null pointer&quot;</td>
</tr>
<tr>
<td>ALOC</td>
<td>3</td>
<td>&quot;Memory allocation error&quot;</td>
</tr>
<tr>
<td>OPEN</td>
<td>4</td>
<td>&quot;Error opening file&quot;</td>
</tr>
<tr>
<td>FCLO</td>
<td>5</td>
<td>&quot;Error closing file&quot;</td>
</tr>
<tr>
<td>NGRA</td>
<td>6</td>
<td>&quot;Already have max number of graphs in array&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants `LALXMGRINTERFACEH_E<name>`, and the status descriptions in `LALXMGRINTERFACEH_MSGE<name>`. The source code with these messages is in `LALXMGRInterface.h` on line 1.85.

### Types

Author: Brady P., R., and Brown, D. A.

$Id: LALXMGRInterface.h,v 1.4 2007/06/08 14:41:54 bema Exp$
32.9.1 Module LALXMGRInterface.c

Functions for creating XMGR graphs from LAL structures and functions.
32.10 Header LIGOLwXML.h

Provides functions for writing the LIGO metadata database table structures to LIGO light weight XML files.

Synopsis

#include <lal/LIGOLwXML.h>

Error conditions

<table>
<thead>
<tr>
<th>name</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>1</td>
<td>&quot;Null pointer&quot;</td>
</tr>
<tr>
<td>NNUU</td>
<td>2</td>
<td>&quot;Non-null pointer&quot;</td>
</tr>
<tr>
<td>ALOC</td>
<td>3</td>
<td>&quot;Memory allocation error&quot;</td>
</tr>
<tr>
<td>UTAB</td>
<td>4</td>
<td>&quot;Unknown metadata table type&quot;</td>
</tr>
<tr>
<td>OPEN</td>
<td>5</td>
<td>&quot;Error opening XML file&quot;</td>
</tr>
<tr>
<td>CLOS</td>
<td>6</td>
<td>&quot;Closing an XML file with an open table&quot;</td>
</tr>
<tr>
<td>BGNT</td>
<td>7</td>
<td>&quot;Begining a table without ending previous table&quot;</td>
</tr>
<tr>
<td>NTAB</td>
<td>8</td>
<td>&quot;No table type specified&quot;</td>
</tr>
<tr>
<td>ENDT</td>
<td>9</td>
<td>&quot;Ending a table without an beginning a table&quot;</td>
</tr>
<tr>
<td>TNSM</td>
<td>10</td>
<td>&quot;Table type mismatch&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants LIGOLwXMLH_\langle name\rangle, and the status descriptions in LIGOLwXMLH_MSGE\langle name\rangle. The source code with these messages is in LIGOLwXML.h on line 1.94.

Structures

Type tagLIGOLwXMLStream

typedef struct tagLIGOLwXMLStream
{
    LALFILE *fp;
    INT4 first;
    UINT8 rowCount;
    MetadataTableType table;
} LIGOLwXMLStream;

This structure contains the file stream and current table type for writing to LIGO lightweight XML files. It should not be manipulated directly, but passed to the LIGOLwXML functions for their use.

fp The file stream pointer of the XML file.
first Is this the first entry in the table.
rowCount Counter for the number of rows in the current table.
table The database table currently open.
32.10.1 Module LIGOLwXML.c

Routines to write LIGO metadata database structures to LIGO lightweight XML files.

Prototypes

```c
void LALOpenLIGOLwXMLFile (LALStatus *status,
                           LIGOLwXMLStream *xml,
                           const CHAR *path)
```

```c
void LALCloseLIGOLwXMLFile (LALStatus *status,
                            LIGOLwXMLStream *xml)
```

```c
void LALBeginLIGOLwXMLTable (LALStatus *status,
                              LIGOLwXMLStream *xml,
                              MetadataTableType table)
```

```c
void LALEndLIGOLwXMLTable (LALStatus *status,
                         LIGOLwXMLStream *xml)
```

```c
void LALWriteLIGOLwXMLTable (LALStatus *status,
                           LIGOLwXMLStream *xml,
                           MetadataTable tablePtr,
                           MetadataTableType table)
```

Description

The routine `LALOpenLIGOLwXMLFile` calls the C standard library function `fopen` to open a file specified by the `path` argument. The file is truncated to zero length if already exists. The standard LIGO lightweight XML header, `LIGOLW_XML_HEADER` given in LIGOLwXMLHeaders.h, is then written to the file and the the pointer to the file stream is returned in the `xml->fp` argument.

The routine `LALCloseLIGOLwXMLFile` prints the standard LIGO lightweight XML footer, `LIGOLW_XML_FOOTER` given in LIGOLwXMLHeaders.h, and closes the file stream pointed to by `xml->fp`.

The routine `LALBeginLIGOLwXMLTable` prints the table header. The type of table to begin is specified by the `table` argument. The appropriate headers are again contained in LIGOLwXMLHeaders.h and contain the table name as well as the names and data types of each of the columns in the table. In addition, it sets `xml->first` to 1 and `xml->table` to the requested table.

The routine `LALEndLIGOLwXMLTable` prints the table footer. This is the same for all tables, and given by `LIGOLW_XML_TABLE_FOOTER` in LIGOLwXMLHeaders.h. Additionally, `xml->table` is set to `no_table`.

The routine `LALWriteLIGOLwXMLTable` writes the content of the xml table. The type of table to be written is specified by `table`. The contents of the table should be stored as a linked list in `tablePtr->table`. The data is written using the row format for the specified table given in LIGOLwXMLHeaders.h.

Algorithm

None.
Uses

`fopen()` `fprintf()` `fclose()`

Notes

In order to change a table definition in LAL, changes must be made in several places. It is necessary to update the structure which is used to store the information in memory as well as the reading and writing codes. Below is a list of all the files which must be updated.

- Update the LAL table definition in `LIGOMetaDataTables.h`
- Update the LIGOLwXML writing code:
  1. Change the table header written at to the LIGOLwXML file. This is `#defined` in `LIGOLWXMLHeaders.h`. For example, to change the `sngl_inspiral` table, you must edit `LIGOLW_XML_SNGL_INSPIRAL`.
  2. Change the row format of the LIGOLwXML file. This is `#defined` in `LIGOLWXMLHeaders.h`. For example, to change the `sngl_inspiral` table, you must edit `SNGL_INSPIRAL_ROW`.
  3. Change the `fprintf` command which writes the table rows. This is contained in `LIGOLwXML.c`.
- Update the LIGOLwXML reading code:
  1. Add/remove columns from the table directory of the table in question. This is contained in `LIGOLwXMLRead.c`, either in `LALCreateMetaTableDir` or in the specific reading function.
  2. Check that all columns read in from the XML table are stored in memory. This requires editing the table specific reading codes in `LIGOLwXMLRead.c`.
32.11 Header LIGOLwXMLHeaders.h

Synopsis

#include <lal/LIGOLwXMLHeaders.h>

This header provides \texttt{#defines} for the common elements of LIGO light weight XML files. It provides the XML header and footer, as well as table definitions for the various metadata tables. It will need to be kept up to date with changes in the LIGO database table definitions. The quantities which are defined in this file are

- LIGOLW\_XML\_HEADER
- LIGOLW\_XML\_FOOTER
- LIGOLW\_XML\_TABLE\_FOOTER
- LIGOLW\_XML\_PROCESS
- PROCESS\_ROW
- LIGOLW\_XML\_PROCESS\_PARAMS
- PROCESS\_PARAMS\_ROW
- LIGOLW\_XML\_SEARCH\_SUMMARY
- SEARCH\_SUMMARY\_ROW
- LIGOLW\_XML\_SEARCH\_SUMMVARS
- SEARCH\_SUMMVARS\_ROW
- LIGOLW\_XML\_SNGL\_BURST
- SNGL\_BURST\_ROW
- LIGOLW\_XML\_SIM\_BURST
- SIM\_BURST\_ROW
- LIGOLW\_XML\_SIM\_RINGDOWN
- SIM\_RINGDOWN\_ROW
- LIGOLW\_XML\_SUMM\_VALUE
- SUMM\_VALUE\_ROW
- LIGOLW\_XML\_SIM\_INST\_PARAMS
- SIM\_INST\_PARAMS\_ROW
- LIGOLW\_XML\_STOCHASTIC
- STOCHASTIC\_ROW
- LIGOLW\_XML\_STOCH\_SUMM
- STOCH\_SUMM\_ROW
- LIGOLW\_XML\_EXT\_TRIGGERS
- EXT\_TRIGGERS\_ROW
32.12 Header **LIGOLwXMLRead.h**

Provides functions for reading LIGO lightweight XML files to LIGO metadata database tables.

**Synopsis**

```
#include <lal/LIGOLwXMLRead.h>
```

**Error conditions**

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>1</td>
<td>&quot;Null pointer&quot;</td>
</tr>
<tr>
<td>NNUL</td>
<td>2</td>
<td>&quot;Non-null pointer&quot;</td>
</tr>
<tr>
<td>ALOC</td>
<td>3</td>
<td>&quot;Memory allocation error&quot;</td>
</tr>
<tr>
<td>UTAB</td>
<td>4</td>
<td>&quot;Unknown metadata table type&quot;</td>
</tr>
<tr>
<td>NCOL</td>
<td>5</td>
<td>&quot;Unable to find table column&quot;</td>
</tr>
<tr>
<td>NTAB</td>
<td>6</td>
<td>&quot;Requested table not found in file&quot;</td>
</tr>
<tr>
<td>PARS</td>
<td>7</td>
<td>&quot;Error parsing table&quot;</td>
</tr>
<tr>
<td>MTAB</td>
<td>8</td>
<td>&quot;No table type specified&quot;</td>
</tr>
<tr>
<td>ENDT</td>
<td>9</td>
<td>&quot;Ending a table without an beginning a table&quot;</td>
</tr>
<tr>
<td>TMSM</td>
<td>10</td>
<td>&quot;Table type mismatch&quot;</td>
</tr>
<tr>
<td>TNOP</td>
<td>11</td>
<td>&quot;Table not begun for writing&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants `LIGOLWXMLREADH_<name>`, and the status descriptions in `LIGOLWXMLREADH_MSGE_<name>`. The source code with these messages is in `LIGOLwXMLRead.h` on line 1.101.

**Structures**

**Type** `tagMetaTableDirectory`

```
typedef struct tagMetaTableDirectory
{
  const CHAR *name;
  INT4  pos;
  INT4  idx;
}
MetaTableDirectory;
```

This structure allows for the association of entries in a MetaDataTable with columns in an xml file.

- **name** The name of the column in the XML table.
- **pos** The position of this column in the XML table.
- **idx** The id number of the column.
32.12.1 Module CreateMetaTableDir.c

Routines to create a directory of a LIGO LW XML file.

Prototypes

```c
MetaTableDirectory* XLALCreateMetaTableDir(
    const MetaioParseEnv env,
    MetadataTableType table
);

void
LALCreateMetaTableDir(
    LALStatus *status,
    MetaTableDirectory **tableDir,
    const MetaioParseEnv env,
    MetadataTableType table
);
```

Description

The routine LALCreateMetaTableDir constructs a MetaTableDirectory for a given LIGOLwXML table. It determines the location of each column expected to be present in the XML table and populates the pos field with this information. This then allows other routines to parse the contents of an XML file and correctly interpret the entries. When reading these tables, a call is made to LALCreateMetaTableDir. For all other tables, the directory is constructed internally by the reading code.

Algorithm

None.

Uses

Functions in the Metaio library:

- MetaioFindColumn
- MetaioGetRow
- MetaioOpenTable
- MetaioClose

Notes

Author: Brown, D. A., and Brady, P. R.

$Id: CreateMetaTableDir.c,v 1.11 2008/01/24 07:20:45 kipp Exp$
32.12.2 Module LIGOLwXMLRead.c

Routines to write LIGO metadata database structures to LIGO lightweight XML files.

Prototypes

```c
ProcessTable * XLALProcessTableFromLIGOLw (CHAR *fileName);
ProcessParamsTable * XLALProcessParamsTableFromLIGOLw (CHAR *fileName);
MultiInspiralTable * XLALMultiInspiralTableFromLIGOLw (CHAR *fileName);
void LALSnglBurstTableFromLIGOLw (LALStatus *status, SnglBurstTable **eventHead, CHAR *fileName);
int LALSnglInspiralTableFromLIGOLw (SnglInspiralTable **eventHead, CHAR *fileName, INT4 startEvent, INT4 stopEvent);
int InspiralTmpltBankFromLIGOLw (InspiralTemplate **bankHead, CHAR *fileName, INT4 startTmplt, INT4 stopTmplt);
int SimInspiralTableFromLIGOLw (SimInspiralTable **simHead, CHAR *fileName, INT4 startTime, INT4 endTime);
SearchSummaryTable * XLALSearchSummaryTableFromLIGOLw (const CHAR *fileName);
int SummValueTableFromLIGOLw (SummValueTable **sumHead, CHAR *fileName);
```

int LALStochasticTableFromLIGOLw (StochasticTable **stochHead, CHAR *fileName)

int LALStochSummTableFromLIGOLw (StochSummTable **stochSummHead, CHAR *fileName)

int LALExtTriggerTableFromLIGOLw (ExtTriggerTable **eventHead, CHAR *fileName, INT4 startEvent, INT4 stopEvent)

Description

The routine LALSnglBurstTableFromLIGOLw reads in a single_burst table from a LIGOLwXML file specified in fileName; eventHead provides a pointer to the head of a linked list of SnglBurstTables containing the events. The routine is passed the fileName of an XML file containing a single_burst table. First, the table is opened using MetaioOpenTable. Then a directory of the table is generated using LALCreateMetaTableDir. Rows of the table are read in sequentially from the file. Each entry in the row is stored in the appropriate entry of a SnglBurstTable which is appended to the end of a linked list of such tables. When all rows have been read in, the file is closed using MetaioClose. eventHead is set to point to the head of the linked list of SnglBurstTables.

The routine LALSnglInspiralTableFromLIGOLw reads in a sngl_inspiral table from the LIGOLwXML file specified in fileName. It returns the number of triggers read in and eventHead provides a pointer to the head of a linked list of SnglInspiralTables containing the events. It will return all events between the startEvent and stopEvent; if these are set to 0 and -1 respectively, all events are returned.

The routine InspiralTmpltBankFromLIGOLw reads in a sngl_inspiral table from the LIGOLwXML file specified in fileName. It returns the number of templates read in and bankHead provides a pointer to the head of a linked list of InspiralTemplates containing the templates read in. It will return all events between the startTmplt and stopTmplt; if these are set to 0 and -1 respectively, all events are returned. Although a sngl_inspiral table is read in, only those entries relevant for an InspiralTemplate are read in and stored.

The routine SimInspiralTableFromLIGOLw reads in a sim_inspiral table from the LIGOLwXML file specified in fileName. It returns the number of rows read in and SimHead provides a pointer to the head of a linked list of SimInspiralTables containing the events. Additionally, a startTime and endTime are specified. Only simulated events occurring between these times are returned. If the endTime is set to zero, then all events are returned.

The routine XLALSearchSummaryTableFromLIGOLw reads in a search_summary table from the LIGOLwXML file specified in fileName. It returns a pointer to the head of a linked list of SearchSummaryTables.

The routine SummValueTableFromLIGOLw reads in a summ_value table from the LIGOLwXML file specified in fileName. It returns the number of rows read in and sumHead provides a pointer to the head of a linked list of SummValueTables.

The routine LALStochasticTableFromLIGOLw reads in a stochastic_table table from the LIGOLwXML file specified in fileName. It returns the number of rows read in and stochHead provides a pointer to the head of a linked list of StochasticTables.

The routine LALStochSummTableFromLIGOLw reads in a stoch_summ_table table from the LIGOLwXML file specified in fileName. It returns the number of rows read in and stochSummHead provides a pointer to the head of a linked list of StochSummTables.

Algorithm

None.
Uses

Functions in the Metaio library:

- MetaioFindColumn
- MetaioGetRow
- MetaioOpenTable
- MetaioClose

Notes
32.12.3 Module LIGOLwXMLRingdownRead.c

Routines to read the various ringdown search XML data into LAL structures.

Prototypes

\begin{verbatim}
SnglRingdownTable* XLALSnglRingdownTableFromLIGOLw (CHAR *fileName);

SimRingdownTable* XLALSimRingdownTableFromLIGOLw (CHAR *fileName, INT4 startTime, INT4 stopTime);

INT4 XLALReadRingdownTriggerFile (SnglRingdownTable **ringdownEventList, SnglRingdownTable **lastTrigger, SearchSummaryTable **searchSummList, SearchSummvarsTable **inputFileList, CHAR *fileName);
\end{verbatim}

Description

Algorithm
None.

Uses

Functions in the Metaio library:

- MetaioFindColumn
- MetaioGetRow
- MetaioOpenTable
- MetaioClose

Notes

Author: Brown, D. A. and Goggin, L. M.

$Id: LIGOLwXMLRingdownRead.c,v 1.9 2007/06/08 14:41:55 bema Exp$
32.13 Header SegmentsIO.h

Provides segment list reading and writing functions as part of the lalsupport library.

Synopsis

```
#include <lal/Segments.h>
#include <lal/SegmentsIO.h>
```

Notes

The baseline format of a segment list file is described at

Error codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>1</td>
<td>&quot;Null pointer passed to function&quot;</td>
</tr>
<tr>
<td>INVAL</td>
<td>2</td>
<td>&quot;LALSegList structure was not properly initialized&quot;</td>
</tr>
<tr>
<td>BADOPT</td>
<td>3</td>
<td>&quot;Invalid option letter in options string&quot;</td>
</tr>
<tr>
<td>NOFMT</td>
<td>4</td>
<td>&quot;No output format specified in options string&quot;</td>
</tr>
<tr>
<td>OPENR</td>
<td>5</td>
<td>&quot;Error opening segment list file for reading&quot;</td>
</tr>
<tr>
<td>OPENW</td>
<td>6</td>
<td>&quot;Error opening segment list file for writing&quot;</td>
</tr>
<tr>
<td>FMT</td>
<td>7</td>
<td>&quot;Segment list file is not in a recognized format&quot;</td>
</tr>
<tr>
<td>PARSE</td>
<td>8</td>
<td>&quot;Parsing error while reading from file&quot;</td>
</tr>
<tr>
<td>DOM</td>
<td>9</td>
<td>&quot;GPS times do not represent a valid segment&quot;</td>
</tr>
<tr>
<td>INT</td>
<td>10</td>
<td>&quot;Internal error in SegmentsIO module&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants SEGMENTSIOH_E<name>, and the status descriptions in SEGMENTSIOH_MSGE<name>. The source code with these messages is in SegmentsIO.h on line 1.66.
32.13.1 Module SegmentsIO.c

Segment list input/output routines in the lalsupport library.

Prototypes

```c
void LALSegListRead( LALStatus *status, LALSegList *seglist, const CHAR *fileName, const CHAR *options )

void LALSegListWrite( LALStatus *status, LALSegList *seglist, const CHAR *fileName, const CHAR *options )
```

Description

Reading a segment list file

The function `LALSegListRead()` reads a segment list from a file and appends the segments to the specified segment list. The segment list must previously have been initialized. If it already has some segments in it, then they are retained.

If the segment read from the file includes the optional 'id' at the beginning of the line, that is recorded in the segment list. Otherwise, an 'id' of 0 is recorded. Any additional information about the segment (appearing on the line after the stop time) is ignored.

The function syntax includes an 'options' argument, but no reading options are currently implemented. In the meantime, the user may pass either a null pointer or a CHAR pointer to a null string.

Writing a segment list file

The function `LALSegListWrite()` writes a segment list to a file with the specified file name, following the standard format described at [http://www.lsc-group.phys.uwm.edu/daswg/docs/technical/seglist_format.html](http://www.lsc-group.phys.uwm.edu/daswg/docs/technical/seglist_format.html). If the file already exists, it is overwritten.

The function syntax includes an 'options' argument which determines how the segments are written out. This argument must be a non-null CHAR pointer to a string. The string must contain one or more of the following lowercase letters:

<table>
<thead>
<tr>
<th>Character</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Causes the segment list to be written out in ASCII format. <strong>Currently, this is the only format supported, so a must appear in the options string.</strong> In the future, some other format (e.g. XML) could be supported.</td>
</tr>
<tr>
<td>i</td>
<td>Write the 'id' of each segment to the file, appearing before the GPS start and end times on the line. If this option is not requested, then the GPS start and end times will be the first things on the line.</td>
</tr>
<tr>
<td>d</td>
<td>Include the duration (in seconds) of the segment on each line of the file, appearing after the GPS start and end times on the line.</td>
</tr>
</tbody>
</table>
32.13.2 Program SegmentsIOTest.c

Tests the routines in SegmentsIO.c.

Usage

SegmentsIOTest [ lalDebugLevel ]

The default value of lalDebugLevel is 4.

If the lalDebugLevel argument is omitted, the test program sets it to 4 to turn on info messages only. Note that this default value does not enable LAL/XLAL error messages, since many of the tests intentionally create error conditions and verify that the proper error codes are generated. If you want to turn on the LAL/XLAL error and warning messages, specify a lalDebugLevel value of 7, or 23 if you also want informational messages related to memory checking. If you specify 0, then no messages will be printed under any circumstances. However, in all cases, the return status of the program will be 0 if all tests passed, 1 if one or more tests failed.

Description

NOTE: This test program does not currently do an exhaustive test of functionality and failure modes; it is more like a starting point for spot-checking by modifying, recompiling and running this test program and inspecting the output.

Exit codes

0 if all tests passed.
1 if one or more tests failed.

Uses

LALSegListRead()
LALSegListWrite()

Notes

* Author: Peter Shawhan
* Revision: $Id: SegmentsIOTest.c,v 1.2 2007/06/08 14:41:56 bema Exp $
32.14 Header `ConfigFile.h`

Routines for general config-file reading.

Synopsis

```
#include <lal/ConfigFile.h>
```

Error codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>1</td>
<td>&quot;Arguments contained an unexpected null pointer.&quot;</td>
</tr>
<tr>
<td>FILE</td>
<td>2</td>
<td>&quot;File error.&quot;</td>
</tr>
<tr>
<td>VAR</td>
<td>3</td>
<td>&quot;Config variable not found.&quot;</td>
</tr>
<tr>
<td>FMT</td>
<td>4</td>
<td>&quot;Config variable not readable using given format-string.&quot;</td>
</tr>
<tr>
<td>TOKENS</td>
<td>5</td>
<td>&quot;The input ConfigData seems corrupted.&quot;</td>
</tr>
<tr>
<td>NONULL</td>
<td>6</td>
<td>&quot;Output pointer is not NULL&quot;</td>
</tr>
<tr>
<td>UNKNOWN</td>
<td>8</td>
<td>&quot;Unknown config-file entry found&quot;</td>
</tr>
<tr>
<td>MEM</td>
<td>9</td>
<td>&quot;Out of memory&quot;</td>
</tr>
<tr>
<td>BOOL</td>
<td>10</td>
<td>&quot;Illegal BOOLEAN entry&quot;</td>
</tr>
<tr>
<td>STRING</td>
<td>11</td>
<td>&quot;Malformed quoted string&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants `CONFIGFILEH_E<name>`, and the status descriptions in `CONFIGFILEH_MSGE<name>`. The source code with these messages is in `ConfigFile.h` on line 1.86.
32.14.1 Module ConfigFile.c

Some general-purpose routines for config-file reading

Prototypes

Description

This module provides routines for reading formatted config-files containing definitions of the form
variable = value. The general syntax is somewhat similar to the one provided by the perl-module
ConfigParser (cf. http://www.python.org/doc/current/lib/module-ConfigParser.html) but (current-
ly) without the possibility of ”chapters”. Comments are allowed using either ’#’, ’;’ or ’standard line-
continuation using a ”\” at the end of the line. Also note that ’#’, ’;’ or ’treated as comment-characters.
The general syntax is best illustrated using a simple example:

# comment line
var1 = 1.0 ; you can also comment using semi-colons
somevar = some text.\n    You can also use
    line-continuation
var3 = 4    # whatever that means
note = "this is also possible, and # here does nothing"
a_switch = true #possible values: 0,1,true,false,yes,no, case insensitive
# etc etc.

Note that TABS generally get replaced by a single space, which can be useful in the case of line-
continuation (see example). All leading and trailing spaces in are ignore (except within double-quotes).
The general approach of reading from such a config-file, is to first call
LALLoadConfigFile(status, LALConfigData *cfg, FILE *fp), which loads and pre-parses the contents
of the config-file into the structure LALConfigData. Then one can then read in config-variables either using
one of the custom-wrappers:
LALReadConfig<TYPE>Variable(status, <TYPE> *cvar, LALConfigData *cfg, CHAR *varname) or the
general-purpose reading function:
LALReadConfigVariable(status, void *cvar, LALConfigData *cfg, LALConfigVar *var)

A boolean variable read by LALReadConfigBOOLVariable() can have any of the values
{1, 0, yes, no, true, false}, where the comparison is done case-insensitively, i.e. you also use ”True”
or ”FALSE”....

If one wishes a “tight” syntax for the config-file, one can check that there are no ”illegal” entries in the
config-file. This is done by checking at the end that all config-file entries have been successfully parsed,
using:
LALCheckConfigReadComplete (status, LALConfigData *cfg, INT2 strictness), where strictness
is either CONFIGFILE_WARN or CONFIGFILE_ERROR. In the first case only a warning is issued, while in the
second it is treated as a LAL-error if some config-file entries have not been read-in. (The use of this function
is optional).

The configfile-data should be freed at the end using
void LALDestroyParsedDataFile(LALStatus *status, LALConfigData *cfg).

Algorithm

Uses

LALCHARReadSequence()
LALCreateTokenList() LALDestroyTokenList()
LALCalloc() LALMalloc() LALFree()
LALPrintError() LALOpenDataFile() fclose()

Notes

LALReadConfigSTRINGVariable() and LALReadConfigSTRINGVariable() are not the same as using "%s"
as a format string, as they read the rest of the logical line (excluding comments) as a string.
In the case of \texttt{LALReadConfigSTRINGVariable()}, the required memory is allocated and has to be freed by the caller, while for \texttt{LALReadConfigSTRINGVariable()} the caller has to provide a \texttt{CHARVector} of length $N$, which defines the maximum length of string to be read.

\textbf{Note:} instead of using these functions directly, it might be more convenient to use the \texttt{UserInput} infrastructure (cf. ??).
32.14.2 Program ConfigFileTest.c

Tests the routines in ConfigFile.h.

Usage

ConfigFileTest

Description

Do some standard-tests for the config-file reading routines. No extensive error-condition checking is done here, we only check if the basic functionality works.

Exit codes

<table>
<thead>
<tr>
<th>name</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NORM</td>
<td>0</td>
<td>&quot;Normal exit&quot;</td>
</tr>
<tr>
<td>FLOAT</td>
<td>1</td>
<td>&quot;Read-in REAL8 variable is not what it should be...&quot;</td>
</tr>
<tr>
<td>INT</td>
<td>2</td>
<td>&quot;Read-in INT4 variable is not what it should be...&quot;</td>
</tr>
<tr>
<td>BOOL</td>
<td>3</td>
<td>&quot;Read-in BOOL variable is not what it should be...&quot;</td>
</tr>
<tr>
<td>STRING</td>
<td>4</td>
<td>&quot;Read-in STRING-variable is not what it should be...&quot;</td>
</tr>
<tr>
<td>SUB</td>
<td>5</td>
<td>&quot;Error occurred in sub-routine&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants CONFIGFILETESTC_E<name>, and the status descriptions in CONFIGFILETESTC_MSGE<name>. The source code with these messages is in ConfigFileTest.c on line 177.

Uses

Notes
32.15 Header **LALMathematica.h**

Provides structures, functions and macro definitions for modules that generate *Mathematica*® notebooks. Currently, the only modules using this header file are **LALMath3DPlot()**, which generates 3D animated plots of template banks having three parameters and **LALMathNDFPlot()** which plots the 3-dimensional projections of a bank that is N-dimensional.

**Synopsis**

```c
#include <lal/LALMathematica.h>
```

This header file defines macros containing *Mathematica*® syntax that is otherwise messy to implement into C source files. Here is how to use these macros to make your own program generate a *Mathematica*® notebook.

1. Open a file with a pointer named “nb” and a file extension “.nb”.
2. Use **BEG_NOTEBOOK** to start the notebook file.
3. Use the appropriate **BEG** and **END** macros with `fprintf(nb, “Your Text”)` in between to write your text to the cells of the notebook. If you are writing *Mathematica*® commands use the INPUT macros; for plain text, use TEXT Macros.
4. Denote titles and sections with the appropriate macros.
5. Use **END_NOTEBOOK** to end the notebook and use `fclose(nb)` to close the file “nb”.

The result is very readable/changeable source similar in style to most markup languages. An example program might look like:

```c
FILE *nb;
nb = fopen("YourFileName.nb", "rw");
BEG_NOTEBOOK;
BEG_TITLECELL;
fprintf(nb, "Sample Program Title");
END_TITLECELL;
BEG_SECTIONCELL;
fprintf(nb, "Sample Program Section Name");
END_SECTIONCELL;
BEG_NOTEBOOK;
fclose(nb);
```

* Author: Hanna, C. R.
* $Id: LALMathematica.h,v 1.15 2007/06/08 14:41:54 bema Exp $
Error codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>1</td>
<td>&quot;NULL pointer to a LALMathematica.h input structure&quot;</td>
</tr>
<tr>
<td>FILE</td>
<td>2</td>
<td>&quot;Could not open file to write a Mathematica Notebook&quot;</td>
</tr>
<tr>
<td>VAL</td>
<td>3</td>
<td>&quot;Invalid parameter value&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants LALMATHEMATICAH_E<name>, and the status descriptions in LALMATHEMATICAH_MSGE<name>. The source code with these messages is in LALMathematica.h on line 1.165.

Macros

```c
#define BEG_NOTEBOOK fprintf(nb, "Notebook[{|\\n"
#define END_NOTEBOOK fprintf(nb, "]|\\n")
#define BEG_TITLECELL fprintf(nb, "Cell["\n"
#define END_TITLECELL_ fprintf(nb, "]", "Title"]|\n"
#define BEG_GROUPCELL fprintf(nb, "Cell[CellGroupData[{|\\n"
#define END_GROUPCELLC fprintf(nb, "]}, Closed ]]|\n"
#define BEG_SECTIONCELL fprintf(nb, "Cell["\n"
#define END_SECTIONCELL_ fprintf(nb, "]", "Section"]|\n"
#define BEG_INPUTCELL fprintf(nb, "BoxData[\\")
#define END_INPUTCELL_ fprintf(nb, "]", "Input"]|\n"
#define BEG_TEXTCELL fprintf(nb, "\"
#define END_TEXTCELL_ fprintf(nb, "]", "Text"]|\n"
```

See the source file Math3DPlot.c for an example of how to use these macros to generate a Mathematica® notebook in your own program.

NOTEBOOK Denotes the beginning and ending of the notebook file. A BEG_NOTEBOOK tag must start the file and an END_NOTEBOOK tag must end it.

TITLE Placing an fprintf(nb, “Your Title”) between BEG and END tags will place a title font cell in the notebook.

GROUP Cells placed in between these tags will be grouped together

SECTION Same as title except the text printed will be in section font. Subsequent input and text cells following the END_SECTIONCELL tag will be grouped with that section until a new BEG_SECTIONCELL tag is encountered.

INPUT provides cells to input Mathematica® commands.

TEXT provides cells to input plain text.

Notice that the file pointer must be named “nb” in order to use the macros defined in this header. When grouping several cell objects together the last object in the list should have an underscored END tag instead of an END tag without an underscore. Although the notebook will compile (usually) if you use the tags without an ending underscore, the dangling comma is taken as a null member of the list of grouped cells. Therefore, when you view the notebook in Mathematica® you may see the word “NULL” printed on a line. That is an indication that you should use the underscore version of the tag which preceded the “NULL” statement.
Types

```c
typedef struct Math3DPointList{
    struct Math3DPointList *next;
    REAL4 x;
    REAL4 y;
    REAL4 z;
    REAL4 grayLevel;
}Math3DPointList;

typedef struct MathNDPointList{
    struct MathNDPointList *next;
    REAL4Vector *coordinates;
    INT4 dimension;
    REAL4 grayLevel;
} MathNDPointList;
```

The Math3DPointList type is used by `Math3DPlot.c` as an input structure to plot 3-dimensional template banks. It is a linked list with parameters for each coordinate x,y,z and a next pointer. It also has a parameter called grayLevel which must be $\epsilon[0,1]$. It specifies the shading of the point in the final plot with 0 representing black and 1 representing white. By creatively assigning its value the grayscale shade of the points may convey additional information.

The MathNDPointList type is similar except the coordinates are stored as data in the REAL4Vector `coordinates`.

Notes

- Obviously the definitions and functions associated with this header are NOT lal compliant and thus do not belong in any lal routines except test programs.
- There are many more commands to manipulate the structure of `Mathematica®` notebooks that are not included in this header. The macros are only what is necessary for a bare minimum interface.
Figure 32.1: Here is an example template bank produced by running InspiralSpinBankTest.c to generate roughly 5000 templates. Currently the plot doesn’t show the contour of the templates; it renders them as spheres. In the case of metrics with disimilar scales along the principle directions you will notice considerable space between points accordingly.

### 32.15.1 Module LALMath3DPlot.c

#### Prototypes

```c
void LALMath3DPlot( LALStatus *stat,
    Math3DPointList *first,
    INT4 *ntiles,
    REAL4 *pointSize)
```

*stat LALStatus structure pointer

*first Math3DPointList stucture pointer

*ntiles INT4 pointer to the number of templates you plan to plot. This may be called as NULL. If it is called with a value this function will check to see if the Math3DPointList has the correct number of templates. If it does not a warning will be printed.

pointSize REAL4 $\epsilon [0,1]$ which specifies the relative size of each point to the final display area. (e.g. 1 would fill the entire plot.) This may be called as NULL and a calculated value will be assigned. (It’s only a rough guess)

#### Description

This module contains a function for plotting 3D template banks by creating a *MATHEMATICA*® notebook. The notebook renders the templates as points in a three dimensional lattice. The plot is animated so the user can see the template bank from different perspectives. See figure 1.1.

#### Notes

- The output of this function is “Math3DNotebook.nb” and will appear in the directory of the program that called this function.
- Exported *Mathematica*® graphics will appear in your home directory for unix users and in the \Mathematica directory for Windows users unless you have another path configured in your *Mathematica*® installation. It is necessary to change the file name within the notebook to avoid overwriting previous files.
32.15.2 Module LALMathNDPlot.c

Prototypes

```c
void LALMathNDPlot( LALStatus *stat,
                       MathNDPointList *first,
                       INT4 *ntiles,
                       REAL4 *pointSize)
```

*stat LALStatus structure pointer

*first MathNDPointList structure pointer

*ntiles INT4 pointer to the number of templates you plan to plot. This may be called as NULL. If it is called with a value this function will check to see if the MathNDPointList has the correct number of templates. If it does not a warning will be printed.

pointSize REAL4 ∈ [0,1] which specifies the relative size of each point to the final display area. (e.g. 1 would fill the entire plot.) This may be called as NULL and a calculated value will be assigned. (It's only a rough guess)

Description

This module contains a function for plotting N-Dimensional template banks by creating a Mathematica® notebook. The notebook renders the templates as points in all of the 3-Dimensional projection permutations. Each projection may be animated so the user can see the template bank from different perspectives.

Notes

- The output of this function is “MathNDNotebook.nb” and will appear in the directory of the program that called this function.

- Exported Mathematica® graphics will appear in your home directory for unix users and in the \Mathematica directory for Windows users unless you have another path configured in your Mathematica® installation. It is necessary to change the file name within the notebook to avoid overwriting previous files.

- The number of projections is N!/(3!(N-3)!). Thus plotting 6 dimensions would produce 20 projections, 7 dimensions would yield 35 and 8 gives 56.
32.15.3 Program LALMath3DPlotTest.c

Tests LALMath3DPlot().

LALMath3DPlotTest

Description

This program generates a set of points simulating a template bank and calls LALMath3DPlot() to generate a \textit{Mathematica}\textsuperscript{\copyright} notebook to display a 3D image of the bank. Instructions on how to evaluate the notebook appear when it is opened.

Exit codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NORM</td>
<td>0</td>
<td>&quot;Normal exit&quot;</td>
</tr>
<tr>
<td>MEM</td>
<td>1</td>
<td>&quot;Memory allocation error&quot;</td>
</tr>
<tr>
<td>SUB</td>
<td>2</td>
<td>&quot;Subroutine error&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants LALMATH3DPLOTTESTC_E<name>, and the status descriptions in LALMATH3DPLOTTESTC_MSGE<name>. The source code with these messages is in LALMath3DPlotTest.c on line 1.108.

Notes

- For a more interesting test of LALMath3DPlot() see InspiralSpinBankTest.c in the bank package.
32.15.4 Program LALMathNDPlotTest.c

Tests LALMathNDPlot().

LALMathNDPlotTest

Description

This program generates a set of points simulating a 4-D template bank and calls LALMathNDPlot() to generate a Mathematica® notebook to display the permutations of 3D projections of the template bank. Instructions on how to evaluate the notebook appear when it is opened.

Exit codes

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NORM</td>
<td>0</td>
<td>&quot;Normal exit&quot;</td>
</tr>
<tr>
<td>MEM</td>
<td>1</td>
<td>&quot;Memory allocation error&quot;</td>
</tr>
<tr>
<td>SUB</td>
<td>2</td>
<td>&quot;Subroutine error&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants LALMATHNDPLOTTESTC_E<name>, and the status descriptions in LALMATHNDPLOTTESTC_MSGE<name>. The source code with these messages is in LALMathNDPlotTest.c on line l.106.

Notes

- No notes yet.
Section 12

LAL Framedata Interface (optional)
Chapter 33

Package **framedata**

Package for reading frame-format data files.
33.1 Header FrameCache.h

Routines for manipulating a cache of available frame files.

Synopsis

```
#include <lal/FrameCache.h>
```

Frame file catalogues contain URLs for available frame files along with various pieces of metadata. These routines allow frame catalogues to be imported, exported, or generated, and stored in a frame cache structure, which can be manipulated.

Error conditions

<table>
<thead>
<tr>
<th>name</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>1</td>
<td>&quot;Null pointer&quot;</td>
</tr>
<tr>
<td>NNUL</td>
<td>2</td>
<td>&quot;Non-null pointer&quot;</td>
</tr>
<tr>
<td>ALOC</td>
<td>4</td>
<td>&quot;Memory allocation error&quot;</td>
</tr>
<tr>
<td>IEIO</td>
<td>8</td>
<td>&quot;Import/export I/O error&quot;</td>
</tr>
<tr>
<td>LINE</td>
<td>16</td>
<td>&quot;Input line too long&quot;</td>
</tr>
<tr>
<td>PATH</td>
<td>32</td>
<td>&quot;Unable to glob frame files to build cache&quot;</td>
</tr>
<tr>
<td>NFRM</td>
<td>64</td>
<td>&quot;No frame files&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants FRAMECACHEH_E<name>, and the status descriptions in FRAMECACHEH_MSGE<name>. The source code with these messages is in FrameCache.h on line 1.76.

Structures

```c
typedef struct
tagFrStat
{
    CHAR *source;
    CHAR *description;
    INT4  startTime;
    INT4  duration;
    CHAR *url;
} FrStat;
```

This structure contains a frame file status. The fields are:

- **source**: the source detector(s) of the data in the frame file, or other identifier.
- **description**: the description of the type of data contained in the frame file, or other identifier.
- **startTime**: the GPS time of the second equal to (or just before) the start of the data contained in the frame file.
- **duration**: the number of seconds between **startTime** and the GPS time of the second equal to (or just after) the end of the data contained in the frame file.
- **url**: the URL of the frame file.

```c
typedef struct
tagFrCache
{
    UINT4 numFrameFiles;
    FrStat *frameFiles;
} FrCache;
```

```c
```
This structure contains a list of all frame files available. The fields are:

- **numFrameFiles**: the total number of frame files in the list.
- **frameFiles**: array of frame file status descriptors.

```c
typedef struct
tagFrCacheSieve
{
    const CHAR *srcRegEx;
    const CHAR *dscRegEx;
    const CHAR *urlRegEx;
    INT4 earliestTime;
    INT4 latestTime;
} FrCacheSieve;
```

This structure contains parameters to use to extract those frame files of interest from a cache. The parameters include regular expressions and time ranges. The fields are:

- **srcRegEx**: regular expression to use in selecting frame files with a specified source identifier. (Not used if NULL.)
- **dscRegEx**: regular expression to use in selecting frame files with a specified description identifier. (Not used if NULL.)
- **urlRegEx**: regular expression to use in selecting frame files with a specified URL. (Not used if NULL.)
- **earliestTime**: earliest time (GPS seconds) of frame files of interest. (Not used if zero or less.)
- **latestTime**: latest time (GPS seconds) of frame files of interest. (Not used if zero or less.)
33.1.1 Module FrameCache.c

Routines for importing, exporting, generating, and manipulating frame catalogs and cache structures.

Prototypes

```c
void LALFrCacheImport(
    LALStatus *status,
    FrCache **output,
    const CHAR *fname
)

void LALFrCacheExport(
    LALStatus *status,
    FrCache *cache,
    const CHAR *fname
)

void LALDestroyFrCache(
    LALStatus *status,
    FrCache **cache
)

void LALFrCacheSieve(
    LALStatus *status,
    FrCache **output,
    FrCache *input,
    FrCacheSieve *params
)

void LALFrCacheGenerate(
    LALStatus *status,
    FrCache **output,
    const CHAR *dirstr,
    const CHAR *fnptrn
)
```

Description

A frame catalogue file has several pieces of metadata, including:

- **source** the source identifier, often a combination of upper-case letters representing the detector sites (e.g., ‘H’ for Hanford, ‘L’ for Livingston) from which the frame data was generated.

- **description** the description identifier, often a single upper-case letter describing what kind of frame data is present (e.g., ‘R’ for raw data, ‘M’ for minute-trend data).

- **GPS-time** the GPS time in seconds (rounded down) of the start of the data contained in the frame file.

- **duration** the difference between the GPS time in seconds (rounded up) of the end of the data contained in the frame file and the GPS time in seconds (rounded down) of the start of the data contained in the frame file.

- **URL** the URL of the frame data. If the protocol (i.e., `file://` or `http://`) is missing then it is assumed that this is the absolute or relative path of a frame file. File URLs must have a fully-qualified domain name or the word `localhost` if on the localhost (or else localhost is assumed if absent). Examples:

```plaintext
// file://example/frame_data/030201_H_R.raw
// http://example/frame_data/030201_L_M.raw
```
Other types of metadata, such as the md5 checksum and the file size, are not used by these LAL routines. The format of the catalogues is

```
source description GPS-time duration URL (ignored additional fields)
```

for example:

```
F TEST 600000000 60 file://localhost/usr/share/lal/F-TEST-600000000-60.gwf
F TEST 600000060 60 file://localhost/usr/share/lal/F-TEST-600000060-60.gwf
F TEST 600000120 60 file://localhost/usr/share/lal/F-TEST-600000120-60.gwf
```

If any data is missing or unknown, it is represented in the catalogue by a single hyphen (−).

The routine `LALFrCacheImport` reads in a specified frame catalogue file and creates a frame cache. The routine `LALFrCacheExport` exports a frame cache as a frame catalogue file. The routine `LALFrCacheGenerate` scans a colon-delimited list of directory paths (or . if NULL) for files that match a given glob pattern (default is *.gwf if NULL), and uses these to generate a frame cache (with the metadata extracted from the file names). The routine `LALFrCacheSieve` applies regular expression filters to various pieces of metadata to distill a frame cache into a sorted sub-cache of frame files of interest. The routine `LALDestroyFrCache` destroys a frame cache.
33.2 Header FrameStream.h

Low-level routines for manipulating frame data streams.

Synopsis

```c
#include <stdio.h>
#include <lal/FrameStream.h>
```

A frame stream is like a file stream except that it streams along the set of frames in a set of frame files. These routines are low-level routines that allow you to extract frame data. Many of these routines have names similar to the standard C file stream manipulation routines and perform similar functions.

Error conditions

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>00001</td>
<td>&quot;Null pointer&quot;</td>
</tr>
<tr>
<td>NNUL</td>
<td>00002</td>
<td>&quot;Non-null pointer&quot;</td>
</tr>
<tr>
<td>ALOC</td>
<td>00004</td>
<td>&quot;Memory allocation error&quot;</td>
</tr>
<tr>
<td>FILE</td>
<td>00010</td>
<td>&quot;Frame data files not found&quot;</td>
</tr>
<tr>
<td>OPEN</td>
<td>00020</td>
<td>&quot;Frame file open error&quot;</td>
</tr>
<tr>
<td>READ</td>
<td>00040</td>
<td>&quot;Frame file read error&quot;</td>
</tr>
<tr>
<td>TIME</td>
<td>00100</td>
<td>&quot;Invalid ADC offset time&quot;</td>
</tr>
<tr>
<td>SIZE</td>
<td>00200</td>
<td>&quot;Invalid vector length&quot;</td>
</tr>
<tr>
<td>CHAN</td>
<td>00400</td>
<td>&quot;Could not find ADC channel&quot;</td>
</tr>
<tr>
<td>TYPE</td>
<td>01000</td>
<td>&quot;Invalid ADC type&quot;</td>
</tr>
<tr>
<td>RROR</td>
<td>02000</td>
<td>&quot;Frame stream error&quot;</td>
</tr>
<tr>
<td>DONE</td>
<td>04000</td>
<td>&quot;End of frame data&quot;</td>
</tr>
<tr>
<td>TREQ</td>
<td>010000</td>
<td>&quot;No data at time requested&quot;</td>
</tr>
<tr>
<td>DGAP</td>
<td>020000</td>
<td>&quot;Gap in the data&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants `FRAMESTREAMH_<name>`, and the status descriptions in `FRAMESTREAMH_MSGE_<name>`. The source code with these messages is in `FrameStream.h` on line 1.93.

Structures

```c
typedef enum {
    LAL_FR_OK = 0, /* nominal */
    LAL_FR_ERR = 1, /* error in frame stream */
    LAL_FR_END = 2, /* end of frame stream */
    LAL_FR_GAP = 4, /* gap in frame stream */
    LAL_FR_URL = 8, /* error opening frame URL */
    LAL_FR_TOC = 16 /* error reading frame TOC */
} FrState;

typedef enum {
    LAL_FR_SILENT_MODE = 0,
    LAL_FR_TIMEWARN_MODE = 1, /* display warning for invalid time requests */
    LAL_FR_GAPINFO_MODE = 2, /* display info for gaps in data */
    LAL_FR_VERBOSE_MODE = 3, /* display warnings and info */
    LAL_FR_IGNOREGAP_MODE = 4, /* ignore gaps in data */
    LAL_FR_IGNORETIME_MODE = 8, /* ignore invalid times requested */
    LAL_FR_DEFAULT_MODE = 15, /* ignore time/gaps but report warnings & info */
} FrMode;
```
33.2. Header FrameStream.h

```c
LAL_FR_CHECKSUM_MODE = 16 /* ensure that file checksums are OK */
```

```c
FrMode;
struct FrFile;
typedef struct tagFrFileInfo
{
    INT4 ind;
    CHAR *url;
    INT4 t0;
    INT4 dt;
}
FrFileInfo;
typedef struct tagFrStream
{
    FrState state;
    INT4 mode;
    LIGOTimeGPS epoch;
    UINT4 nfile;
    FrFileInfo *flist;
    UINT4 fnum;
    struct FrFile *file;
    INT4 pos;
}
FrStream;
```

This structure details the state of the frame stream. The contents are private; you should not tamper with them!

```c
typedef struct
tagFrPos
{
    LIGOTimeGPS epoch;
    UINT4 fnum;
    INT4 pos;
}
FrPos;
```

This structure contains a record of the state of a frame stream; this record can be used to restore the stream to the state when the record was made (provided the stream has not been closed). The fields are:

- `epoch` the GPS time of the open frame when the record was made.
- `fnum` the file number of a list of frame files that was open when the record was made.
- `pos` the position within the frame file that was open when the record was made.

```c
typedef enum
{ LAL_ADC_CHAN, LAL_SIM_CHAN, LAL_PROC_CHAN }
FrChanType;
/* for backwards compatability... */
#define ChannelType FrChanType
#define ProcDataChannel LAL_PROC_CHAN
#define ADCDataChannel LAL_ADC_CHAN
#define SimDataChannel LAL_SIM_CHAN
```

These are the various types of channel that can be specified for read/write. They are “post-processed data” (ProcDataChannel), “ADC data” (ADCDataChannel), and “simulated data” (SimDataChannel).

```c
typedef struct
tagFrChanIn
{
    const CHAR *name;
    ChannelType type;
```
This structure specifies the channel to read as input. The fields are:

- **name**: the name of the channel.
- **type**: the channel type.

```c
typedef struct
tagFrOutPar
{
   const CHAR *source;
   const CHAR *description;
   ChannelType type;
   UINT4 nframes;
   UINT4 frame;
   UINT4 run;
} FrOutPar;
```

This structure specifies the parameters for output of data to a frame. The fields are:

- **source**: the source identifier to attach to the output frame file name.
- **description**: the description identifier to attach to the output frame file name.
- **type**: the type of channel to create in the output frames.
- **nframes**: the number of frames to output in the frame file.
- **frame**: the number the first frame of output.
- **run**: the number this data run.

The output frame file name will be `⟨source⟩-⟨description⟩-⟨GPS start time⟩-⟨duration⟩.gwf`.
33.2.1 Module FrameStream.c

These are the low-level functions for manipulating a frame stream.

Prototypes

```c
void LALFrCacheOpen(
    LALStatus *status,
    FrStream **output,
    FrCache *cache
)

void LALFrOpen(
    LALStatus *status,
    FrStream **stream,
    const CHAR *dirname,
    const CHAR *pattern
)

void LALFrClose(
    LALStatus *status,
    FrStream **stream
)

void LALFrSetMode(
    LALStatus *status,
    INT4 mode,
    FrStream *stream
)

void LALFrEnd(
    LALStatus *status,
    INT4 *end,
    FrStream *stream
)

void LALFrRewind(
    LALStatus *status,
    FrStream *stream
)

void LALFrNext(
    LALStatus *status,
    FrStream *stream
)

void LALFrSeek(
    LALStatus *status,
    const LIGOTimeGPS *epoch,
    FrStream *stream
)
```
void
LALFrTell(
    LALStatus *status,
    LIGOTimeGPS *epoch,
    FrStream *stream
)

void
LALFrGetPos(
    LALStatus *status,
    FrPos *position,
    FrStream *stream
)

void
LALFrSetPos(
    LALStatus *status,
    FrPos *position,
    FrStream *stream
)

Description

Many of these routines perform functions that are similar to standard C file stream manipulation routines. The names have been chosen to be also similar to the standard C routines.

The routines LALFrOpen() and LALFrClose() are used to open and close a frame stream. The stream is created by LALFrOpen(), and must be a pointer to NULL before it is opened. It must have been created prior to calling LALFrClose(), and after this call, the stream will be a pointer to NULL. The routine LALFrOpen() requires the user to specify the directory name of the frame files and the head names. If the directory is NULL, the routine uses the current directory (.). The head names specifies which files are the wanted files in the specified directory. Wildcards are allowed. For example, to get LLO frames only, the head names could be set to L-*.gwf. If the head name is NULL, the default value *.gwf is used. The routine LALFrCacheOpen() is like LALFrOpen() except that the list of frame files is taken from a frame file cache. [In fact, LALFrOpen() simply uses LALFrCacheGenerate() and LALFrCacheOpen() to create the stream.]

The routine LALFrSetMode() is used to change the operating mode of a frame stream, which determines how the routines try to accomodate gaps in data and requests for times when there is no data (e.g., before the beginning of the data, after the end of the data, or in some missing data). The default mode, which is given the value LAL_FR_DEFAULT_MODE, prints warnings if a time requested corresponds to a time when there is no data (but then skips to the first available data) and prints an info message when a gap in the data occurs (but then skips beyond the gap). This default mode is equal to the combination LAL_FR_VERBOSE_MODE | LAL_FR_IGNOREGAP_MODE | LAL_FR_IGNORETIME_MODE where LAL_FR_VERBOSE_MODE is equal to the combination LAL_FR_TIMEWARN_MODE | LAL_FR_GAPINFO_MODE. Use LAL_FR_VERBOSE_MODE to print out warnings when requesting times with no data and print out an info message when a gap in the data is encountered. Unless the mode is supplemented with LAL_FR_IGNOREGAP_MODE, gaps encountered in the data will cause a routine to exit with a non-zero status code; similarly, LAL_FR_IGNORETIME_MODE prevents routines from failing if a time when there is not data is requested. Set the mode to LAL_FR_SILENT_MODE to suppress the warning and info messages but still cause routines to fail when data is not available. Note: the default value LAL_FR_DEFAULT_MODE is assumed initially, but this is not necessarily the recommended mode — it is adopted for compatibility reasons.

The routine LALFrEnd() determines if the end-of-frame-data flag for the data stream has been set.

The routine LALFrNext() advances the frame stream to the beginning of the next frame.

The routine LALFrRewind() rewinds the frame stream to the first frame.

The routine LALFrSeek() sets the frame stream to a specified time, or the earliest time after the specified time if that time is not available (e.g., if it is before the beginning of the frame stream or if it is in a gap in the frame data). The routine LALFrTell() returns the current time within the frame stream.

The routine LALFrGetPos() returns a structure containing the current frame stream position. The frame stream can later be restored to this position using LALFrSetPos().
33.2.2 Module FrameSeries.c

These routines are used to read/write time series data from/to frame files.

Prototypes

```c
void LALFrGetTimeSeriesType(  
    LALStatus *status,  
    LALTYPECODE *output,  
    FrChanIn *chanin,  
    FrStream *stream  
)
```

```c
int XLALFrGetCOMPLEX16FrequencySeries ( COMPLEX16FrequencySeries *series, FrStream *stream )
```

```c
void LALFrGetCOMPLEX16FrequencySeries (  
    LALStatus *status,  
    COMPLEX16FrequencySeries *series,  
    FrChanIn *chanin,  
    FrStream *stream  
)
```

```c
void LALFrGetCOMPLEX16TimeSeries (  
    LALStatus *status,  
    COMPLEX16TimeSeries *series,  
    FrChanIn *chanin,  
    FrStream *stream  
)
```

```c
void LALFrGetCOMPLEX16TimeSeriesMetadata (  
    LALStatus *status,  
    COMPLEX16TimeSeries *series,  
    FrChanIn *chanin,  
    FrStream *stream  
)
```

```c
void LALFrWriteCOMPLEX16TimeSeries (  
    LALStatus *status,  
    COMPLEX16TimeSeries *series,  
    FrOutPar *params  
)
```

```c
void LALFrWriteCOMPLEX16FrequencySeries (  
    LALStatus *status,  
    COMPLEX16FrequencySeries *series,  
    FrOutPar *params,  
    INT4 subtype  
)
```

```c
int XLALFrGetCOMPLEX8FrequencySeries ( COMPLEX8FrequencySeries *series, FrStream *stream )
```

```c
void LALFrGetCOMPLEX8FrequencySeries (  
    LALStatus *status,  
    COMPLEX8FrequencySeries *series,  
    FrChanIn *chanin,  
    FrStream *stream  
)
```
void LALFrGetCOMPLEX8TimeSeries (  
    LALStatus  *status,  
    COMPLEX8TimeSeries  *series,  
    FrChanIn  *chanin,  
    FrStream  *stream  
  )

void LALFrGetCOMPLEX8TimeSeriesMetadata (  
    LALStatus  *status,  
    COMPLEX8TimeSeries  *series,  
    FrChanIn  *chanin,  
    FrStream  *stream  
  )

void LALFrWriteCOMPLEX8TimeSeries (  
    LALStatus  *status,  
    COMPLEX8TimeSeries  *series,  
    FrOutPar  *params  
  )

void LALFrWriteCOMPLEX8FrequencySeries (  
    LALStatus  *status,  
    COMPLEX8FrequencySeries  *series,  
    FrOutPar  *params,  
    INT4  subtype  
  )

int XLALFrGetREAL8FrequencySeries ( REAL8FrequencySeries *series, FrStream *stream )

void LALFrGetREAL8FrequencySeries (  
    LALStatus  *status,  
    REAL8FrequencySeries  *series,  
    FrChanIn  *chanin,  
    FrStream  *stream  
  )

void LALFrGetREAL8TimeSeries (  
    LALStatus  *status,  
    REAL8TimeSeries  *series,  
    FrChanIn  *chanin,  
    FrStream  *stream  
  )

void LALFrGetREAL8TimeSeriesMetadata (  
    LALStatus  *status,  
    REAL8TimeSeries  *series,  
    FrChanIn  *chanin,  
    FrStream  *stream  
  )

void LALFrWriteREAL8TimeSeries (  
    LALStatus  *status,  
    REAL8TimeSeries  *series,  
    FrOutPar  *params  
  )
void LALFrWriteREAL8FrequencySeries (LALStatus *status, REAL8FrequencySeries *series, FrOutPar *params, INT4 subtype)

int XLALFrGetREAL4FrequencySeries (REAL4FrequencySeries *series, FrStream *stream)

void LALFrGetREAL4FrequencySeries (LALStatus *status, REAL4FrequencySeries *series, FrChanIn *chanin, FrStream *stream)

void LALFrGetREAL4TimeSeries (LALStatus *status, REAL4TimeSeries *series, FrChanIn *chanin, FrStream *stream)

void LALFrGetREAL4TimeSeriesMetadata (LALStatus *status, REAL4TimeSeries *series, FrChanIn *chanin, FrStream *stream)

void LALFrWriteREAL4TimeSeries (LALStatus *status, REAL4TimeSeries *series, FrOutPar *params)

void LALFrWriteREAL4FrequencySeries (LALStatus *status, REAL4FrequencySeries *series, FrOutPar *params, INT4 subtype)

int XLALFrGetINT8FrequencySeries (INT8FrequencySeries *series, FrStream *stream)

void LALFrGetINT8FrequencySeries (LALStatus *status, INT8FrequencySeries *series, FrChanIn *chanin, FrStream *stream)
void LALFrGetINT8TimeSeries (  
    LALStatus        *status,  
    INT8TimeSeries   *series,  
    FrChanIn         *chanin,  
    FrStream         *stream  
)

void LALFrGetINT8TimeSeriesMetadata (  
    LALStatus        *status,  
    INT8TimeSeries   *series,  
    FrChanIn         *chanin,  
    FrStream         *stream  
)

void LALFrWriteINT8TimeSeries (  
    LALStatus        *status,  
    INT8TimeSeries   *series,  
    FrOutPar         *params  
)

void LALFrWriteINT8FrequencySeries (  
    LALStatus        *status,  
    INT8FrequencySeries   *series,  
    FrOutPar         *params,  
    INT4           subtype  
)

int XLALFrGetINT4FrequencySeries ( INT4FrequencySeries *series, FrStream *stream )

void LALFrGetINT4FrequencySeries (  
    LALStatus        *status,  
    INT4FrequencySeries   *series,  
    FrChanIn         *chanin,  
    FrStream         *stream  
)

void LALFrGetINT4TimeSeries (  
    LALStatus        *status,  
    INT4TimeSeries   *series,  
    FrChanIn         *chanin,  
    FrStream         *stream  
)

void LALFrGetINT4TimeSeriesMetadata (  
    LALStatus        *status,  
    INT4TimeSeries   *series,  
    FrChanIn         *chanin,  
    FrStream         *stream  
)

void LALFrWriteINT4TimeSeries (  
    LALStatus        *status,  
    INT4TimeSeries   *series,  
    FrOutPar         *params  
)
Description

The routines `LALFrGet/datatypeTimeSeries()` search the frame for a specified channel. If the time series supplied has data storage allocated, then the specified amount of data is filled from the frame stream. If no space has been allocated (so that the data field is `NULL`), then only the channel information is returned in the time series (e.g., the start time of the next data and the time step size).

Because it is good coding practice to not tinker directly with the innards of structures like time series, the behaviour described above is undesirable whereby the time series reading functions do useful things when part of the time series structures passed into them is not initialized. To address this, the routines `LALFrGet/datatypeTimeSeriesMetadata()` are provided. These routines accept a fully initialized time
series structure, and populate only the metadata from the frame stream. New code should be written to use these functions when only the time series meta data is desired.

The routine `LALFrGetTimeSeriesType` returns the type of the time series corresponding to the specified channel. This is needed if it is not known in advance what type of data is stored in the channel.

The routines `LALFrWrite(datatype)TimeSeries()` outputs a given time series as a new frame file with the filename `<source>-<description>-<GPS-seconds>-<duration>.gwf` where source and description are the specified frame source and description identifiers, GPS-seconds is the start time of the data in seconds since 0h UTC 6 Jan 1980 (GPS time origin), or the nearest second before the start of the data, and duration is the number of seconds between value of GPS-seconds and the GPS end time of the data, rounded up.
33.2.3 Program FrameStreamTest.c

Tests the low-level frame stream routines.

Usage

FrameStreamTest

Description

This program reads the channels H1:LSC-AS_Q from all the fake frames F-TEST-*.gwf in the directory set in the environment LAL_FRAME_PATH * (or the current directory if this environment is not set) and prints them to files.
### 33.3 Header FrameCalibration.h

High-level routines for extracting calibration data from frames

#### Synopsis

```c
#include <lal/FrameCalibration.h>
```

Provides a high level interface for building a transfer or response functions from raw calibration data provided by the calibration team.

#### Error conditions

These error conditions are generated by the function `ExtractFrameCalibration()` if it encounters an error. If codes 1 through 7 are returned, the no calibration has been generated since there was an error reading the reference calibration. If an error occurs once the reference calibration has been read in, then the reference calibration is returned without being updated. This allows the user the option to trap the error and fall back on the reference calibration or to give up completely. This can be done in the case of error code 8, or error code −1, which indicates an error in one of the functions used to update the reference calibration.

<table>
<thead>
<tr>
<th><code>&lt;name&gt;</code></th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>1</td>
<td>&quot;No calibration generated: Null pointer&quot;</td>
</tr>
<tr>
<td>NNUL</td>
<td>2</td>
<td>&quot;No calibration generated: Non-null pointer&quot;</td>
</tr>
<tr>
<td>MCHE</td>
<td>3</td>
<td>&quot;No calibration generated: unable to open calibration cache file&quot;</td>
</tr>
<tr>
<td>CREF</td>
<td>4</td>
<td>&quot;No calibration generated: no reference calibration in cache&quot;</td>
</tr>
<tr>
<td>OREF</td>
<td>5</td>
<td>&quot;No calibration generated: unable to open reference frame&quot;</td>
</tr>
<tr>
<td>DCHE</td>
<td>6</td>
<td>&quot;No calibration generated: error reference calibration cache&quot;</td>
</tr>
<tr>
<td>REFR</td>
<td>7</td>
<td>&quot;No calibration generated: error reading reference calibration&quot;</td>
</tr>
<tr>
<td>CFAC</td>
<td>8</td>
<td>&quot;Calibration not updated: no update factor frames in cache&quot;</td>
</tr>
<tr>
<td>DTMM</td>
<td>9</td>
<td>&quot;Calibration not updated: mismatch between sample rate of alpha and alpha*beta&quot;</td>
</tr>
<tr>
<td>METH</td>
<td>10</td>
<td>&quot;Calibration cache must either be read from a file or globbed&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants `FRAMECALIBRATIONH_E<name>`, and the status descriptions in `FRAMECALIBRATIONH_MSGE<name>`. The source code with these messages is in `FrameCalibration.h` on line 1.93.

#### Structures

* Author: Brown, D. A.
* $Id: FrameCalibration.h,v 1.10 2007/06/08 14:41:46 bema Exp $
33.3.1 Module FrameCalibration.c

This module contains code used to extract calibration information contained in frame files, and to construct a response (or transfer) function. This is supposed to provide a high-level interface for search authors to obtain a response function in the desired form.

Prototypes

```c
void LALExtractFrameResponse(
    LALStatus *status,
    COMPLEX8FrequencySeries *output,
    FrCache *calCache,
    CalibrationUpdateParams *calfacts
)
```

```c
void LALCreateCalibFrCache(
    LALStatus *status,
    FrCache **output,
    const CHAR *calCacheName,
    const CHAR *dirstr,
    const CHAR *calGlobPattern
)
```

The routine `LALFrameExtractResponse()` extracts the necessary calibration information from the frames. The frames used to construct the calibration are located using the specified LAL frame cache. The function constructs a response function (as a frequency series) from this information. The fourth argument is a pointer to a CalibrationUpdateParams structure. If the ifo field is non-NULL then this string specifies the detector (H1, H2, L1, etc.) for which the calibration is required. If the duration field is non-zero then the calibration will be averaged over the specified duration. If the duration is set to zero, then the first calibration at or before the start time is used. The alpha and alphabeta fields of the structure are required to be zero. Certain fields of the output should be set before this routine is called. In particular:

1. The epoch field of the frequency series should be set to the correct epoch so that the routine can generate a response function tailored to that time (accounting for calibration drifts).

2. The units of the response function should be set to be either strain-per-count (for a response function) or count-per-strain (for a transfer function); the routine will then return either the response function or its inverse depending on the specified units. Furthermore, the power-of-ten field of the units is examined to scale the response function accordingly.

3. The data vector should be allocated to the required length and the frequency step size should be set to the required value so that the routine can interpolate the response function to the required frequencies.

The format of the LAL frame cache must contain frames of the following types.

1. It must contain an entry for one frame of type `CALREF` which contains the response and cavity gain frequency series that will be up dated to the specified point in time. For example it must contain an entry such as

   ```
   L CALREF 715388533 64 file://localhost/path/to/L-CALREF-715388533-64.gwf
   ```

   where the frame file contains the channels `L1:CAL-RESPONSE` and `L1:CAL-CAV_GAIN`.

2. It must also contain entries for the frames needed to update the point calibration to the current time. These must contain the `L1:CAL-OLOOP_FAC` and `L1:CAL-CAV_FAC` channels. The update factor frames may either be SenseMon type frames, containing the factor channels as `real_8` trend data or frames generated by the lalapps program `lalapps_mkcalfac` which creates channels of type `complex_8`. The entries in the cache file must be of the format
for `lalapps_mkcalfac` type frames or

```
L SenseMonitor_L1_M 729925200 3600 file://localhost/path/to/L-SenseMonitor_L1_M-729925200-3600.gwf
```

for SenseMon type frames. If both types of frame are present in the cache, SenseMon frames are used in preference.
33.3.2 Program FrameCalibrationTest.c

Tests the high-level function to obtain an up-to-date calibration from frame files.

Usage

FrameCalibrationTest

Usage

FrameCalibrationTest [options]

Options:
- `h` print this message
- `o` write calibration to file
- `v` verbose: print extra information

Description

For each interferometer in the array \texttt{ifoCode} the program does the following: For each GPS time in the array \texttt{calTime}, an attempt is made to generate a calibration for that time. It reads a frame cache file named \texttt{ifo-CAL\_V03-gpstimerange\_catalog}, where \texttt{ifo} is the interferometer obtained from the array \texttt{ifoCode}, \texttt{gpstimerange} is obtained from the array \texttt{cacheTime}. Even elements of the array \texttt{calTime} use the first element of the array \texttt{cacheTime} and odd elements the second. This is done to allow the developer to test a range of GPS times and catalogs quickly.

The GPS times for which calibrations are attempted are 729331981, 729332039, 729332040, 729332041, 729332099 and 800000000. The catalog times are 729273600-734367600 and 729273600-734367600. The test uses that calibration data files

- \texttt{H-CAL\_FAC\_V03-729273600-5094000.gwf}
- \texttt{H-CAL\_REF\_V03-734367939-64.gwf}
- \texttt{L-CAL\_FAC\_V03-729273600-5094000.gwf}
- \texttt{L-CAL\_REF\_V03-731488397-64.gwf}

\texttt{FIXME:} the following test is not yet implemented: waiting for Steve’s change to add the return of the actual values used. The returned values of $\alpha$ and $\alpha\beta$ are checked against the values obtained from the files S2 version 3 calibration factor files \texttt{H1AlphaBeta.txt} and \texttt{L1AlphaBeta.txt} which can be found on the calibration home page.

Exit codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success, normal exit.</td>
</tr>
<tr>
<td>1</td>
<td>Subroutine failed.</td>
</tr>
<tr>
<td>77</td>
<td>Ignored failure: Test frame data not found.</td>
</tr>
</tbody>
</table>

Uses

Notes
Section 13

LAL MPI Interface (optional)
Chapter 34

Package **comm**

This package contains routines for using MPI to exchange LAL data structures.
34.1 Header Comm.h

Provides routines for MPI communication.

Synopsis

#include <lal/Comm.h>

This header covers the routines for doing MPI communication.

Error conditions

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>1</td>
<td>&quot;Null pointer&quot;</td>
</tr>
<tr>
<td>NNUL</td>
<td>2</td>
<td>&quot;Non-Null pointer&quot;</td>
</tr>
<tr>
<td>SIZE</td>
<td>4</td>
<td>&quot;Invalid size&quot;</td>
</tr>
<tr>
<td>SZMM</td>
<td>8</td>
<td>&quot;Exchange size mismatch&quot;</td>
</tr>
<tr>
<td>MPIE</td>
<td>16</td>
<td>&quot;MPI error&quot;</td>
</tr>
<tr>
<td>HAND</td>
<td>32</td>
<td>&quot;Wrong handshake&quot;</td>
</tr>
<tr>
<td>NOBJ</td>
<td>64</td>
<td>&quot;Invalid number of objects&quot;</td>
</tr>
</tbody>
</table>

The status codes in the table above are stored in the constants COMMH_E<name>, and the status descriptions in COMMH_MSGE<name>. The source code with these messages is in Comm.h on line 1.83.

Types

Structure MPIMessage

This structure is sent to a remote process, via LALMPISendMsg(), to alert that process that there is a message. Note that LALMPIRecvMsg() is the only Recv-type function that does not require the source to be identified; the receiver can then identify the source from the message received.

Essentially, LALMPISendMsg() and LALMPIRecvMsg() form a handshake for a subsequent transmission, and MPIMessage specifies the protocol. The local process uses LALMPISendMsg() to communicate with a remote process, which is waiting to hear from any process using LALMPIRecvMsg(). The message the local process sends specifies

1. An integer code telling the remote process what operation it should take (e.g., get ready to exchange some data, tell the remote process to terminate, etc.).
2. A boolean integer that is zero if the remote process is expected to send something to the local process, or non-zero if the local process will send something to the remote process.
3. An integer representing the MPI process number of the local process.

The fields are:

INT4 msg   An integer code specifying to the receiver what type of operation is to be taken.
INT4 send  A boolean that is non-zero if the originator of the message will be sending something to the recipient of the message in a subsequent communication, or zero if the recipient of the message is expected to send something to the originator of the message.
INT4 source The MPI process number of the originator of the message.

Structures ExchParams and InitExchParams

These structures are used in the Exch-type routines. The structure InitExchParams are the parameters used in initializing an exchange protocol using LALInitializeExchange. The fields are:

INT4 myProcNum  The MPI process number of the local process.
MPI_Comm mpiComm The MPI communicator.
The structure `ExchParams` is created by `LALInitializeExchange()`, destroyed by `LALFinalizeExchange()`, and serves as the parameter for the various `LALExchange(type)` functions. It is also required as the input to `LALInitializeExchange()` for the originator of the exchange request. The fields are:

**INT4 send** A code that indicates whether this process is sending (non-zero) or receiving (zero). The process that initializes the exchange chooses whether it will send in the subsequent exchanges (non-zero value for the `send` field of the input `ExchParams`), or receive (zero value).

**INT4 numObjects** The (maximum) number of objects to be exchanged. (The partners in the exchange may have some mechanism to decide to terminate the exchange early, e.g., by exchanging a negative integer.)

**INT4 partnerProcNum** The MPI process number of the partner in the exchange.

**INT4 myProcNum** The MPI process number of the local process.

**INT4 exchObjectType** An integer code representing the type of object that will be exchanged.

**MPI_Comm mpiComm** The MPI communicator.
34.1.1 Module SendRecv.c

Routines to perform basic MPI sending and receiving of LAL data types.

Prototypes

- `void LALMPISendMsg( LALStatus *status, MPIMessage *msg, INT4 dest, MPI_Comm comm )`
- `void LALMPIRecvMsg( LALStatus *status, MPIMessage *msg, MPI_Comm comm )`
- `void LALMPISendCOMPLEX16( LALStatus *status, COMPLEX16 *element, INT4 dest, MPI_Comm comm )`
- `void LALMPIRecvCOMPLEX16( LALStatus *status, COMPLEX16 *element, INT4 source, MPI_Comm comm )`
- `void LALMPISendCOMPLEX8( LALStatus *status, COMPLEX8 *element, INT4 dest, MPI_Comm comm )`
- `void LALMPIRecvCOMPLEX8( LALStatus *status, COMPLEX8 *element, INT4 source, MPI_Comm comm )`
- `void LALMPISendREAL8( LALStatus *status, REAL8 *element, INT4 dest, MPI_Comm comm )`
- `void LALMPIRecvREAL8( LALStatus *status, REAL8 *element, INT4 source, MPI_Comm comm )`
- `void LALMPISendREAL4( LALStatus *status, REAL4 *element, INT4 dest, MPI_Comm comm )`
- `void LALMPIRecvREAL4( LALStatus *status, REAL4 *element, INT4 source, MPI_Comm comm )`
- `void LALMPISendINT2( LALStatus *status, INT2 *element, INT4 dest, MPI_Comm comm )`
- `void LALMPIRecvINT2( LALStatus *status, INT2 *element, INT4 source, MPI_Comm comm )`
- `void LALMPISendINT4( LALStatus *status, INT4 *element, INT4 dest, MPI_Comm comm )`
- `void LALMPIRecvINT4( LALStatus *status, INT4 *element, INT4 source, MPI_Comm comm )`
- `void LALMPISendINT8( LALStatus *status, INT8 *element, INT4 dest, MPI_Comm comm )`
- `void LALMPIRecvINT8( LALStatus *status, INT8 *element, INT4 source, MPI_Comm comm )`
- `void LALMPISendUINT2( LALStatus *status, UINT2 *element, INT4 dest, MPI_Comm comm )`
- `void LALMPIRecvUINT2( LALStatus *status, UINT2 *element, INT4 source, MPI_Comm comm )`
- `void LALMPISendUINT4( LALStatus *status, UINT4 *element, INT4 dest, MPI_Comm comm )`
- `void LALMPIRecvUINT4( LALStatus *status, UINT4 *element, INT4 source, MPI_Comm comm )`
- `void LALMPISendUINT8( LALStatus *status, UINT8 *element, INT4 dest, MPI_Comm comm )`
- `void LALMPIRecvUINT8( LALStatus *status, UINT8 *element, INT4 source, MPI_Comm comm )`
- `void LALMPISendCHAR( LALStatus *status, CHAR *element, INT4 dest, MPI_Comm comm )`
- `void LALMPIRecvCHAR( LALStatus *status, CHAR *element, INT4 source, MPI_Comm comm )`
- `void LALMPISendCOMPLEX16Vector( LALStatus *status, COMPLEX16Vector *vector, INT4 dest, MPI_Comm comm )`
- `void LALMPIRecvCOMPLEX16Vector( LALStatus *status, COMPLEX16Vector *vector, INT4 source, MPI_Comm comm )`
- `void LALMPISendCOMPLEX8Vector( LALStatus *status, COMPLEX8Vector *vector, INT4 dest, MPI_Comm comm )`
- `void LALMPIRecvCOMPLEX8Vector( LALStatus *status, COMPLEX8Vector *vector, INT4 source, MPI_Comm comm )`
- `void LALMPISendREAL8Vector( LALStatus *status, REAL8Vector *vector, INT4 dest, MPI_Comm comm )`
- `void LALMPIRecvREAL8Vector( LALStatus *status, REAL8Vector *vector, INT4 source, MPI_Comm comm )`
void LALMPISendREAL4Vector ( LALStatus *status, REAL4Vector *vector, INT4 dest, MPI_Comm comm )
void LALMPIRecvREAL4Vector ( LALStatus *status, REAL4Vector *vector, INT4 source, MPI_Comm comm )
void LALMPISendINT2Vector ( LALStatus *status, INT2Vector *vector, INT4 dest, MPI_Comm comm )
void LALMPIRecvINT2Vector ( LALStatus *status, INT2Vector *vector, INT4 source, MPI_Comm comm )
void LALMPISendINT4Vector ( LALStatus *status, INT4Vector *vector, INT4 dest, MPI_Comm comm )
void LALMPIRecvINT4Vector ( LALStatus *status, INT4Vector *vector, INT4 source, MPI_Comm comm )
void LALMPISendINT8Vector ( LALStatus *status, INT8Vector *vector, INT4 dest, MPI_Comm comm )
void LALMPIRecvINT8Vector ( LALStatus *status, INT8Vector *vector, INT4 source, MPI_Comm comm )
void LALMPISendUINT2Vector ( LALStatus *status, UINT2Vector *vector, INT4 dest, MPI_Comm comm )
void LALMPIRecvUINT2Vector ( LALStatus *status, UINT2Vector *vector, INT4 source, MPI_Comm comm )
void LALMPISendUINT4Vector ( LALStatus *status, UINT4Vector *vector, INT4 dest, MPI_Comm comm )
void LALMPIRecvUINT4Vector ( LALStatus *status, UINT4Vector *vector, INT4 source, MPI_Comm comm )
void LALMPISendUINT8Vector ( LALStatus *status, UINT8Vector *vector, INT4 dest, MPI_Comm comm )
void LALMPIRecvUINT8Vector ( LALStatus *status, UINT8Vector *vector, INT4 source, MPI_Comm comm )
void LALMPISendCHARVector ( LALStatus *status, CHARVector *vector, INT4 dest, MPI_Comm comm )
void LALMPIRecvCHARVector ( LALStatus *status, CHARVector *vector, INT4 source, MPI_Comm comm )
void LALMPISendCOMPLEX16TimeSeries ( LALStatus *status, COMPLEX16TimeSeries *series, INT4 dest, MPI_Comm comm )
void LALMPIRecvCOMPLEX16TimeSeries ( LALStatus *status, COMPLEX16TimeSeries *series, INT4 source, MPI_Comm comm )
void LALMPISendCOMPLEX8TimeSeries ( LALStatus *status, COMPLEX8TimeSeries *series, INT4 dest, MPI_Comm comm )
void LALMPIRecvCOMPLEX8TimeSeries ( LALStatus *status, COMPLEX8TimeSeries *series, INT4 source, MPI_Comm comm )
void LALMPISendREAL8TimeSeries ( LALStatus *status, REAL8TimeSeries *series, INT4 dest, MPI_Comm comm )
void LALMPIRecvREAL8TimeSeries ( LALStatus *status, REAL8TimeSeries *series, INT4 source, MPI_Comm comm )
void LALMPISendREAL4TimeSeries ( LALStatus *status, REAL4TimeSeries *series, INT4 dest, MPI_Comm comm )
void LALMPIRecvREAL4TimeSeries ( LALStatus *status, REAL4TimeSeries *series, INT4 source, MPI_Comm comm )
void LALMPISendINT2TimeSeries ( LALStatus *status, INT2TimeSeries *series, INT4 dest, MPI_Comm comm )
void LALMPIRecvINT2TimeSeries ( LALStatus *status, INT2TimeSeries *series, INT4 source, MPI_Comm comm )
void LALMPISendINT4TimeSeries ( LALStatus *status, INT4TimeSeries *series, INT4 dest, MPI_Comm comm )
void LALMPIRecvINT4TimeSeries ( LALStatus *status, INT4TimeSeries *series, INT4 source, MPI_Comm comm )
void LALMPISendINT8TimeSeries ( LALStatus *status, INT8TimeSeries *series, INT4 dest, MPI_Comm comm )
void LALMPIRecvINT8TimeSeries ( LALStatus *status, INT8TimeSeries *series, INT4 source, MPI_Comm comm )
void LALMPISendUINT2TimeSeries ( LALStatus *status, UINT2TimeSeries *series, INT4 dest, MPI_Comm comm )
void LALMPIRecvUINT2TimeSeries ( LALStatus *status, UINT2TimeSeries *series, INT4 source, MPI_Comm comm )
void LALMPISendUINT4TimeSeries ( LALStatus *status, UINT4TimeSeries *series, INT4 dest, MPI_Comm comm )
void LALMPIRecvUINT4TimeSeries ( LALStatus *status, UINT4TimeSeries *series, INT4 source, MPI_Comm comm )
void LALMPIRecvUINT4TimeSeries ( LALStatus *status, UINT4TimeSeries *series, INT4 source, MPI_Comm comm )
void LALMPISendUINT8TimeSeries ( LALStatus *status, UINT8TimeSeries *series, INT4 dest, MPI_Comm comm )
void LALMPIRecvUINT8TimeSeries ( LALStatus *status, UINT8TimeSeries *series, INT4 source, MPI_Comm comm )
void LALMPISendCOMPLEX16FrequencySeries ( LALStatus *status, COMPLEX16FrequencySeries *series, INT4 dest, MPI_Comm comm )
void LALMPIRecvCOMPLEX16FrequencySeries ( LALStatus *status, COMPLEX16FrequencySeries *series, INT4 source, MPI_Comm comm )
void LALMPISendCOMPLEX8FrequencySeries ( LALStatus *status, COMPLEX8FrequencySeries *series, INT4 dest, MPI_Comm comm )
void LALMPIRecvCOMPLEX8FrequencySeries ( LALStatus *status, COMPLEX8FrequencySeries *series, INT4 source, MPI_Comm comm )
void LALMPISendREAL8FrequencySeries ( LALStatus *status, REAL8FrequencySeries *series, INT4 dest, MPI_Comm comm )
void LALMPIRecvREAL8FrequencySeries ( LALStatus *status, REAL8FrequencySeries *series, INT4 source, MPI_Comm comm )
void LALMPISendREAL4FrequencySeries ( LALStatus *status, REAL4FrequencySeries *series, INT4 dest, MPI_Comm comm )
void LALMPIRecvREAL4FrequencySeries ( LALStatus *status, REAL4FrequencySeries *series, INT4 source, MPI_Comm comm )
void LALMPISendINT2FrequencySeries ( LALStatus *status, INT2FrequencySeries *series, INT4 dest, MPI_Comm comm )
void LALMPIRecvINT2FrequencySeries ( LALStatus *status, INT2FrequencySeries *series, INT4 source, MPI_Comm comm )
void LALMPISendINT4FrequencySeries ( LALStatus *status, INT4FrequencySeries *series, INT4 dest, MPI_Comm comm )
void LALMPIRecvINT4FrequencySeries ( LALStatus *status, INT4FrequencySeries *series, INT4 source, MPI_Comm comm )
void LALMPISendINT8FrequencySeries ( LALStatus *status, INT8FrequencySeries *series, INT4 dest, MPI_Comm comm )
void LALMPIRecvINT8FrequencySeries ( LALStatus *status, INT8FrequencySeries *series, INT4 source, MPI_Comm comm )
void LALMPISendUINT2FrequencySeries ( LALStatus *status, UINT2FrequencySeries *series, INT4 dest, MPI_Comm comm )
void LALMPIRecvUINT2FrequencySeries ( LALStatus *status, UINT2FrequencySeries *series, INT4 source, MPI_Comm comm )
void LALMPISendUINT4FrequencySeries ( LALStatus *status, UINT4FrequencySeries *series, INT4 dest, MPI_Comm comm )
void LALMPIRecvUINT4FrequencySeries ( LALStatus *status, UINT4FrequencySeries *series, INT4 source, MPI_Comm comm )
void LALMPISendUINT8FrequencySeries ( LALStatus *status, UINT8FrequencySeries *series, INT4 dest, MPI_Comm comm )
void LALMPIRecvUINT8FrequencySeries ( LALStatus *status, UINT8FrequencySeries *series, INT4 source, MPI_Comm comm )
Description

The routines \texttt{LALMPISend\langle type\rangle} and \texttt{LALMPIRecv\langle type\rangle} must be used in pairs: the sender must specify the MPI process number of the receiving process as the input variable \texttt{dest}, while the receiving process must specify the MPI process number of the sending process as its input variable \texttt{source}. The object that is sent/received is the output for both classes of functions. The parameter is the MPI communicator \texttt{comm}.

The exception is the function \texttt{LALMPIRecvMsg()}. This function does not specify a source—it receives from any process. The function \texttt{LALMPISendMsg()} is used to inform the receiver, which should be listening with \texttt{LALMPIRecvMsg()}, that the sender wishes to exchange information.

The following is a simple example of a commissar-comrad situation.

```c
#include <stdio.h>
#include <math.h>
#include <mpi.h>
#include <lal/LALStdlib.h>
#include <lal/Comm.h>

INT4 lalDebugLevel = 0;

int Commissar( INT4 size, MPI_Comm comm );
int Comrad( INT4 size, INT4 rank, MPI_Comm comm );

int main( int argc, char *argv[] )
{
    int size, rank, code;

    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );

    code = rank ? Comrad( size, rank, MPI_COMM_WORLD ) :
               Commissar( size, MPI_COMM_WORLD );

    MPI_Finalize();
    return code;
}

int Commissar( INT4 size, MPI_Comm comm )
{
    static LALStatus status;
    MPIMessage msg;
    REAL4 data;

    while ( size > 1 ) /* while there are still comrades */
    {
        /* receive order from comrad */
        LALMPIRecvMsg( &status, &msg, comm );

        /* handle the message code */
        switch ( msg.msg )
        {
            case 0: /* comrad is quitting */
                --size;
                break;
            case 1: /* comrad is sending data */
                LALMPIRecvREAL4( &status, &data, msg.source, comm );
                printf( "data from comrad %d: %f", msg.source, data );
                break;
            
        }
    }
}
```
default: /* should never happen */
    return 1;
}

return 0;
}

int Comrad( INT4 size, INT4 rank, MPI_Comm comm )
{
    static LALStatus status;
    MPIMessage msg;
    REAL4 data = fmod( rank, size ); /* some junk data */

    /* send message to commissar indicating that comrad is ready to send data */
    msg.msg = 1; /* indicates that data is to be communicated */
    msg.send = 1; /* indicates that comrad (local process) will send */
    msg.source = rank; /* rank of local process */
    LALMPISendMsg( &status, &msg, 0, comm ); /* rank zero is commissar */

    /* commissar now knows data is coming so send it */
    LALMPISendREAL4( &status, &data, 0, comm );

    /* send message to commissar indicating that comrad is quitting */
    msg.msg = 0; /* indicates that comrad is quitting */
    msg.source = rank; /* rank of local process */
    LALMPISendMsg( &status, &msg, 0, comm );

    return 0;
}
34.1.2 Module Exchange.c

Routines to perform MPI exchanges of LAL data types.

Prototypes

```c
void LALInitializeExchange(
    LALStatus *status,
    ExchParams **exchParamsOut,
    ExchParams *exchParamsInp,
    InitExchParams *params
)

void LALFinalizeExchange(
    LALStatus *status,
    ExchParams **exchParams
)

void LALExchangeCOMPLEX16 ( LALStatus *status, COMPLEX16 *object, ExchParams *exchParams )

void LALExchangeCOMPLEX8 ( LALStatus *status, COMPLEX8 *object, ExchParams *exchParams )

void LALExchangeREAL8 ( LALStatus *status, REAL8 *object, ExchParams *exchParams )

void LALExchangeREAL4 ( LALStatus *status, REAL4 *object, ExchParams *exchParams )

void LALExchangeINT8 ( LALStatus *status, INT8 *object, ExchParams *exchParams )

void LALExchangeINT4 ( LALStatus *status, INT4 *object, ExchParams *exchParams )

void LALExchangeINT2 ( LALStatus *status, INT2 *object, ExchParams *exchParams )

void LALExchangeUINT8 ( LALStatus *status, UINT8 *object, ExchParams *exchParams )

void LALExchangeUINT4 ( LALStatus *status, UINT4 *object, ExchParams *exchParams )

void LALExchangeUINT2 ( LALStatus *status, UINT2 *object, ExchParams *exchParams )

void LALExchangeCHAR ( LALStatus *status, CHAR *object, ExchParams *exchParams )

void LALExchangeCOMPLEX16Vector ( LALStatus *status, COMPLEX16Vector *object, ExchParams *exchParams )

void LALExchangeCOMPLEX8Vector ( LALStatus *status, COMPLEX8Vector *object, ExchParams *exchParams )

void LALExchangeREAL8Vector ( LALStatus *status, REAL8Vector *object, ExchParams *exchParams )

void LALExchangeREAL4Vector ( LALStatus *status, REAL4Vector *object, ExchParams *exchParams )

void LALExchangeINT8Vector ( LALStatus *status, INT8Vector *object, ExchParams *exchParams )

void LALExchangeINT4Vector ( LALStatus *status, INT4Vector *object, ExchParams *exchParams )

void LALExchangeINT2Vector ( LALStatus *status, INT2Vector *object, ExchParams *exchParams )

void LALExchangeUINT8Vector ( LALStatus *status, UINT8Vector *object, ExchParams *exchParams )

void LALExchangeUINT4Vector ( LALStatus *status, UINT4Vector *object, ExchParams *exchParams )

void LALExchangeUINT2Vector ( LALStatus *status, UINT2Vector *object, ExchParams *exchParams )

void LALExchangeCHARVector ( LALStatus *status, CHARVector *object, ExchParams *exchParams )
```

void LALExchangeCOMPLEX16TimeSeries ( LALStatus *status, COMPLEX16TimeSeries *object, ExchParams *exchParams )
void LALExchangeCOMPLEX8TimeSeries ( LALStatus *status, COMPLEX8TimeSeries *object, ExchParams *exchParams )
void LALExchangeREAL8TimeSeries ( LALStatus *status, REAL8TimeSeries *object, ExchParams *exchParams )
void LALExchangeREAL4TimeSeries ( LALStatus *status, REAL4TimeSeries *object, ExchParams *exchParams )
void LALExchangeINT8TimeSeries ( LALStatus *status, INT8TimeSeries *object, ExchParams *exchParams )
void LALExchangeINT4TimeSeries ( LALStatus *status, INT4TimeSeries *object, ExchParams *exchParams )
void LALExchangeINT2TimeSeries ( LALStatus *status, INT2TimeSeries *object, ExchParams *exchParams )
void LALExchangeUINT8TimeSeries ( LALStatus *status, UINT8TimeSeries *object, ExchParams *exchParams )
void LALExchangeUINT4TimeSeries ( LALStatus *status, UINT4TimeSeries *object, ExchParams *exchParams )
void LALExchangeUINT2TimeSeries ( LALStatus *status, UINT2TimeSeries *object, ExchParams *exchParams )
void LALExchangeCOMPLEX16FrequencySeries ( LALStatus *status, COMPLEX16FrequencySeries *object, ExchParams *exchParams )
void LALExchangeCOMPLEX8FrequencySeries ( LALStatus *status, COMPLEX8FrequencySeries *object, ExchParams *exchParams )
void LALExchangeREAL8FrequencySeries ( LALStatus *status, REAL8FrequencySeries *object, ExchParams *exchParams )
void LALExchangeREAL4FrequencySeries ( LALStatus *status, REAL4FrequencySeries *object, ExchParams *exchParams )
void LALExchangeINT8FrequencySeries ( LALStatus *status, INT8FrequencySeries *object, ExchParams *exchParams )
void LALExchangeINT4FrequencySeries ( LALStatus *status, INT4FrequencySeries *object, ExchParams *exchParams )
void LALExchangeINT2FrequencySeries ( LALStatus *status, INT2FrequencySeries *object, ExchParams *exchParams )
void LALExchangeUINT8FrequencySeries ( LALStatus *status, UINT8FrequencySeries *object, ExchParams *exchParams )
void LALExchangeUINT4FrequencySeries ( LALStatus *status, UINT4FrequencySeries *object, ExchParams *exchParams )
void LALExchangeUINT2FrequencySeries ( LALStatus *status, UINT2FrequencySeries *object, ExchParams *exchParams )

Description

The routine LALInitializeExchange() is used to set up an exchange protocol that contains the identities of the partners in the communication, which one will be sending and which receiving, and the communicator. The protocol also contains additional information about the type of object that is being exchanged and the number of objects to be exchanged.

In initializing the exchange, there will be one active and one passive partner. The passive partner passes a NULL pointer as input to LALInitializeExchange(), which indicates that this partner is listening for exchange requests. The active partner sends the requested protocol as input. After LALInitializeExchange() returns, both partners will have the agreed upon protocol allocated as output. This protocol is to be destroyed by LALFinalizeExchange() when all the the communication for this exchange is completed. The function LALInitializeExchange() also requires a parameter structure that contains the MPI communicator and the rank of the local process.

The exchange protocol is used as the parameter to the various LALExchange(type) functions. Since the exchange protocol knows who is the sender and who is the receiver, there does not need to be a Send-type and a Recv-type function, just the Exchange-type. If there are multiple objects to be exchanged, the exchange function must be called the specified number of times.

The following is an example with comrads which always initiate exchanges with a commissar:
#include <assert.h>
#include <stdio.h>
#include <math.h>
#include <mpi.h>
#include <lal/LALStdlib.h>
#include <lal/Comm.h>

INT4 lalDebugLevel = 0;

int Commissar( INT4 size, InitExchParams *params );
int Comrad( INT4 rank, InitExchParams *params );

/* the types of exchanges */
enum { ExchData, ExchResults, ExchFinished }

int main( int argc, char *argv[] )
{
  InitExchParams params;
  int size, rank, code;

  MPI_Init( &argc, &argv );
  MPI_Comm_size( MPI_COMM_WORLD, &size );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );

  initParams.mpiComm = comm;
  initParams.myProcNum = rank;

  code = rank ? Comrad( size, rank, &params ) : Commissar( size, &params );

  MPI_Finalize();
  return code;
}

int Commissar( INT4 size, InitExchParams *params )
{
  static LALStatus status;
  INT4 a = 0;
  INT4 b = 1;

  while ( size > 1 )
  {
    ExchParams *thisExch = NULL;
    INT4 obj;

    /* commissar expects to be told the nature of the exchange */
    LALInitializeExchange( &status, &thisExch, NULL, params );

    /* handle the exchange type */
    switch ( thisExch->exchObjectType )
    {
      case ExchData: /* commissar requested to send data */
        assert( thisExch->send == 1 ); /* commissar should send */
        for ( obj = 0; obj < thisExch->numObjects; ++obj )
        {
          INT4 fib = a + b; /* data is next Fibonacci number */
          LALExchangeINT4( &status, &fib, thisExch );
          a = b;
          b = fib;
          a = fib;
          b = fib;
        }
case ExchResults: /* commissar receives results */
    assert( thisExch->send == 0 ); /* commissar should receive */
    for ( obj = 0; obj < thisExch->numObjects; ++obj )
    {
        REAL4 result;
        LALExchangeREAL4( &status, &result, thisExch );
        printf( "result from slave %d: %f\n", thisExch->partnerProcNum, result );
    }
    break;

case ExchFinished: /* comrad is finished */
    --size;
    break;

default: /* this should not happen */
    return 1;
}

LALFinalizeExchange( &status, &thisExch );
}

return 0;
}

int Comrad( INT4 rank, InitExchParams *params )
{
    static LALStatus status;
    ExchParams *thisExch = NULL;
    ExchParams exchData;
    ExchParams exchResults;
    ExchParams exchFinished;
    INT4 data[3];
    INT4 obj;

    /* define the various types of exchanges */
    exchData.exchObjectType = ExchData; /* data will be exchanged */
    exchData.send = 0; /* commissar sends the data */
    exchData.numObjects = 3; /* always get three bits of data */
    exchData.partnerProcNum = 0; /* partner is commissar */

    exchResults.exchObjectType = ExchResults; /* results will be exchanged */
    exchResults.send = 1; /* comrad sends the data */
    exchResults.numObjects = rank; /* different for each comrad */
    exchResults.partnerProcNum = 0; /* partner is commissar */

    exchFinished.exchObjectType = ExchFinished; /* comrad is finished signal */
    exchFinished.send = 0; /* doesn't matter */
    exchFinished.numObjects = 0; /* no things are sent */
    exchFinished.partnerProcNum = 0; /* partner is commissar */

    /* get some data */
    LALInitializeExchange( &status, &thisExch, &exchData, params );
    LALExchangeINT4( &status, &a, thisExch );
    LALExchangeINT4( &status, &b, thisExch );
LALExchangeINT4( &status, &c, thisExch );
LALFinalizeExchange( &status, &thisExch );

/* send some results */
LALInitializeExchange( &status, &thisExch, &exchResults, params );
for ( obj = 0; obj < rank; ++obj )
{
    REAL4 result = fabs( data[obj % 3], rank ); /* some stupid result */
    LALExchangeREAL4( &status, &result, thisExch );
}
LALFinalizeExchange( &status, &thisExch );

/* tell commissar that comrad is finished */
LALInitializeExchange( &status, &thisExch, &exchFinished, params );
LALFinalizeExchange( &status, &thisExch );

return 0;
}
34.1.3 Program *SendRecvTest.c*

Tests the LAL MPI send and receive commands.

**Usage**

Example: run five processes on local machine using LAM

```
/bin/sh
  echo 'hostname' > lamhosts
  rm -f schema
  i=0; while [ $i -lt 5 ]; do echo SendRecvTest >> schema; i=`expr $i + 1`; done
  lamboot -v lamhosts
  mpirun -v schema
  wipe -v lamhosts
```

Example: run five processes on local machine using MPICH

```
/bin/sh
  rm -f machines
  i=0; while [ $i -lt 5 ]; do echo 'hostname' >> schema; i=`expr $i + 1`; done
  mpirun -np 5 -machinefile machines SendRecvTest
```

**Exit codes**

<table>
<thead>
<tr>
<th>Code</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success, normal exit.</td>
</tr>
<tr>
<td>1</td>
<td>Failure.</td>
</tr>
</tbody>
</table>
34.1.4 Program **ExchangeTest.c**

Tests the LAL MPI exchange commands.

**Usage**

Example: run five processes on local machine using LAM

```
/bin/sh
  echo 'hostname' > lamhosts
  rm -f schema
  i=0; while [ $i -lt 5 ]; do echo ExchangeTest >> schema; i=`expr $i + 1`; done
  lamboot -v lamhosts
  mpirun -v schema
  wipe -v lamhosts
```

Example: run five processes on local machine using MPICH

```
/bin/sh
  rm -f machines
  i=0; while [ $i -lt 5 ]; do echo 'hostname' >> schema; i=`expr $i + 1`; done
  mpirun -np 5 -machinefile machines ExchangeTest
```

**Exit codes**

<table>
<thead>
<tr>
<th>Code</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success, normal exit.</td>
</tr>
<tr>
<td>1</td>
<td>Failure.</td>
</tr>
</tbody>
</table>
Manual III

Coding and Documentation Instructions
Section 14

How to Develop Code for LAL
Chapter 35

An Introduction to the LIGO/LSC Algorithm Library (LAL)

35.1 The LAL webpage

An up to date release of the LAL – and a wealth of other information – can be obtained from the LAL webpage http://www.lsc-group.phys.uwm.edu/lal/

- **Current distribution**: You can click to down load the tar ball with all the code.
- **Report a bug**: You can click to enter the LAL bug reporting system.
- **LAL software librarian**: You can click to email the librarian.
- **LSC software coordinator**: Click to email your gripes to the software coordinator.
- **The LAL-Spec**: www.ligo.caltech.edu/docs/T/T990030-07.pdf
- **The CVS tree**: There are instructions on how to download the latest version of LAL from the cvs. **Installation instructions** can be found in the lal directory in a file called INSTALL
- **Previous releases of LAL**: If you want an older version for sentimental reasons, you can find it here.

- **Data**: You can surf to the LIGO data archive to get data. There is a short snippet of 40-m data called *lal.small* for testing. This snippet of data is not subject to the access restrictions that some of the other segments are.
- **Links to other useful software**: In order to fully use the LAL, you will need to install the these packages:
  - **FFTW**: Fastest Fourier Transform in the West. This *must* be installed in order to install the LAL.
  - **MPI**: For parallel coding. This is optional.
  - **frames**: Package for reading frame format data. This is optional. In order to test these routines you will need some frame data. [See Data above.]
Chapter 36

Notes about coding

As mentioned above, the marching orders for code development laid out in the LAL-Spec are often quite general. The purpose of this chapter is try to record the collective interpretation of the LAL-Spec so that we can consistently apply it.

**Use of MACROS:** The LAL-Spec says “macros” are deprecated. What the hell does that mean? The interpretation we are using clearly captures the intent. There are a number of macros for common use in the `std` LAL files, but these under strict control of the librarian. In your modules (.c files), you may use macros to replace small snippets of code. However because header files may be included in other code, you should not use macros in your `.h` files.

If you have a “`#define SomeMacro`” that needs to be included in many different files, it probably belongs in the `LALConstant.h` file or one of the other `/lal/std/include` files. Please contact the Librarian.

**Defining the RCS ID string:** The LAL-Spec requires all header, module and test files to define the RCS ID String with the following macro:

```c
```

The reason we assign the Id with this macro is that without it, the compiler prints annoying warning messages. Note this macro is an example of one macros for “common use” described above.

**Error Codes and Messages:** The LAL-Spec discusses the statuscode and statusDesription, i.e. the error codes and messages returned by a LAL function. Here we solidify the name space convention. These should be hash-defined in the header file, e.g. in `MyHeader.h` we would have

```c
/* <lalErrTable file="MyHeaderHErrorTable"> */
#define MYHEADERH_ENUL 1
#define MYHEADERH_EOUT 2
#define MYHEADERH_EDIV 3

#define MYHEADERH_MSGENUL "Null pointer"
#define MYHEADERH_MSGEOUT "Output already exists"
#define MYHEADERH_MSGEDIV "Division by zero"

/* </lalErrTable> */
```

The names should begin with the file name and extension (`h`) all converted to upper case. The error codes are followed by `E<name>`. The error messages are followed by `MSGE<name>`.

The `<>`ed key-words shown before and after the codes are the key-words for automatic documentation system. You must use these to automate the inclusion of the error codes in the documentation. More on this later.
Chapter 37

The directory structure of the LAL
37.1 Schematic Diagram of Directory Structure

This is a diagram showing the directory structure of the LAL. This comes straight from the LAL-Spec. In particular, this shows an example package tdfilter (time domain filters).

Contents of a "Typical" package submitted to the LAL by an LSC member. (Not all files shown.)
Chapter 38

Documenting your code

Along with any code submission to the LAL library, you will need to supply documentation. Keep in mind, the documentation, like the code, is a deliverable and it must be written to the standard outlined in the LAL-Spec. This chapter gives specific instructions on how to meet the standard.

Also keep in mind that, unlike most code projects that physicists work on, this code may still need to be maintained long after the author has been denied tenure and starts working for a dot-com company. This puts a heavy burden on the documentation: not only should it help you maintain your code, but it should require minimum effort for anyone to figure out how the code works and how to fix it. If you find yourself saying “The easiest way for me to maintain my code is ...”, you have missed the point.

In this chapter we discuss a few preliminaries, and give a general outline of the documentation for a package. Chapter 39 is an example of how a the documentation for a LAL package should be laid out. In Chapter 40 we explain how to use the auto-documentation system (laldoc).

38.1 Use $\LaTeX$

The documentation should be written in $\LaTeX$. This decision was made by the LSC software committee. The primary reason for this choice was the need for the equation-writing capability of $\LaTeX$.

The danger in using $\LaTeX$ is that not everyone will have the same version of $\LaTeX$ installed on their machine. [Actually, this problem might be as bad, or worse, with some other documenting tool.] Try to help us minimize this problem by using vanilla $\LaTeX$.

Along with the LAL distribution, we supply a class file (lal/doc/lal.sty). Although this is helpful and it makes thing look nicer, it is not essential. You can remove it from the $\texttt{\usepackage}[]$ command in the various files and they should still successfully $\LaTeX$ without it.
38.2 The lay out of the documentation

Most of the LDAS software is written in c++, and therefore the documentation is naturally built around “classes”. However, the LAL is written in c, and thus we cannot directly adopt the LDAS style of documentation. None the less, we try to mimic the style as close as possible by building the LAL documentation around header files and the modules and functions that include them. This choice is also natural because the hierarchical lay out of the documentation exactly follows the hierarchy of the code. [This is also the way books on programming in c document the c-libraries.]

- Documentation for the N’th package forms chapter N.
  - Documentation of header1.h in the package forms section N.1.
    * Documentation of Module1.1.c that “#include’s” header1.h is in subsection N.1.1.
    * Documentation of Module1.2.c that “#include’s” header1.h is in subsection N.1.2.
    * ... additional modules.
  - Documentation of header2.h in the package forms section. N.2.
    * Documentation of Module2.1.c that “#include’s” header2.h is in subsection N.2.1.
    * Documentation of Module2.2.c that “#include’s” header2.h is in subsection N.2.2.
    * ... additional modules
  - Documentation of header3.h in the package forms section. N.3.
    * ... additional modules
  - ... additional headers

- Documentation for the N+1’th package forms chapter N+1.

- ... additional packages

Note: Although the LAL-Spec is not rigid on the subject, it suggests that the number of modules and functions encompassed by a header file should be small: only “small sets of related functions” should share the same header file. This means for a given header-file section, there shouldn’t be too many module subsections.

38.3 Documentation for a single package versus a comprehensive LAL manual

The LAL-Spec mentions two distinct forms the documentation must take: stand-alone documentation for a single package and a comprehensive manual (with an exhaustive index and table of contents) for the entire LAL. In this section we describe how we implement these two competing documentation specifications using the same \LaTeX source to build both type of documents.

The requirement of both forms of the documentation in the LAL-Spec is not capricious: both forms of the documentation are useful. When you are working on a single package, having a short, single-package document is handy. This way you won’t have to wait while \LaTeX runs on the complete manual for the entire LAL every time you want to see if your equations line up. On the other hand, as the LAL code becomes more mature and some of it starts to perform highly integrated tasks, it will be necessary for coders to have quick access to the entire body of documentation.

**Documentation for a single package:** The guts of the documentation for a package (e.g. sample) should reside in the file /lal/packages/samplepackage/doc/sample.tex. This file may itself have many \input{} commands in it to include files that were auto extracted from the source-code files by laldoc.

When LAL is built, the file sample.tex will be automatically included in the file main.tex to build the stand alone documentation main.pdf for this (or any package). In other words, the file main.pdf in any lal/package/doc is the documentation for that package.

One problem you can have when you build the documentation of a single package is that if their are references (\ref{}’s) to objects outside the package, these will be left unresolved by \LaTeX. Try to minimize these by referring to the objects by name rather than section number. Remember, when the comprehensive document is built, it will have a complete index and table of contents, so the reader should be able to easily find the documentation for the objects by name.

**Comprehensive manual for the entire LAL:** In the comprehensive documentation directory (/lal/doc) their is a file lsd.tex (lsd = LAL Software Documentation). This plays essentially the same role as main.tex described above; however this file has an \include{} statement for every package in the LAL.
38.4 What about figures?

Figures are fine. When you submit a package of code and documentation the figures should be in the
We need both so we can build the documentation either with \LaTeX or pdflatex.

The syntax to use for putting in the figure should be something like this
\resizebox{0.5\textwidth}{!}{\includegraphics{myHeaderFileNameMyFig}}

Note. Don’t include the extension on the figure file name in the \includegraphics command. If you
leave the extension off, whatever method you use for building the documentation automatically looks for the
appropriate file.

Naming convention for figure files: Using the base name of the header file as the first part of
the figure file name is a good idea. The reason: when the documentation is built, everything is \LaTeXed in the
lal/doc/autodoc directory. There are hundreds of files there, and following this convention will reduce the
probability of a name-space collision.

38.5 Autodocumentation and Indexing

38.5.1 Autodocumentation requirements

There is an automatic documentation tool supplied with the LAL. To what extent the coders wish to use
the autodocumentation system, is largely left up to their judgment and patience. However, the LAL-Spec
require several items to be auto-extracted from the code to include the documentation. See the LAL-Spec
for the official list, but here are some examples

- Function Prototypes
- Error code tables
- The author and version-control information. This should appear as footnote at the bottom of
  all header sections, and module and test subsections. This can be done with the \LaTeX command
  \vfill\footnotesize{\input{MyFileHAuthVer}}, where MYFILEHAuthVer.tex is the file where the
  Author-Version information from MyFile.H was extracted to. You can find examples of this to mimic.

38.5.2 Indexing requirements

When this document is built an index is also constructed. There are a few code items that you must place
in the index.

- Functions must be entered in the index, so users can find them. The \idx{} command should be right
  after the prototypes themselves are entered. This insures the page number that appear in the index
  will be the page where the prototype is explained in the document. Use the LAL \LaTeX command
  \idx{MyFunction()}

- Non LAL Data Structures must appear in the index. The LAL-Spec strongly encourages the using
  LAL datatypes as the arguments for a function, but this isn’t always possible. If you do use non-LAL
  datatypes, you must document them, and they must be included the index, so someone looking at
  your code can easily find the documentation. The \idx[type]{MyType} command should be right after in
  the section where they are explained in the documentation. Here is an example of the LAL \LaTeX
  command you would use to get the name of your structure into the index.

    \idx[type]{MyType}

- Other Indexable Things should also be indexed. If the thing is a constant variable, a constant-like
  macro, or an enum constant, index it using \idx[Constant]{thing}. If the thing is a function-like
  macro, index it using \idx[Function]{thing} (which is equivalent to \idx[Function]{thing}). If it is a macro
  that is non-function-like and non-constant-like then index it using \idx[Macro]{thing}. If the thing
  is a variable (including function-like variables) then index it using \idx[Variable]{thing}. If the
  thing is a concept, then just use the standard \LaTeX command \index{thing}. 
Chapter 39

Package samplepackage

An introductory description of what is in the package. There is no specific length limit, but $O[1]$ page seems reasonable. Note the naming conventions for packages: all lower case.
39.1 Header SampleHeader1.h

Put a one sentence description of the header right after the section heading.

Synopsis and description

#include "SampleHeader1.h"

Since it is possible that a few modules will use the same header file you can put a general description of what is to come here, and save more specific comments for the module and function documentation.

Error conditions

To insure that these are current with the code, a table of these must be automatically extracted from the source code with the laldoc auto-documenter. Additional explanation (if necessary) of the error conditions can follow the table.

Structures

The LAL-Spec encourages the use of LAL datatypes whenever possible, but, if you must use non-LAL structures, they must be documented here. They must also be included in the index with a \index{sampleStruct} \LaTeX command.
39.1.1 Module SampleModule1.c

Put a one sentence description explaining what this module will do. [Also, start a new page for documenting each module.]

Prototypes

Function prototypes must be extracted verbatim from the source code and \texttt{\textbackslash input{}} here. Every function must be placed into the index with a \texttt{\textbackslash index{SampleFunction{}}} \LaTeX command.

The LAL-Spec encourages modularization: \textit{“Small sets of related functions may grouped together into a single file.”} [Emphasis added.] Therefore, if the list of prototypes gets too long and some aren’t closely related to others, perhaps it is time to break it up into a separate modules.

Description and operating instructions

Describe the arguments of the function, and explain how to use the function. Remember to document any non LAL structs in the header file documentation.

Algorithm

Explanation of the algorithm.

Uses

A list of all the other routines that this module uses.

Notes

Validation Information

This section will be formally filled in when the code is officially validated. In the mean time, if you have timing or bench-mark testing information, put it here.
Program SampleTest.c

Brief description, e.g. “Performs tests on all routines associated with SampleHeader.h.”

Usage

Show and explain the command line syntax used to run the program.

Description and operating instructions

Explanation of the algorithm.

Exit Codes

A table containing all the exit codes for the program. We strongly suggest that the exit codes be coded in exactly the same way as the error codes in the header file. If you do this you can use the laldoc Error Table tool to build a table to insert here. (See below.) If you don’t use laldoc to make the table, please \LaTeX{} it by hand.

The table may followed with additional explanation.

Uses

A list of all the other routines that this module uses.

Notes
References

This should be a stand-alone bibliography for the references used in this package only. We recognize there may be entries that are repeated in other packages and there are more clever ways to do this, but this seems to be the simplest.
Index

The index for the Package *samplepackage* should start on a new page. The \texttt{\textbackslash printindex} command should be in the \texttt{main.tex} file, because we only want this index to appear in the stand-alone documentation for this package. When we the comprehensive documentation is built, only the all-encompassing index will appear at the end of the entire document.
Section 15

LALDoc: Automated Documentation
Chapter 40

The automatic documentation system

We will first describe the code parser that sifts through the source code and extracts the parts that are to be incorporated into the documentation. Then we describe how the documentation is automatically built when the LAL is installed.

40.1 A four step introduction to the code parser laldoc

These four-step instructions are also outlined in the file /lal/doc/laldoc/README.

1. Copy the executable (/lal/doc/laldoc/laldoc) and the sample input file (/lal/doc/laldoc/LalDocDemo.h) to an empty /home/alice/junk directory.

2. In /home/alice/junk run the command “./laldoc LalDocDemo.h Errors.out”

3. L\TeX the file LalDocDemo_LaTeX_This_File.tex.

4. Read the document created in step 3 (LalDocDemo_LaTeX_This_File.dvi). It will explain how the various debris files were created from the original source file and how they were input to make the document. If you follow your nose, you will see exactly what laldoc does and how it works. Also, make sure you look at the file Errors.out.

40.2 Brief Description of the parser laldoc

The purpose of the auto-documentation procedure we are using with the LAL is to adhere the common wisdom: “keep the documentation close to the code”. The system is built around a code parser laldoc that allows LAL programmers automatically extract fragments of code or comments from the source files and include them in their documentation. This is done in such a way that if the fragment in the source code is modified, then the change is automatically incorporated into the documentation the next time the document is built.

This extraction is accomplished by having the programmer surround the fragments of code or comments he or she wishes to incorporate in the document with key-words. Currently, we only have three pairs of key-words, so the learning curve is flat and short! The source code (the .c and .h files) is then parsed and the fragments written to storage files. (The key-word also includes a space for user-specified file name where the fragment will be stored.) When you write your documentation simply use the L\TeX command \input{} to put the contents of the storage file in the document where you need it.

When you install the LAL on your machine, the parser is automatically run on all .c and .h files. The extracted files are stuffed into the directory /lal/doc/autodoc/. If you have installed the LAL software package, take a look at the contents of /lal/doc/autodoc/. Their are dozens of extracted files their.

40.2.1 The laldoc command line

The full functionality of the command line is:

laldoc inputFile.c errorFile /home/alice/errorDir/ /home/alice/inputDir/
The zeroth command line argument is the executable itself. The first argument is the input file name, the second is an error reporting file, the third argument is the directory where the error file will be written, and the fourth is the directory where \textit{laldoc} will look for the input file. The third and the fourth arguments are optional.

\subsection{The three \textit{laldoc} environments}

One general comment about the \textit{laldoc} environments: None of the text on the line with the opening key-word or on the line with the closing key-word will be extracted.

- The verbatim environment is opened key-word \texttt{<lalVerbatim file="myVerbatimJunk">} and closed with the key-word \texttt{</lalVerbatim>}. The material between the two key-words will be wrapped in a \LaTeX \texttt{verbatim} environment for later inclusion. This is useful for including such things as a function prototypes or data structures: they will appear “verbatim” in the documentation. When the information is included with \texttt{<lalVerbatim>}, \textit{laldoc} supplies a small \texttt{marginpar} gives the source-file name and line number where the snippet came from.

- The \LaTeX environment is opened and closed with the key-words \texttt{<lalLaTeX file="myLatexJunk">} and \texttt{</lalLaTeX>}. This is used to write \LaTeX in the source-code. The material between the two key-words is stored in a file \texttt{myLatexJunk.tex}. This allows (not recommended) a programmer to put large sections of \LaTeX prose in the source code.

- The error-table environment. There is a special environment for translating the source code that assign the error codes and messages directly into a \LaTeX table. All you need to do is wrap the code between the key-words \texttt{<lalErrTable file="myErrTabJunk">} and \texttt{</lalErrTable>}. This insures that if an error code is added in the source, it will automatically be added to the documentation.

\subsection{How \textit{laldoc} handles the output files}

\textbf{Default file names:} In any of the \textit{laldoc} environments, if you do not specify an output file in the opening key-word line, \textit{laldoc} will assign one automatically. The file name will be constructed from the input file name, e.g. if the input is \texttt{MyHeader.h}, the default output will be \texttt{MyHeaderH.tex}.

\textbf{Appending to files:} If the output file doesn’t already exist, \textit{laldoc} will create it. If the output file already exists, \textit{laldoc} will append to it. When the environment-closing key-word is encountered, the output file is closed.

Although this is a fairly obvious feature, it is quite useful. For example, the function prototypes in a module don’t appear together in the source, but they should appear together in the documentation. With \textit{laldoc} this is easy to accomplish. In your source code when you encounter each prototype that must be captured for inclusion, use the same file name for the extraction, i.e surround each prototype with the pair \texttt{<lalVerbatim file="MyModuleCPrototypes">} and \texttt{</lalVerbatim>}. Each prototype will be appended to the file \texttt{MyModuleCPrototypes.tex}, and separate margin pars will tell exactly where each came from.

\section{Examples of how to use the three environments in \textit{laldoc}}

As with most things computer, the best way to learn how use the auto documenter is to snoop around the source tree and find some examples, and then try a few things. The source code and the executable for the parser (\textit{laldoc}), are in the directory \texttt{lal/doc/laldoc}. The directory also has a README file that give some nuts-n-bolts instructions on how to use the parser. This sections gives the basics of what you can do.

\subsection{The \texttt{<lalVerbatim>} environment}

As an example, look at the code fragment in the source file \texttt{lal/packages/tdfilter/src/CreateZPGFilter.c}.

\begin{verbatim}
/* <lalVerbatim file="CreateZPGFilterCP"> */
void CreateCOMPLEX8ZPGFilter(LALStatus *stat, COMPLEX8ZPGFilter **output, INT4 numZeros, INT4 numPoles)
/* </lalVerbatim> */
\end{verbatim}
When `laldoc` parses the source file, it produces the following output in the file `CreateZPGFilterCP.tex`

```latex
\begin{verbatim}
void CreateCOMPLEX8ZPGFilter(LALStatus *stat, 
    COMPLEX8ZPGFilter **output, 
    INT4 numZeros, 
    INT4 numPoles)
\end{verbatim}
```

This is then included in the \LaTeX{} documentation with an `\input{CreateZPGFilterCP}` command, yielding the following output:

```latex
\begin{verbatim}
void CreateCOMPLEX8ZPGFilter(LALStatus *stat, 
    COMPLEX8ZPGFilter **output, 
    INT4 numZeros, 
    INT4 numPoles)
\end{verbatim}
```

The `marginpar` on the far right tells the line number and file name where the fragment came from.

Note the naming convention used for the file where the extracted code was stored: the base name comes from the file where it was extracted (here `CreateZPGFilter`), followed by a “C” (in this case to denote that it came from the `.c` file). Use This Naming Convention! This will avoid most name-space collisions. In this case, the the “P” is for function `prototype`.

40.3.2 The `<lalLaTeX>` environment

The `<lalLaTeX>` environment works much the same was as the `<lalVerbatim>` environment. The distinction being that the extracted material should be valid \LaTeX{} ready for insertion into a \LaTeX{} file.

In the `<lalLaTeX>` environment, leading `*`’s on a line will be stripped out by `laldoc`. This is to accommodate the common practice in c of putting leading `*`’s on comment lines. The way this is done is `laldoc` checks to see if the first non blank character on the line is a `*`. If it is, then it is replaced by blank when the line is written to the output file. The leading blanks are ignored when the file is \LaTeX{}ed.

40.3.3 The `<lalErrTable>` environment, for printing a table of the error codes and warnings.

See the example in Chapter 36

40.4 How the documentation is automatically built

Some of the documentation (perhaps most of it) for a package is in the source files awaiting extraction. The rest of the documentation for the package is in the `lal/packages/mypackage/doc/` directory. How does it all get pulled together to make a coherent document? The short version of the explanation is that all the necessary files either exist in, or are copied to, or are linked to the directory `lal/doc/autodoc/`, then \LaTeX{} is run in the `autodoc/` directory and important results are moved back. You can see the specifics of how this happens (and how to modify it for your own use) if you examine the contents of the `Makefile.am’ s in the various `doc/` directories.

When you type `make dvi` during the installation, a chain of events takes place. First, the `laldoc` parser is run in each source directory (`src`, `include` and `test`) in all the packages. The documentation snippets are culled out of the source files and stuffed into files in `/lal/doc/autodoc` directory. When this is complete, the stand-alone documentation is built one directory at time. First, links from the `.tex`, `.eps` and `.pdf` files to the `autodoc` directory are established. Then the `main.tex` file is copied from the package’s `/doc` directory to the `autodoc` directory. Then `main.tex` is \LaTeX{}ed (actually, we run `pdflatex`). When this is complete, `main.pdf` and `main.tex` are moved back the package’s `doc` directories. This is repeated for each package.

After the documentation for the individual packages is built, then the comprehensive documentation is built in essentially the same way. The file `/lal/doc/lsd.tex` is copied to the `autodoc` directory and \LaTeX{} (pdflatex) is run on it. Then `lsd.tex` and `lsd.pdf` are moved back to the `doc` directory.
Index

[ , 75
AMCoeffsParams, 839
AMCoeffs, 838
AvgVelPar, 761
BlockRho2(), 473
CSBParams, 834
CSParams, 830
ClusterBlobWavelet, 511
CmplxAMCoeffs, 838
ComputeSky(), 832
ComputeSkyBinary(), 837
HOUGHBin2Border, 728
HOUGHBorder, 728
HOUGHDemodPar, 730
HOUGHMapDeriv, 747
HOUGHMapTotalVector, 753
HOUGHMapTotal, 747
HOUGHParamPLUT, 731
HOUGHPatchGrid, 729
HOUGHPeakGramVector, 753
HOUGHPeakGram, 742
HOUGHResidualSpinPar, 753
HOUGHResolutionPar, 730
HOUGHphmd, 742
HOUGHptfLUTVector, 753
HOUGHptfLUT, 729
HexagonOut, 531
HoughStats, 755
InputClusterWavelet, 514
InputCoincidenceWavelet, 513
InputGetMaxLayerWavelet, 512
InputLayerWavelet, 512
InputPercentileWavelet, 513
InputPixelMixerWavelet, 513
InputPixelSwapWavelet, 513
InputPixelWavelet, 511
REAL8Cart2Coor, 730
REAL8Cart3Coor, 730
REAL8Polar2Coor, 730
REAL8UnitPolarCoor, 730
RectangleIn, 531
RectangleOut, 531
Slice, 710
StatsREAL4VectorOut, 697
UINT8FrequencyIndexVectorSequence, 754
UINT8FrequencyIndexVector, 752
VelocityPar, 761
Wavelet, 510
rk4GSLIntegrator, 649
rk4In, 649
LALAvgDetectorVel(), 762
LALColoredNoise(), 711
LALDetectorVel(), 762
LALEOBwaveform(), 668
LALEOBwaveformForInjection(), 668
LALEOBwaveformTemplates(), 668
LALHOUGHAddPHMD2HD(), 749
LALHOUGHcomputeFBinMap(), 754
LALHOUGHcomputeNSizePar(), 734
LALHOUGHcomputeSizePar(), 734
LALHOUGHconstructHMT(), 754
LALHOUGHconstructPLUT(), 753
LALHOUGHconstructSpacePHMD(), 754
LALHOUGHFillPatchGrid(), 734
LALHOUGHinitializeHD(), 749
LALHOUGHinitializeHT(), 749
LALHOUGHIntegrd2HT(), 749
LALHOUGHParamPLUT(), 746
LALHOUGHpeak2PHMD(), 744
LALHOUGHupdateSpacePHMDdn(), 754
LALHOUGHupdateSpacePHMDup(), 754
LALHoughHistogram(), 759
LALHoughStatistics(), 759
LALInspiralChooseModel(), 654
LALInspiralComputeChisq(), 718
LALInspiralComputeSNRIntegrand(), 719
LALInspiralFindEvents(), 714
LALInspiralFindEventsCluster(), 716
LALInspiralFindLoudestEvent(), 715
LALInspiralFrequency3(), 688
LALInspiralPhasing1(), 677
LALInspiralPhasing2(), 680
LALInspiralPhasing3(), 682
LALInspiralPhiofVIntegrand(), 678

1047
Function

LALBarycenter(), 779
ABORT(), 29
ASSERT(), 29
ATTACHSTATUSPTR(), 29
CHECKSTATUSPTR(), 29
DETACHSTATUSPTR(), 29
FREESTATUSPTR(), 55
INITSTATUS(), 28
InspiralTmpltBankFromLIGOLw(), 972
LALAbort(), 83
LALAddFloatToGPS(), 88
LALAddSnglInspiralToCoinc(), 169
LALAddVectors(), 712
LALAdvLIGOPsd(), 704
LALalphaFCutSingleInspiral(), 164
LALApplyResampleRules(), 809
LALAttachStatusPtr(), 55
LALBCVCVetoSingleInspiral(), 164
LALBeginLIGOLwXMLTable(), 966
LALButterworthREAL4TimeSeries(), 312
LALButterworthREAL8TimeSeries(), 312
LALCalllocLong(), 47
LALCalllocShort(), 47
LALCCCoarseGrainFrequencySeries(), 356
LALCCCreateArray(), 99
LALCCCreateGrid(), 65
LALCCCreateVector(), 97
LALCCCreateVectorSequence(), 106
LALCCVectorDivide(), 231
LALCCVectorMultiply(), 231
LALCCVectorMultiplyConjugate(), 231
LALCDestroyArray(), 99
LALCDestroyGrid(), 65
LALCDestroyVector(), 97
LALCDestroyVectorSequence(), 106
LALCHARCreateVector(), 97
LALCHARCreateVectorSequence(), 106
LALCHARDestroyVector(), 97
LALCHARDestroyVectorSequence(), 106
LALCHARPrintVector(), 923
LALCHARReadSequence(), 943
LALCHARReadVector(), 939
LALCHARReadVectorSequence(), 941
LALCheckConfigReadComplete(), 979
LALCheckMemoryLeaks(), 47
LALCheckOutTimeFromSearchSummary(), 160
LALCheckStatusPtr(), 55
LALCleanAll(), 263
LALCloseLIGOLwXMLFile(), 966
LALClustersSnglInspiralTable(), 164
LALClustersPowerThreshold(), 395
LALCMatrixAdd(), 369
LALCMatrixAdjoint(), 369
LALCMatrixMultiply(), 369
LALCMatrixTranspose(), 369
LALCoarseFitToPulsar(), 852, 853
LALCoarseHeterodyne(), 847
LALCoincSnglInspiral(), 169
LALCompareGPS(), 88
LALCompareInspirals(), 164
LALCompareSearchSummaryByInTime(), 160
LALCompareSearchSummaryByOutTime(), 160
LALCompareSnglInspiral(), 164
LALCompareSnglInspiralByMass(), 164
LALCompareSnglInspiralByPsi(), 164
LALCompareSnglInspiralByTime(), 164
LALCompareSummValueByTime(), 160
LALCOMPLEX8VectorFFT(), 277
LALComputeDetAMResponse(), 131
LALComputeDetAMResponseSeries(), 131
LALComputeTFCSSpectrogram(), 493
LALComputeTransfer(), 155
LALComputeXTFCSpectrogram(), 493
LALConvertSkyCoordinates(), 406
LALCopyCLlist(), 497
LALCReadFrequencySeries(), 928
LALCReadTimeSeries(), 926
LALCReadVector(), 923
LALCReadFrequencySeries(), 931, 945
LALCReadGrid(), 949
LALCReadSequence(), 943
LALCReadTimeArraySeries(), 945
LALCReadTimeSeries(), 932, 945
LALCReadTimeVectorSeries(), 945
LALCreateArray(), 99
LALCreateCOMPLEX16FrequencySeries(), 197
LALCreateCOMPLEX16TimeSeries(), 209
LALCreateCOMPLEX16ZPGFilter(), 296
LALCreateCOMPLEX8FrequencySeries(), 197
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LALFindChirpSPTemplate()</td>
<td>607</td>
</tr>
<tr>
<td>LALFindChirpTDNormalize()</td>
<td>610</td>
</tr>
<tr>
<td>LALFindChirpTDTemplate()</td>
<td>610</td>
</tr>
<tr>
<td>LALFindChirpTemplateFinalize()</td>
<td>597</td>
</tr>
<tr>
<td>LALFindChirpTemplateInit()</td>
<td>597</td>
</tr>
<tr>
<td>LALFineFitToPulsar</td>
<td>852</td>
</tr>
<tr>
<td>LALFineHeterodyneToPulsar()</td>
<td>847</td>
</tr>
<tr>
<td>LALFloatToGPS()</td>
<td>86</td>
</tr>
<tr>
<td>LALFloatToInterval()</td>
<td>86</td>
</tr>
<tr>
<td>LALFoldAmplitudes()</td>
<td>818</td>
</tr>
<tr>
<td>LALFopen()</td>
<td>921</td>
</tr>
<tr>
<td>LALForwardREAL4FFT()</td>
<td>269</td>
</tr>
<tr>
<td>LALForwardRealFFT()</td>
<td>269</td>
</tr>
<tr>
<td>LALFrCacheExport</td>
<td>994</td>
</tr>
<tr>
<td>LALFrCacheGenerate</td>
<td>994</td>
</tr>
<tr>
<td>LALFrCacheImport</td>
<td>994</td>
</tr>
<tr>
<td>LALFrCacheOpen</td>
<td>1000</td>
</tr>
<tr>
<td>LALFrCacheSieve</td>
<td>994</td>
</tr>
<tr>
<td>LALFrClose</td>
<td>1000</td>
</tr>
<tr>
<td>LALFrEnd</td>
<td>1000</td>
</tr>
<tr>
<td>LALFreqTimeComplexFFT()</td>
<td>282</td>
</tr>
<tr>
<td>LALFreqTimeRealFFT()</td>
<td>282</td>
</tr>
<tr>
<td>LALFrGetCOMPLEX16FrequencySeries</td>
<td>1005</td>
</tr>
<tr>
<td>LALFrGetCOMPLEX16FrequencySeriesMetadata</td>
<td>1005</td>
</tr>
<tr>
<td>LALFrGetCOMPLEX16TimeSeries</td>
<td>1005</td>
</tr>
<tr>
<td>LALFrGetCOMPLEX16TimeSeriesMetadata</td>
<td>1005</td>
</tr>
<tr>
<td>LALFrGetCOMPLEX8FrequencySeries</td>
<td>1005</td>
</tr>
<tr>
<td>LALFrGetCOMPLEX8FrequencySeriesMetadata</td>
<td>1005</td>
</tr>
<tr>
<td>LALFrGetINT2FrequencySeries</td>
<td>1005</td>
</tr>
<tr>
<td>LALFrGetINT2FrequencySeriesMetadata</td>
<td>1005</td>
</tr>
<tr>
<td>LALFrGetINT2TimeSeries</td>
<td>1005</td>
</tr>
<tr>
<td>LALFrGetINT2TimeSeriesMetadata</td>
<td>1005</td>
</tr>
<tr>
<td>LALFrGetINT4FrequencySeries</td>
<td>1005</td>
</tr>
<tr>
<td>LALFrGetINT4FrequencySeriesMetadata</td>
<td>1005</td>
</tr>
<tr>
<td>LALFrGetINT4TimeSeries</td>
<td>1005</td>
</tr>
<tr>
<td>LALFrGetINT4TimeSeriesMetadata</td>
<td>1005</td>
</tr>
<tr>
<td>LALFrGetINT8FrequencySeries</td>
<td>1005</td>
</tr>
<tr>
<td>LALFrGetINT8FrequencySeriesMetadata</td>
<td>1005</td>
</tr>
<tr>
<td>LALFrGetINT8TimeSeries</td>
<td>1005</td>
</tr>
<tr>
<td>LALFrGetINT8TimeSeriesMetadata</td>
<td>1005</td>
</tr>
<tr>
<td>LALFrGetREAL4FrequencySeries</td>
<td>1005</td>
</tr>
<tr>
<td>LALFrGetREAL4FrequencySeriesMetadata</td>
<td>1005</td>
</tr>
<tr>
<td>LALFrGetREAL4TimeSeries</td>
<td>1005</td>
</tr>
<tr>
<td>LALFrGetREAL4TimeSeriesMetadata</td>
<td>1005</td>
</tr>
<tr>
<td>LALFrGetREAL8FrequencySeries</td>
<td>1005</td>
</tr>
<tr>
<td>LALFrGetREAL8FrequencySeriesMetadata</td>
<td>1005</td>
</tr>
<tr>
<td>LALFrGetREAL8TimeSeries</td>
<td>1005</td>
</tr>
<tr>
<td>LALFrGetREAL8TimeSeriesMetadata</td>
<td>1005</td>
</tr>
<tr>
<td>LALFrGetTimeSeriesType</td>
<td>1005</td>
</tr>
<tr>
<td>LALFrNext</td>
<td>1000</td>
</tr>
<tr>
<td>LALFrOpen</td>
<td>1000</td>
</tr>
<tr>
<td>LALFrRewind</td>
<td>1000</td>
</tr>
<tr>
<td>LALFrSeek</td>
<td>1000</td>
</tr>
<tr>
<td>LALFrSetPos</td>
<td>1000</td>
</tr>
<tr>
<td>LALFrTell</td>
<td>1000</td>
</tr>
<tr>
<td>LALFrWriteCOMPLEX16TimeSeries</td>
<td>1005</td>
</tr>
<tr>
<td>LALFrWriteCOMPLEX8TimeSeries</td>
<td>1005</td>
</tr>
<tr>
<td>LALFrWriteINT2TimeSeries</td>
<td>1005</td>
</tr>
<tr>
<td>LALFrWriteINT4TimeSeries</td>
<td>1005</td>
</tr>
<tr>
<td>LALFrWriteINT8TimeSeries</td>
<td>1005</td>
</tr>
<tr>
<td>LALFrWriteREAL4TimeSeries</td>
<td>1005</td>
</tr>
<tr>
<td>LALFrWriteREAL8TimeSeries</td>
<td>1005</td>
</tr>
<tr>
<td>LALFrWriteREAL8TimeSeriesMetadata</td>
<td>1005</td>
</tr>
<tr>
<td>LALGalacticInspiralParamsToSimInspiralTable()</td>
<td>172</td>
</tr>
<tr>
<td>LALGalacticToEquatorial()</td>
<td>459</td>
</tr>
<tr>
<td>LALGenerateEllipticSpinOrbitCW</td>
<td>448</td>
</tr>
<tr>
<td>LALGenerateHyperbolicSpinOrbitCW</td>
<td>452</td>
</tr>
<tr>
<td>LALGenerateInsipral</td>
<td>124</td>
</tr>
<tr>
<td>LALGenerateInspiralPopulateInspiral</td>
<td>124</td>
</tr>
<tr>
<td>LALGenerateInspiralPopulatePPN</td>
<td>124</td>
</tr>
<tr>
<td>LALGenerateInspiralSmooth</td>
<td>433</td>
</tr>
<tr>
<td>LALGenerateParabolicSpinOrbitCW</td>
<td>450</td>
</tr>
<tr>
<td>LALGeneratePPNampCorInspiral</td>
<td>432</td>
</tr>
<tr>
<td>LALGeneratePPNampTruncInspiral</td>
<td>433</td>
</tr>
<tr>
<td>LALGeneratePPNInspiral</td>
<td>429</td>
</tr>
<tr>
<td>LALGenerateSpinOrbitCW</td>
<td>447</td>
</tr>
<tr>
<td>LALGenerateTaylorCW</td>
<td>140</td>
</tr>
<tr>
<td>LALGenerateTaylorCW</td>
<td>140</td>
</tr>
<tr>
<td>LALGeographicToEquatorial</td>
<td>461</td>
</tr>
<tr>
<td>LALGEOPsd()</td>
<td>703</td>
</tr>
<tr>
<td>LALGetApproxFromString</td>
<td>424</td>
</tr>
<tr>
<td>LALGetClusters</td>
<td>494</td>
</tr>
<tr>
<td>LALGetEarthTimes</td>
<td>769</td>
</tr>
<tr>
<td>LALGetInspiralParams</td>
<td>434</td>
</tr>
<tr>
<td>LALGetOrderFromString</td>
<td>424</td>
</tr>
<tr>
<td>LALGMST1()</td>
<td>79</td>
</tr>
<tr>
<td>LALGPISisPlayground()</td>
<td>91</td>
</tr>
<tr>
<td>LALGPSTimeNow</td>
<td>90</td>
</tr>
<tr>
<td>LALGPSstoFloat</td>
<td>86</td>
</tr>
<tr>
<td>LALGPStoGMST1</td>
<td>79</td>
</tr>
<tr>
<td>LALGPSstoINT8</td>
<td>87</td>
</tr>
<tr>
<td>LALGPSstoMST1</td>
<td>79</td>
</tr>
<tr>
<td>LALGPSstoUTC()</td>
<td>84</td>
</tr>
<tr>
<td>LALGSLErrorHandler</td>
<td>57</td>
</tr>
<tr>
<td>LALHarmonicFinder</td>
<td>261</td>
</tr>
<tr>
<td>LALHorizonToSystem</td>
<td>404</td>
</tr>
<tr>
<td>LALI2CreateArray</td>
<td>106</td>
</tr>
<tr>
<td>LALI2CreateArray</td>
<td>106</td>
</tr>
<tr>
<td>LALI2CreateGrid</td>
<td>106</td>
</tr>
<tr>
<td>LALI2CreateVector</td>
<td>106</td>
</tr>
<tr>
<td>LALI2CreateVectorSequence</td>
<td>106</td>
</tr>
<tr>
<td>LALI2DestroyArray</td>
<td>106</td>
</tr>
<tr>
<td>LALI2DestroyGrid</td>
<td>106</td>
</tr>
<tr>
<td>LALI2DestroyVector</td>
<td>106</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>--------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>LALI2DestroyVectorSequence()</td>
<td>106</td>
</tr>
<tr>
<td>LALI2MatrixAdd()</td>
<td>369</td>
</tr>
<tr>
<td>LALI2MatrixMultiply()</td>
<td>369</td>
</tr>
<tr>
<td>LALI2MatrixTranspose()</td>
<td>369</td>
</tr>
<tr>
<td>LALI2PrintFrequencySeries()</td>
<td>928</td>
</tr>
<tr>
<td>LALI2PrintTimeSeries()</td>
<td>926</td>
</tr>
<tr>
<td>LALI2PrintVector()</td>
<td>923</td>
</tr>
<tr>
<td>LALI2ReadFrequencySeries()</td>
<td>945</td>
</tr>
<tr>
<td>LALI2ReadGrid()</td>
<td>949</td>
</tr>
<tr>
<td>LALI2ReadTimeArraySeries()</td>
<td>945</td>
</tr>
<tr>
<td>LALI2ReadTimeSeries()</td>
<td>945</td>
</tr>
<tr>
<td>LALI2ReadTimeVectorSeries()</td>
<td>945</td>
</tr>
<tr>
<td>LALI2ReadVector()</td>
<td>939</td>
</tr>
<tr>
<td>LALI2ReadVectorSequence()</td>
<td>941</td>
</tr>
<tr>
<td>LALI2WriteFSeries()</td>
<td>956</td>
</tr>
<tr>
<td>LALI2WriteGrid()</td>
<td>958</td>
</tr>
<tr>
<td>LALI2WriteTArraySeries()</td>
<td>956</td>
</tr>
<tr>
<td>LALI2WriteTSeries()</td>
<td>956</td>
</tr>
<tr>
<td>LALI2WriteTVectorSeries()</td>
<td>956</td>
</tr>
<tr>
<td>LALI4CreateArray()</td>
<td>99</td>
</tr>
<tr>
<td>LALI4CreateGrid()</td>
<td>65</td>
</tr>
<tr>
<td>LALI4CreateVector()</td>
<td>97</td>
</tr>
<tr>
<td>LALI4CreateVectorSequence()</td>
<td>106</td>
</tr>
<tr>
<td>LALI4DestroyArray()</td>
<td>99</td>
</tr>
<tr>
<td>LALI4DestroyGrid()</td>
<td>65</td>
</tr>
<tr>
<td>LALI4DestroyVector()</td>
<td>97</td>
</tr>
<tr>
<td>LALI4DestroyVectorSequence()</td>
<td>106</td>
</tr>
<tr>
<td>LALI4MatrixAdd()</td>
<td>369</td>
</tr>
<tr>
<td>LALI4MatrixMultiply()</td>
<td>369</td>
</tr>
<tr>
<td>LALI4MatrixTranspose()</td>
<td>369</td>
</tr>
<tr>
<td>LALI4PrintFrequencySeries()</td>
<td>928</td>
</tr>
<tr>
<td>LALI4PrintTimeSeries()</td>
<td>926</td>
</tr>
<tr>
<td>LALI4PrintVector()</td>
<td>923</td>
</tr>
<tr>
<td>LALI4ReadFrequencySeries()</td>
<td>945</td>
</tr>
<tr>
<td>LALI4ReadGrid()</td>
<td>949</td>
</tr>
<tr>
<td>LALI4ReadTimeArraySeries()</td>
<td>945</td>
</tr>
<tr>
<td>LALI4ReadTimeSeries()</td>
<td>945</td>
</tr>
<tr>
<td>LALI4ReadTimeVectorSeries()</td>
<td>945</td>
</tr>
<tr>
<td>LALI4ReadVector()</td>
<td>939</td>
</tr>
<tr>
<td>LALI4ReadVectorSequence()</td>
<td>941</td>
</tr>
<tr>
<td>LALI4WriteFSeries()</td>
<td>956</td>
</tr>
<tr>
<td>LALI4WriteGrid()</td>
<td>958</td>
</tr>
<tr>
<td>LALI4WriteTArraySeries()</td>
<td>956</td>
</tr>
<tr>
<td>LALI4WriteTSeries()</td>
<td>956</td>
</tr>
<tr>
<td>LALI4WriteTVectorSeries()</td>
<td>956</td>
</tr>
<tr>
<td>LALI8CreateArray()</td>
<td>99</td>
</tr>
<tr>
<td>LALI8CreateGrid()</td>
<td>65</td>
</tr>
<tr>
<td>LALI8CreateVector()</td>
<td>97</td>
</tr>
<tr>
<td>LALI8CreateVectorSequence()</td>
<td>106</td>
</tr>
<tr>
<td>LALI8DestroyArray()</td>
<td>99</td>
</tr>
<tr>
<td>LALI8DestroyGrid()</td>
<td>65</td>
</tr>
<tr>
<td>LALI8DestroyVector()</td>
<td>97</td>
</tr>
<tr>
<td>LALI8DestroyVectorSequence()</td>
<td>106</td>
</tr>
<tr>
<td>LALI8MatrixAdd()</td>
<td>369</td>
</tr>
<tr>
<td>LALI8MatrixMultiply()</td>
<td>369</td>
</tr>
<tr>
<td>LALI8MatrixTranspose()</td>
<td>369</td>
</tr>
<tr>
<td>LALI8PrintFrequencySeries()</td>
<td>928</td>
</tr>
<tr>
<td>LALI8PrintTimeSeries()</td>
<td>926</td>
</tr>
<tr>
<td>LALI8PrintVector()</td>
<td>923</td>
</tr>
<tr>
<td>LALI8ReadFrequencySeries()</td>
<td>945</td>
</tr>
<tr>
<td>LALI8ReadGrid()</td>
<td>949</td>
</tr>
<tr>
<td>LALI8ReadSequence()</td>
<td>943</td>
</tr>
<tr>
<td>LALI8ReadTimeArraySeries()</td>
<td>945</td>
</tr>
<tr>
<td>LALI8ReadTimeSeries()</td>
<td>945</td>
</tr>
<tr>
<td>LALI8ReadTimeVectorSeries()</td>
<td>945</td>
</tr>
<tr>
<td>LALI8ReadVector()</td>
<td>939</td>
</tr>
<tr>
<td>LALI8ReadVectorSequence()</td>
<td>941</td>
</tr>
<tr>
<td>LALI8WriteFSeries()</td>
<td>956</td>
</tr>
<tr>
<td>LALI8WriteGrid()</td>
<td>958</td>
</tr>
<tr>
<td>LALI8WriteTArraySeries()</td>
<td>956</td>
</tr>
<tr>
<td>LALI8WriteTSeries()</td>
<td>956</td>
</tr>
<tr>
<td>LALI8WriteTVectorSeries()</td>
<td>956</td>
</tr>
<tr>
<td>LALIfoCountSingleInspiral()</td>
<td>164</td>
</tr>
<tr>
<td>LALIfoCutSingleInspiral()</td>
<td>164</td>
</tr>
<tr>
<td>LALIfoScanSearchSummary()</td>
<td>160</td>
</tr>
<tr>
<td>LALIfoScanSummValue()</td>
<td>160</td>
</tr>
<tr>
<td>LALIIRFilterREAL4()</td>
<td>305</td>
</tr>
<tr>
<td>LALIIRFilterREAL8Vector()</td>
<td>306</td>
</tr>
<tr>
<td>LALIIRFilterREAL8VectorR()</td>
<td>307</td>
</tr>
<tr>
<td>LALIIRFilterREAL8VectorR5()</td>
<td>307</td>
</tr>
<tr>
<td>LALInfo()</td>
<td>31</td>
</tr>
<tr>
<td>LALInitBarycenter()</td>
<td>961</td>
</tr>
<tr>
<td>LALInitCList()</td>
<td>500</td>
</tr>
<tr>
<td>LALInitializeDataSegmentVector()</td>
<td>594</td>
</tr>
<tr>
<td>LALInitializeExchange()</td>
<td>1022</td>
</tr>
<tr>
<td>LALInitStatus()</td>
<td>55</td>
</tr>
<tr>
<td>LALInsidePolygon()</td>
<td>564</td>
</tr>
<tr>
<td>LALInspiralBankUtils()</td>
<td>539</td>
</tr>
<tr>
<td>LALInspiralBCVCutBank()</td>
<td>537</td>
</tr>
<tr>
<td>LALInspiralComputeMetric()</td>
<td>544</td>
</tr>
<tr>
<td>LALInspiralComputeParams()</td>
<td>553</td>
</tr>
<tr>
<td>LALInspiralCreateBCVBank()</td>
<td>556</td>
</tr>
<tr>
<td>LALInspiralCreateCoarseBank()</td>
<td>532</td>
</tr>
<tr>
<td>LALInspiralCreateFineBank()</td>
<td>542</td>
</tr>
<tr>
<td>LALInspiralCreateFlatBank()</td>
<td>535</td>
</tr>
<tr>
<td>LALInspiralDerivatives()</td>
<td>675</td>
</tr>
<tr>
<td>LALInspiralDistanceCutCleaning()</td>
<td>169</td>
</tr>
<tr>
<td>LALInspiralHybridHexagonalBank()</td>
<td>565</td>
</tr>
<tr>
<td>LALInspiralITStructureHelp()</td>
<td>690</td>
</tr>
<tr>
<td>LALInspiralITStructureParseParameters,</td>
<td>690</td>
</tr>
<tr>
<td>LALInspiralITStructurePrint,</td>
<td>690</td>
</tr>
<tr>
<td>LALInspiralITStructureSetDefault,</td>
<td>690</td>
</tr>
<tr>
<td>LALInspiralLongestTemplateInBank()</td>
<td>548</td>
</tr>
<tr>
<td>LALInspiralMoments()</td>
<td>549</td>
</tr>
<tr>
<td>LALInspiralMomentsIntegrand()</td>
<td>550</td>
</tr>
<tr>
<td>LALInspiralNextTemplate()</td>
<td>552</td>
</tr>
<tr>
<td>LALInspiralParameterCalc()</td>
<td>650</td>
</tr>
<tr>
<td>LALInspiralParseParametersForInjection,</td>
<td>690</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>LALInspiralRestrictedAmplitude()</td>
<td>632</td>
</tr>
<tr>
<td>LALInspiralRestrictedInit()</td>
<td>652</td>
</tr>
<tr>
<td>LALInspiralSetParams()</td>
<td>562</td>
</tr>
<tr>
<td>LALInspiralSetSearchParams()</td>
<td>551</td>
</tr>
<tr>
<td>LALInspiralSetup()</td>
<td>656</td>
</tr>
<tr>
<td>LALInspiralSetSearchLimits()</td>
<td>651</td>
</tr>
<tr>
<td>LALInspiralSiteTimeAndDist()</td>
<td>657</td>
</tr>
<tr>
<td>LALInspiralStationaryPhaseApprox()</td>
<td>172</td>
</tr>
<tr>
<td>LALInspiralStationaryPhaseApprox2()</td>
<td>172</td>
</tr>
<tr>
<td>LALInspiralUpdateParams()</td>
<td>562</td>
</tr>
<tr>
<td>LALInspiralValidParams()</td>
<td>555</td>
</tr>
<tr>
<td>LALInspiralVelocity()</td>
<td>676</td>
</tr>
<tr>
<td>LALInspiralWave()</td>
<td>659</td>
</tr>
<tr>
<td>LALInspiralWaveLength()</td>
<td>653</td>
</tr>
<tr>
<td>LALInspiralWaveOverlap()</td>
<td>73</td>
</tr>
<tr>
<td>LALInspiralWaveTemplates()</td>
<td>659</td>
</tr>
<tr>
<td>LALINT8NanoSecIsPlayground()</td>
<td>91</td>
</tr>
<tr>
<td>LALINT8toGPS()</td>
<td>87</td>
</tr>
<tr>
<td>LALInterpolateRangeGrid()</td>
<td>797</td>
</tr>
<tr>
<td>LALInterpolateRangePolygon()</td>
<td>797</td>
</tr>
<tr>
<td>LALIntervalToFloat()</td>
<td>86</td>
</tr>
<tr>
<td>LALInverse3()</td>
<td>561</td>
</tr>
<tr>
<td>LALJulianDate()</td>
<td>76</td>
</tr>
<tr>
<td>LALJulianDay()</td>
<td>76</td>
</tr>
<tr>
<td>LALLALBCVSpinWaveform()</td>
<td>670</td>
</tr>
<tr>
<td>LALLALBCVWaveform()</td>
<td>670</td>
</tr>
<tr>
<td>LALLeapSecs()</td>
<td>84</td>
</tr>
<tr>
<td>LALLIGOIPsd()</td>
<td>706</td>
</tr>
<tr>
<td>LALLMSI()</td>
<td>79</td>
</tr>
<tr>
<td>LALMallocLong()</td>
<td>47</td>
</tr>
<tr>
<td>LALMallocShort()</td>
<td>47</td>
</tr>
<tr>
<td>LALMath3DPlot()</td>
<td>985</td>
</tr>
<tr>
<td>LALMathNDPlot()</td>
<td>987</td>
</tr>
<tr>
<td>LALMatrixTransform()</td>
<td>559</td>
</tr>
<tr>
<td>LALMergeClusterLists()</td>
<td>496</td>
</tr>
<tr>
<td>LALMersenneRandomVector()</td>
<td>323</td>
</tr>
<tr>
<td>LALModJulianDate()</td>
<td>76</td>
</tr>
<tr>
<td>LALMPIRecvCHAR()</td>
<td>1018</td>
</tr>
<tr>
<td>LALMPIRecvCHARVector()</td>
<td>1018</td>
</tr>
<tr>
<td>LALMPIRecvCOMPLEX16()</td>
<td>1018</td>
</tr>
<tr>
<td>LALMPIRecvCOMPLEX16FrequencySeries()</td>
<td>1018</td>
</tr>
<tr>
<td>LALMPIRecvCOMPLEX16TimeSeries()</td>
<td>1018</td>
</tr>
<tr>
<td>LALMPIRecvCOMPLEX16Vector()</td>
<td>1018</td>
</tr>
<tr>
<td>LALMPIRecvCOMPLEX8()</td>
<td>1018</td>
</tr>
<tr>
<td>LALMPIRecvCOMPLEX8FrequencySeries()</td>
<td>1018</td>
</tr>
<tr>
<td>LALMPIRecvCOMPLEX8TimeSeries()</td>
<td>1018</td>
</tr>
<tr>
<td>LALMPIRecvCOMPLEX8Vector()</td>
<td>1018</td>
</tr>
<tr>
<td>LALMPIRecvINT2()</td>
<td>1018</td>
</tr>
<tr>
<td>LALMPIRecvINT2FrequencySeries()</td>
<td>1018</td>
</tr>
<tr>
<td>LALMPIRecvINT2TimeSeries()</td>
<td>1018</td>
</tr>
<tr>
<td>LALMPIRecvINT2Vector()</td>
<td>1018</td>
</tr>
<tr>
<td>LALMPIRecvINT4()</td>
<td>1018</td>
</tr>
<tr>
<td>LALMPIRecvINT4FrequencySeries()</td>
<td>1018</td>
</tr>
<tr>
<td>LALMPIRecvINT4TimeSeries()</td>
<td>1018</td>
</tr>
<tr>
<td>LALMPIRecvINT4Vector()</td>
<td>1018</td>
</tr>
<tr>
<td>LALMPIRecvINT8()</td>
<td>1018</td>
</tr>
<tr>
<td>LALMPIRecvINT8FrequencySeries()</td>
<td>1018</td>
</tr>
<tr>
<td>LALMPIRecvINT8TimeSeries()</td>
<td>1018</td>
</tr>
<tr>
<td>LALMPIRecvINT8Vector()</td>
<td>1018</td>
</tr>
<tr>
<td>LALMPIRecvMsg()</td>
<td>1018</td>
</tr>
<tr>
<td>LALMPIrecvREAL4()</td>
<td>1018</td>
</tr>
<tr>
<td>LALMPIrecvREAL8()</td>
<td>1018</td>
</tr>
<tr>
<td>LALMPIrecvREAL8FrequencySeries()</td>
<td>1018</td>
</tr>
<tr>
<td>LALMPIrecvREAL8TimeSeries()</td>
<td>1018</td>
</tr>
<tr>
<td>LALMPIrecvREAL8Vector()</td>
<td>1018</td>
</tr>
<tr>
<td>LALMPIrecvUINT2()</td>
<td>1018</td>
</tr>
<tr>
<td>LALMPIrecvUINT4()</td>
<td>1018</td>
</tr>
<tr>
<td>LALMPIrecvUINT8()</td>
<td>1018</td>
</tr>
<tr>
<td>LALMPIrecvUINT8FrequencySeries()</td>
<td>1018</td>
</tr>
<tr>
<td>LALMPIrecvUINT8TimeSeries()</td>
<td>1018</td>
</tr>
<tr>
<td>LALMPIrecvUINT8Vector()</td>
<td>1018</td>
</tr>
</tbody>
</table>
XLALUnitRaiseRAT4, 118
XLALUnitSqrt, 118
XLALUnitSquare, 118

Macro
BEGINFAIL(), 30
ENDFAIL(), 30
LALCalloc(), 47
LALMalloc(), 47
LALRealloc(), 47
TRY(), 29

Power Spectrum, 390
Spectral Estimation, 390
Taper, 390
Type
AddVectorsIn, 697
AverageSpectrumParams, 280
AvgSpecMethod, 280
BarycenterInput, 778
BurstTmpltBankCInput, 181
CalibrationFunctions, 132
CalibrationRecord, 132
CalibrationType, 132
CalibrationUpdateParams, 132
CellEvolution, 551
CellList, 551
ChannelType, 996
Clustering, 898
CoarseFitInput, 850
CoarseFitOutput, 850
CoarseFitParams, 851
CoarseHeterodyneInput, 846
CoarseHeterodyneOutput, 846
CoarseHeterodyneParams, 846
CoherentGW, 108
CoincInspiralTable, 142
CoincRingdownTable, 142
CompanionNode, 855
COMPLEX16Grid, 64
COMPLEX8Grid, 64
COMPLEX8TimeFrequencyPlane, 476
COMPLEX8WithUnits, 868
ComplexDFTParams, 476
ComputeMoments, 531
ConvertSkyParams, 457
CoordinateSpace, 531
CoordinateSystem, 457
CreateArraySequenceIn, 105
CreateVectorSequenceIn, 105
CZeroPadAndFFTParameters, 869
DataSegmentVector, 579
DemodPar, 325
Detector, 697
DetectorResponse, 408
DetTimeAndASource, 92
DirichletParameters, 531
EarthPosition, 457
EarthState, 778
EmissionTime, 778
EphemerasData, 777
EphemerasFilenames, 777
EtaT02In, 649
EtaT04In, 649
ExchParams, 1014
expnCoeffs, 649
expnFunc, 649
FindChirpChisqInput, 611
FindChirpChisqParams, 611
FindChirpDataParams, 587
FindChirpFilterInput, 591
FindChirpFilterOutputVetoParams, 589
FindChirpFilterParams, 590
FindChirpInitParams, 586
FindChirpSegment, 581
FindChirpSegmentVector, 582
FindChirpStandardCandle, 579
FindChirpSubBank, 580
FindChirpSubBankData, 592
FindChirpTemplate, 583
FindChirpTmpltParams, 588
FineHeterodyneInput, 846
FineHeterodyneOutput, 846
FineHeterodyneParams, 846
FlatMeshParamStruc, 782
FoldAmplitudesInput, 817
FoldAmplitudesParams, 847
FrChanIn, 996
FreqCut, 551
FrequencySamplingParams, 355
FrFileInfo, 996
FrOutPar, 996
FrPos, 996
FrStat, 992
FrState, 996
FrStream, 996
GalacticInspiralParamStruc, 428
Generation, 531
GridSpacing, 531
HexaGridParam, 531
HorizontalTFTransformIn, 470
InitExchParams, 1014
InputMasses, 649
InsidePolygon, 531
InspiralACSTParamsH, 649
InspiralBankMassRange, 531
InspiralBankParams, 531
InspiralCell, 531
InspiralCoarseBankIn, 531
InspiralDerivativesIn, 649
InspiralFineBankIn, 531
InspiralMetric, 531
InspiralMomentsEtc, 531
InspiralMomentsEtcBCV, 531
InspiralMomentsIn, 531
TwoDetsTimeAndASource, 92
TwoDMeshNode, 790
TwoDMeshParamStruc, 790
TwoDMeshPlotStruc, 798
UINT2Grid, 64
UINT4Grid, 94
UINT8Grid, 93
VerticalTFTransformIn, 476

Variable
lalAbortHook, 53
lalCachedDetectors[], 122, 125
lalDebugLevel, 27, 32
lalGSLGlobalStatusPtr, 57
lalGSLPthreadMutex, 57
lalRaiseHook, 53