

USERS MANUAL

GRASP: a data analysis package for gravitational wave detection

Bruce Allen*
Department of Physics
University of Wisconsin - Milwaukee
PO Box 413
Milwaukee WI 53201, USA

May 19, 2000

Contributors:

Warren Anderson, R. Balasubramanian, Kent Blackburn, Patrick Brady, Jim Brau, Jolien Creighton, Teviet Creighton, Steve Drasco, Serge Droz, Eanna Flanagan, Wensheng Hua, Scott Hughes, Dustin Laurence, Adrian Ottewill, Ben Owen, Jorge Pullin, Joseph Romano, and Alan Wiseman.

Abstract

GRASP (**G**ravitational **R**adiation **A**nalysis & **S**imulation **P**ackage) is a public-domain software tool-kit designed for analysis and simulation of data from gravitational wave detectors. This users manual describes the use and features of this package. Note: an up-to-date version of this manual may be obtained at: <http://www.lsc-group.phys.uwm.edu/~ballen/grasp-distribution/>. The software package is also available from this site.

Copyright 1999 ©Bruce Allen

GRASP RELEASE 1.9.8

*ballen@dirac.phys.uwm.edu

Contents

1	ACKNOWLEDGEMENTS	10
2	Introduction	11
2.1	The Purpose of GRASP	11
2.2	Printing/Reading the Manual	11
2.3	Quick Start	11
2.4	A few words about data formats	12
2.5	GRASP Hardware & Software Requirements	13
2.6	GRASP Installation	14
2.6.1	GRASP File Structure	14
2.6.2	Accessing <i>Numerical Recipes in C</i> libraries	16
2.6.3	Accessing MPI and MPE libraries	17
2.6.4	Accessing <i>MESA</i> libraries	17
2.6.5	Accessing <i>CLAPACK</i> libraries	18
2.6.6	Accessing <i>FRAME</i> libraries	19
2.6.7	Real-time 40-meter analysis	19
2.6.8	The Matlab Interface	19
2.6.9	Making the GRASP binaries and libraries	19
2.6.10	Stupid Pet Tricks	23
2.7	Conventions used in this manual	24
2.8	How to add your contributions to future GRASP releases.	25
2.9	How to use the GRASP library from ROOT.	28
3	GRASP Routines: Reading/using Caltech 40-meter prototype data	30
3.1	The data format	31
3.2	Function: <code>read_block()</code>	35
3.3	Example: <code>reader</code> program	37
3.4	Function: <code>find_locked()</code>	38
3.5	Example: <code>locklist</code> program	39
3.6	Function: <code>get_data()</code>	40
3.7	Example: <code>gwoutput</code> program	41
3.8	Example: <code>animate</code> program	42
3.9	Function: <code>read_sweptsine()</code>	46
3.10	Function: <code>calibrate()</code>	49
3.11	Example: <code>print_ss</code> program	50
3.12	Function: <code>normalize_gw()</code>	51
3.13	Example: <code>power_spectrum</code> program	53
3.14	Example: <code>calibrate</code> program	56
3.15	Example: <code>transfer</code> program	59
3.16	Example: <code>diag</code> program	63
4	GRASP Routines: Reading/using FRAME format data	66
4.1	Time-stamps in the November 1994 data-set	68
4.2	Function: <code>fget_ch()</code>	69
4.3	Function: <code>framefiles()</code>	74
4.4	Example: <code>locklistF</code> program	75

Section	TABLE OF CONTENTS	Page
0		3
4.5	Example: gwoutputF program	78
4.6	Example: animateF program	80
4.7	Swept-sine calibration information	85
4.8	Function: GRcalibrate()	87
4.9	Example: print_ssF program	88
4.10	Function: GRnormalize()	90
4.11	Example: power_spectrumF program	92
4.12	Example: calibrateF program	96
4.13	Example: transferF program	100
4.14	Example: diagF program	104
4.15	Example: seismicF program	107
5	GRASP Routines: Signal-to-noise enhancement techniques	108
5.1	Signal-to-noise enhancement by environmental cross-correlation	108
5.2	Outline	110
5.3	Function: calc_rho()	112
5.4	Function chan_clean()	113
5.5	Example: Correlations in data from the 40m interferometer	115
5.6	Example: corr_init	118
5.7	Example: env_corr	122
6	GRASP Routines: Gravitational Radiation from Binary Inspiral	130
6.1	Chirp generation routines	131
6.2	Function: phase_frequency()	132
6.3	Example: phase_evoltn program	134
6.4	Detailed explanation of phase_frequency() routine	137
6.5	Function: chirp_filters()	139
6.6	Detailed explanation of chirp_filters() routine	141
6.7	Example: filters program	143
6.8	Practical Suggestion for Setting Up a Large Bank of Filters:	145
6.9	Additional contributions to the phase and frequency of the chirp	146
6.9.1	Spin Effects	146
6.9.2	2.5 Post-Newtonian corrections to the inspiral chirp	149
6.10	Function: make_filters()	151
6.11	Stationary phase approximation to binary inspiral chirps	152
6.12	Function: sp_filters()	154
6.13	Example: compare_chirps program	155
6.14	Wiener (optimal) filtering	157
6.15	Comparison of signal detectability for single-phase and two-phase searches	162
6.16	Function: correlate()	164
6.17	Function: avg_inv_spec()	166
6.18	Function: orthonormalize()	167
6.19	Dirty details of optimal filtering: wraparound and windowing	168
6.20	Function: find_chirp()	174
6.21	Function: freq_inject_chirp()	175
6.22	Function: time_inject_chirp()	176
6.23	Vetoing techniques (time domain outlier test)	177
6.24	Vetoing techniques (r^2 time/frequency test)	180

Section	TABLE OF CONTENTS	Page
0		4
6.25	How does the r^2 test work ?	187
6.26	Function: splitup()	189
6.27	Function: splitup_freq()	190
6.28	Function: splitup_freq2()	191
6.29	Function: splitup_freq3()	192
6.30	Example: optimal program	193
6.31	Some output from the optimal program	199
6.32	The effective distance to which a source can be seen	204
6.33	Function: inspiral_dist()	206
6.34	Function: merger_dist()	207
6.35	Example: compute_dist program	208
7	GRASP Routines: Waveforms from perturbation theory	211
7.1	The waveform	211
7.2	Chirp generation for test mass signals	213
7.3	Function: testmass_chirp()	214
7.4	Function: calculate_testmass_phase()	216
7.5	Function: Get_Duration()	217
7.6	Function: Get_Fmax()	218
7.7	Function: ReadData()	219
7.8	Function: Clean_Up_Memory()	220
7.9	Function: Set_Up_Data()	221
7.10	Function: minustwoSIm()	222
7.11	Function: read_modes()	223
7.12	Function: read_real_data_file()	224
7.13	Function: integrate_function()	225
7.14	Function: integrateODE()	226
7.15	Errors	227
7.16	Example: tmwave program	228
7.17	Example: lorenz program	231
7.18	Example: plot_ambig program	233
8	GRASP Routines: Black hole ringdown	238
8.1	Quasinormal modes of black holes	239
8.2	Function: qn_eigenvalues()	241
8.3	Example: eigenvalues program	242
8.4	Function: sw_spheroid()	244
8.5	Example: spherical program	245
8.6	Example: spheroid program	248
8.7	Function: qn_ring()	250
8.8	Example: ringdown program	251
8.9	Function: qn_qring()	253
8.10	Function: qn_filter()	255
8.11	Function: qn_normalize()	256
8.12	Function: find_ring()	257
8.13	Function: qn_inject()	258
8.14	Vetoing techniques for ringdown waveforms	259
8.15	Example: qn_optimal program	260

Section	TABLE OF CONTENTS	Page
0		5
8.16	Structure: struct qnTemplate	264
8.17	Structure: struct qnScope	265
8.18	Function: qn_template_grid()	266
8.19	The close-limit approximation and numerical simulations	268
8.20	Inspiralling collisions	270
8.21	Example: ring-corr program	272
9	GRASP Routines: Template Bank Generation & Searching	279
9.1	Structure: struct Template	279
9.2	Structure: struct Scope	282
9.3	Function: tau_of_mass()	283
9.4	Function: m_and_eta()	284
9.5	Function: template_area()	286
9.6	Example: area program	287
9.7	The match between two templates	288
9.8	Function: compute_match()	289
9.9	Function: match_parab()	290
9.10	Function: match_cubic()	293
9.11	Example: match_fit program	294
9.12	Structure: struct cubic_grid	296
9.13	Function: generate_cubic	297
9.14	Function: regenerate_cubic	298
9.15	Function: read_cubic	299
9.16	Function: get_cubic	300
9.17	Function: free_cubic	301
9.18	Function: transform_cubic	302
9.19	Example: make_grid program	303
9.20	Example: read_grid program	305
9.21	Function: template_grid()	307
9.22	Function: plot_template()	310
9.23	Example: template program	312
9.24	Example: multifilter program	313
9.25	Optimization and computation-speed considerations	326
9.26	Template Placement	328
9.27	Structure: struct tile	330
9.28	Function: tiling_2d	331
9.29	Function: plot_list	333
9.30	Constants in tiling_2d.c	334
9.31	Structure: struct chirp_space	335
9.32	Structure: struct chirp_template	336
9.33	Function: set_chirp_space	338
9.34	Function: chirp_metric	339
9.35	Function: get_chirp_boundary	340
9.36	Function: get_chirp_grid	341
9.37	Function: get_chirp_templates	342
9.38	Function: plot_chirp_templates	343
9.39	Example: make_mesh program	344

10 GRASP Routines: Time-Frequency Methods	347
10.1 Construction of the TF map	347
10.1.1 Wigner-Ville Distribution	347
10.1.2 Windowed Fourier transform	349
10.1.3 Choi-William's distribution	349
10.2 Steger's Line Detection Routines	349
10.3 Structure: struct struct_tfparam	350
10.4 Function: time_freq_map()	351
10.5 Function compute_scalefactor()	351
10.6 Function rescale()	352
10.7 Function normalize_picture()	352
10.8 Function gen_quasiperiodic_signal()	352
10.9 Function: ppmprint()	354
10.10Function: pgmprint()	354
10.11Function: plottf()	354
10.12Function: get_line_lens()	355
10.13Function: get_lines()	355
10.14Example: tfmain program	356
10.14.1 Environment variables used by tfmain	356
10.14.2 File: tfmain.h	357
10.14.3 File: tfmain.c	359
10.14.4 File: tf_get_data.c	364
10.14.5 File: tfmain.in	365
10.14.6 File: combine.c	367
10.14.7 File: readertf.c	367
11 GRASP Routines: Stochastic background detection	368
11.1 Data File: detectors.dat	368
11.2 Function: detector_site()	371
11.3 Comment: noise power spectra for "advanced" LIGO & the Cutler-Flanagan model	373
11.4 Function: noise_power()	374
11.5 Function: whiten()	376
11.6 Function: overlap()	378
11.7 Example: overlap program	379
11.8 Function: get_IFO12()	381
11.9 Function: simulate_noise()	382
11.10Function: simulate_sb()	384
11.11Function: combine_data()	387
11.12Function: monte_carlo()	388
11.13Example: monte_carlo program	390
11.14Function: test_data12()	397
11.15Function: extract_noise()	398
11.16Function: extract_signal()	400
11.17Function: optimal_filter()	402
11.18Example: optimal_filter program	404
11.19Discussion: Theoretical signal-to-noise ratio for the stochastic background	408
11.20Function: calculate_var()	410
11.21Example: snr program	412

Section	TABLE OF CONTENTS	Page
0		7
11.22	Example: omega_min program	414
11.23	Function: analyze()	417
11.24	Function: prelim_stats()	419
11.25	Function: statistics()	421
11.26	Example: simulation program	422
11.27	Some output from the simulation program	425
12	Galactic Modelling	428
12.1	Function: local_sidereal_time()	429
12.2	Example: caltech_lst program	430
12.3	Function: galactic_to_equatorial()	432
12.4	Example: galactic2equatorial program	433
12.5	Function: equatorial_to_horizon()	435
12.6	Function: beam_pattern()	436
12.7	Function: mc_chirp()	437
12.8	Example: inject program	438
13	Binary Inspiral Search on November 1994 Data	442
13.1	The Statistical Theory of Reception	443
13.1.1	Maximum Likelihood Receiver	443
13.1.2	A Receiver for a Known Signal	443
13.1.3	A Receiver for a Signal of Unknown Phase	444
13.1.4	Reception of a Signal with Unknown Arrival Time	445
13.1.5	Reception of a Signal with Additional Unknown Parameters	446
13.2	Details of Normalization	447
13.3	Function: strain_spec()	448
13.4	Function: corr_coef()	449
13.5	Function: receiver1()	450
13.6	Function: receiver2()	451
13.7	Example: binary_search program	453
13.7.1	Environment variables used by binary_search	455
13.7.2	File: binary_params.h	456
13.7.3	File: binary_search.c (MPI multiprocessor code)	459
13.7.4	File: binary_get_data.c	473
13.8	Scripts for running binary_search	493
13.9	Example: binary_reader program	494
13.10	Identification of Spurious Events	497
14	GRASP Routines: Supernovae and other transient sources	500
14.1	Centrifugal Hang-up of Core Collapse Supernovae	501
14.2	Structure: LS_physical_constants	503
14.3	Function: LS_freq_deriv()	504
14.4	Function: LS_phas_and_freq()	505
14.5	Function: LS_waveform()	506
14.6	Example: LS_filter program	507
15	GRASP Routines: Periodic and quasi-periodic sources	508

16 GRASP Routines: General purpose utilities	509
16.1 GRASP Error Handling	510
16.1.1 Reporting Errors In GRASP Code	510
16.1.2 How GRASP Error Reports Are Handled	511
16.1.3 Customizing The Default Handlers	511
16.1.4 Writing Custom Error Handlers	512
16.1.5 Functions: GR_start_error(), GR_report_error(), GR_end_error()	514
16.2 Function: grasp_open()	515
16.3 Function: avg_spec()	516
16.4 Function: binshort()	518
16.5 Function: is_gaussian()	519
16.6 Function: clear()	521
16.7 Function: product()	522
16.8 Function: productc()	523
16.9 Function: ratio()	524
16.10Function: reciprocal()	525
16.11Function: graph()	526
16.12Function: graph_double()	527
16.13Function: graph_short()	528
16.14Function: sgraph()	529
16.15Function: audio()	530
16.16Example: makesounds program	531
16.17Function: sound()	533
16.18Example: translate	534
16.19Multi-taper methods for spectral analysis	544
16.20Function: slepian_tapers()	545
16.21Function: multitaper_spectrum()	547
16.22Function: multitaper_cross_spectrum()	549
16.23Structure: struct removed_lines	550
16.24Function: fvalue_cmp()	551
16.25Function: index_cmp()	552
16.26Function: remove_spectral_lines()	553
16.27Example: river	555
16.28Example: ifo_clean	558
16.29Example: tracker	562
16.30Example: trackerF	564
17 Time Standards: UTC, GPS, TAI, and Unix-C times.	570
17.1 Function: utctime()	574
17.2 Function: gpstime()	575
17.3 Example: testutctime	576
18 Matlab Interface: Gravitational Radiation Toolbox	577
18.1 Using GRtool	578
18.2 Functions	581
18.2.1 Function: frextract	582
18.2.2 Function: getfri	582
18.2.3 Function: inspfilt	583

18.2.4	Function: mxAvg_inv_spec	584
18.2.5	Function: mxChirp_filters	584
18.2.6	Function: mxCompute_match	584
18.2.7	Function: mxCorrelate	584
18.2.8	Function: mxDetector_site	584
18.2.9	Function: mxFget_ch	585
18.2.10	Function: mxFind_chirp	586
18.2.11	Function: mxFreq_inject_chirp	586
18.2.12	Function: mxGRcalibrate	586
18.2.13	Function: mxGRnormalize	586
18.2.14	Function: mxM_and_eta	586
18.2.15	Function: mxMake_filters	587
18.2.16	Function: mxMatch_cubic	587
18.2.17	Function: mxOrthonormalize	587
18.2.18	Function: mxPhase_frequency	587
18.2.19	Function: mxSp_filters	587
18.2.20	Function: mxSplitup	588
18.2.21	Function: mxSplitup_freq	588
18.2.22	Function: mxSplitup_freq2	588
18.2.23	Function: mxTau_of_mass	588
18.2.24	Function: mxTemplate_area	588
18.2.25	Function: mxTime_inject_chirp	588
18.2.26	Function: mxUrlopen	589
18.3	Examples	589
18.3.1	Example: print_ssF	589
18.3.2	Example: power_spectrumF	590
18.3.3	Example: phase_evoltn	591
18.3.4	Example: filters	591
18.3.5	Example: area	592
18.3.6	Example: match_fit	592
18.3.7	Example: readfri	593
18.3.8	Example: oneFget	594
18.3.9	Example: twoFget	594

19 REFERENCES

1 ACKNOWLEDGEMENTS

This work has been partially supported by National Science Foundation grants to: the University of Wisconsin - Milwaukee PHY9507740 and PHY9728704, the LIGO project PHY9210038 and the LIGO visitors program PHY9603177. It has also benefited from the contributions of several individuals. Alan Wiseman wrote the chirp generation routines with me in 1995, then entirely re-wrote them for GRASP. Joseph Romano took my (AVS versions of the) stochastic background code, helped to fix a number of the problems, and produced the stochastic background simulation routines contained in GRASP. Sathyaprakash generously provided me with a copy of his routine `grid4.f` which eventually evolved into `template_grid`. Jolien Creighton wrote and documented the section on black hole ringdown, and incorporated the FFTW optimized Fourier transform routines. Teviet Creighton, Ben Owen, Scott Hughes and Patrick Brady contributed essential pieces towards the binary-inspiral search performed on the Caltech 40-meter data from November 1994. Jim Mason was “instrumental” in helping me to understand and document the data format used in the 40-meter prototype experiment. Fred Raab, Bob Spero, Stan Whitcomb, Kent Blackburn and others have contributed many useful ideas and insights about how to understand the data stream from a real interferometer. Kip Thorne and his research group provided or called attention to many of these ideas.

2 Introduction

2.1 The Purpose of GRASP

The analysis and modeling of data from gravitational wave detectors requires specialized numerical techniques. GRASP was developed in collaboration with the Laser Interferometer Gravitational Wave Observatory (LIGO) project in the United States, and contains a collection of software tools for this purpose. The first release of GRASP was in early 1997; since that time many individuals have made extensive contributions.

In order that it be of the most use to the physics community, this package (including all source code) is being released in the public domain. It may be freely used for any purpose, although we do ask that GRASP and its author be acknowledged or referenced in any work or publications to which GRASP made a contribution. If possible this reference should include a link to the GRASP distribution web site:

<http://www.lsc-group.phys.uwm.edu/~ballen/grasp-distribution/>. The citation should specify the *version number* (for example, 1.9.1) of GRASP. In addition, if the code has been modified please state this. We suggest that if GRASP is installed at a site, one person at the institution should be designated as the “responsible party” in charge of the GRASP package.

GRASP is intended for a broad audience, including those users whose main interest is in running simulations and analyzing data, and those users whose main interest is in testing new data analysis techniques or incorporating searches for new types of gravitational wave sources. The GRASP package includes a “cook-book” of documented and tested low-level routines which may be incorporated in user code, and simple example programs illustrating the use of these routines. GRASP also includes a number of high level user applications built from these routines.

We are always interested in extending the capabilities of GRASP. Suggestions for changes or additions, including reports of bugs or corrections, improvements, or extensions to the source code, should be communicated directly to the author.

2.2 Printing/Reading the Manual

The manual is distributed with GRASP in three forms. In the GRASP directory `doc` you can find a Portable Document Format (PDF) file `manual.pdf`, a Postscript file `manual.ps` and a Device Independent file `manual.dvi`. We suggest using the PDF file. Not only is it compact, but all the sections, references, and equations are represented as clickable links. Even the WWW links (URLs) can be clicked on and will fire up your favorite Web browser. You can also easily “zoom-in” on interesting graphs.

If you want a printed copy of the manual, there are two options. We find that the most readable form is “2-up”. You can make a postscript file of this form using the `psnup` utility, available as part of the public-domain package `psutils`. Use the commands:

```
psnup -2 /usr/local/GRASP/doc/manual.ps man2.ps
```

and then print the file that you have just created (`man2.ps`) on a two-sided postscript printer. You’ll end up with four pages of this manual on a single sheet of paper.

If you want a copy of the manual with the color graphs in color rather than gray-scale, we’ve included postscript files containing the color and black-and-white pages separately. Print `doc/manual_color.ps` on a color printer, and `doc/manual_bw.ps` on a black and white printer, and start collating!

2.3 Quick Start

If you hate to read manuals, and you just want to try something, here’s a suggestion. This assumes that the GRASP package has been installed by your local system administrator in a directory accessible to you, such

as `/usr/local/GRASP` and that some 40-meter data (old-format) has also been installed, for example in `/usr/local/GRASP/data`.

If you want to try running a GRASP program, type
`setenv GRASP_DATAPATH /usr/local/GRASP/data/19nov94.3`
to set up a path to the data, then go to the GRASP directory:
`cd /usr/local/GRASP/src/examples/examples_40meter`
and try running one of the executables:

```
./locklist
```

will print out a list of the locked data segments from run 3 on 19 November 1994. A more interesting program to run (in the same directory) is
`./animate | xmgr -pipe`
which will produce an animated display of the IFO output. Note that in order for this to work, you will need to have the `xmgr` graphing program in your path. (Please see the comment about `xmgr` in Section 3.8).

If you only have data that has been distributed in the FRAME format, type
`setenv GRASP_FRAMEPATH /usr/local/GRASP/data/19nov94.3.frame`
to set up a path to the data, then go to the GRASP directory:
`cd /usr/local/GRASP/src/examples/examples_frame`
and try running one of the executables:

```
./locklistF
```

will print out a list of the locked data segments from run 3 on 19 November 1994. A more interesting program to run (in the same directory) is
`./animateF | xmgr -pipe`
which will produce an animated display of the IFO output. Note that in order for this to work, you will need to have the `xmgr` graphing program in your path. (Please see the comment about `xmgr` in Section 3.8).

If you want to try writing some GRASP code, a simple way to start is to copy one of the example programs, and the Makefile, into your personal directory, and edit that:

```
mkdir ~/GRASP
cp /usr/local/GRASP/src/examples/examples_40meter/gwoutput.c ~/GRASP
cp /usr/local/GRASP/src/examples/examples_40meter/Makefile ~/GRASP
cd ~/GRASP
```

Now make editing changes to the file `gwoutput.c`, and when you are done, edit the `Makefile` that you have copied into your home directory. Find the line that reads:

```
all: ... gwoutput ...
```

and delete everything to the right of the colon except `gwoutput` from that line (but leave a space after the colon). Then type:

```
make gwoutput
```

to recompile this program. To run it, simply type:

```
gwoutput.
```

In general, if you want to modify GRASP programs, this is the simplest way to start.

2.4 A few words about data formats

The GRASP package was originally written for analysis of data in the “old” format, which was used in the Caltech 40-meter IFO laboratory prior to 1996. Starting in 1997, the LIGO project, and a number of other gravity-wave detector groups, have adopted the VIRGO FRAME data format. Almost every example in the GRASP package has equivalent programs to read and analyze data in either format. For example `animate` and `animateF` are two versions of the same program. The first reads data in the old format, the second reads data in the FRAME format. We have also included with GRASP a translation program that translates

data from the old format to the new format (see `translate` in Section 16.18).

After careful thought, the LIGO management has decided to only distribute the November 1994 data in the FRAME format, except to a small number of groups (belonging to the *Data Translation Group*) who are responsible for ensuring that the translated data set contains the same information as the original! The initial distributions of GRASP will include both old-format and new-format code. However after a reasonable period of time, the old-format data and code will be removed from the package. So please be aware that the old-format material will be reaching the end of its useful lifetime fairly soon; we do not recommend investing much effort in these.

If you want to develop or work on data analysis algorithms, you will want to have access to this data archive. Because many people contributed to taking this data, and because the LIGO project wants to maintain control of its use and distribution, *this data set is NOT in the public domain*. However, you may request a copy for your use, or for use by your research group. Write to: Director of the LIGO Laboratory, Mail Stop 51-33, California Institute of Technology, Pasadena, CA 91125. The data set is available in `tar` format on two Exabyte 8500c format tapes.

In order to use the data in the FRAME format, you will need to have access to the FRAME libraries. These are available from the VIRGO project; they may be downloaded from the site <http://www.lapp.in2p3.fr/virgo/FrameL/>. The current release of GRASP is compatible with versions of the FRAME library ≤ 3.72 . Contact Benoit Mours mours@lapp.in2p3.fr for further information.

2.5 GRASP Hardware & Software Requirements

GRASP was developed under the Unix (tm) operating system, on a Sun workstation network. The package is written in POSIX/ANSI C, so that GRASP can be compiled and used on any machine with an ANSI C compiler. All operating system calls are POSIX-compliant, which is intended to keep GRASP as portable to different platforms as possible. The main routines could also be linked to user code written in other languages such as Fortran or Pascal; the details of this linking, and the conventions by which Fortran and C (or Pascal and C) routines communicate are implementation dependent, and not discussed here.

Several of the high-level applications in GRASP can be run on parallel computer systems. These can be either dedicated parallel computers (such as the Intel Paragon or IBM SP2 machines) or a network of scientific workstations. The parallel programming in GRASP is implemented with version 1.1 of the Message Passing Interface (MPI) library specification [2]. All major computer system vendors currently support this standard, so GRASP can be easily compiled and used on virtually any parallel machine. In addition, there is a public-domain implementation of MPI called “mpich” [3] which will run MPI-based programs on networks of scientific workstations. This makes it easy to do “super-computing at night” by running GRASP on a network of workstations. Further information on MPI is available from the web site <http://www.mcs.anl.gov/mpi/>. The mpich implementation is available from <http://www.mcs.anl.gov/mpi/mpich/>. By the way, if you don’t have access to parallel machines (or have no interest in parallel computing) don’t worry! The only parallel code in GRASP is found in “top-level” applications; all of the functions in the GRASP library, and most of the examples, can be used without any modifications on a single processor, stand-alone computer.

GRASP makes use of a number of standard numerical techniques. In general, we use version 2.06 of the routines from “*Numerical Recipes in C: the art of scientific computing*” [1]. [Later versions should work OK – please let me know if they don’t.] These routines are widely used in the scientific community. The full source code, examples, and complete documentation are provided in the book, and are also available (for about \$50) in computer readable form. Ordering information and further details are available from <http://www.nr.com/>. These routines are extremely useful and beautifully-documented; if you don’t already have them available for your use, you should!

Certain routines in that use inter-channel correlations to ‘clean’ a signal channel also use CLAPACK numerical linear algebra libraries. These are extremely robust and well tested libraries and are an extremely valuable complement to *Numerical Recipes*. Note that all GRASP programs can be compiled without CLAPACK but that some inter-channel correlation functions will not be available without it. The full source code for these may be downloaded from <http://www.netlib.org/clapack/>.

The time-frequency routines in the GRASP package also come with a function (`plottf()`) to display time frequency-maps on the screen using calls to the MESA graphics library. This library is a GL lookalike and available freely from <http://www.mesa3d.org/>.

In general, output from GRASP is in the form of ASCII text files. We assume that the user has graphing packages available to visualize and interpret this output. Our personal favorite is `xmgr`, available in the public domain from the site <http://plasma-gate.weizmann.ac.il/Xmgr/> which also lists mirror sites in Europe and USA. (Please see the comment about `xmgr` in Section 3.8). In some cases we do output “complete graphs” for `xmgr`. We do also output some data in the form of PostScript (tm) files. Previewers for postscript files are widely available in the public domain (we like GhostView).

2.6 GRASP Installation

As we have just explained, GRASP requires access to *Numerical Recipes in C* libraries and to MPI and MPE libraries and optionally to the CLAPACK libraries. These packages must be installed, and then within GRASP a path to these libraries must be defined. This can be done by editing a single file, and then running a shell script. This section explains each of these steps in detail.

All of the site-specific information is contained in a single file `SiteSpecific` in the top-level directory of GRASP. This file contains a number of variables whose purpose is explained in this section. These variables must be correctly set before GRASP can be used; the definitions contained in `SiteSpecific` (as distributed) are probably *not* appropriate for your system, and will therefore require modification. A number of example `SiteSpecific` files are included in the GRASP distribution, in the directory `Examples_SiteSpecific/`.

2.6.1 GRASP File Structure

The code for GRASP can be installed in a publicly-available directory, for example `/usr/local/GRASP`. (It can also be installed “privately” in a single user’s home directory, if desired.) The name of this top-level directory must be set in the file `SiteSpecific` which is contained in the top-level GRASP directory. To do this, edit the file `SiteSpecific` and set the variable `GRASP_HOME` to the appropriate value, for example `GRASP_HOME=/usr/local/GRASP`. Please note that the installation scripts are not designed to “build” in one location and “install” in a separate location. You should go through the installation procedure in the same directory where you eventually want the GRASP package to reside.

Within this top level directory resides the entire GRASP package. The directories within this top level are:

`Examples_SiteSpecific` Contains examples of `SiteSpecific` files for different sites, machine-types, and installations. You may find this helpful in the installation process if you want to look at an example, or you are stuck.

`bin/` Contains links to all the example programs and scripts in the GRASP package.

`data/` Contains (both real and simulated) interferometer data, or symbolic links to this data. See the comments in Section 3 to find out how to obtain this data.

`doc/` Documentation (in TeX, PostScript, DVI, and PDF formats) including this users guide.

`include/` Header files used to define structures and other common types in the code. This also include the ANSI C prototypes for all the GRASP functions.

`lib/` Contains the GRASP library archive: `libgrasp.a`. To use any of the GRASP functions within your own code, simply link this library with you own code.

`man/` This may be used in the future for UNIX on-line manual pages.

`parameters/` Contains parameters such as site location information, and estimated power spectra and whitening functions of future detectors.

`src/` Source code for analyzing various aspects of the data stream, distributed among the following directories:

`40-meter/` Reading data tapes produced on the Caltech 40 meter prototype prior to 1997.

`GRtoolbox/` Source code for the Gravitational Radiation Toolbox, a Matlab (command line and GUI) interface to GRASP.

`mexfiles/` Mex-files for use with the Gravitational Radiation Toolbox.

`examples/` The source code for all of the examples given in this manual (organized by section). These include:

`examples_40meter/` Examples of reading/using old-format 40-meter data.

`examples_GRtoolbox/` M-file examples for the Gravitational Radiation Toolbox.

`examples_binary-search/` The source code and documentation for a binary-inspiral search carried out on the Caltech 40-meter data from November 1994.

`examples_correlation/` Examples of determining correlations between different channels and using the knowledge of these correlations to ‘clean up’ a particular channel.

`examples_frame/` Examples of reading/using new-format FRAME data.

`examples_galaxy/` Examples of using galactic models to predict source distribution parameters.

`examples_inspirial/` Examples of generating inspiral waveforms and searching for them in the data stream using matched filtering.

`examples_ringdown/` Examples of generating black-hole-horizon formation ringdown waveforms and searching for them in the data stream using matched filtering.

`examples_stochastic/` Examples of simulated production of a stochastic background correlated signal between two detector sites and a pipeline to search the data stream for such signals.

`examples_template_bank/` Example code for setting up a bank of binary-inspiral templates and graphing their locations in parameter space.

`examples_testmass/` Example code for evaluating binary inspiral waveforms in the test-mass limit $m_1 \rightarrow 0$ and comparing the resulting waveforms with those calculated by other methods.

`examples_timefreq/` Example code illustrating the use of time-frequency techniques for signal detection.

`examples_transient/` Example code to generate and search for transient waveforms such as those arising from supernovae.

`examples_utility/` Examples of various utility functions, including a translator to produce new-format FRAME data from old format 40-meter data.

`correlation/` Code for calculating correlations between different channels and ‘cleaning’ a particular channel.

`galaxy/` Modelling the distribution of sources in our galaxy (needed in order to set physical upper-limits using the 40-meter prototype data).

`inspiral/` Binary inspiral analysis (including optimal filtering and vetoing).

`optimization/` Additional library routines for optimizing GRASP operation of specific platforms (i.e., supercomputers).

`ringdown/` Black hole horizon ringdown (including optimal filtering). This can be used to filter for *any* exponentially-decaying sinusoid.

`stochastic/` Stochastic background detection (including optimal filtering and simulated signal production)

`transient/` Supernovae and other transient sources.

`periodic/` Searches for pulsars and other periodic and quasi-periodic sources.

`template_bank/` Code for “placing” optimal filters in parameter space.

`testmass/` Code for calculating binary inspiral waveforms in the test mass limit $m_1 \rightarrow 0$.

`timefreq/` Code for time-frequency transforms, and searching for line-like features in the time-frequency maps.

`utility/` General purpose utility routines, including the interface to the FRAME library, error handler routines, etc.

`testing/` This will eventually contain a suite of programs that test the GRASP installation.

2.6.2 Accessing Numerical Recipes in C libraries

GRASP makes use of many of the functions and subroutines from *Numerical Recipes in C* [1]. The web site <http://www.nr.com/> is a good source of further information. These functions and subroutines are available in Fortran, Pascal, Basic, Kernighan and Ritchie (K&R) C, and ANSI-C versions; you will need the ANSI-C routines. The source code for these functions (both *.c and *.h files) must be installed in a directory (for example, `/usr/local/recipes/src`) and the compiled object modules (*.o files) must be archived into a single library file (*.a file). The instructions for this are included in the distribution of the source code for *Numerical Recipes*. In the end, a file called `librecipes_c.a` must be put into a directory where it is available to the linker for compilation. A good place to put this library is in `/usr/local/recipes/lib/librecipes_c.a`. When you run the command that installs GRASP, the linker needs to be able to find these libraries. The file `SiteSpecific` must then contain the line `RECIPES_LIB = /usr/local/recipes/lib` near the top of the file.

It is frequently useful, for debugging purposes, to be able to link with both “debug” and “profile” versions of the libraries. For this reason, we recommend that users actually create *three separate libraries* of *Numerical Recipes* functions:

`/usr/local/recipes/lib/librecipes_c.a`: a library compiled for fast execution, with optimization options (for example, `-O3` or `-xO4`) turned on during compilation.

`/usr/local/recipes/lib/librecipes_cg.a`: a library compiled for debugging, with the debug option (typically, `-g`) turned on during compilation. Note that in order to use a debugger with this library, and to be able to step “within” the *Numerical Recipes* functions, the debugger must be able to locate the source code for *Numerical Recipes*. Thus, after *Numerical Recipes* is compiled and installed, its *.c and *.h source files must be left in their original locations and not deleted or moved.

`/usr/local/recipes/lib/librecipes_cp.a`: a library compiled for profiling, with the profiling option (typically, `-pg` or `-xpg` for “gprof” or `-p` for “prof”) turned on during compilation.

One can then easily compile GRASP code with the appropriate library by setting `LRECIPES` in `SiteSpecific`. For example to run code as rapidly as possible one would set `LRECIPES = recipes_c`. However to compile code for debugging it would be preferable to set `LRECIPES = recipes_cg`. (Note that rather than recompiling the entire GRASP package in this way, one can simply modify the value of `LRECIPES` within the desired `Makefiles` and then recompile only the code of interest.)

We have encountered one minor problem with the *Numerical Recipes in C* routines. Unfortunately the authors of these routines choose to name one of their routines `select()`. This name conflicts with a POSIX name for one of the standard operating system calls. In linking with certain libraries (for example the MPI/MPE libraries) this can generate conflicts where the linker attaches the `select()` call to the entry point from the wrong library. Starting with release 1.6.3 of GRASP, the `select()` routine from Numerical Recipes is used in GRASP. For this reason, you must fix this as follows. Before building the *Numerical Recipes* libraries, edit the source files `recipes/rofunc.c`, `recipes/select.c`, and `recipes/select.c.orig` changing each occurrence of `select(` to `NRselect(`. You will have to do this in (respectively) four places, one place and one place in these files. Then edit the file `include/nr.h` making the same change of `select(` to `NRselect(` in one place. This will eliminate the `select()` routine from the *Numerical Recipes* library, replacing it with a routine called `NRselect()`, and eliminating any possible naming conflict from the library. So, to summarize, the routine called `select()` in the *Numerical Recipes* library is used in GRASP, but is called `NRselect()` there.

2.6.3 Accessing MPI and MPE libraries

To enable use of the parallel processing code included with GRASP, one needs to link the code with an MPI function call library. (If you do not intend to use any of the multiprocessing code, we’ll tell you what to do.) For performance monitoring purposes, we also make calls to the Message Passing Environment (MPE) library, which is included with `mpich` [3]. If these function libraries are not currently available on your system, you should obtain the public domain implementation `mpich` from the URL <http://www.mcs.anl.gov/mpi/mpich/> and follow the instructions required to build the MPI/MPE libraries for your system. After the installation process is complete, the necessary libraries will be contained in a library archive, for example `/usr/local/mpi/lib/libmpi.a` and `/usr/local/mpe/lib/libmpe.a`. The path to these libraries is set in the file `SiteSpecific` by means of the variable `MPI_LIBS`. A typical line in `SiteSpecific` might then read:

```
MPI_LIBS=-L/usr/local/mpi/lib -lmpi -lmpe.
```

You must also set `BUILD_MPI= true` in `SiteSpecific`. Finally, in order to include appropriate header files in any MPI programs, you will need to include a path to these header files in the file `SiteSpecific`. You can do this by setting `MPI_INCLUDES` in the file `SiteSpecific`. A typical installation might have

```
MPI_INCLUDES = -I/usr/local/mpi/include.
```

NOTE: If you don’t want to use *any* of the MPI code, just set:

```
BUILD_MPI= false
```

in `SiteSpecific`. All the other MPI-specific defines are then ignored.

2.6.4 Accessing MESA libraries

Currently one of the routines available in GRASP, `plottf()`, requires the Mesa library to display the time-frequency maps on the screen. Mesa is a 3-D graphics library with an API which is very similar to that of OpenGL. Mesa is distributed under the terms of the GNU Library General Public License. The

Mesa library may be downloaded from <http://www.mesa3d.org/> If you are not interested in using the `plottf()` routine, you may set `HAVE_GL= false` in `SiteSpecific` and ignore the rest of this section.

The Installation is extremely simple. Download the file `MesaLib-3.0.tar.gz`. The unzipped untarred file produces a directory tree under `Mesa-3.0`. Enter the directory `Mesa-3.0`, and key in `make`. This lists a variety of systems on which the Mesa library has been compiled. Select the one which most accurately describes your system and key in, `make my_system`, where `my_system` is what you have selected from the list. This will compile the programs and create the Mesa libraries in the directory, `Mesa-3.0/lib`. Copy the libraries to a common location such as `/usr/local/lib` and copy the include files in `Mesa-3.0/include-` to a common location such as `/usr/local/include`. (The files `README` and `README.*` files have detailed instructions to install the software, if required.)

2.6.5 Accessing CLAPACK libraries

As mentioned above GRASP uses routines from CLAPACK to perform the numerical linear algebra required in some of the environmental correlation routines. The routines that require these libraries are those which 'clean up' one channel based on an analysis of the correlations between a number of channels. In the case of the data stream from an interferometric gravitational radiation detector, the primary interest would be in the cleaning the signal determining the differential displacement of suspended test masses using information from environmental channels. If you are not interested in such routines you may set

```
WITH_CLAPACK= false
```

in `SiteSpecific` and ignore the rest of this section.

The CLAPACK routines may be downloaded from <http://www.netlib.org/clapack/>. It is simplest to download the complete package `clapack/clapack.tgz` although it is possible to download individual elements if disk space is at a premium (the complete package includes testing and timing routines which may be discarded after successful installation). The unzipped untarred file produces a directory tree under CLAPACK. The directory CLAPACK contains LAPACK make include file `make.inc` where compiler flags etc are set. You may wish to change the lines

```
BLASLIB      = ../..blas$(PLAT).a
LAPACKLIB    = lapack$(PLAT).a
```

to

```
BLASLIB      = ../..libblas$(PLAT).a
LAPACKLIB    = liblapack$(PLAT).a
```

CLAPACK uses the `f2c` libraries so the first step is to create these by typing `cd F2CLIBS/libF77; make` and `cd F2CLIBS/libI77; make` each time starting from the CLAPACK directory. Next one builds the BLAS (Basic Linear Algebra Subprograms) libraries with `cd BLAS/SRC; make`. Finally one builds the CLAPACK library with `cd SRC; make`. The `f2c` libraries `libF77.a` and `libI77.a` and include file `f2c.h` are now in the subdirectory `F2CLIBS` of CLAPACK while the libraries `libblas$(PLAT).a` and `liblapack$(PLAT).a` are in the directory CLAPACK. From here they may be installed into appropriate directories. Fuller details, including the building and running of the test and timing programs may be found in the `README` file in the CLAPACK directory. As for the Numerical Recipes libraries it can be convenient to have both optimised and debugging versions of the libraries available for development work.

2.6.6 Accessing FRAME libraries

The LIGO and VIRGO detector projects have recently decided to standardize the format which their data will be recorded in (see Section 2.4). The standard is called the FRAME format, and is still under development. It appears quite possible that a number of other gravitational-wave detector groups will also adopt this same format. The GRASP package contains, for every example program, both FRAME format and old format versions. It also contains an translation program which converts data from the “old 1994” format into the new FRAME format.

Unless you are in one of the small number of groups with access to the old-format data, you will need to obtain the FRAME libraries. These are available from the VIRGO project; they may be downloaded from the site <http://wwwlapp.in2p3.fr/virgo/FrameL/>. Contact Benoit Mours mours@lapp.in2p3.fr for further information. In the `SiteSpecific` file, if you need the FRAME libraries, set a pointer to the directory containing them. NOTE: If you don't need the FRAME libraries, just set:

```
BUILD_FRAME = false
```

in `SiteSpecific`. All the other FRAME-specific defines are then ignored.

The GRASP interface to the FRAME library should work properly with every version of the FRAME library from 2.30 onwards. The GRASP interface to the FRAME library looks to see which version of the FRAME library you are using, and then generates the appropriate code. The FRAME library is designed to be backwards-compatible. For example, version 3.42 of the FRAME library can read files written with version 2.37 of the FRAME library. GRASP has been tested with versions of the FRAME ≤ 3.72 .

2.6.7 Real-time 40-meter analysis

The analysis tools in the GRASP package can be used to analyze data in real-time, as it is recorded by the DAQ system. This facility is primarily for the use of experimenters working in the Caltech 40-meter lab, and will probably not be of use to anyone outside of that group.

In order to use the GRASP tools in real time, one needs to link to a set of EPICS (Experimental Physics and Industrial Control System) libraries, that are not otherwise needed. These permit the GRASP code to interrogate the EPICS system to find out the names and locations of the most-recently written FRAMES of data.

2.6.8 The Matlab Interface

The Gravitational Radiation Toolbox provides a Matlab interface to both GRASP and the Frame Library. The Gravitational Radiation Toolbox links these two packages with Matlab—simultaneously exposing data to a familiar, commercially developed, problem solving environment and efficient algorithms designed specifically for analyzing gravitational radiation data.

2.6.9 Making the GRASP binaries and libraries

To make the GRASP libraries and executables described in this manual, please follow these directions. It should only take a few minutes to do this.

1. Within the main GRASP directory is a file called `SiteSpecific`. Make a copy of `SiteSpecific` called `SiteSpecific.save`. This way, if you mess up the installation, you can start over easily. (Alternatively, copy `SiteSpecific` to a file called `SiteSpecific.mysite` and, everywhere below, when we refer to editing the `SiteSpecific` file, edit `SiteSpecific.mysite` instead.) Note: you can find a number of example `SiteSpecific` files in the directory `Examples_SiteSpecific/`. These are for different installation sites and machine types (Sun, DEC,

Intel Paragon, IBM SP2, Linux) – you may find them helpful if you are stuck or the instructions below are ambiguous or unclear. Once you have customized the `SiteSpecific` file for your own installation, if you wish you can email it to us and we will include it in future releases of GRASP.

2. Now edit `SiteSpecific` so that `GRASP_HOME` has the correct path, for example
`GRASP_HOME=/usr/local/GRASP.`
This must be the name of the directory on your system in which GRASP resides. If you are not the superuser and are installing GRASP only for your own use, you can set this path to point somewhere in your own home directory, and install GRASP there.
3. Find out where *Numerical Recipes in C* is installed on your system. Within `SiteSpecific` set `RECIPES_LIB` to point to the directory containing these libraries. For example
`RECIPES_LIB=/usr/local/numerical_recipes/lib.`
If *Numerical Recipes in C* is not installed on your system, you will have to obtain a copy, and install it, following the directions to create the library file `librecipes_c.a`. Note that as described above, you might also want to create debugging libraries `librecipes_cg.a` and profiling libraries `librecipes_cp.a`.
4. Within `SiteSpecific` set `LRECIPES` to the name of the *Numerical Recipes in C* library you wish to use, for example
`LRECIPES=recipes_c.`
5. If you intend to use the MPI code, set `BUILD_MPI= true`, otherwise set it to `false`. In this latter case, any MPI-specific defines are ignored, and no code that makes use of MPI/MPE function calls is compiled. (This is a shame – these are some of the nicest programs in the GRASP package. We urge you to reconsider building the `mpich` package on your system!)
6. Within `SiteSpecific` set `MPI_LIBS` to point to the directory containing the MPI/MPE libraries, and to specify the names of the link archives, for example
`MPI_LIBS=-L/usr/local/mpi/lib -lmpi -lmpe.`
Note that if you use the version of `mpicc` which is distributed with `mpich` you may not need to have any of the MPI libraries referenced here; the compiler may find them automatically.
7. Within `SiteSpecific` set `MPI_INCLUDES` to point to the directory which contains the MPI and MPE header (`*.h`) files, for example
`MPI_INCLUDES = -I/usr/local/mpi/include.`
8. Within `SiteSpecific` set `MPICC` to the name of your local MPI C compiler, for example:
`MPICC = /usr/local/bin/mpicc.`
You can include any compilation flags (say, `-g`) on this line also.
9. If you have the MESA or GL library installed set `HAVE_GL= true`, otherwise set it to `false`. In this latter case, the routines making GL/MESA calls will not be compiled.
10. Within `SiteSpecific` set `GL_LIBS` to point to the directory containing the GL/MESA libraries, and to specify the names of the link archives, for example
`GL_LIBS= -L/usr/local/lib -lMesaGLU -lMesaGL $(XLIBS).`
Note that the functions in the MESA/GL library make calls to the X library and you will have to specify the location of the X libraries for example
`XLIBS = -L/usr/X11/lib -L/usr/X11R6/lib -lX11 -lXext -lXmu -lXt -lXi -lSM -lICE .`

11. Within `SiteSpecific` set `GL_I` to point to the directory which contains the GL/MESA header (*.h) files, for example
`GL_I = -I/usr/local/include/GL.`
12. If you intend to use CLAPACK, set `WITH_CLAPACK = true`, otherwise set it to false. Within `SiteSpecific` set `CLAPACK_LIB` to point to the directory containing the CLAPACK libraries and `LCLAPACK` and `LBLAS` to the (platform-specific) names of the clapack and blas libraries respectively excluding the leading 'lib'. Further set `F2C_LIB` to point to the directory containing the f2c libraries and `F2C_INC` to point to the directory containing the f2c.h include file.
13. If you intend to use the FRAME code, set `BUILD_FRAME = true`, otherwise set it to false. In this latter case, any FRAME-specific defines are ignored, and no code that makes use of FRAME function calls is compiled.
14. Within `SiteSpecific` set `FRAME_DIR` to point to the directory which contains the LIGO/VIRGO format FRAME software, for example
`FRAME_DIR=/usr/local/frame.`
This directory should contain `lib/libFrame.a` and `include/FrameL.h`. If you don't need the FRAME libraries, just leave this entry blank.
15. Within `SiteSpecific`, if you want to use GRASP for real-time analysis in the Caltech 40-meter lab, set `EPICS_INCLUDES` to point to the directory containing the EPICS *.h include files, and set `EPICS_LIBS` to point to the directory containing the EPICS libraries. Finally, you need to uncomment the `BUILD_REALTIME` define statement. If you do not intend to use your GRASP installation for real-time analysis in the 40-meter lab, simply leave these three definitions commented out with a hash sign (#).
16. At the bottom of `SiteSpecific` are several define statements, which are currently commented out. These are primarily intended for production code; by undefining these lines you replace a cube root function and some trig functions in the code with faster (but less accurate) in-line approximations. We suggest that you leave these commented out. (You might want to consider uncommenting them if you are burning thousands of node hours on a large parallel machine - but you do so at your own risk!)
17. There are also lines that are currently commented out, which allow you to overload functions defined in the libraries and reference libraries of optimized functions. Once again, leave these commented out unless you want to replace standard *Numerical Recipes* functions with optimized versions. Currently, we support several sets of optimized libraries:
 - The CLASSPACK optimized FFT's for the Intel Paragon.
 - The Sun Performance Library's optimized FFT for the Sun SPARC architecture. Note: believe it or not, this is *slower* than the public domain equivalent. We recommend that you use the FFTW package instead!
 - The Cray/SGI optimized FFT for the RS10000 and other MIPS architectures. Note: believe it or not, this is *slower* than the public domain equivalent. We recommend that you use the FFTW package instead!
 - The DEC extended math library (DXML) optimized FFT for the DEC AXP architecture. [This is slightly faster, or slightly slower, than FFTW, depending upon the array size.]
 - The FFTW (Fastest Fourier Transform in the West), which will run on any computer. This is a public domain optimized FFT package, available from the web site:

<http://www.fftw.org/>. If you don't have an optimized FFT routine for your computer, we highly recommend this – it is a factor of three (or more) faster than *Numerical Recipes*. We include glue routines for both FFTW version 1 and version 2. The latter is simpler to install and fractionally more efficient.

- The IBM Extended Scientific Subroutine Library (ESSL) optimized FFT routine. Note: this has only been tested on the IBM SP2 machine.

Further details may be found in the `src/optimization` subdirectory of GRASP. If you want to use these optimized library routines, first go into the appropriate subdirectory of `src/optimization` and build the optimized library routine using the `makefiles`'s that you find there, then uncomment the appropriate lines in `SiteSpecific` and follow the instructions given here.

18. To install the Gravitational Radiation Toolbox which provides an interface between GRASP and Matlab, comment out the line `BUILD_GRTOOLBOX = false` and uncomment `BUILD_GRTOOLBOX = true`. Then edit the variables `MEX`, `MEXFLAGS`, and `EXT` appropriately. You will then have add the directories `src/GRtoolbox` and `src/examples/examples_GRtoolbox` to your Matlab path.
19. Now, in the top level GRASP directory, execute the shell script `InstallGRASP`, by typing the commands:

```
chmod +x InstallGRASP
./InstallGRASP SiteSpecific (or SiteSpecific.mysite if appropriate)
```

From here on, the remainder of the installation should proceed automatically. The `InstallGRASP` script takes information contained in the `SiteSpecific` file (or in the file named in the first argument of `InstallGRASP` such as `SiteSpecific.mysite`) and uses it to create `Makefile`'s in each `src` subdirectory, and runs `make` in each of those directories.
20. If you want to “uninstall” GRASP so that you can begin the installation procedure again, cleanly, a script has been provided for this purpose. To execute it, type:

```
./RemoveGRASP
```

and wait until the script reports that it has finished. Note: when you have successfully completed this process, please email us a copy of your `SiteSpecific` file and we will put it into the `Examples_SiteSpecific/` directory of future GRASP releases.

The `Makefile` in each directory is constructed by concatenating the file named in the first argument of `InstallGRASP` (typically `SiteSpecific`) with a file called `Makefile.tail` in each individual directory. If you want to try changing the compilation procedure, you can modify the `Makefile` in a given directory. However this will be created each time that you run `InstallGRASP`; for changes to become permanent they should either be made in `SiteSpecific` or in the `Makefile.tail`'s.

Note that this installation procedure and code has been tested on the following types of machines: Sun 4 (Solaris), DEC AXP (OSF), IBM SP2 (AIX), HP 700 (HPUX), Intel (Linux), Intel Paragon. There is a problem on some SGI (IRIX) machines. If you get error messages reading:

```
...
if: Expression Syntax.
*** Error code 1 (bu21)
GRASP did NOT complete installation successfully
```

this can be fixed by setting the shell to `bash` before running `InstallGRASP`:

```
SGI> setenv SHELL /usr/local/bin/bash
SGI> ./InstallGRASP SiteSpecific
```

If you run into problems with our installation scripts, please let us know so that we can fix them.

If you want to experiment with GRASP or to write code of your own, a good way to start is to copy the `Makefile` and the example (`*.c`) programs from the `src/examples` directory into a directory of your own. You can then edit one of the example programs, and type “make” within your directory to compile a modified version of the program.

If you wish to modify the code and libraries distributed with GRASP (in other words, modify the functions described in this manual!) the best idea is to use `cp -r` to recursively copy the entire GRASP directory structure (and all associated files) into a private directory which you own. You can then install your personal copy of GRASP, by following the directions above. This will permit you to modify source code within any of the `src` subdirectories; typing `make` within that directory will automatically re-build the GRASP libraries that you are using. By the way, if you are modifying these functions to fix bugs or repair problems, or if you have a “better way” of doing something, please let us know so that we can consider incorporating those changes in the general GRASP distribution.

2.6.10 Stupid Pet Tricks

There are a number of simple things that one can do during or after the installation process that may make GRASP easier to maintain and/or use at your site. For example, it is often extremely convenient for debugging purposes to have a GRASP library (`libgrasp.a`) constructed with all the symbol table information turned on, and another library constructed with all the optimization switches turned on. Users who want their code to run as fast as possible can link to the optimized library. Users who want to track down problems within GRASP, or to step through internal GRASP functions can link to the debug library. You can accomplish this easily by building two separate GRASP libraries, as follows. (Note: since the normal C-compiler debugging option is `-g` the debug library has a `_g` appended to its name.)

- Edit your `SiteSpecific.mysite` file so that the debugging switches are turned on (`CFLAGS = -g`, typically). You may also want to build the GRASP example programs with the “debug” versions of the Numerical Recipes libraries, in which case you should set `LRECIPES=recipes_cg` in `SiteSpecific.mysite`. In similar fashion you might also choose to link to FRAME libraries compiled with debugging turned on.
- Build GRASP as described above, by running `InstallGRASP SiteSpecific.mysite`.
- Make a copy of the GRASP library:

```
cp /usr/local/GRASP/lib/libgrasp.a /usr/local/GRASP/lib/libgrasp_g.a
```
- Make a copy of the “debug” GRASP example programs:

```
cp -r /usr/local/GRASP/bin /usr/local/GRASP/bin_g
```

Note: the `/usr/local/GRASP/bin` directory contains *links* to the actual executables, but for most unix systems this copy command will copy the actual files. Your mileage may vary – choose the copy option which copies the files *not* the links!
- Remove your GRASP installation (i.e. everything but the library and the executables, and original GRASP package) by typing:
`RemoveGRASP`

- Modify `SiteSpecific.mysite` so that the optimization options are turned on (typically, `CFLAGS = -O`). You should also set the Numerical Recipes library to the optimized versions, typically via `LRECIPIES=recipes_c` in `SiteSpecific.mysite`. In similar fashion you might also choose to link to FRAME libraries compiled with optimization turned on.
- Install GRASP again:
`InstallGRASP SiteSpecific.mysite`

Your GRASP installation will now contain *two* GRASP libraries: `/usr/local/GRASP/lib/libgrasp.a` and `/usr/local/GRASP/lib/libgrasp_g.a` and two sets of executables, in `/usr/local/GRASP/bin` and `/usr/local/GRASP/bin_g`.

Another useful trick is if you are building versions of GRASP for several different architectures, on a shared `/usr/local/` disk. Here the procedure is the following:

- Create a `SiteSpecific.arch1` file for the first machine type.
- Install GRASP in the usual way:
`InstallGRASP SiteSpecific.arch1`
- Copy the libraries:
`cp /usr/local/GRASP/lib/libgrasp.a /usr/local/GRASP/lib/libgrasp_arch1.a`
- Make a copy of the GRASP example programs:
`cp -r /usr/local/GRASP/bin /usr/local/GRASP/bin_arch1`
Note: the `/usr/local/GRASP/bin` directory contains *links* to the actual executables, but for most unix systems this copy command will copy the actual files. Your mileage may vary – choose the copy option which copies the files *not* the links!
- *Remove* your GRASP installation (i.e. everything but the library and the executables, and original GRASP package) by typing:
`RemoveGRASP`
- Return to the first step above, and begin this process again, but this time for the second machine architecture (i.e. change `arch1` to `arch2` above).

This method will avoid duplication of source files, documentation, etc, while still providing a set of libraries and executables for different machine types.

2.7 Conventions used in this manual

The conventions used in this manual are not strict ones. However we do observe a few general rules:

1. Words or lines that you might type on a computer (commands, filenames, names of C-language functions, and so are) are generally indicated in *teletype* font.
2. When a function is described, the arguments which are *inputs* and those which are *outputs* (or those which are both) are indicated. Thus, for example the (fictional!) addition function `add(int a, int b, int* c)` which sets `*c = a+b` is described by:
 - a: Input. One of the two integers that are added together.
 - b: Input. The second of these integers.
 - c: Output. Set to the sum of a and b.

Note that technically this is incorrect, because of course in C even the “output arguments” are really just inputs; they are pointers to an address in memory that the routine is supposed to modify. And technically, the statement that “c is set to...” is not correct, since in fact it is the integer pointed to by c (denoted *c) that is set. However we find that this convention makes it much easier to read the function descriptions!

3. Most of the time, the example programs using GRASP functions are given explicitly in the manual, so you can see the GRASP functions “in use”. Because these examples are illustrative, they are generally “pared down” as much as possible (for example, default values of adjustable parameters are hard-wired in, rather than prompted for).
4. Routines and example programs in GRASP generally begin with the line:


```
#include "grasp.h"
```

 which includes the prototypes for all GRASP functions as well as the library header files `stdio.h`, `stdlib.h`, `math.h`, `values.h`, and `time.h`. The GRASP include file `"grasp.h"` can be found in the `include` subdirectory of GRASP.

2.8 How to add your contributions to future GRASP releases.

As we have explained, the general idea of GRASP is to have a collection of documented and tested code available for use by the gravitational-wave detection community. Many people have made significant contributions to this package, and we would welcome any additional contributions.

In order to minimize the effort involved in making additions to GRASP, and in order to ensure that they are properly included and available to all, here are some guidelines about how to contribute:

- The contributions must be structured in such a way that they can be installed using the standard GRASP installation scripts, and they respect the GRASP file hierarchy.
- The contributions must be documented in **TeX**, following the same general style as this manual (we try not to be *too* nit-picky!).
- In general, be sure that you are using (and modifying!) the current release of GRASP. This makes it much easier for me to merge your additions in with the existing code. The danger is that if you modify files from an old release of GRASP, where other corrections/changes have subsequently been made, then I need to try and merge these changes into what you have done. It’s easier for everyone if you are modifying the most current files. If you contact me in advance, I can also give you some idea about how many changes have already been made to the current release, and what the schedule is for the next release. One way to find the absolute “most current” version of a file is to get it from the GRASP development source tree, which is at
<http://www.lsc-group.phys.uwm.edu/~ballen/grasp-distribution/GRASP/>.
 Before sending me a revised file to incorporate in GRASP, please `diff` it with the corresponding file in this directory, to make sure that the only differences are ones that you have deliberately made!

If you want to make “small” changes to GRASP, for example to modify a function to add extra functionality, to repair something that is broken, or to add some additional functions in one section, then please do the following:

1. Provide documentation in the form of a modified file: `doc/man_*.tex`. I will merge your changes into the general GRASP distribution. For clarity, let’s assume that you have added a utility function, and have modified `doc/man_utility.tex` by adding a description of your function(s) to it. Remember, *software can never be better than its documentation*.

2. In any modifications of the documentation `*.tex` files, please be sure to *spell-check* the files before you send them to me. Use either the `spell` or `ispell` utilities, or some other alternative.
3. You should provide additional lines to add to `doc/make_tex_from_C`. This is a script that automatically converts any `*.c` example files that you would like to include in the manual into `.tex` files. In general, you should have an example program which shows a very simple use of your function.
4. You should provide any figures which have been included in your "modified" `doc/man_*.tex` file in postscript: `doc/Figures/MYPACKAGE1.ps`, `doc/Figures/MYPACKAGE2.ps`, and so on. Note that the postscript files produced by many plotting packages are excessively large, often several MB. For figures produced from programs like this, it is more efficient to use a bitmap to describe the entire image. Full details of how to produce such bitmaps may be found at <http://xxx.lanl.gov/help/bitmap>. The following extract describes 'the easiest way to do bitmapping' using XV:

"First display the original figure on the screen somehow (e.g. with `ghostview`). Then use the 'Grab' button in XV to snatch a copy of the displayed image into XV's buffer (after selecting 'Grab' you can do this either by clicking the left mouse button on the desired window (which grabs the whole window), or by holding down the middle mouse button and dragging (which selects a region)).

Once the image is in XV's buffer, you can manipulate it. You should use 'Autocrop' or 'Crop' to remove any excess margins around the figure. Then save it (as gif, jpeg, color postscript or greyscale postscript). If resaving as postscript you must click the XV 'compress' box for extra compression."
5. You should provide a modified version of (for example) `src/utility/utility.c` (this modified source contains your additions, merged into the standard GRASP release) and additional example programs demonstrating your functions in `src/examples/examples_utility/example1.c`, `src/examples/examples_utility/example2.c`, and so on.
6. You should provide a modified version of `include/grasp.h` (or the additional lines to merge into this header file). This header should contain proto-types for any functions which you have added to GRASP, which you would like to make publicly-available. In general, please try to avoid putting "documentation" in this header file: it should go into the manual and into the source files.

If you are doing this (modifying or extending existing GRASP functions) please do a search of the existing GRASP source code to verify that your changes do not break existing code. Or, if necessary, make modifications to the existing code, and send me those modified files as well as the materials above.

On the other hand, you might have grander plans! You might not want to make "small" changes - you might want to include a major new section in GRASP, for modelling another type of source, or for a type of analysis which is different than anything currently in the package. Let's assume that you want to provide a major new GRASP package or facility called MYPACKAGE. In this case:

1. You should provide documentation in the form of a file: `doc/man_MYPACKAGE.tex`, which has the same format as (for example) `doc/man_inspiral.tex`. I will modify `doc/manual.tex` by putting a line:


```
include{man_MYPACKAGE}
```

 into `doc/manual.tex`, to include your contribution to the manual. Reference should be in the same format as the existing ones, and will be added to the *references* section of `manual.tex`. Make sure that you modify `doc/man_intro.tex` to describe any additional directories that you have added to the `src` tree. Remember, *software can never be better than its documentation*.

2. In any modifications of or additions to the documentation `*.tex` files, please be sure to *spell-check* the files before you send them to me. Use either the `spell` or `ispell` utilities, or some other alternative.
3. You should provide additional lines to add to `doc/make_tex_from_C`. This is a script that automatically converts any `*.c` example files that you would like to include in the manual into `.tex` files.
4. You should provide figures in postscript form with names derived from your package name if that is possible. For example: `doc/Figures/MYPACKAGE1.ps`, `doc/Figures/MYPACKAGE2.ps`. These figures should be included in your `doc/man_MYPACKAGE.tex` file.
5. You should provide source code in `src/MYPACKAGE/MYPACKAGE.c` and example programs in `src/examples/examples_MYPACKAGE/example1.c`, `src/examples/examples_MYPACKAGE/example2.c`, and so on. The code in `src/MYPACKAGE/MYPACKAGE.c` contains the actual functions that you have provided. Any executable example programs that use these functions should be in the `src/examples` path.
6. You should provide a modified header file `include/grasp.h` or additional lines to merge into this, declaring prototypes for any publicly-available functions.
7. You should provide “tail” parts of the Makefiles:
`src/examples/examples_MYPACKAGE/Makefile.tail`, and
`src/MYPACKAGE/Makefile.tail`. You can see
`src/examples/examples_inspirial/Makefile.tail` for an example. Please follow the syntax of this fairly closely: the structure is there for good reasons. If you provide a file that works on your own system it may not work on other people’s systems – but if you follow our style it probably will.
8. Be sure to modify the `InstallGRASP` utility to include a “build” in any directories that you have added to the `src` tree.

In general, the “rule of thumb” here is that you should try not to add functions which substantially overlap existing GRASP functions. Either modify the existing GRASP functions (as described below) to add the extra functionality or use them “as is”.

I would be grateful for a bit of advanced warning about any additions to GRASP (but a uuencoded gzipped tarfile or shar file dropped in my mailbox *will* get my rapid attention, if it contains these different items, because that makes it *easy* for me to incorporate it!) The best format for this file is to make it contain only the files that you are adding to GRASP, or those files from the most current GRASP distribution that are being modified, *with exactly the correct directory tree structure*. This makes it easy for me to unpack your contributions and merge them into the general GRASP distribution. One way to find the absolute “most current” version of a file is to get it from the GRASP development source tree, which can be reached from <http://www.lsc-group.phys.uwm.edu/~ballen/grasp-distribution/GRASP/>. Before sending me a revised file to incorporate in GRASP, please `diff` it with the file in this directory, to make sure that the only differences are ones that you have deliberately made!

Note: it is a good idea to check your code in the following ways:

- Error messages in your code should not be done with `fprintf(stderr, ...)`, but should be dealt with by calling the GRASP error handler. Look at any of the current GRASP code to see how this is done, or read Section 16.1 on the error handler.

- Make sure that it passes by `lint` cleanly.
- Stick to POSIX operating systems calls, only. Remember that your code needs to run on *other* platforms. The fact that the code works correctly on your platform does not guarantee similar behavior on other types of machines. This item, and the previous one, will go a long way towards ensuring that.
- If you are using the `gcc` compiler, make sure that the `-Wall` option (which enables all warnings) does not issue any warnings.
- If you have access to more than one type of machine, please test your code on both a 32-bit and 64-bit machine if possible.
- If you have access to both big-endian and little-endian machines, please test you code on both of those also.

Please remember that many people will be running *your* code on platforms which are different than yours. The fact that your code runs properly on your platform does not mean that it will run properly on other ones as well. The best way to ensure this is to eliminate the types of problematic constructions that `lint` and `-Wall` warn about, and to test your code on a couple of different machines.

One final (**important**) note. When your code has been sucessfully integrated into the GRASP package, we will issue a new release of GRASP as quickly as possible. As soon as this release is available, we *strongly* recommend that you *throw away* your “personal” working copies of the files that you have been creating or modifying, install this latest release of GRASP, and make *new* copies of the various files to work from. The reason is this: in the process of including your material into GRASP we have probably made a number of changes to it. If you *don't* follow our suggestion, make additional changes to your own files and send them to us, we will not be very receptive (as you are forcing us to perform the unpleasant task of merging your changes with our earlier ones).

2.9 How to use the GRASP library from ROOT.

ROOT is an interactive public-domain environment for data analysis developed at CERN. Details about it can be found at <http://root.cern.ch/>. Within the root environment, you can make use of the GRASP library. To do this, however, you need to produce a shared-object version of the GRASP library, and then load it into ROOT. The following instructions on how to do this were contributed by Damir Buskulic (buskulic@lapp.in2p3.fr), and have been tested under Linux.

1. In building `libgrasp.a` and `librecipes_c.a` be sure that you have a *position independent code* option. For the `gcc` compiler this is `-fPIC`. This is needed to build a proper shared library.
2. The ROOT environment variables should have been set during install, especially `$ROOTSYS`.
3. Issue the following three commands:
 - To build the interface functions for the ROOT C interpreter:

```
cint -wl -zlibgrasp -nG_cpp_grasp.cc -D_MAKECINT_ -DG_MAKECINT
-c-1 -DG_REGEX -DG_SHAREDLIB -DG_OSFDDL -D_cplusplus
-I$ROOTSYS/include -I<path.to.GRASP>/include/grasp.h
```

This will create files `G_cpp_grasp.cc` and `G_cpp_grasp.h` which need to be compiled.

- Compile these files with:

```
gcc -O -fPIC -DG_REGEX -DG_SHARED_LIB -DG_OSFDLL  
-I$ROOTSYS/include -c G_cpp_grasp.cc
```
- Build the shared library with the command:

```
gcc -shared -o libgrasp.so ./G_c_grasp.o <path-to-libgrasp.a>/libgrasp.a  
<path-to-numrecipes>/librecipes.c.a
```

creating a `libgrasp.so` file. You can put this file anywhere that you wish.

Note: it is important to put the files and libraries in the order given above.

4. Launch ROOT and use the command `gSystem->Load("libgrasp");` to have the routines from the GRASP library available within ROOT.

Note that `cint` has some trouble with prototypes for variable-length-argument (`varargs`) functions. These types of functions are used to implement the GRASP error handler 16.1. For this reason the GRASP include file `include/grasp.h` has some lines which are automatically *not* included if the file is being read by `cint`. Thus the GRASP error-handling functions can't be called from within ROOT (but you wouldn't want to do this anyway).

Note that one can compile the Frame library for use with ROOT in exactly the same way. You replace `grasp.h` by `FrameL.h` and `libgrasp.a` by `libFrame.a` (always compiled with `-fPIC`). There is no need for `librecipes` in the `FrameLib` case, because it does not make use of *Numerical Recipes*.

3 GRASP Routines: Reading/using Caltech 40-meter prototype data

This Section of the GRASP manual is now OBSOLETE (except for historical interest) - it describes the OLD 1994 40-meter data format. The 40-meter data from 1994 has been translated into FRAME format, and is distributed in FRAME format only. Please go to Section 4 of the GRASP manual, which contains details about the FRAME format 1994 data.

There is a good archive of data from the Caltech 40-meter prototype interferometer. (Note that this name is slightly misleading: the average length of the optical arm cavities is 38.25 meters.) Although the interferometer is only sensitive enough to detect events like binary inspiral within ≈ 10 kpc (the distance to the galactic center) its output is nevertheless very useful in studying data analysis algorithms on real-world interferometer noise. This data was taken during the period from 1993 to 1996; for our purposes here we will concentrate on data taken during a one-week long observation run from November 14-21, 1994. The original data is contained on 11 exabyte tapes with about 46 total hours of data; the instrument was in lock about 88% of the time. The details of this run, the status of the instrument, and the properties of this data are well-described in theses by Gillespe [31] and Lyons [32].

The GRASP package includes routines for reading this data. The data is not read directly from the tapes themselves; the data instead must be read off the tapes and put onto disk (or into pipes) using a program called `extract`. The GRASP routines can then be used to read the resulting files. While the GRASP routines can be used without any further understanding of the data format, it is very helpful to understand this in more detail. Note that these data formats and the associated structures were defined years before GRASP was written; we did not choose this data format and should not be held accountable for its shortcomings. We have included a preliminary translator that translates the data from this old 1994 format into the new LIGO/VIRGO frame format. The program `translate` may be found in the GRASP `src/examples/examples_utility` directory, and is documented in the Section on GRASP general purpose utilities.

If you want to develop or work on data analysis algorithms, you will want to have access to this data archive. Because many people contributed to taking this data, and because the LIGO project wants to maintain control of its use and distribution, *this data set is NOT in the public domain*. However, you may request a copy for your use, or for use by your research group. Write to: Director of the LIGO Laboratory, Mail Stop 51-33, California Institute of Technology, Pasadena, CA 91125. The data set is available in `tar` format on two Exabyte 8500c format tapes. Each directory (for a different run on a different day) occupies the following amount of space (in mbytes):

14nov94.1	647
14nov94.2	913
18nov94.1	1041
18nov94.2	1121
19nov94.1	1554
19nov94.2	1074
19nov94.3	1250
19nov94.4	1206
20nov94.1	1146
20nov94.2	1173
20nov94.3	1543

Each of these directories contains the `channel.*` files and the `swept-sine.ascii` swept-sine calibration files. In this manual, we assume that these directories (or links to them) have been placed where you can access them. The GRASP programs that use this data determine its location by means of the environment variable `GRASP_DATAPATH`. You can set this by typing (for example)

setenv GRASP_DATAPATH /usr/local/data/19nov94.3
to access the data from run 3 on November 19th. System administrators: after installing these directories in a convenient place on your machine, we recommend that you install a set of links to them in the directory data within the GRASP home directory. This way your users can find them without asking you for the location!

WARNING: this data was written on a “big-endian” machine (the sun-4 workstation is an example of such a machine). The floats are in IEEE 754 floating-point format. Attempts to read the data in its distributed form on a “little-endian” machine (such as Intel 80*86 computers) will yield garbage unless the bytes are properly swapped. The routines used to read data (in particular, the function read_block()) test the byte order of the machine being used, and swaps the byte order if the machine is “little-endian”. This introduces some inefficiency if you are running on a “little-endian” machine, but is preferable to having two copies of the data, one for each architecture. If you are doing all of your work on a “little-endian” machine and you want to avoid this inefficiency, write a program which properly swaps the byte orders of the header blocks (which are in 4-byte units) and then also properly swaps the byte order of the data blocks (which are 2-byte units) and reformat the raw data files. Then modify the read_block() data so that it no longer swaps the bytes on your machine.

3.1 The data format

Data is written onto the exabyte tapes in blocks about 1/2 megabyte in size. The format of the data on the tapes is as shown in Table 1. The tape begins with a main header (denoted “mh” in the table). This is

mh	0's	0's	mh	0's	0's	mh	gh	0's	data	mh	gh	0's	data	...
1024		1024		1024		1024		1024 × n		1024		1024 × n		...

Table 1: Format of Exabyte data tapes (first row: content, second row: length in bytes).

followed by a set of zeros, padding the length of the header block to 1024 bytes. There is then an empty block of 1024 bytes containing zeros. This pattern is repeated until the first block containing actual data. This is signaled by the appearance of a main header, followed by a gravity header (denoted “gh” in the figure above). These two headers are padded with zeros to a length of 1024 bytes. This is then followed by a set of data (the length of this set is a multiple of 1024 bytes). Information about the length of the data sets is contained in the headers. The data sets themselves consist of data from a total of 16 channels, each of which comes from a 12-bit A to D converter. Four of the 16 channels are fast (sample rates a bit slower than 10kHz) and the remaining 12 channels are slow (sample rates a bit slower than 1kHz). The ratio of sample rates is exactly 10 : 1. Within the blocks labeled “data”, these samples are interleaved. The information content of the different channels is detailed on page 136 of Lyon’s thesis [32], and is summarized in Table 3.

The program extract reads data off the tapes and writes them into files. One file is produced for each channel; typically these files are named channel.0 → channel.15. The complete set of these files for the November 1994 run fits onto two Exabyte tapes (in the 8500c compressed format). The information in these files begins only at the moment when the useful data (starting with the gravity header blocks) begins to arrive. The format of the data in these channel.* files is shown in Table 2. Here the main headers are

block 0				block 1				block 2				block 3				...
mh	bh	0's	data	...												
1024		cs		1024		cs		1024		cs		1024		cs		...

Table 2: Format of a channel.0 → 15 file (first row: block number, second row: content, third row: length in bytes).

the same as before, however the headers that follow them are called binary headers (denoted by “bh” in the table). The length of the data stream (in bytes) is called the “chunksize” and is denoted by “cs” in Table 2. We frequently reference the data in these files by “block number” and “offset”. The block number is an integer ≥ 0 and is shown in Table 2. The offset is an integer which, within a given block, defines the offset of a data element from the first data element in the block. In a block containing 5000 samples, these offsets would be numbered from 0 to 4999.

The structure of the binary headers is

```
struct ld_binheader {
```

```
float elapsed_time: This is the total elapsed time in seconds, typically starting from the first valid  
block of data, from the beginning of the run.
```

```
float datarate: This is the sample rate of the channel, in Hz.
```

```
};
```

The structure of the main headers is

```
struct ld_mainheader {
```

```
int chunksize: The size of the data segment that follows, in bytes.
```

```
int filetype: Undocumented; often 1 or 2.
```

```
int epoch_time_sec: The number of seconds after January 1, 1970, Coordinated Universal Time  
(UTC) for the first sample. This is the quantity returned by the function time() in the standard C  
library.
```

```
int epoch_time_msec: The number of milliseconds which should be added to the previous quantity.
```

```
int tod_second: Seconds after minute, 0-61 for leap second, local California time.
```

```
int tod_minute: Minutes after hour 0-59, local California time.
```

```
int tod_hour: Hour since midnight 0-23, local California time.
```

```
int date_day: Day of the month, 1-31, local California time.
```

```
int date_month: Month of the year, 0-11 is January-December, local California time.
```

```
int date_year: Years since 1900, local California time.
```

```
int date_dow: Days since Sunday, 0-6, local California time.
```

```
int sub_hdr_flag: Undocumented.
```

```
};
```

Note: in the original headers, these `int` were declared as `long`. They are in fact 4-byte objects, and on some modern machines, if they are declared as `long` they will be incorrectly interpreted as 8-byte objects. For this reason, we have changed the header definitions to what is show above. Also please note that the time values `tod_minute` ... `date_year` are the local California time, not UTC.

For several years, the `extract` program contained several bugs. One of these caused the `channel.*` to have no valid header information apart from the `elapsed_time` and `datarate` entries in the binary header, and the `chunksize` entry in the main header. All the remaining entries in the main header were either incorrect or nonsensical. This bug was corrected by Allen on 14 November 1996; data files produced from the tapes after that time should have valid header information.

There was also a more serious bug in the original versions of `extract`. The typical chunksize of most slow channels is 10,000 bytes (5,000 samples) and the chunksize of most fast channels is 100,000 bytes (50,000 samples) but until it was corrected by Allen on 14 November 1996, the `extract` program would in apparently unpredictable (though actually quite deterministic) fashion “skip” the last data point from the slow channels or the last ten data points from the fast channels, giving rise to sequences of 4,999 samples from the slow channels, and correspondingly 49,990 samples from the fast channels. Not surprisingly, these missing data points gave rise to strange “gremlins” in the early data analysis work; these are described in Lyon’s thesis [32] on pages 150-151. These missing points were simply cut out of the data stream as shown in Figure 1; rather like cutting out 1 millisecond of a symphony orchestra every 5.1 seconds; this gives rise to “clicks” which excited the optimal filters. This problem is shown below; data taken off the tapes after 14 November 1996 should be free of these problems.

There are a couple of caveats regarding use of these “raw data” files. First, in the `channel.*` files, there can be, with no warning, large segments of missing data. In other words, a block of data with time stamp 13,000 sec, lasting 5 sec, can be followed by another data block with a time stamp of 14,000 sec (i.e., 995 sec of missing data). Also, the time stamps are stored in single precision floats, so that after about 10,000 sec they no longer have a resolution better than a single sample interval. When we read the data, we typically use the time-stamp on the first data segment to establish the time at which the first sample was taken. Starting from that time, we then determine the time of a data segment by using `elapsed_time`, since the millisecond time resolution of `epoch_time_msec` is not good enough. (See the comments in Section 4.1).

For our purposes, the most useful channels are `channel.0` and `channel.10`. Channel 0 contains the actual voltage output of the IFO. This is typically in the range of ± 100 . Later, we will discuss how to calibrate this signal. Channel 10 contains a TTL locked level signal, indicating if the interferometer was in lock. This is typically in the range from 1 to 10 when locked, and exceeds several hundred when the interferometer is out of lock. Note: after coming into lock you will notice that the IFO output is often zero (with a bit of DC offset) for periods ranging from a few seconds to a minute. This is because the instrument output amplifiers are typically overloaded (saturated) when the instrument is out-of-lock. Because they are AC coupled, this leads to zero output. After the instrument comes into lock, the charge on these amplifiers gradually bleeds off (or one of the operators remembers to hit the reset button) and then the output “comes alive”. So don’t be puzzled if the instrument drops into lock and the output is zero for 40 seconds afterwards!

The contents of the `channel.*` files was not the same for all of the runs. Lyon’s thesis [32] gives a chart on page 136 with some “typical” channel assignments. The channel assignments during these November 1994 data runs are listed in a log book; they were initially chosen on November 14, then changed on November 15th and again on November 18th; these assignments are shown in Table 3. (Note that the chart on page 136 of Lyon’s thesis describes the channel assignments on 15 November 94, a day when no data was taken.)

Channel Number	Description \leq 14 November 94	Description \geq 18 November 94
0	IFO output	IFO output
1	unused	magnetometer
2	unused	microphone
3	microphone	unused
4	dc strain	dc strain
5	mode cleaner pzt	mode cleaner pzt
6	seismometer	seismometer
7	unused	slow pzt
8	unused	power stabilizer
9	unused	unused
10	TTL locked	TTL locked
11	arm 1 visibility	arm 1 visibility
12	arm 2 visibility	arm 2 visibility
13	mode cleaner visibility	mode cleaner visibility
14	slow pzt	unused
15	arm 1 coil driver	arm 1 coil driver

Table 3: Channel assignments for the November 1994 data runs. Channels 0-3 are the “fast” channels, sampled at about 10 kHz; the remaining twelve are the “slow” channels, sampled at about 1KHz.

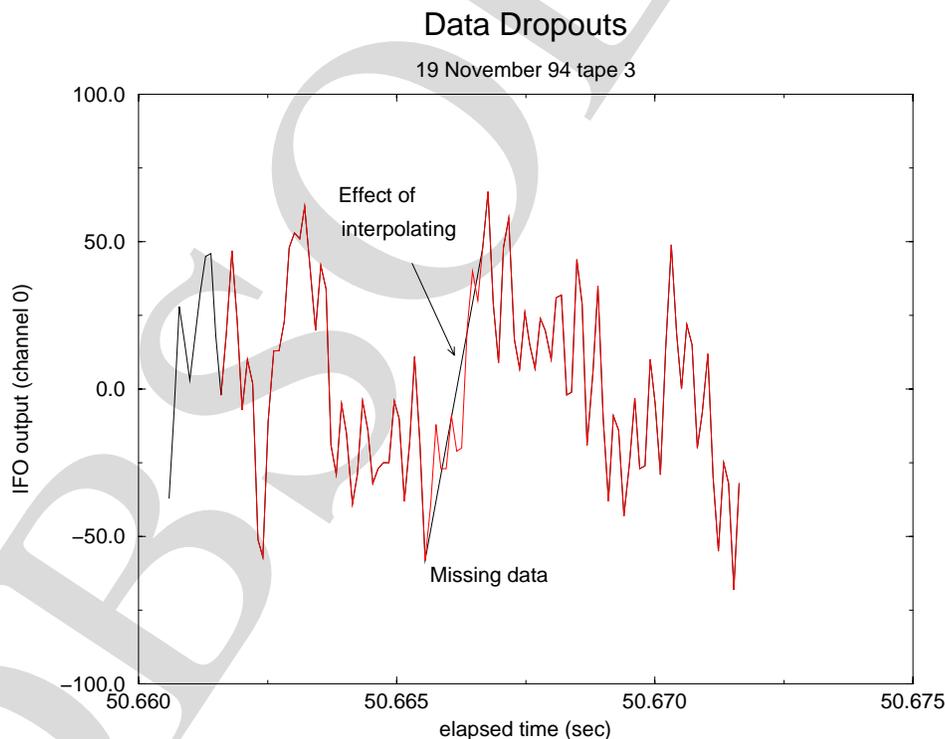


Figure 1: This shows the appearance of channel . 0 before and after the extract program was repaired (on 14 November 1996) to correctly extract data from the Exabyte data tapes. The old version of extract dropped the ten data points directly above the words “missing data”; in effect these were interpolated by the diagonal line (but with ten times the slope shown since everything in between was missing).

3.2 Function: read_block()

```
int read_block(FILE *fp, short **here, int *n, float *tstart, float *srate, int
allocate, int *nalloc, int seek, struct ld_binheader* bh, struct ld_mainheader*
mh)
```

This function efficiently reads one block of data from one of the channel . * data files, operating in sequential (not random) access. On first entry, it detects the byte-ordering of the machine that it is running on, and swaps the byte order if the machine is “little-endian” (the data was originally written in “big-endian” format, and is distributed that way). It will also print a comment (on first entry) if the machine is not big-endian.

The arguments are:

fp: Input. A pointer to the channel . * file being read.

here: Input/Output. A pointer to an array of shorts, which is where the data will be found when read_block() returns. If allocate=0, then this pointer is input. If allocate is non-zero, then this pointer is output.

n: Output. A pointer to an integer, which is the number of data items read from the block, and written to *here. These data items are typically short integers, so the number of bytes output is twice *n.

tstart: Output. The time stamp (elapsed time since beginning of the run) at the start of the data block. Taken from the binary header.

srate: Output. The sample rate, in Hz, taken from the binary header.

allocate: Input. The read_block() function will place the data that it has read in a user defined array if allocate is zero. If allocate is set, it will use malloc() to allocate a block of memory, and set *here to point to that block of memory. Further calls to read_block() will then use calls to realloc() if necessary to re-allocate the size of the block of memory, to accommodate additional data points. Note that in either case, read_block() puts into the array only the data from the next block; it over-writes any existing data in memory.

nalloc: Input/Output. If allocate is zero, then this is used to tell read_block() the size (in shorts) of the array *here. An error message will be generated by read_block() if this array is too small to accommodate the data. If allocate is nonzero, then this integer is set (and reset, if needed) to the number of array entries allocated by malloc()/realloc(). In this case, be sure that *nalloc is zero before the first call to read_block(), or the function will think that it has already allocated memory!

seek: Input. If seek is set to zero, then the function reads data. If seek is set nonzero, then read_block() does not copy any data into *here. Instead it simply skips over the actual data.

bh: Output. A pointer to the binary header structure defined above.

mh: Output. A pointer to the main header structure defined above.

This is a low-level function, which reads a block of data. It is designed to either write the data into a user-defined array or block of memory, if allocate is off, or to allocate the memory itself. In the latter mode, the function uses nalloc to track the amount of memory allocated, and reallocates if necessary. It is often useful to be able to quickly skip over sections of data (for example, just after the interferometer locks, a few minutes is needed for the violin modes to damp down). Or if the IFO is out of lock, one needs to

quickly read ahead to the next locked section. If `seek` is set, then this routine behaves exactly as it would in normal (read) mode but does not copy any data.

The function `read_block()` returns the number of data items that will be returned on the *next* call to `read_block()`. It returns -1 if it has just read the final block of data (implying that the next call will return 0). It returns 0 if it can not read any further data, because none remains.

Note that if the user has set `allocate`, then the `read_block()` will allocate memory using `malloc()/realloc()`. It is the users responsibility to free this block of memory when it is no longer needed, using `free()`.

Author: Bruce Allen, ballen@dirac.phys.uwm.edu

Comments: This function was designed for variable-length blocks. It might be possible to simplify it for fixed-length block sizes.

3.3 Example: reader program

This example uses the function `read_block()` described in the previous section to read the first 20 blocks out of the file `channel.0`. It prints the header information for each block of data, and the 100th data item from each block, along with the time associated with that data item.

The data is located with the utility function `grasp_open()`, which is documented in Section 16.2. In order for this example program to work, you *must* set the environment variable `GRASP_DATAPATH` to point to a directory containing 40-meter data. You can do this with a command such as

```
setenv GRASP_DATAPATH /usr/local/data/19nov94.3
to access the data from run 3 on November 19th.
```

```
/* GRASP: Copyright 1997,1998 Bruce Allen */
#include "grasp.h"

int main(){
    FILE *fp;
    short *data;
    float tblock,time,srate;
    int code,num,size=0,count=0,which=100;
    struct ld_binheader bheader;
    struct ld_mainheader mheader;

    /* open the IFO channel for reading */
    fp=grasp_open("GRASP_DATAPATH","channel.0","r");

    /* read the first 20 blocks of lock data */
    while (count <20) {
        /* read a block of data */
        code= read_block(fp,&data,&num,&tblock,&srate,1,&size,0,&bheader,&mheader);

        /* if there is no data left, then break */
        if (code==0) break;

        /* print some information about the data.*/
        printf("Data block %d from file channel.0 starts at t = %f sec.\n",count,tblock);
        printf("This block sampled at %f Hz and contains %d shorts.\n",srate,num);

        /* print out some information about a single data point from block */
        time=tblock+(which-1.0)/srate;
        printf("Data item %d at time %f is %d.\n",which,time,data[which-1]);
        printf("The next block of data contains %d shorts.\n\n",code);

        /* increment count of # of blocks read.*/
        count++;
    }

    /* print information about the largest memory block allocated */
    printf("The largest memory block allocated by read_block() was %d shorts long\n",size);

    /* free the array allocated by read_block() */
    free(data);
    return 0;
}
```

3.4 Function: find_locked()

```
int find_locked(FILE *fp, int *s_offset, int *s_block, int *e_offset, int *e_block, float *tstart, float *tend, float *srate)
```

This mid-level function looks in a TTL-locked signal channel (typically, `channel.10`) and finds the regions of time when the interferometer is locked. The first time it is called, it returns information identifying the start and end times of the first locked region. The second time it is called it returns the start and end times of the second locked region, and so on.

The arguments are:

`fp`: Input. A pointer to the file containing the TTL lock signal. A typical file name might be “`channel.10`”.

`s_offset`: Output. The offset (number of shorts) into the block where the IFO locks. This ranges from 0 to `n-1` where the number of data items in block `s_block` is `n`. This offset points to the first locked point.

`s_block`: Output. The number of the data block where the IFO locks. This ranges from 0 to `n-1` where the total number of data blocks in the file is `n`.

`e_offset`: Output. The offset (number of shorts) into the block where the IFO loses locks. This ranges from 0 to `n-1` where the number of data items in the block `e_block`. This offset points to the last locked point (not to the first unlocked point).

`e_block`: Output. The number of the data block where the IFO loses lock. This ranges from `s_block` to `n-1` where the total number of data blocks in the file is `n`.

`tstart`: Output. The elapsed time in seconds, since the beginning of the run, of the data block in which the first locked point was found. Note: This is not the time of lock acquisition!

`tend`: Output. The elapsed time in seconds, since the beginning of the run, of the data block in which the last locked point was found. Note: this is not the time at which lock was lost!

`srate`: Output. The sample rate of the TTL-locked channel, in Hz.

This routine uses `read_block()` to examine successive sections of the `channel.10` data file. It looks for continuous sequences of data points where the value lies between limits (inclusive) `LOCKL=1` and `LOCKH=10`. It returns the start and end points of each successive such sequence. The upper and lower limits can be changed in the code, if desired, however these values appear to be reliable ones.

The integer returned by `find_locked()` is the actual number of data points in the *fast* channels, during the locked period. It returns 0 if there are no remaining locked segments.

If there is a gap in the data stream, arising not because the instrument went out of lock, but rather because the tape-writing program was interrupted and then later restarted, `find_locked()` will print out a warning message, but will otherwise treat this simply as a loss of lock during the period of the missing data.

Author: Bruce Allen, ballen@dirac.phys.uwm.edu

Comments: This function was designed for variable-length blocks. It might be possible to simplify it for fixed-length block sizes.

3.5 Example: locklist program

This example uses the function `find_locked` described in the previous section to print out location information and times for all the locked sections in the file `channel.10`. Note that this example only prints out information for locked sections longer than 30 sec.

```
/* GRASP: Copyright 1997,1998 Bruce Allen */
#include "grasp.h"

int main() {
    float tstart,tend,srate,totalltime,begin,end;
    int start_offset,start_block,end_offset,end_block,points,zero=0;
    struct ld_mainheader mh;
    struct ld_binheader bh;
    double doubleutc;
    FILE *fplock;
    time_t unixtime;

    /* Open the file for reading */
    fplock=grasp_open("GRASP_DATAPATH","channel.10","r");

    /* print the absolute start time (in UTC) of the run */
    read_block(fplock,NULL,&zero,&tstart,&srate,0,&zero,1,&bh,&mh);
    doubleutc=mh.epoch_time_sec+0.001*mh.epoch_time_msec;
    printf("Starting time of first frame: %13f Unix-C time\n",doubleutc);
    printf("Starting time of first frame: %13f GPS time\n",doubleutc-UTCTOGPS);
    unixtime=mh.epoch_time_sec;
    printf(" ~ UTC time %s",asctime(utctime(&unixtime)));
    printf(" ~ Unix gmtime %s\n",asctime(gmtime(&unixtime)));

    /* rewind the file pointer */
    rewind(fplock);

    while (1) {

        /* find the next locked section of the data */
        points=find_locked(fplock,&start_offset,
                          &start_block,&end_offset,&end_block,&tstart,&tend,&srate);

        /* if no data remains, then exit */
        if (points==0)
            break;

        /* calculate start and end of lock times */
        begin=tstart+start_offset/srate;
        end=tend+end_offset/srate;
        totalltime=end-begin;

        /* print out info for lock intervals > 30 seconds */
        if (totalltime>30.0) {
            printf("Locked from t = %f to %f for %f sec\n",begin,end,totalltime);
            printf("Number of data points is %d\n",points);
            printf("Start block: %d End block: %d\n",start_block,end_block);
            printf("Start offset: %d End offset %d\n\n",start_offset,end_offset);
        }
    }
    return 0;
}
```

3.6 Function: `get_data()`

```
int get_data(FILE *fp, FILE *fplock, float *tstart, int npoint, short *location, int *rem, float *srate, int seek)
```

This high-level function is an easy way to get the IFO output (gravity wave signal) during periods when the IFO is locked. When called, it returns the next locked data section of a user-specified length. It also specifies if the section of data is part of a continuous locked stream, or the beginning of a new locked section.

The arguments are:

`fp`: Input. Pointer to a file (typically `channel.0`) containing the channel 0 data.

`fplock`: Input. Pointer to a file (typically `"channel.10"`) containing the TTL lock signal.

`tstart`: Output. The time of the zeroth point in the returned data.

`npoint`: Input. The number of data points requested by the user.

`location`: Input. Pointer to the location where the data should be put.

`rem`: Output. The number of points of data remaining in this locked segment of data.

`srate`: Output. The sample rate of the fast data channel, in Hz.

`seek`: Input. If this is zero, then the data is returned in the array `location[]`. However if this input is non-zero, then `get_data` performs exactly as described, except that it does not actually read any data from the file or write to `location[]`. This is useful to quickly skip over un-interesting regions of the data, for example the first several minutes after the interferometer acquires lock.

This function returns 0 if there is no remaining locked data stream of the requested length. It returns 1 if it is just starting on a new locked section of the data stream, and it returns 2 if the data is part of an on-going locked sequence.

WARNING: The `get_data()` function contains internal (static) variables which mean that you can *not* use it as follows:

1. Open a file pointer `fp`
2. Call `get_data(fp, ...)` some number of times
3. Close the file pointer `fp` and then (say) open it again
4. Call `get_data(fp, ...)` some number of times.

This sequence will leave you and the code very confused: it does not correspond to “rewinding” the file. If this is desired then you will have to modify the `get_data()` function by adding a helper “reset()” function.

Author: Bruce Allen, ballen@dirac.phys.uwm.edu

Comments: This function was designed for variable-length blocks. It is possible to simplify it for fixed-length block sizes. It should also be modified to return a complete set of different channels, by adding additional arguments to specify which channels are desired and where the data should be placed. This could also be used to eliminate the `seek` argument.

3.7 Example: gwoutput program

This example uses the function `get_data()` described in the previous section to print out a two-column file containing the IFO output for the first locked section containing 100 sample points. In the output, the left column is time values, and the right column is the actual IFO output (note that because this comes from a 12 bit A-D converter, the output is an integer value from -2047 to 2048). The program works by acquiring data 100 points at a time, then printing out the values, then acquiring 100 more points, and so on. Whenever a new locked section begins, the program prints a banner message to alert the user. Note that typical locked sections contain $\approx 10^7$ points of data, so this program should not be used for real work – it's just a demonstration!

```
/* GRASP: Copyright 1997,1998 Bruce Allen */
#include "grasp.h"

int main() {
    float tstart,time,srate;
    int remain,i,npoint,code;
    FILE *fp,*fplock;
    short *data;

    /* open the IFO output file and lock file */
    fp=grasp_open("GRASP_DATAPATH","channel.0","r");
    fplock=grasp_open("GRASP_DATAPATH","channel.10","r");

    /* specify the number of points of output & allocate array */
    npoint=100;
    data=(short *)malloc(sizeof(short)*npoint);

    while (1) {
        /* fill the array with npoint points of data */
        code=get_data(fp,fplock,&tstart,npoint,data,&remain,&srate,0);
        /* if no data remains, exit loop */
        if (code==0) break;
        /* if starting a new locked segment, print banner */
        if (code==1) {
            printf("_____ NEW LOCKED SEGMENT _____\n\n");
            printf("  Time (sec)\t\t IFO output\n");
        }
        /* now output the data */
        for (i=0;i<npoint;i++) {
            time=tstart+i/srate;
            printf("%f\t%d\n",time,data[i]);
        }
    }
    /* close the data files, and return */
    fclose(fp);
    fclose(fplock);
    return 0;
}
```

3.8 Example: animate program

This example uses the function `get_data()` described in the previous section to produce an animated display showing the time series output of the IFO in a lower window, and a simultaneously calculated FFT power spectrum in the upper window. This output from this program must be piped into a public domain graphing program called `xmgr`. This may be obtained from <http://plasma-gate.weizmann.ac.il/Xmgr/>. (This lists mirror sites in the USA and Europe also). Some sample output of `animate` is shown in Figure 2.

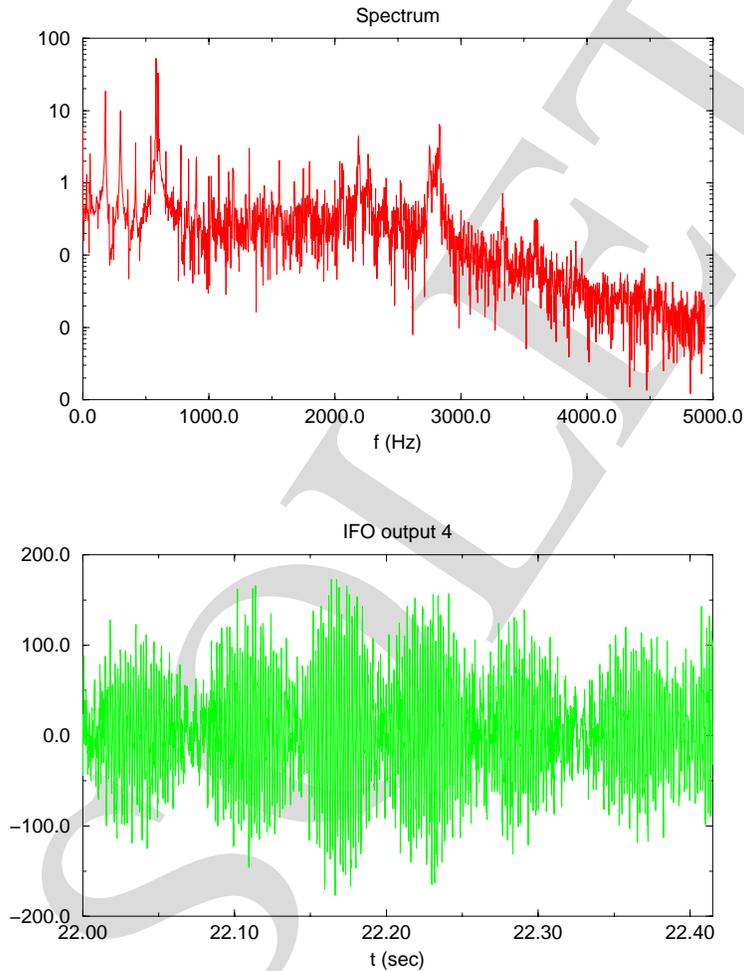


Figure 2: Snapshot of output from `animate`. This shows the (whitened) CIT 40-meter IFO a few seconds after acquiring lock, before the violin modes have damped down

After compilation, to run the program type:

```
animate | xmgr -pipe &
```

to get an animated display showing the data flowing by and the power spectrum changing, starting from the first locked data. You can also use this program with command-line arguments, for example

```
animate 100 4 500 7 900 1.5 | xmgr -pipe &
```

will show the data from time $t = 100$ to time $t = 104$ seconds, then from $t = 500$ to $t = 507$, then from $t = 900$ to $t = 901.5$. Notice that the sequence of start times must be increasing.

Note: The `xmgr` program as commonly distributed has a simple bug that needs to be repaired, in order for the frequency scale of the Fourier transform to be correct. The corrected version of `xmgr` is shown in Figure 3.

```
case 0:
==>   delt=(x[ilen-1]-x[0])/(ilen-1.0);
==>   T=(x[ilen-1]-x[0]);
      setlength(cg,specset,ilen/2);
      xx=getx(cg,specset);
...

case 1:
==>   delt=(x[ilen-1]-x[0])/(ilen-1.0);
==>   T=(x[ilen-1]-x[0]);
```

Figure 3: The corrections to a bug in the `xmgr` program are indicated by the arrows above. This bug is in the routine `do_fourier()` in the file `computils.c`. This bug has been repaired in `xmgr` version 4.1 and greater.

```
/* GRASP: Copyright 1997,1998 Bruce Allen */
#include "grasp.h"

int main(int argc,char **argv) {
  void graphout(float,float,int);
  float tstart=1.e35,srate=1.e-30,tmin,tmax,dt;
  double time=0.0;
  int remain,i,seq=0,code,npoint=4096,seek;
  FILE *fp,*fplock;
  short *data;

  /* open the IFO output file and lock file in correct path */
  fp=grasp_open("GRASP_DATAPATH","channel.0","r");
  fplock=grasp_open("GRASP_DATAPATH","channel.10","r");

  /* allocate storage space for data */
  data=(short *)malloc(sizeof(short)*npoint);
  /* handle case where user has supplied t or dt arguments */
  if (argc==1) {
    tmin=-1.e30;
    dt=2.e30;
    argc=-1;
  }
  /* now loop... */
  seq=argc;
  while (argc!=1) {
    /* get the next start time and dt */
    if (argc!=-1) {
      sscanf(argv[seq-argc+1],"%f",&tmin);
      sscanf(argv[seq-argc+2],"%f",&dt);
      argc-=2;
    }
    /* calculate the end of the observation interval, and get data */
    tmax=tmin+dt;
    while (1) {
      if (tstart<tmin-(npoint+20.)/srate) seek=1; else seek=0;
      code=get_data(fp,fplock,&tstart,npoint,data,&remain,&srate,seek);
```

```
        /* if no data left, return */
        if (code==0) return 0;
        /* we need to be outputting now... */
        if (tmin<=tstart){
            for (i=0;i<npoint;i++) {
                time=tstart+i/srate;
                printf("%f\t%d\n",time,data[i]);
            }
            /* put out information for the graphing program */
            graphout(tstart,tstart+npoint/srate,(argc==1 && time>=tmax));
        }
        /* if we are done with this interval, try next one */
        if (time>=tmax) break;
    }
}
/* close files and return */
fclose(fp);
return 0;
}
/* This routine is pipes output into the xmgr graphing program */
void graphout(float x1,float x2,int last) {
    static int count=0;
    printf("&\n"); /* end of set marker */
    /* first time we draw the plot */
    if (count==0) {
        printf("@doublebuffer true\n"); /* keeps display from flashing */
        printf("@s0 color 3\n"); /* IFO graph is green */
        printf("@view 0.1, 0.1, 0.9, 0.45\n"); /* set the viewport for IFO */
        printf("@with g1\n"); /* reset the current graph to FFT */
        printf("@view 0.1, 0.6, 0.9, 0.95\n"); /* set the viewport FFT */
        printf("@with g0\n"); /* reset the current graph to IFO */
        printf("@world xmin %f\n",x1); /* set min x */
        printf("@world xmax %f\n",x2); /* set max x */
        printf("@autoscale\n"); /* autoscale first time through */
        printf("@focus off\n"); /* turn off the focus markers */
        printf("@xaxis label \"t (sec)\" \n"); /* IFO axis label */
        printf("@fft(s0, 1)\n"); /* compute the spectrum */
        printf("@s1 color 2\n"); /* FFT is red */
        printf("@move g0.s1 to g1.s0\n"); /* move FFT to graph 1 */
        printf("@with g1\n"); /* set the focus on FFT */
        printf("@g1 type logy\n"); /* set FFT to log freq axis */
        printf("@autoscale\n"); /* autoscale FFT */
        printf("@subtitle \"Spectrum\" \n"); /* set the subtitle */
        printf("@xaxis label \"f (Hz)\" \n"); /* FFT axis label */
        printf("@with g0\n"); /* reset the current graph IFO */
        printf("@subtitle \"IFO output %d\" \n",count++); /* set IFO subtitle */
        if (!last) printf("@kill s0\n"); /* kill IFO; ready to read again */
    }
    else {
        /* other times we redraw the plot */
        printf("@s0 color 3\n"); /* set IFO green */
        printf("@fft(s0, 1)\n"); /* FFT it */
        printf("@s1 color 2\n"); /* set FFT red */
        printf("@move g0.s1 to g1.s0\n"); /* move FFT to graph 1 */
        printf("@subtitle \"IFO output %d\" \n",count++); /* set IFO subtitle */
        printf("@world xmin %f\n",x1); /* set min x */
        printf("@world xmax %f\n",x2); /* set max x */
        printf("@autoscale yaxes\n"); /* autoscale IFO */
        printf("@clear stack\n"); /* clear the stack */
    }
}
```

```
    if (!last) printf("@kill s0\n");      /* kill IFO data */
    printf("@with g1\n");                /* switch to FFT */
    printf("@g1 type logy\n");           /* set FFT to log freq axis */
    printf("@clear stack\n");           /* clear stack */
    if (!last) printf("@kill s0\n");    /* kill FFT */
    printf("@with g0\n");                /* ready to read IFO again */
}
return;
```

3.9 Function: read_sweptsine()

```
void read_sweptsine(FILE *fpss, int *n, float **freq, float **real, float **imag)
```

This is a low-level routine which reads in a 3-column ASCII file of swept sine calibration data used to calibrate the IFO.

The arguments are:

`fpss`: Input. Pointer to a file in which the swept sine data can be found. The format of this data is described below.

`n`: Output. One greater than the number of entries (lines) in the swept sine calibration file. This is because the `read_sweptsine` returns, in addition to this data, one additional entry at frequency $f = 0$.

`freq`: Output. The array `*freq[1..*n-1]` contains the frequency values from the swept sine calibration file. The routine adds an additional entry at DC, `*freq[0]=0`. Note: the routine allocates memory for the array.

`real`: Output. The array `*real[1..*n-1]` contains the real part of the response function of the IFO. The routine adds `*real[0]=0`. Note: the routine allocates memory for the array.

`imag`: Output. The array `*imag[1..*n-1]` contains the imaginary part of the response function of the IFO. The routine adds `*imag[0]=0`. Note: the routine allocates memory for the array.

The swept sine calibration files are 3-column ASCII files, of the form:

f_1	r_1	i_1
f_2	r_2	i_2
	...	
f_m	r_m	i_m

where the f_j are frequencies, in Hz, and r_j and i_j are dimensionless ratios of voltages. There are typically $m = 801$ lines in these files. Each line gives the ratio of the IFO output voltage to a calibration coil driving voltage, at a different frequency. The r_j are the “real part” of the response, i.e. the ratio of the IFO output in phase with the coil driving voltage, to the coil driving voltage. The i_j are the “imaginary part” of the response, 90 degrees out of phase with the coil driving voltage. The sign of the phase (or equivalently, the sign of the imaginary part of the response) is determined by the following convention. Suppose that the driving voltage (in volts) is

$$V_{\text{coil}} = 10 \cos(\omega t) = 10 \Re e^{i\omega t} \tag{3.9.1}$$

where $\omega = 2\pi \times 60$ radians/sec is the angular frequency of a 60 Hz signal. Suppose the response of the interferometer output to this is (again, in volts)

$$\begin{aligned} V_{\text{IFO}} &= 6.93 \cos(\omega t) + 4 \sin(\omega t) \\ &= 6.93 \cos(\omega t) - 4 \cos(\omega t + \pi/2) \\ &= 8 \Re e^{i(\omega t - \pi/6)} \end{aligned} \tag{3.9.2}$$

This is shown in Figure 4. An electrical engineer would describe this situation by saying that the phase of the response V_{IFO} is lagging the phase of the driving signal V_{coil} by 30° . The corresponding line in the swept sine calibration file would read:

	...	
60.000	0.6930	-0.40000
	...	

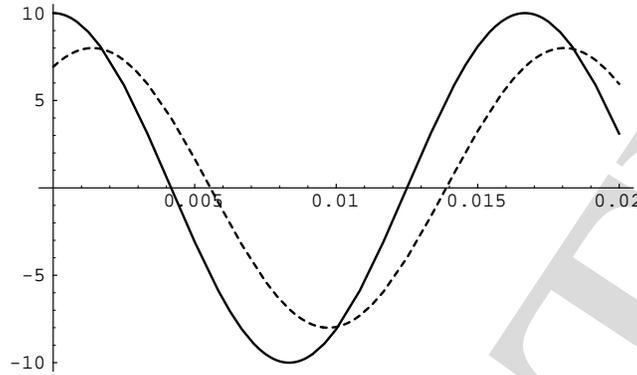


Figure 4: This shows a driving voltage V_{coil} (solid curve) and the response voltage V_{IFO} (dotted curve) as functions of time (in sec). Both are 60 Hz sinusoids; the relative amplitude and phase of the in-phase and out-of-phase components of V_{IFO} are contained in the swept-sine calibration files.

Hence, in this example, the real part is positive and the imaginary part is negative. We will denote this entry in the swept sine calibration file by $S(60) = 0.8 e^{-i\pi/6} = 0.693 - 0.400i$. Because the interferometer output is real, there is also a value implied at negative frequencies which is the complex conjugate of the positive frequency value: $S(-60) = S^*(60) = 0.8 e^{i\pi/6} = 0.693 + 0.400i$.

Because the interferometer has no DC response, it is convenient for us to add one additional point at frequency $f = 0$ into the output data arrays, with both the real and imaginary parts of the response set to zero. Hence the output arrays contain one element more than the number of lines in the input files. Note that both of these arrays are arranged in order of increasing frequency; after adding our one additional point they typically contain 802 points at frequencies from 0 Hz to 5001 Hz.

For the data runs of interest in this section (from November 1994) typically a swept sine calibration curve was taken immediately before each data tape was generated.

We will shortly address the following question. How does one use the dimensionless data in the `chan-
nel.0` files to reconstruct the differential motion $\Delta l(t)$ (in meters) of the interferometer arms? Here we address the closely related question: given V_{IFO} , how do we reconstruct V_{coil} ? We choose the sign convention for the Fourier transform which agrees with that of *Numerical Recipes*: equation (12.1.6) of [1]. The Fourier transform of a function of time $V(t)$ is

$$\tilde{V}(f) = \int e^{2\pi i f t} V(t) dt. \quad (3.9.3)$$

The inverse Fourier transform is

$$V(t) = \int e^{-2\pi i f t} \tilde{V}(f) df. \quad (3.9.4)$$

With these conventions, the signals (3.9.1) and (3.9.2) shown in in Figure 4 have Fourier components:

$$\tilde{V}_{\text{coil}}(60) = 5 \quad \text{and} \quad \tilde{V}_{\text{coil}}(-60) = 5, \quad (3.9.5)$$

$$\tilde{V}_{\text{IFO}}(60) = 4e^{i\pi/6} \quad \text{and} \quad \tilde{V}_{\text{IFO}}(-60) = 4e^{-i\pi/6}. \quad (3.9.6)$$

At frequency $f_0 = 60$ Hz the swept sine file contains

$$S(60) = 0.8 e^{-i\pi/6} \Rightarrow S(-60) = S^*(60) = 0.8 e^{i\pi/6}. \quad (3.9.7)$$

since $S(-f) = S^*(f)$.

With these choices for our conventions, one can see immediately from our example (and generalize to all frequencies) that

$$\tilde{V}_{\text{coil}}(f) = \frac{\tilde{V}_{\text{IFO}}}{S^*(f)}. \quad (3.9.8)$$

In other words, with the *Numerical Recipes* [1] conventions for forward and reverse Fourier Transforms, the (FFT of the) calibration-coil voltage is the (FFT of the) IFO-output voltage divided by the complex conjugate of the swept sine response.

Author: Bruce Allen, ballen@dirac.phys.uwm.edu

Comments: The swept-sine calibration curves are usually quite smooth but sometimes they contain a “glitch” in the vicinity of 1 kHz; this may be due to drift of the unity-gain servo point.

3.10 Function: `calibrate()`

```
void calibrate(FILE *fpss, int num, float *complex, float srate, int method, int order)
```

This is an intermediate-level routine which reads in a 3-column ASCII file of swept sine calibration data used to calibrate the IFO, and outputs an array of interpolated points suitable for calibration of FFT's of the interferometer output.

The arguments are:

`fpss`: Input. Pointer to the file in which the swept sine data can be found. The format of this data is described below.

`srate`: Input. The sample rate F_{sample} (in Hz) of the data that we are going to be calibrating.

`num`: Input. The number of points N in the FFT that we will be calibrating. This is typically $N = 2^k$ where k is an integer. In this case, the number of distinct frequency values at which a calibration is needed is $2^{k-1} + 1 = N/2 + 1$, corresponding to the number of distinct frequency values from 0 (DC) to the Nyquist frequency f_{Nyquist} . See for example equation (12.1.5) of reference [1]. The frequencies are $f_i = \frac{i}{N} F_{\text{sample}}$ for $i = 0, \dots, N/2$.

`complex`: Input. Pointer to an array `complex[0..s]` where $s = 2^k + 1$. The routine `calibrate()` fills in this array with interpolated values of the swept sine calibration data, described in the previous section. The real part of the DC response is in `complex[0]`, and the imaginary part is in `complex[1]`. The real/imaginary parts of the response at frequency f_1 are in `complex[2]` and `complex[3]` and so on. The last two elements of `complex[]` contain the real/imaginary parts of the response at the Nyquist frequency $F_{\text{sample}}/2$.

`method`: Input. This integer sets the type of interpolation used to determine the real and imaginary part of the response, at frequencies that lie in between those given in the swept sine calibration files. Rational function interpolation is used if `method=0`. Polynomial interpolation is used if `method=1`. Spline interpolation with natural boundary conditions (vanishing second derivatives at DC and the Nyquist frequency) is used if `method=2`.

`order`: Input. Ignored if spline interpolation is used. If polynomial interpolation is used, then `order` is the order of the interpolating polynomial. If rational function interpolation is used, then the numerator and denominator are both polynomials of order `order/2` if `order` is even; otherwise the degree of the denominator is `(order+1)/2` and that of the numerator is `(order-1)/2`.

The basic problem solved by this routine is that the swept sine calibration files typically contain data at a few hundred distinct frequency values. However to properly calibrate the IFO output, one usually needs this calibration information at a large number of frequencies corresponding to the distinct frequencies associated with the FFT of a data set. This routine allows you to choose different possible interpolation methods. If in doubt, we recommend spline interpolation as the first choice. The interpolation methods are described in detail in Chapter 3 of reference [1].

Author: Bruce Allen, ballen@dirac.phys.uwm.edu

Comments: It might be better to interpolate values of f^2 times the swept-sine response function, as this is the quantity needed to compute the IFO response function.

3.11 Example: print_ss program

This example uses the function `calibrate()` to read in a swept sine calibration file, and then prints out a list of frequencies, real, and imaginary parts interpolated from this data. The frequencies are appropriate for the FFT of a 4096 point data set with sample rate `srate`. The technique used is spline interpolation.

```
/* GRASP: Copyright 1997,1998 Bruce Allen */
#include "grasp.h"
#define NPOINT 4096

int main() {
    float cplx[NPOINT+2],srate,freq;
    int npoint,i;
    FILE *fp;

    /* open the swept-sine calibration file */
    fp=grasp_open("GRASP_DATAPATH","swept-sine.ascii","r");

    /* number of points of (imagined) FFT */
    npoint=NPOINT;

    /* a sample rate often used for fast channels */
    srate=9868.4208984375;

    /* swept sine calibration filename is first argument */
    calibrate(fp,npoint,cplx,srate,2,0);

    /* print out frequency, real, imaginary interpolated values */
    printf("# Freq (Hz)\tReal\tImag\n");
    for (i=0;i<=NPOINT/2;i++) {
        freq=i*srate/NPOINT;
        printf("%e\t%e\t%e\n",freq,cplx[2*i],cplx[2*i+1]);
    }
    return 0;
}
```

3.12 Function: normalize_gw()

```
void normalize_gw(FILE *fpss, int npoint, float srate, float *response)
```

This routine generates an array of complex numbers $R(f)$ from the information in the swept sine file and an overall calibration constant. Multiplying this array of complex numbers by (the FFT of) `channel.0` yields the (FFT of the) differential displacement of the interferometer arms Δl , in meters: $\widetilde{\Delta l}(f) = R(f)\widetilde{C}_0(f)$. The units of $R(f)$ are meters/ADC-count.

The arguments are:

`fpss`: Input. Pointer to the file in which the swept sine normalization data can be found.

`npoint`: Input. The number of points N of `channel.0` which will be used to calculate an FFT for normalization. Must be an integer power of 2.

`srate`: Input. The sample rate in Hz of `channel.0`.

`response`: Output. Pointer to an array `response[0..s]` with $s = N + 1$ in which $R(f)$ will be returned. By convention, $R(0) = 0$ so that `response[0]=response[1]=0`. Array elements `response[2i]` and `response[2i + 1]` contain the real and imaginary parts of $R(f)$ at frequency $f = israte/N$. The response at the Nyquist frequency `response[N]=0` and `response[N+1]=0` by convention.

The absolute normalization of the interferometer can be obtained from the information in the swept sine file, and one other normalization constant which we denote by Q . It is easy to understand how this works. In the calibration process, one of the interferometer end mirrors of mass m is driven by a magnetic coil. The equation of motion of the driven end mass is

$$m \frac{d^2}{dt^2} \Delta l = F(t) \quad (3.12.1)$$

where $F(t)$ is the driving force and Δl is the differential length of the two interferometer arms, in meters. Since the driving force $F(t)$ is proportional to the coil current and thus to the coil voltage, in frequency space this equation becomes

$$(-2\pi if)^2 \widetilde{\Delta l} = \text{constant} \times \widetilde{V}_{\text{coil}} = \text{constant} \times \frac{\widetilde{V}_{\text{IFO}}}{S^*(f)}. \quad (3.12.2)$$

We have substituted in equation (3.9.8) which relates $\widetilde{V}_{\text{IFO}}$ and $\widetilde{V}_{\text{coil}}$. The IFO voltage is directly proportional to the quantity recorded in `channel.0`: $V_{\text{IFO}} = \text{ADC} \times C_0$, with the constant ADC being the ratio of the analog-to-digital converter's input voltage to output count.

Putting together these factors, the properly normalized value of Δl , in meters, may be obtained from the information in `channel.0`, the swept sine file, and the quantities given in Table 4 by

$$\widetilde{\Delta l} = R(f) \times \widetilde{C}_0 \quad \text{with} \quad R(f) = \frac{Q \times \text{ADC}}{-4\pi^2 f^2 S^*(f)}, \quad (3.12.3)$$

where the \sim denotes Fourier transform, and f denotes frequency in Hz. (Note that, apart from the complex conjugate on S , the conventions used in the Fourier transform drop out of this equation, provided that identical conventions (3.9.3,3.9.4) are applied to both Δl and to C_0). The constant quantity Q indicated in the above equations has been calculated and documented in a series of calibration experiments carried out by Robert Spero. In these calibration experiments, the interferometer's servo was left open-loop, and the end mass was driven at a single frequency, hard enough to move the end mass one-half wavelength and shift

Table 4: Quantities entering into normalization of the IFO output.

Description	Name	Value	Units
Gravity-wave signal (channel . 0)	C_0	varies	ADC counts
A→D converter sensitivity	ADC	10/2048	$V_{\text{IFO}} (\text{ADC counts})^{-1}$
Swept sine calibration	S(f)	from file	$V_{\text{IFO}} (V_{\text{coil}})^{-1}$
Calibration constant	Q	1.428×10^{-4}	meter $\text{Hz}^2 (V_{\text{coil}})^{-1}$

the interference fringe's pattern over by one fringe. In this way, the coil voltage required to bring about a given length motion at a particular frequency was established, and from this information, the value of Q may be inferred. During the November 1994 runs the value of Q was given by

$$Q = \frac{\sqrt{9.35 \text{ Hz}}}{k} = 1.428 \times 10^{-4} \frac{\text{meter Hz}^2}{V_{\text{coil}}} \quad \text{where } k = 21399 \frac{V_{\text{coil}}}{\text{meter Hz}^{3/2}}. \quad (3.12.4)$$

Author: Bruce Allen, ballen@dirac.phys.uwm.edu

Comments: See comment for calibrate().

3.13 Example: power_spectrum program

This example uses the function `normalize_gw()` to produce a normalized, properly calibrated power spectrum of the interferometer noise, using the gravity-wave signal from `channel . 0`, the TTL-lock signal from `channel . 10` and a swept-sine calibration curve.

The output of this program is a 2-column file; the first column is frequency and the second column is the noise in units of meters/ $\sqrt{\text{Hz}}$.

A couple of comments are in order here:

1. Even though we only need the modulus, for pedagogic reasons, we explicitly calculate both the real and imaginary parts of $\widetilde{\Delta l}(f) = R(f)\widetilde{C}_0(f)$.
2. The fast Fourier transform of Δl , which we denote $\text{FFT}[\Delta l]$, has the same units (meters!) as Δl . As can be immediately seen from *Numerical Recipes* equation (12.1.6) the Fourier transform $\widetilde{\Delta l}$ has units of meters-sec and is given by $\widetilde{\Delta l} = \Delta t \text{FFT}[\Delta l]$, where Δt is the sample interval. The (one-sided) power spectrum of Δl in meters/ $\sqrt{\text{Hz}}$ is $P = \sqrt{\frac{2}{T}}|\widetilde{\Delta l}|$ where $T = N\Delta t$ is the total length of the observation interval, in seconds. Hence one has

$$P = \sqrt{\frac{2}{N\Delta t}} \Delta t |\text{FFT}[\Delta l]| = \sqrt{\frac{2\Delta t}{N}} |\text{FFT}[\Delta l]|. \quad (3.13.1)$$

This is the reason for the factor which appears in this example.

3. To get a spectrum with decent frequency resolution, the time-domain data must be windowed (see the example program `calibrate` and the function `avg_spec()` to see how this works).

A sample of the output from this program is shown in Figure 5.

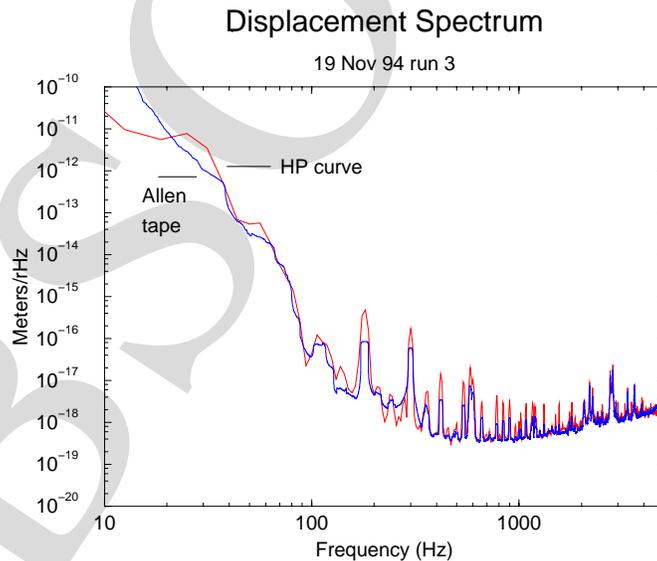


Figure 5: An example of a power spectrum curve produced with `power_spectrum`. The spectrum produced off a data tape (with 100 point smoothing) is compared to that produced by the HP spectrum analyzer in the lab.

```
#include "grasp.h"
#define NPOINT 65536

int main() {
    void realft(float*, unsigned long, int);
    float response[NPOINT+2], data[NPOINT], tstart, freq;
    float res_real, res_imag, dl_real, dl_imag, c0_real, c0_imag, spectrum, srate, factor;
    FILE *fpifo, *fplock, *fpss;
    int i, npoint, remain;
    short datas[NPOINT];

    /* open the IFO output file, lock file, and swept-sine file */
    fpifo=grasp_open("GRASP_DATAPATH", "channel.0", "r");
    fplock=grasp_open("GRASP_DATAPATH", "channel.10", "r");
    fpss=grasp_open("GRASP_DATAPATH", "swept-sine.ascii", "r");

    /* number of points to sample and fft (power of 2) */
    npoint=NPOINT;
    /* skip 200 seconds into locked region (seek=1) */
    while (tstart<200.0)
        get_data(fpifo, fplock, &tstart, npoint, datas, &remain, &srate, 1);
    /* and get next stretch of data from TTL locked file (seek=0) */
    get_data(fpifo, fplock, &tstart, npoint, datas, &remain, &srate, 0);
    /* convert gw signal (ADC counts) from shorts to floats */
    for (i=0; i<NPOINT; i++) data[i]=datas[i];
    /* FFT the data */
    realft(data-1, npoint, 1);
    /* get normalization R(f) using swept sine file */
    normalize_gw(fpss, npoint, srate, response);
    /* one-sided power-spectrum normalization, to get meters/rHz */
    factor=sqrt(2.0/(srate*npoint));
    /* compute dl. Leave off DC (i=0) or Nyquist (i=npoint/2) freq */
    for (i=1; i<npoint/2; i++) {
        /* frequency */
        freq=i*srate/npoint;
        /* real and imaginary parts of tilde c0 */
        c0_real=data[2*i];
        c0_imag=data[2*i+1];
        /* real and imaginary parts of R */
        res_real=response[2*i];
        res_imag=response[2*i+1];
        /* real and imaginary parts of tilde dl */
        dl_real=c0_real*res_real-c0_imag*res_imag;
        dl_imag=c0_real*res_imag+c0_imag*res_real;
        /* |tilde dl| */
        spectrum=factor*sqrt(dl_real*dl_real+dl_imag*dl_imag);
        /* output freq in Hz, noise power in meters/rHz */
        printf("%e\t%e\n", freq, spectrum);
    }
    return 0;
}
```

Author: Bruce Allen, ballen@dirac.phys.uwm.edu

Comments: The IFO output typically consists of a number of strong line sources (harmonics of the 60 Hz line and the 180 Hz laser power supply, violin modes of the suspension, etc) superposed on a continuum background (electronics noise, laser shot noise, etc) In such situations, there are better

ways of finding the noise power spectrum (for example, see the multi-taper methods of David J. Thomson [39], or the textbook by Percival and Walden [40]). Using methods such as the F-test to remove line features from the time-domain data stream might reduce the sidelobe contamination (bias) from nearby frequency bins, and thus permit an effective reduction of instrument noise near these spectral line features. Further details of these methods, and some routines that implement them, may be found in Section 16.19.

3.14 Example: `calibrate` program

This example uses the function `normalize_gw()` and `avg_spec()` to produce an animated display, showing the properly normalized power spectrum of the interferometer, with a 30-second characteristic time moving average. After compilation, to run the program type:

```
calibrate | xmgr -pipe &
```

to get an animated display showing the calibrated power spectrum changing. An example of the output from `calibrate` is shown in Figure 6. Note that most of the execution time here is spent passing data down the pipe to `xmgr` and displaying it. The display can be speeded up by a factor of ten by binning the output values to reduce their number to a few hundred lines (the example program `calibrate_binned.c` implements this technique; it can be run by typing `calibrate_binned | xmgr -pipe`).

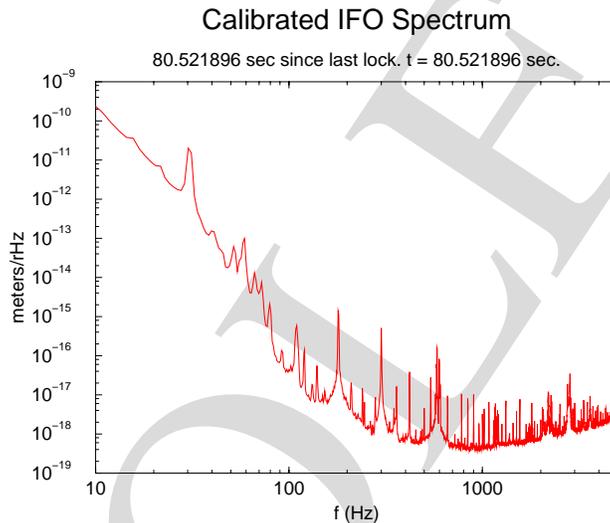


Figure 6: This shows a snapshot of the output from the program `calibrate` which displays an animated average power spectrum (Welch windowed, 30-second decay time).

```
/* GRASP: Copyright 1997,1998 Bruce Allen */
#include "grasp.h"
#define NPOINT 4096

int main(int argc, char **argv) {
    void graphout(int, float, float);
    void realft(float*, unsigned long, int);
    float data[NPOINT], average[NPOINT/2], response[NPOINT+4];
    float spec, decaytime;
    float srate, tstart=0, freq, tlock=0.0;
    FILE *fpifo, *fpss, *fplock;
    int i, j, code, npoint, remain, ir, ii, reset=0, pass=0;
    short datas[NPOINT];
    double mod;

    /* open the IFO output file, lock file, and swept-sine file */
    fpifo=grasp_open("GRASP_DATAPATH", "channel.0", "r");
    fplock=grasp_open("GRASP_DATAPATH", "channel.10", "r");
    fpss=grasp_open("GRASP_DATAPATH", "swept-sine.ascii", "r");
}
```

```
/* number of points to sample and fft (power of 2) */
npoint=NPOINT;

/* set the decay time (sec) */
decaytime=30.0;
/* get data */
while ((code=get_data(fpifo,fplock,&tstart,npoint,datas,&remain,&srates,0)) {
    /* put data into floats */
    for (i=0;i<npoint;i++) data[i]=datas[i];
    /* get the normalization */
    if (!pass++)
        normalize_gw(fpss,npoint,srates,response);
    /* Reset if just locked */
    if (code==1) {
        reset=0;
        tlock=tstart;
        avg_spec(data,average,npoint,&reset,srates,decaytime,2,1);
    } else {
        /* track average power spectrum, with Welch windowing. */
        avg_spec(data,average,npoint,&reset,srates,decaytime,2,1);
        /* loop over all frequencies except DC (j=0) & Nyquist (j=npoint/2) */
        for (j=1;j<npoint/2;j++) {
            /* subscripts of real, imaginary parts */
            ii=(ir=j+j)+1;
            /* frequency of the point */
            freq=srates*j/npoint;
            /* determine power spectrum in (meters/rHz) & print it */
            mod=response[ir]*response[ir]+response[ii]*response[ii];
            spec=sqrt(average[j]*mod);
            printf("%e\t%e\n",freq,spec);
        }
        /* print out useful things for xmgr program ... */
        graphout(0,tstart,tlock);
    }
}
return 0;
}

void graphout(int last,float time,float tlock) {
    static int count=0;
    printf("&\n"); /* end of set marker */
    /* first time we draw the plot */
    if (count++==0) {
        printf("@doublebuffer true\n"); /* keeps display from flashing */
        printf("@focus off\n"); /* turn off the focus markers */
        printf("@s0 color 2\n"); /* FFT is red */
        printf("@g0 type logxy\n"); /* set graph type to log-log */
        printf("@autoscale \n"); /* autoscale FFT */
        printf("@world xmin %e\n",10.0); /* set min x */
        printf("@world xmax %e\n",5000.0); /* set max x */
        printf("@world ymin %e\n",1.e-19); /* set min y */
        printf("@world ymax %e\n",1.e-9); /* set max y */
        printf("@yaxis tick minor on\n"); /* turn on tick marks */
        printf("@yaxis tick major on\n"); /* turn on tick marks */
        printf("@yaxis tick minor 2\n"); /* turn on tick marks */
        printf("@yaxis tick major 1\n"); /* turn on tick marks */
        printf("@redraw \n"); /* redraw graph */
        printf("@xaxis label \"f (Hz)\"\n"); /* FFT horizontal axis label */
    }
}
```

```
printf("@yaxis label \"meters/rHz\"\n"); /* FFT vertical axis label */
printf("@title \"Calibrated IFO Spectrum\"\n"); /* set title */
/* set subtitle */
printf("@subtitle \"%.2f sec since last lock. t = %.2f sec.\"\n",time-tlock,time);
if (!last) printf("@kill s0\n"); /* kill graph; ready to read again */
}
else {
/* other times we redraw the plot */
/* set subtitle */
printf("@subtitle \"%.2f sec since last lock. t = %.2f sec.\"\n",time-tlock,time);
printf("@s0 color 2\n"); /* FFT is red */
printf("@g0 type logxy\n"); /* set graph type to log-log */
printf("@world xmin %e\n",10.0); /* set min x */
printf("@world xmax %e\n",5000.0); /* set max x */
printf("@world ymin %e\n",1.e-19); /* set min y */
printf("@world ymax %e\n",1.e-9); /* set max y */
printf("@yaxis tick minor on\n"); /* turn on tick marks */
printf("@yaxis tick major on\n"); /* turn on tick marks */
printf("@yaxis tick minor 2\n"); /* turn on tick marks */
printf("@yaxis tick major 1\n"); /* turn on tick marks */
printf("@redraw\n"); /* redraw the graph */
if (!last) printf("@kill s0\n"); /* kill graph, ready to read again */
}
return;
}
```

Author: Bruce Allen, ballen@dirac.phys.uwm.edu

Comments: See comments for power_spectrum example program.

3.15 Example: transfer program

This example uses the function `normalize_gw()` to calculate the response of the interferometer to a specified gravitational-wave strain $h(t)$. [Note: for clarity, in this example, we have NOT worried about getting the overall normalization correct.] The code includes two possible $h(t)$'s. The first of these is a binary-inspiral chirp (see Section 6). Or, if you un-comment one line of code, you can see the response of the detector to a unit-impulse gravitational wave strain, in other words, the impulse response of the detector.

Note that to run this program, you must specify a path to the 40-meter data, for example by typing:

```
setenv GRASP_DATAPATH /usr/local/data/19nov94.3
```

so that the code can find a swept-sine calibration file to use.

The response of the detector to a pair of inspiraling stars is shown in Figure 7. You will notice that although the chirp starts at a (gravitational-wave) frequency of 140 Hz on the left-hand side of the figure, the low-frequency response of the detector is so poor that the chirp does not really become visible until about half-a-second later, at somewhat higher frequency. In the language of the audiophile, the IFO has crummy bass response! Of course this is entirely deliberate; the whitening filters of the instrument are designed to attenuate the low-frequency seismic contamination, and consequently also attenuate any possible low-frequency gravitational waves.

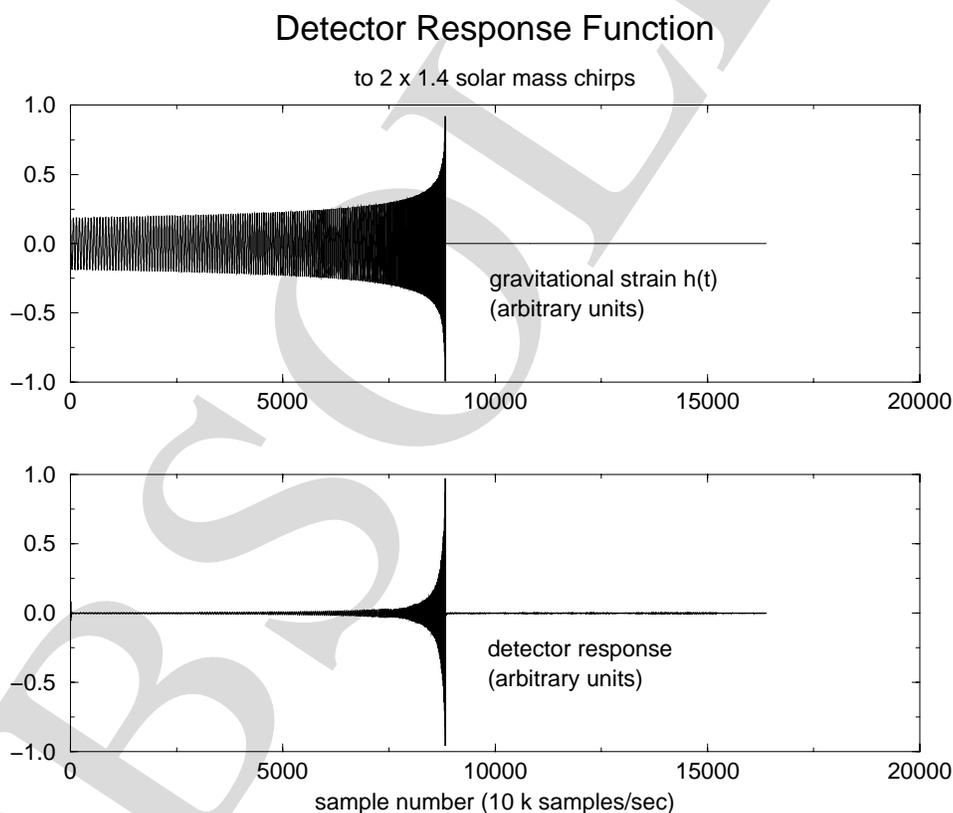


Figure 7: Output produced by the `transfer` example program. The top graph shows the gravitational-wave strain produced by an inspiraling binary pair. The lower graph shows the calculated interferometer output [channel.0 or IFO_DMRO] produced by this signal. Notice that because of the poor low-frequency response of the instrument, the IFO output does not show significant response before the input frequency has increased. The sample rate is slightly under 10 kHz.

The response of the detector to a unit gravitational strain impulse is shown as a function of time-offset

in Figure 8. Here the predominant effect is the ringing of the anti-aliasing filter. The impulse response of the detector lasts about 30 samples, or 3 msec. For negative offset times the impulse response is quite close to zero; its failure to vanish is partly a wrap-around effect, and partly due to errors in the actual measurement of the transfer function.

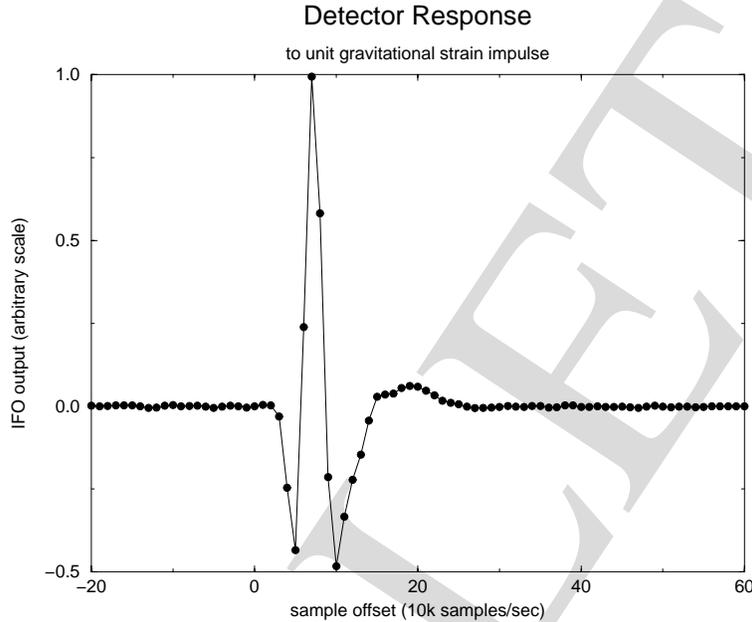


Figure 8: Output produced by the `transfer` example program. This shows the calculated interferometer output [channel.0 or IFO_DMRO] produced by an impulse in the gravitational-wave strain at sample number zero. This (almost) causal impulse response lasts about 3 msec.

This is a good place to insert a cautionary note. Now that we have determined the transfer function $R(f)$ of the instrument, you might be tempted to ask: “Why should I do any of my analysis in terms of the instrument output? After all, my real interest is in gravitational waves. So the first thing that I will do in my analysis is convert the instrument output into a gravitational wave strain $h(t)$ at the detector, by convolving the instrument’s output with (the time-domain version of) $R(f)$.” **Please do not make this mistake!** A few moment’s reflection will show why this is a remarkably bad idea. The problem is that the response function $R(f)$ is extremely large at low frequencies. This is just a reflection of the poor low frequency response of the instrument: any low-frequency energy in the IFO output corresponds to an extremely large amplitude low frequency gravitational wave. So, if you calculate $h(t)$ in the way described: take a stretch of (perhaps zero-padded) data, FFT it into the frequency domain, multiply it by $R(f)$ and invert the FFT to take it back into the frequency domain, you will discover the following:

- Your $h(t)$ is dominated by a single low-frequency noisy sinusoid (whose frequency is determined by the low frequency cutoff imposed by the length of your data segment or the low-frequency cutoff of the response function).
- Your $h(t)$ has *lost* all the interesting information present at frequencies where the detector is quiet (say, around 600 Hz). Because the noise power spectrum (see Figure 5) covers such a large dynamic range, you can not even represent $h(t)$ in a floating point variable (though it will fit, though barely, into a double). This is why the instrument uses a whitening filter in the first place.

- It is possible to construct " $h(t)$ " if you filter out the low-frequency garbage by setting $R(f)$ to zero below (say) 100 Hz.

If you are unconvinced by this, do the following exercise: calculate the power spectrum in the frequency domain as was done with Figure 5, then construct $h(t)$ in time domain, then take $h(t)$ back into the frequency domain, and graph the power spectrum again. You will discover that it has completely changed above 100 Hz and is entirely dominated by numerical quantization noise (round-off errors).

```
/* GRASP: Copyright 1997,1998 Bruce Allen */
#include <stdio.h>
#include <memory.h>
#include "grasp.h"
#define HSCALE 1.e20
#define NBINS 16384

int main() {
    float fstart,srate,tcoal,*c0,*c90,*response;
    int filled,i;
    void realft(float*,unsigned long, int);
    FILE *fp;

    /* allocate memory */
    c0=(float*)malloc(sizeof(float)*NBINS);
    c90=(float*)malloc(sizeof(float)*NBINS);
    response=(float*)malloc(sizeof(float)*(NBINS+1));

    /* set start frequency, sample rate, make chirp */
    make_filters(1.4,1.4,c0,c90,fstart=140.0,NBINS,srate=9868.0,&filled,&tcoal,4000,4);
    printf("Chirp length is %d.\n",filled);

    /* Uncomment this line to see the impulse response of the instrument */
    /* for (i=0;i<NBINS;i++) c0[i]=0.0; c0[100]=1.0; */

    /* put chirps into frequency domain */
    realft(c0-1,NBINS,1);

    /* open file containing calibration data, get response, and scale */
    fp=grasp_open("GRASP_DATAPATH","swept-sine.ascii","r");
    normalize_gw(fp,NBINS,srate,response);
    for (i=0;i<NBINS;i++) response[i]*=HSCALE;

    /* avoid floating point errors in inversion */
    response[0]=response[1]=1.e10;

    /* determine IFO channel0 input which would have produced waveform */
    ratio(c0,c0,response,NBINS/2);

    /* invert FFT */
    realft(c0-1,NBINS,-1);

    /* make a graph showing channel.0 */
    printf("File temp.graph contains channel.0 produced by 2 x 1.4 solar masses.\n");
    graph(c0,NBINS,1);

    return 0;
}
```

Author: Bruce Allen, ballen@dirac.phys.uwm.edu

Comments: None.

3.16 Example: diag program

This program is a frequency-domain “novelty detector” and provides a simple example of a time-frequency diagnostic method. The actual code is not printed here, but may be found in the GRASP directory `src/examples/example` in the file `diag.c`.

The method used by `diag` is as follows:

1. A buffer is loaded with a short stretch of data samples (2048 in this example, about 1/5 of a second).
2. A (Welch-windowed) power spectrum is calculated from the data in the buffer. In each frequency bin, this provides a value $S(f)$.
3. Using the same auto-regressive averaging technique described in `avg_spec()` the mean value of $S(f)$ is maintained in a time-averaged spectrum $\langle S(f) \rangle$. The exponential-decay time constant for this average is `AVG_TIME` (10 seconds, in this example).
4. The absolute difference between the current spectrum and the average $\Delta S(f) \equiv |S(f) - \langle S(f) \rangle|$ is determined. Note that the absolute value used here provides a more robust first-order statistic than would be provided by a standard variance $(\Delta S(f))^2$.
5. Using the same auto-regressive averaging technique described in `avg_spec()` the value of $\Delta S(f)$ is maintained in a time-averaged absolute difference $\langle \Delta S(f) \rangle$. The exponential-decay time constant for this average is also set by `AVG_TIME`.
6. In each frequency bin, $\Delta S(f)$ is compared to $\langle \Delta S(f) \rangle$. If $\Delta S(f) > \text{THRESHOLD} \times \langle \Delta S(f) \rangle$ then a point is plotted for that frequency bin; otherwise no point is plotted for that frequency bin. In this example, `THRESHOLD` is set to 6.
7. In each frequency bin, $\Delta S(f)$ is compared to $\langle \Delta S(f) \rangle$. If $\Delta S(f) < \text{INCLUDE} \times \langle \Delta S(f) \rangle$ then the values of $S(f)$ and $\Delta S(f)$ are used to “refine” or “revise” the auto-regressive means described previously. In this example, `INCLUDE` is set to 10.
8. Another set of points (1024 in this example) is loaded into the end of the buffer, pushing out the oldest 1024 points from the start of the buffer, and the whole loop is restarted at step 2 above.

The `diag` program can be used to analyze any of the different channels of fast-sampled data, by setting `CHANNEL` appropriately. It creates one output file for each locked segment of data. For example if `CHANNEL` is set to 0 (the IFO channel) and there are four locked sections of data, one obtains a set of files:

`ch0diag.000`, `ch0diag.001`, `ch0diag.002`, and `ch0diag.003`.

In similar fashion, if `CHANNEL` is set to 1 (the magnetometer) one obtains files:

`ch1diag.000`, `ch1diag.001`, `ch1diag.002`, and `ch1diag.003`.

These files may be used as input to the `xmgr` graphing program, by typing:

```
xmgr ch0diag.000 ch1diag.000
```

(one may specify as many channels as desired on the input line). A typical pair of outputs is shown in Figures 9 and 10. By specifying several different channels on the command line for starting `xmgr`, you can overlay the different channels output with one another. This provides a visual tool for identifying correlations between the channels (the graphs shown below may be overlaid in different colors).

Author: Bruce Allen, ballen@dirac.phys.uwm.edu

Comments: This type of time-frequency event detector appears quite useful as a diagnostic tool. It might be possible to improve its high-frequency time resolution by being clever about using intermediate

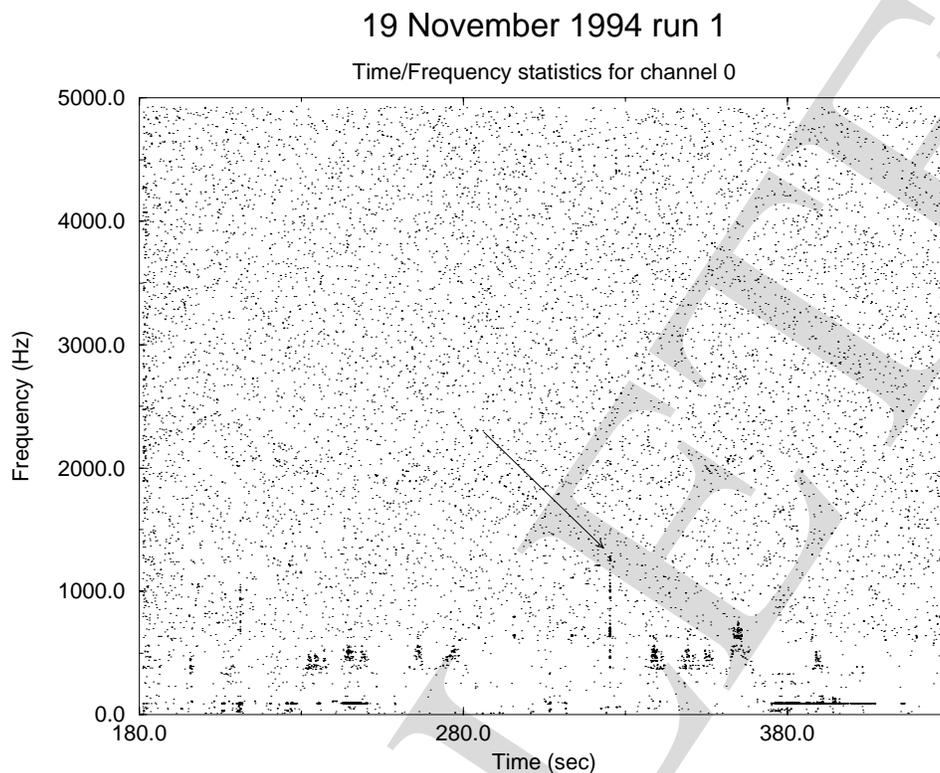


Figure 9: A time-frequency diagnostic graph produced by `diag`. The vertical line pointed to by the arrow is a non-stationary noise event in the IFO output, 325 seconds into the locked section. It sounds like a “drip” and might be due to off-axis modes in the interferometer optical cavities.

information during the recursive calculation of the FFT. One should probably also experiment with using other statistical measures to assess the behavior of the different frequency bins. It would be nice to modify this program to also examine the slow sampled channels (see comment for `get_data()`).

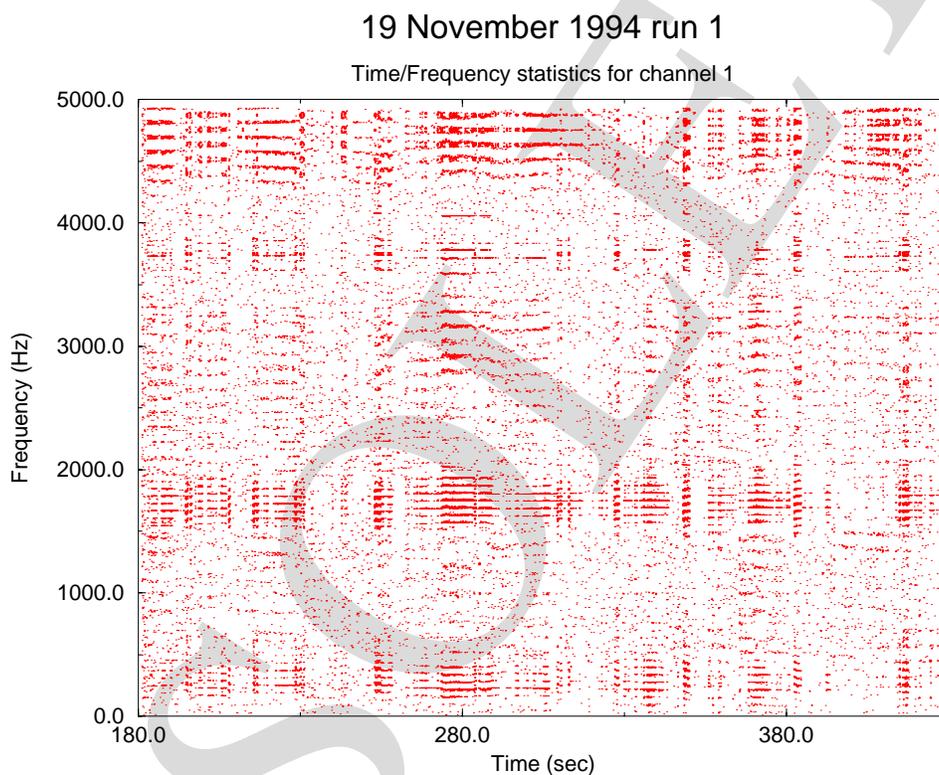


Figure 10: A time-frequency diagnostic graph produced by `diag`. This shows the identical period as the previous graph, but for the magnetometer output. Notice that the spurious event was not caused by magnetic field fluctuations.

4 GRASP Routines: Reading/using FRAME format data

The LIGO and VIRGO projects have recently adopted a data format standard called the FRAME format for time-domain data. The 40-meter laboratory at Caltech implemented this data format in Spring 1997; data taken after that time is in the FRAME format. The FRAME libraries are publicly available from the VIRGO project; they may be downloaded from the site <http://wwwlapp.in2p3.fr/virgo/FrameL>. The GRASP package has been tested up to release 3.72 of the frame library. Contact Benoit Mours mours@lapp.in2p3.fr for further information.

The GRASP package includes routines for reading and using data in the FRAME format. Also included in the GRASP package is a translator (see Section 16.18) which translates data from the old data format used in 1994 to the new FRAME format. Data distributed for use with GRASP will primarily be distributed in this new FRAME format, and over a period of time we will remove from the GRASP package all of the code and routines which make use of the old format. In order to help make the transition from old format to FRAME format as smooth as possible, the GRASP package currently contains both old format and FRAME format versions of all of the example programs. For example `animate` and `animateF` are two versions of the same program. The first reads data in the old format, the second reads data in the FRAME format. If you are new to GRASP, we don't recommend that you waste your time with the old data format; start using the FRAME format immediately.

Data distributed in the FRAME format may not be compatible with future releases of the FRAME library, so if the FRAME libraries are updated you may need to obtain a new copy of the standard 40-meter test data set from November 1994. The data that has been distributed and is currently being distributed makes use of either version 2.20, 2.30 or 2.37 of the FRAME library. We will shortly begin distributing data in version 3.50 of the FRAME format. Only two files in the GRASP package (`src/utility/frameinterface.c` and `src/examples/examples_utility/translate.c`) depend upon the version of the FRAME library. We distribute GRASP with versions of these files appropriate for different releases. The files determine the version of the frame library at compilation time, and then include the appropriate code. This code works correctly with any version of the frame library $2.37 \leq \text{version} \leq 3.70$. Note that version ≥ 3.50 of the frame library can read data written by any version back to and including 2.37.

One of the nice properties of the FRAME formats ≥ 3.30 is that they support a "compressed" format. This is transparent to the user (except that reading the "compressed" frames takes a bit longer because the frame library then needs to uncompress the data). Data distributed in version 3.50 of the FRAME format is being distributed in this compressed form and occupies somewhat less space than the old-format original data. As shown in Section 3 the old-format data for the November 1994 runs occupied about 13.6 Gbytes. For comparison, the FRAME-format data occupies less than half of that space:

<code>14nov94.1.frame</code>	314	
<code>14nov94.2.frame</code>	397	
<code>18nov94.1.frame</code>	503	
<code>18nov94.2.frame</code>	543	
<code>19nov94.1.frame</code>	551	The space occupied
<code>19nov94.2.frame</code>	535	is shown in Mbytes
<code>19nov94.3.frame</code>	641	
<code>19nov94.4.frame</code>	605	
<code>20nov94.1.frame</code>	553	
<code>20nov94.2.frame</code>	422	
<code>20nov94.3.frame</code>	755	

The total storage space required for FRAME 3.50 data totals only 5.8 Gbytes.

In order to give the 1994 40-meter data a form as similar as possible to the data being taken in 1997 and beyond, the channel names used have been given equivalent "FRAME" forms. These are shown in Table 5.

Note that new data created in the frame format attempts to address at least a couple of the problems in the "old format" data. In particular, new frame format data (i.e., post 1996) has sample rate in Hz always being powers of 2, for example, 4,096 Hz or 16 Hz or 16,384 Hz. In addition, each frame always contains a power-of-two number of seconds of data. These conventions will make it easy to "match up" sample of channels taken at different rates, and to do FFT's of the channels. However the 1994 data does not conform to either of these conventions: each frame of 1994 data contains 5000 samples of the slow channels, and 50,000 samples of the fast channels, during a 5.06666... second interval.

Channel #	≤ 14 Nov 94	FRAME name	≥ 18 Nov 94	FRAME name
0	IFO output	IFO_DMRO	IFO output	IFO_DMRO
1	unused		magnetometer	IFO_Mag_x
2	unused		microphone	IFO_Mike
3	microphone	IFO_Mike	unused	
4	dc strain	IFO_DCDM	dc strain	IFO_DCDM
5	mode cleaner pzt	PSL_MC_V	mode cleaner pzt	PSL_MC_V
6	seismometer	IFO_Seis_1	seismometer	IFO_Seis_1
7	unused		slow pzt	IFO_SPZT
8	unused		power stabilizer	PSL_PSS
9	unused		unused	
10	TTL locked	IFO_Lock	TTL locked	IFO_Lock
11	arm 1 visibility	IFO_EAT	arm 1 visibility	IFO_EAT
12	arm 2 visibility	IFO_SAT	arm 2 visibility	IFO_SAT
13	mode cleaner visibility	IFO_MCR	mode cleaner visibility	IFO_MCR
14	slow pzt	IFO_SPZT	unused	
15	arm 1 coil driver	SUS_EE_Coil_V	arm 1 coil driver	SUS_EE_Coil_V

Note: On 18 November 1994 run 1 the power stabilizer was accidentally disconnected until approximately 20:00 local time.

Table 5: Channel assignments for the November 1994 data runs. Channels 0-3 are the "fast" channels, sampled at about 10 kHz; the remaining twelve are the "slow" channels, sampled at about 1KHz. The equivalent "FRAME" format names are also given.

4.1 Time-stamps in the November 1994 data-set

There is a serious problem in the original data format used in November 1994. To understand the nature of this problem, remember that the individual data samples (fast channels) are taken at about 10kHz, so that the time between samples is about 100 μ sec. Ideally, the time-stamps of the individual blocks should be recorded with a precision which is substantially greater than this, i.e. a few μ sec at the most. However the November 1994 time stamps are recorded in two ways: as an integer number of seconds and msec (with 1000 μ sec resolution) and as a floating point elapsed time. This latter quantity has a resolution of less than one μ sec at early times, but a resolution of about 2000 μ sec at late times (say 15,000 sec into a run).

Thus, in translating the November 1994 data into frames (which have 1 nanosec resolution time-stamps), a reasonable effort was made to “correct” these time-stamps as much as possible, and to specify the time at which each data block begins as precisely as possible. After some research, we believe that the each block of old-format data is precisely $76/15 = 5.0666666 \dots$ seconds long. So we have corrected the time stamps accordingly. One can show that in general, our time stamps agree with those in the original data, when they are expressed as floats, i.e. with the precision recorded in the original data set. There are some blocks where there is an error in the least-significant bit of the cast-into-float quantity; we do not understand this as well as we would like.

Please, *be warned that the absolute time indicated by these stamps is not correct!* These time stamps were not taken with a modern GPS clock system, or even with an old-fashioned WWV system. Our understanding is that the real-time computer system on which these data were originally taken had its clock set by wristwatch, with an accuracy of perhaps ± 5 minutes.. Indeed the computer system crashed on November 15, 1994 and the clock was subsequently reset again, so even the time difference can not be trusted between November14 and November18 data. It appears that the computer clock was not reset after November15th, so the relative times in the remaining data may be trustworthy with somewhat better than ± 1 msec accuracy.

In any data analysis work (such as pulsar searching) where it is important to have precise time-stamps, these shortcomings must be taken into account. If you really want to determine the times more precisely than a millisecond, our only suggestion is to examine the seismometer data channel and correlate it with similar data taken by a system with good time-stamps. We don't know where to find such data, but it might exist, somewhere, in the public domain. If you do go to this trouble, please write to us and tell us the conclusions of your study. We would be delighted to correct the absolute offset error in these November 1994 time-stamps, if someone could show us how to do it!

4.2 Function: `fget_ch()`

```
int fget_ch(struct fgetoutput *fgetoutput, struct fgetinput *fgetinput)
```

This is a general function for sequentially reading one or more channels of FRAME format data. It can be used to obtain either locked sections only, or both locked and unlocked sections, and to retrieve calibration information from the FRAME data. It concatenates multiple frames and multiple files containing frames as necessary, to return continuous-in-time sequences.

The inputs to the routine `fget_ch()` are contained in a structure:

```
struct fgetinput {
    int nchan;
    char **chnames;
    int npoint;
    short **locations;
    char *(*files)();
    int (*filedes)();
    int inlock;
    int seek;
    int calibrate;
    char *datatype;
};
```

The different elements of the structure are:

`nchan`: Input. The number of channels that you want to retrieve (≥ 1).

`chnames`: Input. The list of channel names. Each element of `chnames[0..nchan-1]` is a pointer to a null-terminated string. Note that the number of channels requested, and their names, must not be changed after the first call to `fget_ch`. It is assumed that the first channel in the list has the fastest sample rate of any of the requested channels. As long as this assumption is satisfied, the channels may be accessed in any order.

`npoint`: Input. The number of points requested from the first channel. (May change with each call.)

`locations`: Input. The locations in memory where the arrays corresponding to each channel should be placed are `locations[0..nchan-1]`. (May change with each call.)

`files()`: Input. A pointer to a function, which takes no arguments, and returns a pointer to a null-terminated character string. This string is the name of the file to look in for FRAME format data. If no further frames remain in the file, then the function `files()` is called again. When this function returns a null pointer, it is assumed that no further data remains. A useful utility function called `framefiles()` has been provided with GRASP, and may be used as this argument. (May change with each call.)

`filedes()`: Input. This argument is used if and only if the previous argument, `fgetinput.files` is NULL. If `fgetinput.files` is not NULL then this argument is not used. This argument is a pointer to a function, which takes no arguments, and returns an integer *file descriptor*. The integer returned is a file descriptor for a file containing FRAME format data. If no further frames remain in the file, then the function `filedes()` is called again. When this function returns a negative file descriptor, it is assumed that no further data remains. (May change with each call.)

`inlock`: Input. Set to zero, return all data; set to non-zero, return only the locked sections of data. If set nonzero, then on output `fgetoutput.locklow` and `fgetoutput.lockhi` will be set.

`seek`: Input. Set to zero, return data. Set to non-zero, seek past the data, performing all normal operations, but do not actually write any data into the arrays pointed to by `locations[0..nchan-1]`. (May change with each call.) This is useful for skipping rapidly past uninteresting regions of data, for example, the first few minutes after coming into lock.

`calibrate`: Input. If set non-zero, return calibration information. If set to zero, do not return calibration information. (May change with each call.)

`datatype`: Output. A character string indicating the data type in each channel. The coding is: C = char, S = short, D = double, F = float, I = int, L = long, f = complex float, d = complex double, s = string, u = unsigned short, i = unsigned int, l = unsigned long, c = unsigned char.

Except as noted above, it is assumed that none of these input arguments are changed after the first call to `fget_ch()`. It is also assumed that within any given frame, the numbers of points contained in different channels are exact integer multiples or fractions of the numbers of points contained in the other channels.

The outputs from the routine `fget_ch()` are contained in a structure:

```
struct fgetoutput {
    double tstart;
    double tstart_gps;
    double srate;
    int *npoint;
    int *ratios;
    int discarded;
    double tfirst;
    double tfirst_gps;
    double dt;
    double lostlock;
    double lostlock_gps;
    double lastlock;
    double lastlock_gps;
    int returnval;
    int frinum;
    float *fri;
    int tcalibrate;
    int tcalibrate_gps;
    int locklow;
    int lockhi;
    char *filename;
    char *slow_names;
};
```

The different elements of the structure are:

`tstart`: Output. Time stamp of the first point output in channel `chnames[0]`. Note: please see the comments in Section 4.1. Units are Unix-C time in seconds defined in Section 17.

`tstart_gps`: Output. Same as previous quantity, but with GPS time in seconds.

`srate`: Output. Sample rate (in Hz) of channel `chnames[0]`.

`npoint`: Output. The number of points returned in channel `chnames[i]` is `npoint[i]`. Note that `npoint[0]` is precisely the number of points requested in the input structure `fgetinput.npoint`.

`ratios`: Output. The sample rate of channel `chnames[0]` divided by the sample rate of channel `chnames[i]` is given in `ratios[i]`. Thus `ratios[0]=1`.

`discarded`: The number of points discarded from channel `chnames[0]`. These points are discarded because there is a missing period of time between two consecutive frames, or because the instrument was not in lock for long enough to return the requested number of points (or for both reasons).

`tfirst`: Output. The time stamp of the first point returned in the first call to `fget_ch()`. Units are Unix-C time in seconds defined in Section 17.

`tfirst_gps`: Output. Same as previous quantity, but with GPS time in seconds.

`dt`: Output. By definition, `tstart-tfirst`, which is the elapsed time since the first time stamp.

`lostlock`: Output. The time at which we last lost lock (if searching only for locked segments). Units are Unix-C time in seconds defined in Section 17.

`lostlock_gps`: Output. Same as previous quantity, but with GPS time in seconds.

`lastlock`: Output. The time at which we last regained lock (if searching only for locked segments). Units are Unix-C time in seconds defined in Section 17.

`lastlock_gps`: Output. Same as previous quantity, but with GPS time in seconds.

`returnval`: Output. The return value of `fget_ch()`: 0 if it is unable to satisfy the request, 1 if the request has been satisfied by beginning a new locked or continuous-in-time section, and 2 if the data returned is part of an ongoing locked or continuous-in-time sequence.

`frinum`: Output. Three times the number of frequency values for which we are returning static calibration information. If this number is not divisible by three, something is wrong!

`fri`: Output. A pointer to the array of calibration data. This data is arranged with a frequency, then the real part, then the imaginary part of the response, followed by another frequency, then real part, then imaginary part, etc. So `fri[0]=f0`, `fri[1]=r0`, `fri[2]=i0`, `fri[3]=f1`, `fri[4]=r1`, `fri[5]=i1,...` and the total length of the array is `fri[0..frinum-1]`.

`tcalibrate`: Output. The time at which the current calibration information became valid. Units are Unix-C time in seconds defined in Section 17.

`tcalibrate_gps`: Output. Same as previous quantity, but with GPS time in seconds.

`locklow`. Output. The minimum value (inclusive) for "in-lock" in the lock channel. Set if and only if `fgetinput.inlock` is nonzero.

`lockhi`. Output. The maximum value (inclusive) for "in-lock" in the lock channel. Set if and only if `fgetinput.inlock` is nonzero.

`filename`. Output. Points to a static character string containing the name of the frame file currently in use, or NULL if there is no frame file open.

`slow_names`. Output. Names of the slow channels packed into one fast channel "SLOW".

Note that the time-stamps available in two different formats: Unix-C time in seconds and GPS time in seconds. The relationship between these is described in detail in Section 17. In general, in new code the GPS time stamps should be used, and taken as the more fundamental quantity. The Unix-C time is the number of seconds after 00:00:00 Jan 1, 1970 UTC. This is also known as "Calendar Time" on Unix systems. It is the quantity returned by the Standard C-library function `time()`. Note that starting with versions of the Frame library greater than 3.23, the time stored in the frames is GPS time, which is (roughly - up to leap seconds) defined as the Unix-C time minus 315964811 (this value may be found in the defined constant `UTCTOGMT` in the `grasp.h` header file. The origin of GPS time is 00:00:00 January 6, 1980 UTC, which was

$$\begin{aligned}
 315964811 \text{ sec} &= 3600 \text{ sec/hour} \times 24 \text{ hours/day} \times \\
 &\quad (365 \text{ days/year} \times 8 \text{ years} + 366 \text{ days/year} \times 2 \text{ years} + 5 \text{ days}) \\
 &\quad + 11 \text{ leap sec}
 \end{aligned}$$

after 00:00:00 Jan 1, 1970 UTC.

This routine is a useful interface to the FRAME library. It reads frames from files. To get the name of the first file to open, this routine calls the function `files()` specified in the input structure. Then, whenever there are no remaining frames in this file, it calls `files()` again. This function must return the name of the desired file, or NULL if no files remain. For example:

```

static char *filelist[]={
"C1-94_11_19_23_50_46", "C1-94_11_19_23_53_28",
"C1-94_11_19_23_56_10", "C1-94_11_19_23_58_52",
"C1-94_11_20_00_01_34", "C1-94_11_20_00_04_16" };

char *files() {
    static int entry=0;
    if (entry>=6)
        return NULL;
    else
        return filelist[entry++];
}

```

or the exact same fragment of code, but with:

```

static char *filelist[]={
"C1-468915467.F", "C1-468915629.F", "C1-468915791.F",
"C1-468915953.F", "C1-468916115.F", "C1-468916278.F" };

```

The difference between the labeling of the frame files here is that in the first instance (early versions of the frame library) the files are assumed to be labeled by the UTC time in "human-readable" form, and in the latter case they are assumed to be labeled by the GPS time in seconds. Further details may be found in Section 16.18 and Section 17.

The function `fget_ch()` returns 0 if it is unable to satisfy the request for `fgetinput.npoint` points. It returns 1 if the request has been satisfied, and it is beginning a new locked section (or if the frames

were not contiguous in time, and it is beginning with a new frame). It returns 2 if the data returned is part of an ongoing locked or continuous sequence.

When several channels are requested, and they have different sample rates, the first channel requested must always have the fastest sample rate. Other requested channels may have this same sample rate, but none may have a faster sample rate. Points are returned from the slower channels if and only if they satisfy the following condition. Suppose that r is the ratio of the channel 0 sample rate to the channel K sample rate, and label the points in channel 0 by $i = 0, \dots, nr - 1$, and the points in channel K by $j = 0, \dots, n - 1$. Then point j in channel K is returned if and only if point $i = rj$ is returned from channel 0.

Authors: Bruce Allen, ballen@dirac.phys.uwm.edu

Comments: None.

4.3 Function: `framefiles()`

```
char *framefiles()
```

This is a “utility function” for frame access. It takes no arguments, and returns a pointer to a static character string. It is intended primarily as an argument to be passed to the function `fget_ch()` via the structure member `fgetinput.files`.

The operation of the `framefiles()` is determined by two environment variables: `GRASP_FRAMEPATH`, and `GRASP_REALTIME`. If `GRASP_REALTIME` is set, then the `framefiles()` interrogates the EPICS control system and returns a pointer to a character string containing the name of the frame file most recently written to disk. This option is only intended for use in the 40-meter lab control room, for real-time analysis of data. For most users of GRASP, this option will never be used. Note: to set/unset the environment variable, use the commands:

```
setenv GRASP_REALTIME
unsetenv GRASP_REALTIME
```

respectively.

If the `GRASP_REALTIME` environment variable is NOT set, then the behavior of `framefiles()` is determined by the value of the `GRASP_FRAMEPATH` environment variable. This variable should point to a directory, and may be set with a command like:

```
setenv GRASP_DATAPATH /usr/local/GRASP/data/18nov94.1frame
```

The first time that `framefiles()` is called, it looks for all files with names of the type:

```
C1-[0-9]
C1-*.F
H-*.F
H-*.T
L-*.F
L-*.T
```

in the directory pointed to by `GRASP_FRAMEPATH`. These correspond, respectively, to Caltech 40-m frame files labeled by date, Caltech 40-m frame files labeled by GPS time, Hanford frame files, Hanford trend frames, Livingston frame files, and Livingston trend frames. Note that the directory should contain files with only one type of label: if several label types exist in the directory, only the files whose type matches the first entry found on the list above will be used. The labeling conventions are explained in Section 16.18. The file names are stored internally, `framefiles()` returns a pointer to a character string containing the name of the first of these files. The second call to `framefiles()` returns the name of the second file found in the directory, and so on. When no more files remain, `framefiles()` returns a NULL pointer.

A simple way to analyze a subset of data is to create a directory containing symbolic links to the FRAME files containing data that you want to analyze, and to set the environment variable `GRASP_FRAMEPATH` to point to that directory.

Authors: Bruce Allen, ballen@dirac.phys.uwm.edu

Comments: None.

4.4 Example: locklistF program

This example uses the function `fget_ch` described in the previous section to print out location information and times for all the locked sections in the directory pointed to by the environment variable `GRASP_FRAMEPATH`. To run this program, type

```
setenv GRASP_FRAMEPATH /usr/local/GRASP/18nov94.1frame
locklistF
```

and a list of locked time intervals will be printed out. Here is some typical output:

```
locklistF
FrameL Version:April 10, 1997; v2.23(Apr 10 1997 18:32:52 ../src/FrameL.c)
GRASP: framefiles(): using 83 files from directory /ballen2/18nov94.1frame
Id: frameinterface.c,v 1.4 1997/04/30 07:00:39 ballen Exp
Name: RELEASE_1_3
In lock from t = 0.000000 into run to 526.450008 into run for 526.450008 sec
Out of lock from t = 526.450008 into run to 555.384692 into run for 28.934683 sec
In lock from t = 555.384692 into run to 667.775527 into run for 112.390835 sec
Out of lock from t = 667.775527 into run to 708.000798 into run for 40.225272 sec
In lock from t = 708.000798 into run to 2268.670924 into run for 1560.670125 sec
Out of lock from t = 2268.670924 into run to 2283.429062 into run for 14.758138 sec
In lock from t = 2283.429062 into run to 3954.517061 into run for 1671.088000 sec
Out of lock from t = 3954.517061 into run to 3966.367962 into run for 11.850901 sec
GRASP: fget_ch(): FRAMES NOT SEQUENTIAL
run 3 frame 842 ended at time: 785221840.948503 sec
run 3 frame 843 started at time: 785223012.765137 sec
Time gap is 1171.816634 sec
Gap starts 4266.133503 seconds into run; ends 5437.950137 seconds into run.
Discarding 294210 points remaining in the previous frame(s).
Id: frameinterface.c,v 1.4 1997/04/30 07:00:39 ballen Exp
Name: RELEASE_1_3
In lock from t = 3966.367962 into run to 4266.133503 into run for 299.765540 sec
Out of lock from t = 4266.133503 into run to 5437.950137 into run for 1171.816634 sec
GRASP: fget_ch(): FRAMES NOT SEQUENTIAL
run 3 frame 1040 ended at time: 785224015.965104 sec
run 3 frame 1041 started at time: 785224175.714844 sec
Time gap is 159.749740 sec
Gap starts 6441.150104 seconds into run; ends 6600.899844 seconds into run.
Discarding 132000 points remaining in the previous frame(s).
Id: frameinterface.c,v 1.4 1997/04/30 07:00:39 ballen Exp
Name: RELEASE_1_3
In lock from t = 5437.950137 into run to 6441.150104 into run for 1003.199968 sec
Out of lock from t = 6441.150104 into run to 6600.899844 into run for 159.749740 sec
In lock from t = 6600.899844 into run to 7375.472558 into run for 774.572714 sec
Out of lock from t = 7375.472558 into run to 7391.474039 into run for 16.001482 sec
In lock from t = 7391.474039 into run to 7685.337699 into run for 293.863659 sec
Out of lock from t = 7685.337699 into run to 7719.763049 into run for 34.425351 sec
In lock from t = 7719.763049 into run to 7973.647310 into run for 253.884261 sec
Out of lock from t = 7973.647310 into run to 8083.507974 into run for 109.860664 sec
In lock from t = 8083.507974 into run to 9160.715956 into run for 1077.207982 sec
Out of lock from t = 9160.715956 into run to 9220.780081 into run for 60.064125 sec
In lock from t = 9220.780081 into run to 10552.863624 into run for 1332.083544 sec
Out of lock from t = 10552.863624 into run to 10585.141461 into run for 32.277837 sec
In lock from t = 10585.141461 into run to 11466.650847 into run for 881.509386 sec
Out of lock from t = 11466.650847 into run to 11483.939559 into run for 17.288712 sec
In lock from t = 11483.939559 into run to 13268.796352 into run for 1784.856793 sec
Out of lock from t = 13268.796352 into run to 13297.379120 into run for 28.582768 sec
GRASP: fget_ch(): could not open NULL file name
had 0 points; still need 296000 points...
Discarding 0 points remaining in the previous frame(s).
Id: frameinterface.c,v 1.4 1997/04/30 07:00:39 ballen Exp
```

Name: RELEASE_1_3

Note that this example only prints out information for locked sections longer than 30 sec. Also notice that because there are time gaps in between some of the successive frames, error messages are printed out. Notice the form of the GRASP error and warning messages. These typically begin with a line like:

```
GRASP: fget_ch(): this is the warning or error message
which specifies which GRASP function the error messages come from. They end with a pair of lines like
  Id:  frameinterface.c,v 1.4 1997/04/30 07:00:39 ballen Exp
  Name: Name:  RELEASE_1_3
```

which are information about the file from which the warning or error message came, including its version/release numbers. Here is the code for the locklistF example program:

```
/* GRASP: Copyright 1997,1998 Bruce Allen */
#include "grasp.h"

int main() {
    double begin,end,saveend=0.0;
    struct fgetoutput fgetoutput;
    struct fgetinput fgetinput;
    int firstpass=1;
    time_t unixtime;

    /* this number of samples is about 30 seconds of data */
    fgetinput.npoint=296000;

    /* number of channels needed is one */
    fgetinput.nchan=1;

    /* storage for channel names, data locations, points returned, ratios */
    fgetinput.chnames=(char **)malloc(fgetinput.nchan*sizeof(char *));
    fgetinput.locations=(short **)malloc(fgetinput.nchan*sizeof(short *));
    fgetoutput.npoint=(int *)malloc(fgetinput.nchan*sizeof(int));
    fgetoutput.ratios=(int *)malloc(fgetinput.nchan*sizeof(int));

    /* since we operate in SEEK mode, no space needed for data storage */
    fgetinput.locations[0]=NULL;

    /* use utility function framefiles() to retrieve file names */
    fgetinput.files=framefiles;

    /* get only the locked sections */
    fgetinput.inlock=1;

    /* seek over data (we don't care what the values are!) */
    fgetinput.seek=1;

    /* don't need calibration information */
    fgetinput.calibrate=0;

    /* set channel name */
    fgetinput.chnames[0]="IFO_DMRO";

    /* start the main loop */
    while (1) {

        /* find the next locked section of the data */
        fget_ch(&fgetoutput,&fgetinput);
```

```
/* print out absolute start time of run */
if (firstpass) {
    printf("Starting time of first frame: %13f Unix-C time\n",fgetoutput.tffirst);
    printf("Starting time of first frame: %13f GPS time\n",fgetoutput.tffirst_gps);
    unixtime=fgetoutput.tffirst;
    printf("    ~ UTC time %s",asctime(utctime(&unixtime)));
    printf(" ~ Unix gmtime %s\n",asctime(gmtime(&unixtime)));
    firstpass=0;
}

/* see if we fell out of lock, and print if we did */
if (fgetoutput.returnval==1) {

    /* time at which lock lost (relative to start of run) */
    begin=fgetoutput.lostlock-fgetoutput.tffirst;

    /* time at which lock aquired (relative to start of run) */
    end=fgetoutput.lastlock-fgetoutput.tffirst;
    if (begin>0.0) {
        printf("In lock from t = %f into run to %f into run for %f sec\n",
            saveend,begin,begin-saveend);
        printf("Out of lock from t = %f into run to %f into run for %f sec\n",
            begin,end,end-begin);
    }
    saveend=end;
}

/* if no data remains, then exit */
if (fgetoutput.returnval==0) {
    printf("End of data at time %f\n",fgetoutput.tstart-fgetoutput.tffirst);
    break;
}
}
return 0;
}
```

4.5 Example: gwoutputF program

This example uses the function `fget_ch()` described in the previous section to print out a two-column file containing the IFO output for the first locked section containing 100 sample points. To run this program, type

```
setenv GRASP_FRAMEPATH /usr/local/GRASP/18nov94.1frame
gwoutputF
```

In the output, the left column is time values, and the right column is the actual IFO output (note that because this comes from a 12 bit A-D converter, the output is an integer value from -2047 to 2048). The program works by acquiring data 100 points at a time, then printing out the values, then acquiring 100 more points, and so on. Whenever a new locked section begins, the program prints a banner message to alert the user. Note that typical locked sections contain $\approx 10^7$ points of data, so this program should not be used for real work – it's just a demonstration!

```
/* GRASP: Copyright 1997,1998 Bruce Allen */
#include "grasp.h"

int main() {
    float tstart,time,srate;
    int i,npoint,code;
    short *data;
    struct fgetinput fgetinput;
    struct fgetoutput fgetoutput;

    /* specify the number of points of output & allocate array */
    npoint=100;
    data=(short *)malloc(sizeof(short)*npoint);
    fgetinput.npoint=npoint;

    /* we only want one channel of data */
    fgetinput.nchan=1;

    /* use the framefiles() function to find it */
    fgetinput.files=framefiles;

    /* allocate space to store channel names */
    fgetinput.chnames=(char **)malloc(fgetinput.nchan*sizeof(char *));

    /* allocate space for data storage location addresses */
    fgetinput.locations=(short **)malloc(fgetinput.nchan*sizeof(short *));

    /* allocate space for numbers of points returned in each channel */
    fgetoutput.npoint=(int *)malloc(fgetinput.nchan*sizeof(int));

    /* allocate space for ratios of channel sample rates */
    fgetoutput.ratios=(int *)malloc(fgetinput.nchan*sizeof(int));

    /* channel name */
    fgetinput.chnames[0]="IFO_DMRO";

    /* set up different cases */
    if (NULL!=getenv("GRASP_REALTIME")) {
        /* don't care if locked */
        fgetinput.inlock=0;
    }
    else {
```

```
    /* only locked */
    fgetinput.inlock=1;
}

fgetinput.seek=0;
fgetinput.calibrate=0;
fgetinput.locations[0]=data;

while (1) {
    /* get npoint points of data */
    code=fget_ch(&fgetoutput,&fgetinput);
    tstart=fgetoutput.dt;
    srate=fgetoutput.srate;

    /* if no data remains, exit loop */
    if (code==0) break;
    /* if starting a new locked segment, print banner */
    if (code==1) {
        printf("_____ NEW LOCKED SEGMENT _____\n\n");
        printf("   Time (sec)\t   IFO output\n");
    }
    /* now output the data */
    for (i=0;i<npoint;i++) {
        time=tstart+i/srate;
        printf("%f\t%d\n",time,(int)data[i]);
    }
}
/* close the data files, and return */
return 0;
}
```

4.6 Example: animateF program

This example uses the function `fget_ch()` described in the previous section to produce an animated display showing the time series output of the IFO in a lower window, and a simultaneously calculated FFT power spectrum in the upper window. To run this program, type

```
setenv GRASP_FRAMEPATH /usr/local/GRASP/18nov94.1frame
animateF | xmgr -pipe
```

This output from this program must be piped into a public domain graphing program called `xmgr`. This may be obtained from <http://plasma-gate.weizmann.ac.il/Xmgr/>. (This lists mirror sites in the USA and Europe also). Some sample output of `animateF` is shown in Figure 11.

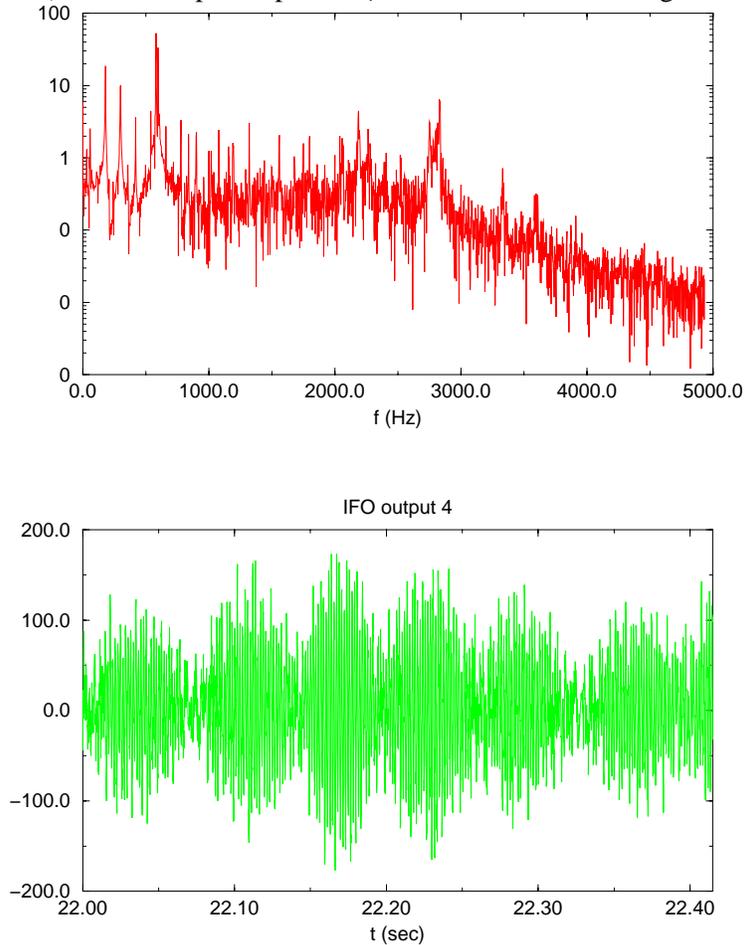


Figure 11: Snapshot of output from `animateF`. This shows the (whitened) CIT 40-meter IFO a few seconds after acquiring lock, before the violin modes have damped down

After compilation, to run the program type:

```
animateF | xmgr -pipe &
```

to get an animated display showing the data flowing by and the power spectrum changing, starting from the first locked data. You can also use this program with command-line arguments, for example

```
animateF 100 4 500 7 900 1.5 | xmgr -pipe &
```

will show the data from time $t = 100$ to time $t = 104$ seconds, then from $t = 500$ to $t = 507$, then from $t = 900$ to $t = 901.5$. Notice that the sequence of start times must be increasing. Note: the start times are measured relative to the first data point in the first frame of data.

```
        case 0:
==>      delt=(x[ilen-1]-x[0])/(ilen-1.0);
==>      T=(x[ilen-1]-x[0]);
          setlength(cg,specset,ilen/2);
          xx=getx(cg,specset);
...

        case 1:
==>      delt=(x[ilen-1]-x[0])/(ilen-1.0);
==>      T=(x[ilen-1]-x[0]);
```

Figure 12: The corrections to a bug in the `xmgr` program are indicated by the arrows above. This bug is in the routine `do_fourier()` in the file `computils.c`. This bug has been corrected in `xmgr` version 4.1 and greater.

Note 1: The `xmgr` program as commonly distributed has a simple bug that needs to be repaired, in order for the frequency scale of the Fourier transform to be correct. The corrected version of `xmgr` is shown in Figure 12.

Note 2: Two closely related programs `animateT` and `animateF` are also included. `animateT` is identical to `animateF` except that it does *not* assume that the data is in shorts. Hence it is appropriate, for example, for producing an animated display of ‘trend’ files in which frames contain channels stored as doubles. `animateS` is designed to produce an animated display of data from a channel name ‘SLOW’. This channel is used as a way of packing a approximately 230 channels sampled at 1Hz into a single fake channel (incorrectly labelled with sample rate 256Hz).

```
/* GRASP: Copyright 1997,1998 Bruce Allen */
#include "grasp.h"

int main(int argc,char **argv) {
    void graphout(float,float,int,char*);
    float tstart=1.e35,srate=1.e-30,tmin,tmax,dt;
    double time=0.0;
    int i,seq=0,code,npoint=4096;
    short *data;
    struct fgetinput fgetinput;
    struct fgetoutput fgetoutput;
    char *nstring;

    /* number of channels */
    fgetinput.nchan=1;

    /* number of samples per update */
    if (NULL!=(nstring=getenv("GRASP_NSAMPLE"))) {
        npoint=atoi(nstring);
        fprintf(stderr,"Display points set to environment variable: GRASP_NSAMPLE = %d\n",npoint);
    }

    /* source of files */
    fgetinput.files=framefiles;
```

```
/* storage for channel names, data locations, points returned, ratios */
fgetinput.chnames=(char **)malloc(fgetinput.nchan*sizeof(char *));
fgetinput.locations=(short **)malloc(fgetinput.nchan*sizeof(short *));
fgetoutput.npoint=(int *)malloc(fgetinput.nchan*sizeof(int));
fgetoutput.ratios=(int *)malloc(fgetinput.nchan*sizeof(int));

/* set up channel names, etc. for different cases */
fgetinput.chnames[0]="IFO_DMRO";

/* set up for different cases */
if (NULL!=getenv("GRASP_CHANNEL")) {
    /* 40 meter lab */
    fgetinput.chnames[0]=getenv("GRASP_CHANNEL");
    fgetinput.inlock=0;
}
else {
    /* Nov 1994 data set */
    fgetinput.inlock=1;
}
/* fgetinput.inlock=0; */
}

/* number of points to get */
fgetinput.npoint=npoint;

/* don't seek, we need the sample values! */
fgetinput.seek=0;

/* but we don't need calibration information */
fgetinput.calibrate=0;

/* allocate storage space for data */
data=(short *)malloc(sizeof(short)*npoint);
fgetinput.locations[0]=data;

/* handle case where user has supplied t or dt arguments */
if (argc==1) {
    tmin=-1.e30;
    dt=2.e30;
    argc=-1;
}

/* now loop ... */
seq=argc;
while (argc!=1) {
    /* get the next start time and dt */
    if (argc!=-1) {
        sscanf(argv[seq-argc+1],"%f",&tmin);
        sscanf(argv[seq-argc+2],"%f",&dt);
        argc-=2;
    }
    /* calculate the end of the observation interval, and get data */
    tmax=tmin+dt;
    while (1) {
        /* decide whether to skip (seek) or get sample values */
        if (tstart<tmin-(npoint+20.)/srate)
            fgetinput.seek=1;
        else
            fgetinput.seek=0;
    }
}
```

```
/* seek, or get the sample values */
code=fget_ch(&fgetoutput,&fgetinput);

/* elapsed time, sample rate */          tstart=fgetoutput.dt;
srate=fgetoutput.srate;

/* if no data left, return */
if (code==0) return 0;

/* we need to be outputting now... */
if (tmin<=tstart){
  for (i=0;i<npoint;i++) {
    time=tstart+i/srate;
    printf("%f\t%d\n",time,data[i]);
  }

  /* put out information for the graphing program */
  graphout(tstart,tstart+npoint/srate,(argc==1 && time>=tmax),fgetinput.chnames[0]);
}
/* if we are done with this interval, try next one */
if (time>=tmax) break;
}
}
return 0;
}
```

```
/* This routine is pipes output into the xmgr graphing program */
void graphout(float x1,float x2,int last,char* ch_name) {
  static int count=0;
  printf("&\n");                               /* end of set marker */
  /* first time we draw the plot */
  if (count==0) {
    printf("@doublebuffer true\n");           /* keeps display from flashing */
    printf("@s0 color 3\n");                  /* IFO graph is green */
    printf("@view 0.1, 0.1, 0.9, 0.45\n");   /* set the viewport for IFO */
    printf("@with g1\n");                     /* reset the current graph to FFT */
    printf("@view 0.1, 0.6, 0.9, 0.95\n");   /* set the viewport FFT */
    printf("@with g0\n");                     /* reset the current graph to IFO */
    printf("@world xmin %f\n",x1);            /* set min x */
    printf("@world xmax %f\n",x2);           /* set max x */
    printf("@autoscale\n");                  /* autoscale first time through */
    printf("@focus off\n");                 /* turn off the focus markers */
    printf("@xaxis label \"t (sec)\"\\n");    /* IFO axis label */
    printf("@fft(s0, 1)\n");                 /* compute the spectrum */
    printf("@s1 color 2\n");                 /* FFT is red */
    printf("@move g0.s1 to g1.s0\n");        /* move FFT to graph 1 */
    printf("@with g1\n");                     /* set the focus on FFT */
    printf("@g1 type logy\n");               /* set FFT to log freq axis */
    printf("@autoscale\n");                  /* autoscale FFT */
    printf("@subtitle \"Spectrum\"\\n");     /* set the subtitle */
    printf("@xaxis label \"f (Hz)\"\\n");    /* FFT axis label */
    printf("@with g0\n");                    /* reset the current graph IFO */
    printf("@subtitle \"IFO output %d\"\\n",count++); /* set IFO subtitle */
    if (!last) printf("@kill s0\n");        /* kill IFO; ready to read again */
  }
  else {
    /* other times we redraw the plot */
    printf("@s0 color 3\n");                 /* set IFO green */
    printf("@fft(s0, 1)\n");                 /* FFT it */
  }
}
```

```
printf("@s1 color 2\n");          /* set FFT red */
printf("@move g0.s1 to g1.s0\n"); /* move FFT to graph 1 */
printf("@subtitle \"%s %d\"\n", ch_name, count++); /* set IFO subtitle */
printf("@world xmin %f\n", x1);   /* set min x */
printf("@world xmax %f\n", x2);   /* set max x */
printf("@autoscale yaxes\n");     /* autoscale IFO */
printf("@clear stack\n");         /* clear the stack */
if (!last) printf("@kill s0\n");  /* kill IFO data */
printf("@with g1\n");            /* switch to FFT */
printf("@g1 type logy\n");        /* set FFT to log freq axis */
printf("@clear stack\n");         /* clear stack */
if (!last) printf("@kill s0\n");  /* kill FFT */
printf("@with g0\n");            /* ready to read IFO again */
}
return;
}
```

4.7 Swept-sine calibration information

The swept sine calibration files are 3-column ASCII files, of the form:

$$\begin{array}{ccc} f_0 & r_0 & i_0 \\ f_1 & r_1 & i_1 \\ f_2 & r_2 & i_2 \\ & \dots & \\ f_m & r_m & i_m \end{array}$$

where the f_j are frequencies, in Hz, and r_j and i_j are dimensionless ratios of voltages. There are typically $m = 801$ lines in these files. The data from these files (as well as one additional line of the form 0.0 0.0 0.0

showing vanishing response at DC) have been included in the frames. Each line gives the ratio of the IFO output voltage to a calibration coil driving voltage, at a different frequency. The r_j are the “real part” of the response, i.e. the ratio of the IFO output in phase with the coil driving voltage, to the coil driving voltage. The i_j are the “imaginary part” of the response, 90 degrees out of phase with the coil driving voltage. The sign of the phase (or equivalently, the sign of the imaginary part of the response) is determined by the following convention. Suppose that the driving voltage (in volts) is

$$V_{\text{coil}} = 10 \cos(\omega t) = 10 \Re e^{i\omega t} \quad (4.7.1)$$

where $\omega = 2\pi \times 60$ radians/sec is the angular frequency of a 60 Hz signal. Suppose the response of the interferometer output to this is (again, in volts)

$$\begin{aligned} V_{\text{IFO}} &= 6.93 \cos(\omega t) + 4 \sin(\omega t) \\ &= 6.93 \cos(\omega t) - 4 \cos(\omega t + \pi/2) \\ &= 8 \Re e^{i(\omega t - \pi/6)} \end{aligned} \quad (4.7.2)$$

This is shown in Figure 13. An electrical engineer would describe this situation by saying that the phase of the response V_{IFO} is lagging the phase of the driving signal V_{coil} by 30° . The corresponding line in the swept sine calibration file would read:

$$\begin{array}{ccc} & \dots & \\ 60.000 & 0.6930 & -0.40000 \\ & \dots & \end{array}$$

Hence, in this example, the real part is positive and the imaginary part is negative. We will denote this entry in the swept sine calibration file by $S(60) = 0.8 e^{-i\pi/6} = 0.693 - 0.400i$. Because the interferometer output is real, there is also a value implied at negative frequencies which is the complex conjugate of the positive frequency value: $S(-60) = S^*(60) = 0.8 e^{i\pi/6} = 0.693 + 0.400i$.

Because the interferometer has no DC response, it is convenient for us to add one additional point at frequency $f = 0$ into the output data arrays, with both the real and imaginary parts of the response set to zero. Hence the output arrays contain one element more than the number of lines in the input files. Note that both of these arrays are arranged in order of increasing frequency; after adding our one additional point they typically contain 802 points at frequencies from 0 Hz to 5001 Hz.

For the data runs of interest in this section (from November 1994) typically a swept sine calibration curve was taken immediately before each data tape was generated.

We will shortly address the following question. How does one use the dimensionless data in the swept-sine calibration curve to reconstruct the differential motion $\Delta l(t)$ (in meters) of the interferometer arms? Here we address the closely related question: given V_{IFO} , how do we reconstruct V_{coil} ? We choose the sign

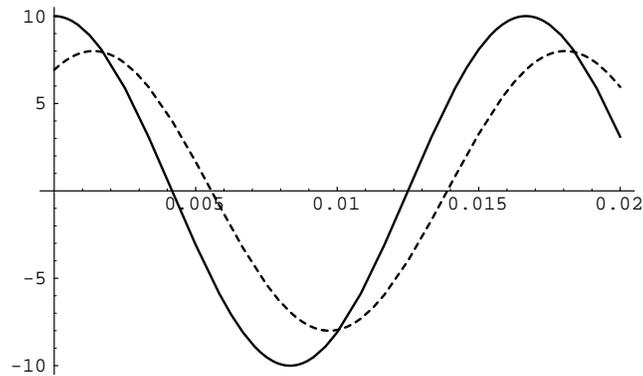


Figure 13: This shows a driving voltage V_{coil} (solid curve) and the response voltage V_{IFO} (dotted curve) as functions of time (in sec). Both are 60 Hz sinusoids; the relative amplitude and phase of the in-phase and out-of-phase components of V_{IFO} are contained in the swept-sine calibration files.

convention for the Fourier transform which agrees with that of *Numerical Recipes*: equation (12.1.6) of [1]. The Fourier transform of a function of time $V(t)$ is

$$\tilde{V}(f) = \int e^{2\pi i f t} V(t) dt. \quad (4.7.3)$$

The inverse Fourier transform is

$$V(t) = \int e^{-2\pi i f t} \tilde{V}(f) df. \quad (4.7.4)$$

With these conventions, the signals (4.7.1) and (4.7.2) shown in in Figure 13 have Fourier components:

$$\tilde{V}_{\text{coil}}(60) = 5 \quad \text{and} \quad \tilde{V}_{\text{coil}}(-60) = 5, \quad (4.7.5)$$

$$\tilde{V}_{\text{IFO}}(60) = 4e^{i\pi/6} \quad \text{and} \quad \tilde{V}_{\text{IFO}}(-60) = 4e^{-i\pi/6}. \quad (4.7.6)$$

At frequency $f_0 = 60$ Hz the swept sine file contains

$$S(60) = 0.8 e^{-i\pi/6} \Rightarrow S(-60) = S^*(60) = 0.8 e^{i\pi/6}. \quad (4.7.7)$$

since $S(-f) = S^*(f)$.

With these choices for our conventions, one can see immediately from our example (and generalize to all frequencies) that

$$\tilde{V}_{\text{coil}}(f) = \frac{\tilde{V}_{\text{IFO}}}{S^*(f)}. \quad (4.7.8)$$

In other words, with the *Numerical Recipes* [1] conventions for forward and reverse Fourier Transforms, the (FFT of the) calibration-coil voltage is the (FFT of the) IFO-output voltage divided by the complex conjugate of the swept sine response.

Author: Bruce Allen, ballen@dirac.phys.uwm.edu

Comments: The swept-sine calibration curves are usually quite smooth but sometimes they contain a “glitch” in the vicinity of 1 kHz; this may be due to drift of the unity-gain servo point.

4.8 Function: GRcalibrate()

```
void GRcalibrate(float *fri,int frinum,int num,float *complex,float srate,int method,int order)
```

This is an intermediate-level routine which takes as input a pointer to an array containing the swept sine data, and outputs an array of interpolated points suitable for calibration of FFT's of the interferometer output.

The arguments are:

fri: Input. Pointer to an array containing swept sine data. The format of this data is `fri[0]=f0`, `fri[1]=r0`, `fri[2]=i0`, `fri[3]=f1`, `fri[4]=r1`, `fri[5]=i1`,... and the total length of the array is `fri[0..frinum-1]`.

frinum: Input. The number of entries in the array `fri[0..frinum-1]`. If this number is not divisible by three, something is wrong!

num: Input. The number of points N in the FFT that we will be calibrating. This is typically $N = 2^k$ where k is an integer. In this case, the number of distinct frequency values at which a calibration is needed is $2^{k-1} + 1 = N/2 + 1$, corresponding to the number of distinct frequency values from 0 (DC) to the Nyquist frequency f_{Nyquist} . See for example equation (12.1.5) of reference [1]. The frequencies are $f_i = \frac{i}{N} F_{\text{sample}}$ for $i = 0, \dots, N/2$.

srate: Input. The sample rate F_{sample} (in Hz) of the data that we are going to be calibrating.

complex: Input. Pointer to an array `complex[0..s]` where $s = 2^k + 1$. The routine `calibrate()` fills in this array with interpolated values of the swept sine calibration data, described in the previous section. The real part of the DC response is in `complex[0]`, and the imaginary part is in `complex[1]`. The real/imaginary parts of the response at frequency f_1 are in `complex[2]` and `complex[3]` and so on. The last two elements of `complex[]` contain the real/imaginary parts of the response at the Nyquist frequency $F_{\text{sample}}/2$.

method: Input. This integer sets the type of interpolation used to determine the real and imaginary part of the response, at frequencies that lie in between those given in the swept sine calibration files. Rational function interpolation is used if `method=0`. Polynomial interpolation is used if `method=1`. Spline interpolation with natural boundary conditions (vanishing second derivatives at DC and the Nyquist frequency) is used if `method=2`.

order: Input. Ignored if spline interpolation is used. If polynomial interpolation is used, then `order` is the order of the interpolating polynomial. If rational function interpolation is used, then the numerator and denominator are both polynomials of order `order/2` if `order` is even; otherwise the degree of the denominator is `(order+1)/2` and that of the numerator is `(order-1)/2`.

The basic problem solved by this routine is that the swept sine calibration data in a frame typically contain data at a few hundred distinct frequency values. However to properly calibrate the IFO output, one usually needs this calibration information at a large number of frequencies corresponding to the distinct frequencies associated with the FFT of a data set. This routine allows you to choose different possible interpolation methods. If in doubt, we recommend spline interpolation as the first choice. The interpolation methods are described in detail in Chapter 3 of reference [1].

Author: Bruce Allen, ballen@dirac.phys.uwm.edu

Comments: It might be better to interpolate values of f^2 times the swept-sine response function, as this is the quantity needed to compute the IFO response function.

4.9 Example: print_ssF program

This example uses the function `GRcalibrate()` to read the swept sine calibration information from a frame, and then prints out a list of frequencies, real, and imaginary parts interpolated from this data. The frequencies are appropriate for the FFT of a 4096 point data set with sample rate `srate`. The technique used is spline interpolation. To run this program, and display a graph, type

```
setenv GRASP_FRAMEPATH /usr/local/GRASP/18nov94.1frame
print_ssF > outputfile
xmgr -nxy outputfile
```

```
/* GRASP: Copyright 1997,1998 Bruce Allen */
#include "grasp.h"
#define NPOINT 4096

int main() {
    float cplx[NPOINT+2],srate,freq;
    int npoint,i;
    struct fgetoutput fgetoutput;
    struct fgetinput fgetinput;

    /* we need to ask for some sample values, even though all we want is calibration */
    fgetinput.npoint=256;

    /* number of channels */
    fgetinput.nchan=1;

    /* storage for channel names, data locations, points returned, ratios */
    fgetinput.chnames=(char **)malloc(fgetinput.nchan*sizeof(char *));
    fgetoutput.npoint=(int *)malloc(fgetinput.nchan*sizeof(int));
    fgetoutput.ratios=(int *)malloc(fgetinput.nchan*sizeof(int));

    /* use utility function framefiles() to retrieve file names */
    fgetinput.files=framefiles;

    /* don't care if IFO is in lock */
    fgetinput.inlock=0;

    /* don't need data anyway, so might as well seek */
    fgetinput.seek=1;

    /* but we DO need the calibration information */
    fgetinput.calibrate=1;

    /* set the channel name */
    fgetinput.chnames[0]="IFO_DMRO";

    /* number of points of (imagined) FFT */
    npoint=NPOINT;

    /* now get the data (none) and calibration (what we want) */
    fget_ch(&fgetoutput,&fgetinput);

    /* the fast-channel sample rate */
    srate=fgetoutput.srate;

    /* swept sine calibration array is first argument */
```

```
GRcalibrate(fgetoutput.fri,fgetoutput.frinum,npoint,cplx,srate,2,0);  
  
/* print out frequency, real, imaginary interpolated values */  
printf("# Freq (Hz)\tReal\t\tImag\n");  
for (i=0;i<=NPOINT/2;i++) {  
    freq=i*srate/NPOINT;  
    printf("%e\t%e\t%e\n",freq,cplx[2*i],cplx[2*i+1]);  
}  
return 0;  
}
```

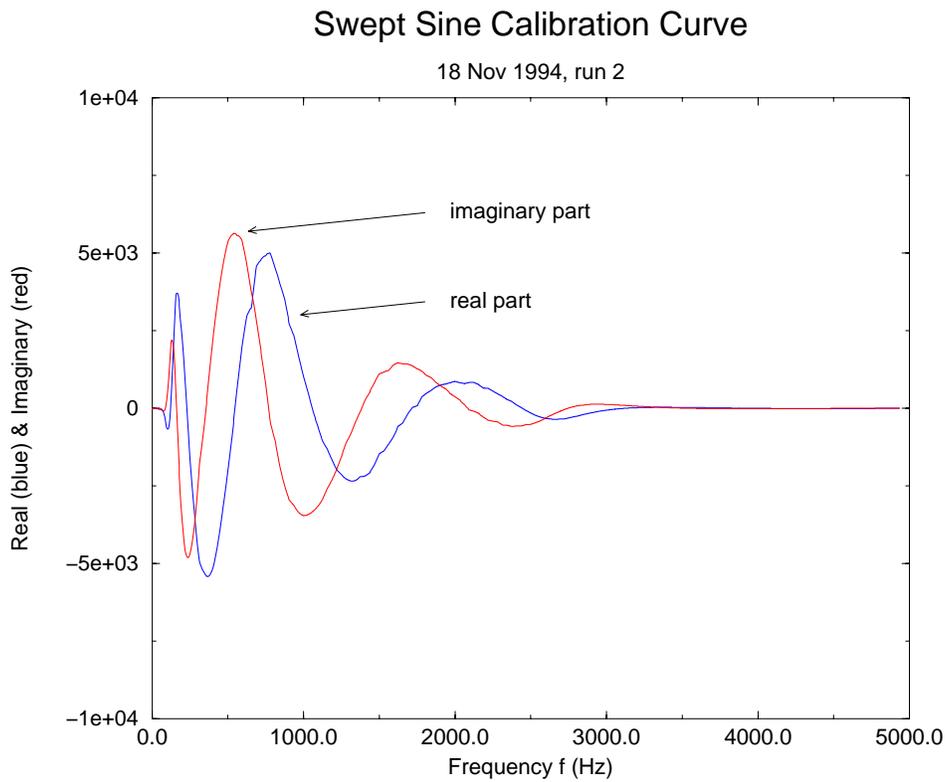


Figure 14: A swept sine calibration curve, showing the real and imaginary parts, produced by the example program print_ssF.

4.10 Function: GRnormalize()

```
void GRnormalize(float *fri, int frinum, int npoint, float srate, float *response)
```

This routine generates an array of complex numbers $R(f)$ from the swept sine information in a frame, and an overall calibration constant. Multiplying this array of complex numbers by (the FFT of) the raw IFO data yields the (FFT of the) differential displacement of the interferometer arms Δl , in meters: $\widetilde{\Delta l}(f) = R(f)\widetilde{C}_{\text{IFO}}(f)$. The units of $R(f)$ are meters/ADC-count.

The arguments are:

fri: Input. Pointer to an array containing swept sine data. The format of this data is $\text{fri}[0]=f_0$, $\text{fri}[1]=r_0$, $\text{fri}[2]=i_0$, $\text{fri}[3]=f_1$, $\text{fri}[4]=r_1$, $\text{fri}[5]=i_1, \dots$ and the total length of the array is $\text{fri}[0..\text{frinum}-1]$.

frinum: Input. The number of entries in the array $\text{fri}[0..\text{frinum}-1]$. If this number is not divisible by three, something is wrong!

npoint: Input. The number of points N of IFO output which will be used to calculate an FFT for normalization. Must be an integer power of 2.

srate: Input. The sample rate in Hz of the IFO output.

response: Output. Pointer to an array $\text{response}[0..s]$ with $s = N + 1$ in which $R(f)$ will be returned. By convention, $R(0) = 0$ so that $\text{response}[0]=\text{response}[1]=0$. Array elements $\text{response}[2i]$ and $\text{response}[2i + 1]$ contain the real and imaginary parts of $R(f)$ at frequency $f = israte/N$. The response at the Nyquist frequency $\text{response}[N]=0$ and $\text{response}[N+1]=0$ by convention.

The absolute normalization of the interferometer can be obtained from the information in the swept sine file, and one other normalization constant which we denote by Q . It is easy to understand how this works. In the calibration process, one of the interferometer end mirrors of mass m is driven by a magnetic coil. The equation of motion of the driven end mass is

$$m \frac{d^2}{dt^2} \Delta l = F(t) \quad (4.10.1)$$

where $F(t)$ is the driving force and Δl is the differential length of the two interferometer arms, in meters. Since the driving force $d(t)$ is proportional to the coil current and thus to the coil voltage, in frequency space this equation becomes

$$(-2\pi i f)^2 \widetilde{\Delta l} = \text{constant} \times \widetilde{V}_{\text{coil}} = \text{constant} \times \frac{\widetilde{V}_{\text{IFO}}}{S^*(f)}. \quad (4.10.2)$$

We have substituted in equation (4.7.8) which relates $\widetilde{V}_{\text{IFO}}$ and $\widetilde{V}_{\text{coil}}$. The IFO voltage is directly proportional to the quantity recorded in the IFO output channel: $V_{\text{IFO}} = \text{ADC} \times C_{\text{IFO}}$, with the constant ADC being the ratio of the analog-to-digital converters input voltage to output count.

Putting together these factors, the properly normalized value of Δl , in meters, may be obtained from the information in the IFO output channel, the swept sine calibration information, and the quantities given in Table 6 by

$$\widetilde{\Delta l} = R(f) \times \widetilde{C}_{\text{IFO}} \quad \text{with} \quad R(f) = \frac{Q \times \text{ADC}}{-4\pi^2 f^2 S^*(f)}, \quad (4.10.3)$$

Table 6: Quantities entering into normalization of the IFO output.

Description	Name	Value	Units
Gravity-wave signal (IFO output)	C_{IFO}	varies	ADC counts
A→D converter sensitivity	ADC	10/2048	$V_{\text{IFO}} (\text{ADC counts})^{-1}$
Swept sine calibration	S(f)	from file	$V_{\text{IFO}} (V_{\text{coil}})^{-1}$
Calibration constant	Q	1.428×10^{-4}	meter $\text{Hz}^2 (V_{\text{coil}})^{-1}$

where the \sim denotes Fourier transform, and f denotes frequency in Hz. (Note that, apart from the complex conjugate on S , the conventions used in the Fourier transform drop out of this equation, provided that identical conventions (4.7.3,4.7.4) are applied to both Δl and to C_{IFO}). The constant quantity Q indicated in the above equations has been calculated and documented in a series of calibration experiments carried out by Robert Spero. In these calibration experiments, the interferometer's servo was left open-loop, and the end mass was driven at a single frequency, hard enough to move the end mass one-half wavelength and shift the interferences fringes pattern over by one fringe. In this way, the coil voltage required to bring about a given length motion at a particular frequency was established, and from this information, the value of Q may be inferred. During the November 1994 runs the value of Q was given by

$$Q = \frac{\sqrt{9.35 \text{ Hz}}}{k} = 1.428 \times 10^{-4} \frac{\text{meter Hz}^2}{V_{\text{coil}}} \quad \text{where } k = 21399 \frac{V_{\text{coil}}}{\text{meter Hz}^{3/2}}. \quad (4.10.4)$$

Author: Bruce Allen, ballen@dirac.phys.uwm.edu

Comments: See comment for calibrate().

4.11 Example: power_spectrumF program

This example uses the function `GRnormalize()` to produce a normalized, properly calibrated power spectrum of the interferometer noise, using the gravity-wave signal and the swept-sine calibration information from the frames.

The output of this program is a 2-column file; the first column is frequency and the second column is the noise in units of meters/ $\sqrt{\text{Hz}}$. To run this program, and display a graph, type

```
setenv GRASP_FRAMEPATH /usr/local/GRASP/18nov94.1frame
power_spectrumF > outputfile
xmgr -log xy outputfile
```

A couple of comments are in order here:

1. Even though we only need the modulus, for pedagogic reasons, we explicitly calculate both the real and imaginary parts of $\widetilde{\Delta l}(f) = R(f)\widetilde{C}_{\text{IFO}}(f)$.
2. The fast Fourier transform of Δl , which we denote $\text{FFT}[\Delta l]$, has the same units (meters!) as Δl . As can be immediately seen from *Numerical Recipes* equation (12.1.6) the Fourier transform $\widetilde{\Delta l}$ has units of meters-sec and is given by $\widetilde{\Delta l} = \Delta t \text{FFT}[\Delta l]$, where Δt is the sample interval. The (one-sided) power spectrum of Δl in meters/ $\sqrt{\text{Hz}}$ is $P = \sqrt{\frac{2}{T}}|\widetilde{\Delta l}|$ where $T = N\Delta t$ is the total length of the observation interval, in seconds. Hence one has

$$P = \sqrt{\frac{2}{N\Delta t}} \Delta t |\text{FFT}[\Delta l]| = \sqrt{\frac{2\Delta t}{N}} |\text{FFT}[\Delta l]|. \quad (4.11.1)$$

This is the reason for the factor which appears in this example.

3. To get a spectrum with decent frequency resolution, the time-domain data must be windowed (see the example program `calibrate` and the function `avg_spec()` to see how this works).

A sample of the output from this program is shown in Figure 15.

```
/* GRASP: Copyright 1997,1998 Bruce Allen */
#include "grasp.h"
#define NPOINT 65536

int main() {
    void realft(float*,unsigned long,int);
    float response[NPOINT+2],data[NPOINT],freq;
    float res_real,res_imag,dl_real,dl_imag,c0_real,c0_imag,spectrum,srate,factor;
    int i,npoint;
    short datas[NPOINT];
    struct fgetinput fgetinput;
    struct fgetoutput fgetoutput;

    /* We need only the IFO output */
    fgetinput.nchan=1;

    /* use utility function framefiles() to retrieve file names */
    fgetinput.files=framefiles;

    /* storage for channel names, data locations, points returned, ratios */
    fgetinput.chnames=(char **)malloc(fgetinput.nchan*sizeof(char *));
    fgetinput.locations=(short **)malloc(fgetinput.nchan*sizeof(short *));
```

```
fgetoutput.npoint=(int *)malloc(fgetinput.nchan*sizeof(int));
fgetoutput.ratios=(int *)malloc(fgetinput.nchan*sizeof(int));

/* set channel name */
fgetinput.chnames[0]="IFO_DMRO";

/* are we in the 40-meter lab? */
if (NULL!=getenv("GRASP_REALTIME")) {
    /* for Caltech 40-meter lab */
    fgetinput.inlock=0;
}
else {
    /* for Nov 1994 data set */
    fgetinput.inlock=1;
}
/* number of points to sample and fft (power of 2) */
fgetinput.npoint=npoint=NPOINT;
fgetinput.calibrate=1;

/* the array where we want the data to be put */
fgetinput.locations[0]=datas;

/* skip 200 seconds into locked region (just seek, no need for data) */
fgetinput.seek=1;
fgetoutput.tstart=fgetoutput.lastlock=0.0;
while (fgetoutput.tstart-fgetoutput.lastlock<200.0)
    fget_ch(&fgetoutput,&fgetinput);

/* and get next stretch of data (don't seek, we need data) */
fgetinput.seek=0;
fget_ch(&fgetoutput,&fgetinput);

/* the sample rate */
srate=fgetoutput.srate;

/* convert gw signal (ADC counts) from shorts to floats */
for (i=0;i<NPOINT;i++) data[i]=datas[i];

/* FFT the data */
realft(data-1,npoint,1);

/* get normalization R(f) using swept sine calibration information from frame */
GRnormalize(fgetoutput.fri,fgetoutput.frinum,npoint,srate,response);

/* one-sided power-spectrum normalization, to get meters/rHz */
factor=sqrt(2.0/(srate*npoint));
/* compute dl. Leave off DC (i=0) or Nyquist (i=npoint/2) freq */
for (i=1;i<npoint/2;i++) {
    /* frequency */
    freq=i*srate/npoint;
    /* real and imaginary parts of tilde c0 */
    c0_real=data[2*i];
    c0_imag=data[2*i+1];
    /* real and imaginary parts of R */
    res_real=response[2*i];
    res_imag=response[2*i+1];
    /* real and imaginary parts of tilde dl */
    dl_real=c0_real*res_real-c0_imag*res_imag;
```

```
    dl_imag=c0_real*res_imag+c0_imag*res_real;
    /* |tilde dl| */
    spectrum=factor*sqrt(dl_real*dl_real+dl_imag*dl_imag);
    /* output freq in Hz, noise power in meters/rHz */
    printf("%e\t%e\n",freq,spectrum);
}
return 0;
}
```

Author: Bruce Allen, ballen@dirac.phys.uwm.edu

Comments: The IFO output typically consists of a number of strong line sources (harmonics of the 60 Hz line and the 180 Hz laser power supply, violin modes of the suspension, etc) superposed on a continuum background (electronics noise, laser shot noise, etc) In such situations, there are better ways of finding the noise power spectrum (for example, see the multi-taper methods of David J. Thomson [39], or the textbook by Percival and Walden [40]). Using methods such as the F-test to remove line features from the time-domain data stream might reduce the sidelobe contamination (bias) from nearby frequency bins, and thus permit an effective reduction of instrument noise near these spectral line features. Further details of these methods, and some routines that implement them, may be found in Section 16.19.

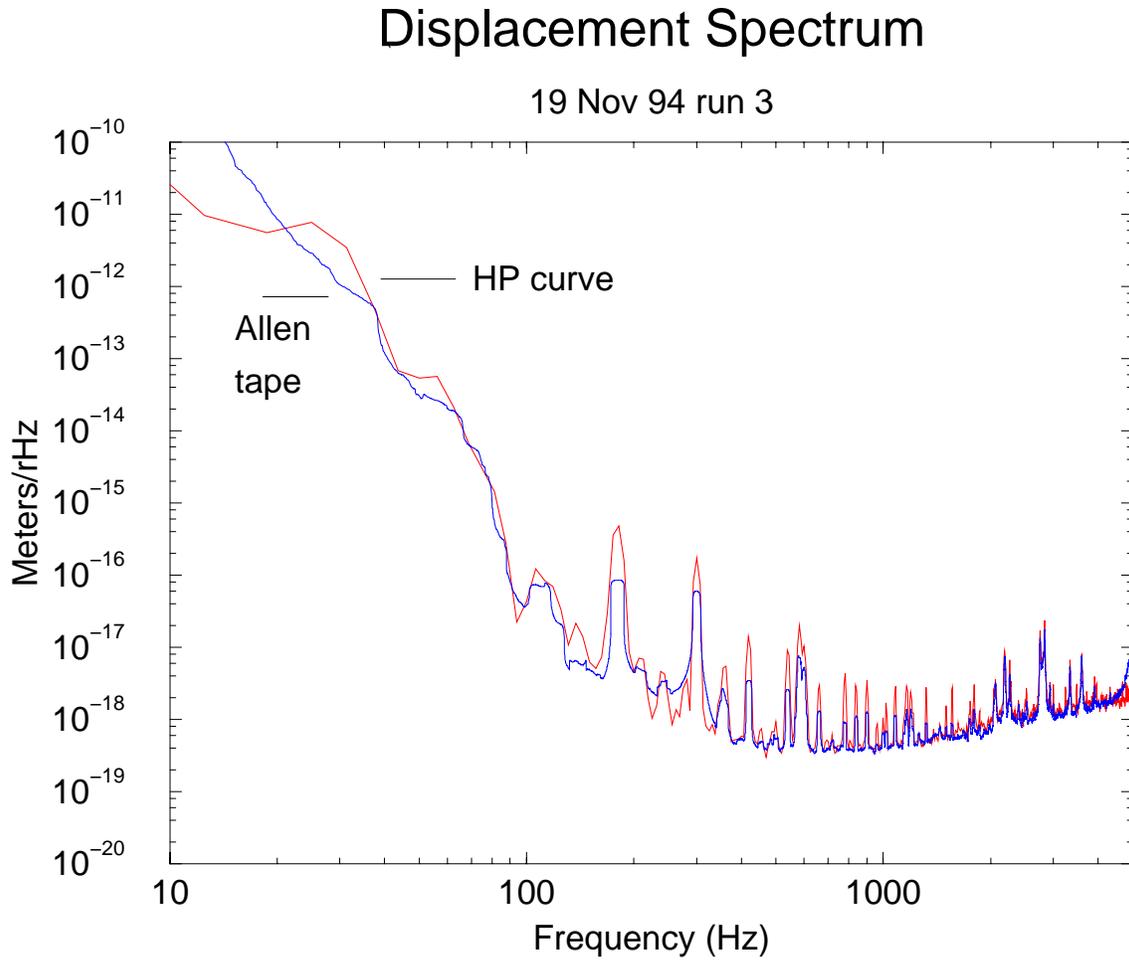


Figure 15: An example of a power spectrum curve produced with `power_spectrumF`. The spectrum produced off a data tape (with 100 point smoothing) is compared to that produced by the HP spectrum analyzer in the lab.

4.12 Example: calibrateF program

This example uses the function `GRnormalize()` and `avg_spec()` to produce an animated display, showing the properly normalized power spectrum of the interferometer, with a 30-second characteristic time moving average. After compilation, to run the program type:

```
setenv GRASP_FRAMEPATH /usr/local/GRASP/18nov94.1frame
calibrateF | xmgr -pipe &
```

to get an animated display showing the calibrated power spectrum changing. An example of the output from `calibrateF` is shown in Figure 16. Note that most of the execution time here is spent passing data down the pipe to `xmgr` and displaying it. The display can be speeded up by a factor of ten by binning the output values to reduce their number to a few hundred lines (the example program `calibrate_binnedF.c` implements this technique; it can be run by typing `calibrate_binnedF | xmgr -pipe`).

Calibrated IFO Spectrum

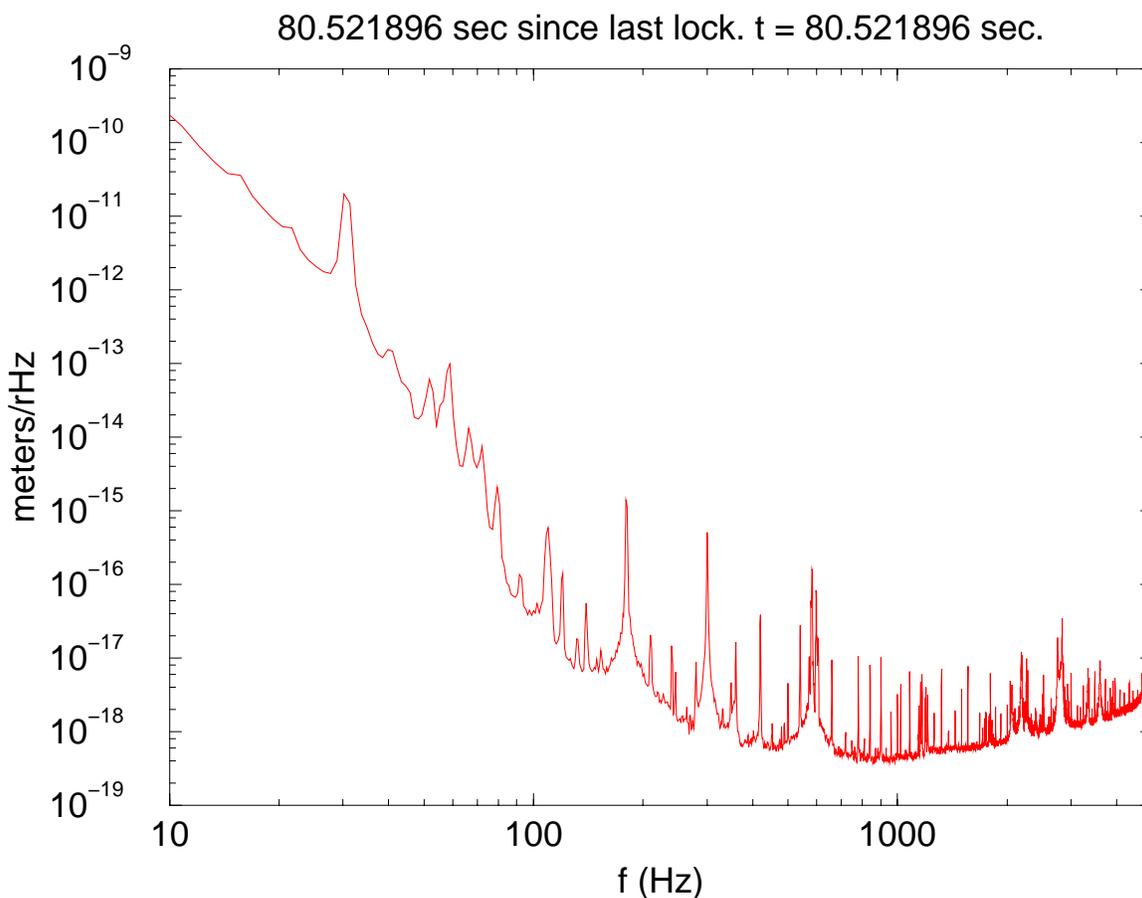


Figure 16: This shows a snapshot of the output from the program `calibrateF` which displays an animated average power spectrum (Welch windowed, 30-second decay time).

```
/* GRASP: Copyright 1997,1998 Bruce Allen */
#include "grasp.h"
#define NPOINT 4096
```

```
int main() {
    void graphout(int, float, float);
    float data[NPOINT], average[NPOINT/2], response[NPOINT+4];
    float spec, decaytime;
    float srate, tstart=0, freq, tlock=0.0;
    int i, j, code, npoint, ir, ii, reset=0, pass=0;
    short datas[NPOINT];
    double mod;
    struct fgetinput fgetinput;
    struct fgetoutput fgetoutput;

    /* number of channels needed is one */
    fgetinput.nchan=1;

    /* use utility function framefiles() to retrieve file names */
    fgetinput.files=framefiles;

    /* storage for channel names, data locations, points returned, ratios */
    fgetinput.chnames=(char **)malloc(fgetinput.nchan*sizeof(char *));
    fgetinput.locations=(short **)malloc(fgetinput.nchan*sizeof(short *));
    fgetoutput.npoint=(int *)malloc(fgetinput.nchan*sizeof(int));
    fgetoutput.ratios=(int *)malloc(fgetinput.nchan*sizeof(int));

    /* set up channel name */
    fgetinput.chnames[0]="IFO_DMRO";

    /* set up channel names for different cases */
    if (NULL!=getenv("GRASP_REALTIME")) {
        /* for Caltech 40-meter lab */
        fgetinput.inlock=0;
    }
    else {
        /* for Nov 1994 data set */
        fgetinput.inlock=1;
    }

    /* number of points to sample and fft (power of 2) */
    fgetinput.npoint=npoint=NPOINT;

    /* we do need the data, so don't seek */
    fgetinput.seek=0;

    /* do need calibration information */
    fgetinput.calibrate=1;

    /* where to put the data points */
    fgetinput.locations[0]=datas;

    /* set the decay time (sec) */
    decaytime=30.0;

    /* get data */
    while ((code=fget_ch(&fgetoutput, &fgetinput))) {
        tstart=fgetoutput.dt;
        srate=fgetoutput.srate;

        /* put data into floats */
        for (i=0; i<npoint; i++) data[i]=datas[i];
    }
}
```

```
/* use the swept-sine calibration (properly interpolated) to get R(f) */
if (!pass++) GRnormalize(fgetoutput.fri,fgetoutput.frinum,npoint,srate,response);

/* Reset if just locked */
if (code==1) {
    reset=0;
    tlock=tstart;
    avg_spec(data,average,npoint,&reset,srate,decaytime,2,1);
} else {

    /* track average power spectrum, with Welch windowing. */
    avg_spec(data,average,npoint,&reset,srate,decaytime,2,1);

    /* loop over all frequencies except DC (j=0) & Nyquist (j=npoint/2) */
    for (j=1;j<npoint/2;j++) {
        /* subscripts of real, imaginary parts */
        ii=(ir=j+j)+1;
        /* frequency of the point */
        freq=srate*j/npoint;
        /* determine power spectrum in (meters/rHz) & print it */
        mod=response[ir]*response[ir]+response[ii]*response[ii];
        spec=sqrt(average[j]*mod);
        printf("%e\t%e\n",freq,spec);
    }
    /* print out useful things for xmgr program ... */
    graphout(0,tstart,tlock);
}
}
return 0;
}

void graphout(int last,float time,float tlock) {
    static int count=0;
    printf("&\n"); /* end of set marker */
    /* first time we draw the plot */
    if (count++==0) {
        printf("@doublebuffer true\n"); /* keeps display from flashing */
        printf("@focus off\n"); /* turn off the focus markers */
        printf("@s0 color 2\n"); /* FFT is red */
        printf("@g0 type logxy\n"); /* set graph type to log-log */
        printf("@autoscale \n"); /* autoscale FFT */
        printf("@world xmin %e\n",10.0); /* set min x */
        printf("@world xmax %e\n",5000.0); /* set max x */
        printf("@world ymin %e\n",1.e-19); /* set min y */
        printf("@world ymax %e\n",1.e-9); /* set max y */
        printf("@yaxis tick minor on\n"); /* turn on tick marks */
        printf("@yaxis tick major on\n"); /* turn on tick marks */
        printf("@yaxis tick minor 2\n"); /* turn on tick marks */
        printf("@yaxis tick major 1\n"); /* turn on tick marks */
        printf("@redraw \n"); /* redraw graph */
        printf("@xaxis label \"f (Hz)\"\n"); /* FFT horizontal axis label */
        printf("@yaxis label \"meters/rHz\"\n"); /* FFT vertical axis label */
        printf("@title \"Calibrated IFO Spectrum\"\n"); /* set title */
        /* set subtitle */
        printf("@subtitle \"%0.2f sec since last lock. t = %0.2f sec.\"\n",time-tlock,time);
        if (!last) printf("@kill s0\n"); /* kill graph; ready to read again */
    }
}
```

```
else {
  /* other times we redraw the plot */
  /* set subtitle */
  printf("@subtitle \".2f sec since last lock. t = %.2f sec.\"\n",time-tlock,time);
  printf("@s0 color 2\n");
  printf("@g0 type logxy\n");
  printf("@world xmin %e\n",10.0);
  printf("@world xmax %e\n",5000.0);
  printf("@world ymin %e\n",1.e-19);
  printf("@world ymax %e\n",1.e-9);
  printf("@yaxis tick minor on\n");
  printf("@yaxis tick major on\n");
  printf("@yaxis tick minor 2\n");
  printf("@yaxis tick major 1\n");
  printf("@redraw\n");
  if (!last) printf("@kill s0\n");
}
return;
}
```

Author: Bruce Allen, ballen@dirac.phys.uwm.edu

Comments: See comments for power_spectrumF example program.

4.13 Example: transferF program

This example uses the function `GRnormalize()` to calculate the response of the interferometer to a specified gravitational-wave strain $h(t)$. [Note: for clarity, in this example, we have NOT worried about getting the overall normalization correct.] The code includes two possible $h(t)$'s. The first of these is a binary-inspiral chirp (see Section 6). Or, if you un-comment one line of code, you can see the response of the detector to a unit-impulse gravitational wave strain, in other words, the impulse response of the detector.

Note that to run this program, you must specify a path to the 40-meter data, for example by typing:

```
setenv GRASP_FRAMEPATH /usr/local/data/19nov94.3.frame
```

so that the code can find a frame containing a swept-sine calibration file to use.

The response of the detector to a pair of inspiraling stars is shown in Figure 17. You will notice that although the chirp starts at a (gravitational-wave) frequency of 140 Hz on the left-hand side of the figure, the low-frequency response of the detector is so poor that the chirp does not really become visible until about half-a-second later, at somewhat higher frequency. In the language of the audiophile, the IFO has crummy bass response! Of course this is entirely deliberate; the whitening filters of the instrument are designed to attenuate the low-frequency seismic contamination, and consequently also attenuate any possible low-frequency gravitational waves.

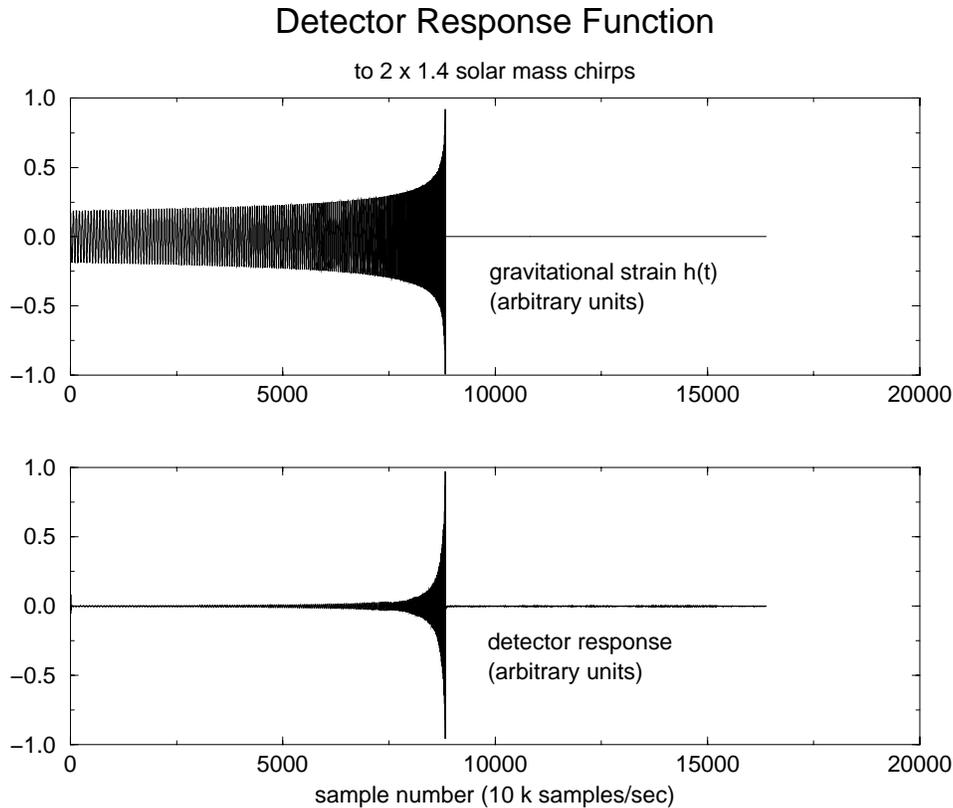


Figure 17: Output produced by the `transfer` example program. The top graph shows the gravitational-wave strain produced by an inspiraling binary pair. The lower graph shows the calculated interferometer output [channel.0 or IFO_DMRO] produced by this signal. Notice that because of the poor low-frequency response of the instrument, the IFO output does not show significant response before the input frequency has increased. The sample rate is slightly under 10 kHz.

The response of the detector to a unit gravitational strain impulse is shown as a function of time-offset in

Figure 18. Here the predominant effect is the ringing of the anti-aliasing filter. The impulse response of the detector lasts about 30 samples, or 3 msec. For negative offset times the impulse response is quite close to zero; its failure to vanish is partly a wrap-around effect, and partly due to errors in the actual measurement of the transfer function.

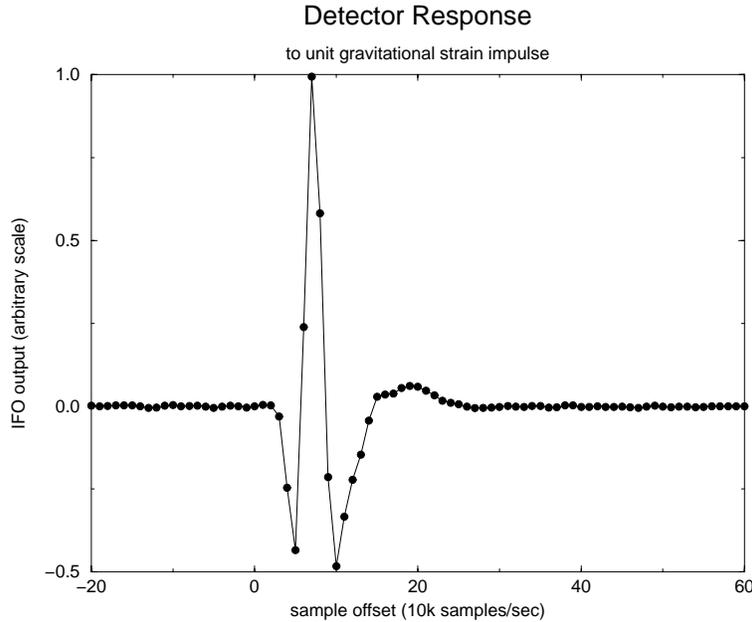


Figure 18: Output produced by the `transfer` example program. This shows the calculated interferometer output [channel.0 or IFO_DMRO] produced by an impulse in the gravitational-wave strain at sample number zero. This (almost) causal impulse response lasts about 3 msec.

This is a good place to insert a cautionary note. Now that we have determined the transfer function $R(f)$ of the instrument, you might be tempted to ask: “Why should I do any of my analysis in terms of the instrument output? After all, my real interest is in gravitational waves. So the first thing that I will do in my analysis is convert the instrument output into a gravitational wave strain $h(t)$ at the detector, by convolving the instrument’s output with (the time-domain version of) $R(f)$.” **Please do not make this mistake!** A few moment’s reflection will show why this is a remarkably bad idea. The problem is that the response function $R(f)$ is extremely large at low frequencies. This is just a reflection of the poor low frequency response of the instrument: any low-frequency energy in the IFO output corresponds to an extremely large amplitude low frequency gravitational wave. So, if you calculate $h(t)$ in the way described: take a stretch of (perhaps zero-padded) data, FFT it into the frequency domain, multiply it by $R(f)$ and invert the FFT to take it back into the frequency domain, you will discover the following:

- Your $h(t)$ is dominated by a single low-frequency noisy sinusoid (whose frequency is determined by the low frequency cutoff imposed by the length of your data segment or the low-frequency cutoff of the response function).
- Your $h(t)$ has *lost* all the interesting information present at frequencies where the detector is quiet (say, around 600 Hz). Because the noise power spectrum (see Figure 15) covers such a large dynamic range, you can not even represent $h(t)$ in a floating point variable (though it will fit, though barely, into a double). This is why the instrument uses a whitening filter in the first place.

- It is possible to construct " $h(t)$ " if you filter out the low-frequency garbage by setting $R(f)$ to zero below (say) 100 Hz.

If you are unconvinced by this, do the following exercise: calculate the power spectrum in the frequency domain as was done with Figure 15, then construct $h(t)$ in time domain, then take $h(t)$ back into the frequency domain, and graph the power spectrum again. You will discover that it has completely changed above 100 Hz and is entirely dominated by numerical quantization noise (round-off errors).

```
/* GRASP: Copyright 1997,1998 Bruce Allen */
#include <stdio.h>
#include <memory.h>
#include "grasp.h"
#define HSCALE 1.e20
#define NBINS 16384

int main() {
    float fstart,srate,tcoal,*c0,*c90,*response;
    int filled,i;
    void realft(float*,unsigned long, int);
    struct fgetinput fgetinput;
    struct fgetoutput fgetoutput;
    short int data;

    /* allocate memory */
    c0=(float*)malloc(sizeof(float)*NBINS);
    c90=(float*)malloc(sizeof(float)*NBINS);
    response=(float*)malloc(sizeof(float)*(NBINS+1));

    /* set start frequency, sample rate, make chirp */
    make_filters(1.4,1.4,c0,c90,fstart=140.0,NBINS,srate=9868.0,&filled,&tcoal,4000,4);
    printf("Chirp length is %d.\n",filled);

    /* Uncomment this line to see the impulse response of the instrument */
    /* for (i=0;i<NBINS;i++) c0[i]=0.0; c0[100]=1.0; */

    /* put chirps into frequency domain */
    realft(c0-1,NBINS,1);

    /* open frame, read one point, get calibration data, get response, and scale */
    fgetinput.nchan=1;
    fgetinput.files=framefiles;
    fgetinput.chnames=(char **)malloc(fgetinput.nchan*sizeof(char *));
    fgetinput.locations=(short **)malloc(fgetinput.nchan*sizeof(short *));
    fgetoutput.npoint=(int *)malloc(fgetinput.nchan*sizeof(int));
    fgetoutput.ratios=(int *)malloc(fgetinput.nchan*sizeof(int));
    fgetinput.chnames[0]="IFO_DMRO";
    fgetinput.inlock=0;
    fgetinput.npoint=fgetinput.seek=fgetinput.calibrate=1;
    fgetinput.locations[0]=&data;
    fget_ch(&fgetoutput,&fgetinput);
    GRnormalize(fgetoutput.fri,fgetoutput.frinum,NBINS,srate,response);
    for (i=0;i<NBINS;i++) response[i]*=HSCALE;

    /* avoid floating point errors in inversion */
    response[0]=response[1]=1.e10;

    /* determine IFO channel0 input which would have produced waveform */
```

```
ratio(c0,c0,response,NBINS/2);

/* invert FFT */
realft(c0-1,NBINS,-1);

/* make a graph showing channel.0 */
printf("File temp.graph contains channel.0 produced by 2 x 1.4 solar masses.\n");
graph(c0,NBINS,1);

return 0;
}
```

Author: Bruce Allen, ballen@dirac.phys.uwm.edu

Comments: None.

4.14 Example: diagF program

This program is a frequency-domain “novelty detector” and provides a simple example of a time-frequency diagnostic method. The actual code is not printed here, but may be found in the GRASP directory `src/examples/examples_frame` in the file `diagF.c`. To run the program type:

```
setenv GRASP_FRAMEPATH /usr/local/GRASP/18nov94.1frame
diagF &
```

which will start the `diagF` program in the background.

The method used by `diagF` is as follows:

1. A buffer is loaded with a short stretch of data samples (2048 in this example, about 1/5 of a second).
2. A (Welch-windowed) power spectrum is calculated from the data in the buffer. In each frequency bin, this provides a value $S(f)$.
3. Using the same auto-regressive averaging technique described in `avg_spec()` the mean value of $S(f)$ is maintained in a time-averaged spectrum $\langle S(f) \rangle$. The exponential-decay time constant for this average is `AVG_TIME` (10 seconds, in this example).
4. The absolute difference between the current spectrum and the average $\Delta S(f) \equiv |S(f) - \langle S(f) \rangle|$ is determined. Note that the absolute value used here provides a more robust first-order statistic than would be provided by a standard variance $(\Delta S(f))^2$.
5. Using the same auto-regressive averaging technique described in `avg_spec()` the value of $\Delta S(f)$ is maintained in a time-averaged absolute difference $\langle \Delta S(f) \rangle$. The exponential-decay time constant for this average is also set by `AVG_TIME`.
6. In each frequency bin, $\Delta S(f)$ is compared to $\langle \Delta S(f) \rangle$. If $\Delta S(f) > \text{THRESHOLD} \times \langle \Delta S(f) \rangle$ then a point is plotted for that frequency bin; otherwise no point is plotted for that frequency bin. In this example, `THRESHOLD` is set to 6.
7. In each frequency bin, $\Delta S(f)$ is compared to $\langle \Delta S(f) \rangle$. If $\Delta S(f) < \text{INCLUDE} \times \langle \Delta S(f) \rangle$ then the values of $S(f)$ and $\Delta S(f)$ are used to “refine” or “revise” the auto-regressive means described previously. In this example, `INCLUDE` is set to 10.
8. Another set of points (1024 in this example) is loaded into the end of the buffer, pushing out the oldest 1024 points from the start of the buffer, and the whole loop is restarted at step 2 above.

The `diagF` program can be used to analyze any of the different channels of fast-sampled data, by setting `CHANNEL` appropriately. It creates one output file for each locked segment of data. For example if `CHANNEL` is set to 0 (the IFO channel) and there are four locked sections of data, one obtains a set of files:

```
ch0diag.000, ch0diag.001, ch0diag.002, and ch0diag.003.
```

In similar fashion, if `CHANNEL` is set to 1 (the magnetometer) one obtains files:

```
ch1diag.000, ch1diag.001, ch1diag.002, and ch1diag.003.
```

These files may be used as input to the `xmgr` graphing program, by typing:

```
xmgr ch0diag.000 ch1diag.000
```

(one may specify as many channels as desired on the input line). A typical pair of outputs is shown in Figures 19 and 20. By specifying several different channels on the command line for starting `xmgr`, you can overlay the different channels output with one another. This provides a visual tool for identifying correlations between the channels (the graphs shown below may be overlaid in different colors).

Author: Bruce Allen, ballen@dirac.phys.uwm.edu

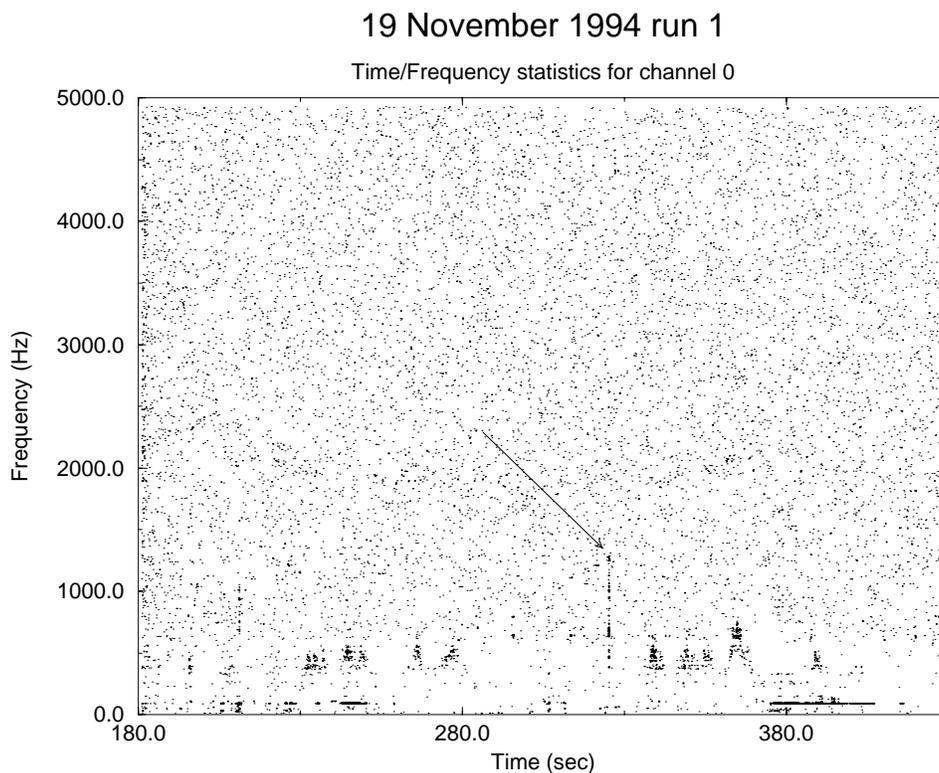


Figure 19: A time-frequency diagnostic graph produced by `diag`. The vertical line pointed to by the arrow is a non-stationary noise event in the IFO output, 325 seconds into the locked section. It sounds like a “drip” and might be due to off-axis modes in the interferometer optical cavities.

Comments: This type of time-frequency event detector appears quite useful as a diagnostic tool. It might be possible to improve its high-frequency time resolution by being clever about using intermediate information during the recursive calculation of the FFT. One should probably also experiment with using other statistical measures to assess the behavior of the different frequency bins. It would be nice to modify this program to also examine the slow sampled channels (see comment for `get_data()`).

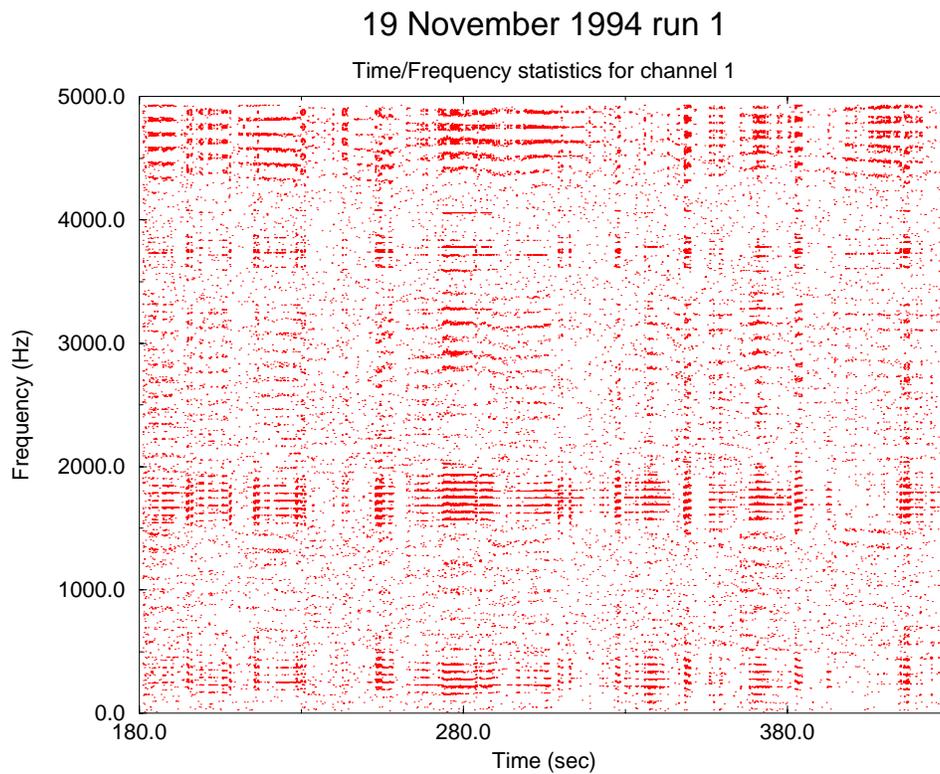


Figure 20: A time-frequency diagnostic graph produced by `diag`. This shows the identical period as the previous graph, but for the magnetometer output. Notice that the spurious event was not caused by magnetic field fluctuations.

4.15 Example: seismicF program

This is an example program which produces power spectra of seismometer ground motion. It is intended for use with a Guralp CMG-40T Broadband Seismometer <http://www.guralp.demon.co.uk/>, with the velocity sensing output set for the 0.033 → 100Hz band and the gain set to 400 volts-sec/meter. The normalization constants are defined with a line that reads something like this.

```
const float norm=((0.25/65536.0)*(1.0/400.0)*(1.0/(2.0*M_PI)));
```

Here the constants assume that the ADC that the seismometer is connected to records 65536 ADC counts per 0.25 volts input and that the gain (single-ended) is set to 400 volts-sec/meter. The 2π and a factor of frequency f arise in the code in converting velocity to position. The program outputs postscript and/or jpeg files labeled by GPS time.

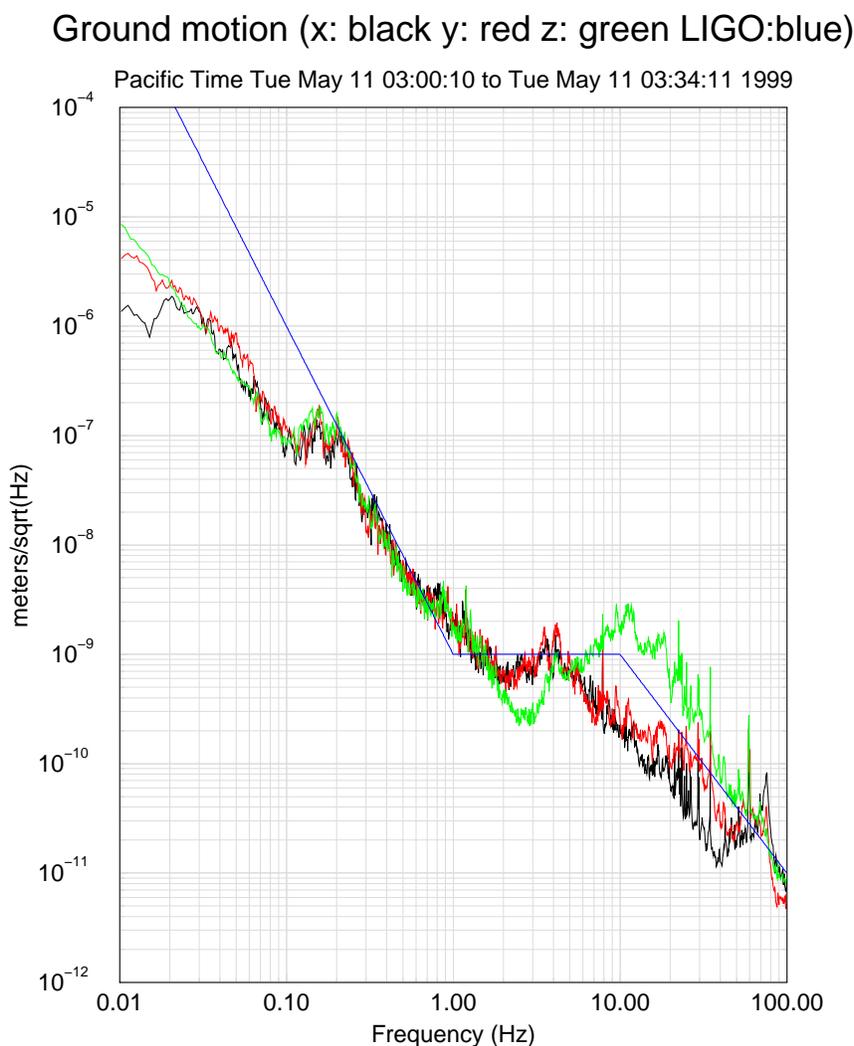


Figure 21: A seismometer (one-sided) power spectrum produced by the seismicF example program. The x,y, and z (vertical) motion are shown in black, red, and green. The blue curve shows the LIGO standard noise power spectrum, for comparison. This spectrum was taken at the LIGO Hanford site, during the installation. The portable clean rooms may be responsible for much of the excess noise. The file name is Spec.610452023.ps

5 GRASP Routines: Signal-to-noise enhancement techniques

5.1 Signal-to-noise enhancement by environmental cross-correlation

There are many situations of interest in which data are contaminated by the environment. Often this contamination is understood, and by monitoring the environment it is possible to “clean up” or “reduce” the data, by subtracting the effects of the environment from the signal or signals of interest. In the case of the data stream from an interferometric gravitational radiation detector, the signal of interest is the differential displacement of suspended test masses. This displacement arises from gravitational waves but also has contributions arising from other contaminating sources, such as the shaking of the optical tables (seismic noise) and forces due to ambient environmental magnetic fields. The key point is that the gravitational waves are not correlated with any of these environmental artifacts.

The method implemented here works by estimating the linear transfer function between the IFO_DMRO channel and specified environmental channels on the basis of the correlations over a certain bandwidth in Fourier space. The method is explained in detail in the paper ‘*Automatic cross-talk removal from multi-channel data*’ (WISC-MILW-99-TH-04)¹ Here we will just give a very brief overview to introduce the quantities calculated.

We denote the channel of interest, normally the InterFerOmeter Differential Mode Read-out (IFO_DMRO), by X or Y_1 . The other sampled channels consist of environmental and instrumental monitors which we denote Y_2, \dots, Y_N . We assume that all fast channels have been decimated so that all channels are sampled at the same (slow) rate, 986.842 \dots Hz for the November 1994 40-meter data.

We assume that the contribution of channel i to channel 1 is described by an (unknown!) linear transfer function $R_i(t - t')$. The basic idea of the method is to use the data to estimate the transfer functions R_i . For the reasons discussed in the paper, we work with the data in Fourier space. The transfer function is estimated by averaging over a frequency band, that is a given number of frequency bins. The number of bins in any band is denoted by F in the cited paper and `correlation_width` in the programs below. The method assumes that \tilde{R}_i can be well approximated by a *complex constant* within each frequency band, in other words that the transfer function does not vary rapidly over the frequency bandwidth $\Delta f = F/T$ where T is total time of the data section under consideration. The choices 32, 64 and 128 appear most appropriate for F for the 40-meter data.

Within a given band, b , the Fourier components of the field may be thought of as the components of an F -dimensional vector, $\mathbf{Y}_i^{(b)}$. Correlation between two channels (or the auto-correlation of a channel with itself) may be expressed by the standard inner product $(\mathbf{Y}_i^{(b)}, \mathbf{Y}_j^{(b)}) = \mathbf{Y}_i^{(b)} \cdot \mathbf{Y}_j^{(b)*}$ (no summation over b). Our assumption that \tilde{R}_i is constant over each band means that the ‘true’ channel of interest (the IFO_DMRO channel with environmental influences subtracted) can be written

$$\bar{\mathbf{x}}^{(b)} = \tilde{\mathbf{X}}^{(b)} - \sum_{j=2}^N r_j^{(b)} \tilde{\mathbf{Y}}_j^{(b)}. \quad (5.1.1)$$

where $r_j^{(b)}$, $j = 2, \dots, N$ are constants. The fundamental assumption is that the best estimate of the transfer function in the frequency band b is given by the complex vector $(r_2^{(b)}, \dots, r_N^{(b)})$ that minimises $|\bar{\mathbf{x}}^{(b)}|^2$. To measure the ‘improvement’ in the signal we define $|\rho|^2$ by

$$|\bar{\mathbf{x}}^{(b)}|^2 = |\tilde{\mathbf{X}}^{(b)}|^2 (1 - |\rho|^2). \quad (5.1.2)$$

denoted by `rho2` in the programs below. By definition $0 \leq |\rho|^2 \leq 1$. If any of the environmental channels

¹available from http://www.lsc-group.phys.uwm.edu/~www/docs/pub_table/gravpub.html.

are strongly correlated with the channel of interest, a significant reduction in $|\bar{\mathbf{x}}^{(b)}|^2$ is obtained, that is, $|\rho|^2$ will be close to 1.

To understand the origin of the ‘improvement’ it is also convenient to study the best estimate that can be obtained using any given single environmental channel. Thus we define

$$\bar{\mathbf{x}}_i^{(b)} = \tilde{\mathbf{X}}^{(b)} - r_i^{(b)} \tilde{\mathbf{Y}}_i^{(b)} \quad (5.1.3)$$

and choose the complex number $r_i^{(b)}$ to minimise $|\bar{\mathbf{x}}_i^{(b)}|^2$. Of course, in general this will not correspond to the i th component of the vector used in the multi-channel case. The corresponding improvement $|\rho_i|^2$ given by

$$|\bar{\mathbf{x}}_i^{(b)}|^2 = |\tilde{\mathbf{X}}^{(b)}|^2 (1 - |\rho_i|^2) \quad (5.1.4)$$

is denoted by `rho2_pairwise` in the programs below. By definition $0 \leq |\rho_i|^2 \leq 1$. If the i th environmental channel is strongly correlated with the channel of interest, a significant reduction in $|\bar{\mathbf{x}}_i^{(b)}|^2$ is obtained, that is, $|\rho_i|^2$ will be close to 1.

5.2 Outline

Calculation of environmental correlations using the routines presented in this chapter proceeds through the establishment of a configuration file, called here `40m.config` with the following structure:

```
# Correlations between 40m Channels over a period
# of approximately 266 seconds. The IFO sample rate is
# 9868.4\dots Hz (hence 9868.4 x 266 = 2621440 samples).
# The sample rate for the 'slow' channels is
# (1/10)th that of the 'fast' so (1/10)th the
# number of samples are requested
C1
4
IFO_DMRO      S      2621440
IFO_Mike      S      2621440
IFO_Seis_1    S      262144
IFO_SPZT      S      262144
1
```

The file may *begin* with any number of comment lines beginning with an initial #. The next line is a character string describing the detector, this is just used for naming intermediate files. The following line gives the total number of channels (signal plus environmental). There follow this number of lines each containing three columns, the first of these lines pertains to the signal the remainder to environmental channels. The three columns are:

1. the name of the channel,
2. the data type of the channel (here short) – see `animateT.c` or `corr_init.c` for a description of the possible types, and
3. the number of data points from that channel to be analysed – these should correspond to the same period of time.

Finally, there is a line containing a single number. This should be set to 1 if the user wants to obtain ‘cleaned’ output and 0 if the user just wants to see correlation data. (Note: This line is not used by `corr_init` described below, but only by `env_corr`. Thus it is possible to change one’s mind about whether to find the cleaned signal without having to rerun `corr_init`.)

The configuration file is used by the two basic programs:

1. `corr_init` which calculates the Fourier transforms and writes binary data files in a data directory named ‘configuration name’_fft, so `40m_fft` in the above example. Only those frequencies appropriate to the slowest channel are saved.
2. `env_corr` which calculates the correlations between each environmental channel and the signal channel and pops up a graph of these correlations. The data for this graph is stored in the same data directory as the FFT data. `env_corr` also produces a file `corr_view...` which enables this graph to be reproduced later without running `env_corr` again. If the configuration file asks for the signal to be cleaned `env_corr` will also produce an ASCII data file giving the (FFT of the) ‘cleaned’ signal and also the total fractional reduction in noise obtained by the method. Again this file is stored in the same data directory, its first line gives the frequency spacing and the following lines the real and imaginary parts of the FFT of the cleaned signal. (To avoid plotting difficulties with `xmgr` the DC component is arbitrarily set equal to that of the first bin.)

Thus, having created the appropriate configuration file one would type `corr_init 40m.config` and then `env_corr 40m.config`. (Of course, the environment variable `GRASP_FRAMEPATH` must first be set to the directory containing the appropriate frames.)

Note: These programs perform linear algebra by calls to `clapack` routines. These may be obtained from <http://www.netlib.org/>. These routines use `f2c` and, in particular, use complex numbers defined in `f2c.h` through the structure:

```
typedef struct {  
float r; /* real part */  
float i; /* imaginary part */  
} complex;
```

5.3 Function: `calc_rho()`

```
int calc_rho(int offset,int correlation_width,float threshold, float *rp_signal,
float *ip_signal,int nenv_chan, float **rp_env,float **ip_env,
float *rho2_pairwise, complex *A,complex *B,float *modx2sum)
```

This function takes sections of length `length` of the Fourier transform of the ‘signal’ channel and `nenv_chan` environmental channels and estimates the transfer function on the basis of correlations over a width of `correlation_width` bins at an offset frequency of `offset` bins. It returns an array `rho2_pairwise` containing the values of $|\rho|^2$ that would arise from a cross-correlation with each single channel thereby giving information on *which* environmental channels contain the strongest evidence for correlation with the signal.

The arguments are:

`length`: Input. The total length of the Fourier transform.

`offset`: Input. The offset to the beginning of the section of the Fourier transform over which correlations are being searched for.

`correlation_width`: Input. The width of the section of the Fourier transform over which correlations are being searched for.

`threshold`: Input. The threshold for determining whether a correlation is statistically significant.

`rp_signal`: Input. `rp_signal[0..length-1]` contains the real parts of the Fourier transform of the signal.

`ip_signal`: Input. `ip_signal[0..length-1]` contains the imaginary parts of the Fourier transform of the signal.

`nenv_chan`: Input. The number of environmental channels under consideration.

`rp_env`: Input. `rp_env[0..nenv_chan-1][0..length-1]` contains the real parts of the Fourier transform of the `nenv_chan` environmental channels.

`ip_env`: Input. `ip_env[0..nenv_chan-1][0..length-1]` contains the imaginary parts of the Fourier transform of the `nenv_chan` environmental channels.

`rho2_pairwise`: Output. `rho2_pairwise[0..nenv_chan-1]` contain the level of *pairwise* correlation between the signal and each environmental channel in turn, *i.e.*, the value of $|\rho|^2$ obtained if just the *i*th environmental channel had been used.

`A`: Input/Output. `A[0..nenv_chan2-1]` is a working array that is used by the *clapack* routine `chesv()` called by `clean_chan()`. The elements of the array are structure of type `complex` (see note above). Memory allocation should be performed in the calling routine.

`B`: Input/Output. `B[0..nenv_chan-1]` is a working array used by the *clapack* routine `chesv()` called by `clean_chan()`. The elements of the array are structure of type `complex` (see note above). Memory allocation should be performed in the calling routine.

`modx2sum`: Output. A float used by `clean_chan`.

Author: Bruce Allen (ballen@dirac.phys.uwm.edu), Wensheng Hua (hua@bondi.phys.uwm.edu) and Adrian Ottewill (ottewill@relativity.ucd.ie).

Comments:

5.4 Function `chan_clean()`

```
int chan_clean(int offset,int correlation_width,float threshold,float *rho2,
float *rp_signal,float *ip_signal, int nenv_chan,float **rp_env,float **ip_env,
float *rp_clean,float *ip_clean,complex *A,complex *B,
float modx2sum,complex *R,complex *work,integer lwork,integer *ipivot)
```

Note: The data type `integer` is defined in `f2c.h`.

`offset`: Input. The offset to the beginning of the section of the Fourier transform over which correlations are being searched for.

`correlation_width`: Input. The width of the section of the Fourier transform over which correlations are being searched for.

`threshold`: Input. The threshold for determining whether a correlation is statistically significant.

`rp_signal`: Input. `rp_signal[0..length-1]` contains the real parts of the Fourier transform of the signal.

`ip_signal`: Input. `rp_signal[0..length-1]` contains the imaginary parts of the Fourier transform of the signal.

`nenv_chan`: Input. The number of environmental channels under consideration.

`rp_env`: Input. `rp_env[0.. nenv_chan-1][0..length-1]` contains the real parts of the Fourier transform of the `nenv_chan` environmental channels.

`ip_env`: Input. `ip_env[0.. nenv_chan-1][0..length-1]` contains the imaginary parts of the Fourier transform of the `nenv_chan` environmental channels.

`rp_clean`: Output. `rp_clean[0..length-1]` contains the real parts of the Fourier transform of the signal cleaned by removing those contributions that have been assigned by the method to environmental influences.

`ip_clean`: Output. `ip_clean[0..length-1]` contains the imaginary parts of the Fourier transform of the signal cleaned by removing those contributions that have been assigned by the method to environmental influences.

`A`: Input. `A[0..nenv_chan2-1]` is a working array that is calculated by `calc_rho()` and used by the *clapack* routine `chesv()` called by `clean_chan()`. The elements of the array are structure of type `complex` (see note above). Memory allocation should be performed in the calling routine.

`B`: Input. `B[0..nenv_chan-1]` is a working array that is calculated by `calc_rho()` used by the *clapack* routine `chesv()` called by `clean_chan()`. The elements of the array are structure of type `complex` (see note above). Memory allocation should be performed in the calling routine.

`modx2sum`: Input. A float that is calculated by `calc_rho` used by `clean_chan`.

`R`: Input. `R[0..nenv_chan-1]` is a working array used by the *clapack* routine `chesv()` called by `clean_chan()`. The elements of the array are structure of type `complex` (see note above). Memory allocation should be performed in the calling routine.

`work`: Input. `work[0..lwork-1]` is a working array used by the *clapack* routine `chesv()` called by `clean_chan()`. The elements of the array are structure of type `complex` (see note above). Memory allocation should be performed in the calling routine.

`lwork`: Input. The size of the array `work`.

`ipivot`: Input. `ipivot[0..nenv_chan-1]` is a working array used by the *clapack* routine `chesv()` called by `clean_chan()`. The elements of the array are structure of type `integer`. Memory allocation should be performed in the calling routine.

Author: Bruce Allen (ballen@dirac.phys.uwm.edu), Wensheng Hua (hua@bondi.phys.uwm.edu) and Adrian Ottewill (ottewill@relativity.ucd.ie).

Comments: `clean_chan()` currently uses *clapack* routines to perform the required matrix manipulations. While this it is highly desirable to have such optimised routines when considering a large number of environmental channels it would also be useful to have a replacement for the *clapack* routine `chesv()` constructed from Numerical Recipes routines for when *clapack* is not available.

5.5 Example: Correlations in data from the 40m interferometer

The output below was produced starting with 2621440 samples from the 19 November 1994 run 3 data set, covering about 266 seconds.

The fast channels, including the IFO_DMRO channel, were decimated so that all channels are effectively sampled at the slow channel rate of 986.842... Hz. This yields a (real) time series with 262144 samples and correspondingly a (complex) FFT of length 131072 as specified by the macro name LENGTH. Averaging is carried out over 128 frequency bins but this may be varied to include a range of bandwidths MIN_BANDWIDTH and MAX_BANDWIDTH.

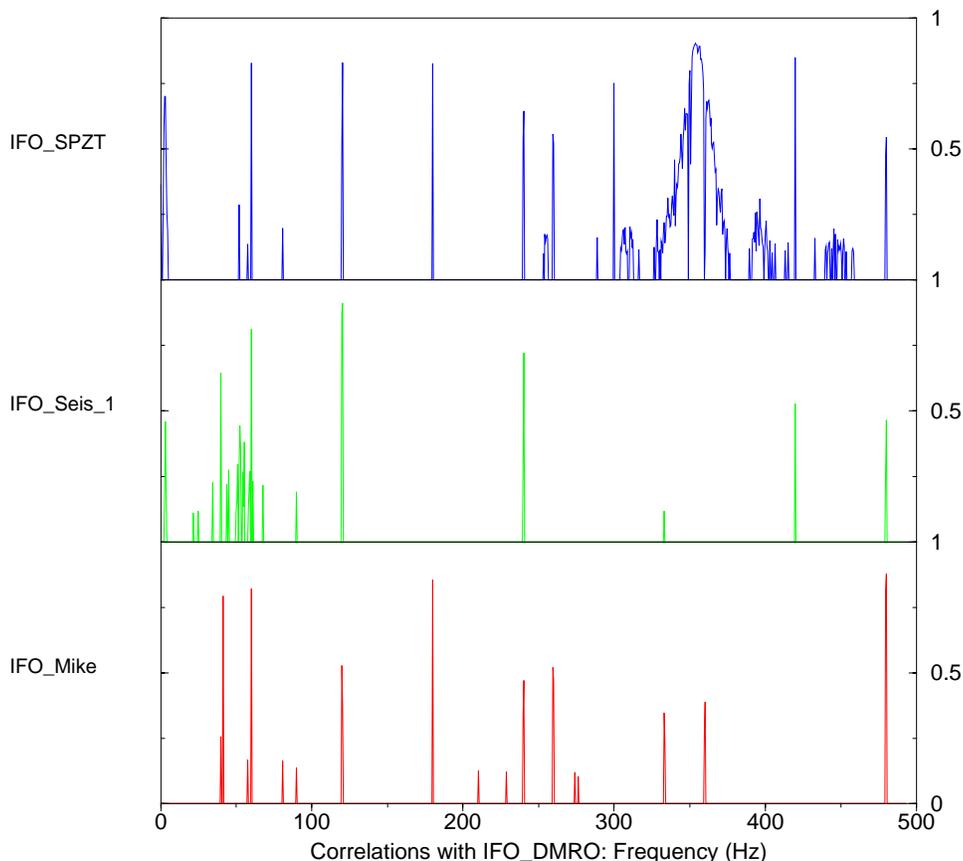


Figure 22: The graphical display of the contents of the output file `40m_fft/rho2_IFO_128.dat` produced by `env_corr` illustrating strong environmental cross-correlation. The three graphs show the correlation between the IFO_DMRO channel and each individual environmental channel. This graph is produced by the file `corr_view128` which `env_corr` produces.

The output was obtained from the commands

```
corr_init 40m.config
```

followed by

```
env_corr 40m.config
```

where `40m.config` is the configuration file printed above. This corresponding to seeking correlations between the interferometer output (IFO_DMRO) channel and

IFO_Mike: The microphone output.

IFO_Seis_1: The seismometer output.

IFO_SPZT: The slow pzt.

The data below show sections of the output file `40m_fft/rho2_IFO_128.dat` illustrating strong environmental cross-correlation at around 40Hz with the seismometer and microphone output, at around 120Hz with all three channels, and in a broad band around 360Hz with the slow pzt.

```
...
39.271      0.000      0.000 0.000 0.000
39.753      0.654      0.256 0.644 0.000
40.235      0.000      0.000 0.000 0.000
40.717      0.000      0.000 0.000 0.000
41.199      0.794      0.794 0.000 0.000
41.681      0.000      0.000 0.000 0.000
...
118.778     0.000      0.000 0.000 0.000
119.259     0.000      0.000 0.000 0.000
119.741     0.940      0.528 0.876 0.519
120.223     0.973      0.371 0.911 0.830
120.705     0.000      0.000 0.000 0.000
121.187     0.000      0.000 0.000 0.000
...
352.478     0.886      0.000 0.000 0.886
352.960     0.897      0.000 0.000 0.897
353.442     0.899      0.000 0.000 0.899
353.924     0.904      0.000 0.000 0.904
354.405     0.898      0.000 0.000 0.898
354.887     0.897      0.000 0.000 0.897
355.369     0.869      0.000 0.000 0.869
355.851     0.878      0.000 0.000 0.878
356.333     0.893      0.000 0.000 0.893
356.815     0.893      0.000 0.000 0.893
357.297     0.843      0.000 0.000 0.843
357.778     0.845      0.000 0.000 0.845
...
```

The output file `40m_fft/fftclean_IFO_128.dat` contains the Fourier transform of the corresponding signal 'cleaned' by estimating the transfer functions over a correlation width of 128 bins. Figure 23 shows the spectrum of the IFO_DMRO channel before and after 'cleaning' based on environmental channels 1, 2 and 5.

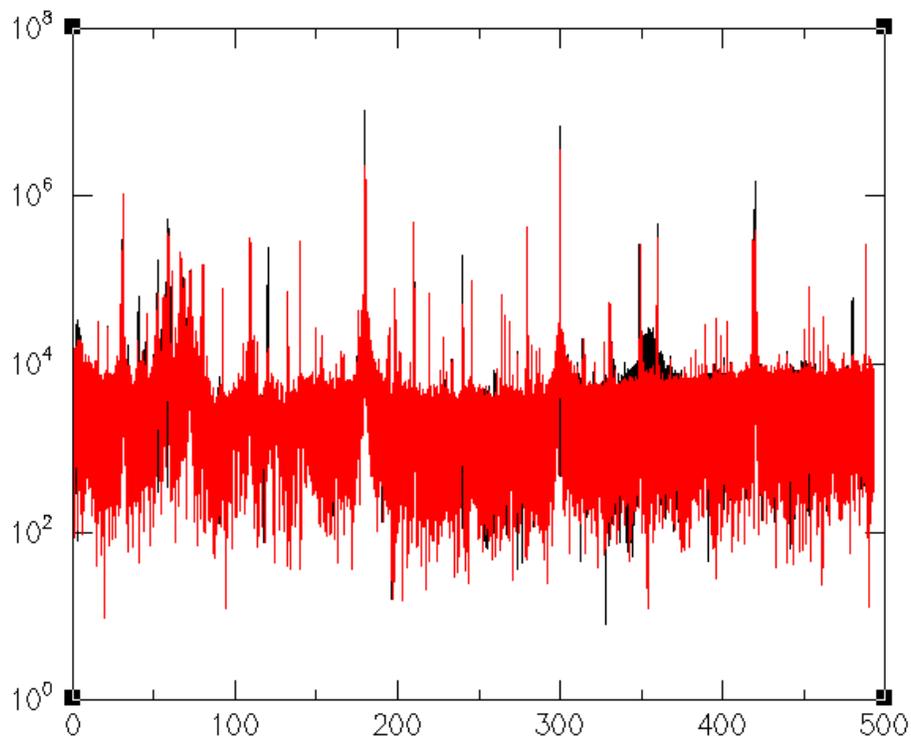


Figure 23: The spectrum of the IFO_DMRO channel before (black) and after ‘cleaning’ based on the three environmental channels discussed in the text using a correlation width of 128 bins (red).

5.6 Example: corr_init

This program calculates the FFTs of the various channels specified in the configuration file and stores them in binary files in a subdirectory of the working directory, whose name is determined by the detector name specified in the configuration file.

Typical usage: `corr_init 40m.config`

```
/* GRASP: Copyright 1997,1998 Bruce Allen */
#include "grasp.h"

int main(int argc, char *argv[]) {
    FILE *fp;
    struct fgetinput fgetinput;
    struct fgetoutput fgetoutput;
    float *data,delta_f;
    int *npoints;
    int i,j,check,min_points;
    char fname[256],detector[256],fft_dir[256],cmd[256];

    if ( argc != 2) {
        printf("Usage: corr_init configuration-file\n");
        exit(1);
    }

    fp = fopen(argv[1],"r");
    if ( fp == NULL ) {
        fprintf(stderr,"Problems opening %s\n",argv[1]);
        exit(1);
    }
    fprintf(stderr,"Reading %s\n",argv[1]);

    while (1) {
        fgets(detector,sizeof(detector),fp);
        if (detector[0] != '#') break;
    }
    detector[strlen(detector)-1]='\0';
    check=fscanf(fp,"%d",&fgetinput.nchan);

    /* storage for channel names, data locations, points returned, ratios */
    fgetinput.chnames=(char **)malloc(fgetinput.nchan*sizeof(char *));
    for (i=0;i<fgetinput.nchan;i++)
        fgetinput.chnames[i]=(char *)malloc(256*sizeof(char));
    fgetinput.locations=(short **)malloc(fgetinput.nchan*sizeof(short *));
    fgetoutput.npoint=(int *)malloc(fgetinput.nchan*sizeof(int));
    fgetoutput.ratios=(int *)malloc(fgetinput.nchan*sizeof(int));
    fgetinput.datatype=(char *)malloc(fgetinput.nchan*sizeof(char));
    npoints=(int *)malloc(fgetinput.nchan*sizeof(int));

    for (i=0;i<fgetinput.nchan;i++) {
        check=fscanf(fp,"%s %c %d",fgetinput.chnames[i],&fgetinput.datatype[i],&npoints[i]);
        /* the next fgetinput.nchan lines of the configuration file should contain 3 columns */
        /* - if not print an error message */
        if (check != 3) {
            fprintf(stderr,"Problems reading data from %s\n",argv[1]);
            exit(1);
        }
    }
}
```

```
}

fclose(fp);

/* number of points to get */
fgetinput.npoint=npoints[0];

/* allocate storage space for data */
for(i=0;i<fgetinput.nchan;i++) {
  switch (fgetinput.datatype[i]) {
    case 'S': /* short data */
    case 'u': /* unsigned short data */
      fgetinput.locations[i]=(short *)malloc(npoints[i]*sizeof(short));
      break;
    case 'I': /* integer data */
    case 'i': /* unsigned integer data */
      fgetinput.locations[i]=(short *)malloc(npoints[i]*sizeof(int));
      break;
    case 'L': /* long data */
    case 'l': /* unsigned long data */
      fgetinput.locations[i]=(short *)malloc(npoints[i]*sizeof(long));
      break;
    case 'F': /* float data */
      fgetinput.locations[i]=(short *)malloc(npoints[i]*sizeof(float));
      break;
    case 'D': /* double data */
      fgetinput.locations[i]=(short *)malloc(npoints[i]*sizeof(double));
      break;
    case 'C': /* character data */
    case 'f': /* complex float data */
    case 'd': /* complex double data */
    case 's': /* character string data */
    case 'c': /* unsigned character data */
      fprintf(stderr,"Data type %c cannot be plotted\n",fgetinput.datatype[0]);
      break;
    default:
      fprintf(stderr,"Unknown data type %c\n",fgetinput.datatype[0]);
      exit(1);
  }
}

/* don't have inlock channel if not 40m */
if (detector != "C1") {
  fgetinput.inlock=0;
}

/* but we don't need calibration information */
fgetinput.calibrate=0;

/* don't seek, we need the sample values! */
fgetinput.seek=0;

/* source of files */
fgetinput.files=framefiles;

check=fget_ch(&fgetoutput,&fgetinput);
if ( check == 0) {
  fprintf(stderr,"not enough data!!!!!!!!!!");
  exit(1);
}
```

```
}
fprintf(stderr, "Signal (%s) sample rate is %f\n", fgetinput.chnames[0], fgetoutput.srate);

min_points=npoints[0];
for(i=1;i<fgetinput.nchan;i++) {
    if (npoints[i]<min_points) min_points=npoints[i];
}

delta_f=fgetoutput.srate/npoints[0];

for (i=0;i<256;i++) {
    if (argv[1][i] == '\0' || argv[1][i] == '.') {
        fft_dir[i] = '\0';
        break;
    }
    fft_dir[i]=argv[1][i];
}

strcat(fft_dir, "_fft");
sprintf(cmd, "mkdir %s 2> /dev/null", fft_dir);
system(cmd);

for (i=0;i<fgetinput.nchan;i++) {
    /* check frames are consistent with configuration file */
    if (npoints[i] != fgetoutput.srate/fgetoutput.ratios[i]) {
        fprintf(stderr, "Sample rates in %s is %f\n",
            fgetinput.chnames[i], fgetoutput.srate/fgetoutput.ratios[i]);
    }

    /* assign memory for storing data in floats */
    data=(float *)malloc(sizeof(float)*npoints[i]);

    switch (fgetinput.datatype[i]) {
    case 'S': /* short data */
    case 'u': /* unsigned short data */
        for(j=0;j<npoints[i];j++) data[j]=fgetinput.locations[i][j];
        break;
    case 'I': /* integer data */
    case 'i': /* unsigned integer data */
        for(j=0;j<npoints[i];j++) data[j]=((int *) (fgetinput.locations[i]))[j];
        break;
    case 'L': /* long data */
    case 'l': /* unsigned long data */
        for(j=0;j<npoints[i];j++) data[j]=((long *) (fgetinput.locations[i]))[j];
        break;
    case 'F': /* float data */
        for(j=0;j<npoints[i];j++) data[j]=((float *) (fgetinput.locations[i]))[j];
        break;
    case 'D': /* double data */
        for(j=0;j<npoints[i];j++) data[j]=((double *) (fgetinput.locations[i]))[j];
        break;
    case 'C': /* character data */
    case 'f': /* complex float data */
    case 'd': /* complex double data */
    case 's': /* character string data */
    case 'c': /* unsigned character data */
        fprintf(stderr, "Data type %c cannot be converted to float\n", fgetinput.datatype[0]);
        break;
    default:
```

```
    fprintf(stderr, "Unknown data type %c\n", fgetinput.datatype[0]);
    exit(1);
}

/* We can now free fgetinput.locations[i] */
free(fgetinput.locations[i]);
/* Take the Fourier transforms */
realft(data-1, npoints[i], 1);
data[1]=0;

/* Print the Fourier transform to file */
sprintf(fname, "%s/%s-%s_fft.%i", fft_dir, detector, fgetinput.chnames[i]);
fp = fopen(fname, "wb");
if ( fp == NULL) {
    printf("Cannot open %s\n", fname);
    exit(1);
}
fprintf(stderr, "writing %s\n", fname);
fwrite(&delta_f, sizeof(float), 1, fp);
fwrite(data, sizeof(float), min_points, fp);
fclose(fp);

    free(data);
}

/* free memory */
free(fgetinput.locations);
free(fgetinput.chnames);
free(fgetinput.locations);
free(fgetoutput.npoint);
free(fgetoutput.ratios);
free(npoints);

return 0;
}
```

5.7 Example: env_corr

This program calls `calc_rho` and `clean_chan` to determine environmental correlations. It pops up a graph plotting these correlation as well as writing files containing data on the correlations and the 'cleaned' signal.

Typical usage: `env_corr 40m.config`

```
#include "grasp.h"
static char *rcsid="Development code";
#if defined (CLAPACK)
#include "f2c.h"
extern int chan_clean(int,int,float,float *,float *,float *,int,float **,float **,float *,float *,
                    complex *,complex *,float,complex *,complex *,integer,integer *);
#else
typedef float real;
typedef struct { real r, i; } complex; /* from f2c.h */
#endif

#define MIN(x,y) ((x) < (y) ? (x) : (y) )
#define MIN_BANDWIDTH 128
#define MAX_BANDWIDTH 128

void fopen_check(FILE *fp,char *fname);
char fft_dir[256],signal_name[256];

int main(int argc, char *argv[])
{
    int calc_rho(int,int,float,float *,float *,int,float **,float **,float *,
                complex *,complex *,float *);
    void read_binary_fft(char *fname,int length,float *rp_fft,float *ip_fft,float *delta_f);
    void write_fft(char *fname,int length,float *rp_fft,float *ip_fft,float delta_f);
    void write_rho2(FILE *fp,float freq,float rho2,int nenv_chan,float *rho2_pairwise);
    void *errmalloc(char *arrayname,size_t bytes);
    void xmgr_files(int nenv_chan,char **chnames,int correlation_width);
    FILE *fp,*fp_rho2;
    char fname[256],rho2_fname[256],detector[256],cmd[256]**chnames,temp;
    complex *A=NULL, *B=NULL;
    float **rp_env,**ip_env;
    float *rp_signal,*ip_signal,*rho2_pairwise,*rp_clean,*ip_clean;
    float threshold,rho2,modx2sum,delta_f,freq;
    int *chpts;
    int i,clean,offset,correlation_width,chan,n_chan,nenv_chan,length,check;
#if defined (CLAPACK)
    complex *R=NULL,*work=NULL;
    integer lwork,*ipivot=NULL;
#endif
    /******
    /* Read the configuration file */
    /******

    if ( argc != 2) {
        printf("Usage: %s configuration-file-name\n",argv[0]);
        exit(1);
    }
}
```

```
fp=fopen(argv[1], "r");
fopen_check(fp, argv[1]);

/* Comments start with a # - */
/* the first line after any comments should contain the detector name */

while (1) {
    fgets(detector, sizeof(detector), fp);
    if (detector[0] != '#') break;
}
detector[strlen(detector)-1]='\0';

/* the next line gives the total number of channels including the 'signal' */
check=fscanf(fp, "%d", &n_chan);
if (check != 1) {
    fprintf(stderr, "Problems reading number of channels from %s\n", argv[1]);
    exit(1);
}

chnames=(char **)errmalloc("chnames", n_chan*sizeof(char *));
for (i=0; i<n_chan; i++)
    chnames[i]=(char *)malloc(256*sizeof(char));
chpts=(int *)errmalloc("ip_signal", n_chan*sizeof(int));

for (i=0; i<n_chan; i++) {
    check=fscanf(fp, "%s %c %d", chnames[i], &temp, &chpts[i]);
    /* the next n_chan lines of the configuration file should contain 3 columns */
    /* - if not print an error message */
    /* the 3 columns are the channel name, the data type and the number of points */
    /* the data type is important for corr_init but is irrelevant here */
    if (check != 3) {
        fprintf(stderr, "Problems reading 3-columns: channel-name data-type number-of-samples from %s\n",
            argv[1]);
        exit(1);
    }
}

/* do we want to calculate (the fft of) the 'cleaned' signal */
/* clean should be set to 1 if we want to calculate the cleaned signal */
check=fscanf(fp, "%d", &clean);
if (check != 1) {
    fprintf(stderr, "Problems reading 'clean' bit from %s\n", argv[1]);
    exit(1);
}

fclose(fp);

/*****
/* Determine the number of frequency bins we will use */
*****/

length=chpts[0];
for(i=1; i<n_chan; i++)
    if (chpts[i]<length) length=chpts[i];

length/=2;          /* Because length is the length of the complex FFT array */

if ((length%MAX_BANDWIDTH) !=0) {
    length-=length%MAX_BANDWIDTH;
    fprintf(stderr, "Using %d FFT values (MAX_BANDWIDTH = %d)\n", length, MAX_BANDWIDTH);
}
```

```
}

strcpy(signal_name, chnames[0]);
nenv_chan=n_chan-1; /* because the configuration file includes the signal channel */

/*****
/* Allocate memory */
*****/

/* Allocate memory for |x_i|^2, |y_i|^2 and x_i y_i* */

rp_signal=(float *)errmalloc("rp_signal", length*sizeof(float));
ip_signal=(float *)errmalloc("ip_signal", length*sizeof(float));
rp_env=(float **)errmalloc("rows of rp_env", nenv_chan*sizeof(float *));
rp_env[0]=(float *)errmalloc("rows of rp_env[0]", nenv_chan*length*sizeof(float));
for (i=1; i<nenv_chan; i++)
    rp_env[i]=rp_env[i-1]+length;

ip_env=(float **)errmalloc("rows of ip_env", nenv_chan*sizeof(float *));
ip_env[0]=(float *)errmalloc("rows of ip_env[0]", nenv_chan*length*sizeof(float));
for (i=1; i<nenv_chan; i++)
    ip_env[i]=ip_env[i-1]+length;

if (clean) {
    rp_clean=(float *)errmalloc("rp_clean", length*sizeof(float));
    ip_clean=(float *)errmalloc("ip_clean", length*sizeof(float));
}

rho2_pairwise=(float *)errmalloc("rho2_pairwise", nenv_chan*sizeof(float));

/*****
/* Allocate memory for lapack arrays */
*****/

A=(complex *)errmalloc("A", nenv_chan*nenv_chan*sizeof(complex));
B=(complex *)errmalloc("B", nenv_chan*sizeof(complex));
if (clean) {
#ifdef CLAPACK
    lwork=(integer) nenv_chan;
    R=(complex *)errmalloc("R", nenv_chan*sizeof(complex));
    work=(complex *)errmalloc("work", lwork*sizeof(complex));
    ipivot=(integer *)errmalloc("ipivot", nenv_chan*sizeof(integer));
#endif
}

/*****
*****/
/* Read data from file */
/*****
*****/

/*****
/* Determine name of data directory */
*****/
for (i=0; i<256; i++) {
    if (argv[1][i] == '\\0' || argv[1][i] == '.') {
        fft_dir[i] = '\\0';
        break;
    }
}
```

```
    fft_dir[i]=argv[1][i];
}
strcat(fft_dir, "_fft");

/*****
/* Read data from signal channel */
*****/
sprintf(fname, "%s/%s-%s_fft.b", fft_dir, detector, chnames[0]);
read_binary_fft(fname, length, rp_signal, ip_signal, &delta_f);

/*****
/* Read data from environmental channels */
*****/
for (chan=0; chan<nenv_chan; chan++) {
    sprintf(fname, "%s/%s-%s_fft.b", fft_dir, detector, chnames[chan+1]);
    read_binary_fft(fname, length, rp_env[chan], ip_env[chan], &delta_f);
}

/*****
/* Cycle through range of bandwidths */
*****/
for (correlation_width=MIN_BANDWIDTH; correlation_width<=MAX_BANDWIDTH; correlation_width*=2) {

    threshold=MIN(0.1, 5.0/correlation_width); /* see discussion in Hua et al. */

    sprintf(rho2_fname, "%s/rho2_%s_%d.dat", fft_dir, signal_name, correlation_width);
    fp_rho2 = fopen(rho2_fname, "w");
    fopen_check(fp_rho2, rho2_fname);
    fprintf(stderr, "Writing %s\n", rho2_fname);

    /*****
    /* Step through the range of offsets and call the */
    /* function calc_rho where the major calculation is done */
    *****/

    for (offset=0; offset<length; offset+=correlation_width) {

        calc_rho(offset, correlation_width, threshold, rp_signal, ip_signal, nenv_chan,
                rp_env, ip_env, rho2_pairwise, A, B, &modx2sum);

        /*****
        /* Calculate the 'cleaned' signal */
        *****/

        rho2=0.0; /* set rho2=0.0 if we don't clean */
        if (clean) {
#ifdef CLAPACK
            chan_clean(offset, correlation_width, threshold, &rho2, rp_signal, ip_signal, nenv_chan,
                    rp_env, ip_env, rp_clean, ip_clean, A, B, modx2sum, R, work, lwork, ipivot);
#else
            fprintf(stderr, "Sorry cannot calculate cleaned channel without clapack installed\n");
            clean=0;
#endif
        }

        /*****
        /* Print out |rho|^2 and the level of *pairwise* correlation between */
        /* the signal and each environmental channel in turn */
        *****/
    }
}
}
```

```
/******  
    freq=(offset+0.5*correlation_width)*delta_f;  
    write_rho2(fp_rho2,freq,rho2,nenv_chan,rho2_pairwise);  
  
}    /* end of offset loop */  
  
if (clean) {  
    sprintf(fname,"%s/ftclean_%s_%d.dat",fft_dir,signal_name,correlation_width);  
    write_fft(fname,length,rp_clean,ip_clean,delta_f);  
}  
  
/******  
/* Write xmgr parameter and shell files */  
/******  
  
xmgr_files(nenv_chan,chnames,correlation_width);  
sprintf(cmd,"chmod +x corr_view%d",correlation_width);  
system(cmd);  
sprintf(cmd,"corr_view%d &",correlation_width);  
system(cmd);  
  
} /* end of correlation_width loop */  
  
/******  
/* Free memory */  
/******  
free(rp_signal);  
free(ip_signal);  
free(rp_env[0]);  
free(rp_env);  
free(ip_env[0]);  
free(ip_env);  
free(rp_clean);  
free(ip_clean);  
free(rho2_pairwise);  
/* free lapack arrays */  
free(A);  
free(B);  
#if defined (CLAPACK)  
    free(R);  
    free(work);  
    free(ipivot);  
#endif  
  
    return(0);  
}  
  
void read_binary_fft(char *fname,int length,float *rp_fft,float *ip_fft,float *delta_f)  
{  
    /******  
    /* Read fft data from binary file: the first line contains the frequency spacing */  
    /* then follow lines containing the real and imaginary part */  
    /******  
    FILE *fp;
```

```
int i,check1,check2;

fp = fopen(fname,"rb");
fopen_check(fp,fname);
fprintf(stderr,"Reading %s\n",fname);

/* first read the frequency spacing */
check1=fread(delta_f,sizeof(float),1,fp);
if (check1 != 1) {
    fprintf(stderr,"Problems reading delta_f from %s\n",fname);
    exit(1);
}
for (i=0;i<length;i++) {
    check1=fread((rp_fft+i),sizeof(float),1,fp);
    check2=fread((ip_fft+i),sizeof(float),1,fp);
    if ((check1 != 1) || (check2 != 1)) {
        fprintf(stderr,"Problems reading delta_f from %s\n",fname);
        exit(1);
    }
}

fclose(fp);
rp_fft[0]=ip_fft[0]=0.0; /* set dc signal to 0 */
}

void write_fft(char *fname,int length,float *rp_fft,float *ip_fft,float delta_f)
{
    /******
    /* Write fft data to an ascii file: the first line contains the frequency spacing */
    /* then follow lines containing the real and imaginary part */
    /******
    FILE *fp;
    int i;

    fp = fopen(fname,"w");
    fopen_check(fp,fname);
    fprintf(stderr,"Writing %s\n",fname);

    fprintf(fp,"%f\n",delta_f);
    /* We fill the DC component with data from the first frequency bin so that
    we can do lin-log plots in xmgr without complaints.
    Note that the DC component is never used and is set to zero
    by read_fft - still there should be a better way of doing this! */

    fprintf(fp,"%f\t%f\n",rp_fft[1],ip_fft[1]);

    for (i=1;i<length;i++) {
        fprintf(fp,"%f\t%f\n",rp_fft[i],ip_fft[i]);
    }
    fclose(fp);
}

void write_rho2(FILE *fp,float freq,float rho2,int nenv_chan,float *rho2_pairwise)
{
    /******
    /* Write the list of rho values to file */
    /******
```

```
int chan;

fprintf(fp, "%.3f\t\t%.3f\t\t", freq, rho2);
for (chan=0;chan<nenv_chan;chan++) {
    fprintf(fp, "%.3f ", rho2_pairwise[chan]);
}
fprintf(fp, "\n");
}

void fopen_check(FILE *fp, char *fname)
{
    /******
    /* Checks to see if a file has been opened properly */
    /* and if not write an appropriate error message */
    /******
    if ( fp == NULL ) {
        fprintf(stderr, "Problems opening %s\n", fname);
        exit(1);
    }
}

void *errmalloc(char *arrayname, size_t bytes)
{
    /******
    /* Allocate memory and print an error message if unsuccessful */
    /******
    void *pointer;
    pointer=malloc(bytes);
    if (pointer==NULL) {
        fprintf(stderr, "Cannot allocate %d bytes of memory for %s\n", (int) bytes, arrayname);
        exit(1);
    }
    return pointer;
}

void xmgr_files(int nenv_chan, char **chnames, int correlation_width)
{
    /******
    /* Write the xmgr parameter files */
    /******
    FILE *fp;
    char param_fname[256], view_fname[256];
    int i;

    for (i=0;i<nenv_chan;i++) {
        sprintf(param_fname, "xmgr.param%d_%d", correlation_width, i);

        fp = fopen(param_fname, "w");
        fopen_check(fp, param_fname);
        fprintf(stderr, "Writing %s\n", param_fname);

        fprintf(fp, "focus g%d\n", i);
        fprintf(fp, "autoscale\n");
        if (i == 0) {
            fprintf(fp, "xaxis label \"Correlations with %s: Frequency (Hz)\"\n",
                    chnames[0]);
            fprintf(fp, "xaxis tick out\n");
        }
    }
}
```

```
    fprintf(fp,"xaxis  tick op bottom\n");
}
else {
    fprintf(fp,"xaxis  tick major off\n");
    fprintf(fp,"xaxis  tick minor off\n");
    fprintf(fp,"xaxis  ticklabel off\n");
    fprintf(fp,"yaxis  ticklabel start type spec\n");
    fprintf(fp,"yaxis  ticklabel start 0.5\n");
}
fprintf(fp,"yaxis  label layout perp\n");
fprintf(fp,"yaxis  label char size 0.9\n");
fprintf(fp,"yaxis  label place spec\n");
fprintf(fp,"yaxis  label place -0.84, 0\n");
fprintf(fp,"yaxis  ticklabel op right\n");

fprintf(fp,"yaxis label \"%s\"\n",chnames[i+1]);
fprintf(fp,"yaxis  tick major 0.5\n");
fprintf(fp,"yaxis  tick minor 0.25\n");
fprintf(fp,"view ymin %f\n",0.1+0.88*i/nenv_chan);
fprintf(fp,"view ymax %f\n",0.1+0.88*(i+1)/nenv_chan);
fprintf(fp,"world ymin 0\n");
fprintf(fp,"world ymax 1\n");

fprintf(fp,"s0 color %d\n",i+2);
fprintf(fp,"\n");
fclose(fp);
}

/*****
/* Write the xmgr shell files */
*****/

sprintf(view_fname,"corr_view%d",correlation_width);
fp = fopen(view_fname,"w");
fopen_check(fp,view_fname);
fprintf(stderr,"Writing %s\n",view_fname);

fprintf(fp,"xmgr -block %s/rho2_%s_%d.dat ",
        fft_dir,signal_name,correlation_width);

for (i=0;i<nenv_chan;i++) {
    sprintf(param_fname,"xmgr.param%d_%d",correlation_width,i);
    fprintf(fp,"-graph %d -bxy 1:%d -param %s ",i,i+3,param_fname);
}
fprintf(fp,"&\n");

fclose(fp);
}
```

Author: Bruce Allen (ballen@dirac.phys.uwm.edu), Wensheng Hua (hua@bondi.phys.uwm.edu) and Adrian Ottewill (ottewill@relativity.ucd.ie).

Comments: None.

6 GRASP Routines: Gravitational Radiation from Binary Inspiral

One of the principal sources of gravitational radiation which should be detectable with the first or second generation of interferometric detectors is *binary inspiral*. This radiation is produced by a pair of massive and compact orbiting objects, such as neutron stars or black holes.

The simplest case is when the two objects are describing a circular orbit about their common center-of-mass, and neither object is spinning about its own axis. With these assumptions the system is then described, at any time, by the masses m_1 and m_2 of the objects, and their orbital frequency Ω . (It is also necessary to describe the orientation of the orbital plane and the positions of the masses at a given time; these are details we will sort out later).

For convenience in dealing with dimensional quantities, we introduce the *Solar Mass* M_\odot and the *Solar Time* T_\odot defined by

$$M_\odot = 1.989 \times 10^{33} \text{ grams} \quad (6.0.1)$$

$$T_\odot = \left(\frac{G}{c^3}\right) M_\odot = 4.925491 \times 10^{-6} \text{ sec.} \quad (6.0.2)$$

GRASP functions typically measure masses in units of M_\odot and times in units of seconds.

6.1 Chirp generation routines

The next several subsections document a number of routines for generating “chirps” from coalescing binaries. This package of routines is intended to be versatile, flexible and robust; and yet still fairly simple to use. The implementation we have included in this package is based on the second post-Newtonian treatment of binary inspiral presented in [7] and augmented by the spin-orbit and spin-spin corrections presented in [8] and 2.5 post-Newtonian order corrections in [9]. The notation we use – even in the source code – closely reflects the notation used in those papers. In keeping with that notation, these routines calculate the **orbital phase** and **orbital frequency**. The gravitational-wave phase of the dominant quadrupolar radiation can be obtained by multiplying the orbital phase by two. The routines can be used to compute a few chirp waveforms (say to make transparencies for a seminar), or for wholesale computations of a bank of matched filters.

All of the chirp generation routines, in particular the ubiquitous `make_filters()`, compute what has come to be known as **“restricted” post-Newtonian chirps**. This means they include all post-Newtonian corrections (up to the specified order) in the phase evolution, but only the dominant quadrupole amplitude.

The routines are flexible in the sense that they have a number of *run-time* options available for choosing the post-Newtonian order of the phase calculations, or choosing whether or not to include spin effects. We have also isolated those parts of the code where the messy post-Newtonian coefficients appear; thus the routines may be easily modified to include yet higher-order post-Newtonian terms as they become available.

The post-Newtonian equations for the orbital phase evolution are notoriously ill-behaved [10, 11] as the binary system nears coalescence. In this regime the expansion parameters [namely the relative velocity v/c of the bodies and/or the field strength $GM_{\text{tot}}/(c^2 r_{\text{orbit}})$] used in the derivation are comparable to unity. In post²-Newtonian calculations higher orders such as post³-Newtonian terms have been discarded. Because of this truncation, quantities that are positive definite in an exact calculation (say the energy-loss rate, or the time derivative of the orbital frequency) often become negative in their post-Newtonian expansion when the orbital separation becomes small. When this happens you are using a post-Newtonian expression in a regime where its validity is questionable. This is cause for concern, and it may be cause for terminating a chirp calculation; but, it need not crash your code. A full-scale gravitational-wave search will need to compute chirps over a broad range of parameters, virtually assuring that any post-Newtonian chirp generator will be pushed into a region of parameter space where it doesn’t belong. These routines are designed to traverse these dangerous regions of parameter space as well as possible and gently warn the user of the dangers encountered. The calling routines may wish to act on the warnings coming from the chirp generator. For example a severe warning may prompt the calling routine to discard a given filter from a data search, because the second post-Newtonian calculation of the chirp is so dubious that it can’t give meaningful results.

In the next several sections we detail the use of three routines used to compute the “chirp” of a coalescing binary system. The first routine we describe is `phase_frequency()`. This is the underlying routine for the other chirp routines. Given a set of parameters (*e.g.* the two masses, and the upper and lower cut-off frequency for the chirp) it returns the orbital phase and orbital frequency evolution as a function of time. Next we describe `chirp_filter()` which returns two (unnormalized) chirp signals. This routine can be used for wholesale production of a bank of templates for a coalescing binary search.

6.2 Function: phase_frequency()

```
int phase_frequency(float m1, float m2, float spin1, float spin2, int n_phaseterms,
float *phaseterms, float Initial_Freq, float Max_Freq_Rqst, float *Max_Freq_Actual,
float Sample_Time, float **phase, float **frequency, int *steps_alloc, int
*steps_filld, int err_cd_sprs)
```

This function computes the **orbital phase** and **orbital frequency** evolution of an inspiraling binary. It returns an integer termination code indicating how and why the chirp calculation terminated. This routine is the engine that powers the other chirp generation routines. The arguments are:

`m1`: Input. The mass of body-1 in solar masses.

`m2`: Input. The mass of body-2 in solar masses.

`spin1`: Input. The dimensionless spin parameter of body-1. See section on spin effects.

`spin2`: Input. The dimensionless spin parameter of body-2. See section on spin effects.

`n_phaseterms`: Input. Integer describing the number of post-Newtonian (pN) approximation terms implemented in the phase and frequency calculations. In the present implementation this should be set to 5.

`phaseterms`: Input. The array `phase_terms[0..n_phaseterms-1]` specifies which pN approximation terms will be included in the phase frequency calculations. Setting `phase_terms[i]=0.0` nullifies the term. Setting `phase_terms[i]=1.0` includes the term. This allows for easy run-time nullification of any term in the phase and frequency evolution, *e.g.* setting `phase_terms[4]=0.0` eliminates the *second* post-Newtonian terms from the calculation.

`Initial_Freq`: Input. The starting orbital frequency of the chirp in Hz.

`Max_Freq_Rqst`: Input. The requested orbital frequency where the chirp will stop. However, the actual calculation may not proceed all the way to this orbital frequency. This is discussed at length below.

`Max_Freq_Actual`: Output. The floating number `*Max_Freq_Actual` is the orbital frequency in Hz where the chirp actually terminated.

`Sample_Time`: Input. The time interval between successive samples, in seconds.

`phase`: Input/Output. The phase ephemeris Ω in radians is stored in the array `*phase[0..steps_filld-1]`. Input in the sense that much of the internal logic of `phase_frequency()` depends on how the pointers `*phase` (and `*frequency` below) are set. If either is set to NULL memory allocation will be performed inside `phase_frequency()`. If both are not NULL then it is assumed the calling routine has allocated the memory before calling `phase_frequency()`.

`frequency`: Input/Output. Similar to `phase` above. The frequency ephemeris $f = d\Omega/dt$ is stored in the array `*frequency[0..steps_filld-1]`.

`steps_alloc`: Input/Output. The integer `*steps_alloc` is the number of floating point entries allocated for storing the phase and frequency evolution, *i.e.* the length of `**phase` and `**frequency`. This integer should be set in the calling routine if memory is allocated there, or it will be set inside `phase_frequency()` if memory is to be allocated there. If both of the pointers `*phase` and `*frequency` are not NULL then `phase_frequency()` understands that the calling routine is taking responsibility for allocating the memory for the chirp, and the calling routine must set

`*steps_alloc` accordingly. In this case `phase_frequency()` will fill up the arrays `**phase` and `**frequency` until the memory is full (*i.e* fill them with `*steps_alloc` of floats) or until the chirp terminates, whichever is less.

`steps_filld`: Output. The integer `*steps_filld` is the integer number of time steps actually computed for this evolution. It is less than or equal to `*steps_alloc`.

`clscnc_time`: Output. The float `*clscnc_time` is the time to coalescence in seconds, measured from the instant when the orbital frequency is `Initial_Freq` given by t_c in Eqs.(6.4.1) and (6.4.2).

`err_cd_sprs`: Input. Error code suppression. This integer determines at what level of disaster encountered in the computation of the chirp the user will be explicitly warned about with a printed message. Set to 0: prints all the termination messages. Set to 4000: suppresses all but a few messages which are harbingers of complete disaster. The termination messages are numbered from 0 to 3999 loosely in accordance with their severity (the larger numbers corresponding to more severe warnings). Any message with a number less than `err_cd_sprs` will not be printed. A termination code of 0 means the chirp calculation was executed as requested. A termination code in the 1000's means the chirp was terminated early because the post-Newtonian approximation was deemed no longer valid. A termination code in the 2000's generally indicates some problem with memory allocation. A termination code in the 3000's generally indicates a serious logic fault. Many of these "3000" errors result in the termination of the routine. If you get an error message number it is easy to find the portion of source code where the fault occurred; just do a character string search on the four digit number.

This phase and frequency generator has a number of very specialized features which will be discussed later. However, before we proceed further, we show a simple example of how `phase_frequency()` can be used.

Authors: Alan Wiseman, agw@tapir.caltech.edu and Bruce Allen, ballen@dirac.phys.uwm.edu

Comments: This function will need to be extended when results of order 2.5 and 3 post-Newtonian calculations have been reported and published.

6.3 Example: phase_evoltn program

This example uses `phase_frequency()` to compute the phase and frequency evolution for an inspiraling binary and prints the results on the screen (`stdout`). The other output messages go to `stderr`.

```
/* GRASP: Copyright 1997,1998 Bruce Allen */
#include "grasp.h"
int main() {
    float m1,m2,spin1,spin2,phaseterms[MAX_PHS_TERMS],clscnc_time,*ptrphase,*ptrfrequency;
    float time,Initial_Freq,Max_Freq_Rqst,Max_Freq_Actual,Sample_Time,time_in_band;
    int steps_alloc,steps_filld,i,n_phaseterms,err_cd_sprs,chirp_ok;

    /* Set masses and spins of the orbital system: */
    m1=m2=1.4;
    spin1=spin2=0.;

    /* Set ORBITAL frequency range of the chirp and sample time: */
    Initial_Freq=60.; /* in cycles/second */
    Max_Freq_Rqst=2000.; /* in cycles/second */
    Sample_Time=1./9868.4208984375; /* in seconds */

    /* Use this block to compare with Will & Wiseman, PRD 54, 4813 (1996) Figure 10, page 4846.
    spin1=0.1; spin2=0.5;
    m1=1.4; m2=12.0;
    Initial_Freq=75.0; Max_Freq_Rqst=180.0; */

    /* post-Newtonian [O(1/c^n)] terms you wish to include (or supress)
    in the phase and frequency evolution: */
    n_phaseterms=5; /* the number of entries in phaseterms */
    phaseterms[0] =1.; /* The Newtonian piece */
    phaseterms[1] =0.; /* There is no O(1/c) correction */
    phaseterms[2] =1.; /* The post-Newtonian correction */
    phaseterms[3] =1.; /* The 3/2 PN correction */
    phaseterms[4] =1.; /* The 2 PN correction */

    /* Set memory-allocation and error-code supression logic: */
    ptrphase=ptrfrequency=NULL;
    err_cd_sprs=0;

    /* Use phase_frequency() to compute phase and frequency evolution: */
    chirp_ok=phase_frequency(m1,m2,spin1,spin2,n_phaseterms,phaseterms,
        Initial_Freq,Max_Freq_Rqst,&Max_Freq_Actual,Sample_Time,&ptrphase,
        &ptrfrequency,&steps_alloc,&steps_filld,&clscnc_time,err_cd_sprs);

    /* ... and print out the results: */
    time_in_band=(float)(steps_filld-1)*Sample_Time;
    fprintf(stderr, "\nm1=%f m2=%f Initial_Freq=%f\n", m1,m2,Initial_Freq);
    fprintf(stderr, "steps_filld=%i steps_alloc=%i Max_Freq_Actual=%f\n",
        steps_filld,steps_alloc,Max_Freq_Actual);
    fprintf(stderr, "time_in_band=%f clscnc_time=%f\n", time_in_band,clscnc_time);
    fprintf(stderr, "Termination code: %i\n\n",chirp_ok);

    for (i=0;i<steps_filld;i++){
        time=i*Sample_Time;
        printf("%i\t%f\t%f\t%f\n",i,time,ptrphase[i],ptrfrequency[i]);
    }
    return 0;
}
```

Here is the output from the phase_evoltn example:

```
GRASP: Message from function phase_frequency() at line number 439 of file "pN_chirp.c".
Frequency evolution no longer monotonic.
Phase evolution terminated at frequency and step: 911.681702    13357
Terminating chirp. Termination code set to:    1201
Returning to calling routine.
$Id: man_inspiral.tex,v 1.42 1999/09/29 19:44:41 ballen Exp $
$Name: RELEASE_1_9_8 $

m1=1.400000 m2=1.400000 Initial_Freq=60.000000
steps_filld=13357 steps_alloc=16384 Max_Freq_Actual=911.681702
time_in_band=1.353408 clscnc_time=1.353573
Termination code: 1201

0      0.000000      0.000000      60.000000
1      0.000101      0.038204      60.001675
2      0.000203      0.076369      60.003353
3      0.000304      0.114627      60.005020
4      0.000405      0.152820      60.006695
5      0.000507      0.191071      60.008366
6      0.000608      0.229173      60.010052

...      ...      ...      ...

13349  1.352699      797.669800      720.294189
13350  1.352800      798.134949      741.157715
13351  1.352901      798.614136      764.565796
13352  1.353003      799.109192      791.015686
13353  1.353104      799.622192      821.015320
13354  1.353205      800.155457      854.720337
13355  1.353307      800.710999      890.133667
13356  1.353408      801.286499      911.681702
```

The first seven lines of output come directly from `phase_frequency()`, and are printed to `stderr`. These give a warning message telling why the chirp calculation was terminated; it no longer had monotonically increasing frequency. It also tells where the chirp was terminated; after computing 13357 points it has reached a frequency of 907Hz. The termination code (1201) is also printed. Knowing the termination code makes it easy to find the segment of source code that produced the termination; just do a search for the character string "1201" and you will find the line of code where the termination code was set. Setting `err_cd_sprs` greater than 1201 would suppress the printing of this warning message and all messages with a termination code less than 1201. However, even without the printed message the calling routine can determine the value of the termination code; it is returned by `phase_frequency()`.

The rest of the output comes from the `phase_evoltn` program. The quantity `time_in_band = (steps_filld - 1) * Sample_Time` is the length (in seconds) of the computed chirp. The quantity `clscnc_time` is the value of t_c that enters Eqs.(6.4.1) below. The four column output from left to right is the integer index of the data points, time stamp of each point in seconds (starting arbitrarily from zero), the orbital phase in radians (starting arbitrarily from zero), and the orbital frequency (starting from the initial frequency of 60Hz).

To summarize: It takes about 1.35 seconds for two $1.4M_{\odot}$ objects to spiral in from an orbital frequency of 60Hz to an orbital frequency of 911Hz. The chirp calculation was terminated at 911Hz – instead of the requested 2000Hz – because the post-Newtonian expression used to compute the chirp is clearly out of its region of validity: the frequency is no longer increasing. Examining the last few data

points shows that the frequency was rising quickly – as expected – until the last two data points. During this inspiral the orbital system went through $811.09/(2\pi) \approx 127.53$ revolutions. The two integer numbers `steps_filld` and `steps_alloc` are the number of actual data points computed and the number of floating point memory slots allocated, respectively. (Memory is allocated in blocks of 4096 floats at a time. Thus `steps_alloc` will generally exceed `steps_filld`.) The values of the phase and frequency at every $1/\text{Sample_Time} = 1.10333 \times 10^{-4}$ seconds starting from when the binary had an orbital frequency of 60Hz until it neared “coalescence” at 911Hz have been calculated.

6.4 Detailed explanation of `phase_frequency()` routine

The `phase_frequency()` routine starts with inputs describing the physical properties of the system (the masses) and an initial frequency from which to start the evolution. We then compute the orbital frequency evolution [in cycles/second] directly from the formula given in [7]

$$f(t) = \frac{M_{\odot}}{16\pi T_{\odot} m_{\text{tot}}} \left\{ \Theta^{-3/8} + \left(\frac{743}{2688} + \frac{11}{32}\eta \right) \Theta^{-5/8} - \frac{3\pi}{10} \Theta^{-3/4} + \left(\frac{1855099}{14450688} + \frac{56975}{258048}\eta + \frac{371}{2048}\eta^2 \right) \Theta^{-7/8} \right\}, \quad (6.4.1)$$

where m_{tot} is the total mass of the binary. The time integral of this equation gives the orbital evolution in cycles. Multiplying by 2π yields the orbital phase in radians

$$\phi(t) = \phi_c - \frac{1}{\eta} \left\{ \Theta^{5/8} + \left(\frac{3715}{8064} + \frac{55}{96}\eta \right) \Theta^{3/8} - \frac{3\pi}{4} \Theta^{1/4} + \left(\frac{9275495}{14450688} + \frac{284875}{258048}\eta + \frac{1855}{2048}\eta^2 \right) \Theta^{1/8} \right\}. \quad (6.4.2)$$

Here Θ is a dimensionless time variable

$$\Theta = \frac{\eta M_{\odot}}{5T_{\odot} m_{\text{tot}}} (t_c - t), \quad (6.4.3)$$

$\eta = \mu/m_{\text{tot}}$, and t_c is the time of coalescence of the two point masses. Similarly the constant ϕ_c is the phase at coalescence, which is arbitrarily set in `phase_frequency()` so that $\phi = 0$ at the initial time. [See the detailed discussion of the phase conventions below.] Also notice that the mass quantities only appear as ratios with the solar Mass M_{\odot} , and the time only appears as a ratio with the quantity $T_{\odot} = 4.925491 \times 10^{-6}$ sec in Eq.(6.0.2).

These formulations of the post-Newtonian equations for the phase and frequency are simple to implement: each pass through the loop increments the time by the sample time (`Sample_Time` in the example) and computes the phase and frequency using Eqs. (6.4.1) and (6.4.2). However, there is an alternative formulation. In deriving these equations the “natural” equation that arises is of the form $\dot{f} = F(f)$. [See *e.g.* [12] Eq.(3).] This in turn can be integrated to give an equation of the form $t_c - t = T(f)$. In our formulation this equation has been inverted – throwing away higher-order post-Newtonian terms as you go – to give Eq.(6.4.1). However the equation in the form $t_c - t = T(f)$ can also be implemented directly. In this type of formulation one would again increment the time, but then use a root-finding routine to find the frequency at each time step. Our chosen method has the advantage of avoiding a time-consuming root-finder at each time step; however the alternative formulation has undergone fewer damaging post-Newtonian transformations, and may therefore be more accurate.

In our formulation we only need to call a root-finding routine at the start of the chirp to find the value of $t_c - t$ when the system is at the initial frequency. In order to insure that we find the correct root for the starting time we begin a search at a time when the leading order prediction of the frequency is well below the desired starting frequency. We step forward in time until we bracket the root; we then call the *Numerical Recipes* root-finder `rtbis()` to compute the root precisely. This is depicted in the lower right corner of figure 24 where we show the value of the “time” coordinate X that corresponds to an initial frequency of 60Hz. This method is virtually assured of finding the *correct* root in that it will find the first solution as we proceed from right to left in figure 24. The primary problem in finding this root is that there may actually be no meaningful start-time for the specified chirp. For example, if you you were to specify a chirp with two $1.4M_{\odot}$ objects with an initial frequency of 1000Hz, you can see from the figure that there is no value of X (*i.e.* $t_c - t$) that corresponds to this frequency. In this case `phase_frequency()` will search from right

to left for the start time. It will notice that it is passing over the peak in the graph and out of the regime of post-Newtonian viability. It will then terminate the search and notify the caller that there is no solution for the requested chirp.

The behavior of the frequency equation is shown in figure 24. As time increases the frequency rises to a maximum and then begins to decrease dramatically. Notice that the maximum occurs when the dimensionless time parameter $\Theta = \frac{\eta(t_c-t)}{5T_\odot m_{tot}} = X^8$ is approximately unity; this feature is only weakly dependent on the mass ratio. The fact that $\Theta \approx 1$ means the post-Newtonian corrections in Eq.(6.4.1) are comparable to the leading order term. Therefore, this peak is a natural place to terminate the post-Newtonian chirp approximation. In the example the code terminated the chirp for precisely this reason. [See the warning message.]

Although it is not shown in the figure the behavior of f as X nears zero is very abrupt; the function goes sharply negative and then turns around and diverges to $+\infty$ as $X \rightarrow 0$ (*i.e.* $t \rightarrow t_c$). This abrupt behavior will happen on a time scale of order T_\odot (a few microseconds). Typical sample times are likely to be on the order of a tenth of a millisecond, and therefore the iterative loop may step right over this maximum-minimum-divergence behavior of the frequency function altogether. Don't worry. The routine `phase_frequency()` handles this case gracefully. The routine will stop the chirp calculation and warn the caller if the time stepper goes beyond the coalescence time. It will also stop the chirp calculation if it senses that the time has stepped over the dip in frequency and is on the strongly divergent part of the frequency curve near the $X = 0$ axis.

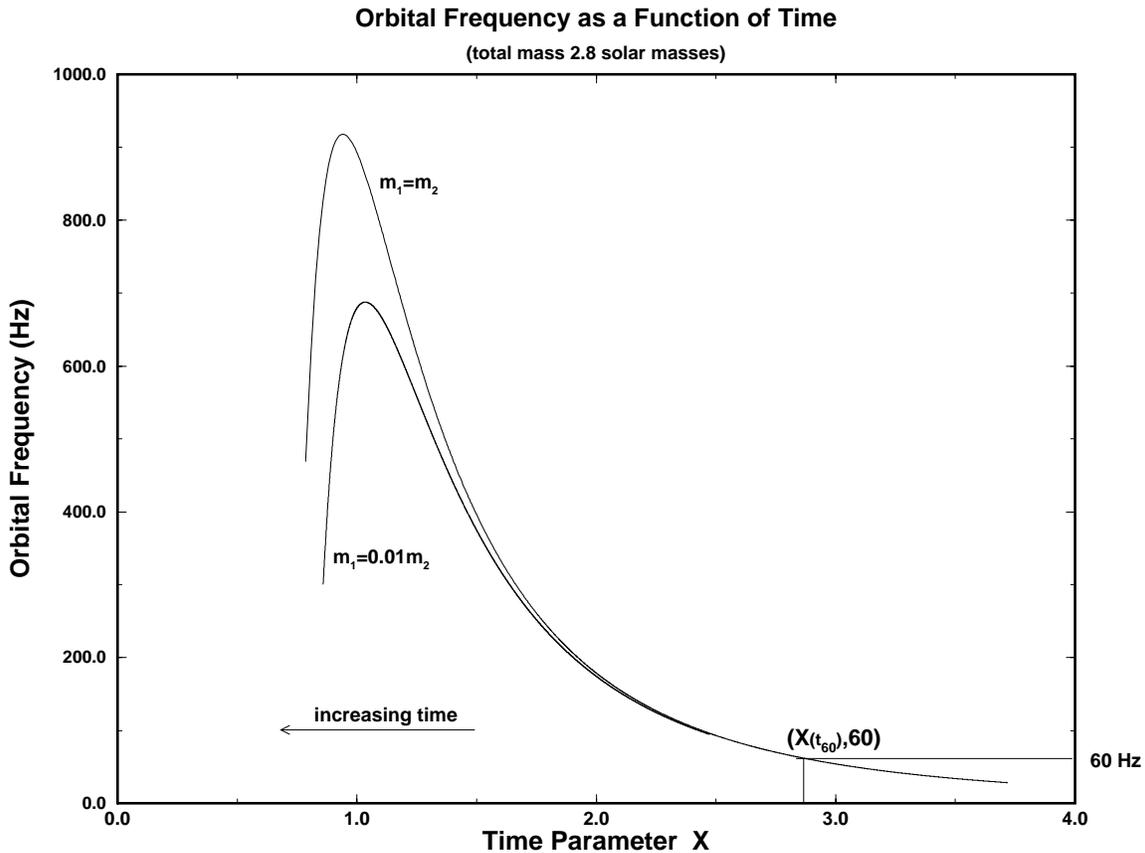


Figure 24: Orbital frequency as a function of the “time” coordinate $X = \left(\frac{\eta(t_c-t)M_\odot}{5T_\odot m_{tot}} \right)^{1/8}$.

6.5 Function: `chirp_filters()`

```
int chirp_filters(float m1, float m2, float spin1, float spin2, int n_phaseterms,
float *phaseterms, float Initial_Freq, float Max_Freq_Rqst, float *Max_Freq_Actual,
float Sample_Time, float **ptrptrCos, float **ptrptrSin, int *steps_alloc,
int *steps_filld, int err_cd_sprs)
```

This function is a basic stripped-down chirp generator. It computes two – nearly orthogonal – chirp waveforms for an inspiraling binary. The two chirps differ in phase by $\pi/2$ radians. The chirp values are given by Eqs.(6.6.1) and (6.6.2). Just as the phase and frequency calculator `phase_frequency()` returns an integer number which describes how the chirp calculation was terminated, this routine does also.

The arguments are:

`m1`: Input. The mass of body-1 in solar masses.

`m2`: Input. The mass of body-2 in solar masses.

`spin1`: Input. The dimensionless spin parameter of body-1. See section on spin effects.

`spin2`: Input. The dimensionless spin parameter of body-2. See section on spin effects.

`n_phaseterms`: Input. Integer describing the number of terms implemented in the phase and frequency calculations. In the present implementation this should be set to 5.

`phaseterms`: Input. The array `phase_terms[0..n_phaseterms-1]` describes which terms will be included in the phase frequency calculations. Setting `phase_terms[i]=0` nullifys the term. Setting `phase_terms[i]=1` includes the term. This allows for easy run-time nullification of any term in the phase and frequency evolution, *e.g.* setting `phase_terms[4]=0` eliminates the second post-Newtonian terms from the calculation.

`Initial_Freq`: Input. The starting orbital frequency of the chirp in Hz.

`Max_Freq_Rqst`: Input. The requested orbital frequency where the chirp will stop. However, the actual calculation may not proceed all the way to this orbital frequency.

`Max_Freq_Actual`: Output. The floating number `*Max_Freq_Actual` is the orbital frequency in Hz where the chirp actually terminated.

`Sample_Time`: Input. The time interval between points in seconds.

`ptrptrCos`: Input/Output. The chirp corresponding to Eq.(6.6.1) is stored in `*ptrptrCos[0..steps_filld-1]`. Input in the sense that much of the internal logic of `chirp_filters()` depends on how the pointers `*ptrptrCos` (and `*ptrptrSin` below) are set. If either is set to NULL memory allocation will be performed inside `chirp_filters()`. If both are not NULL then it is assumed the calling routine has allocated the memory before calling `chirp_filters()`.

`ptrptrSin`: Input/Output. Similar to `ptrptrCos` above. The chirp corresponding to Eq.(6.6.2) is stored in `*ptrptrSin[0..steps_filld-1]`.

`steps_alloc`: Input/Output. The integer `*steps_alloc` is the number of floating point entries allocated for storing the two chirps, *i.e.* the number of valid subscripts in the arrays `**ptrptrCos` and `**ptrptrSin`. This integer should be set in the calling routine if memory is allocated there, or it will be set inside `chirp_filters()` if memory is to be allocated there. If both of the pointers `*ptrptrCos` and `*ptrptrSin` are not NULL then `chirp_filters()` understands that the

calling routine is taking responsibility for allocating the memory for the chirp, and the calling routine must set `*steps_alloc` accordingly. In this case `chirp_filters()` will fill up the arrays `**ptrptrCos` and `**ptrptrSin` until the memory is full (*i.e* fill them with `*steps_alloc` of floats) or until the chirp terminates, whichever is less.

`steps_filld`: Output. The integer `*steps_filld` is the number of time steps (sample values) actually computed for this evolution. It is less than or equal to `*steps_alloc`.

`clscnc_time`: Output. The float `*clscnc_time` is the time to coalescence in seconds, measured from the instant when the orbital frequency is `Initial_Freq` given by t_c in Eqs.(6.4.1) and (6.4.2).

`err_cd_sprs`: Input. Error code suppression. This integer specifies the level of disaster encountered in the computation of the chirp for which the user will be explicitly warned with a printed message. Set to 0: prints all the termination messages. Set to 4000: suppresses all but a few messages which are harbingers of true disaster. The termination messages are numbered from 0 to 3999 loosely in accordance with their severity (the larger numbers corresponding to more severe warnings). Any message with a number less than `err_cd_sprs` will not be printed. A termination code of 0 means the chirp calculation was executed as requested. A termination code in the 1000's means the chirp was terminated early because the post-Newtonian approximation was deemed no longer valid. A termination code in the 2000's generally indicates some problem with memory allocation. A termination code in the 3000's generally indicates a serious logic fault. Many of these "3000" errors result in the termination of the program. If you get an error message number it is easy to find the portion of source code where the fault occurred; just do a character string search on the four digit number.

Authors: Alan Wiseman, agw@tapir.caltech.edu and Bruce Allen, ballen@dirac.phys.uwm.edu

Comments: None.

6.6 Detailed explanation of `chirp_filters()` routine

The routine `chirp_filters()` calls `phase_frequency()` to find out the how the orbital phase and frequency evolve in accordance with the input parameters. It then makes a single pass through that phase and frequency ephemeris, computing the chirps as it goes, and storing the information in the space already allocated for the phase and frequency. Most of the fault checking and computations are done in the `phase_frequency()` routine, and all the errors messages and warnings come from there.

The routine `chirp_filters()` computes

$$h_c(t) = 2 \left(\frac{\mu}{M_\odot} \right) \left[\frac{2\pi T_\odot m_{\text{tot}} f(t)}{M_\odot} \right]^{2/3} \cos 2\phi(t) \quad (6.6.1)$$

and the other orbital-phase chirp which is $\pi/2$ out of phase with $h_c(t)$

$$h_s(t) = 2 \left(\frac{\mu}{M_\odot} \right) \left[\frac{2\pi T_\odot m_{\text{tot}} f(t)}{M_\odot} \right]^{2/3} \sin 2\phi(t) , \quad (6.6.2)$$

with all the leading numerical factors we display.

If the so called “restricted” post²-Newtonian polarizations [leading order in the amplitude, but post²-Newtonian phase corrections] are desired, they can be easily assembled from h_c and h_s . The “+” (plus) polarization is given by

$$h_+(t) = -\frac{T_\odot c}{D} (1 + \cos^2 i) h_c(t) , \quad (6.6.3)$$

and the “×” (cross) polarization is given by

$$h_\times(t) = -2 \frac{T_\odot c}{D} (\cos i) h_s(t) . \quad (6.6.4)$$

Here D is the (luminosity) distance to the source in centimeters, c is the speed of light in centimeters/second, and i is the inclination angle (radians) of the of the angular momentum axis of the source relative to the line-of-sight. See Will and Wiseman [8] figure 7 for the precise definition of the inclination angle.

The restricted post²-Newtonian strain amplitude impinging on the detector can also be calculated from the output of `chirp_filters()` by

$$h(t) = F_+ h_+(t) + F_\times h_\times(t) , \quad (6.6.5)$$

where F_+ and F_\times are the detector beam-pattern functions.

In the remainder of this section we will clarify some technical issues involving the orbital phase. First, in computing $\phi(t)$ in `phase_frequency()` we have arbitrarily set the constant ϕ_c in Eq.(6.4.2) such that $\phi = 0$ at the beginning of the chirp. The astrophysical convention for defining the orbital phase angle ϕ given in [8] measures ϕ in the plane of the orbit from the ascending node. [The ascending node of the orbit is where body-1 passes through the plane of the sky going away from the observer.] Choosing ϕ_c in this way we have assumed that body-1 is passing through the ascending node of the orbit at the instant we start our chirp. Detailed information about the overall phase is not needed for many purposes (*i.e.* matched filters), therefore our choice is of little consequence. If this information needs to be included for some application, `chirp_filters()` can be modified to do so; thus one can leave the computational engine `phase_frequency()` untouched.

The second issue involving the phase is a bit more delicate. We have used the true orbital phase $\phi(t)$ to compute oscillatory part of the chirp in Eqs.(6.6.1) and (6.6.2). But should we use the logarithmically modulated phase variable

$$\psi(t) = \phi - \frac{4Gm_{\text{tot}} \pi f(t)}{c^3} \ln[f(t)/f_o] \quad (6.6.6)$$

in our computation of the chirp? After all, the true phase of the gravitational-wave signal impinging on the detector is 2ψ . Let us examine the effect on our signal replacing $\sin 2\phi$ in Eq.(6.6.2) with the logarithmically corrected $\sin 2\psi$

$$\begin{aligned} \sin 2\psi &= \sin\left(2\phi - \frac{8\pi m_{\text{tot}} f G}{c^3} \ln(f(t)/f_o)\right) \\ &= \sin 2\phi \cos\left(\frac{8\pi m_{\text{tot}} f G}{c^3} \ln(f(t)/f_o)\right) - \cos 2\phi \sin\left(\frac{8\pi m_{\text{tot}} f G}{c^3} \ln(f(t)/f_o)\right) \\ &\approx \left(1 + O(1/c^6)\right) \sin 2\phi - \left(\frac{8\pi m_{\text{tot}} f G}{c^3} \ln(f(t)/f_o)\right) \cos 2\phi . \end{aligned} \quad (6.6.7)$$

The $O[1/c^6]$ is a post³-Newtonian term and can be neglected in the present post²-Newtonian analysis. However the coefficient of the $\cos 2\phi$ is a post^{3/2}-Newtonian order correction to the waveform, and must be included in any full post²-Newtonian analysis. This logarithmic term is included in the waveform calculation in the `strain()` routine. However, the last line of Eq.(6.6.7) also shows that the logarithmic phase correction can be considered a post^{3/2}-Newtonian correction to the amplitude. In our present restricted post-Newtonian chirp calculation we neglect these higher order amplitude corrections, so we are justified in neglecting the logarithmic correction to the phase.

The advantage of neglecting the logarithm is that it speeds up the calculation of the chirps: we don't have to compute a logarithm at each time step. However, this may be at expense of accurately tracking the signal phase of a strongly relativistic source. After all much research has gone into computing the gravitational wave phase from these sources and we shouldn't willy-nilly discard these phase corrections. Is it difficult to modify our code to include this term? Not at all. In fact, the inclusion of the logarithmic correction to the gravitational wave phase would not affect `phase_frequency()`, at all. The fact that this logarithmic propagation effect only enters the `chirp_filters()` routine and not the `phase_frequency()` routine may seem like a computational quirk, but this actually has a physical origin: The routine `phase_frequency()` computes the local orbital phase of the binary; whereas, the physical origin of the logarithmic term is a *propagation* effect and has nothing to do with the orbital phase,

This is not say that no log terms will ever be needed in `phase_frequency()`. Note that at post⁴-Newtonian order there are log terms which do affect the local instantaneous orbital motion of the binary, so if `phase_frequency()` is ever modified to incorporate that order, then log terms will appear there also.

Another issue involving the log term in the phase is the presence of the "arbitrary" scale factor f_o entering the definition of $\psi(t)$ in Eq.(6.6.6). The net effect of adjusting this constant is to change the value of another arbitrary constant in our phase and frequency equations; it shifts the value of t_c in Eq.(6.4.3). In order to facilitate swift computation, we choose f_o to be the minimum frequency of the requested chirp. This insures that the ratio in the logarithm is of order unity during the chirp computation.

6.7 Example: filters program

This example uses `chirp_filters()` to generate two chirps $\pi/2$ out of phase with each other. It also demonstrates a different memory allocation option than the `phase_evoln` example program.

```
/* GRASP: Copyright 1997,1998 Bruce Allen */
#include "grasp.h"
int main() {
    float m1,m2,spin1,spin2,phaseterms[MAX_PHS_TERMS],clscnc_time,*ptrCos,*ptrSin;
    float time,Initial_Freq,Max_Freq_Rqst,Max_Freq_Actual,Sample_Time,time_in_band;
    int steps_alloc,steps_filld,i,n_phaseterms,err_cd_sprs,chirp_ok;

    /* Set physical parameters of the orbital system: */
    m1=m2=1.4;
    spin1=spin2=0.;

    /* Set ORBITAL frequency range of the chirp and sample time: */
    Initial_Freq=60.; /* in cycles/second */
    Max_Freq_Rqst=2000.; /* in cycles/second */
    Sample_Time=1./9868.4208984375; /* in seconds */

    /* post-Newtonian [O(1/c^n)] terms you wish to include (or supress)
       in the phase and frequency evolution: */
    n_phaseterms=5;
    phaseterms[0] =1.; /* The Newtonian piece */
    phaseterms[1] =0.; /* There is no O(1/c) correction */
    phaseterms[2] =1.; /* The post-Newtonian correction */
    phaseterms[3] =1.; /* The 3/2 PN correction */
    phaseterms[4] =1.; /* The 2 PN correction */

    /* Set memory-allocation and error-code supression logic: */
    steps_alloc=10000;
    ptrCos=(float *)malloc(sizeof(float)*steps_alloc);
    ptrSin=(float *)malloc(sizeof(float)*steps_alloc);
    err_cd_sprs=0; /* 0 means print all warnings */

    /* Use chirp_filters() to compute the two filters: */
    chirp_ok=chirp_filters(m1,m2,spin1,spin2,n_phaseterms,phaseterms,
        Initial_Freq,Max_Freq_Rqst,&Max_Freq_Actual,Sample_Time,
        &ptrCos,&ptrSin,&steps_alloc,&steps_filld,&clscnc_time,err_cd_sprs);

    /* ... and print out the results: */
    time_in_band=(float)(steps_filld-1)*Sample_Time;
    fprintf(stderr,"\nm1=%f m2=%f Initial_Freq=%f\n",m1,m2,Initial_Freq);
    fprintf(stderr,"steps_filld=%i steps_alloc=%i Max_Freq_Actual=%f\n",
        steps_filld,steps_alloc,Max_Freq_Actual);
    fprintf(stderr,"time_in_band=%f clscnc_time=%f\n",time_in_band,clscnc_time);
    fprintf(stderr,"Termination code: %i\n\n",chirp_ok);
    for (i=0;i<steps_filld;i++){
        time=i*Sample_Time;
        printf("%i\t%f\t%f\t%f\n",i,time,ptrCos[i],ptrSin[i]);
    }
    return 0;
}
```

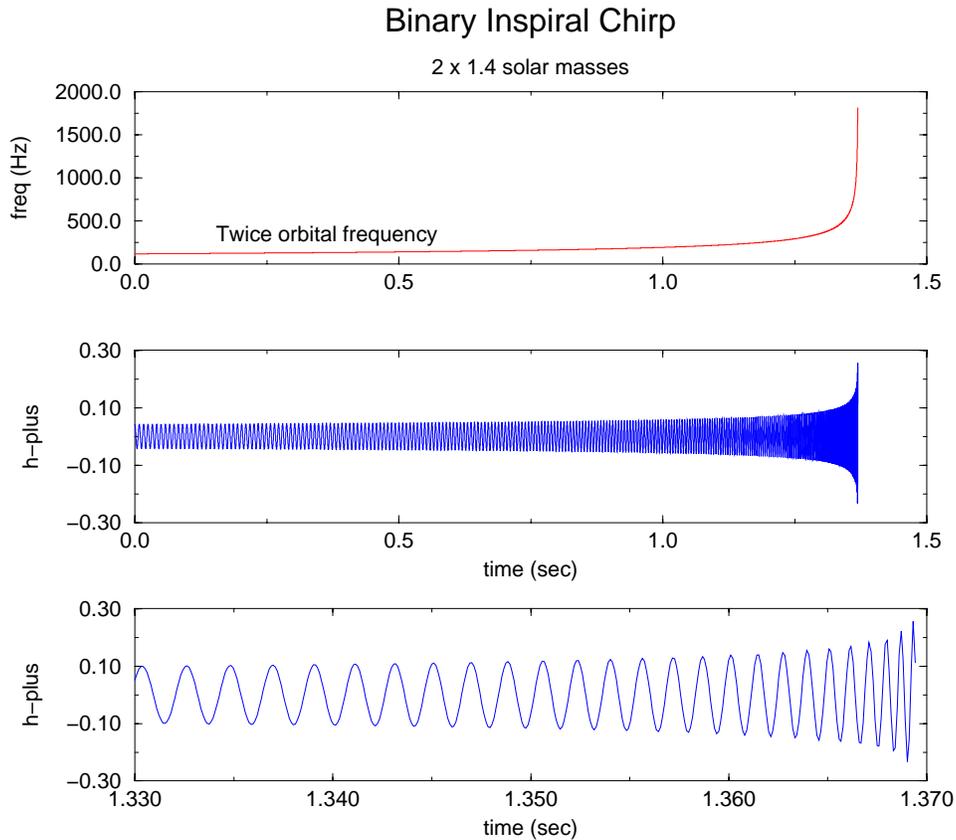


Figure 25: The zero-phase chirp waveform from a $2 \times 1.4M_{\odot}$ binary system, starting at an orbital frequency of 60 Hz. The top graph shows the frequency of the dominant quadrupole radiation as a function of time, and the middle graph shows the waveform. The bottom graph shows a 40-msec stretch near the final inspiral/plunge.

Notice that we only allocated enough memory for 10000 points, and we know from the output from the previous example that this chirp takes 13515 points. Therefore running this example results in following error message printed to `stderr`:

```
GRASP:phase_frequency():Allocated memory is filled up before
reaching the maximum frequency requested for this chirp.
Orbital Frequency Reached(Hz): 98.867607, Number of points: 10000
Terminating chirp. Termination code set to:      2001
Returning to calling routine.
```

However, even though the routine ran out of memory it still computed the first 10000 points of the chirp and returned them in the arrays `*ptrptrCos[0..steps_alloc-1]` and `*ptrptrSin[0..steps_alloc-1]`.

6.8 Practical Suggestion for Setting Up a Large Bank of Filters:

We have carefully explained (how to avoid) a number of the pitfalls in computing post-Newtonian chirps. Before using the chirp generators to spit out hundreds or thousands of chirps needed for a bank of filters and farming out the computations out to dozens of parallel processors in a massive coalescing binary search, we strongly suggest that you edit the examples already given and check the routine against the **three** extreme cases you will encounter in your search.

1. Try the example with both masses set to the minimum mass in your proposed search, *i.e.* compute the phase and frequency evolution and the chirps for the template in the upper right hand corner in figure 59. This is the template of longest duration. If you are going to have a memory allocation problem you will have it with this template. Also, knowing the duration of the longest template in your search will help you decide the length of the segments of data which you filter. In general, you want the length of these data segments to be at least several times longer than the longest chirp. See Section 6.19 for further details.
2. Try the chirp generator with both masses set to the maximum mass in your search, *i.e.* compute the phase and frequency evolution of the template in the lower left corner of figure 59. This is the shortest duration template and the one least likely to make it to the upper cut off frequency before going out of the region of post-Newtonian viability. This case will be the most demanding test of the “chirp-termination” logic in `phase_frequency()`. It is also possible in the case of extremely large masses that there really is no chirp at all in the frequency regime requested. For example a binary composed of two $100M_{\odot}$ object will coalesce long before it reaches the initial chirp frequency of the 60Hz we are using as our a lower cutoff frequency in our example. Don’t worry. The routine `phase_frequency()` will warn you that the root finder was unable to find a viable solution for the initial time. You may have to adjust the search range accordingly.
3. Try the chirp generator with one mass at the minimum allowed value and the other mass at the maximum allowed value, *i.e.* compute the phase and frequency evolution for the template in the upper left corner of figure 59. This is the template which is most dominated by post-Newtonian terms in the evolution.

If the routine gives satisfactory results for these three cases, it should work for all the cases shown in figure 59; you are now ready for wholesale production.

6.9 Additional contributions to the phase and frequency of the chirp

In recent years additional relativistic corrections to the binary inspiral chirp formula have been calculated. These have now been included in GRASP, and are available to users generating template banks. The changes have been implemented in existing GRASP routines, and using them requires minimal modification of your code. Furthermore, *if you have written GRASP code that uses only second post Newtonian chirps – and you want to keep it that way – you don't need to do anything: all the modifications are compatible with previous GRASP releases.*

When the original code for the GRASP chirp generator was written, only the second post-Newtonian relativistic corrections to the phase and frequency evolution were available. These are depicted in Eqs. (6.4.1) and (6.4.2), and implemented in the `phase_frequency()` routine. The routine used for generating template banks, `make_filters()`, uses these formulae by calling `phase_frequency()`. In the next two subsections we discuss extensions of these formulae and routines to include the contributions to the phase and frequency produced by the spins of the objects (spin-orbit coupling, and spin-spin coupling) and also the contribution from the 2.5 post-Newtonian order corrections. In future GRASP releases we hope to include contributions to the phase and frequency produced by the quadrupole moment of the bodies and higher-order post-Newtonian effects.

6.9.1 Spin Effects

In the simple case where the spin vectors of the bodies are aligned (or antialigned) with the orbital angular momentum axis, the GRASP chirp-generating functions have the built-in capability of computing the leading order **spin-orbit** and **spin-spin** corrections to the inspiral chirp. To use this feature no modification of the chirp-generating routines [`phase_frequency()` or `chirp_filters()`] is necessary; simply pass nonzero values of the spin parameters to the functions. This can easily be done by editing the example programs `phase_evoltm.c` and/or `filters.c` to pass nonzero values of the variables `spin1` and `spin2`. [See below for definitions and allowed ranges of `spin1` and `spin2`.]

When spinning bodies are involved, the full gravitational waveform can be quite complicated; the orbital plane and the spin vectors of the individual bodies can precess. The precession causes a modulation of the signal. However, this GRASP routines only implements the the special case when the spins are assumed to be aligned (or antialigned) with the orbital angular momentum axis. In this case there is no precession and, therefore, no modulation of the amplitude of the signal. Also in this case, the spin-corrections to the orbital frequency and phase are given by simple modifications to the nonspin phase and frequency Eqs. (6.4.1) and (6.4.2). The necessary terms can be found in Eq.(F22) in Appendix F of [8], and are given by

$$f(t) = \frac{M_{\odot}}{16\pi T_{\odot} m_{\text{tot}}} \left\{ \Theta^{-3/8} + \dots + \left(\frac{113}{160} [\chi_s + (\delta m/m) \chi_a] - \frac{19}{40} \eta \chi_s \right) \Theta^{-3/4} - \left(\frac{237}{512} \eta [(\chi_s)^2 - (\chi_a)^2] \right) \Theta^{-7/8} \right\}, \quad (6.9.1)$$

and

$$\phi(t) = \phi_c - \frac{1}{\eta} \left\{ \Theta^{5/8} + \dots + \left(\frac{113}{64} [\chi_s + (\delta m/m) \chi_a] - \frac{19}{16} \eta \chi_s \right) \Theta^{1/4} - \left(\frac{1185}{512} \eta [(\chi_s)^2 - (\chi_a)^2] \right) \Theta^{1/8} \right\}. \quad (6.9.2)$$

Here Θ is the dimensionless time variable given by Eq. (6.4.3). The ellipses represent the nonspin (post)ⁿ-Newtonian terms already given in Eqs. (6.4.1) and (6.4.2). The quantities χ_s and χ_a are dimensionless

quantities related to the angular momentum of the bodies by

$$\chi_s = \frac{1}{2} \left(\frac{S_1}{m_1^2} + \frac{S_2}{m_2^2} \right), \quad (6.9.3)$$

$$\chi_a = \frac{1}{2} \left(\frac{S_1}{m_1^2} - \frac{S_2}{m_2^2} \right), \quad (6.9.4)$$

where $S_{1(2)}$ is the signed magnitude of the angular momentum vector of each body expressed in geometrized units (cm^2), and m_i is the mass in geometrized units (cm). [Below we show how to convert from geometrized units to cgs units.] The sign is positive (negative) for spins aligned (antialigned) with the the angular momentum axis. By comparing the nonspin phase and frequency evolution in Eqs. (6.4.1) and (6.4.2) with the spin corrections in Eqs. (6.9.1) and (6.9.2), we see that the spin-orbit corrections (terms linear in χ_s and χ_a) simply modify the (post)^{3/2}-Newtonian contributions and the spin-spin corrections (term quadratic in χ_s and χ_a) modify the (post)²-Newtonian contributions.

Specifically, the spin quantities passed to the chirp generation routines are the signed, dimensionless (Kerr-like) parameters of each body

$$\text{spin1} = \pm \frac{|\mathbf{S}_1|}{m_1^2}, \quad (6.9.5)$$

$$\text{spin2} = \pm \frac{|\mathbf{S}_2|}{m_2^2}, \quad (6.9.6)$$

where the $+(-)$ sign is chosen if the spin is aligned (antialigned) with the orbital angular momentum axis. [Note: only in Eqs.(6.9.3)-(6.9.6) is mass expressed in geometrized units.]

Some calculations (*e.g.* those requiring a precise definition of the orbital phase) are sensitive to the index assigned to the bodies. The GRASP convention is that m_1 is the smaller of the two masses; therefore `spin1` should be the spin assigned to the smaller of the two masses.

How are the dimensionless spin parameters `spin1(2)` and the geometrized angular momentum \mathbf{S}_i related to angular momentum of the bodies in cgs units? Let L_i denote the spin angular momentum of the i -th body in cgs units (*i.e.* gram cm^2/sec). Then L_i is related to S_i by

$$\begin{aligned} S_i \text{ [in geometrized units, } i.e. \text{ cm}^2] &= \left(\frac{G}{c^3} \right) L_i \text{ (in gram cm}^2/\text{sec)} \\ &= 2.477 \times 10^{-39} \text{ (sec/gram)} L_i \text{ (in gram cm}^2/\text{sec)}. \end{aligned} \quad (6.9.7)$$

The conversion of angular momentum in cgs units to the **dimensionless** variable `spin1(2)` (the variable actually sent to the routine) is

$$\text{spini} = \left(\frac{c}{Gm_i^2} \right) L_i = \left(\frac{c}{Gm_\odot^2} \right) \left(\frac{M_\odot}{m_i} \right)^2 L_i = 1.136 \times 10^{-49} \text{ (sec/(gram cm}^2\text{))} \left(\frac{M_\odot}{m_i} \right)^2 L_i \quad (6.9.8)$$

where L_i is the magnitude of the spin angular momentum of the i -th body in standard cgs units (*i.e.* gram cm^2/sec), and m_i is the mass in grams.

What is the allowable range for the spin parameters `spin1` and `spin2`? For Kerr black holes, we know $|\text{spin1}(2)| = (|\mathbf{S}_{1(2)}|/m_{1(2)}^2) \leq 1$. For spinning neutron stars, stability studies (based on relativistic numerical hydrodynamic simulations) show that the spin parameter must satisfy $|\text{spin1}(2)| = (|\mathbf{S}_{1(2)}|/m_{1(2)}^2) \lesssim 0.6$. These limits can serve as a hard upper bound for a choice of spin parameters. However, observed pulsars in binaries have spin parameters substantially smaller than this limit, *e.g.* for the Hulse-Taylor pulsar we have `spin1` $\lesssim 6.5 \times 10^{-3}$. (See [12] for discussion and references.)

As a sanity check and a demonstration of how to calculate the spin parameters, we verify the numbers quoted above for the Hulse-Taylor binary pulsar. The pulsar is a neutron star with $m \approx 1.4M_{\odot}$, a radius $R \approx 10\text{km}$, and a spin frequency of about 17Hz. [Don't confuse the spin period (1/17 sec) with the orbital period (8hrs).] If we model the moment of inertia, I , as that of a sphere with uniform density, we obtain

$$\begin{aligned} L_{\text{Hul-Tay}} &\sim 2\pi I f_{\text{spin}} \\ &\sim \frac{4\pi}{5} f_{\text{spin}} M R^2 \\ &\sim 1.2 \times 10^{47} (\text{gram cm}^2/\text{sec}) \end{aligned}$$

Using Eq.(6.9.8) to convert this to the dimensionless quantity we have

$$\text{spin}_{\text{Hul-Tay}} \sim 6.8 \times 10^{-3} . \quad (6.9.9)$$

This is reasonable agreement with the numbers given above.

We can also use the above conversions to give the angular momentum of the Hulse-Taylor pulsar in geometrized units

$$S_{\text{Hul-Tay}} \approx 3 \times 10^8 \text{ cm}^2 \approx 7 \text{ acres} . \quad (6.9.10)$$

Like all post-Newtonian equations, Eqs. (6.9.1) and (6.9.2) are slow-motion approximations to the fully relativistic equations of motion; therefore they are most accurate – and behave best – for smaller values of the spin parameters. The GRASP routines have been tested for a modest range of masses ($0.1M_{\odot}, 10M_{\odot}$) and spins ($-0.2, +0.2$) in the frequency band $60\text{Hz} \leq f_{\text{orb}} \leq 2000\text{Hz}$; they seem to give reasonable results in this regime.

Finally, the admonitions and suggestions given in Sec. (6.8) about setting up banks of filters hold here also: test the chirp-generating functions with the extreme values of masses and spins you intend to use in your search. If the functions give satisfactory results at the “corners” of the parameter space, they should work on the interior of the parameter space.

6.9.2 2.5 Post-Newtonian corrections to the inspiral chirp

A quick start: Most GRASP users probably generate chirps by calling `make_filters()` (Sec. 6.10). This is all they will need to know:

1. If you are using the routine `make_filters()` (Sec. 6.10) to generate templates and you wish to include the 2.5 post-Newtonian corrections in your chirp calculations, simply set `order = 5` when you call `make_filters()`. The chirps returned will be 2.5 post-Newtonian chirps.
2. If you do not want the 2.5 post-Newtonian corrections – they will slow down your chirp calculations – set `order ≤ 4` when you call `make_filters()`. This is probably what you have been doing, so you won't need to change anything.
3. The behavior of the post-Newtonian series does not get better as you go to higher order: if anything, it gets worse. Therefore, if you use 2.5 post-Newtonian order templates in your search, the admonition in Sec. 6.8 about checking the “corners” of the filter-bank space hold in spades at higher order

Now, for a more thorough explanation: The 2.5 post-Newtonian corrections to the orbital frequency and phase have been calculated by Blanchet [9]. These include corrections of $O[(v/c)^5]$ beyond the quadrupole approximation in the phase and frequency evolution. The expressions are

$$f(t) = \frac{M_{\odot}}{16\pi T_{\odot} m_{\text{tot}}} \left\{ \Theta^{-3/8} + \dots - \left(\frac{7729}{21504} + \frac{3}{256}\eta \right) \pi \Theta^{-1} \right\}, \quad (6.9.1)$$

and

$$\phi(t) = \phi_c - \frac{1}{\eta} \left\{ \Theta^{5/8} + \dots - \left(\frac{38645}{172032} + \frac{15}{2048}\eta \right) \pi \log \left(\frac{\Theta}{\Theta_o} \right) \right\}. \quad (6.9.2)$$

Here Θ is the dimensionless time variable given by Eq. (6.4.3). The ellipses represent the second post-Newtonian terms already given in Eqs. (6.4.1) and (6.4.2), as well as the spin correction given in Eqs. (6.9.1) and (6.9.2). The constant Θ_o is arbitrary; changing its value shifts the phase by a constant. In the code, it is set to the value of the time parameter Θ at the beginning of the chirp; this insures that the argument of logarithm is close to unity throughout the chirp. The value of ϕ_c is then chosen so the phase is zero at the start time, *i.e.* when the orbital frequency is equal to `Initial_Freq`.

Computing the logarithm is slow, therefore the code is designed to logically step over the 2.5 post-Newtonian corrections unless they are explicitly called for. Perhaps, in the future, we will write some optimized code to speed up the log calculation.

How to (not) include the 2.5-post-Newtonian corrections to the waveform in your chirp calculations: As we stated above, simply changing the value of the parameter `order` is all that is needed in `make_filters()`. However, if you are making direct calls to the underlying routines `phase_frequency()` or `chirp_filters()` (as opposed to having `make_filters()` do it for you) you need to set `n_phaseterms=6`, and `phaseterms[5]=1.0`. This will turn on the 2.5 post-Newtonian corrections. To illustrate this, here is how the code block in the examples `phase_evoltn()` and `filters()` has to be modified to include the 2.5 post-Newtonian corrections.

```

/* post-Newtonian [O(1/c^n)] terms you wish to include (or suppress)
   in the phase and frequency evolution: */
n_phaseterms=6;
phaseterms[0] =1.;          /* The Newtonian piece          */
phaseterms[1] =0.;          /* There is no O(1/c) correction */
phaseterms[2] =1.;          /* The post-Newtonian correction */

```

```
phaseterms[3] =1.;      /* The 3/2 PN correction      */
phaseterms[4] =1.;      /* The 2 PN correction        */
phaseterms[5] =1.;      /* The 5/2 PN correction      */
```

Notice that `n_phaseterms=6` and `phaseterms[5] =1.0`. Nothing else needs to be changed in the examples.

6.10 Function: make_filters()

```
void make_filters(float m1, float m2, float *ch1, float *ch2, float fstart,
int n, float srate, int *filled, float *t_coal, int err_cd_sprs, int order)
```

This function is an even more stripped down chirp generator, which fills a pair of arrays with waveforms for an inspiraling binary. The two chirps differ in phase by $\pi/2$ radians and are given by Eqs.(6.6.1) and (6.6.2). This routine assumes spinless masses, and computes a chirp with phase corrections up to a specified post-Newtonian order.

The arguments are:

m1: Input. The mass of body-1 in solar masses.

m2: Input. The mass of body-2 in solar masses.

ch1: Output. Upon return, `ch1[0..filled-1]` contains the 0-phase chirp. The remaining array elements `ch1[filled..n-1]` are set to zero.

ch2: Output. Upon return, `ch2[0..filled-1]` contains the $\pi/2$ -phase chirp. The remaining array elements `ch2[filled..n-1]` are set to zero.

fstart: Input. The starting gravity-wave frequency of the chirp in Hz. Note: this is twice the orbital frequency!

n: Input. The length of the arrays `ch1[]` and `ch2[]`.

srate: Input. The sample rate, in Hz. This is $1/\Delta t$ where Δt is the time interval between successive entries in the `ch1[]` and `ch2[]` arrays.

filled: Output. The number of of time steps actually computed, before the chirp calculation was terminated, or until the arrays were filled (hence $filled \leq n$). Thus, on return, only the array elements `ch1[0..filled-1]` and `ch2[0..filled-1]` are contain the chirp; the remaining array elements are zero-padded.

t_coal: Output. The time to coalescence measured from the first point output, in `ch*[0]`.

err_cd_sprs: Input. Error code suppression. This integer specifies the level of disaster encountered in the computation of the chirp for which the user will be explicitly warned with a printed message. Set to 0: prints all the termination messages. Set to 4000: suppresses all but a few messages which are harbingers of true disaster. (See identical argument in `chirp_filters()`).

order: Input. The order of the post-Newtonian approximation. This ranges from 0 (quadrupole approximation) up to 5 (2.5 post-Newtonian order). Setting `order=4` gives second post-Newtonian chirps. Technically, `order` is the power in (v/c) past the quadrupole approximation to which the post-Newtonian expansion is taken.

This routine assumes that you have already allocated storage arrays for the chirps. Note that the coalescence time may be much later than the last non-zero entry written into the `ch1[]` and `ch2[]` arrays.

Author: Bruce Allen, ballen@dirac.phys.uwm.edu

Comments: None.

6.11 Stationary phase approximation to binary inspiral chirps

Much of the literature on binary inspiral data analysis approximates chirps in the frequency domain by the method of stationary phase. The main reason for this approximation is the need to generate analytical expressions in the frequency domain, where almost all of the optimal filtering algorithm takes place. It is also in some sense more natural to generate waveforms in the frequency domain rather than the time domain because the post-Newtonian energy and flux functions used to construct even the time-domain waveforms are expanded in powers of the orbital frequency. A side benefit is that the post-Newtonian expansion seems better behaved in the frequency domain—that is, there is no nonmonotonic frequency evolution as depicted in figure 24.

Therefore, GRASP includes `sp_filters()`, a stationary phase chirp generator similar to `make_filters()`. The advantage of this function is a considerable savings in CPU time by avoiding FFTs of time-domain chirps in the generation of matched filters. The disadvantages are unknown—the question of which version of the post-Newtonian expansion (time-domain or frequency-domain) is a better approximation to the real thing is currently wide open.

The stationary phase approximation can be found in any textbook on mathematical methods in physics. An excellent discussion in the context of binary inspiral can be found in section II C of [21]. Another inspiral-related discussion can be found in [22], where it is shown that the errors induced by the stationary phase approximation itself [as opposed to differences between $t(f)$ and $f(t)$] are effectively fifth post-Newtonian order.

The stationary phase approximations to the Fourier transforms of $h_c(t)$ and $h_s(t)$ [Eqs. (6.6.1,6.6.2)] are given in the restricted post-Newtonian approximation by

$$\tilde{h}_c(f) = \left(\frac{5\mu}{96M_\odot}\right)^{1/2} \left(\frac{M}{\pi^2 M_\odot}\right)^{1/3} f^{-7/6} T_\odot^{-1/6} \exp[i\Psi(f)], \quad (6.11.1)$$

$$\tilde{h}_s(f) = i\tilde{h}_c(f), \quad (6.11.2)$$

where f is the gravitational wave frequency in Hz, M is the total mass of the binary, and μ is the reduced mass. Note that $\tilde{h}_{c,s}(f)$ have dimensions of 1/Hz. The instrument strain per Hz, $\tilde{h}(f)$, is obtained from a linear superposition of $\tilde{h}_{c,s}(f)$ in exactly the same way as $h(t)$ is obtained from $h_{c,s}(t)$. See the discussion following Eqs. (6.6.1,6.6.2).

The restricted post-Newtonian approximation assumes that the evolution of the waveform amplitude is given by the 0'th-order post-Newtonian expression, but that the phase evolution is accurate to higher order. This phase is given by

$$\begin{aligned} \Psi(f) = & 2\pi f t_c - 2\phi_c - \pi/4 \\ & + \frac{3}{128\eta} \left[x^{-5} + \left(\frac{3715}{756} + \frac{55}{9}\eta\right) x^{-3} - 16\pi x^{-2} \right. \\ & + \left(\frac{15\,293\,365}{508\,032} + \frac{27\,145}{504}\eta + \frac{3085}{72}\eta^2\right) x^{-1} \\ & \left. + \left(\frac{38\,645}{252} + 5\eta\right) \pi \ln x \right], \end{aligned} \quad (6.11.3)$$

where $x = (\pi M f T_\odot / M_\odot)^{1/3}$, the coalescence phase ϕ_c is determined by the binary ephemeris, and the coalescence time t_c is the time at which the bodies collide. The chirps \tilde{h}_c and \tilde{h}_s are given $\phi_c = 0$ and $\phi_c = -\pi/4$, respectively. All but the last term of (6.11.3) can be found in [23]; the last term was computed from [9] by Ben Owen and Alan Wiseman.

The chirps are set to zero for frequencies below the requested starting frequency and above an upper cutoff f_c (see below). This square windowing in the frequency domain produces ringing at the beginning and

end of the waveform in the time domain (see Fig. 26). For data analysis purposes it appears this ringing is not very important: it produces a mismatch (see Sec. 9.7) between waveforms generated by `sp_filters()` and by `make_filters()` of a fraction of a percent—if the stationary phase waveform is cut off at the same frequency f_c as the time-domain waveform.

The choice of the cutoff frequency f_c is somewhat problematic. Physically, f_c should correspond to the epoch when orbital inspiral turns to headlong plunge. The formula for f_c currently is not known for a pair of comparably massive objects, but in the limit of extreme mass ratio (and no spins) it should be equivalent to the well-known innermost stable circular orbit (ISCO) of Schwarzschild geometry. The frequency of the Schwarzschild ISCO can be computed exactly and is given in Hz by

$$f_c = \frac{M_\odot}{6^{3/2} \pi M T_\odot}. \tag{6.11.4}$$

Use of the Schwarzschild f_c for all binaries is a kludge which seems to work surprisingly well in the sense that it yields an f_c close to that at which the time-domain waveforms cut off due to df/dt going negative (see the `compare_chirps` program).

6.12 Function: sp_filters()

```
void sp_filters(float m1, float m2, float *ch1, float *ch2, float fstart,
int n, float srate, float f_c, float t_c, int order)
```

This function generates stationary phase approximations to binary inspiral chirp waveforms. Its input and output are similar to `make_filters()`. The difference is that the chirps are generated in the frequency domain using the stationary phase approximation.

The arguments are:

`m1`: Input. The mass of body-1 in solar masses.

`m2`: Input. The mass of body-2 in solar masses.

`ch1`: Output. Upon return, `ch1[0..n-1]` contains the stationary phase approximation to $\tilde{h}_c(f)$ [Eq. (6.11.1)] in the same format as would be returned by a `realft()` of a time-domain function sampled at rate `srate`. That is, except for DC and Nyquist frequencies, `ch1[2*i]` and `ch1[2*i+1]` contain respectively the real and imaginary parts of $\tilde{h}_c(f)$ for $f = i * \text{srate}/n$. This function sets `ch1[0]` (DC) and `ch1[1]` (Nyquist) to zero. The chirp is also set to zero for $f < \text{fstart}$ and for $f > \text{f}_c$ (see section 6.11 for f_c). The output `ch1[0..n-1]` has dimensions of 1/Hz.

`ch2`: Output. Upon return, `ch2[]` contains $\tilde{h}_s(f)$ in the same way that `ch1[]` contains $\tilde{h}_c(f)$.

`fstart`: Input. The starting gravitational-wave frequency of the chirp in Hz. Note: this is twice the orbital frequency!

`n`: Input. The length of the arrays `ch1[0..n-1]` and `ch2[0..n-1]`.

`srate`: Input. The sample rate, in Hz.

`f_c`: Input. The coalescence frequency f_c , as described in section 6.11. This is the high-frequency cutoff of the chirps.

`t_c`: Input. The coalescence time, in seconds. Note this is the time of the *end* of the chirp (see section 6.13).

`order`: Input. Order of generated chirps in $(\pi M f T_\odot / M_\odot)^{1/3}$ (twice the post-Newtonian order).

This function assumes that you have already allocated storage for the chirps.

Author: Benjamin Owen, owen@tapir.caltech.edu

Comments: The `sp_filters()` function doesn't include spins yet. It will be simple to add higher-order post-Newtonian phase terms as they appear in the literature.

6.13 Example: compare_chirps program

This example compares a chirp generated by `sp_filters()` to a chirp with identical parameters generated by `make_filters()`; the output is shown in Figure 26. The chirp generated by `sp_filters()` is transformed to the time domain, and the two chirps are superimposed on one graph.

```
/* GRASP: Copyright 1997,1998 Bruce Allen */
/* compare_chirps.c: by Benjamin Owen, 1997 */

#include "grasp.h"

#define FSTART 40.      /* GW starting frequency, in Hz */
#define SRATE 10000.   /* Sample rate, in Hz */
#define LENGTH 32768  /* Twice no of freq bins = samples in time domain */
#define MASS1 10.0     /* mass of first body, in solar masses */
#define MASS2 10.0     /* mass of second body, in solar masses */

int main() {
    FILE *fp;
    float t_c, f_c, *sp, *td, *dummy;
    int i, pn_order;
    void realft(float*, unsigned long, int);

    /* Allocate memory for chirps */
    sp = (float *)malloc(sizeof(float)*LENGTH);
    td = (float *)malloc(sizeof(float)*LENGTH);
    dummy = (float *)malloc(sizeof(float)*LENGTH);

    /* order of (v/c) used in chirp calculation comparison */
    pn_order=4;

    /* Generate time-domain chirp for comparison purposes */
    make_filters(MASS1, MASS2, td, dummy, FSTART, LENGTH, SRATE, &i, &t_c, 2000, pn_order);

    /* Generate stationary phase chirp in frequency domain */
    f_c = pow(6, -1.5)/M_PI/(MASS1+MASS2)/TSOLAR;
    sp_filters(MASS1, MASS2, sp, dummy, FSTART, LENGTH, SRATE, f_c, t_c, pn_order);

    /* Transform stationary phase chirp to the time domain */
    realft(sp-1, LENGTH, -1);

    /* Graph both chirps in the time domain. First output file. */
    fp = fopen("compare_chirps.output", "w");
    for (i=0; i<LENGTH; i++)
        fprintf(fp, "%f %f %f\n", i/SRATE, td[i], sp[i]*2*SRATE/LENGTH);
    fclose(fp);

    /* Now graph the contents of the file using xmgr */
    system("xmgr -nxy compare_chirps.output &");

    return 0;
}
```

Note that to get the graph to show both chirps as simultaneous functions of time, `sp_filters()` needed to know the coalescence time found by `make_filters()`, so the latter function is called first. If the coalescence time input to `sp_filters()` had been zero, its chirp would have finished at the beginning—or equivalently, the end—of the time-domain data.

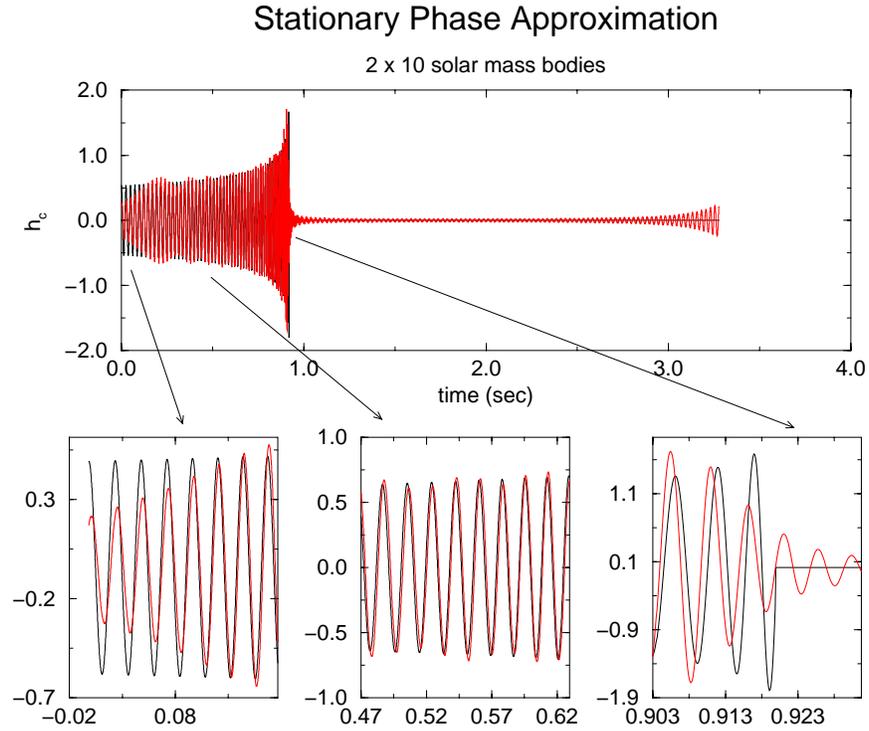


Figure 26: The output of `compare_chirps`, comparing the stationary-phase approximate waveform FFT'd into the time domain (red curve) with a 2nd-order post-Newtonian chirp calculated in the time domain, using `make_filters()` (black curve). The lower part of the graph shows three interesting regions of the upper (complete) graph. The bottom left detail shows the Gibbs startup-transient, the bottom middle detail shows a typical region of good agreement, and the bottom right detail shows the Gibbs turn-off transient. The Gibbs startup transient is also visible at the far right of the upper figure, which is periodically identified with the far left.

Also note that the inverse `realft()` of the stationary phase chirp had to be multiplied by a factor to be comparable to the time-domain chirp. The $2/\text{LENGTH}$ factor is left out of the inverse `realft()`, and the `SRATE` factor is needed to keep the dimensions right. (Also, the forward `realft()` of the time-domain chirp would need to be multiplied by $1/\text{SRATE}$ to compare to the stationary phase chirp.)

6.14 Wiener (optimal) filtering

The technique of *optimal filtering* is a well-studied and well-understood technique which can be used to search for characteristic signals (in our case, chirps) buried in detector noise. In order to establish notation, we begin this section with a brief review of the optimal filtering technique.

Suppose that the detector output is a dimensionless strain $h(t)$. (In Section 3 we show how to construct this quantity for the CIT 40-meter prototype interferometer, using the recorded digital data stream). We denote by $C(t)$ the waveform of the signal (i.e., the chirp) which we hope to find, hidden in detector noise, in the signal stream $h(t)$. Since we would like to know about chirps which start at different possible times t_0 , we'll take $C(t) = \alpha T(t - t_0)$ where $T(t)$ is the canonically normalized waveform of a chirp which enters the sensitivity band of the interferometer at time $t = 0$. The constant α quantifies the strength of the signal we wish to extract as compared to an otherwise identical signal of canonical strength (we will discuss how this canonical normalization is defined shortly). In other words, $T(t)$ contains all the information about the chirp we are searching for apart from the arrival time and the strength, which are given by t_0 and α respectively. For the moment, we will ignore the fact that the chirps come in two different phase "flavors".

We will construct a signal S , which is a number defined by

$$S = \int_{-\infty}^{\infty} dt h(t)Q(t), \quad (6.14.1)$$

where $Q(t)$ is an optimal filter function in time domain, which we will shortly determine in a way that maximizes the signal-to-noise ratio S/N or SNR. We will assume that Q is a real function of time.

We use the Fourier transform conventions of (3.9.3) and (3.9.4), in terms of which we can write the signal S as

$$\begin{aligned} S &= \int_{-\infty}^{\infty} dt \int_{-\infty}^{\infty} df \int_{-\infty}^{\infty} df' e^{-2\pi i f t + 2\pi i f' t} \tilde{h}(f) \tilde{Q}^*(f') \\ &= \int_{-\infty}^{\infty} df \int_{-\infty}^{\infty} df' \delta(f - f') \tilde{h}(f) \tilde{Q}^*(f') \\ &= \int_{-\infty}^{\infty} df \tilde{h}(f) \tilde{Q}^*(f). \end{aligned} \quad (6.14.2)$$

This final expression gives the signal value S written in the frequency domain, rather than in the time domain.

Now we can ask about the expected value of S , which we denote $\langle S \rangle$. This is the average of S over an ensemble of detector output streams, each one of which contains an identical chirp signal $C(t)$ but different realizations of the noise:

$$h(t) = C(t) + n(t). \quad (6.14.3)$$

So for each different realization, $C(t)$ is exactly the same function, but $n(t)$ varies from each realization to the next. We will assume that the noise has zero mean value, and that the phases are randomly distributed, so that $\langle \tilde{n}(f) \rangle = 0$. We can then take the expectation value of the signal in the frequency domain, obtaining

$$\langle S \rangle = \int_{-\infty}^{\infty} df \langle \tilde{h}(f) \rangle \tilde{Q}^*(f) = \int_{-\infty}^{\infty} df \tilde{C}(f) \tilde{Q}^*(f). \quad (6.14.4)$$

We now define the *noise* N to be the difference between the signal value and its mean for any given element of the ensemble:

$$N \equiv S - \langle S \rangle = \int_{-\infty}^{\infty} df \tilde{n}(f) \tilde{Q}^*(f). \quad (6.14.5)$$

The expectation value of N clearly vanishes by definition, so $\langle N \rangle = 0$. The expected value of N^2 is non-zero, however. It may be calculated from the (one-sided) strain noise power spectrum of the detector $S_h(f)$, which is defined by

$$\langle \tilde{n}(f) \tilde{n}^*(f') \rangle = \frac{1}{2} S_h(|f|) \delta(f - f'), \quad (6.14.6)$$

and has the property that

$$\langle n^2(t) \rangle = \int_0^\infty S_h(f) df. \quad (6.14.7)$$

We can now find the expected value of N^2 , by squaring equation (6.14.5), taking the expectation value, and using (6.14.6), obtaining

$$\begin{aligned} \langle N^2 \rangle &= \int_{-\infty}^\infty df \int_{-\infty}^\infty df' \tilde{Q}^*(f) \langle \tilde{n}(f) \tilde{n}^*(f') \rangle \tilde{Q}(f') \\ &= \frac{1}{2} \int_{-\infty}^\infty df S_h(|f|) |\tilde{Q}(f)|^2 \\ &= \int_0^\infty df S_h(f) |\tilde{Q}(f)|^2. \end{aligned} \quad (6.14.8)$$

There is a nice way to write the formulae for the expected signal and the expected noise-squared. We introduce an “inner product” defined for any pair of (complex) functions $A(f)$ and $B(f)$. The inner product is a complex number denoted by (A, B) and is defined by

$$(A, B) = \int_{-\infty}^\infty df A(f) B^*(f) S_h(|f|). \quad (6.14.9)$$

Because S_h is positive, this inner product has the property that $(A, A) \geq 0$ for all functions $A(f)$, vanishing if and only if $A = 0$. This inner product is what a mathematician would call a “positive definite norm”; it has all the properties of an ordinary dot product of vectors in three-dimensional Cartesian space.

In terms of this inner product, we can now write the expected signal, and the expected noise-squared, as

$$\langle S \rangle = \left(\frac{\tilde{C}}{S_h}, \tilde{Q} \right) \quad \text{and} \quad \langle N^2 \rangle = \frac{1}{2} (\tilde{Q}, \tilde{Q}). \quad (6.14.10)$$

(Note that whenever S_h appears inside the inner product, it refers to the function $S_h(|f|)$ rather than $S_h(f)$.) Now the question is, how do we choose the optimal filter function Q so that the expected signal is as large as possible, and the expected noise-squared is as small as possible? The answer is easy! Recall Schwarz’s inequality for inner products asserts that

$$(A, B)^2 \leq (A, A)(B, B), \quad (6.14.11)$$

the two sides being equal if (and only if) A is proportional to B . So, to maximize the signal-to-noise ratio

$$\left(\frac{S}{N} \right)^2 = \frac{\langle S \rangle^2}{\langle N^2 \rangle} = 2 \frac{\left(\frac{\tilde{C}}{S_h}, \tilde{Q} \right)^2}{(\tilde{Q}, \tilde{Q})} \quad (6.14.12)$$

we choose

$$\tilde{Q}(f) \propto \frac{\tilde{C}(f)}{S_h(|f|)} = \alpha \frac{\tilde{T}(f)}{S_h(|f|)} e^{2\pi i f t_0}. \quad (6.14.13)$$

The signal-to-noise ratio defined by equation (6.14.12) is normalized in a way that is generally accepted and used. Note that the definition is independent of the normalization of the optimal filter \tilde{Q} , since that quantity

appears quadratically in both the numerator and denominator. However if we wish to speak about ‘‘Signal’’ values rather than about signal-to-noise values, then the normalization of \tilde{Q} is relevant. If we choose the constant of proportionality to be $2\alpha^{-1}$, (i.e. set $\alpha = 2$, for reasons we will discuss shortly) then we can express the template in terms of the canonical waveform,

$$\tilde{Q}(f) = 2 \frac{\tilde{T}(f)}{S_h(|f|)} e^{2\pi i f t_0} \quad (6.14.14)$$

Going back to the definition of our signal S , you will notice that the signal S for ‘‘arrival time offset’’ t_0 is given by

$$\begin{aligned} S &= \int_{-\infty}^{\infty} df \tilde{h}(f) \tilde{Q}^*(f) \\ &= 2 \int_{-\infty}^{\infty} df \frac{\tilde{h}(f) \tilde{T}^*(f)}{S_h(|f|)} e^{-2\pi i f t_0}. \end{aligned} \quad (6.14.15)$$

Given a template \tilde{T} and the signal \tilde{h} , the signal values can be easily evaluated for any choice of arrival times t_0 by means of a Fourier transform (or FFT, in numerical work). Thus, it is not really necessary to construct a different filter for each possible arrival time; one can filter data for all possible choices of arrival time with a single FFT.

The signal-to-noise ratio for this optimally-chosen filter can be determined by substituting the optimal filter (6.14.14) into equation (6.14.12), obtaining

$$\left(\frac{S}{N}\right)^2 = 2 \int_{-\infty}^{\infty} df \frac{|\tilde{C}(f)|^2}{S_h(|f|)} = 4 \int_0^{\infty} df \frac{|\tilde{C}(f)|^2}{S_h(f)} = 2\alpha^2 \left(\frac{\tilde{T}}{S_h(|f|)}, \frac{\tilde{T}}{S_h(|f|)}\right). \quad (6.14.16)$$

You will notice that the signal-to-noise ratio S/N in (6.14.12) is independent of the overall normalization of the filter Q : if we make Q bigger by a factor of ten, both the expected signal and the expected noise increase by exactly the same amount. For this reason, we can specify the normalization of the filter as we wish. Furthermore, it is obvious from (6.14.14) that normalizing the optimal filter is equivalent to specifying the normalization of the canonical signal waveform. It is traditional (for example in Cutler and Flanagan [21]) to choose

$$\left(\frac{\tilde{T}}{S_h(|f|)}, \frac{\tilde{T}}{S_h(|f|)}\right) = \frac{1}{2}. \quad (6.14.17)$$

With this normalization, the expected value of the squared noise is

$$\langle N^2 \rangle = \frac{1}{2}(\tilde{Q}, \tilde{Q}) = \frac{1}{2} \left(2 \frac{\tilde{T}}{S_h(|f|)}, 2 \frac{\tilde{T}}{S_h(|f|)}\right) = 1 \quad (6.14.18)$$

and the signal-to-noise ratio takes the simple form

$$\left(\frac{S}{N}\right)^2 = \alpha^2. \quad (6.14.19)$$

This adjustment or change of the filter normalization can be obtained by moving the (fictitious) astrophysical system emitting the chirp template either closer or farther away from us. Because the metric strain h falls off as $1/\text{distance}$, the measured signal strength S is then a direct measure of the inverse distance.

For example, consider a system composed of two $1.4 M_{\odot}$ masses in circular orbit. Let us normalize the filter \tilde{T} so that equation (6.14.17) is satisfied. This normalization corresponds to placing the binary system at some distance. For the purpose of discussion, suppose that this distance is 15 megaparsecs (i.e.,

choosing $T(t)$ to be the strain produced by an optimally-oriented two $\times 1.4 M_\odot$ system at a distance of 15 megaparsecs). If we then detect a signal with a signal-to-noise ratio $S/N = 30$, this corresponds to detecting an optimally-oriented source at a distance of half a megaparsec. Note that the normalization we have chosen has the r.m.s. noise $\sqrt{\langle N^2 \rangle} = 1$ and therefore the signal and signal-to-noise values are equal.

The functions `correlate()` and `productc()` are designed to perform this type of optimal filtering. We document these routines in the following section and in Section 16, then provide a simple example of an optimal filtering program.

There is an additional complication, arising from the fact that the gravitational radiation from a binary inspiral event is a linear combination of two possible orbital phases, as may be seen by reference to equations (6.6.1) and (6.6.2). Thus, the strain produced in a detector is a linear combination of two waveforms, corresponding to each of the two possible (0° and 90°) orbital phases:

$$h(t) = \alpha T_0(t) + \beta T_{90}(t) + n(t). \quad (6.14.20)$$

Here the subscripts 0 and 90 label the two possible orbital phases; the constants α and β depend upon the distance to the source (and the normalization of the templates) and the orientation of the source relative to the detector. Thus $T_0(t)$ denotes the (suitably normalized) function $h_c(t)$ given by equation (6.6.1) and $T_{90}(t)$ denotes the (suitably normalized) function $h_s(t)$ given by equation (6.6.2).

In the optimal filtering, we are now searching for a pair of amplitudes α and β rather than just a single amplitude. One can easily do this by choosing a filter function which corresponds to a complex-valued signal in the time-domain:

$$\tilde{Q}(f) = 2 \frac{\tilde{T}_0(f) - i\tilde{T}_{90}(f)}{S_h(|f|)} e^{2\pi i f t_0}. \quad (6.14.21)$$

We will assume that the individual filters for each polarization are normalized by the convention just described, and that they are orthogonal:

$$\left(\frac{\tilde{T}_0}{S_h}, \frac{\tilde{T}_0}{S_h} \right) = \frac{1}{2}, \quad \text{and} \quad \left(\frac{\tilde{T}_{90}}{S_h}, \frac{\tilde{T}_{90}}{S_h} \right) = \frac{1}{2}, \quad \text{and} \quad \left(\frac{\tilde{T}_0}{S_h}, \frac{\tilde{T}_{90}}{S_h} \right) = 0. \quad (6.14.22)$$

Note that T_0 and T_{90} are only exactly orthogonal in the adiabatic limit where they each have many cycles in any frequency interval df in which the noise power spectrum $S_h(f)$ changes significantly. Also note that the filter function $\tilde{Q}(f)$ does not correspond to a real filter $Q(t)$ in the time domain, since $\tilde{Q}(-f) \neq \tilde{Q}^*(f)$, so that the signal

$$S(t_0) = \left(\frac{\tilde{h}}{S_h}, \tilde{Q} \right) \quad (6.14.23)$$

is a complex-valued function of the lag t_0 . We define the noise as before, by $N = S - \langle S \rangle$. Its mean-squared modulus is

$$\begin{aligned} \langle |N|^2 \rangle &= \frac{1}{2} (\tilde{Q}, \tilde{Q}) \\ &= 2 \left(\frac{\tilde{T}_0 - i\tilde{T}_{90}}{S_h}, \frac{\tilde{T}_0 - i\tilde{T}_{90}}{S_h} \right) \\ &= 2 \left[\left(\frac{\tilde{T}_0}{S_h}, \frac{\tilde{T}_0}{S_h} \right) + \left(\frac{\tilde{T}_{90}}{S_h}, \frac{\tilde{T}_{90}}{S_h} \right) \right] = 2, \end{aligned} \quad (6.14.24)$$

where we have made use of the orthonormality relation (6.14.22). This value is twice as large as the expected noise-squared in the case of a single phase waveform considered previously.

The expected signal at zero lag $t_0 = 0$ is

$$\langle S \rangle = \left(\frac{\langle \tilde{h} \rangle}{S_h}, \tilde{Q} \right) = 2 \left(\frac{\alpha \tilde{T}_0 + \beta \tilde{T}_{90}}{S_h}, \frac{\tilde{T}_0 - i \tilde{T}_{90}}{S_h} \right) = \alpha + i\beta. \quad (6.14.25)$$

Hence the signal-to-noise ratio is

$$\frac{\langle S \rangle}{\sqrt{\langle |N|^2 \rangle}} = \frac{1}{\sqrt{2}}(\alpha + i\beta). \quad (6.14.26)$$

In the absence of a signal $\langle S \rangle = 0$ the expected value of the square of this quantity (from the definition of N) is unity:

$$\frac{\langle |S|^2 \rangle}{\langle |N|^2 \rangle} = 1. \quad (6.14.27)$$

In the presence of a signal, the squared signal-to-noise ratio is

$$\frac{|\langle S \rangle|^2}{\langle |N|^2 \rangle} = \frac{1}{2}(\alpha^2 + \beta^2) \quad (6.14.28)$$

In the case discussed previously, for a single-phase signal, we pointed out that there was general agreement on the definition of signal-to-noise value. In the present case (a complex or two-phase signal) there is no such agreement. The definition given above is the one used by most experimenters: it is a quantity whose square has expected value of unity in the absence of a signal. However the definition often used in this subject is

$$\left(\frac{S}{N} \right)_{\text{Cutler and Flanagan}} = \left(\frac{S}{N} \right)_{\text{Owen}} = \left(\frac{S}{N} \right)_{\text{Thorne}} = \max_{t_0} |S(t_0)| = \sqrt{2} \left(\frac{S}{N} \right)_{\text{GRASP}}. \quad (6.14.29)$$

Note that because $S(t_0)$ is complex, we maximize the modulus. This is a quantity whose expected squared value, in the absence of a signal, is 2. To avoid confusion in the future, we will use a different symbol for this quantity, and define

$$\rho \equiv \left(\frac{S}{N} \right)_{\text{Cutler and Flanagan}} = \left(\frac{S}{N} \right)_{\text{Owen}} = \left(\frac{S}{N} \right)_{\text{Thorne}} = \sqrt{2} \left(\frac{S}{N} \right)_{\text{GRASP}}. \quad (6.14.30)$$

This quantity is equal to the signal value alone (rather than the signal value divided by the expected noise).

Another way to understand these two different choices of normalization, and to understand why the conventional choice of normalization is ρ , is that conventionally one treats the two-phase case in the same way as the single phase case, but regards $\frac{S}{N}$ as a function of a phase parameter, $\theta = \arctan(\beta/\alpha)$. For any fixed θ , $\frac{S}{N}(\theta)$ has rms value one, but the statistic $\max_{\theta} \frac{S}{N}(\theta)$ has rms value $\sqrt{2}$.

The attentive reader will notice, with our choice of filter and signal normalizations, that we have lost a factor of $\sqrt{2}$ in the signal-to-noise ratio compared to the case where we were searching for only a single phase of waveform. The expected signal strength in the presence of a 0-phase signal is the same as in the single-phase case, but the expected (noise)² has been doubled. This is because of the additional uncertainty associated with our lack of information about the relative contributions of the two orbital phases. In other words, if we know in advance that a waveform is composed entirely of the zero-degree orbital phase, then the expectation value of the signal-to-noise, determined by equation (6.14.12) would be given by $\langle S \rangle/N = \sqrt{2}\alpha$. However if we need to search for the correct linear combination of the two possible phase waveforms, then the expectation value of the signal-to-noise is reduced to $\langle S \rangle/N = \alpha$. However, as we will see in the next section, this reduction in signal-to-noise ratio does not significantly affect our ability to detect signals with a given false alarm rate.

6.15 Comparison of signal detectability for single-phase and two-phase searches

The previous Section 6.14 described optimal filtering searches in two cases - looking for:

- A signal of known phase, proportional to T_0 , and
- A signal of unknown phase, which is some linear combinations of T_0 and T_{90} .

With the choice of filter normalizations made previously, the expected signal produced by a source αT_0 would be the same for both searches, but the expected (noise)² was higher in the two-phase case. One might wonder if this reduced SNR means that a two-phase search reduces ones ability to identify signals. The answer turns out to be “not significantly”.

The reason for this is that the distribution of signal values produced by detector noise alone in the single- and two-phase cases are quite different. In order to answer the question: “what is the smallest signal detectable” we need to fix a false alarm rate. For a given time-duration of data, this is equivalent to fixing a false alarm probability. Let us assume that this probability has been fixed to be a small value ϵ , and compare the single- and two-phase searches.

In the single-phase case, in the absence of a source, the values of the signal S (6.14.15) are Gaussian random variables with a mean-squared value of 1. Hence the threshold S_0 determined by the false alarm rate must be set so that there is probability ϵ of S falling outside the range $[-S_0, S_0]$. This means that

$$\epsilon = 2 \frac{1}{\sqrt{2\pi}} \int_{S_0}^{\infty} \exp(-x^2/2) dx = \operatorname{erfc}(S_0/\sqrt{2}). \quad (6.15.1)$$

The solution to this equation is the threshold as a function of the false alarm probability:

$$S_0^{\text{single-phase}}(\epsilon) = \sqrt{2} \operatorname{erfc}^{-1}(\epsilon). \quad (6.15.2)$$

Thus, for example, to obtain a false alarm probability of $\epsilon = 10^{-5}$ we need to set a threshold $S_0^{\text{single-phase}} = \sqrt{2} \operatorname{erfc}^{-1}(10^{-5}) = 4.417$. In this case, our minimum detectable signal has amplitude $\alpha = 4.417$.

In the two-phase case, the probability distribution of the signal in the absence of a source is different, because in this case the signal (6.14.23) is described by the probability distribution of a random variable r , where $r^2 = x^2 + y^2$ and x and y are independent random Gaussian variables with unit rms. Here, x and y are the real and imaginary parts of the signal (6.14.23) in the absence of a source. Their probability distribution is:

$$\begin{aligned} P(x)dxP(y)dy &= \frac{1}{2\pi} e^{-x^2/2} e^{-y^2/2} dx dy \\ &= \frac{1}{2\pi} e^{-r^2/2} r dr d\phi \\ &\Rightarrow \end{aligned} \quad (6.15.3)$$

$$P(r)dr = e^{-r^2/2} r dr. \quad (6.15.4)$$

In the final line, we have integrated over the irrelevant angular variable $\phi \in [0, 2\pi)$. So in the two-phase case, as before, the threshold value of the signal is set by requiring that the false alarm probability be ϵ :

$$\epsilon = \int_{S_0}^{\infty} \exp(-r^2/2) r dr = \exp(-S_0^2/2). \quad (6.15.5)$$

The solution here is that the threshold is $S_0^{\text{two-phase}} = \sqrt{-2 \ln \epsilon}$. For example, to obtain a false alarm probability of $\epsilon = 10^{-5}$ we need to set a threshold $S_0^{\text{two-phase}} = 4.799$. In this case, our minimum

detectable (0-phase) signal has amplitude $\alpha = 4.799$, which is only slightly higher than in the single-phase case.

It is not a coincidence that for a given false alarm rate, the amplitude of the minimum detectable signals are almost the same. Although the expected value of the single-phase signal² in the absence of a source is smaller than the expected value of the two-phase |signal|² in the absence of a source, the *tails* of the two probability distributions are almost identical. For the same false alarm probability ϵ the thresholds in the two instances are related by

$$\epsilon = \sqrt{\frac{2}{\pi}} \int_{S_0^{\text{single-phase}}}^{\infty} \exp(-x^2/2) dx = \int_{S_0^{\text{two-phase}}}^{\infty} \exp(-r^2/2) r dr \quad (6.15.6)$$

But for thresholds of reasonable size (small ϵ) both integrals are dominated by the region just to the right of S_0 , and in this neighborhood the integrands differ by a small factor of approximately $\sqrt{\frac{\pi S_0}{2}}$. Since ϵ varies exponentially with the threshold, there is a logarithmically small difference between the thresholds $S_0^{\text{single-phase}}$ and $S_0^{\text{two-phase}}$.

For a fixed false alarm probability, we can write the the two-phase threshold $S_0^{\text{two-phase}}$ as a function of the one-phase threshold $S_0^{\text{single-phase}}$:

$$S_0^{\text{two-phase}} = \sqrt{-2 \ln \left(\operatorname{erfc} \left(\frac{S_0^{\text{single-phase}}}{\sqrt{2}} \right) \right)}. \quad (6.15.7)$$

The plot of this relationship in Figure 27 shows clearly that once the thresholds are reasonably large, they are very nearly equal.

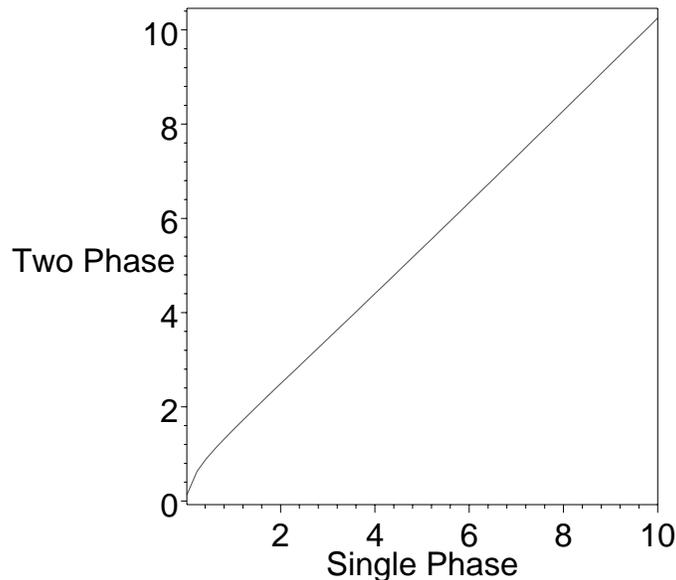


Figure 27: The threshold for a two-phase search $S_0^{\text{two-phase}}$ is shown as a function of the threshold for the single-phase search $S_0^{\text{single-phase}}$ which gives the same false alarm rate. When the false alarm rates are small, they are very nearly equal.

6.16 Function: correlate()

```
void correlate(float *s, float *h, float *c, float *r, int n)
```

This function evaluates the correlation (as a function of lag time t) defined by the discrete equivalent of equation (6.14.15):

$$s(t) = \frac{1}{2} \int_{-\infty}^{\infty} df \tilde{h}(f) \tilde{c}^*(f) \tilde{r}(f) e^{-2\pi i f t}. \quad (6.16.1)$$

It is assumed that $\tilde{h}(f)$ and $\tilde{c}(f)$ are Fourier transforms of real functions, and that $\tilde{r}(f)$ is real. The factor of 1/2 appears in (6.16.1) for efficiency reasons; in order to calculate the integral (6.14.15) one should set $\tilde{r}(f) = 2/S_h(f)$. The routine assumes that \tilde{r} vanishes at both DC and the Nyquist frequency.

The arguments are:

s: Output. Upon return, the array `s[0..n-1]` contains the correlation $s(t)$ at times

$$t = 0, \Delta t, 2\Delta t, \dots, (n-1)\Delta t. \quad (6.16.2)$$

h: Input. The array `h[0..n-1]` contains the positive frequency ($f \geq 0$) part of the complex function $\tilde{h}(f)$. The packing of \tilde{h} into this array follows the scheme used by the *Numerical Recipes* routine `realft()`, which is described between equations (12.3.5) and (12.3.6) of [1]. The DC component $\tilde{h}(0)$ is real, and located in `h[0]`. The Nyquist-frequency component $\tilde{h}(f_{\text{Nyquist}})$ is also real, and is located in `h[1]`. The array elements `h[2]` and `h[3]` contain the real and imaginary parts, respectively, of $\tilde{h}(\Delta f)$ where $\Delta f = 2f_{\text{Nyquist}}/n = (n\Delta t)^{-1}$. Array elements `h[2j]` and `h[2j+1]` contain the real and imaginary parts of $\tilde{h}(j \Delta f)$ for $j = 1, \dots, n/2 - 1$. It is assumed that $\tilde{h}(f)$ is the Fourier transform of a real function, so that `correlate()` can infer the negative frequency components from the equation $\tilde{h}(-f) = \tilde{h}^*(f)$

c: Input. The array `c[0..n-1]` contains the complex function \tilde{c} , packed in the same format as $\tilde{h}(f)$, with the same assumption that $\tilde{c}(-f) = \tilde{c}^*(f)$. Note that while you provide the function $\tilde{c}(f)$ to the routine, it is the *complex-conjugate* of the function contained in the array `c[]` which is used in calculating the correlation. Thus if \tilde{r} is positive, `correlate(s, c, c, r, n)` will always return `s[0] ≥ 0`.

r: Input. The array `r[0..n/2]` contains the values of the real function \tilde{r} used as a weight in the integral. This is often chosen to be (twice!) the inverse of the receiver noise, as in equation (6.14.15), so that $\tilde{r}(f) = 2/\tilde{S}_h(|f|)$. The array elements are arranged in order of increasing frequency, from the DC value at subscript 0, to the Nyquist frequency at subscript `n/2`. Thus, the j 'th array element `r[j]` contains the real value $\tilde{r}(j \Delta f)$, for $j = 0, 1, \dots, n/2$. Again it is assumed that $\tilde{r}(-f) = \tilde{r}^*(f) = \tilde{r}(f)$.

n: Input. The total length of the complex arrays `h` and `c`, and the number of points in the output array `s`. Note that the array `r` contains $n/2 + 1$ points. `n` must be even.

The correlation function calculated by this routine is $\frac{1}{2}FFT^{-1}[\tilde{h}\tilde{c}^*\tilde{r}]$ and has the same dimensions as the product $\tilde{h} \times \tilde{c} \times \tilde{r}$. The definition is

$$s_k = \frac{1}{2} \sum_{j=0}^{n-1} h_j c_j^* r_j e^{-2\pi i j k/n} \quad (6.16.3)$$

where it is understood that $\tilde{h}_{n-j} = \tilde{h}_j^*$ and that $\tilde{c}_{n-j} = \tilde{c}_j^*$, and that $\tilde{r}_{n-j} = \tilde{r}_j$.

Note that the input arrays $h[]$ and $c[]$ can be the same array. For example `correlate(s,c,c,r,n)` calculates the discrete equivalent of

$$s(t) = \frac{1}{2} \int_{-\infty}^{\infty} df |\tilde{c}(f)|^2 \tilde{r}(f) e^{-2\pi i f t}. \quad (6.16.4)$$

Author: Bruce Allen, ballen@dirac.phys.uwm.edu

Comments: For the sake of efficiency, this function does not include the contribution from either DC or Nyquist frequency bins to the correlation (these are negligible in any sensible data).

6.17 Function: avg_inv_spec()

```
void avg_inv_spec(float flo, float srate, int n, double decay, double *norm, float
*htilde, float* mean_pow_spec, float* twice_inv_noise)
```

This function maintains an auto-regressive moving average (see avg_spec()) of the power spectrum $S_h(f)$, and an array containing $2/S_h(f)$, which can be used for optimal filtering. This latter array is set to zero below a specified cuff-off frequency f_{low} .

The arguments are:

`flo`: Input. The low frequency cut-off f_{low} , in Hz.

`srate`: Input. The sample rate, in Hz.

`n`: Input. The number of points in the arrays.

`decay`: Input. The quantity $\exp(-\alpha)$ as defined in avg_spec(). Sets the characteristic decay time for the auto-regressive average.

`norm`: Input/Output. Used for internal storage. Set to 0 when you want to begin a new auto-regressive average. Must not be altered otherwise.

`htilde`: Input. The array `htilde[0..n-1]` contains the positive frequency FFT of the metric perturbation.

`mean_pow_spec`: Output. The array `mean_pow_spec[0..n/2]` contains the mean power spectrum. Should be zeroed when resetting to begin a new average. The array element `mean_pow_spec[0]` contains the power spectrum at DC, and the array element `mean_pow_spec[n/2]` contains the power spectrum at the Nyquist frequency `srate/2`.

`twice_inv_noise`: Output. The array `twice_inv_noise[0..n/2]` contains $2/S_h(f)$. It is set to zero for $f < f_{low}$. The array element `twice_inv_noise[0]` contains the DC value, and the array element `twice_inv_noise[n/2]` contains the value at the Nyquist frequency `srate/2`.

Author: Bruce Allen, ballen@dirac.phys.uwm.edu

Comments: We assume here that the “correct” thing to do is the average the spectrum, then invert it. There may be a better way to construct the weight function for an optimal filter, however.

6.18 Function: orthonormalize()

```
void orthonormalize(float* ch0tilde, float* ch90tilde, float* twice_inv_noise,
int n, float* n0, float* n90)
```

This function takes as input the (positive frequency parts of the) FFT of a pair of chirp signals. Upon return, the 90° phase chirp has been made orthogonal to the 0° phase chirp, with respect to the inner product defined by $2/S_h$. The normalizations of the chirps are also returned.

The arguments are:

`ch0tilde`: Input. The FFT of the zero-phase chirp T_0 .

`ch90tilde`: Input/Output. The FFT of the 90°-phase chirp T_{90} .

`twice_inv_noise`: Input. Array containing $2/S_h$. The array element `twice_inv_noise[0]` contains the DC value, and the array element `twice_inv_noise[n/2]` contains the value at the Nyquist frequency.

`n`: Input. Defines the length of the arrays: `ch0tilde[0..n-1]`, `ch90tilde[0..n-1]`, and `twice_inv_noise[0..n-1]`.

`n0`: Output. The normalization of the 0-phase chirp.

`n90`: Output. The normalization of the 90°-phase chirp.

Using the notation of (6.14.9) one may define an inner product of the chirps. The normalizations are defined as follows:

$$\frac{1}{n_0^2} \equiv \frac{1}{2}(Q_0, Q_0), \tag{6.18.1}$$

where Q_0 is the optimal filter defined for the zero-phase chirp T_0 . The chirps are orthogonalized internally using the Gram-Schmidt procedure. We first calculate (Q_0, Q_0) and (Q_{90}, Q_0) then define $\epsilon = (Q_{90}, Q_0)/(Q_0, Q_0)$. We then modify the 90°-phase chirp setting $\tilde{T}_{90} \rightarrow T_{90} - \epsilon T_0$. This ensures that the inner product (Q_{90}, Q_0) vanishes. The normalization for this newly-defined chirp is then defined by

$$\frac{1}{n_1^2} \equiv \frac{1}{2}(Q_{90}, Q_{90}). \tag{6.18.2}$$

Author: Bruce Allen, ballen@dirac.phys.uwm.edu

Comments: Notice that the filters Q_0 and Q_{90} are not in general orthogonal except in the adiabatic limit as $S_h(f)$ varies very slowly with changing f . Our approach to this is to construct a slightly-modified ninety-degree phase signal. Note however that this may introduce small errors in the determination of the orbital phase. This should be quantified.

6.19 Dirty details of optimal filtering: wraparound and windowing

To carry out optimal filtering, we need to break the data set (which might be hour, days, or weeks in length) into shorter stretches of N points (which might be seconds or minutes in length). We can understand the effects of “chopping up” the data most easily in the case for which (1) the instrument noise is *white*, so that $S_h(f) = 1$; (2) the source is so close that its signal overwhelms the noise in the IFO, and (3) we are looking for a signal with a given phase (not a linear combination of the two orbital phases).

We want to calculate a signal S as a function of lag t_0 using an FFT.

$$S(t_0) = \int h(t)T(t - t_0)dt \approx S(i_0) = \sum_j h_j T_{j-i_0}, \quad (6.19.1)$$

where we have written both the continuous-time and discrete-time version of the same equation. Using the definition of the discrete Fourier transform, and writing

$$h_j = \sum_{k=0}^{N-1} e^{-2\pi i j k / N} \tilde{h}_k \quad \text{and} \quad T_{j-i_0} = \sum_{k'=0}^{N-1} e^{-2\pi i (j-i_0) k' / N} \tilde{T}_{k'} \quad (6.19.2)$$

one can easily compute that the signal as a function of lag i_0 is

$$S(i_0) = \sum_{j=0}^{N-1} \sum_{k=0}^{N-1} \sum_{k'=0}^{N-1} e^{-2\pi i j k / N} \tilde{h}_k e^{-2\pi i (j-i_0) k' / N} \tilde{T}_{k'} \quad (6.19.3)$$

$$= \sum_{k=0}^{N-1} \sum_{k'=0}^{N-1} N \delta_{k,-k'} e^{2\pi i i_0 k' / N} \tilde{h}_k \tilde{T}_{k'} \quad (6.19.4)$$

$$= \sum_{k=0}^{N-1} N e^{-2\pi i i_0 k / N} \tilde{h}_k \tilde{T}_k^* \quad (6.19.5)$$

Thus, if the data is treated as periodic, and the template is treated as periodic, one can compute the correlation as a function of time using only an FFT. In particular, the use of rectangular windowing does create sidelobes of the template’s frequency components. However it also creates identical sidelobes of the signal’s frequency components - so in effect the correlation in the time domain can be calculated exactly, without any windowing of the signal being necessary.

The only complication arises from the fact that the FFT treats the data as being periodic. Let’s consider some simple examples to illustrate the effects of this. In all of our examples, the number of data points is $N = 65,536 = 2^{16}$ and the (schematic) chirp filter has length $m = 13,500$ and is zero-padded after that time. Please remember, in all the figures that follow, to identify the far right hand side of the graph ($i = 65535$) with the far left hand side ($i = 0$). Figure 28 shows $S(i_0)$ for a schematic chirp which begins at the first data point in the rectangular window. You will notice that the filter output peaks at $i = 0$. If the incoming chirp arrives somewhat later (it starts at $i = 15,000$) as shown in Figure 29 then the filter output peaks at the start time, as shown. A chirp in the signal which starts at the $i = 65,535 - 13,500$ as shown in Figure 30 causes the filter output to peak at $i = 52,035$. Thus, in order to find chirps, we need to find the maxima of the filter output over the interval $i = 0 \dots, N - m$.

Chirp filters can be “stimulated” or “triggered” by events that are not chirps. We will shortly discuss some techniques that can be used to distinguish triggering events that are chirps from those that are simply noise spikes or other transient (but non-chirp) varieties of non-stationary interferometer noise. Suppose that a chirp filter is triggered by some kind of transient event in the IFO output. At what time did this transient event occur? The answer to this question can be seen by examining the impulse response of the “periodic filter” scheme, as shown in the following figures. Thus, by searching for maxima in the filter output over

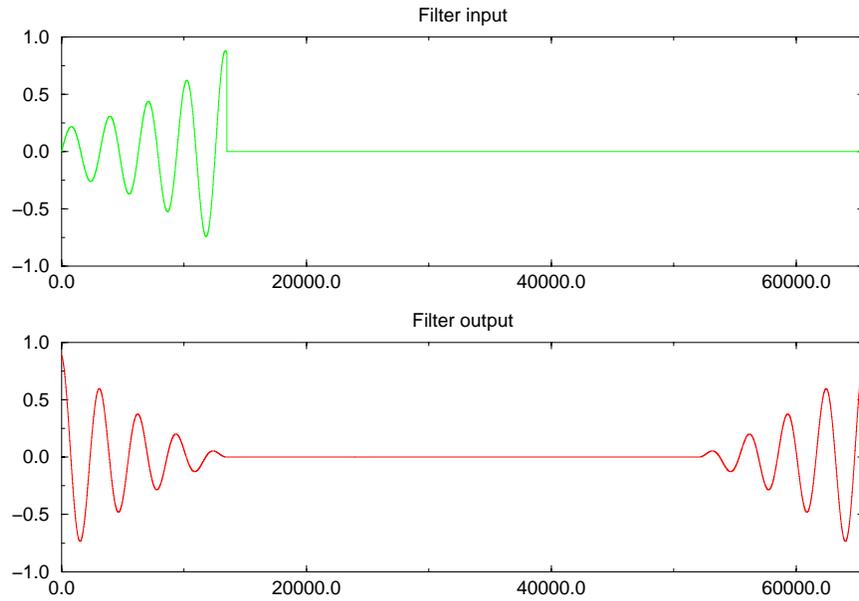


Figure 28: A chirp starting at initial time $i = 0$ and ending at time $i = 13500$ is processed through a chirp filter, whose output peaks at time $i = 0$. Notice that because of wraparound, the (non-causal) filter output begins “earlier” than $i = 0$.

the range $i = 0, \dots, N - m - 1$ we can detect either true chirps in the data stream, starting in the time interval $i = 0, \dots, N - m - 1$ and coalescing (roughly speaking) in the time interval $i = m, \dots, N - 1$, or we can detect transient impulse-like events in the data stream, which take place in the time interval $i = m, \dots, N - 1$. In the GRASP optimal filtering code, after examining the stretch of N data points, we then shift the data points $i = N - m, \dots, N - 1$ into the range $i = 0, \dots, m - 1$ and acquire a new additional set of $N - m$ data points covering remaining (new) time interval.

To indicate the time at which the filter output reached its maximum, several different conventions can be used. First, we can indicate the *peak offset*. This is the offset from the start of the filter output at which the filter output reaches a maximum value. Alternatively, we can use the *impulse offset*. This is the offset at which the filter would have peaked if the maximum were due to a delta-function like impulse at the input. These quantities are defined in equation (13.7.1).

Note that in practice, because the chirp signal has to be convolved with the response function $R(f)$ of the detector, the impulse response of the filter is typically a few points longer than the actual chirp signal. For this reason it is smart to assume that the impulse response of your optimal filter is slightly longer (say a hundred points longer) than the actual time-domain length of the corresponding chirp. This safety margin is set with the `#define SAFETY` statement in the optimal filtering example. You lose a tiny bit of efficiency but reduce the likelihood that boundary effects from the data discontinuity at the start/end of the rectangular window will significantly stimulate the optimal filter output for $i = 0, \dots, N - m - 1$. (See Figs. 31 and 34 to see an illustration of how this windowing discontinuity will corrupt the filter’s output.)

We have demonstrated explicitly that with no windowing (or rather, rectangular windowing) of the data, one can find the appropriate correlation between the signal and a filter exactly: the rectangular window has the same effect on the signal as it does on the template (shifting energy into sidelobes in identical fashion). The only complication was that because of the periodic nature of the FFT one has to be careful about wrap-around errors in relating the output of a filter to the time of occurrence of a signal or impulse.

There is one remaining ugly question. The optimal filter \tilde{Q} depends upon the noise power spectrum

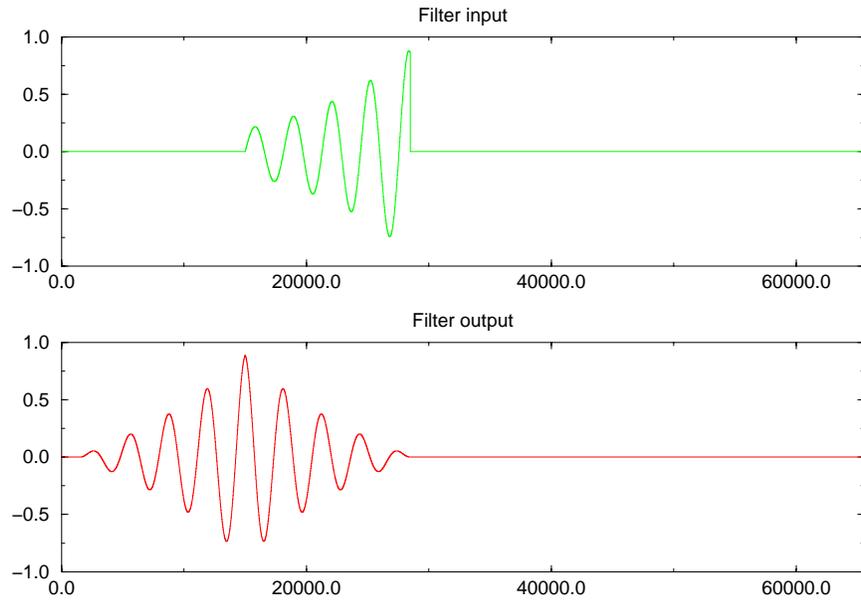


Figure 29: A chirp starting at initial time $i = 15,000$ and ending at time $i = 28,500$ is processed through a chirp filter, whose output peaks at time $i = 15,000$.

of the detector. In real-world filtering, should this noise power spectrum be calculated with windowed, or non-windowed data? We can determine the correlation between signal and template exactly, with only rectangular windowing, because energy in either of these functions is shifted into sidelobes in identical fashion. However a “quiet” part of the IFO spectrum can be corrupted by sidelobes of a nearby noisy region. The effect of this is that the signal get rather less weight from this region of frequency space than it ought, in theory, to receive. This would argue for using only properly-windowed data to find the noise power spectrum to use in determining an optimal filter.

In fact, in our experience, it does not make any difference, at least not when you are searching for binary inspiral chirps. The reason is that the SNR obtained in an optimal filter is only sensitive at second order to errors in the optimal filter function. Thus, the errors due to noise sidelobes which appear if you fail to window the data to calculate an optimal filter are typically not large.

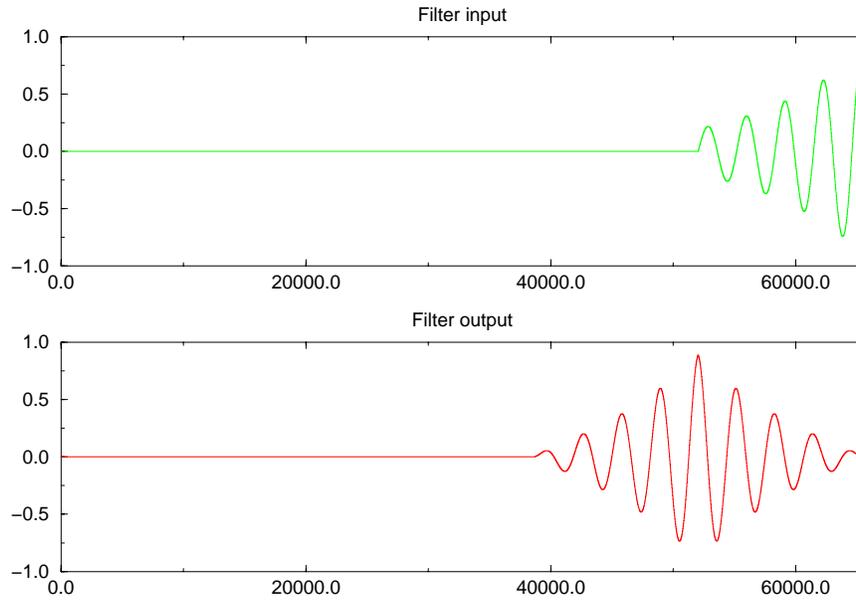


Figure 30: A chirp starting at initial time $i = 52,035$.

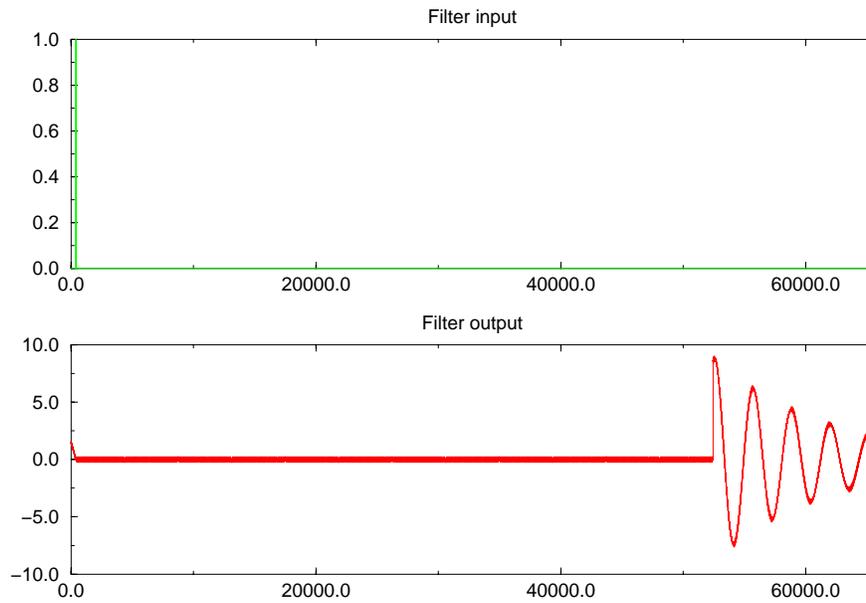


Figure 31: An impulse shortly after $i = 0$.

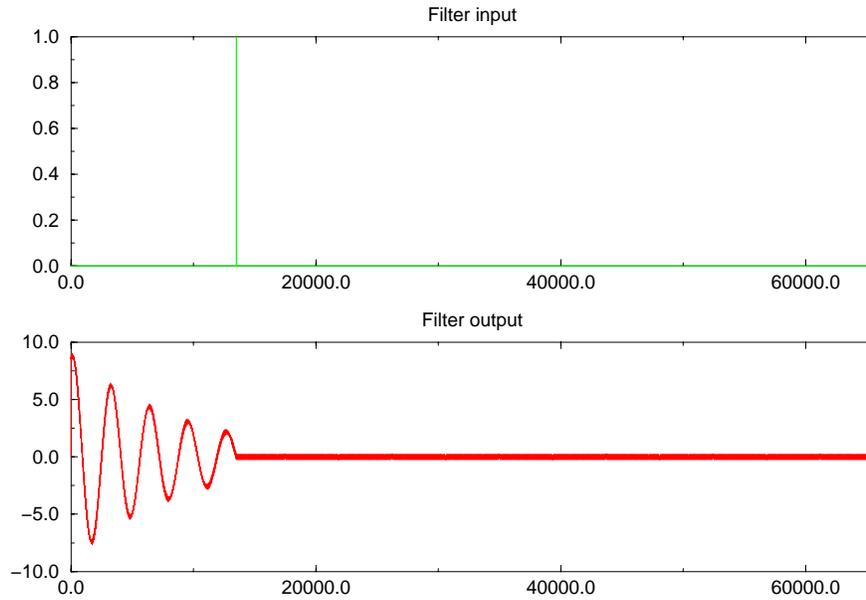


Figure 32: An impulse at $i = 15,000$.

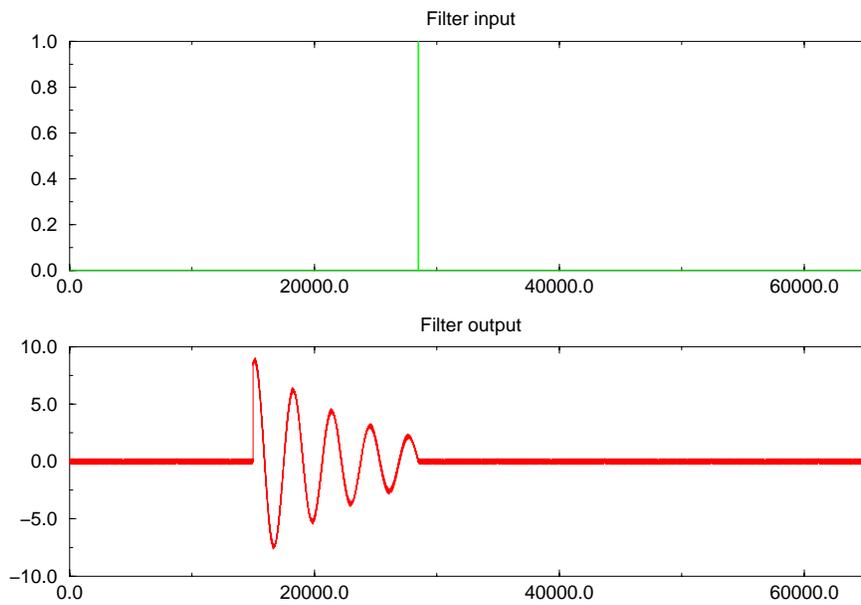


Figure 33: An impulse at $i = 28,500$.

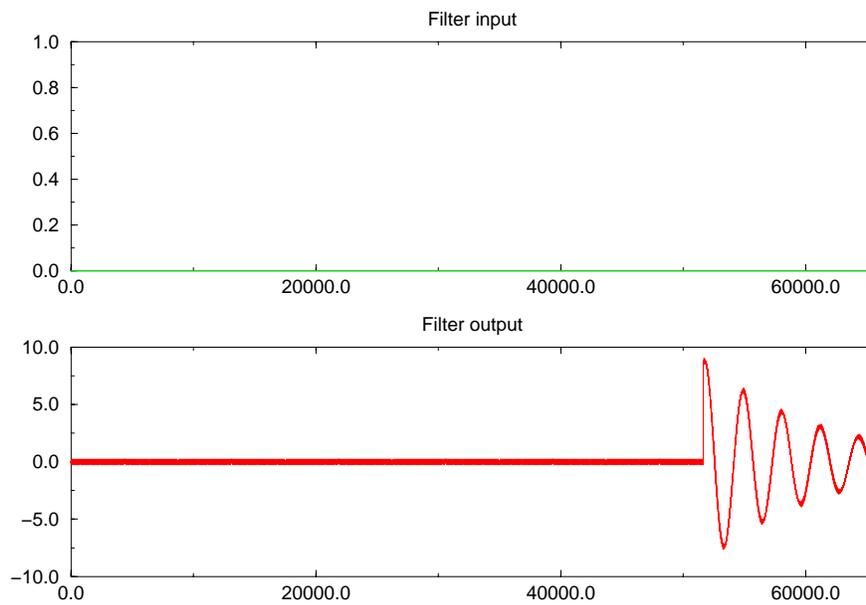


Figure 34: An impulse shortly before $i = 65535$

6.20 Function: find_chirp()

```
void find_chirp(float* htilde, float* ch0tilde, float* ch90tilde, float*
twice_inv_noise, float n0, float n90, float* output0, float* output90, int
n, int chirplen, int* offset, float* snr_max, float* c0, float* c90, float
*var)
```

This routine filters the gravity-wave strain through a pair of optimal filters corresponding to the two phases of a binary chirp, then finds the time at which the SNR peaks.

The arguments are:

`htilde`: Input. The FFT of the gravity-wave strain.

`ch0tilde`: Input. The FFT of the 0-degree chirp.

`ch90tilde`: Input. The FFT of the 90-degree chirp (assumed orthogonal to the 0-degree chirp).

`twice_inv_noise`: Input. Twice the inverse noise power spectrum, used for optimal filtering. The array element `twice_inv_noise[0]` contains the DC value, and the array element `twice_inv_noise[n/2]` contains the value at the Nyquist frequency.

`n0`: Input. Normalization of the 0-degree chirp.

`n90`: Input. Normalization of the 90-degree chirp.

`output0`: Output. A storage array. Upon return, contains the filter output of the 0-degree phase optimal filter.

`output90`: Output. A storage array. Upon return, contains the filter output of the 90-degree phase optimal filter.

`n`: Input. Defines the lengths of the various arrays: `ch0tilde[0..n-1]`, `ch90tilde[0..n-1]`, `output0[0..n-1]`, `output90[0..n-1]`, and `twice_inv_noise[0..n/2]`.

`chirplen`: Input. The number of bins in the time domain occupied by the chirp that you are searching for. This is necessary in order to untangle the wrap-around ambiguity explained earlier.

`offset`: Output. The offset, from 0 to `n-chirplen-1`, at which the signal output (for an arbitrary linear combination of the two filters) peaks.

`snr_max`: Output. The maximum signal-to-noise ratio (SNR) found.

`c0`: Output. The coefficient of the 0-phase template which achieved the highest SNR.

`c90`: Output. The coefficient of the 90°-phase template which achieved the highest SNR. Note that $c_0^2 + c_{90}^2$ should be 1.

`var`: Output. The variance of the filter output. Would be 1 if the input to the filter were colored Gaussian noise with a spectrum defined by S_h .

Author: Bruce Allen, ballen@dirac.phys.uwm.edu

Comments: None.

6.21 Function: freq_inject_chirp()

```
void freq_inject_chirp(float c0, float c90, int offset, float invMpc, float*
ch0tilde, float* ch90tilde, float* htilde, int n)
```

The bottom-line test of any optimal filtering code or searching routines is: can you inject “fake” signals into the data stream, and properly detecting them, while properly rejecting all other signatures of instrumental effects, etc. This routine injects artificial signals into the frequency-domain strain $\tilde{h}(f)$. The plane of the binary system is assumed to be normal to the line to the detector.

The arguments are:

`c0`: Input. The coefficient of the 0-phase template to inject.

`c90`: Input. The coefficient of the 90°-phase to inject. Note that $c_0^2 + c_{90}^2$ should be 1.

`offset`: Input. The offset number of samples at which the injected chirp starts, in the time domain.

`invMpc`: Input. The inverse of the distance to the system (measured in Mpc).

`ch0tilde`: Input. The FFT of the phase-0 chirp (strain units) at a distance of 1 Mpc.

`ch90tilde`: Input. The FFT of the phase-90 chirp (strain units) at a distance of 1 Mpc.

`htilde`: Output. The FFT of the gravity-wave strain. Note that this routine *adds into* and increments this array, so that if it contains another “signal” like IFO noise, the chirp is simply super-posed onto it.

`n`: Input. Defines the lengths of the various arrays `ch0tilde[0..n-1]`, `ch90tilde[0..n-1]`, and `htilde[0..n-1]`.

Note that in making use of this injection routine, you must determine the level of the quantization noise of the ADC, and be careful to inject a properly dithered version of this signal when its amplitude is small compared to the ADC quantization step size.

Author: Bruce Allen, ballen@dirac.phys.uwm.edu

Comments: See the comments for `time_inject_chirp`, particularly with respect to the digital quantization noise.

6.22 Function: time_inject_chirp()

```
void time_inject_chirp(float c0, float c90, int offset, float invMpc, float*
chirp0, float* chirp90, float* data, float *response, float *work, int n)
```

This is a time-domain version of the previous function `freq_inject_chirp()` which injects chirps in the time-domain (after deconvolving them with the detector's response function). This routine injects artificial signals into the time-domain strain $h(t)$. The plane of the binary system is assumed to be normal to the line to the detector.

The arguments are:

`c0`: Input. The coefficient of the 0-phase template to inject.

`c90`: Input. The coefficient of the 90°-phase to inject. Note that $c_0^2 + c_{90}^2$ should be 1.

`offset`: Input. The offset number of samples at which the injected chirp starts, in the time domain.

`invMpc`: Input. The inverse of the distance to the system (measured in Mpc).

`chirp0`: Input. The time-domain phase-0 chirp (strain units) at a distance of 1 Mpc.

`chirp90`: Input. The time-domain phase-90 chirp (strain units) at a distance of 1 Mpc.

`data`: Output. The detector response in time that would be produced by the specified binary inspiral. Note that this routine *adds into* and increments this array, so that if it contains another "signal" like IFO noise, the chirp is simply super-posed onto it.

`response`: Input. The function $R(f)$ that specifies the response function of the IFO. This is produced by the routine `normalize_gw()`.

`work`: Output. A working array.

`n`: Input. Defines the lengths of the various arrays `chirp0[0..n-1]`, `chirp90[0..n-1]`, `data[0..n-1]`, `work[0..n-1]`, and `response[0..n+1]` (note that this "+" sign is *not* a typo!).

Note that in making use of this injection routine, you must determine the level of the quantization noise of the ADC, and be careful to inject a properly dithered version of this signal when its amplitude is small compared to the ADC quantization step size.

Author: Bruce Allen, ballen@dirac.phys.uwm.edu

Comments: A short look at the time-domain signal which is injected shows that it has a low-amplitude spike at the very start. This may be an un-avoidable Gibbs phenomenon associated with the turn-on of the waveform. A second interesting point is that for many interesting signals, the amplitude of the injected signal in the time domain is *below* the level of the quantization noise. Thus, a sensible injection scheme would be to add it into an appropriately dithered (float) version of the integer signal stream, then cast that back into an integer. This should be tried.

6.23 Vetoing techniques (time domain outlier test)

In an ideal world, the output of an interferometer would be a stationary signal described by Gaussian statistics (with very rare superposed binary inspiral chirps and other gravitational-wave signals). This is unfortunately not the case, as can be quickly determined by simply listening to the raw (whitened) interferometer output. Typically the output is a stationary-sounding hiss, interrupted every few minutes by an obvious irregularity in the data stream. These are typically “pops”, “bumps”, “clicks”, “howlers”, “scrapers” and other recognizable categories of noises. In at least some cases, there are “suspects” for these events. For example the pops and bumps might be problems in any of the hundreds of BNC cable connectors used in the instrument.

It is an unfortunate fact that the output of an optimal filter strongly reflects these events. As you have seen in the previous section, a delta-function-like impulse signal in the IFO output can cause a large signal in the optimal filter. And in practice, this happens all of the time - the outputs of optimal chirp filters are frequently triggered by identifiable events in the IFO data stream that are clearly not binary inspiral chirps. Distinguishing these events from real inspiral chirps is called *vetoing*. We have found that two vetoing techniques work particularly well.

The first technique operates in the time domain, and is documented in the routine `is_gaussian()`. The idea is straightforward: if a chirp detector (optimal filter) is triggered, then we look in the data stream for an impulse event that might be responsible. Such events can be found by looking at the statistical distribution of the points in the time domain. If this distribution is significantly non-Gaussian then it indicates that some large transient event caused the filter to trigger, and the event is rejected. In Figures 35 and 36 we show a typical stretch of time-domain raw interferometer output, that does not contain any outlier points. This stretch of raw data “passes” the time-domain outlier test. Figures 37 and 38 show a typical stretch of time-domain raw interferometer output, that does contain outlier points, and “fails” the time-domain outlier test.

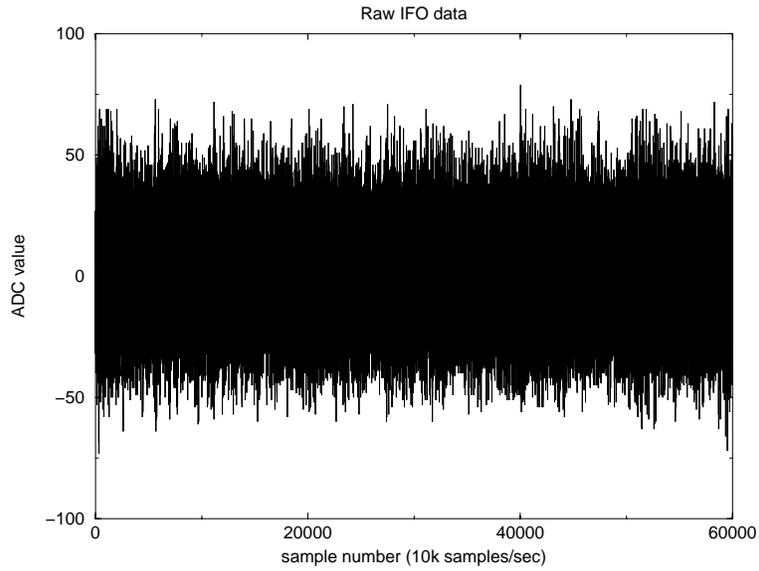


Figure 35: A short stretch of raw IFO data in the time domain, which passes the outlier test.

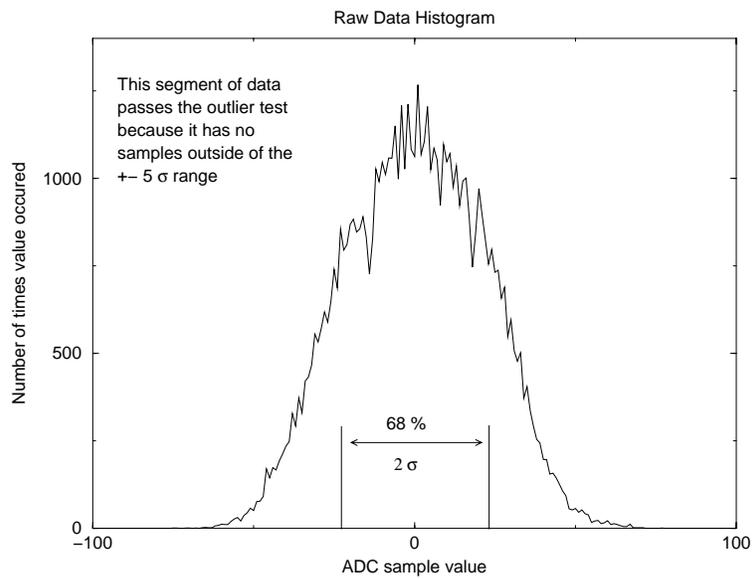


Figure 36: A histogram of this data shows that it has no outlier points.

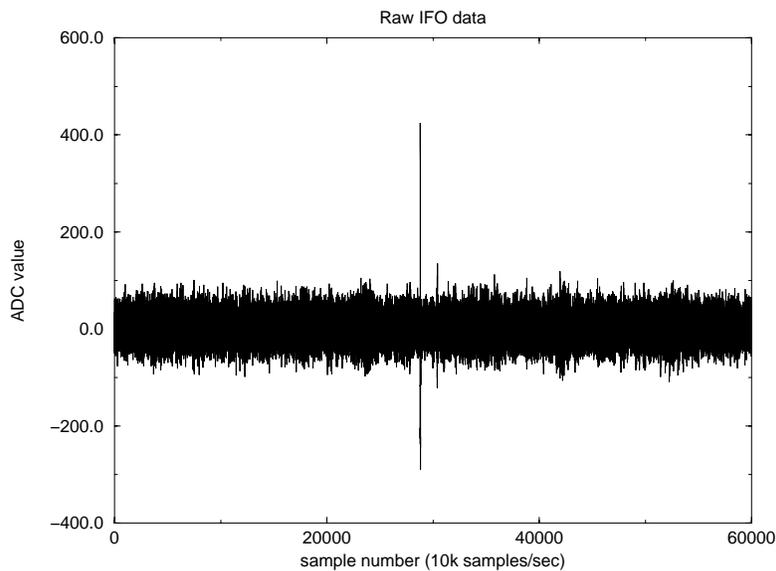


Figure 37: A short stretch of raw IFO data in the time domain, which fails the outlier test.

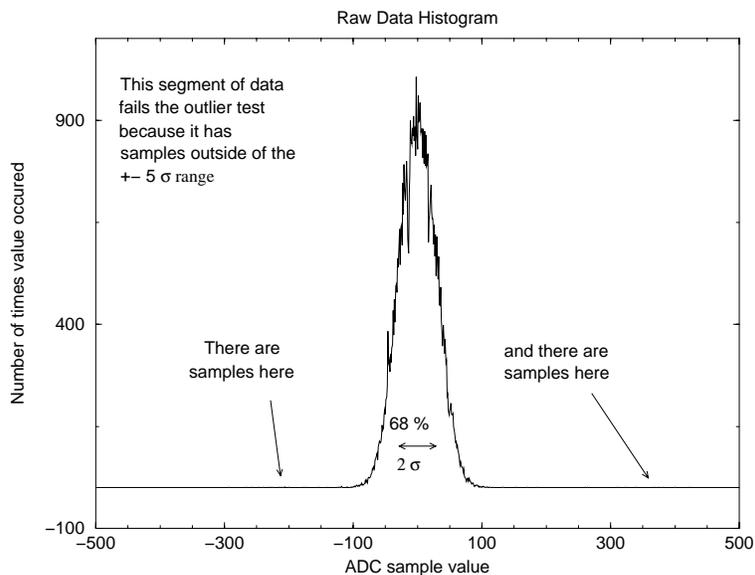


Figure 38: A histogram of this data shows that it has a number of outlier points – which is why it fails to outlier test.

6.24 Vetoing techniques (r^2 time/frequency test)

The second technique vetoing or discrimination test operates in the frequency domain, and is described here. It is a very stringent test, which determines if the hypothetical chirp which has been found in the data stream is consistent with a true binary inspiral chirp summed with Gaussian interferometer noise. If this is true, it should be possible to subtract the (best fit) chirp from the signal, and be left with a signal stream that is consistent with Gaussian IFO noise. One of the nice features of this technique is that it can be statistically characterized in a rigorous way. We follow the same general course as in Section 6.14 on Wiener filtering, first considering the case of a “single phase” phase signal, then considering the case of an “unknown phase” signal.

In the single-phase case, suppose that one of our optimal chirp filters \tilde{Q} is triggered with a large SNR at time t_0 . We suppose that the signal which was responsible for this trigger may be written in either the time or the frequency domain as

$$\begin{aligned} h(t) &= C(t) + n(t) = \alpha T(t - t_0) + n(t) \\ &\iff \\ \tilde{h}(f) &= \alpha \tilde{T}(f) e^{2\pi i f t_0} + \tilde{n}. \end{aligned} \quad (6.24.1)$$

We assume that we have identified what is believed to be the “correct” template T , by the procedure already described of maximizing the SNR over arrival time and template, and have used this to estimate α . We assume that t_0 has been determined exactly (a good approximation since it can be estimated to high accuracy). Our goal is to construct a statistic which will indicate if our estimate of α and identification of T are credible.

We will denote the signal value at time offset t_0 by the real number S :

$$S = 2 \int_{-f_{\text{Ny}}}^{f_{\text{Ny}}} df \frac{\tilde{h}(f) \tilde{T}^*(f)}{S_h(|f|)} e^{-2\pi i f t_0}. \quad (6.24.2)$$

(Here, f_{Ny} denotes the Nyquist frequency, one-half of the sampling rate.) The expected value of S is $\langle S \rangle = \alpha$, although of course since we are discussing a single instance, it’s actual value will in general be different. The chirp template T is normalized so that the expected value $\langle N^2 \rangle = 1$:

$$4 \int_0^{f_{\text{Ny}}} df \frac{|\tilde{T}(f)|^2}{S_h(|f|)} = 1. \quad (6.24.3)$$

We are going to investigate if this signal is “really” due to a chirp by investigating the way in which S gets its contribution from different ranges of frequencies. To do this, break up the integration region in this integral into a set of p disjoint subintervals $\Delta f_1, \dots, \Delta f_p$ whose union is the entire range of frequencies from DC to Nyquist. Here p is a small integer (for example, $p = 8$). This splitup can be performed using the GRASP function `splitup()`. The frequency intervals:

$$\begin{aligned} \Delta f_1 &= \{f \mid 0 < f < f_1\} \\ \Delta f_2 &= \{f \mid f_1 < f < f_2\} \\ &\dots \\ \Delta f_p &= \{f \mid f_{p-1} < f < f_{\text{Ny}}\}, \end{aligned} \quad (6.24.4)$$

are defined by the condition that the *expected signal contributions in each frequency band from a chirp are equal*:

$$4 \int_{\Delta f_i} df \frac{|\tilde{T}(f)|^2}{S_h(|f|)} = \frac{1}{p} \quad (6.24.5)$$

A typical set of frequency intervals is shown in Figure 39.

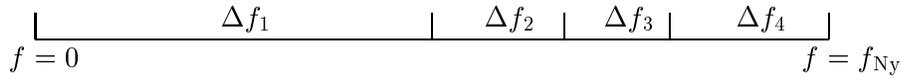


Figure 39: A typical set of frequency intervals Δf_i for the case $p = 4$.

Because the filter is optimal, this also means that the expected noise contributions in each band from the chirp is the same. The frequency subintervals Δf_i are narrow in regions of frequency space where the interferometer is quiet, and are broad in regions where the IFO is noisy.

Now, define a set of p signal values, one for each frequency interval:

$$S_i = 2 \int_{-\Delta f_i \cup \Delta f_i} df \frac{\bar{h}(f)\tilde{T}^*(f)}{S_h(|f|)} e^{-2\pi i f t_0} \quad \text{for } i = 1, \dots, p. \quad (6.24.6)$$

We have included both the positive and negative frequency subintervals to ensure that the S_i are real. If the detector output is Gaussian noise plus a true chirp, the S_i are p normal random variables, with a mean value of $\langle S_i \rangle = \langle S \rangle / p$ and a variance determined by the expected value of the noise-squared:

$$\sigma = \langle S_i^2 \rangle - \langle S_i \rangle^2 = \frac{1}{p}. \quad (6.24.7)$$

From these signal values we can construct a useful time/frequency statistic.

To characterize the statistic, we will need the probability distribution of the S_i . Because each of these values is a sum over different (non-overlapping) frequency bins, they are independent random Gaussian variables with unknown mean values. Their a-priori probability distribution is

$$P(S_1, \dots, S_p) = \prod_{i=1}^p (2\pi\sigma)^{-1/2} e^{-(S_i - \alpha/p)^2 / 2\sigma} \quad (6.24.8)$$

The statistic that we will construct addresses the question, “are the measured values of S_i consistent with the assumption that the measured signal is Gaussian detector noise plus αT ?” One small difficulty is that the value of α that appears in (6.24.8) is not known to us: we only have an *estimate* of its value. To overcome this, we first construct a set of values denoted

$$\Delta S_i \equiv S_i - S/p. \quad (6.24.9)$$

These are *not* independent normal random variables: they are correlated since $\sum_{i=1}^p \Delta S_i$ vanishes. To proceed, we need to calculate the probability distribution of the ΔS_i , which we denote by $\bar{P}(\Delta S_1, \dots, \Delta S_p)$. This quantity is defined by the relation that the integral of any function of p variables $F(y_1, \dots, y_p)$ with respect to the measure defined by this probability distribution satisfies

$$\begin{aligned} \int dy_1 \cdots dy_p \bar{P}(y_1, \dots, y_p) F(y_1, \dots, y_p) \\ = \\ \int dx_1 \cdots dx_p P(x_1, \dots, x_p) F\left(x_1 - \frac{1}{p} \sum_{i=1}^p x_i, \dots, x_p - \frac{1}{p} \sum_{i=1}^p x_i\right). \end{aligned} \quad (6.24.10)$$

[Note that in this expression and the following ones, all integrals are from $-\infty$ to ∞ .] This may be used to find a closed form for \bar{P} : let $F(y_1, \dots, y_p) = \delta(y_1 - \Delta S_1) \cdots \delta(y_p - \Delta S_p)$. This gives

$$\bar{P}(\Delta S_1, \dots, \Delta S_p) = \int \prod_{i=1}^p dx_i (2\pi\sigma)^{-1/2} e^{-(x_i - \alpha/p)^2 / 2\sigma} \delta\left(x_i - \Delta S_i - \frac{1}{p} \sum_{j=1}^p x_j\right). \quad (6.24.11)$$

To evaluate the integral, change variables from (x_1, \dots, x_p) to (z_1, \dots, z_{p-1}, W) defined by

$$\begin{aligned} W &= x_1 + \dots + x_p \\ z_1 &= x_1 - W/p \\ &\dots \\ z_{p-1} &= x_{p-1} - W/p \end{aligned} \quad (6.24.12)$$

which can be inverted to yield

$$\begin{aligned} x_1 &= z_1 + W/p \\ &\dots \\ x_{p-1} &= z_{p-1} + W/p \\ x_p &= W/p - z_1 - \dots - z_{p-1} \end{aligned} \quad (6.24.13)$$

The Jacobian of this coordinate transformation is:

$$J = \det \left[\frac{\partial(x_1, \dots, x_p)}{\partial(z_1, \dots, z_{p-1}, W)} \right] = \det \begin{bmatrix} 1 & 0 & \dots & 0 & 1/p \\ 0 & 1 & \dots & 0 & 1/p \\ & & \dots & & \\ 0 & 0 & \dots & 1 & 1/p \\ -1 & -1 & \dots & -1 & 1/p \end{bmatrix} \quad (6.24.14)$$

Using the linearity in rows of the determinant, it is straightforward to show that $J = 1$.

The integral may now be written as

$$\begin{aligned} \bar{P}(\Delta S_1, \dots, \Delta S_p) &= \int dz_1 \dots dz_{p-1} dW (2\pi\sigma)^{-p/2} e^{-[(x_1 - \alpha/p)^2 + \dots + (x_p - \alpha/p)^2]/2\sigma} \\ &\times \delta(z_1 - \Delta S_1) \dots \delta(z_{p-1} - \Delta S_{p-1}) \delta(z_1 + \dots + z_{p-1} + \Delta S_p). \end{aligned} \quad (6.24.15)$$

A few moments of algebra shows that the exponent may be expressed in terms of the new integration variables as

$$(x_1 - \alpha/p)^2 + \dots + (x_p - \alpha/p)^2 = z_1^2 + \dots + z_{p-1}^2 + (W - \alpha)^2/p + (z_1 + \dots + z_{p-1})^2 \quad (6.24.16)$$

and thus the integral yields

$$\begin{aligned} \bar{P}(\Delta S_1, \dots, \Delta S_p) &= \int dW (2\pi\sigma)^{-p/2} e^{-[\Delta S_1^2 + \dots + \Delta S_p^2 + (W - \alpha)^2/p]/2\sigma} \delta(\Delta S_1 + \dots + \Delta S_p) \\ &= (2\pi\sigma)^{-p/2} (2\pi\sigma p)^{1/2} e^{-[\Delta S_1^2 + \dots + \Delta S_p^2]/2\sigma} \delta(\Delta S_1 + \dots + \Delta S_p) \end{aligned} \quad (6.24.17)$$

This probability distribution arises because we do not know the true mean value of S which is $\alpha = \langle S \rangle$ but can only estimate it using the actual measured value of S . Similar problems arise whenever the mean of a distribution is not known but must be estimated (problem 14-7 of [24]). This probability distribution is “as close as you can get to Gaussian” subject to the constraint that the sum of the ΔS_i must vanish. It is significant that this probability density function is completely independent of α , which means that the properties of the ΔS_i do not depend upon whether a signal is present or not.

The individual ΔS_i have identical mean and variance, which may be easily calculated from the probability distribution function (6.24.10). For example the mean is zero: $\langle \Delta S_i \rangle = 0$. To calculate the variance, let $F(y_1, \dots, y_p) = y_1^2$ in (6.24.10). One finds

$$\langle \Delta S_i^2 \rangle = \frac{1}{p} \left(1 - \frac{1}{p} \right) \quad (6.24.18)$$

Now that we have calculated the probability distribution of the ΔS_i it is straightforward to construct and characterize a χ^2 -like statistic which we will call r^2 .

Define the statistic

$$r^2 = \sum_{i=1}^p (\Delta S_i)^2. \quad (6.24.19)$$

From (6.24.18) it is obvious that for Gaussian noise plus a chirp the statistical properties of r^2 are *independent of α* : it has the same statistical properties if a chirp signal is present or not. For this reason, the value of r^2 provides a powerful method of testing the hypothesis that the detector's output is Gaussian noise plus a chirp. If the detector's output is of this form, then the value of r^2 is unlikely to be much larger than its expected value (this statement is quantified below). On the other hand, if the filter was triggered by a spurious noise event that does *not* have the correct time/frequency distribution, then r^2 will typically have a value that is *very* different than the value that it has for Gaussian noise alone (or equivalently, for Gaussian noise plus a chirp).

The expected value of r^2 is trivial to calculate

$$\langle r^2 \rangle = p \langle \Delta S_i^2 \rangle = 1 - \frac{1}{p} \quad (6.24.20)$$

One can also easily compute the probability distribution of r using (6.24.17). The probability that $r > R$ in the presence of a true chirp signal is the integral of (6.24.17) over the region $r > R$. In the p -dimensional space, the integrand vanishes except on a $p - 1$ -plane, where it is spherically-symmetric. To evaluate the integral, introduce a new set of orthonormal coordinates (u_1, \dots, u_p) obtained from any orthogonal transformation on $(\Delta S_1, \dots, \Delta S_p)$ for which the new p 'th coordinate is orthogonal to the hyperplane $\Delta S_1 + \dots + \Delta S_p = 0$. Hence $u_p = p^{-1/2} [\Delta S_1 + \dots + \Delta S_p]$. Our statistic r^2 is also the squared radius $r^2 = u_1^2 + \dots + u_p^2$ in these coordinates. Hence

$$\begin{aligned} P(r > R) &= \int_{r^2 > R^2} \bar{P} \\ &= (2\pi\sigma)^{-p/2} (2\pi\sigma p)^{1/2} \int_{r^2 > R^2} e^{-r^2/2\sigma} \delta(\sqrt{p}u_p) du_1 \cdots du_p. \end{aligned} \quad (6.24.21)$$

It's now easy to do the integral over the coordinate u_p , and having done this, we are left with a spherically-symmetric integral over R^{p-1} :

$$\begin{aligned} P(r > R) &= (2\pi\sigma)^{(1-p)/2} \int_{r^2 > R^2} e^{-r^2/2\sigma} du_1 \cdots du_{p-1} \\ &= \Omega_{p-2} \int_R^\infty r^{p-2} e^{-r^2/2\sigma} dr \\ &= \frac{1}{\Gamma(\frac{p-1}{2})} \int_{R^2/2\sigma}^\infty x^{(p-3)/2} e^{-x} dx \\ &= Q\left(\frac{p-1}{2}, \frac{R^2}{2\sigma}\right), \end{aligned} \quad (6.24.22)$$

where $\Omega_n = \frac{2\pi^{(n+1)/2}}{\Gamma(\frac{n+1}{2})}$ is the n -volume of a unit-radius n -sphere S^n . The incomplete gamma function Q is the same function that describes the likelihood function in the traditional χ^2 test [the *Numerical Recipes* function `gammapq(a, x)`]. Figure 40 show a graph of $P(r > R)$ for some different values of the parameter p . The appearance of $p - 1$ in these expressions reflects the fact that although r^2 is a sum of the squares of p Gaussian variables, these variables are subject to a single constraint (their sum vanishes) and thus the number of degrees of freedom is $p - 1$.

In practice (based on CIT 40-meter data) breaking up the frequency range into $p = 8$ intervals provides a very reliable veto for rejecting events that trigger an optimal filter, but which are not themselves chirps. The value of $Q(3.5, 10.0) = 0.0056 \dots$ so if $r^2 > 2.5$ then one can conclude that the likelihood that a given trigger is actually due to a chirp is less than 0.6%; rejecting or vetoing such events will only reduce the “true event” rate by 0.6%. However in practice it eliminates almost all other events that trigger an optimal filter; a noisy event that stimulates a binary chirp filter typically has $r^2 \approx 100$ or larger!

The previous analysis for the “single-phase” case assumes that we have found the correct template T describing the signal. In searching for a binary inspiral chirp however, the signal is a linear combination of the two different possible phases:

$$\begin{aligned} h(t) &= C(t) + n(t) = \alpha T_0(t - t_0) + \beta T_{90}(t - t_0) + n(t) \\ &\iff \\ \tilde{h}(f) &= \left[\alpha \tilde{T}_0(f) + \beta \tilde{T}_{90}(f) \right] e^{2\pi i f t_0} + \tilde{n}. \end{aligned} \quad (6.24.23)$$

and the amplitudes α and β are unknown. The reader might well wonder why we can’t simply construct a single properly normalized template as

$$T = \left(\frac{\alpha}{\sqrt{\alpha^2 + \beta^2}} \right) T_0 + \left(\frac{\beta}{\sqrt{\alpha^2 + \beta^2}} \right) T_{90} \quad (6.24.24)$$

and then use the previously-described “single phase” method. In principle, this would work properly. The problem is that *we do not know the correct values of α and β* . Since $\alpha = \text{Re} \langle S(t_0) \rangle$ and $\beta = \text{Im} \langle S(t_0) \rangle$, we can *estimate* the values of α and β from the real and imaginary parts of the measured signal, however these estimates will not give the true values. For this reason, an r^2 statistic and test can be constructed for the “two-phase” case, but it has twice the number of degrees of freedom as the “single-phase” case.

The description and characterization of the r^2 test for the two phase case is similar to the single-phase case. For the two phase case, the signal is a complex number

$$S = 2 \int_{-f_{Ny}}^{f_{Ny}} df \frac{\tilde{h}(f) \left[\tilde{T}_0^*(f) + i \tilde{T}_{90}^*(f) \right]}{S_h(|f|)} e^{-2\pi i f t_0}. \quad (6.24.25)$$

The templates for the individual phases are normalized as before:

$$4 \int_0^{f_{Ny}} df \frac{|\tilde{T}_0(f)|^2}{S_h(|f|)} = 4 \int_0^{f_{Ny}} df \frac{|\tilde{T}_{90}(f)|^2}{S_h(|f|)} = 1 \text{ and } \int_0^{f_{Ny}} df \frac{\tilde{T}_0(f) \tilde{T}_{90}^*(f)}{S_h(|f|)} = 0. \quad (6.24.26)$$

This assume the same adiabatic limit discussed earlier: $\dot{f}/f \ll f$. In this limit, the frequency intervals Δf_i are identical for either template. We define signal values in each frequency band in the same way as before, except now these are complex:

$$S_i = 2 \int_{-\Delta f_i \cup \Delta f_i} df \frac{\tilde{h}(f) \left[\tilde{T}_0^*(f) + i \tilde{T}_{90}^*(f) \right]}{S_h(|f|)} e^{-2\pi i f t_0} \quad \text{for } i = 1, \dots, p. \quad (6.24.27)$$

The mean value of the signal in each frequency band is

$$\langle S_i \rangle = \langle S \rangle / p = (\alpha + i\beta) / p, \quad (6.24.28)$$

and the variance of either the real or imaginary part is $\sigma = 1/p$ as before, so that the total variance is twice as large as in the single phase case:

$$\langle |S_i|^2 \rangle - |\langle S_i \rangle|^2 = \frac{2}{p}. \quad (6.24.29)$$

The signal values are now characterized by the probability distribution

$$P(S_1, \dots, S_p) = \prod_{i=1}^p (2\pi\sigma)^{-1} e^{-|S_i - \alpha/p - i\beta/p|^2 / 2\sigma}. \quad (6.24.30)$$

Note that the arguments of this function are *complex*; for this reason the overall normalization factors have changed from the single-phase case. We now construct complex quantities which are the difference between the actual signal measured in a frequency band and the expected value for our templates and phases:

$$\Delta S_i \equiv S_i - S/p. \quad (6.24.31)$$

The probability distribution of these differences is still defined by (6.24.10) but in that expression, the variables of integration x_i and y_i are integrated over the complex plane (real and imaginary parts from $-\infty$ to ∞), and F is any function of p complex variables. As before, we can calculate \bar{P} by choosing F correctly, in this case as $F(y_1, \dots, y_p) = \delta^2(y_1 - \Delta S_1) \cdots \delta^2(y_p - \Delta S_p)$, where $\delta^2(z) \equiv \delta(\text{Re } z)\delta(\text{Im } z)$. The same procedure as before then yields the probability distribution function

$$\bar{P}(\Delta S_1, \dots, \Delta S_p) = (2\pi\sigma)^{-p} (2\pi\sigma p) e^{-[|\Delta S_1|^2 + \dots + |\Delta S_p|^2] / 2\sigma} \delta^2(\Delta S_1 + \dots + \Delta S_p) \quad (6.24.32)$$

It is now easy to see that the expectation of the signal differences is still zero $\langle \Delta S_i \rangle = 0$ but the variances are twice as large as in the single-phase case:

$$\langle |\Delta S_i|^2 \rangle = \frac{2}{p} \left(1 - \frac{1}{p} \right). \quad (6.24.33)$$

The r^2 statistic is now defined by

$$r^2 = \sum_{i=1}^p |\Delta S_i|^2. \quad (6.24.34)$$

and has an expectation value which is twice as large as in the single-phase case:

$$\langle r^2 \rangle = p \langle |\Delta S_i|^2 \rangle = 2 - \frac{2}{p}. \quad (6.24.35)$$

The calculation of the distribution function of r^2 is similar to the single phase case (but with twice the number of degrees of freedom) and gives the incomplete Γ -function

$$\begin{aligned} P(r > R) &= (2\pi\sigma)^{-p} (2\pi\sigma p) \int_{r^2 > R^2} e^{-r^2/2\sigma} \delta^2(\sqrt{p}u_p) du_1 \cdots du_p \\ &= (2\pi\sigma)^{(1-p)} \int_{r^2 > R^2} e^{-r^2/2\sigma} du_1 \cdots du_{p-1} \\ &= P(r > R) = Q(p-1, \frac{R^2}{2\sigma}) = Q(p-1, \frac{pR^2}{2}) \end{aligned} \quad (6.24.36)$$

This is precisely the distribution of a χ^2 statistic with $2p-2$ degrees of freedom: each of the p variables ΔS_i has 2 degrees of freedom, and there are two constraints since the sum of both the real and imaginary parts vanishes. In fact since the expectation value of the χ^2 statistic is just the number of degrees of freedom:

$$\langle \chi^2 \rangle = 2p - 2 \quad (6.24.37)$$

the relationship between the r^2 and χ^2 statistic may be obtained by comparing equations (6.24.37) and (6.24.35), giving

$$\chi^2 = pr^2. \quad (6.24.38)$$

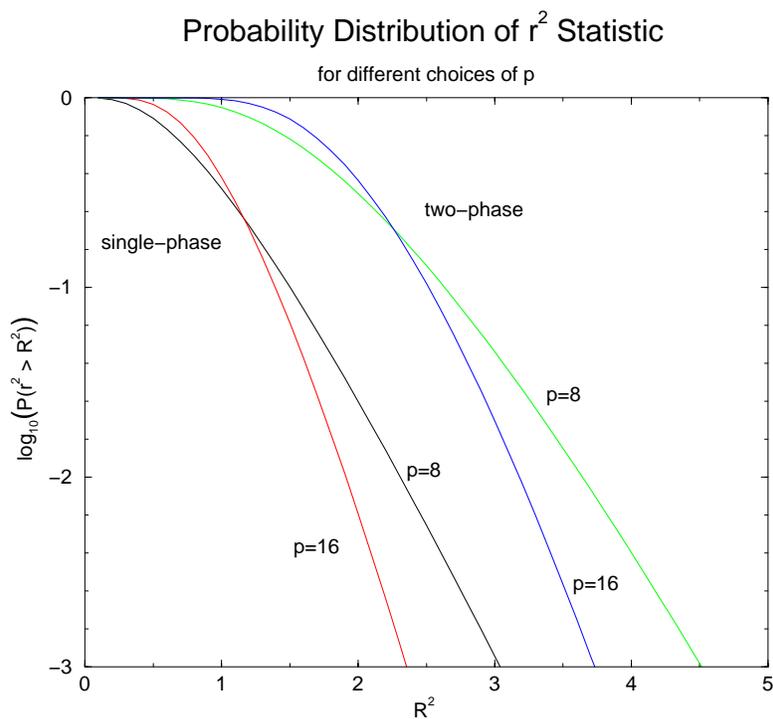


Figure 40: The probability that the r^2 statistic exceeds a given threshold R^2 is shown for both the single-phase and two-phase test, for $p = 8$ and $p = 16$ frequency ranges. For example, for the single-phase $p = 8$ test, the probability that $r^2 > 2.31$ is 1% for a chirp plus Gaussian noise. For the single-phase test with $p = 16$ the probability of exceeding the same threshold is about 10^{-3} .

6.25 How does the r^2 test work ?

In Section 6.24 we have derived the statistical properties of the r^2 test, and described it in mathematical terms. This is a bit deceptive, because this test was actually developed based on some simple physical intuition. We noticed with experience that many of the high SNR events that were not found by the outlier `is_gaussian()` test did not sound anything like chirps (when listened to with the `audio()` and `sound()` functions). It was clear from just listening that for these spurious signals did not have the low frequency signal arriving first, followed by the high frequency signal arriving last, in the same way as a chirp signal. So in fact the r^2 test was designed to discriminate the way in which the different frequencies arrived with time. In effect, the filter used to construct the signal S_1 passes only the lowest frequencies, the filter used to construct the signal S_2 passes the next-to-lowest frequencies, and so on. The filter which produces the signal S_p passes the highest range of frequencies which would make a significant contribution (i.e. a fraction $1/p$) of the SNR for a true chirp.

If the signal is a true chirp, then the outputs of each of these different filters (the $S_i(t_0)$) may be thought of as functions of lag t_0 all peak at the same time-offset t_0 , the *same* time-offset that maximizes the total signal $S(t_0)$. This is illustrated in Figure 41. It is also instructive to compare the values of the filter outputs

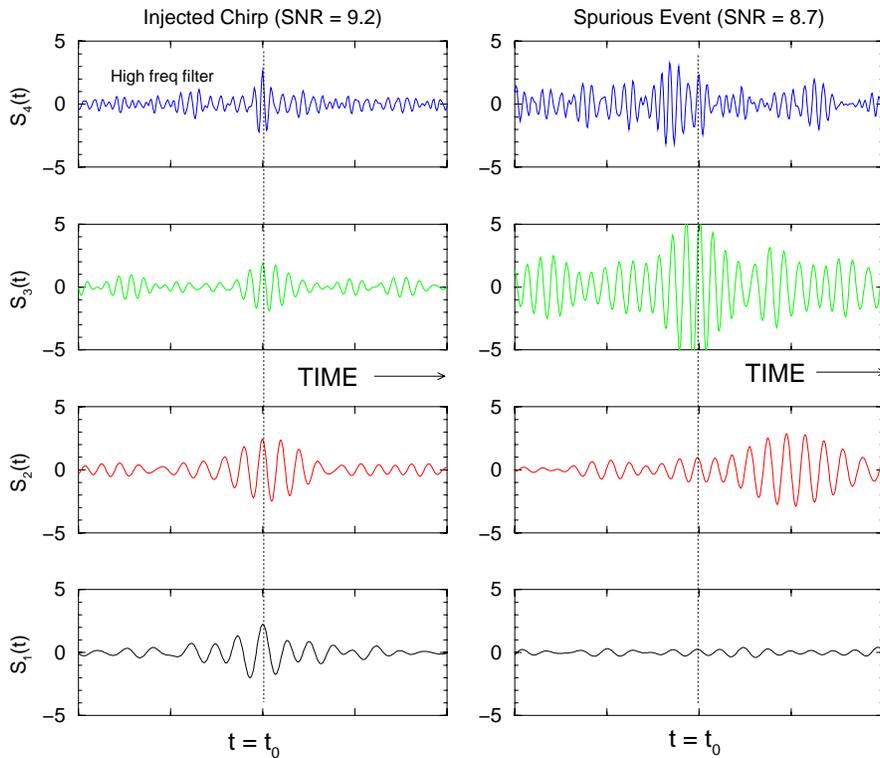


Figure 41: This figure shows the output of four single-phase filters for the $p = 4$ case, for a “true chirp” injected into a stream of real IFO data (left set of figures) and a transient noise burst already present in another stream of real IFO data (right set of figures). When a true chirp is present, the filters in the different frequency bands all peak at the same time offset t_0 : the time offset which maximizes the SNR. At this instant in time, all of the S_i are about the same value. However when the filter was triggered by a non-chirp signal, the filters in the different frequency bands peak at different times, and in fact at time t_0 they have very different values (some large, some small, and so on).

(single-phase test) for the two cases shown in Figure 41. For the injected chirp, the signal-to-noise ratio was

9.2, and the signal values in the different bands were

$$\begin{aligned} S_1 &= 2.25 \\ S_2 &= 2.44 \\ S_3 &= 1.87 \\ S_4 &= 2.64 \\ S &= S_1 + S_2 + S_3 + S_4 = 9.2 \\ r^2 &= \sum_{i=1}^4 (S/4 - S_i)^2 = 0.324 \\ P &= Q(3/2, 2r^2) = 0.730, \end{aligned} \tag{6.25.1}$$

so there is a large probability P of having r^2 this large.

For the spurious noise event shown in Figure 41 the SNR was quite similar (8.97) but the value of r^2 is very different:

$$\begin{aligned} S_1 &= 0.23 \\ S_2 &= 0.84 \\ S_3 &= 5.57 \\ S_4 &= 2.33 \\ S &= S_1 + S_2 + S_3 + S_4 = 8.97 \\ r^2 &= \sum_{i=1}^4 (S/4 - S_i)^2 = 17.1 \\ P &= Q(3/2, 2r^2) = 9.4 \times 10^{-15}, \end{aligned} \tag{6.25.2}$$

so the probability that this value of r^2 would be obtained for a chirp plus Gaussian noise is extremely small.

6.26 Function: splitup()

```
void splitup(float *working, float template, float *r, int n, float total,  
int p, int *indices)
```

This routine takes as inputs a template and a noise-power spectrum, and splits up the frequency spectrum into a set of sub-intervals to use with the vetoing technique just described.

The arguments are:

working: Input. An array `working[0..n-1]` used for working space.

template: Input. The array `template[0..n-1]` contains the positive frequency ($f \geq 0$) part of the complex function $\tilde{T}(f)$. The packing of \tilde{T} into this array follows the scheme used by the *Numerical Recipes* routine `realft()`, which is described between equations (12.3.5) and (12.3.6) of [1]. The DC component $\tilde{T}(0)$ is real, and located in `template[0]`. The Nyquist-frequency component $\tilde{T}(f_{\text{Nyquist}})$ is also real, and is located in `template[1]`. The array elements `template[2]` and `template[3]` contain the real and imaginary parts, respectively, of $\tilde{T}(\Delta f)$ where $\Delta f = 2f_{\text{Nyquist}}/n = (n\Delta t)^{-1}$. Array elements `template[2j]` and `template[2j+1]` contain the real and imaginary parts of $\tilde{T}(j \Delta f)$ for $j = 1, \dots, n/2 - 1$.

r: Input. The array `r[0..n/2]` contains the values of the real function \tilde{r} which is twice the inverse of the receiver noise, as in equation (6.14.15), so that $\tilde{r}(f) = 2/\tilde{S}_h(|f|)$. The array elements are arranged in order of increasing frequency, from the DC value at subscript 0, to the Nyquist frequency at subscript $n/2$. Thus, the j 'th array element `r[j]` contains the real value $\tilde{r}(j \Delta f)$, for $j = 0, 1, \dots, n/2$. Again it is assumed that $\tilde{r}(-f) = \tilde{r}^*(f) = \tilde{r}(f)$.

n: Input. The total length of the complex arrays `template` and `working`, and the number of points in the output array `s`. Note that the array `r` contains $n/2 + 1$ points. n must be even.

total: Input. This is the total value of the integrated template squared over S_h ; the frequency subintervals are chosen so that each of the p subintervals contains $1/p$ of this total.

p: Input. The number of frequency bands into which you want to divide the range from DC to f_{Nyquist} .

indices: Output. The frequency bins of the first frequency band are `i=0..indices[0]`. The next frequency band is `i=indices[0]+1..indices[1]`. The p 'th frequency band is `i=indices[p-2]+1..indices[p-1]`. Note that `indices[p-1]=n-1`.

Author: Bruce Allen, ballen@dirac.phys.uwm.edu

Comments: None.

6.27 Function: splitup_freq()

```
float splitup_freq(float c0, float c90, float *chirp0, float *chirp90, float
norm, float* twice_inv_noise, int n, int offset, int p, int* indices, float*
stats, float* working, float* htilde)
```

This routine returns the value of the statistic $r^2 = \sum_{i=1}^p (\Delta S_i)^2$. This is a less-efficient version, which internally constructs filters for each of the different frequency subintervals, and then filters the metric perturbation through those filters. It is useful to understand how the different frequency components behave in the time domain, after filtering.

The arguments are:

`c0`: Input. The coefficient of the 0-phase template.

`c90`: Input. The coefficient of the 90°-phase template. Note that $c_0^2 + c_{90}^2$ should be 1.

`chirp0`: Input. An array `chirp0[0..n-1]` containing the FFT of the 0-phase chirp.

`chirp90`: Input. An array `chirp90[0..n-1]` containing the FFT of the 90°-phase chirp.

`norm`: Input. The normalization of the 0-phase chirp.

`twice_inv_noise`: Input. The array `twice_inv_noise[0..n/2]` contains $2/S_h(f)$, as described previously. The array element `twice_inv_noise[0]` contains the DC value, and the array element `twice_inv_noise[n/2]` contains the value at the Nyquist frequency.

`n`: Input. Defines the lengths of the previous arrays.

`offset`: Input. The offset of the moment of maximum signal in the filter output.

`p`: Input. The number of frequency bands p for the vetoing test.

`indices`: Output. An array `indices[0..p-1]` used for internal storage of the frequency subintervals (see `splitup()`).

`stats`: Output. An array `stats[0..p-1]` containing the values of the S_i for $i = 1, \dots, p$.

`working`: Output. An array `working[0..n-1]` used for internal storage.

`htilde`: Input. An array `htilde[0..n-1]` containing the positive frequency part of $\tilde{h}(f)$.

Author: Bruce Allen, ballen@dirac.phys.uwm.edu

Comments: None.

6.28 Function: splitup_freq2()

```
float splitup_freq2(float c0, float c90, float *chirp0, float *chirp90, float
norm, float* twice_inv_noise, int n, int offset, int p, int* indices, float*
stats, float* working, float* htilde)
```

This routine returns the value of the statistic $r^2 = \sum_{i=1}^p (\Delta S_i)^2$. This is a more computationally-efficient version, which does not filter \tilde{h} through each of the p independent time domain filters. The arguments are identical to those of `splitup_freq()`.

The arguments are:

`c0`: Input. The coefficient of the 0-phase template.

`c90`: Input. The coefficient of the 90°-phase template. Note that $c_0^2 + c_{90}^2$ should be 1.

`chirp0`: Input. An array `chirp0[0..n-1]` containing the FFT of the 0-phase chirp.

`chirp90`: Input. An array `chirp90[0..n-1]` containing the FFT of the 90°-phase chirp.

`norm`: Input. The normalization of the 0-phase chirp.

`twice_inv_noise`: Input. The array `twice_inv_noise[0..n/2]` contains $2/S_h(f)$, as described previously. The array element `twice_inv_noise[0]` contains the DC value, and the array element `twice_inv_noise[n/2]` contains the value at the Nyquist frequency.

`n`: Input. Defines the lengths of the previous arrays.

`offset`: Input. The offset of the moment of maximum signal in the filter output.

`p`: Input. The number of frequency bands p for the vetoing test.

`indices`: Output. An array `indices[0..p-1]` used for internal storage of the frequency subintervals (see `splitup()`).

`stats`: Output. An array `stats[0..p-1]` containing the values of the S_i for $i = 1, \dots, p$.

`working`: Output. An array `working[0..n-1]` used for internal storage.

`htilde`: Input. An array `htilde[0..n-1]` containing the positive frequency part of $\tilde{h}(f)$.

Author: Bruce Allen, ballen@dirac.phys.uwm.edu

Comments: None.

6.29 Function: `splitup_freq3()`

```
float splitup_freq2(float c0, float c90, float *chirp0, float *chirp90, float
norm, float* twice_inv_noise, int n, int offset, int p, int* indices, float*
stats, float* working, float* htilde)
```

This routine implements the two-phase r^2 statistic test. It returns the value of the statistic $r^2 = \sum_{i=1}^p |\Delta S_i|^2$ as defined in Eq. (6.24.34). It is algorithmically similar to `splitup_freq2()`, except that it allows for the case where the phase of the signal is unknown. The arguments are identical to those of `splitup_freq2()` but the array `stats` has $2p$ elements since the signals are complex.

Note: The GRASP library includes two additional functions which are operationally identical to `splitup_freq3()`, called `splitup_freq4()` and `splitup_freq5()`. The last of these is currently the most efficient implementation of the two-phase r^2 test. All the arguments of `splitup_freq[3-5]()` are *identical*.

The arguments are:

`c0`: Input. Used in the same way as in `splitup_freq2()`.

`c90`: Input. Used in the same way as in `splitup_freq2()`. Note that if templates have unit norm you can set $c_0^2 + c_1^2 = 4$.

`chirp0`: Input. An array `chirp0[0..n-1]` containing the FFT of the 0-phase chirp.

`chirp90`: Input. An array `chirp90[0..n-1]` containing the FFT of the 90°-phase chirp.

`norm`: Input. The normalization of the 0-phase chirp.

`twice_inv_noise`: Input. The array `twice_inv_noise[0..n/2]` contains $2/S_h(f)$, as described previously. The array element `twice_inv_noise[0]` contains the DC value, and the array element `twice_inv_noise[n/2]` contains the value at the Nyquist frequency.

`n`: Input. Defines the lengths of the previous arrays.

`offset`: Input. The offset of the moment of maximum signal in the filter output.

`p`: Input. The number of frequency bands p for the vetoing test.

`indices`: Output. An array `indices[0..p-1]` used for internal storage of the frequency subintervals (see `splitup()`).

`stats`: Output. An array `stats[0..2p-1]` containing the real and imaginary parts of the S_i for $i = 1, \dots, p$.

`working`: Output. An array `working[0..n-1]` used for internal storage.

`htilde`: Input. An array `htilde[0..n-1]` containing the positive frequency part of $\tilde{h}(f)$.

Authors Bruce Allen, ballen@dirac.phys.uwm.edu, and Patrick Brady, patrick@tapir.caltech.edu, and Jolien Creighton jolien@tapir.caltech.edu.

Comments: None.

6.30 Example: optimal program

This program reads the 40-meter data stream, and then filters it through a chirp template corresponding to a pair of inspiraling $1.4M_{\odot}$ neutron stars.

The correspondence between different arrays in this program, and the quantities discussed previously in this section, is given below. In these equations, $\Delta t = 1/\text{srate}$ is the sample time in seconds, and $\Delta f = (n\Delta t)^{-1} = \text{srate}/\text{npoint}$ is the size of a frequency bin, in Hz. Here $n = \text{npoint}$ is the number of points in the data stream which are being optimally filtered in one pass.

Chirp templates (in frequency space) for the two polarizations are related to the arrays `chirp0[]` and `chirp1[]` by

$$\tilde{T}_0(f) = \frac{\Delta t}{\text{HSCALE}} \text{chirp0}[] \quad (6.30.1)$$

$$\tilde{T}_{90}(f) = \frac{\Delta t}{\text{HSCALE}} \text{chirp1}[] \quad (6.30.2)$$

where the elements `chirp0[2j]` and `chirp0[2j+1]` are the real and imaginary parts at frequency $f = j\Delta f$ (with the exception of the Nyquist frequency, stored in `chirp0[1]`). Note that to ensure that quantities within the code remain within the dynamic range of floating point numbers, we have scaled up the template strain by a constant factor `HSCALE`; we also scale up the interferometer output by the same factor, so that all program output (such as signal-to-noise ratios) is independent of the value of `HSCALE`. If you're not comfortable with this, go ahead and change `HSCALE` to 1. It won't change anything, provided that you don't overflow the dynamic range of the floating point variables! The scaled interferometer response function is

$$\text{response}[] = \text{HSCALE}/\text{ARMLENGTH} \times R(f), \quad (6.30.3)$$

where the function $R(f)$ is defined by equation (3.12.3). The Fourier transform \tilde{h} of the dimensionless strain is obtained by multiplying Δt and the FFT of `channel.0` by `response[]`, yielding

$$\tilde{h}(f) = \frac{\Delta t}{\text{HSCALE}} \text{htilde}[]. \quad (6.30.4)$$

The one-sided noise power spectrum $S_h(f)$ is the average of

$$S_h(f) = \frac{2}{n\Delta t} |\tilde{h}(f)|^2 = \frac{2}{n\Delta t} \frac{(\Delta t)^2}{\text{HSCALE}^2} |\text{htilde}[]|^2 = \frac{2\Delta t}{n \text{HSCALE}^2} |\text{htilde}[]|^2. \quad (6.30.5)$$

The power spectrum $S_h(f)$ is averaged using the same exponential averaging technique described for the routine `avg_spec()`. This average is stored as

$$S_h(f) = \frac{2\Delta t}{n \text{HSCALE}^2} \langle |\text{htilde}[]|^2 \rangle = \frac{\Delta t}{n \text{HSCALE}^2} \text{mean_pow_spec}[] \quad (6.30.6)$$

Twice the inverse of this average is stored in the array `twice_inv_noise[]`, so that

$$\frac{2}{S_h(f)} = \frac{n \text{HSCALE}^2}{\Delta t} \text{twice_inv_noise}[]. \quad (6.30.7)$$

The expected noise-squared for the plus polarization is given by equation (6.14.8):

$$\begin{aligned} \langle N^2 \rangle &= \frac{1}{2} (Q, Q) = \frac{1}{2} \int_{-\infty}^{\infty} df \frac{|\tilde{T}_0(f)|^2}{S_h(f)} \\ &= \frac{1}{2} \frac{1}{n\Delta t} \text{FFT}_0^{-1} \left[\frac{(\Delta t)^2}{\text{HSCALE}^2} |\text{chirp0}[]|^2 \frac{n\text{HSCALE}^2}{\Delta t} \frac{1}{2} \text{twice_inv_noise}[] \right] \end{aligned}$$

$$\begin{aligned}
 &= \frac{1}{2} FFT_0^{-1} \left[|\text{chirp0}[]|^2 \frac{1}{2} \text{twice_inv_noise}[] \right] \\
 &\rightarrow \frac{1}{2} \text{correlate}(\dots, \text{chirp0}[], \text{chirp0}[], \text{twice_inv_noise}[], \text{npoint}).
 \end{aligned}$$

where the subscript on the inverse FFT means “at zero lag”, and “ $\rightarrow f$ ” means “returned by the call to the function f ”. We have chosen a distance for the system producing the “chirp” $\tilde{T}(f)$ so that the expected value of $\langle N^2 \rangle = 1$.

In similar fashion, the signal S at lag t_0 is given by

$$\begin{aligned}
 S &= \left(\frac{\tilde{h}}{S_h}, \tilde{Q} \right) \\
 &= \int_{-\infty}^{\infty} df \frac{\tilde{h}(f) \tilde{T}_0^*(f)}{S_h(f)} e^{-2\pi i f t_0} \\
 &= \frac{1}{n\Delta t} FFT_i^{-1} \left[\frac{\Delta t}{\text{HSCALE}} \text{htilde}[] \frac{\Delta t}{\text{HSCALE}} (\text{chirp0}[])^* \frac{n \text{HSCALE}^2}{\Delta t} \frac{1}{2} \text{twice_inv_noise}[] \right] \\
 &= FFT_i^{-1} \left[\text{htilde}[] \text{chirp0}[] \frac{1}{2} \text{twice_inv_noise}[] \right] \\
 &\rightarrow \text{correlate}(\dots, \text{htilde}[], \text{chirp0}[], \text{twice_inv_noise}[], \text{npoint}), \tag{6.30.8}
 \end{aligned}$$

where now the subscript on the FFT means “at lag $t = i \Delta t$ ”.

You might wonder why we have been so careful – after all, both the signal and the noise, as we’ve defined them, are dimensionless, so it’s not surprising that all of the factors of Δt drop out of the final formulae for the signal and the expected noise-squared. The main reason we’ve been so long winded is to show exactly how the units cancel out, and to demonstrate that there aren’t any missing dimensionless constants, like `npoint`, left out of the program. Some sample output from this program is shown in the next section.

```
/* GRASP: Copyright 1997,1998 Bruce Allen */
#include "grasp.h"
#define NPOINT 131072 /* The size of our segments of data (13.1 secs) */
#define FLO 120.0 /* The low frequency cutoff for filtering */
#define HSCALE 1.e21 /* A convenient scaling factor; results independent of it */
#define MIN_INT0_LOCK 3.0 /* Number of minutes to skip into each locked section */
#define SAFETY 1000 /* Padding safety factor to avoid wraparound errors */
#define PR2 8 /* Value of p for the R^2 splitup test */

int main() {
    void realft(float*, unsigned long, int);
    int i, code=0, npoint, remain=0, maxi, chirplen, needed, diff, impulseoff, chirppoints, indices[PR2];
    float distance, snr_max, srate=9868.4208984375, tstart, *mean_pow_spec, timeoff, timestart;
    float *data, *htilde, *output90, *output0, *chirp0, *chirp90, *ch0tilde, *ch90tilde;
    float n0, n90, inverse_distance_scale, decaytime, *twice_inv_noise, datastart, tc;
    float lin0, lin90, invMpc_inject, varsplit, stats[2*PR2], gammq(float, float), var, *response;
    double decay=0.0, norm, prob;
    short *datas;
    FILE *fpifo, *fpss, *fplock;

    /* open the IFO output file, lock file, and swept-sine file */
    fpifo=grasp_open("GRASP_DATAPATH", "channel.0", "r");
    fplock=grasp_open("GRASP_DATAPATH", "channel.10", "r");
    fpss=grasp_open("GRASP_DATAPATH", "swept-sine.ascii", "r");

    /* number of points to sample and fft (power of 2) */
    needed=npoint=NPOINT;

    /* stores ADC data as short integers */
    datas=(short*)malloc(sizeof(short)*npoint);

    /* stores ADC data in time & freq domain, as floats */
    data=(float *)malloc(sizeof(float)*npoint);

    /* The phase 0 and phase pi/2 chirps, in time domain */
    chirp0=(float *)malloc(sizeof(float)*npoint);
    chirp90=(float *)malloc(sizeof(float)*npoint);

    /* Orthogonalized phase 0 and phase pi/2 chirps, in frequency domain */
    ch0tilde=(float *)malloc(sizeof(float)*npoint);
    ch90tilde=(float *)malloc(sizeof(float)*npoint);

    /* The response function (transfer function) of the interferometer */
    response=(float *)malloc(sizeof(float)*(npoint+2));

    /* The gravity wave signal, in the frequency domain */
    htilde=(float *)malloc(sizeof(float)*npoint);

    /* The autoregressive-mean averaged noise power spectrum */
    mean_pow_spec=(float *)malloc(sizeof(float)*(npoint/2+1));

    /* Twice the inverse of the mean noise power spectrum */
    twice_inv_noise=(float *)malloc(sizeof(float)*(npoint/2+1));

    /* Ouput of matched filters for phase0 and phase pi/2, in time domain, and temp storage */
    /* factor of 2 in size of output0 because it is used in splitup_freq4 for temp storage */
    output0=(float *)malloc(sizeof(float)*2*npoint);
    output90=(float *)malloc(sizeof(float)*npoint);
}
```

```
/* get the response function, and put in scaling factor */
normalize_gw(fpss,npoint,srate,response);
for (i=0;i<npoint+2;i++)
    response[i]*=HSCALE/ARMLENGTH_1994;

/* manufacture two chirps (dimensionless strain at 1 Mpc distance) */
make_filters(1.4,1.4,chirp0,chirp90,FLO,npoint,srate,&chirppoints,&tc,0,4);
/* normalization of next line comes from GRASP (5.6.3) and (5.6.4) */
inverse_distance_scale=2.0*HSCALE*(TSOLAR*C_LIGHT/MPC);
for (i=0;i<chirppoints;i++){
    ch0tilde[i]=chirp0[i]*=inverse_distance_scale;
    ch90tilde[i]=chirp90[i]*=inverse_distance_scale;
}

/* zero out the unused elements of the tilde arrays */
for (i=chirppoints;i<npoint;i++)
    ch0tilde[i]=ch90tilde[i]=0.0;

/* and FFT the chirps */
realft(ch0tilde-1,npoint,1);
realft(ch90tilde-1,npoint,1);

/* set length of template including a safety margin */
chirplen=chirppoints+SAFETY;
if (chirplen>npoint) abort();

/* This is the main program loop, which acquires data, then filters it */
while (1) {

    /* Seek MIN_INT0_LOCK minutes into a locked stretch of data */
    while (remain<needed) {
        code=get_data(fpifo,fplock,&tstart,MIN_INT0_LOCK*60*srate,
                    datas,&remain,&srate,1);
        if (code==0) return 0;
    }

    /* if just entering a new locked stretch, reset averaging over power spectrum */
    if (code==1) {
        norm=0.0;
        clear(mean_pow_spec,npoint/2+1,1);

        /* decay time for spectrum, in sec. Set to 15x length of npoint sample */
        decaytime=15.0*npoint/srate;
        decay=exp(-1.0*npoint/(srate*decaytime));
    }

    /* Get the next needed samples of data */
    diff=npoint-needed;
    code=get_data(fpifo,fplock,&tstart,needed,datas+diff,&remain,&srate,0);
    datastart=tstart-diff/srate;

    /* copy integer data into floats */
    for (i=0;i<npoint;i++) data[i]=datas[i];

    /* inject signal in time domain (note output0[] used as temp storage only) */
    invMpc_inject=0.0; /* To inject a signal at 10 kpc, set this to 100.0 */
    time_inject_chirp(1.0,0.0,12345,invMpc_inject,chirp0,chirp90,data,
                    response,output0,npoint);
}
```

```
/* find the FFT of data*/
realft(data-1,npoint,1);

/* normalized delta-L/L tilde */
product(htilde,data,response,npoint/2);

/* update the inverse of the auto-regressive-mean power-spectrum */
avg_inv_spec(FLO,srate,npoint,decay,&norm,htilde,mean_pow_spec,twice_inv_noise);

/* inject a signal in frequency domain, if desired */
invMpc_inject=0.0; /* For a signal at 10 kpc, set this to 100.0, else 0.0 */
freq_inject_chirp(-0.406,0.9135,23456,invMpc_inject,ch0tilde,ch90tilde,htilde,
                 npoint);

/* orthogonalize the chirps: we never modify ch0tilde, only ch90tilde */
orthonormalize(ch0tilde,ch90tilde,twice_inv_noise,npoint,&n0,&n90);

/* distance scale Mpc for SNR=1 */
distance=sqrt(1.0/(n0*n0)+1.0/(n90*n90));

/* find the moment at which SNR is a maximum */
find_chirp(htilde,ch0tilde,ch90tilde,twice_inv_noise,n0,n90,output0,output90,
           npoint,chirplen,&maxi,&snr_max,&lin0,&lin90,&var);

/* identify when an impulse would have caused observed filter output */
impulseoff=(maxi+chirppoints)%npoint;
timeoff=datastart+impulseoff/srate;
timestart=datastart+maxi/srate;

/* if SNR greater than 5, then print details, else just short message */
if (snr_max<5.0)
    printf("max snr: %.2f offset: %d data start: %.2f sec. variance: %.5f\n",
           snr_max,maxi,datastart,var);
else {
    /* See if the nominal chirp can pass a frequency-space single-phase veto test */
    varsplit=splitup_freq2(lin0*n0/sqrt(2.0),lin90*n90/sqrt(2.0),ch0tilde,
                           ch90tilde,2.0/(n0*n0),twice_inv_noise,npoint,maxi,PR2,
                           indices,stats,output0,htilde);
    prob=gammq(0.5*(PR2-1),0.5*PR2*varsplit);
    /* See if the nominal chirp can pass a frequency-space two-phase veto test */
    varsplit=splitup_freq3(lin0*n0/sqrt(2.0),lin90*n90/sqrt(2.0),ch0tilde,
                           ch90tilde,2.0/(n0*n0),twice_inv_noise,npoint,maxi,PR2,
                           indices,stats,output0,htilde);
    prob=gammq(PR2-1,0.5*PR2*varsplit);
/* printf("Splitup 3 returns variance: %f\n",varsplit); */

    varsplit=splitup_freq5(lin0*n0/sqrt(2.0),lin90*n90/sqrt(2.0),ch0tilde,
                           ch90tilde,2.0/(n0*n0),twice_inv_noise,npoint,maxi,PR2,
                           indices,stats,output0,htilde);
    prob=gammq(PR2-1,0.5*PR2*varsplit);
/* printf("Splitup 5 returns variance: %f\n",varsplit); */

    printf("\nMax SNR: %.2f (offset %d) variance %f\n",snr_max,maxi,var);
    printf("    If impulsive event, offset %d or time %.2f\n",impulseoff,timeoff);
    printf("    If inspiral, template start offset %d (time %.2f) ",maxi,timestart);
    printf("coalescence time %.2f\n",timestart+tc);
    printf("    Normalization: S/N=1 at %.2f kpc\n",1000.0*distance);
    printf("    Lin combination of max SNR: %.4f x phase_0 + %.4f x phase_pi/2\n",
           lin0,lin90);
}
```

```
if (prob<0.01)
    printf("    Less than 1%% probability that this is a chirp (p=%f).\n",prob);
else
    printf("    POSSIBLE CHIRP!  with > 1%% probability (p=%f).\n",prob);

/* See if the time-domain statistics are unusual or appears Gaussian */
if (is_gaussian(datas,npoint,-2048,2047,1))
    printf("    Distribution does not appear to have outliers...\n\n");
else
    printf("    Distribution has outliers! Reject\n\n");
}

/* shift ends of buffer to the start */
needed=npoint-chirplen+1;
for (i=0;i<chirplen-1;i++)
    datas[i]=datas[i+needed];

/* reset if not enough points remain to fill the buffer */
if (remain<needed)
    needed=npoint;
}
}
```

6.31 Some output from the optimal program

Some output from the optimal program follows:

...

max snr: 3.11 offset: 23623 data start: 180.00 sec. variance: 0.94044
max snr: 2.91 offset: 3311 data start: 185.17 sec. variance: 0.84484

...

max snr: 2.53 offset: 19041 data start: 309.26 sec. variance: 0.70333
max snr: 2.98 offset: 35711 data start: 314.43 sec. variance: 0.67523

Max SNR: 8.71 (offset 42109) variance 0.805030

If impulsive event, offset 55624 or time 325.23

If inspiral, template start offset 42109 (time 323.86) coalescence time 325.23

Normalization: S/N=1 at 116.75 kpc

Linear combination of max SNR: $0.9315 \times \text{phase}_0 + 0.3638 \times \text{phase}_{\pi/2}$

Less than 1% probability that this is a chirp ($p=0.000000$).

Distribution: $s=23$, $N>3s=12$ (expect 176), $N>5s=0$ (expect 0)

Distribution does not appear to have outliers...

max snr: 2.51 offset: 31183 data start: 324.77 sec. variance: 0.63028

max snr: 2.56 offset: 49909 data start: 329.94 sec. variance: 0.66853

...

max snr: 2.82 offset: 35080 data start: 3002.03 sec. variance: 0.77306

max snr: 2.61 offset: 33141 data start: 3007.20 sec. variance: 0.74268

Max SNR: 89.75 (offset 16678) variance 82.547005

If impulsive event, offset 30193 or time 3015.43

If inspiral, template start offset 16678 (time 3014.06) coalescence time 3015.43

Normalization: S/N=1 at 128.49 kpc

Linear combination of max SNR: $-0.3955 \times \text{phase}_0 + 0.9185 \times \text{phase}_{\pi/2}$

Less than 1% probability that this is a chirp ($p=0.000000$).

Distribution: $s=29$, $N>3s=157$ (expect 176), $N>5s=30$ (expect 0)

Distribution has outliers! Reject

max snr: 3.24 offset: 22412 data start: 3017.54 sec. variance: 0.99474

max snr: 2.73 offset: 37777 data start: 3022.71 sec. variance: 0.75325

...

max snr: 2.80 offset: 5893 data start: 4140.89 sec. variance: 0.73240

max snr: 2.75 offset: 46932 data start: 4146.06 sec. variance: 0.69654

Max SNR: 6.08 (offset 30002) variance 0.883380

If impulsive event, offset 43517 or time 4155.64

If inspiral, template start offset 30002 (time 4154.27) coalescence time 4155.64

Normalization: S/N=1 at 113.04 kpc

Linear combination of max SNR: $-0.4773 \times \text{phase}_0 + 0.8787 \times \text{phase}_{\pi/2}$

POSSIBLE CHIRP! with $> 1\%$ probability ($p=0.024142$).

Distribution: $s=31$, $N>3s=399$ (expect 176), $N>5s=53$ (expect 0)

Distribution has outliers! Reject

```
max snr: 2.77 offset: 15985 data start: 4156.40 sec. variance: 0.72095
max snr: 2.69 offset: 47338 data start: 4161.57 sec. variance: 0.69708
...
```

This output shows three events that triggered an optimal filtering routine. The first and second of these events were rejected for different reasons. The first was rejected because it failed the frequency-distribution test. The second was rejected because it had 30 outlier points. The third failed for the same reason: it had 53 outlier points.

Next, we show some output when a fake chirp signal is injected into the data stream. This can be done for example by modifying `optimal` to read:

```
invMpc_inject=100.0; /* To inject a signal at 10 kpc, set this to 100.0 */
time_inject_chirp(1.0,0.0,12345,invMpc_inject,chirp0,chirp90,data,response,output0
```

This produces the following output:

```
...
Max SNR: 9.96 (offset 12345) variance 0.872624
  If impulsive event, offset 25860 or time 187.79
  If inspiral, template start offset 12345 (time 186.42) coalescence time 187.79
  Normalization: S/N=1 at 152.17 kpc
  Linear combination of max SNR: 0.9995 x phase_0 + -0.0304 x phase_pi/2
  POSSIBLE CHIRP! with > 1% probability (p=0.421294).
  Distribution: s= 23, N>3s= 12 (expect 176), N>5s= 0 (expect 0)
  Distribution does not appear to have outliers...

Max SNR: 12.84 (offset 12345) variance 0.834527
  If impulsive event, offset 25860 or time 192.96
  If inspiral, template start offset 12345 (time 191.59) coalescence time 192.96
  Normalization: S/N=1 at 132.47 kpc
  Linear combination of max SNR: 0.9953 x phase_0 + 0.0973 x phase_pi/2
  POSSIBLE CHIRP! with > 1% probability (p=0.949737).
  Distribution: s= 22, N>3s= 28 (expect 176), N>5s= 0 (expect 0)
  Distribution does not appear to have outliers...

Max SNR: 14.86 (offset 12345) variance 0.801640
  If impulsive event, offset 25860 or time 198.13
  If inspiral, template start offset 12345 (time 196.76) coalescence time 198.13
  Normalization: S/N=1 at 127.90 kpc
  Linear combination of max SNR: 0.9993 x phase_0 + -0.0372 x phase_pi/2
  POSSIBLE CHIRP! with > 1% probability (p=0.999236).
  Distribution: s= 22, N>3s= 35 (expect 176), N>5s= 0 (expect 0)
  Distribution does not appear to have outliers...
...
```

The code is correctly finding the chirps, getting the distance and phase and time location of the chirps about as accurately as one would expect given the level of the IFO noise.

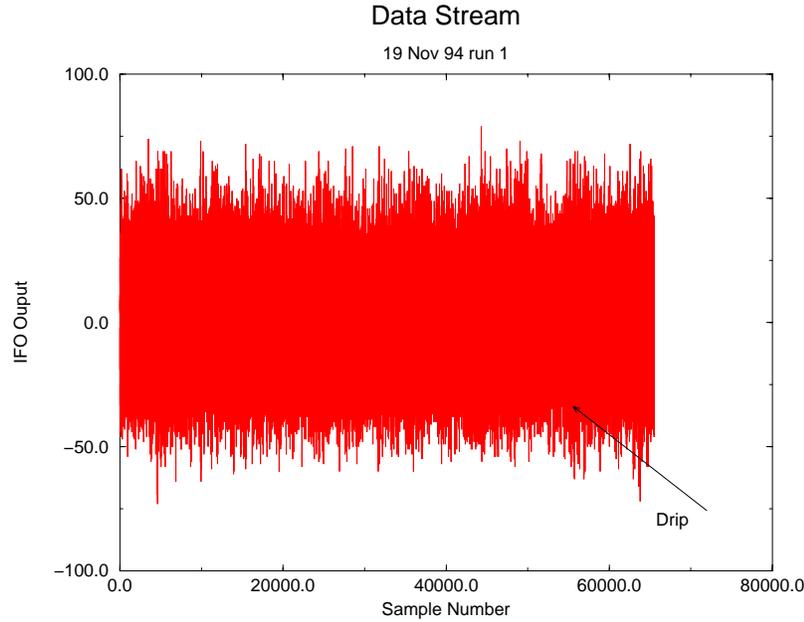


Figure 42: This shows the event that triggered the 2×1.4 solar mass binary inspiral filter with a SNR of 8.71 (see the first set of sample output from the optimal filtering code above, at time 325.23). This same “event” can also be seen in Figure 9. The horizontal axis is sample number, with samples $\approx 10^{-4}$ seconds apart; the vertical axis is the raw (whitened) IFO output. The event labeled “drip” can be heard in the data (it sounds like a faucet drip) and is picked up by the optimal filtering technique, but it is NOT visible to the naked eye. This event is vetoed by the splitup technique described earlier - it has extremely low probability of being a chirp plus stationary noise.

There are several interesting lessons that one can learn from this optimal filtering experience. The first is that (roughly speaking) the events that trigger an optimal filter (driving the output to a value much larger than would be expected for a colored-noise Gaussian input) can be broken into two classes: those which can be seen in the raw data stream, and those which can not. Here, by “seen in the raw data stream”, we mean “visible to the naked eye upon examination of a graph”. Shown in the following two figures are examples of each type of spurious event.

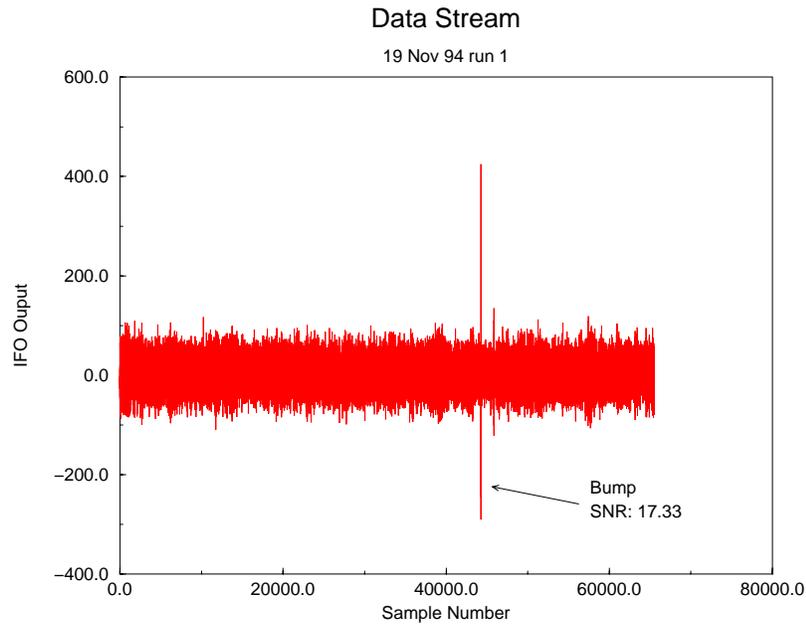


Figure 43: This another event that triggered the 2×1.4 solar mass binary inspiral filter with a SNR of 17.33. This event sounds like a “bump”; it is probably due to a bad cable connection. It can be easily seen (and vetoed) in the time domain. A close-up of this is shown in the next figure.

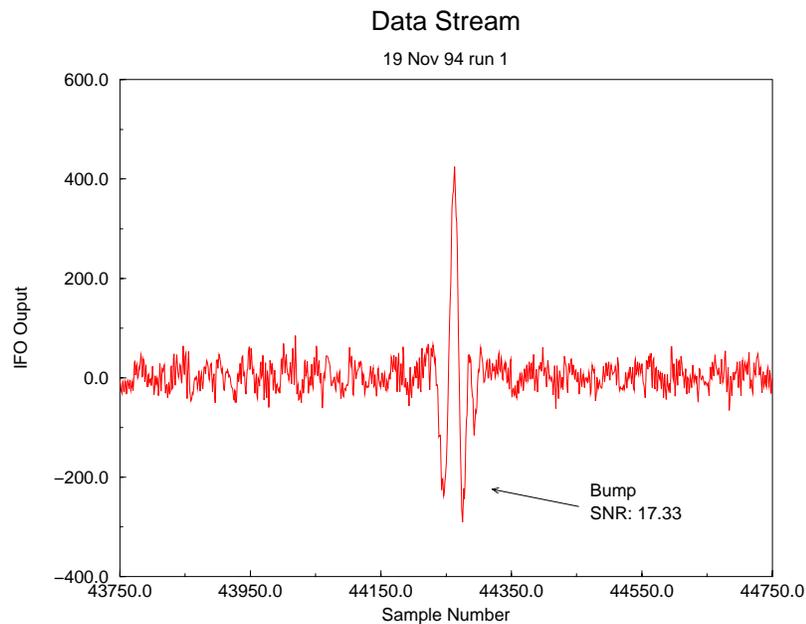


Figure 44: A close-up of the previous graph, showing the structure of the “bump”.

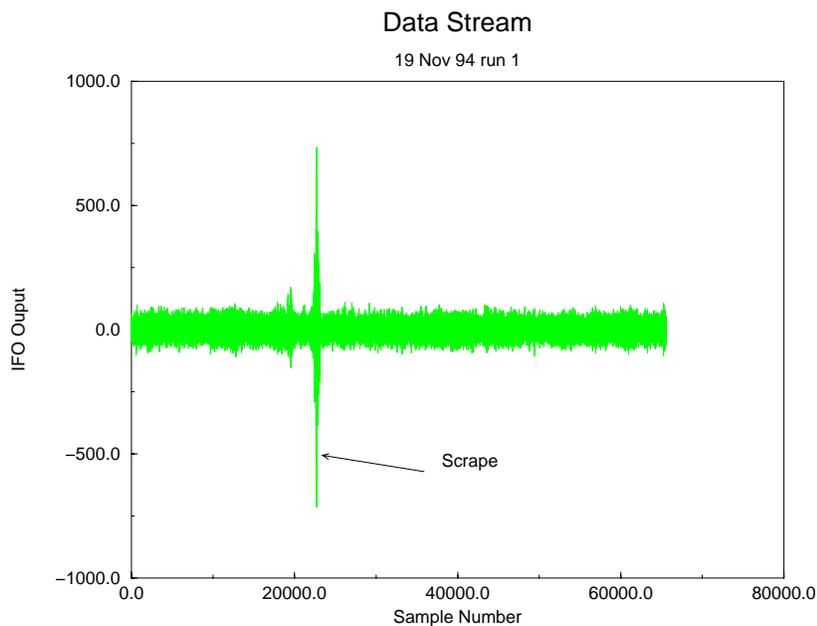


Figure 45: This another event that triggered the 2×1.4 solar mass binary inspiral filter with a SNR of 32.77. This event sounds like a shovel scraping on the ground; its origin is unknown. It can be easily seen (and vetoed) in the time domain.

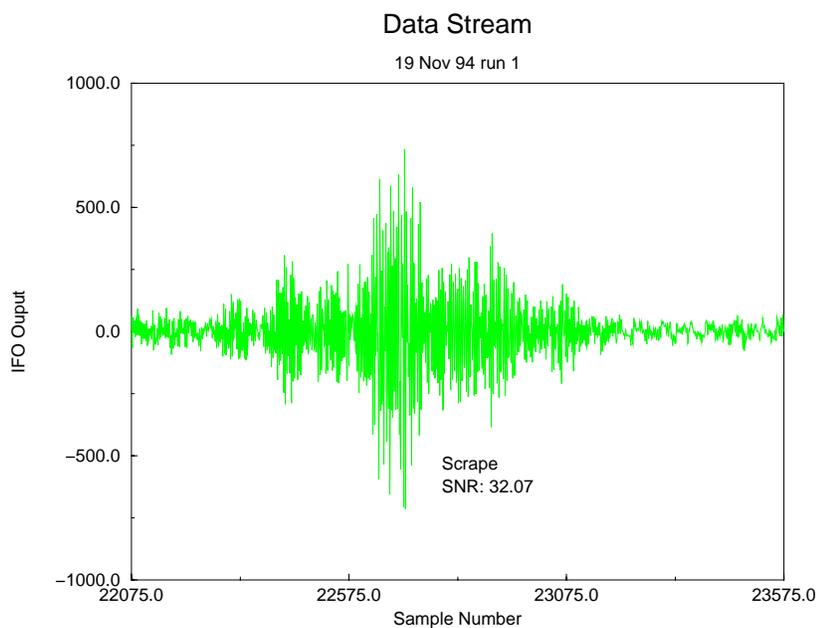


Figure 46: A close-up of the previous graph, showing the structure of the “scrape”.

6.32 The effective distance to which a source can be seen

Given a gravitational-wave detector with some known noise spectrum, it would clearly be useful to define an “effective distance” D_{eff} to which some given gravitational-wave source can be seen. To first order, any source located farther away from the detector than D_{eff} would be too weak to be detectable in the datastream. Sources closer than D_{eff} would be detectable.

This naive, heuristic picture of D_{eff} doesn’t make much sense in the real world because the source does not emit isotropically, and the detector does not detect isotropically: there are positions on the sky for which the detector can “see” farther, and the source radiates more strongly into some angles than others. A useful definition of D_{eff} must therefore average over angles in a meaningful, well-understood way.

One simple way to average over angles is to use Eq. (2.30) of Ref. [13]. In that reference, Flanagan and Hughes show that the signal-to-noise ratio, rms angle averaged over all source orientations and all positions on the sky, depends only on the spectrum of emitted gravitational-wave energy, dE_{gw}/df . Rearranging their formula (2.30) slightly, the effective distance to which a source can be seen with some rms angle-averaged signal-to-noise ratio ρ_0 is then

$$D_{\text{eff}}^{\text{FH}}(\rho_0)^2 = \frac{2(1+z)^2}{5\pi^2\rho_0^2} \left(\frac{G}{c^3}\right) \int_0^\infty df \frac{1}{f^2 S_h(f)} \frac{dE_{\text{gw}}}{df} [(1+z)f]. \quad (6.32.1)$$

The distance $D_{\text{eff}}^{\text{FH}}$ so defined is actually a luminosity distance; assuming some set of cosmological parameters and using standard formulae, one can then easily convert $D_{\text{eff}}^{\text{FH}}$ to an effective redshift $z_{\text{eff}}^{\text{FH}}$, and thence compute the comoving volume $V_c(z_{\text{eff}}^{\text{FH}})$ that is contained to that distance. If one assumes that the event rate of sources locked into Hubble flow does not evolve with redshift, this allows one to simply convert from an event rate density R [with units number/(Mpc³ year)] to a detected event rate N (with units number/year). (This assumption is clearly a rather bad one: the event rate will undoubtedly evolve with redshift. However, we don’t currently know *how* it will so evolve. This simple, albeit stupid, assumption is a useful one for estimating event rates for gravitational-wave sources.)

Notice that the cosmological redshift z explicitly appears in Eq. (6.32.1). These factors enter in such a way that the mass “imprinted” on the gravitational waveform (*i.e.*, the mass that gravitational-wave detections measure at the earth) will be redshift from M to $(1+z)M$.

Finn and Chernoff [15] define an effective distance in a somewhat different and more careful manner. Given a noise spectrum and given a threshold signal-to-noise ratio ρ_0 , they define an effective distance $D_{\text{eff}}^{\text{FC}}(\rho_0)$ as

$$N(\rho > \rho_0) = \frac{4\pi}{3} D_{\text{eff}}^{\text{FC}}(\rho_0)^3 R. \quad (6.32.2)$$

In words, the detection rate of events with signal-to-noise ratio ρ greater than the threshold ρ_0 is given the event rate density in space R times the volume of a sphere of radius $D_{\text{eff}}^{\text{FC}}(\rho_0)$. Finn and Chernoff then calculate $D_{\text{eff}}^{\text{FC}}$ using Monte-Carlo integration; see [15] for details.

Thorne [16],[17] has shown that for a pair of $1.4 M_\odot - 1.4 M_\odot$ neutron stars *and* for distances small enough that cosmological effects are negligible, the definitions given in Eqs. (6.32.1) and (6.32.2) are related by

$$\frac{D_{\text{eff}}^{\text{FC}}}{D_{\text{eff}}^{\text{FH}}} = 1.10. \quad (6.32.3)$$

For the purposes of GRASP, we will use an effective distance that is based on (6.32.1) because it is quick and simple to calculate, but correct using (6.32.3) in the hope that this will put us in reasonable agreement with the very careful calculations of Finn and Chernoff. (This factor of 1.10 probably varies somewhat with total system mass and with cosmological effects.) The formula for $D_{\text{eff}}(\rho_0)$ that we use is

$$D_{\text{eff}}(\rho_0)^2 = (1.10)^2 \times \frac{2(1+z)^2}{5\pi^2\rho_0^2} \int_0^\infty df \frac{1}{f^2 S_h(f)} \frac{dE_{\text{gw}}}{df} [(1+z)f]. \quad (6.32.4)$$

Section	GRASP Routines: Gravitational Radiation from Binary Inspiral	Page
6.32	The effective distance to which a source can be seen	205

The following routine calculates this effective distance, providing also the associated redshift and co-moving volume.

6.33 Function: `inspiral_dist()`

```
void inspiral_dist(double *deff, double *z, double *Vc, double m1_z,
                  double m2_z, double snr, double S_h[], int npoint,
                  double srate, double h100)
```

This function computes the effective distance to which a binary inspiral with redshifted masses `m1_z` and `m2_z` can be seen with the noise spectrum `S_h[]`.

It uses the energy spectrum

$$\frac{dE_{\text{gw}}}{df} = \frac{\pi^{2/3}}{3} \mu M^{2/3} f^{-1/3} \quad (6.33.1)$$

to describe the inspiral for frequencies $f < f_{\text{merge}} = 0.02/M$, and zero above f_{merge} (as in reference [13]). To convert from luminosity distance to redshift, it assumes a universe flat cosmology ($\Lambda = 0, \Omega = 1$) with a Hubble constant $H_0 = 75 \text{ km/Mpc sec}$, and uses an Eq. (11) from [18]. To convert from redshift to comoving volume, it uses Eq. (27) of [19] or Eq. (2.56) with $q_0 = 1/2$ of [20].

The arguments to the function are:

`deff`: Output. The effective distance in megaparsecs.

`z`: Output. Redshift corresponding that effective distance.

`Vc`: Output. Comoving volume at the redshift in cubic megaparsecs.

`m1_z`: Input. Redshifted mass one, $(1 + z)m_1$.

`m2_z`: Input. Redshifted mass two, $(1 + z)m_2$.

`snr`: Input. The signal-to-noise ratio at which the effective distance is `deff`.

`S_h`: Input. The spectral density of noise in Hz^{-1} .

`npoint`: Input. The number of data points in `S_h`.

`srate`: Input. The sampling rate used to construct the noise spectrum, Hz.

`h100`: Input. The Hubble constant in units of 100 km/sec/Mpc.

Author: Scott Hughes, hughes@tapir.caltech.edu

6.34 Function: merger_dist()

```
void merger_dist(double *deff, double *z, double *Vc, double m1_z,  
                double m2_z, double snr, double S_h[], int npoint,  
                double srate, double h100)
```

This function computes the effective distance to which a binary merger with redshifted masses $m_{1,z}$ and $m_{2,z}$ can be seen with the noise spectrum S_h [].

It uses the energy spectrum

$$\frac{dE_{\text{gw}}}{df} = \frac{\epsilon M}{f_{\text{qnr}} - f_{\text{merge}}} \quad (6.34.1)$$

to describe the merger, using parameters $\epsilon = 0.1$, $f_{\text{qnr}} = 0.13/M$, $f_{\text{merge}} = 0.02/M$ (as in reference [13]). As such it is, strictly speaking, only applicable to binary black hole mergers. Its operation is otherwise identical to `inspiral_dist()`.

The arguments to the function are:

`deff`: Output. The effective distance in megaparsecs.

`z`: Output. Redshift corresponding that effective distance.

`Vc`: Output. Comoving volume at the redshift in cubic megaparsecs.

`m1_z`: Input. Redshifted mass one, $(1+z)m_1$.

`m2_z`: Input. Redshifted mass two, $(1+z)m_2$.

`snr`: Input. The signal-to-noise ratio at which the effective distance is `deff`.

`S_h`: Input. The spectral density of noise in Hz^{-1} .

`npoint`: Input. The number of data points in `S_h`.

`srate`: Input. The sampling rate used to construct the noise spectrum, Hz.

`h100`: Input. The Hubble constant in units of 100 km/sec/Mpc.

Author: Scott Hughes, hughes@tapir.caltech.edu

6.35 Example: compute_dist program

This program will tell you the distance to which a binary with given redshifted masses $[(1+z)m_1, (1+z)m_2]$ can be seen with a given signal-to-noise rate in some given detector (which must be listed in the GRASP file `detectors.dat`). It will also tell you the redshift at that effective (luminosity) distance and the corresponding comoving volume. It uses that comoving volume to convert a given event rate density to a measured event rate.

The code specifies its various input parameters with command line flags. Thus, `compute_dist -m1 5 -m2 7 -snr 6 -d 24 -R 2.e-7 -h100 0.75` will compute the effective distance for a binary that has $(1+z)m_1 = 5M_\odot$, $(1+z)m_2 = 7M_\odot$ with signal-to-noise ratio 6 in detector 24 (the zeroth stage of enhancement at the Livingston LIGO site); and it will compute the detected event rate with an assumption that the event rate density is 2×10^{-7} events per cubic megaparsec per year, in a cosmology with Hubble constant today of 75 km/sec/Mpc. If a flag or parameter is omitted, default values are used; type

```
compute_dist -h
```

to see those default values.

```
#include "grasp.h"

#define NPOINTS 32768

int main(int argc, char **argv) {
    void HELP();
    double *S_h;
    float site_parameters[8];
    double delta_f, srate = 20000.; /* Hz */
    char noise_file[128], whiten_file[128], site_name[128];
    int i, npoint=131072;

    /* default parameter values */
    double m1_z=1.4, m2_z=1.4, R=1.e-8, snr=5.5, h100=0.75;
    int detector=21;
    double dinsp, dmerge, zinsp, zmerge, Vcinsp, Vcmerge;

    /* get parameters from the command line */
    for(i=0; i<argc-1; i++) {
        if(!strcmp(argv[i+1], "-h")) {
            HELP();
            exit(0);
        }
        if(!strcmp(argv[i+1], "-snr")) {
            if(!(snr=strtod(argv[i+2], NULL))) {
                fprintf(stderr, "Error assigning SNR, defaulting to 5.5.\n");
                snr=5.5;
            }
            if(snr<0.) snr=5.5;
        }
        if(!strcmp(argv[i+1], "-h100")) {
            if(!(h100=strtod(argv[i+2], NULL))) {
                fprintf(stderr, "Error assigning Hubble constant h100, defaulting to 0.75\n");
                h100=0.75;
            }
            if(h100<0.) h100=fabs(h100);
        }
        if(!strcmp(argv[i+1], "-m1")) {
```

```
    if(!(m1_z=strtod(argv[i+2],NULL))) {
        fprintf(stderr,"Error assigning redshifted m1, defaulting to 1.4\n");
        m1_z = 1.4;
    }
    if(m1_z<0.) m1_z=1.4;
}
if(!strcmp(argv[i+1],"-m2")) {
    if(!(m2_z=strtod(argv[i+2],NULL))) {
        fprintf(stderr,"Error assigning redshifted m2, defaulting to 1.4\n");
        m2_z = 1.4;
    }
    if(m2_z<0.) m2_z=1.4;
}
if(!strcmp(argv[i+1],"-d")) {
    if(!(detector=atoi(argv[i+2]))) {
        fprintf(stderr,"Error assigning detector number, defaulting to 21\n");
        detector=21;
    }
    if(detector<1) detector=21;
}
if(!strcmp(argv[i+1],"-R")) {
    if(!(R=strtod(argv[i+2],NULL))) {
        fprintf(stderr,"Error assigning rate density, defaulting to 1.e-8\n");
        R=1.e-8;
    }
    if(R<0.) R=1.e-8;
}
}

/* Get info for that detector */
detector_site("detectors.dat",detector,site_parameters,site_name,
             noise_file,whiten_file);

/* allocate memory for the noise power spectrum */
S_h=(double *)malloc(sizeof(double)*(npoint/2+1));
delta_f=srate/((double)npoint);

/* Fill in the noise power spectrum for the detector */
noise_power(noise_file,npoint/2+1,delta_f,S_h);

/* compute effective distance for which inspiral has SNR = value */
inspiral_dist(&dinsp,&zinsp,&Vcinsp,m1_z,m2_z,snr,S_h,npoint,srate,h100);

/* compute effective distance for which merger has SNR = value */
merger_dist(&dmerge,&zmerge,&Vcmerge,m1_z,m2_z,snr,S_h,npoint,srate,h100);

printf("%s: D_insp = %e  z_insp = %e  Vc_insp = %e  N = %e\n",
       site_name,dinsp,zinsp,Vcinsp,R*Vcinsp);

printf("%s: D_merge = %e  z_merge = %e  Vc_merge = %e  N = %e\n\n",
       site_name,dmerge,zmerge,Vcmerge,R*Vcmerge);

return 0;
}

void HELP()
{
    fprintf(stderr,"\nThis GRASP code takes the following flags:\n\n");
    fprintf(stderr," -h:          Show this help message.\n\n");
}
```

```
fprintf(stderr, " -snr [snr]:    Use a signal to noise ratio of snr.\n");
fprintf(stderr, "                    Default value is 5.5.\n\n");
fprintf(stderr, " -m1 [(1+z)m1]: Set redshifted mass 1.\n");
fprintf(stderr, "                    Default value is 1.4.\n\n");
fprintf(stderr, " -m2 [(1+z)m2]: Set redshifted mass 2.\n");
fprintf(stderr, "                    Default value is 1.4.\n\n");
fprintf(stderr, " -d [dn]       : Set detector number to dn.  dn must\n");
fprintf(stderr, "                    be a detector integer defined in the\n");
fprintf(stderr, "                    GRASP file detectors.dat.\n");
fprintf(stderr, "                    Default value is 16, corresponding to\n");
fprintf(stderr, "                    Hanford enhancement 0.\n\n");
fprintf(stderr, " -R [R]       : Set rate density to R.  Units are events\n");
fprintf(stderr, "                    per (year Mpc^3).\n");
fprintf(stderr, "                    Default value is 1.e-8.\n\n");
fprintf(stderr, " -h100 [h100] : Set Hubble constant in units of 100 km/sec/Mpc\n");
fprintf(stderr, "                    Default value is 0.75.\n\n");
return;
}
```

Author: Scott Hughes, hughes@tapir.caltech.edu

7 GRASP Routines: Waveforms from perturbation theory

An alternative method of calculating the waveforms generated during a binary inspiral is provided by black hole perturbation theory. In the limit of a small mass ratio $\eta = m_1 m_2 / (m_1 + m_2)^2$ (the test mass limit) we can treat the effects of the smaller body as perturbations of the gravitational field of the larger mass. If the latter is a black hole then the Teukolsky formalism [30] can be used to calculate these perturbations. The corrections to the metric are then expressed as an infinite sum over multipole moments. Here we will consider the case of a Schwarzschild black hole.

7.1 The waveform

The Teukolsky formalism gives the two components h_+ and h_\times of the gravitational waves produced by a test mass in a fixed, circular orbit of radius r_0 as

$$h_+(u) - ih_\times(u) = \frac{2G\mu}{rc^2} \sum_{l \geq 2, |m| \leq l} A_{lm}(M\Omega) e^{-im\Omega(t-cr^*)} {}_{-2}Y_{lm}(\vartheta, \varphi), \quad (7.1.1)$$

where $u = t - cr^*$ is the standard retarded time. Here G is Newton's constant and c the speed of light. Here the functions ${}_{-2}Y_{lm}(\vartheta, \varphi)$ are the spherical harmonics of spin weight -2 , Ω is the angular velocity and $M = m_1 + m_2$ is the total mass of the system. The angles ϑ and φ are the usual spherical coordinates, as defined in Section 8.1, Figure 50 ($\vartheta = \iota$ and $\vartheta = \beta$). Since the motion is symmetric around the z -axis φ does not have an intrinsic meaning. Throughout this section angles are measured in radians. The amplitudes A_{lm} are constants and have to be calculated numerically by solving the Teukolsky equation. We provide A_{lm} tabulated in a datafile [45] as a function of the orbital velocity v (measured in units of the speed of light c)

$$cv = r_0\Omega = \sqrt{\frac{GM}{r_0}} = (GM\Omega)^{\frac{1}{3}} = (2\pi GMf)^{\frac{1}{3}}. \quad (7.1.2)$$

Here f is the *orbital* frequency measured in units on Hertz.

To account for the decay of the orbit due to the emission of gravitational waves we use an adiabatic approximation: The energy radiated away as calculated from expression (7.1.1) is used to calculate the change in orbital frequency. By doing so, Ω , and thus also v , become functions of time and we have to replace the product Ωu by an integral $\int \Omega(u) du$.

Following [46] we find that the time evolution of the velocity is governed by

$$\dot{v} = \frac{32}{5} \frac{\mu c^3}{GM^2} v^9 \frac{P(v)}{Q(v)}, \quad (7.1.3)$$

where $\mu = m_1 m_2 / M$ is the reduced mass of the system. The function $P(v)$, which is determined by the A_{lm} 's, is defined by the equation

$$\dot{E} = \frac{dE}{dt} = P(v) \left(\frac{dE}{dt} \right)_N$$

where $(dE/dt)_N = -32\mu^2 v^{10} c^4 / 5GM^2$ is the quadrupole-formula expression for the gravitational-wave luminosity. We also provide $P(v)$ in tabulated form [45]. The function $Q(v) = (1 - 6v^2)(1 - 3v^2)^{-\frac{3}{2}}$ relates the energy and frequency of the orbiting particle:

$$\frac{dE}{df} = E(v) \left(\frac{dE}{df} \right)_N.$$

This can be calculated by solving the geodesic equation for circular orbits in the Schwarzschild spacetime.

Note that $v(t)$ can be calculated from equation (7.1.3) once and for all: First (numerically) calculate the solution $V(t)$ of equation (7.1.3) with the factor $\mu c^3/GM^2$ set to one; then the solution $v(t)$ for general μ/M^2 is simply

$$v(t) = V\left(\frac{\mu c^3}{GM^2}t\right). \quad (7.1.4)$$

As mentioned before, we have to replace the product Ωu in equation (7.1.1) by an appropriate integral. We first note that we want to look at waves at a fixed radius $r^* \rightarrow \infty$. We can thus simply ignore the dependence on r^* because it will only contribute a fixed phase φ_0 . Since our data are tabulated as functions of the orbital velocity v , it is convenient to use v instead of the time t as an independent variable. This is permissible because v depends monotonically on t . Thus the phase becomes:

$$-im\Omega t \longrightarrow -im\frac{M}{\mu}\frac{5}{32}\int_{v_0}^v dv' \frac{Q(v')}{P(v')v'^6} =: -im\frac{M}{\mu}\Phi(v_0, v). \quad (7.1.5)$$

It is important to note that Φ is *not* unique, but depends on an arbitrary parameter v_0 . As with r^* , a change in v_0 will only affect the phase φ_0 . Since Φ can become rather large, the freedom in choosing v_0 can be used to keep Φ small in the region of interest. A good choice of v_0 will improve the numerical accuracy tremendously. (For example, the standard trigonometric functions in C become hopelessly inaccurate for arguments $> 10^6$ for floats and $> 10^{10}$ for doubles).

The signal is now given by

$$h_+(t) - ih_\times(t) = \frac{2G\mu}{rc^2} \sum_{l \geq 2, |m| \leq l} A_{lm}(v(t)) e^{-im\frac{M}{\mu}\Phi(v_0, v(t))} {}_{-2}Y_{lm}(\vartheta, \varphi), \quad (7.1.6)$$

where $v(t)$ is given by equation (7.1.4).

To extract h_+ and h_\times independently we use the fact that $A_{l-m} = (-1)^l \bar{A}_{lm}$ and, since ${}_{-2}Y_{lm}(\vartheta, \varphi) = {}_{-2}Y_{lm}(\vartheta, 0) e^{im\varphi}$ and ${}_{-2}Y_{lm}(\vartheta, 0)$ is real, we have ${}_{-2}Y_{l-m}(\vartheta, \varphi) = {}_{-2}Y_{lm}(\vartheta, 0) e^{-im\varphi}$ [26]. We can now split the sum (7.1.6) into real and imaginary part. This gives the two components as

$$h_+ = \frac{2G\mu}{rc^2} \sum_{2 \leq l, 1 \leq m \leq l} (c_m \Re_{lm} - s_m \Im_{lm}) \left({}_{-2}Y_{lm}(\vartheta, 0) + (-1)^l {}_{-2}Y_{l-m}(\vartheta, 0) \right) \quad (7.1.7)$$

$$h_\times = \frac{2G\mu}{rc^2} \sum_{2 \leq l, 1 \leq m \leq l} (s_m \Re_{lm} + c_m \Im_{lm}) \left(-{}_{-2}Y_{lm}(\vartheta, 0) + (-1)^l {}_{-2}Y_{l-m}(\vartheta, 0) \right).$$

Here we have introduced the notation $\Re_{lm} := \text{Re } A_{lm}$, $\Im_{lm} := \text{Im } A_{lm}$ and $s_m := \sin(-m(\Phi M/\mu - \varphi))$ and $c_m := \cos(-m(\Phi M/\mu - \varphi))$.

The GRASP routine which calculates h_+ and h_\times uses expression (7.1.7) truncated to a finite number of terms determined by the user. Finally we note that a change in φ has the same effect on the waveforms as a change in r^* and v_0 .

A note on the allowed frequency range

The frequency range of a signal calculated from black hole perturbation theory is bounded from above by the innermost stable circular orbit. This corresponds, by virtue of equation (7.1.3), to an orbital velocity of $v_{\max} = 1/\sqrt{6}$. In practice there is also a lower bound. Since zero frequency would correspond to an infinite orbital radius we have to introduce a cutoff. In the present data files $v_{\min} \simeq 0.0395$. Figure 47 shows the allowed frequency range as a function of the total mass M .

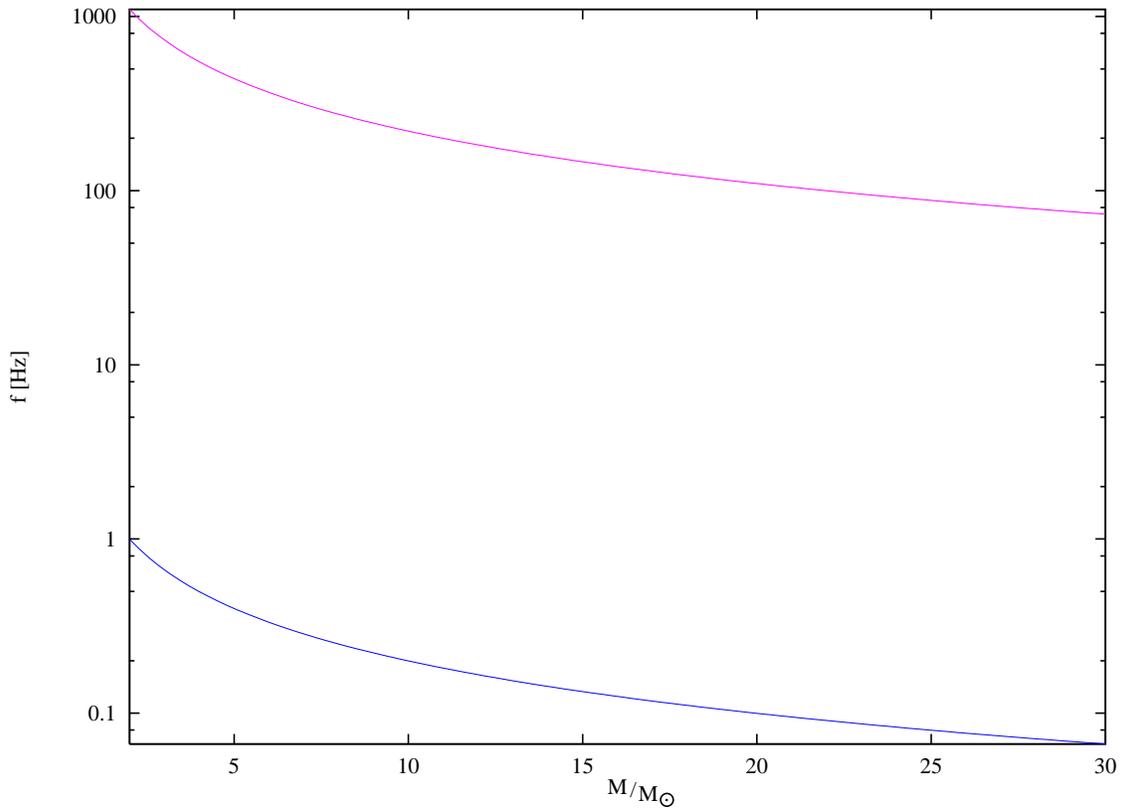


Figure 47: Maximum and minimum orbital frequencies as a function of the total mass M/M_{\odot} of the system.

Since \dot{v} in equation (7.1.3) diverges at $v = v_{\max}$ calculating $v(t)$ at the upper limit is numerically difficult. However this is not a problem in the time domain, since the system spends very little time near $v = v_{\min}$. An value of f_{\max} inaccurate by several percent will typically not change the signal by even one wave cycle.

7.2 Chirp generation for test mass signals

The routines provided in this package were designed to resemble as closely as possible the chirp generating routines for post-Newtonian signals. However, because of the different functional forms for the two types of waveforms some differences were unavoidable.

The main routine is `testmass_chirp()` which returns h_+ and h_{\times} in a given frequency interval. However, before being able to use `testmass_chirp()`, one has to read in the data files, calculate the phase $\Phi(v_0, v)$, etc. There are a number of routines provided that allow one to perform these tasks with a minimum of amount of work.

For `testmass_chirp()` to work, the program must make sure that the two data files containing the mode amplitudes $A_{lm}(v)$ and the luminosity $P(v)$ have been read. It then has to calculate $V(t)$ (which can be inverted to give $t(V)$) and $\Phi(v_0, v)$. Furthermore, it has to make the number of data points known to the package, in order for the interpolation routines to work. The package currently assumes that all data files have the same number of points and that v is equally spaced. Note that all the functions in the package take the orbital frequency $f = v^3 c^3 / GM\pi$, instead of v , as input.

7.3 Function: testmass_chirp()

```
int testmass_chirp(float m1, float m2, float theta, float phi, float *Phase,
float f_start, float f_end, float *f_started, float *f_ended, float dt, float
**hplus, float **hcross, float **frequency, int *number_of_points, int MaxL,
int *modes)
```

This function calculates the two unnormalized signals h_+ and h_\times . To normalize just multiply the output by the prefactor $2\mu/r$. This is the main routine of the test mass package. The arguments are:

m1: Input. The mass of the first body in units of the solar mass.

m2: Input. The mass of the second body in units of the solar mass.

theta: Input. The inclination angle ϑ in radians.

phi: Input. The azimuth φ in radians.

Phase: Input: A pointer to an array containing the phase function $\Phi(f_0, v)$. The number of points calculated must have been set beforehand, either by the supplied routines or through an explicit call of `Set_Up_Data()`.

f_start: Input. Starting *orbital* frequency in Hertz. If the frequency is too low it will be adjusted to the minimum allowed frequency.

f_end: Input. Final *orbital* frequency in Hertz. If set too high the program will terminate at the maximum frequency.

f_started: Output. The frequency in Hertz where the chirp actually started. This is $\max(f_{\text{start}}, v_{\text{min}}^3 c^3 / (2\pi GM))$.

f_ended: Output. The frequency in Hertz where the chirp terminated. This is $\min(f_{\text{end}}, v_{\text{max}}^3 c^3 / (2\pi GM))$

dt: Input. The time interval between successive samples in seconds.

hplus: Input/Output. The signal h_+ is stored in the array `*hplus[0..number_of_points-1]`. If `**hplus == NULL` memory will be allocated, otherwise the user has to provide the memory. The allocated memory is given by $((\text{number_of_points}/\text{kNumberOfFloats} + 1) * \text{kNumberOfFloats} * \text{sizeof}(\text{floats}))$. Note that this performs integer arithmetic, so it's not what you might expect.

hcross: Input/Output. The signal h_\times is stored in the array `*hcross[0..number_of_points-1]`. If `**hcross == NULL` memory will be allocated, otherwise the user has to provide the memory. Use the same expression as above to get the memory allocated.

frequency: Input/Output. The orbital frequency $f(t)$ is stored in the array `*frequency[0..number_of_points-1]`. If `**frequency == NULL` memory will be allocated, otherwise the user has to provide the memory.

number_of_points: Input/Output. The number of points requested. If `number_of_points == 0` then memory will be allocated for you. If `number_of_points` is nonzero, then at most this number of points will be returned. You must give the number of points if you allocated memory for any of the arrays `hplus`, `hcross` or `frequency` yourself. On exit this variable holds the actual number of points calculated.

MaxL: Input. The maximum number of modes to be used. For the supplied data file this has to be less than or equal to five. It is assumed that all m 's are available for a given l .

modes: Input. An array containing a list of modes to include in the sum (7.1.1). The array contains 1's for modes to be included and 0's otherwise. The sequence of modes is

index	0	1	2	3	4	5	6	7	8	9	10	11	...
l	2	2	2	2	3	3	3	3	3	3	4	4	...
m	-2	-1	+1	+2	-3	-2	-1	+1	+2	+3	-4	-3	...

Use the macro

```
#define mode2(l,m) ((l)*(l) + m - 2 - ((m > 0) ? 1 : 0)) to calculate the index.
```

Return value: Output. `testmass_chirp()` returns 0 if there was no error, and an error code otherwise. These codes are described in Section 7.15.

Author: Serge Droz, droz@physics.uoguelph.ca

Comments: As was mentioned above, you will get an error if the required data files are not read into memory. See Section 7.15 for a detailed description of the errors.

7.4 Function: calculate_testmass_phase()

```
int calculate_testmass_phase(float fo, float M, float **Phi)
```

This function calculates the phase $\Phi(f_0, v)$ which is needed to get the wave form.

fo: Input. The orbital frequency in Hertz at which Φ should vanish. f_0 is related to v_0 by $v_0^3 = 2\pi GM f_0 / c^3$.

M: Input. The total mass of the system in units of the solar mass. M is only used to convert the orbital frequency f_0 into the orbital velocity v_0 .

Phi: Input/Output. The array `*Phi` will contain the calculated phase $\Phi(f_0, v)$. It will contain the same number of points as the data read in from the stored files. If `**Phi=NULL`, memory will be allocated.

Return value: Output. Returns 0 if there was no error, and an error code otherwise. These codes are described in Section 7.15.

Author: Serge Droz, droz@physics.uoguelph.ca

Comments: You will get an error if the required data files are not read into memory. You must call this routine at least once before you can use `testmass_chirp()`.

7.5 Function: Get_Duration()

float Get_Duration(float f1, float f2, float m1, float m2)

Calculates how many seconds it takes for the system to evolve from the initial frequency f_1 to final frequency f_2 , i.e. the “duration” of a chirp.

f1: Input. The initial frequency f_1 in Hertz.

f2: Input. The final frequency f_2 in Hertz.

m1: Input. The mass of the first body in units of the solar mass.

m2: Input. The mass of the second body in units of the solar mass.

Return value: Output. The duration of the chirp in seconds or a value < 0 if an error occurred (if $t(v)$ has not been calculated).

Author: Serge Droz, droz@physics.uoguelph.ca

Comments: You must have read in the data files and calculated $t(v)$ for this to work.

7.6 Function: Get_Fmax()

float Get_Fmax(float m1, float m2)

Get the maximum frequency.

m1: Input. The mass of the first body in units of the solar mass.

m2: Input. The mass of the second body in units of the solar mass.

Return value: Output. The maximum frequency in Hertz for the given masses.

Author: Serge Droz, droz@physics.uoguelph.ca

Comments: You must have read in the data files and calculated $t(v)$ for this to work.

7.7 Function: ReadData()

int ReadData(char *filenameP, char *filenameAlm, float **v, int *number_of_points)

This function reads in all the required data and calculates $v(t)$. Using this function is probably the easiest method to ensure that the data is read into memory correctly. This routine can allocate all the necessary memory automatically.

`filenameP`: Input. The filename of the data file for the function $P(v)$. You must set the environment variable `GRASP_PARAMETERS` to the directory where the data files are stored (normally the parameter directory of your GRASP installation, for example `/usr/local/GRASP/parameters`). If `filename == NULL` the default file will be read.

`filenameAlm`: Input. The filename of the data file for the function $A_{lm}(v)$. See comments for `filenameP`.

`v`: Input. The array `v[0..number_of_points-1]` will contain all the read in values of v . If `v==NULL` memory is allocated.

`number_of_points`: Input/Output. If not set to zero, at most `*number_of_points` data points will be read. If you allocate memory yourself this variable must contain the maximal number of points that can be saved. On exit this variable will contain the actual number of points read into memory.

Return value: Output. Returns 0 if there was no error, and an error code otherwise. These codes are described in Section 7.15.

Author: Serge Droz, droz@physics.uoguelph.ca

Comments: None.

7.8 Function: Clean_Up_Memory()

```
void Clean_Up_Memory( float *Phase )  
Frees the memory allocated by ReadData().
```

Phase: Input: A pointer to the array containing the phase. If not NULL free the memory pointed to. Do only use this if memory was allocated from within GRASP or by using `malloc()`.

Author: Serge Droz, droz@physics.uoguelph.ca

Comments: None.

7.9 Function: Set_Up_Data()

```
void Set_Up_Data( float *v, float *P, float *T, float *ReA, float *ImA, int  
num_of_datapoints)
```

Allows a user to supply their own arrays containing data. If a pointer is non null, the array it points to will be used for the specific data.

Warning: This routine does very little error checking.

v: Input. An array containing the orbital velocities.

P: Input. An array containing $P(v)$.

T: Input. An array containing the time $t(v)$.

ReA: Input. An array containing the real part of $A_{lm}(v)$.

ImA: Input. An array containing the imaginary part of $A_{lm}(v)$.

num_of_datapoints: Input. The number of data points.

Author: Serge Droz, droz@physics.uoguelph.ca

Comments: It's assumed that all arrays contain the same number of data points, and that the values of v are equally spaced.

7.10 Function: minustwoS1m()

float minustwoS1m(float theta, int l, int m)
Calculate $_{-2}Y_{lm}(\vartheta, 0)$.

theta: Input. ϑ in radians.

l: Input. l .

Return value: Output. $_{-2}Y_{lm}(\vartheta, 0)$.

Author: Serge Droz, droz@physics.uoguelph.ca

Comments: This function calls the more general GRASP function `sw_spheroid()` to calculate $_{-2}Y_{lm}(\vartheta, 0)$.

7.11 Function: read_modes()

```
int read_modes(const char *filename, float **x, float **ReA, float **ImA,  
int *number_of_points, int *MaxL, int ReadX)
```

Read the modes $A_{lm}(v)$ from a data file. The data file is assumed to be of the form

```
2 1  
v0 (Re(A21)0,Im(A21)0)  
v1 (Re(A21)1,Im(A21)1)  
    ⋮  
2 2  
v0 (Re(A22)0,Im(A22)0)  
    ⋮  
2 1  
v0 (Re(A31)0,Im(A31)0)  
    ⋮
```

There is some consistency checking done during the reading of the file (e.g. the number of points per mode have to agree for all modes, etc.).

filename: The name of the file containing the modes A_{lm} . If NULL then the default file is use.

x: Input/Output. The array `*v[0..number_of_points-1]` will contain the values $v_0 \dots$. If `**x == NULL` allocate the memory. If `ReadX` is false do not read the v-values (they still have to be in the data file though).

ReA: Input/Output. The array `*ReA` will contain the real parts of the A_{lm} 's. If set to NULL memory is allocated and a pointer to it will be returned.

ImA: Input/Output. The array `*ReA` will contain the imaginary parts of the A_{lm} 's. If NULL allocate memory.

number_of_points: Input/Output. The number of points read. If zero the routine reads all available data points. If memory is provided by the user for any of the arrays mentioned above, this must be the maximum number of points you can store.

MaxL: Input/Output. The number of l modes to read. If zero read all of them (currently five). The output value is the number of successfully read l 's.

ReadX: Input. If false (=0) don't save the v-values in `*x`.

Return value: An error code described in Section 7.15.

Author: Serge Droz, droz@physics.uoguelph.ca

Comments: You must set the environment variable `GRASP_PARAMETERS` to the name of the GRASP parameter directory.

7.12 Function: read_real_data_file()

```
int read_real_data_file(const char *filename, float **x, float **y, int *num-  
ber_of_points, int ReadX)
```

Read in a simple data file consisting of just two columns of data.

filename: Input. The file to read.

x: Input/Output. If ReadX is true ($\neq 0$) the array `x[0..*number_of_points-1]` will contain the first column of data. If `**x == NULL` allocate the memory. If `number_of_points` is nonzero allocate space for `number_of_points` points.

y: Input/Output. The array `y[0..*number_of_points-1]` will contain the second column of data. If `**y == NULL` allocate the memory. If `number_of_points` is nonzero allocate space for `number_of_points` points.

ReadX: Input. If false (=0) don't read the x-values.

Return value: Output. Errors.

Author: Serge Droz, droz@physics.uoguelph.ca

Comments: You must set the environment variable `GRASP_PARAMETERS` to the name of the GRASP parameter directory.

7.13 Function: integrate_function()

```
int integrate_function(float vl, float vr, float vo, float (*f)(float ),  
float **F, int number_of_points)
```

This function returns an array containing the values of $F(v) = \int_{v_0}^v f(x)dx$ in the interval $[v_l, v_r]$.

vl: Input. v_l .

vr: Input. v_r .

vo: Input. v_0 .

f: Input. The function to be integrated.

F: Input. An array containing the result at equally spaced values of v . If $**F=NULL$ allocate the memory.

number_of_points: Input. The number of points desired.

Return value: Output. Returns 0 if there was no error, and an error code otherwise. These codes are described in Section 7.15.

Author: Serge Droz, droz@physics.uoguelph.ca

Comments: None.

7.14 Function: integrateODE()

```
int integrateODE(float ystart[], int nvar, float *x1, float x2, float eps,  
float h1, float hmin, void (*derivs)(float x, float *y, float *dy))
```

Integrate a set of ordinary, coupled first order differential equations from x_1 to x_2 . On return all the variables are set up, so that only a new value of x_2 has to be given to continue integration.

`ystart`: Input/Output. Contains the initial values for input and the calculated solution as output.

`nvar`: Input. The number of equations.

`x1`: Input/Output. The starting value. Becomes x_2 on return.

`x2`: Input. The final value.

`eps`: Input. The desired accuracy as discussed in chapter 16.2 of [1].

`h1`: Input. The initial step size.

`hmin`: Input. The smallest allowed step size.

`derivs`: Input. A function describing the ode's. `derivs(x, y, dy)` should set `dy` according to $dy = \frac{dy}{dx} = F(x, y)$.

Return value: Output. Returns 0 if there was no error, and an error code otherwise. These codes are described in Section 7.15.

Author: Serge Droz, droz@physics.uoguelph.ca

Comments: You have to use Numerical Recipe notation, i.e. the first element in an array `x` is `x[1]` and *not* `x[0]`. See the program `Lorenz` for an example.

7.15 Errors

Most routines return error codes (in addition to reporting them through the GRASP error mechanism) from the following list: `enum testmass_errors { kBhptNoError, kBhptCantOpenFile, kBhptOutOfMemory, kBhptUnknownMemory, kBhptNotEnoughPoints, kBhptCorruptFile, kBhptStepTooSmall, kBhptTooManySteps, kBhptNoDataRead, kBhptNoPhase, kBhptNoTime, kBhptFOutOfRange };`

`kBhptNoError`: No error occurred.

`kBhptCantOpenFile`: The requested file could not be opened.

`kBhptOutOfMemory`: Not enough memory to finish an operation.

`kBhptUnknownMemory`: An array was passed to a routine without any information about its size. You probably passed a `number_of_points` variable set to zero, but an `**X != NULL` to some routine.

`kBhptNotEnoughPoints`: There is not enough data available to finish the operation.

`kBhptCorruptFile`: The data file which was tried to be read into memory seems corrupt. This happens mostly with corrupt files for the modes A_{lm} .

`kBhptStepTooSmall`: An integration could not be finished because the minimum step size was reached.

`kBhptTooManySteps`: An integration could not be finished because too many steps are needed.

`kBhptNoDataRead`: The data to perform a given calculation has not been read into memory.

`kBhptNoPhase`: The phase $\Phi(f_0, v)$ has not been calculated.

`kBhptNoTime`: $t(V)$ has not been calculated.

`kBhptFOutOfRange`: The frequency requested is out of range.


```
files, so we can give NULL as filenames. x[0..NoPo-1] will contain all the
v - values. This routine sets up memory for the A_{lm}(v)'s and P(v)
internally. It will also calculate V(t) and save it. */
printf(" Reading data ... \n");
error = ReadData(NULL, NULL, &x, &NoPo);
if ( error ) return error;

/* We now have to calculate the phase function
Phi(f_0,v). This function already knows
how many points to calculate; the same number
as the number of datapoints. Since we want to plot
the wave function around ~ 100 Hz we set fo = 400.0 */
printf(" Calculating the phase ... \n");
error = calculate_testmass_phase(400.0, (m1+m2), &Phase);
if ( error ) printf("Error calculating the phase\n");

/* Uncomment the following code if you want to save Phi(f_0,v) */
/*
fp = fopen("Phase.dat", "w");
for (i = 0; i < NoPo; i++)
    fprintf(fp, "%f %f %20.18f\n", x[i], pow(x[i], 3.0)/(2.0*(m1+m2)*TSOLAR*Pi), Phase[i]);
fclose(fp);
*/

/* We're now ready to calculate the chirp itself. */
printf(" Calculating the chirp ... \n");

dt = Get_Duration(60.0, 785.0, m1, m2)/(1.0*NoOfWavePoints-1.0); /* Set the timestep in seconds */

testmass_chirp(m1, m2, theta, phi, Phase, 60.0, 785.0, &fstart, &fend,
              dt, &hplus, &hcross, &f, &NoOfWavePoints, 3, modes);

printf(" Calculated %d data points\n in the frequency intervall [%f, %f].\n",
       NoOfWavePoints, fstart, fend);
printf(" The chirp lasted %f seconds.\n", dt*NoOfWavePoints);
printf(" Writing data to disk. This might take a few seconds.\n");
fp = fopen("waveform.dat", "w");
for (i = 0; i < NoOfWavePoints; i++)
    fprintf(fp, "%f %f %f %f %f\n", i*dt, f[i], hplus[i],
          hcross[i], pow(f[i]*2.0*(m1+m2)*TSOLAR*M_PI, 1.0/3.0));
fclose(fp);

#ifdef __MACOS__
printf("Playing wave . . . . \n"); /* Play the wave */
error = PlayAudio(hplus, 1.0/dt, Noofcp-1);
#endif

Clean_Up_Memory(Phase); /* Clean up all the memory which was used internally. */
free(hplus); /* Get rid of the waveforms */
free(hcross);
printf("Goodbye. . . \n"); /* That's all folks. */
return error;
}
```

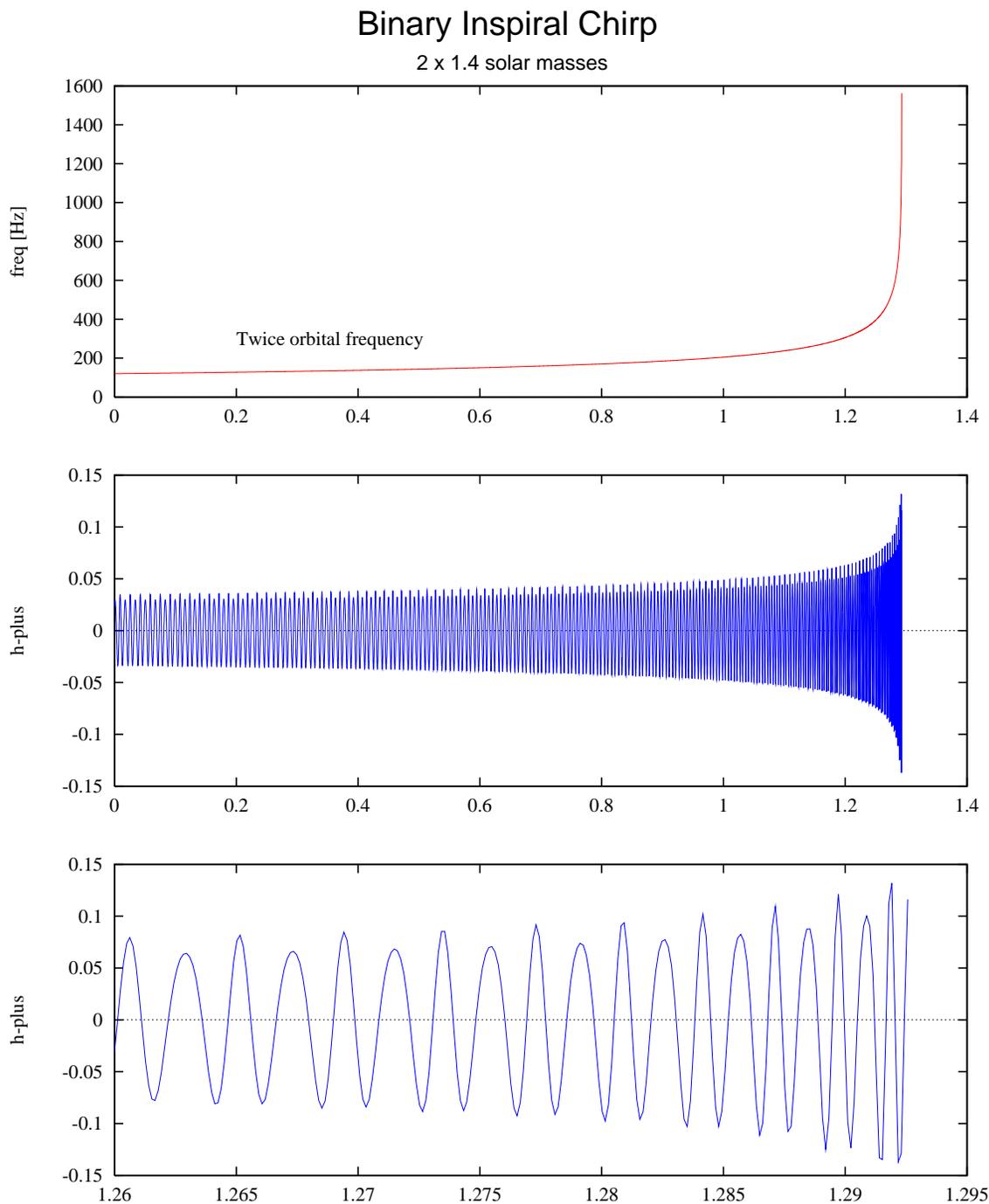


Figure 48: The zero-phase chirp waveform from a $2 \times 1.4M_{\odot}$ binary system, starting at an orbital frequency of 60 Hz. This waveform consists of the $l = 2$ and $l = 3$ modes. The top graph shows the frequency as function of time, and the middle graph shows the waveform. The bottom graph shows a 40-msec stretch near the final inspiral/plunge. Compare this to figure 25 in Section 6.7.

7.17 Example: lorenz program

This program illustrates the use of `integrateODE()` to solve a system of ODEs. Note that `integrateODE()` needs the Numerical Recipes library to run. The program solves the Lorenz equations. Invoke the program by typing `Lorenz s r b number_of_Points filename`, where s , t and b are parameters of the Lorenz equations.

```
/* Example lorenz
   Solve the Lorenz equations:

   dx/dt = s * (y - x),
   dy/dt = (r * x - y - xz),
   dz/dt = -b*z + xy,
   starting at the point (0.0, 1.0, 1.0).
   Try eg. lorenz 11.0 28.0 2.6666 2000 test.dat
   Author: S. Droz
   For more info see for example
   http://pineapple.apmaths.uwo.ca/~blair/lorenzintro.html
   Note that these equations are chaotic, and thus extremely
   susceptible towards numerical errors over long time spans.
   If you change the accuracy (eps below) by a factor of 10
   you might get a rather different looking picture.
*/
#include "grasp.h"

float s,r,b;

void lorenz(float x, float y[], float dy[])
/* Set up the system of ordinary DE. Since we use the NR
   integrator we use arrays going from 1..3, rather than from
   0..2. */
{
  dy[1] = s * (y[2] - y[1]);
  dy[2] = (r - y[3]) * y[1] - y[2];
  dy[3] = y[1] * y[2] - b*y[3];
}

int main(int argc, char *argv[])
{
  int NoPo,i;
  int error = 0; /* No errors yet */
  float y[3] = { 0, 1.0, 1.0 }; /* Initial point */
  float t = 0.0; /* Time always starts at 0 */
  float dt = 0.02; /* The time step used */
  FILE *fp;

  if (argc != 6) /* Do we have enough command line arguments? */
  {
    fprintf(stderr,"Usage: %s s r b number_of_Points filename\n",argv[0]);
    fprintf(stderr,"E.g. %s 11.0 28.0 2.6666 2000 test.dat\n",argv[0]);
    fprintf(stderr,"The numbers s, r and b are parameters of the Lorenz equations\n");
    fprintf(stderr,"See e.g. http://pineapple.apmaths.uwo.ca/~blair/lorenzintro.html for mor in");
    return 1;
  }
  /* get the command line arguments */
  s = atof(argv[1]);
```

```
r = atof(argv[2]);
b = atof(argv[3]);

NoPo = atoi(argv[4]);
fp = fopen(argv[5], "w");

/* Open the output file */
if (! fp ) { printf("File error.\n"); exit(1); }
fprintf(fp, "%10.8e %10.8e %10.8e\n", y[0], y[1], y[2]);

/* Now we start at t=0 and integrate for NoPo-1 time steps: */
for ( i = 1; i < NoPo; i++)
{
    /* So starting at t integrate y to t+dt, using the
       equations implemented in the function lorenz.
       We start with an initial step of dt/10 and go
       as low as dt*10^{-10}. We require an accuracy of at least 10^{-6},
       See the NR for a detailed explanation of what all these numbers mean. */
    error = integrateODE(y-1, 3, &t, t+dt, 1.0e-6, dt/10.0, dt*1.0e-10,
        &lorenz);
    if (error) break; /* We chicken out if something goes wrong. */
    fprintf(fp, "%10.8e %10.8e %10.8e\n", y[0], y[1], y[2]);
}
fclose(fp);
return error; /* Good bye */
}
```

7.18 Example: plot_ambig program

This program creates a scan of the ambiguity function.

To following is based on sections 6.14 and 9.7. Using the definition (9.7.2) for the scalar product $\langle a, b \rangle$ we can rewrite the expectation value (6.14.12) of the signal to noise ratio (SNR) ρ as

$$\langle \rho \rangle = 2 \frac{|\langle C, T_i \rangle_{t_0}|}{\sqrt{|T_i|}}, \quad (7.18.1)$$

where $|T_i| = \sqrt{\langle T_i, T_i \rangle}$. Here $C(t)$ is the signal (i.e. the chirp), and T_i is the i -th template. Obviously $\langle \rho \rangle_{t_0}$ is maximized if $T_i = C$ and $t_0 = 0$. We thus can rewrite equation (7.18.1) as

$$\langle \rho \rangle = \underbrace{\frac{|\langle C, T_i \rangle_{t_0}|}{\sqrt{|T_i||C|}}}_{\mathcal{A}_{it_0}} \langle \rho \rangle_{\max}.$$

The function \mathcal{A}_{it_0} gives the reduction of the SNR due to a nonoptimal template T_i . It is commonly called the *ambiguity function*. Since maximization over the parameter t_0 is trivially achieved by a FFT we often work with the *reduced ambiguity function*

$$\mathcal{A}_i = \max_{t_0} \mathcal{A}_{it_0}.$$

As was mentioned in section 6.14 every signal is a linear combination of two orthogonal modes T_0 and T_{90} (we suppress the index i for now), where $\langle T_0, T_{90} \rangle = 0$. We can filter for any linear combination by using the template

$$T = \frac{1}{\sqrt{2}} \left(\frac{T_0}{|T_0|} + i \frac{T_{90}}{|T_{90}|} \right).$$

Using T , the ambiguity function becomes

$$\mathcal{A}_i = \max_{t_0} \sqrt{\frac{\langle C, T_0 \rangle_{t_0}^2}{|T_0||C|} + \frac{\langle C, T_{90} \rangle_{t_0}^2}{|T_{90}||C|}}. \quad (7.18.2)$$

The sample program `plot_ambig` produces a file containing \mathcal{A}_i as a function of the chirp mass $\mathcal{M} = (m_1 m_2)^{3/5} (m_1 + m_2)^{-1/5}$ and the mass ratio $\eta = (m_1 m_2) (m_1 + m_2)^{-2}$. The templates are taken to be the 2 pN spin-less wave forms and the signal C is one of the modes calculated from perturbation theory. The output is saved to the file `scan.dat`.

```

/* plot_ambig.c
   Calculate a series of values of the ambiguity function,
   using 2 pN waveforms as templates and a mode calculated from
   black hole perturbation theory as signal.

   Author: S. Droz (droz at physics.uoguelph.ca)
*/

#include "grasp.h"

/* Prototypes: */
void realft(float *, unsigned long, int);
float norm(float* T, float* twice_inv_noise, int npoint);

float norm(float* That, float* twice_inv_noise, int npoint)

```

```
/* Calculate  $\int df / S(f) T(f) * T^*(f)$  or, in the notation
of the manual  $\langle T/Sh, T/Sh \rangle = \langle T, T \rangle$ . */
{
    int i, im, re;
    float real, imag, c=0;

    /* This loop is equivalent to (but faster than!)
       correlate(output0, That, That, twice_inv_noise, npoint);
       c=output0[0]; */

    for (i=1; i<npoint/2; i++) /* Neglect the DC and fc values */
    {
        im=(re=i+i)+1;
        real=That[re];
        imag=That[im];
        c+=twice_inv_noise[i]*(real*real+imag*imag);
    }
    return sqrt(c); /* Note that the 2 from 2/S compensates for the fact that
we only sum over positive frequencies. */
}

int main()
{
    int NoPo =0; /* Read in as many data points as possible */
    float *x = NULL; /* We let GRASP take care of all the memory */
    float *Phase = NULL; /* allocation. */
    float *hplus = NULL;
    float *hcross = NULL;
    float hNorm;
    int Npoints = 32768; /* 2^20 points */
    int NoOfWavePoints = 10000; /* The number of points we want saved */
    int NoOfPointsGen;
    float *f = NULL;
    float fend, dt;
    int error, i, j;
    FILE *fp;
    float m1 = 4.5; /* Mass of the first body in solar masses */
    float m2 = 4.5; /* Mass of the second body in solar masses */
    float eta, Mc, e, m, xx, yy;
    float fstart = 70.0; /* Starting ORBITAL frequency. */
    float theta = 1.2; /* Pick an angle */
    float phi = 0.0;
    float *pN0, *pN90, n0, n90, c0, c90, *output0, *output90;
    float SNR, var;
    int offset;
    float *twice_inv_noise;
    double *temp;
    float t_coal=0;
    float MaxAmb = 0.0;
    float Mx=0.0, My=0.0;

    /* Which modes should we include? (1 include, 0 omit) */
    /* m = -5 -4 -3 -2 -1 1 2 3 4 5 */
    int modes[28] = {
        1, 1, 1, 1, 1, /* 1 */
        1, 1, 1, 1, 1, 1, /* 1=3 */
        0, 0, 0, 0, 0, 0, 0, 0, /* 1=4 */
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }; /* 1=5 */
    /* 1=2 */
}
```

```
/* First we have to read in the data file. This will only work if you've
   set the environment variable GRASP_PARAMETERS. We just read in the default
   files, so we can give NULL as filenames. x[0..NoPo-1] will contain all the
   v - values. This routine sets up memory for the A_{lm}(v)'s and
   P(v) internally. It will also calculate V(t) and save it. */
printf(" Reading data ... \n");
error = ReadData(NULL, NULL, &x, &NoPo);
if ( error ) return error;

/* We now have to calculate the phase function
   Phi(f_0,v). This function already knows
   how many points to calculate; the same number
   as we've read datapoints. Since we are interested in frequencies
   of a couple of 100 Hz we set f_0 = 200.0 */
printf(" Calculating the phase ... \n");
error = calculate_testmass_phase(200.0, (m1+m2) ,&Phase);
if ( error ) printf("Error calculating the phase\n");

/* We're now ready to calculate the chirp itself. */
printf(" Calculating the chirp ... \n");
printf(" MaxF = %f -> T = %e\n",Get_Fmax(m1,m2),Get_Duration(fstart, Get_Fmax(m1,m2),m1,m2));
dt = Get_Duration(fstart, Get_Fmax(m1,m2),m1,m2)/(NoOfWavePoints-1); /* Set the timestep in seconds */
printf(" dt = %e\n", dt);
NoOfWavePoints = Npoints;

testmass_chirp(m1, m2, theta, phi , Phase, fstart ,Get_Fmax(m1,m2)-10, &fstart, &fend,
              dt, &hplus, &hcross, &f, &NoOfWavePoints, 3, modes);
Clean_Up_Memory(Phase); /* Clean up all the memory which was used internally. */
free(hcross); /* We don't need htimes. */
printf(" Calculated %d data points\n in the frequency intervall [%f, %f].\n",
       NoOfWavePoints,fstart, fend);
printf(" The chirp lasted %f seconds.\n",dt*NoOfWavePoints);
/* Zero out the remaining points */
clear(hplus + NoOfWavePoints, Npoints-NoOfWavePoints,1);

/* Get the spectral density */

twice_inv_noise = (float *)malloc((Npoints/2+1)*sizeof(float));
temp = (double *)malloc((Npoints/2+1)*sizeof(double));
if ( ! ( temp && twice_inv_noise ) ) return -1; /* Not enough memory */

noise_power("noise_40smooth.dat", Npoints/2, 1.0/(Npoints*dt), temp);
for (i=0; i< Npoints/2 ; i++) twice_inv_noise[i] = (float)(2.0e-31/temp[i]);
free(temp);

/* Allocate memory for the templates, etc. */
pN0 = (float *)malloc(Npoints*sizeof(float));
pN90 = (float *)malloc(Npoints*sizeof(float));
output0 = (float *)malloc(Npoints*sizeof(float));
output90 = (float *)malloc(Npoints*sizeof(float));
if ( ! ( pN0 && pN90 && output0 && output90 ) ) return -1; /* Not enough memory */

realft(hplus-1,Npoints,1); /* FFT the signal */
hNorm = norm(hplus, twice_inv_noise, Npoints); /* Get the signal's norm */

/* Now get ready to loop over the mass range. We use the chirp mass and
   mass ratio eta as parameters. */
```

```
Mc = pow(m1*m1*m1*m2*m2*m2/(m1+m2),1.0/5.0);
eta = m1*m2/pow(m1+m2,2);
printf(" Chirpmass Mc = %e Msun, eta = %e\n",Mc,eta);
fp = fopen("scan.dat","w");
for (i = 0; i<=50; i++)
{
  for (j = 0; j<=50;j++)
  {
    xx = (0.25 + j*(1.0 - 0.25)/50.0); /* Deviation from the 'true' value */
    yy = (1.00 + i*(1.3 - 1.0)/50.0);
    e = eta*xx;
    m = Mc*yy;
    m1 = 0.5*m*pow(e,-3.0/5.0)*(1-sqrt(1-4.0*e));
    m2 = 0.5*m*pow(e,-3.0/5.0)*(1+sqrt(1-4.0*e));
    /* Use make_filters to make the templates, then FFT and orthonormalize. */
    make_filters(m1, m2, pN0, pN90, 2.0*fstart, Npoints, 1.0/dt, &NoOfPointsGen,
      &t_coal, 2000, 4);
    realft(pN0-1,Npoints,1);
    realft(pN90-1,Npoints,1);
    orthonormalize(pN0,pN90, twice_inv_noise, Npoints,&n0,&n90);

    find_chirp(hplus,pN0,pN90,twice_inv_noise,n0,n90,output0, output90, Npoints, NoOfWav
      &offset, &SNR, &c0,&c90,&var);
    if ( SNR > MaxAmb)
    {
      MaxAmb = SNR;
      Mx = xx;
      My = yy;
    }
    fprintf(fp, "%e %e %e\n",xx,yy, SNR/hNorm); /* Save {\cal A} */
  }
  fprintf(fp, "\n");fflush(fp);
  printf(".");fflush(stdout);
}
fclose(fp);
MaxAmb /= hNorm;

/* Clean up the remaining memory and exit */
free(hplus); /* Get rid of the waveforms */
free(pN0);
free(pN90);
free(output0);
free(output90);
free(twice_inv_noise);
printf("\n The maximum of A_i in the scanned intervall was %4.1f%% and occured at eta*=%4.3f, Mc*=%4.3f\n");
printf("\nGoodbye. . .\n"); /* That's it folks. */
return error;
}
```

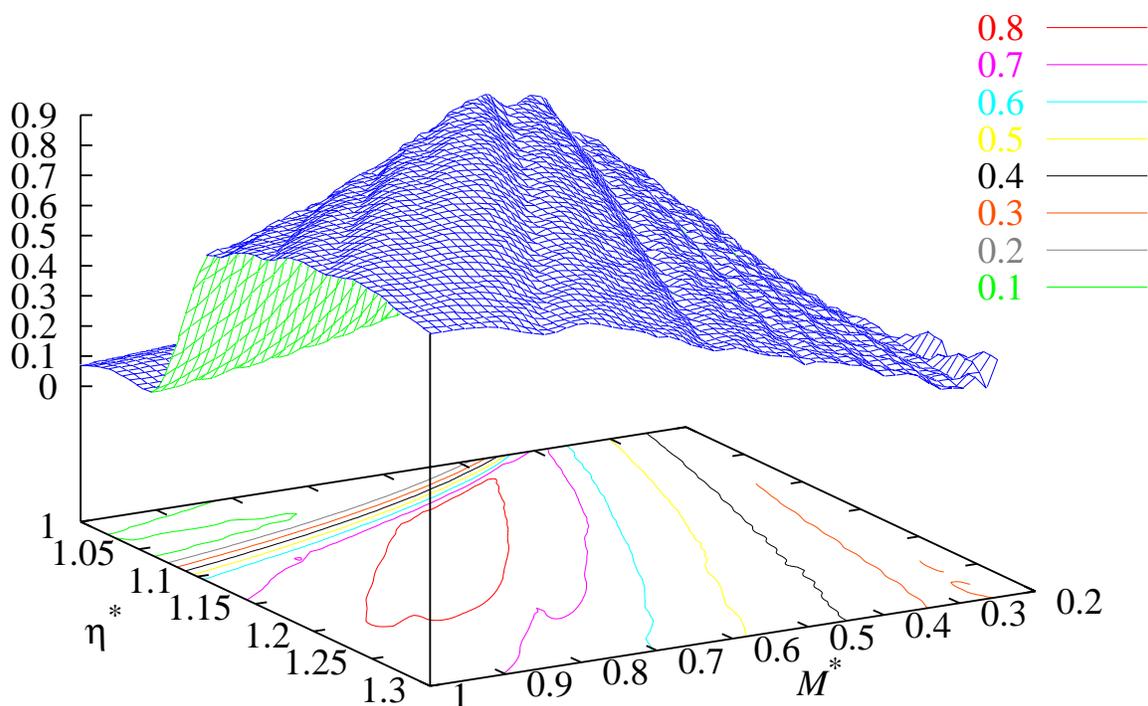


Figure 49: A contour plot of the reduced Ambiguity function \mathcal{A}_i . The axes are labeled by the relative deviations from the true values of $\eta = 0.25$ and the chirp mass $\mathcal{M} = 3.92M_\odot$ corresponding to a $m_1 = m_2 = 4.5M_\odot$ binary system. The Maximum value of 86.8% is attained at $\eta^* = 0.61\eta$ and $M^* = 1.084\mathcal{M}$.

8 GRASP Routines: Black hole ringdown

Stellar-sized black hole binaries are an important source of gravitational radiation for ground-based interferometric detectors. The radiation arises from three phases: the inspiral of the two black hole companions, the merger of these two companions to form a single black hole, and the ringdown of this initially distorted black hole to become a stationary Kerr black hole. The gravitational radiation of the black hole inspiral has been discussed in section 6; calculations of the late stages of inspiral, the merger, and the early stages of the ringdown have not yet been completed; the radiation produced in the late stages of black hole ringdown is the topic of this section.

At late times, the distorted black hole will be sufficiently “similar to” a stationary Kerr black hole that the distortion can be expanded in terms of “resonant modes” of the Kerr black hole. By “resonant modes” we refer to the eigenfunctions of the Teukolsky equation—which describes linear perturbations of the Kerr spacetime—with boundary conditions corresponding to purely ingoing radiation at the event horizon and purely outgoing radiation at large distances. These resonant modes are also called the quasinormal modes; they are described in the next subsection.

8.1 Quasinormal modes of black holes

Gravitational perturbations of the curvature of Kerr black holes can be described by two components of the Weyl tensor: Ψ_0 and Ψ_4 . Because these are components of the curvature tensor, they have dimensions of $[L^{-2}]$. Of particular interest is the quantity Ψ_4 since it is this term that is suitable for the study of outgoing waves in the radiative zone. The formalism for the study of perturbations of rotating black holes was developed originally by Teukolsky [30] who was able to separate the differential equation to obtain solutions of the form

$$(r - i\mu a)^4 \Psi_4 = e^{-i\omega t} {}_{-2}R_{\ell m}(r) {}_{-2}S_{\ell m}(\mu) e^{im\beta} \tag{8.1.1}$$

where ${}_{-2}R_{\ell m}(r)$ is a solution to a radial differential equation, and ${}_{-2}S_{\ell m}(\mu)$ is a spin-weighted spheroidal wave function (see [30], equations (4.9) and (4.10)). The black hole has mass M and *specific angular momentum* $a = cJ/M$ (which has dimensions of length) where J is the angular momentum of the spinning black hole. We shall often refer to the *dimensionless angular momentum parameter*, $\hat{a} = c^2 a/GM = c^3 J/GM^2$. For a Kerr black hole, \hat{a} must be between zero (Schwarzschild limit) and one (extreme Kerr limit). The observer of the perturbation is located at radius r , inclination $\mu = \cos \iota$, and azimuth β (see figure 50). The perturbation itself has the spheroidal eigenvalues ℓ and m , and has a (complex) frequency ω . The constants G and c are Newton’s gravitational constant and the speed of light.

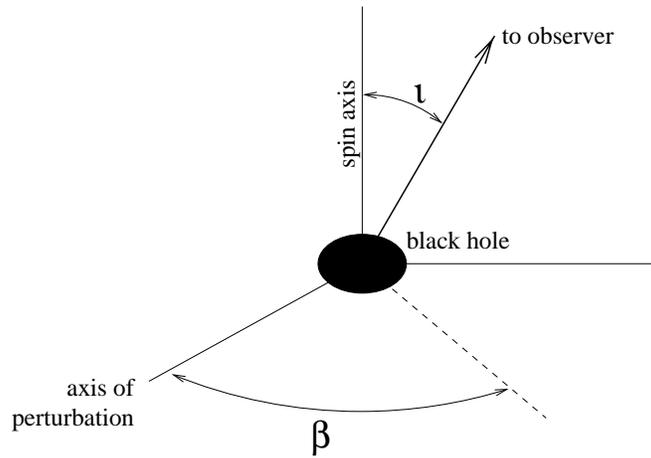


Figure 50: The polar angle, ι , and the azimuthal angle, β , of the observer relative to the spin axis of a black hole and the (somewhat arbitrary) axis of perturbation.

The important physical quantities for the study of the gravitational waves arising from black hole perturbations can be recovered from the field Ψ_4 . In particular, the “+” and “x” polarizations of the strain induced by the gravity waves are found by [30]

$$h_+ - ih_x = -\frac{2c^2}{|\omega|^2} \Psi_4 . \tag{8.1.2}$$

The quantity $h_+ = h_{\hat{i}\hat{i}}$ is the metric perturbation that represents the linear polarization state along $\mathbf{e}_{\hat{i}}$ and $\mathbf{e}_{\hat{\beta}}$, while the quantity $h_x = h_{\hat{i}\hat{\beta}}$ represents the linear polarization state along $\mathbf{e}_{\hat{i}} \pm \mathbf{e}_{\hat{\beta}}$. The power radiated towards the observer (per unit solid angle) is

$$\frac{d^2 E}{dt d\Omega} = \lim_{r \rightarrow \infty} \frac{c^7 r^2}{4\pi G |\omega|^2} |\Psi_4|^2 . \tag{8.1.3}$$

Thus, in order to compute the relevant information about gravitational waves emitted as perturbations to rotating black hole spacetimes, one needs to calculate the value of Ψ_4 at large radii from the black hole.

The quasinormal modes are resonant modes of the Teukolsky equation that describe purely outgoing radiation in the wave-zone and purely ingoing radiation at the event horizon. The quasi-normal modes are described by a spectrum of complex eigenvalues (which include the spectrum of eigenfrequencies ω_n), and eigenfunctions ${}_{-2}R_{\ell m}(r)$ and ${}_{-2}S_{\ell m}(\mu)$ for each value spheroidal mode ℓ and m . These eigenvalues and functions also depend on the mass and angular momentum of the black hole. We shall only consider the fundamental ($n = 0$) mode since the harmonics of this mode have shorter lifetimes. For the same reason, we are most interested in the quadrupole ($\ell = 2$ and $m = 2$) mode. The observer is assumed to be at a large distance; in this case, one can approximate the perturbation as follows:

$$\Psi_4 \approx \frac{A}{r} e^{-i\omega t_{\text{ret}}} {}_{-2}S_{\ell m}(\mu) e^{im\beta}. \quad (8.1.4)$$

Here $t_{\text{ret}} = t - r^*/c$ represents the retarded time, where r^* is a ‘‘tortoise’’ radial parameter. For large radii, the tortoise radius behaves as $r - r_+ \log(r/r_+)$ where r_+ is the ‘‘radius’’ of the black hole event horizon. Thus, we see that the tortoise radius is nearly equal to the distance of the objects surrounding the black hole, and we shall view it as the ‘‘distance to the black hole.’’ The parameter A represents the amplitude of the perturbation, which has the dimensions of $[L^{-1}]$.

Given the asymptotic form of the perturbation in equation 8.1.4, we can integrate equation 8.1.3 over the entire sphere and the interval $t_{\text{ret}} \in [0, \infty)$ to obtain an expression for the total energy radiated in terms of the amplitude A of the perturbation. Thus, we can characterize the amplitude by the total amount of energy emitted: $A^2 = 4Gc^{-7} E|\omega|^2 (-\text{Im } \omega)$. The gravitational waveform is found to be

$$h_+ - ih_\times \approx -\frac{4c}{r} \left(\frac{-\text{Im } \omega}{|\omega|^2} \right)^{1/2} \left(\frac{GE}{c^5} \right)^{1/2} e^{-i\omega t_{\text{ret}}} {}_{-2}S_{\ell m}(\mu) e^{im\beta}. \quad (8.1.5)$$

In order to simulate the quasinormal ringing of a black hole, it is necessary to determine the complex eigenvalues of the desired mode, and then to compute the spheroidal wave function $S_{\ell m}(\mu)$. The routines to perform these computations are discussed in the following sections.

Rather than computing the actual gravitational strain waveforms at the detector, the routines will calculate the quantity $H_+ - iH_\times = (c^2 r/GM_\odot)(h_+ - ih_\times)$; the normalization of these waveforms to the correct source distance is left to the calling routine. The distance normalization can be computed as follows:

$$\frac{c^2 r}{GM_\odot} = \frac{r}{T_\odot c} = \left(\frac{r}{1.4766 \text{ km}} \right) = 2.090 \times 10^{19} \left(\frac{r}{\text{Mpc}} \right). \quad (8.1.6)$$

where $T_\odot = 4.925491 \mu\text{s}$ is the mass of the sun expressed in seconds (see equation 6.0.2). It will be convenient to write the time dependence of the strain as the complex function $\mathcal{H}(t_{\text{ret}})$ so that $H_+ - iH_\times = \mathcal{H}(t_{\text{ret}}) {}_{-2}S_{\ell m}(\mu) e^{im\beta}$. The dimensionless eigenfrequency, $\hat{\omega} = GM\omega/c^3$, depends only on the mode and the dimensionless angular momentum of the black hole. In terms of this quantity, the function $\mathcal{H}(t_{\text{ret}})$ is

$$\mathcal{H}(t_{\text{ret}}) \approx -4\epsilon^{1/2} \frac{(-\text{Im } \hat{\omega})^{1/2}}{|\hat{\omega}|} \left(\frac{M}{M_\odot} \right) \exp \left[-i\hat{\omega} \left(\frac{t_{\text{ret}}}{T_\odot} \right) \left(\frac{M}{M_\odot} \right)^{-1} \right] \quad (8.1.7)$$

where ϵ is the fractional mass loss due to the radiation in the excited quasinormal mode.

8.2 Function: `qn_eigenvalues()`

```
void qn_eigenvalues(float eigenvalues[], float a, int s, int l, int m)
```

This routine computes the eigenvalues associated with the spheroidal and radial wave functions for a specified quasinormal mode. The arguments are:

`eigenvalues`: Output. An array, `eigenvalues[0..3]`, which contains, on output, the real and imaginary parts of the eigenvalues $\hat{\omega}$ and A (see below) as follows: `eigenvalues[0] = Re $\hat{\omega}$` , `eigenvalues[1] = Im $\hat{\omega}$` , `eigenvalues[2] = Re A` , and `eigenvalues[3] = Im A` .

`a`: Input. The dimensionless angular momentum parameter of the Kerr black hole, $|\hat{a}| \leq 1$, which is negative if the black hole is spinning clockwise about the $\iota = 0$ axis (see figure 50).

`s`: Input. The integer-valued spin-weight s , which should be set to 0 for a scalar perturbation (e.g., a scalar field perturbation), ± 1 for a vector perturbation (e.g., an electromagnetic field perturbation), or ± 2 for a spin two perturbation (e.g., a gravitational perturbation).

`l`: Input. The mode integer $l \geq |s|$.

`m`: Input. The mode integer $|m| \leq l$.

For a Kerr black hole of a given dimensionless angular momentum parameter, \hat{a} , with a perturbation of spin-weight s and mode ℓ and m , there is a spectrum of quasinormal modes which are specified by the eigenvalues $\hat{\omega}_n$ and A_n . As discussed in the previous subsection, the eigenvalue $\hat{\omega}_n$ is associated with the separation of the time dependence of the perturbation, and it specifies the frequency and damping time of the radiation from the perturbation. The additional complex eigenvalue A_n results from the separation of the radial and azimuthal dependence into the spheroidal and radial wave functions. Both of these eigenvalues will be necessary for the computation of the spheroidal wave function (below).

The routine `qn_eigenvalues()` can be used to compute the eigenvalues of the fundamental ($n = 0$) mode. To convert the dimensionless eigenvalue $\hat{\omega}$ to the (complex) frequency of the ringdown of a Kerr black hole of mass M , one simply computes $\omega = c^3 \hat{\omega} / GM$. The eigenfrequency is computed using the method of Leaver [27]. Note that Leaver adopts units in which $2M = 1$, so one finds that $\hat{\omega} = \frac{1}{2} \omega_{\text{Leaver}}$ and $\hat{a} = 2a_{\text{Leaver}}$ in our notation. The eigenvalues satisfy the following symmetry: if $\rho_m = -i\hat{\omega}_m$ and A_m are the eigenvalues for an azimuthal index m , then $\rho_{-m} = \rho_m^*$ and $A_{-m} = A_m^*$ are the eigenvalues for the azimuthal index $-m$.

Author: Jolien Creighton, jolien@tapir.caltech.edu

Comment: For simplicity, we require that the spin-weight number, s , be an integer. Thus, the spinor perturbations χ_0 and χ_1 , associated with $s = \pm \frac{1}{2}$ respectively [30], are not allowed.

8.3 Example: eigenvalues program

This example uses the function `qn_eigenvalues()` to compute the eigenvalues ${}_s\hat{\omega}_{\ell m}$ and ${}_sA_{\ell m}$ for the s spin-weighted quasinormal mode specified by ℓ and m , and for a range of values of the dimensionless angular momentum parameter, \hat{a} . To invoke the program, type:

```
eigenvalues s l m
```

for the desired (integer) values of s , ℓ , and m . Make sure that $\ell \geq |s|$ and $0 \leq m \leq \ell$ (the eigenvalues for negative values of m can be inferred from the symmetries discussed in subsection 8.2). The output of the program is five columns of data: the first column is the value of \hat{a} running from just greater than -1 to just less than 1 (or between 0 and 1 if $m = 0$), the second and third columns are the real and imaginary parts of the eigenfrequency $\hat{\omega}$, and the fourth and fifth columns are the real and imaginary parts of the angular separation eigenvalue A . For the values of $\hat{a} < 0$, the eigenvalues correspond to the mode with azimuthal index $-m$ so that the real part of the eigenfrequency is positive. A plot of the eigenfrequency output of the program `eigenvalues` for several runs with $s = -2$ is shown in figure 51. The blue curves in figure 51 can be compared to figure 5 of reference [28] keeping in mind the conversion factors between Leaver's convention (which is also used in [28]) and the convention used here (see subsection 8.2).

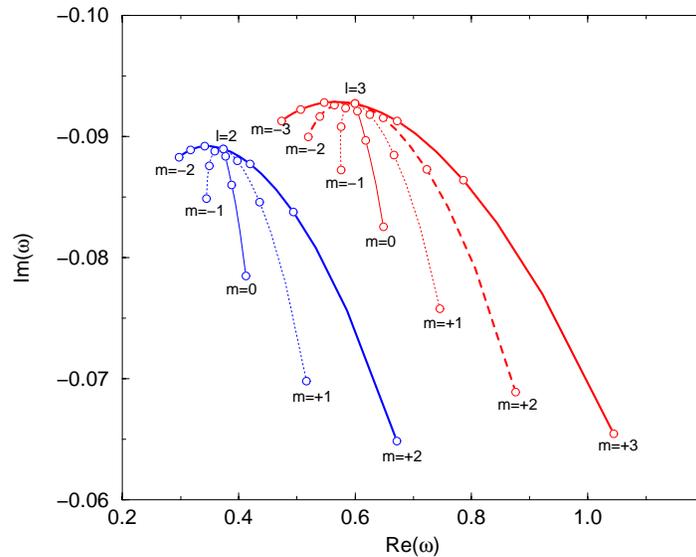


Figure 51: The real and imaginary parts of the eigenfrequencies, $\hat{\omega}$, as computed by the program `eigenvalues` with $s = -2$. Each curve corresponds to a range of values of \hat{a} from -0.9 (left) to $+0.9$ (right) for a single mode ℓ and $|m|$. The open circles are placed at the values $\hat{a} = -0.9, -0.6, -0.3, 0, +0.3, +0.6,$ and $+0.9$ except when $m = 0$ in which case there are no negative values of \hat{a} plotted. The blue curves correspond to the $\ell = 2$ modes and the red curves correspond to the $\ell = 3$ modes.

```
/* GRASP: Copyright 1997,1998 Bruce Allen */
#include "grasp.h"

int main(int argc,const char *argv[])
{
    float a,da=0.1,eigen[4];
    int s,l,m;

    /* process the command line arguments */
    if (argc==4) { /* correct number of arguments */
        s = atoi(argv[1]);
        l = atoi(argv[2]);
        m = atoi(argv[3]);
    } else { /* incorrect number of arguments */
        fprintf(stderr,"usage: qn_eigen_values s l m\n");
        return 1;
    }

    /* scan through the range of a */
    for (a=1-da;a>-1;a-=da) {
        /* compute the eigenvalues */
        if (a<0) {
            if (m==0) break;
            qn_eigenvalues(eigen,a,s,l,-m);
        } else {
            qn_eigenvalues(eigen,a,s,l,m);
        }
        /* print the eigenvalues */
        printf("%f\t%f\t%f\t%f\n",a,eigen[0],eigen[1],eigen[2],eigen[3]);
    }

    return 0;
}
```

Author: Jolien Creighton, jolien@tapir.caltech.edu

8.4 Function: `sw_spheroid()`

```
void sw_spheroid(float *re, float *im, float mu, int reset,
                float a, int s, int l, int m, float eigenvalues[])
```

This routine computes the spin-weighted spheroidal wave function ${}_sS_{\ell m}(\mu)$. The arguments are:

`re`: Output. The real part of the spin-weighted spheroidal wave function.

`im`: Output. The imaginary part of the spin-weighted spheroidal wave function.

`mu`: Input. The independent variable, $\mu = \cos \iota$ with ι being a polar angle, of the spin-weighted spheroidal wave function; $-1 < \mu < 1$.

`reset`: Input. A flag that indicates that the function should reset (`reset = 1`) the internally stored normalization of the spin-weighted spheroidal wave function. The reset flag should be set if any of the following five arguments are changed between calls; otherwise, set `reset = 0` so that the routine does not recompute the normalization.

`a`: Input. The dimensionless angular momentum parameter, $-1 < a < 1$, of the Kerr black hole for which the spin-weighted spheroidal wave function is associated.

`s`: Input. The integer-valued spin-weight s , which should be set to 0 for a scalar perturbation (e.g., a scalar field perturbation), ± 1 for a vector perturbation (e.g., an electromagnetic field perturbation), or ± 2 for a spin two perturbation (e.g., a gravitational perturbation).

`l`: Input. The mode integer $l \geq |s|$.

`m`: Input. The mode integer $|m| \leq l$.

`eigenvalues`: Input. An array, `eigenvalues[0..3]`, which contains the real and imaginary parts of the eigenvalues $\hat{\omega}$ and A (see below) as follows: `eigenvalues[0] = Re $\hat{\omega}$` , `eigenvalues[1] = Im $\hat{\omega}$` , `eigenvalues[2] = Re A` , and `eigenvalues[3] = Im A` . These may be calculated for a quasinormal mode using the routine `qn_eigenvalues()`.

The spin-weighted spheroidal wave function is also computed using the method of Leaver [27]. We have adopted the following normalization criteria for the spin-weighted spheroidal wave functions ${}_sS_{\ell m}(\mu)$. First, the angle-averaged value of the squared modulus of ${}_sS_{\ell m}(\mu)$ is unity: $\int_{-1}^1 |{}_sS_{\ell m}(\mu)|^2 d\mu = 1$. Second, the complex phase is partially fixed by the requirement that ${}_sS_{\ell m}(0)$ is real. Finally, the sign is set to be $(-)^{\ell - \max(m, s)}$ for the real part in the limit that $\mu \rightarrow -1$ in order to agree with the sign of the spin-weighted spherical harmonics ${}_sY_{\ell m}(\mu, 0)$ (see [26]).

It is sufficient to compute the spin-weighted spheroidal wave functions with $s < 0$ and $a\omega = \hat{a}\hat{\omega} \geq 0$ because of the following symmetries [29]:

$$-{}_sS_{\ell m}(\mu, a\omega) = {}_sS_{\ell m}(-\mu, a\omega) \quad \text{with} \quad -{}_sE_{\ell m}(a\omega) = {}_sE_{\ell m}(a\omega) \quad (8.4.1)$$

and

$${}_sS_{\ell m}(\mu, -a\omega) = {}_sS_{\ell, -m}(-\mu, a\omega) \quad \text{with} \quad {}_sE_{\ell m}(-a\omega) = {}_sE_{\ell, -m}(a\omega) \quad (8.4.2)$$

where ${}_sE_{\ell m} = {}_sA_{\ell m} + s(s+1)$.

Author: Jolien Creighton, jolien@tapir.caltech.edu

8.5 Example: spherical program

The program `spherical` is an example implementation of the routine `sw_spheroid()` to compute the standard spin-weighted spherical harmonics [26]. The program also computes these functions using equation (3.1) of [26] for comparison. According to the normalization convention stated in subsection 8.4, the relationship between the spin-weighted spheroidal harmonics and the spin-weighted spherical harmonics is:

$${}_sY_{\ell m}(\theta, \phi) = (2\pi)^{-1/2} {}_sS_{\ell m}(\cos \theta)e^{im\phi} \quad (8.5.1)$$

with $a\omega = 0$ and $A = (\ell - s)(\ell + s + 1)$.

To invoke the program, type:

```
spherical s l m
```

for the desired (integer) values of s , ℓ , and m ($\ell \geq |s|$ and $|m| \leq \ell$). The output is three columns of data: the first column is the independent variable μ between -1 and $+1$, the second column is the value of $(2\pi)^{-1/2} {}_sS_{\ell m}(\mu)$, and the third column is the value of ${}_sY_{\ell m}(\mu, 0)$ as computed from equation (3.1) of [26]. A comparison of the results produced by the program `spherical` for $\ell = m = -s = 2$ with the exact values of ${}_{-2}Y_{22}(\mu, 0) = (5/64\pi)^{1/2}(1 + \mu)^2$ is shown in table 7.

μ	Goldberg	<code>sw_spheroid()</code>	exact
-0.99	1.576955×10^{-5}	1.576955×10^{-5}	1.576958×10^{-5}
-0.95	3.942387×10^{-4}	3.942387×10^{-4}	3.942395×10^{-4}
-0.75	9.855968×10^{-3}	9.855967×10^{-3}	9.855986×10^{-3}
-0.55	3.193334×10^{-2}	3.193333×10^{-2}	3.193340×10^{-2}
-0.35	6.662639×10^{-2}	6.662639×10^{-2}	6.663647×10^{-2}
-0.15	1.139351×10^{-1}	1.139351×10^{-1}	1.139352×10^{-1}
+0.15	2.085525×10^{-1}	2.085525×10^{-1}	2.085527×10^{-1}
+0.35	2.874004×10^{-1}	2.874005×10^{-1}	2.874006×10^{-1}
+0.55	3.788640×10^{-1}	3.788639×10^{-1}	3.788641×10^{-1}
+0.75	4.829430×10^{-1}	4.829430×10^{-1}	4.829433×10^{-1}
+0.95	5.996378×10^{-1}	5.996379×10^{-1}	5.996382×10^{-1}
+0.99	6.244906×10^{-1}	6.244906×10^{-1}	6.244911×10^{-1}

Table 7: A comparison of the values of the spin-weighted spherical harmonic ${}_{-2}Y_{22}(\mu, 0)$ calculated by equation (3.1) of Goldberg [26], the values of $(2\pi)^{-1/2} {}_{-2}S_{22}(\mu)$ using routine `sw_spheroid()`, and the values of the exact result $(5/64\pi)^{1/2}(1 + \mu)^2$. The three methods give excellent agreement.

```
/* GRASP: Copyright 1997,1998 Bruce Allen */
#include "grasp.h"

#define TWOPI 6.28318530718
#define FOURPI 12.5663706144
static int imaxarg1,imaxarg2;
#define IMAX(a,b) (imaxarg1=(a),imaxarg2=(b),(imaxarg1) > (imaxarg2) ?\
    (imaxarg1) : (imaxarg2))
static int iminarg1,iminarg2;
#define IMIN(a,b) (iminarg1=(a),iminarg2=(b),(iminarg1) < (iminarg2) ?\
    (iminarg1) : (iminarg2))

float sw_spherical(float mu, int s, int l, int m)
/* Computes the spin-weighted spherical harmonic (with phi=0) using
   equation (3.1) of Goldberg et al (1967). */
{
    float factrl(int);
    float bico(int, int);
    float sum,coef,x;
    int sign,r,rmin,rmax;

    if (mu==-1.0) {
        fprintf(stderr,"error in sw_spherical(): division by zero");
        return 0;
    } else {
        x = (1 + mu)/(1 - mu);
    }
    coef = factrl(l+m)*factrl(l-m)*(2*l+1)/(factrl(l-s)*factrl(l+s)*FOURPI);
    rmin = IMAX(0,m-s);
    rmax = IMIN(l-s,l+m);
    sum = 0;
    for (r=rmin;r<=rmax;r++) {
        (((l-r+s)%2)==0) ? (sign = 1) : (sign = -1);
        sum += sign*bico(l-s,r)*bico(l+s,r+s-m)*pow(x,0.5*(2*r+s-m));
    }
    sum *= sqrt(coef)*pow(0.5*(1-mu),l);

    return sum;
}

int main(int argc, char *argv[])
{
    float Sre,Sim,Y,norm=1.0/sqrt(TWOPI),mu=0,dmu=0.02;
    float eigenvalues[4];
    int s,l,m;

    /* process arguments */
    if (argc==4) { /* correct number of arguments */
        s = atoi(argv[1]);
        l = atoi(argv[2]);
        m = atoi(argv[3]);
    } else { /* incorrect number of arguments */
        fprintf(stderr,"usage: spherical s l m\n");
        return 1;
    }

    /* set the eigenvalues to produce spin-weighted spherical harmonics */
    eigenvalues[0] = eigenvalues[1] = eigenvalues[3] = 0;
```

```
eigenvalues[2] = (1 - s)*(1 + s + 1);

/* reset the normalization */
sw_spheroid(&Sre,&Sim,mu,1,0.0,s,l,m,eigenvalues);

for (mu=-1+0.5*dmu;mu<1;mu+=dmu) {
    /* compute the spin-weighted spheroidal harmonic */
    sw_spheroid(&Sre,&Sim,mu,0,0.0,s,l,m,eigenvalues);
    /* compute the spin-weighted spherical harmonic */
    Y = sw_spherical(mu,s,l,m);
    /* print results with correct normalization for the spheroidal harmonic */
    printf("%e\t%e\t%e\n",mu,norm*Sre,Y);
}

return 0;
}
```

Author: Jolien Creighton, jolien@tapir.caltech.edu

8.6 Example: spheroid program

This is a second implementation of the function `sw_spheroid()` which is used to compute the spin-weighted spheroidal wave function associated with a quasinormal ringdown mode of a Kerr black hole with a certain (specified in the code) dimensionless angular momentum parameter. To invoke the program, type:

```
spheroid s l m
```

for the desired (integer) values of s , ℓ , and m ($\ell \geq |s|$ and $|m| \leq \ell$) of the desired mode. The output is three columns of data: the first column is the independent variable μ between -1 and $+1$, the second column is the value of the real part of ${}_sS_{\ell m}(\mu)$, and the third column is the value of the imaginary part of ${}_sS_{\ell m}(\mu)$. Figure 52 depicts the output for the spin-weighted spheroidal wave function ${}_{-2}S_{22}(\mu)$.

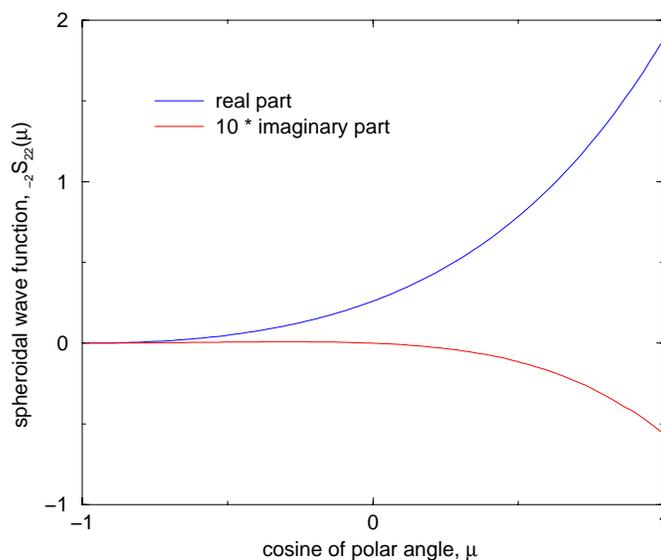


Figure 52: A plot of the real and imaginary parts of the $\ell = m = -s = 2$ spin-weighted spheroidal wave function, ${}_{-2}S_{22}(\mu)$, associated with a black hole with dimensionless angular momentum parameter $\hat{a} = 0.98$. The imaginary part is scaled by a factor of ten.

```
/* GRASP: Copyright 1997,1998 Bruce Allen */
#include "grasp.h"

#define SPIN 0.98 /* the dimensionless angular momentum parameter */

int main(int argc, char *argv[])
{
    float re,im,mu=0,dmu=0.02,a=SPIN;
    float eigenvalues[4];
    int s,l,m;

    /* process arguments */
    if (argc==4) { /* correct number of arguments */
        s = atoi(argv[1]);
        l = atoi(argv[2]);
        m = atoi(argv[3]);
    } else { /* incorrect number of arguments */
        fprintf(stderr,"usage: spheroid s l m\n");
        return 1;
    }

    /* get the eigenvalues for the appropriate quasinormal mode */
    qn_eigenvalues(eigenvalues,a,s,l,m);

    /* reset the normalization */
    sw_spheroid(&re,&im,mu,l,a,s,l,m,eigenvalues);

    for (mu=-1+0.5*dmu;mu<1;mu+=dmu) {
        /* compute the spin-weighted spheroidal harmonic */
        sw_spheroid(&re,&im,mu,0,0.0,s,l,m,eigenvalues);
        /* print results */
        printf("%e\t%e\t%e\n",mu,re,im);
    }

    return 0;
}
```

Author: Jolien Creighton, jolien@tapir.caltech.edu

8.7 Function: `qn_ring()`

```
int qn_ring(float iota, float beta,
           float eps, float M, float a, int l, int m,
           float dt, float atten, int max,
           float **plusPtr, float **crossPtr)
```

This routine is used to compute the “+” and “×” polarizations of the gravitational waveform, $H(t_{\text{ret}})$, produced by a black hole ringdown at a distance $GM_{\odot}/c^2 = T_{\odot}c \simeq 1.4766$ km. To obtain the waveforms at a distance r , multiply the result by $GM_{\odot}/c^2 r = T_{\odot}c/r$. The arguments are:

`iota`: Input. The polar angle (inclination), ι (in radians), of the sky position of the observer with respect to the (positive) spin axis of the black hole, $0 \leq \text{iota} \leq \pi$.

`beta`: Input. The azimuth, β (in radians), of the sky position of the observer with respect to the axis of the perturbation at the start time. ($0 \leq \text{beta} \leq 2\pi$.)

`eps`: Input. The fraction of the total mass lost in gravitational radiation from the particular mode. ($0 < \text{eps} \ll 1$.)

`M`: Input. The mass of the black hole in solar masses.

`a`: Input. The dimensionless angular momentum parameter of the Kerr black hole, $|\hat{a}| \leq 1$, which is negative if the black hole is spinning clockwise about the $\iota = 0$ axis (see figure 50).

`l`: Input. The mode integer ℓ . ($1 \geq 2$)

`m`: Input. The mode integer m . ($|m| \leq 1$)

`dt`: Input. The time interval, in seconds, between successive data points in the returned waveforms.

`atten`: Input. The attenuation level, in dB, at which the routine will terminate calculation of the waveforms. I.e., the routine will terminate when the amplitude, $A = A_0 \exp(-\text{Im} \omega t_{\text{ret}})$, falls below the level $A_{\text{cutoff}} = A_0 \text{alog}_{10}(-0.1 \times \text{atten})$.

`max`: Input. The maximum number of data points to be returned in the waveforms.

`plusPtr`: Input/Output. A pointer to an array which, on return, contains the waveform H_+ sampled at intervals `dt`. If the array has the value `NULL` on input, the routine allocates an amount of memory to `*plusPtr` to hold `max` elements.

`crossPtr`: Input/Output. A pointer to an array which, on return, contains the waveform H_{\times} sampled at intervals `dt`. If the array has the value `NULL` on input, the routine allocates an amount of memory to `*crossPtr` to hold `max` elements.

The routine `qn_ring()` returns the number of data points that were written to the arrays `(*plusPtr)[]` and `(*crossPtr)[]`; this is either the number specified by the input parameter `max` or the number of points computed when the waveform was attenuated by the threshold `atten`. The eigenvalues are obtained from the function `qn_eigenvalues()`. The waveform is then computed using $H_+ - iH_{\times} = \mathcal{H}(t_{\text{ret}})_{-2} S_{\ell m}(\mu) e^{im\beta}$ with $\mathcal{H}(t_{\text{ret}})$ given by equation (8.1.7). The spheroidal wave function is obtained from the function `sw_spheroid()`.

Author: Jolien Creighton, jolien@tapir.caltech.edu

8.8 Example: ringdown program

This example uses the function `qn_ring()` to compute the black hole quasinormal ringdown waveform for a preset mode and inclination. The waveform as a function of time is written to standard output in three columns: the time, the plus polarization, and the cross polarization. A Plot of the quasinormal ringdown waveform data is shown in figure 53.

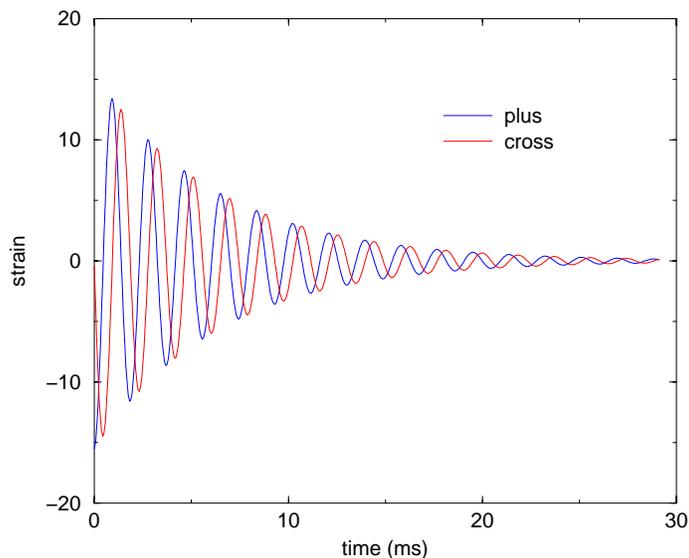


Figure 53: A plot of the plus and cross polarizations of the gravitational wave strain, at a (unphysical!) distance $GM_{\odot}/c^2 = T_{\odot}c \simeq 1.4766$ km, for the fundamental $\ell = m = 2$ mode of a black hole with mass $M = 50M_{\odot}$, dimensionless angular momentum parameter 0.98, and fractional mass loss $\epsilon = 0.03$, with inclination and azimuth $\iota = 0$ and $\beta = 0$. The data was produced by the program `ringdown`.

```
/* GRASP: Copyright 1997,1998 Bruce Allen */
#include "grasp.h"

#define IOTA 0.0          /* inclination (radians) */
#define BETA 0.0         /* azimuth (radians) */
#define EPS 0.03         /* fractional mass loss */
#define MASS 50.0        /* mass (solar masses) */
#define SPIN 0.98        /* specific angular momentum */
#define MODE_L 2         /* mode integer l */
#define MODE_M 2         /* mode integer m */
#define SRATE 16000.0    /* sampling rate (Hz) */
#define ATTEN 20.0       /* attenuation level (dB) */
#define MAX 1024         /* max number of points in waveform */

int main()
{
    float *plus,*cross,t,dt=1/SRATE;
    int i,n;

    /* set arrays to NULL so that memory is allocated in called routines */
    plus = cross = NULL;

    /* generate the waveform function data */
    n = qn_ring(IOTA,BETA,EPS,MASS,SPIN,MODE_L,MODE_M,dt,ATTEN,MAX,&plus,&cross);

    /* output the data */
    for (i=0,t=0;i<n;i++,t+=dt) printf("%e\t%e\t%e\n",t,plus[i],cross[i]);

    return 0;
}
```

Author: Jolien Creighton, jolien@tapir.caltech.edu

8.9 Function: `qn_qring()`

```
int qn_qring(float psi0, float eps, float M, float a,
            float dt, float atten, int max, float **strainPtr)
```

The routine `qn_qring()` is a quick ringdown generator which constructs a damped sinusoid with a frequency and quality approximately equal to that of the $\ell = m = 2$ quasinormal mode of a Kerr black hole and an amplitude approximately equal to angle-averaged strain expected for black hole radiation at a distance $GM_{\odot}/c^2 = T_{\odot}c \simeq 1.4766$ km. To obtain the waveforms at a distance r , multiply the result by $GM_{\odot}/c^2 r = T_{\odot}c/r$. The arguments to the routine are:

`psi0`: Input. The initial phase (in radians) of the waveform (see below).

`eps`: Input. The fractional mass loss in quadrupolar ($\ell = m = 2$) radiation. ($0 < \text{eps} \ll 1$.)

`M`: Input. The mass of the black hole in solar masses.

`a`: Input. The dimensionless angular momentum parameter of the Kerr black hole, $|\hat{a}| \leq 1$, which is negative if the black hole is spinning clockwise about the $\iota = 0$ axis (see figure 50).

`dt`: Input. The time interval, in seconds, between successive data points in the returned waveform.

`atten`: Input: The attenuation level, in dB, at which the routine will terminate calculation of the waveforms.

`max`: Input. The maximum number of data points to be returned in the waveforms.

`strainPtr`: Input/Output. A pointer to an array which, on return, contains the angle-averaged waveform sampled at intervals `dt`. If the array has the value NULL on input, the routine allocates an amount of memory to `*strainPtr` to hold `max` elements.

The routine `qn_qring()` returns the number of data points that were written to the array `(*strainPtr)[]`; this is either the number specified by the input parameter `max` or the number of points computed when the waveform was attenuated by the threshold `atten`. The array contains the *angle averaged waveform*

$$H_{\text{ave}}(t_{\text{ret}}) = \frac{1}{\sqrt{5}} \text{Re} [\mathcal{H}(t_{\text{ret}}) e^{i\psi_0}], \quad (8.9.1)$$

where $\mathcal{H}(t_{\text{ret}})$ is given by equation (8.1.7), sampled at time intervals dt . The constant ψ_0 defines the initial phase of the waveform. The amplitude factor is set by the following argument: The gravitational strain (at a distance $GM_{\odot}/c^2 = T_{\odot}c \simeq 1.4766$ km) that would be observed by an interferometer is given by $H(t_{\text{ret}}) = F_+(\theta, \phi, \psi)H_+(t_{\text{ret}}, \iota, \beta) + F_{\times}(\theta, \phi, \psi)H_{\times}(t_{\text{ret}}, \iota, \beta)$ where F_+ and F_{\times} represent the antenna patterns of the interferometer. When averaged over θ, ϕ , and ψ , one finds $\langle F_+^2 \rangle = \langle F_{\times}^2 \rangle = \frac{1}{5}$ and $\langle F_+ F_{\times} \rangle = 0$. Thus,

$$\begin{aligned} \langle H^2(t_{\text{ret}}) \rangle_{\theta, \phi, \psi, \iota, \beta} &= \frac{1}{5} \langle H_+^2(t_{\text{ret}}, \iota, \beta) + H_{\times}^2(t_{\text{ret}}, \iota, \beta) \rangle_{\iota, \beta} \\ &= \frac{1}{5} \langle |(H_+ - iH_{\times})(t_{\text{ret}}, \iota, \beta)|^2 \rangle_{\iota, \beta} \\ &= \frac{1}{10} |\mathcal{H}(t_{\text{ret}})|^2 \\ &\approx \overline{H_{\text{ave}}^2} \end{aligned} \quad (8.9.2)$$

where the overbar indicates a time average over a single cycle; approximate equality becomes exact in the limit of a high quality ringdown. It is in this sense that the quantity $H_{\text{ave}}(t_{\text{ret}})$ can be viewed as an angle-averaged waveform.

Rather than compute the eigenfrequency using the routine `qn_eigenvalues()`, this routine uses the analytic fits to the eigenfrequency found by Echeverria [25]. These expressions are:

$$\hat{\omega} \simeq f(\hat{a})(1 - \frac{1}{4}ig(\hat{a})) \quad (8.9.3)$$

with

$$f(\hat{a}) = 1 - 0.63(1 - \hat{a})^{3/10} \quad (8.9.4)$$

$$g(\hat{a}) = (1 - \hat{a})^{9/20}. \quad (8.9.5)$$

Author: Jolien Creighton, jolien@tapir.caltech.edu

Comments: Since this routine does not need to compute the spheroidal wave function and uses an analytic approximation to the eigenfrequency, it is much simpler than the routine `qn_ring()`. The approximate eigenfrequencies are typically accurate to within $\sim 5\%$, so this routine is to be preferred when computing quadrupolar ($\ell = m = 2$) quasinormal waveforms unless accuracy is critical.

8.10 Function: `qn_filter()`

```
int qn_filter(float freq, float qual,  
             float dt, float atten, int max, float **filterPtr)
```

Quasinormal ringdown waveforms are characterized by two parameters: the central frequency of the waveform, and the *quality* of the waveform. The parameter space is most easily described in terms of these variables (rather than the mass and the angular momentum of the corresponding black hole), so it is useful to construct filters for quasinormal ringdown waveform searches in terms of the frequency and quality of the waveform. This routine constructs such a filter, with a specified frequency and quality. The routine returns the number of filter elements computed before a specified attenuation level was reached. The arguments are:

`freq`: Input. The central frequency, in Hertz, of the ringdown filter.

`qual`: Input. The quality of the ringdown filter.

`dt`: Input. The time interval, in seconds, between successive data points in the returned waveform.

`atten`: Input. The attenuation level, in dB, at which the routine will terminate calculation of the waveforms.

`max`: Input. The maximum number of data points to be returned in the waveforms.

`filterPtr`: Input/Output. A pointer to an array which, on return, contains the filter sampled at intervals `dt`. If the array has the value `NULL` on input, the routine allocates an amount of memory to `*filterPtr` to hold `max` elements.

The constructed filter, $q(t)$, is the function

$$q(t) = e^{-\pi ft/Q} \cos(2\pi ft) \quad (8.10.1)$$

where f is the central frequency and Q is the quality. The routine `qn_filter()` performs no normalization, nor does it account for different possible starting phases. The latter is not important for detection template construction. Normalization is achieved using the function `qn_normalize()`, which is described later.

Author: Jolien Creighton, jolien@tapir.caltech.edu

8.11 Function: `qn_normalize()`

```
void qn_normalize(float *u, float *q, float *r, int n, float *norm)
```

Given a filter, $\tilde{q}(f)$, and twice the inverse power spectrum, $r(f)$, this routine generates a normalized template $\tilde{u}(f)$ for which $1 = \langle N^2 \rangle \rightarrow \frac{1}{2} \text{correlate}(\dots, u, u, r, n)$. The arguments are:

`u`: Output. The array `u[0..n-1]` contains the positive frequency part of the complex template function $\tilde{u}(f)$, packed as described in the Numerical Recipes routine `realft()`.

`q`: Input. The array `q[0..n-1]` contains the positive frequency part of the complex filter function $\tilde{q}(f)$, also packed as described in the Numerical Recipes routine `realft()`.

`r`: Input. The array `r[0..n/2]` contains the values of the real function $r(f) = 2/S_h(|f|)$ used as a weight in the normalization. The array elements are arranged in order of increasing frequency from the DC component at subscript 0 to the Nyquist frequency at subscript `n/2`.

`n`: Input. The total length of the arrays `u` and `q`. Must be even.

`norm`: Output. The normalization constant, α , defined below.

Given a filter, $q(t)$, this routine computes a template, $u(t) = \alpha q(t)$, which is normalized so that $(u, u) = 2$, where (\cdot, \cdot) is the inner product defined by equation (6.14.9). Thus, the normalization constant is given by

$$\frac{1}{\alpha^2} = \frac{1}{2}(q, q). \quad (8.11.1)$$

Author: Jolien Creighton, jolien@tapir.caltech.edu

8.12 Function: `find_ring()`

```
void find_ring(float *h, float *u, float *r, float *o,  
              int n, int len, int safe, int *off,  
              float *snr, float *mean, float *var)
```

This optimally filters the strain data using an input template and then finds the time at which the SNR peaks. The arguments are:

`h`: Input. The FFT of the strain data $\tilde{h}(f)$.

`u`: Input. The normalized template $\tilde{u}(f)$.

`r`: Input. Twice the inverse power spectrum $2/S_h(|f|)$.

`o`: Output. Upon return, contains the filter output.

`n`: Input. Defines the lengths of the arrays `h[0..n-1]`, `u[0..n-1]`, `o[0..n-1]`, and `r[0..n/2]`.

`len`: Input. The number of time domain bins for which the filter $u(t)$ is non-zero. Needed in order to eliminate the wrap-around ambiguity described in subsection 6.19.

`safe`: Input. The additional number of time domain bins to use as a safety margin. This number of points are ignored at the beginning of the filter output and, along with the number of points `len`, at the ending of the filter output. Needed in order to eliminate the wrap-around ambiguity described in subsection 6.19.

`off`: Output. The offset, in the range `safe` to `n-len-safe-1`, for which the filter output is a maximum.

`snr`: Output. The maximum SNR in the domain specified above.

`mean`: Output. The mean value of the filter output over the domain specified above.

`var`: Output. The variance of the filter output over the domain specified above. Would be unity if the input to the filter were Gaussian noise with a spectrum defined by S_h .

Author: Jolien Creighton, jolien@tapir.caltech.edu

8.13 Function: `qn_inject()`

```
void qn_inject(float *strain, float *signal, float *response, float *work,  
              float invMpc, int off, int n, int len)
```

This routine injects a signal $s(t)$, normalized to a specified distance, into the strain data $h(t)$, with some specified time offset. The arguments to the routine are:

`strain`: Input/Output. The array `strain[0..n-1]` containing the strain data on input, and the strain data plus the input signal on output.

`signal`: Input. The array `signal[0..len-1]` containing the signal, in strain units at 1 Mpc distance, to be input into the strain data stream.

`response`: Input. The array `response[0..n+1]` containing the response function $R(f)$ of the IFO.

`work`: Output. A working array `work[0..n-1]`.

`invMpc`: Input. The inverse distance of the system, measured in 1/Mpc, to be used in normalizing the signal.

`off`: Input. The offset number of samples (in the time domain) at which the injected signal starts.

`n`: Input. Defines the length of the arrays `strain[0..n-1]`, `work[0..n-1]`, and `response[0..n+1]`.

`len`: Input. Defines the length of the array `signal[0..len-1]`.

Author: Jolien Creighton, jolien@tapir.caltech.edu

Comments: See the description of the routine `time_inject()`.

8.14 Vetoing techniques for ringdown waveforms

Vetoing techniques for binary inspirals have already been described in subsection 6.24; these techniques are equally applicable to searches for ringdown waveforms. However, since ringdown waveforms are short lived and have a narrow frequency band, it is much more difficult to distinguish between a ringdown waveform and a purely impulsive event. Furthermore, since the ringdown waveform will be preceded by some unknown waveform corresponding to a black hole merger, one should not be too selective as to which events should be vetoed.

Nevertheless, the Caltech 40 meter interferometer data has many spurious events that will trigger a ringdown filter, and we would expect that other instruments will have similar properties. These spurious events will (hopefully) not be too common, and most will be able to be rejected if they are not reported by other detectors. At present, however, we have only the Caltech 40 meter data to analyze, so we must consider every event that is detected by the optimal filter. The single vetoing technique that we will use at present is to look for non-Gaussian events in the detector output using the routine `is_gaussian()`. Since the expected ringdown waveforms will be only barely discernible in the raw data, such a test has no chance of accidentally vetoing an actual ringdown, but it will veto the obvious irregularities in the data.

8.15 Example: `qn_optimal` program

This program is a reworking of the program `optimal` to be run on simulated 40-meter data. Instead of searching for binary inspiral, `qn_optimal` searches for an injected quasinormal ringdown waveform. Refer to the sections on optimal filtering and the `optimal` program for a detailed discussion.

The program is setup to inject a quasinormal ringdown, produced by the routine `qn_qring()`, due to a black hole of mass $M = 50M_{\odot}$, dimensionless angular momentum parameter $\hat{a} = 0.98$, and fractional mass loss of $\epsilon = 0.03$. The injection occurs at a time of 500 s and the source distance is set to 100 kpc. The filter is constructed from the same waveform.

The following is some sample output from `qn_optimal`:

```
max snr: 3.74 (offset 30469) data start: 466.77 variance: 0.72159
max snr: 4.03 (offset 50156) data start: 479.80 variance: 0.78550
Max SNR: 9.26 (offset 70785) variance 0.796263
  If ringdown, estimated distance: 0.114364 Mpc, start time: 499.999968
  Distribution: s= 40, N>3s= 0 (expect 353), N>5s= 0 (expect 0)
  POSSIBLE RINGDOWN: Distribution does not appear to have outliers
max snr: 3.58 (offset 70974) data start: 505.86 variance: 0.77432
...
max snr: 3.62 (offset 123006) data start: 1339.81 variance: 0.70885
Max SNR: 67.01 (offset 126129) variance 4.637304
  If ringdown, estimated distance: 0.009777 Mpc, start time: 1365.618108
  Distribution: s= 40, N>3s= 320 (expect 353), N>5s= 780 (expect 0)
  Distribution has outliers! Reject
Max SNR: 93.03 (offset 1295) variance 4.444335
  If ringdown, estimated distance: 0.005934 Mpc, start time: 1365.998780
  Distribution: s= 40, N>3s= 109 (expect 353), N>5s= 280 (expect 0)
  Distribution has outliers! Reject
max snr: 2.71 (offset 127389) data start: 1378.90 variance: 0.29810
...
max snr: 4.85 (offset 118137) data start: 2152.18 variance: 0.91870
Max SNR: 12.74 (offset 69426) variance 1.332324
  If ringdown, estimated distance: 0.081144 Mpc, start time: 2172.249524
  Distribution: s= 39, N>3s= 0 (expect 353), N>5s= 0 (expect 0)
  POSSIBLE RINGDOWN: Distribution does not appear to have outliers
max snr: 3.65 (offset 35976) data start: 2178.24 variance: 0.77820
max snr: 3.76 (offset 122854) data start: 2191.28 variance: 0.67849
```

As can be seen, `qn_optimal` is able to find the ringdown and correctly estimates its distance and time of arrival.

Author: Jolien Creighton, jolien@tapir.caltech.edu

```
/* GRASP: Copyright 1997,1998 Bruce Allen */
#include "grasp.h"

#define NPOINT 131072      /* number of data points */
#define HSCALE 1.0e21     /* convenient scaling factor */
#define FLO 120.0         /* low frequency cutoff for filtering */
#define MIN_INT0_LOCK 3.0 /* time (minutes) to skip into each locked section */
#define THRESHOLD 6.0     /* detection threshold SNR */
#define ATTEN 30.0        /* attenuation cutoff for ringdown waveforms */
#define SAFETY 1000       /* padding safety to avoid wraparound errors */
#define DATA_SEGMENTS 3000 /* maximum number of data segments to filter */

double datastart;
float *response, srate=9868.4208984375;
short *datas;
int needed=NPOINT;

int main()
{
    void realft(float *, unsigned long, int);
    int fill_buffer();
    double norm;
    float *data, *htilde, *output;
    float *mean_pow_spec, *twice_inv_noise;
    float *ring, *ringtilde, *template;
    float decaytime, decay=0.0, scale, snr, mean, var, tml_norm, dist;
    float mass=50.0, spin=0.98, eps=0.03, psi0=0.0, invMpc=10.0, ringstart=500.0;
    int i, code, len, safe=SAFETY, diff, off, n=NPOINT;

    /* allocate memory for arrays */
    response=(float *)malloc(sizeof(float)*(NPOINT+2));
    datas=(short *)malloc(sizeof(short)*NPOINT);
    data=(float *)malloc(sizeof(float)*NPOINT);
    htilde=(float *)malloc(sizeof(float)*NPOINT);
    output=(float *)malloc(sizeof(float)*NPOINT);
    ringtilde=(float *)malloc(sizeof(float)*NPOINT);
    template=(float *)malloc(sizeof(float)*NPOINT);
    mean_pow_spec=(float *)malloc(sizeof(float)*(NPOINT/2+1));
    twice_inv_noise=(float *)malloc(sizeof(float)*(NPOINT/2+1));

    /* manufacture quasinormal ring data; obtain length of signal */
    ring = NULL;
    len = qn_qring(psi0, eps, mass, spin, 1.0/srate, ATTEN, n, &ring);

    /* normalize quasinormal ring to one megaparsec */
    scale = HSCALE*M_SOLAR/MPC;
    for (i=0; i<len; i++) ringtilde[i] = ring[i] *= scale;
    for (i=len; i<n; i++) ringtilde[i] = ring[i] = 0;

    /* FFT the quasinormal ring waveform */
    realft(ringtilde-1, n, 1);
    if (n<len+2*safe) abort();

    while (1) {

        /* fill buffer with number of points needed */
        code = fill_buffer();

        /* if no points left, we are done! */
    }
}
```

```
if (code==0) break;

/* if just entering a new locked stretch, reset averaging over power spectrum */
if (code==1) {
    norm = 0;
    clear(mean_pow_spec, n/2+1, 1);

    /* decay time in seconds: set to 15 x length of NPOINT sample */
    decaytime = 15.0*n/srate;
    decay = exp(-1.0*n/(srate*decaytime));
}

/* copy data into floats */
for (i=0; i<NPOINT; i++) data[i] = datas[i];

/* inject a time-domain signal before FFT (note output is used as temp storage only) */
qn_inject(data, ring, response, output, invMpc, (int)(srate*(ringstart-datastart)), n, len);

/* compute the FFT of data */
realft(data-1, n, 1);

/* normalized dL/L tilde */
product(htilde, data, response, n/2);

/* update auto-regressive mean power spectrum */
avg_inv_spec(FLO, srate, n, decay, &norm, htilde, mean_pow_spec, twice_inv_noise);

/* normalize the ring to produce a template */
qn_normalize(template, ringtilde, twice_inv_noise, n, &tmpl_norm);

/* calculate the filter output and find its maximum */
find_ring(htilde, template, twice_inv_noise, output, n, len, safe, &off, &snr, &mean, &var);

/* perform diagnostics on filter output */
if (snr<THRESHOLD) { /* threshold not exceeded: print a short message */
    printf("max snr: %.2f (offset %6d) ", snr, off);
    printf("data start: %.2f variance: %.5f\n", datastart, var);
} else { /* threshold exceeded */
    /* estimate distance to signal (template distance [Mpc] = 1 / tmpl_norm) */
    dist = 2/(tmpl_norm*snr);
    printf("\nMax SNR: %.2f (offset %d) variance %f\n", snr, off, var);
    printf("  If ringdown, estimated distance: %f Mpc, ", dist);
    printf("start time: %f\n", datastart+off/srate);
    /* See if time domain statistics are non-Gaussian */
    if (is_gaussian(datas, n, -2048, 2047, 1))
        printf("  POSSIBLE RINGDOWN: Distribution does not appear to have outliers\n\n");
    else
        printf("  Distribution has outliers!  Reject\n\n");
}

/* shift ends of buffer to the start */
diff = len + 2*safe; /* safety is applied at beginning and end of buffer */
needed = NPOINT - diff;
for (i=0; i<diff; i++) datas[i] = datas[i+needed];
}

return 0;
}
```

```
/* this routine gets the data, overlapping the data buffer as needed */
int fill_buffer()
{
    static FILE *fpifo,*fplock;
    static int first=1,remain=0,num_sent=0;
    float tstart;
    int i,temp,code=2,diff=NPOINT-needed;

    if (first) { /* on first call only */
        FILE *fpss;
        first = 0;
        diff = 0;
        /* open the IFO output file, lock file, and swept-sine file */
        fpifo = grasp_open("GRASP_DATAPATH","channel.0","r");
        fplock = grasp_open("GRASP_DATAPATH","channel.10","r");
        fpss = grasp_open("GRASP_DATAPATH","swept-sine.ascii","r");
        /* get the response function and put in scaling factor */
        normalize_gw(fpss,NPOINT,srate,response);
        for (i=0;i<NPOINT;i++) response[i] *= HSCALE/ARMLENGTH_1994;
        fclose(fpss);
    }

    if (num_sent==DATA_SEGMENTS) return 0;

    /* if new locked section, skip forward */
    while (remain<needed) {
        fprintf(stderr,"\nEntering new locked set of data\n");
        temp = get_data(fpifo,fplock,&tstart,MIN_INT0_LOCK*60*srate,datas,&remain,&srate,1);
        if (temp==0) return 0;

        /* number of points needed will be full length */
        needed = NPOINT;
        diff = 0;
        code = 1;
    }

    /* get the needed data and compute the start time of the buffer */
    temp = get_data(fpifo,fplock,&tstart,needed,datas+diff,&remain,&srate,0);
    if (temp==0) return 0;
    datastart = tstart - diff/srate;

    num_sent++;
    return code;
}
```

8.16 Structure: `struct qnTemplate`

The structure that will hold the filters for quasinormal ringdown waveforms is: `struct qnTemplate{`

`int num`: The number of the particular filter.

`float freq`: The central frequency of the filter template.

`float qual`: The quality of the filter template.

`};`

The actual filter data that corresponds to the parameters set by the fields `freq` and `qual` is generated by the routine `qn_filter()` above.

8.17 Structure: `struct qnScope`

The structure `struct qnScope` specifies a domain of parameter space and contains a set of templates that cover this domain. The fields of this structure are: `struct qnScope`{

`int n_tmplt`: The total number of templates required to cover the region in parameter space. This is typically set by `qn_template_grid()`.

`float freq_min`: The minimum frequency of the region of parameter space.

`float freq_max`: The maximum frequency of the region of parameter space.

`float qual_min`: The minimum quality of the region of parameter space.

`float qual_max`: The maximum quality of the region of parameter space.

`struct qnTemplate *templates`: Pointer to the array of templates. This pointer is usually set by `qn_template_grid()` when it allocates the memory necessary to store the templates and creates the necessary templates.

};

Although we are interested in the physical parameters, such as the mass and angular momentum, of the black hole sources of gravitational radiation, it will be more convenient to work with the frequency and quality parameters of damped sinusoids when creating detection templates. For the fundamental quadrupole quasinormal mode, there is a one-to-one correspondence between the mass and angular momentum parameters and the frequency and quality parameters which is approximately given by Echeverria [25].

8.18 Function: `qn_template_grid()`

```
void qn_template_grid(float dl, struct qnScope *grid)
```

This function is responsible for allocating the memory for a grid of templates on the parameter space and for choosing the location of the templates. The arguments are:

`dl`: Input. The length of the ‘sides’ of the square templates. This quantity should be set to $dl = \sqrt{2ds_{\text{threshold}}^2}$ (see the discussion below).

`grid`: Input/Output. The grid of templates of type `struct qnScope`. On input, the fields that relate to parameter ranges should be set. On output, the field `n_tmplt` is set to the number of templates generated, and these templates are put into the array field `templates[0..n_tmplt-1]` (which is allocated by the function).

The function `qn_template_grid()` attempts to create a set of templates, $\{u_i(t)\}$, which “cover” parameter space finely enough that the distance between an arbitrary point on the parameter space and one of the templates is small. A precise statement of this goal, and how it is achieved, can be found in the paper by Owen [5]. We highlight the relevant parts of reference [5] here.

The templates $\{u_i(t)\}$ are damped sinusoids with a set of frequency and quality parameters $\{(f, Q)_i\}$. They are normalized so that $(u_i|u_i) = 1$ where $(\cdot|\cdot)$ is the inner product defined by Cutler and Flanagan [21]. Since we are most interested in the high quality region of parameter space, it is a good approximation that the value of the one-sided noise power spectrum is approximately constant, $S_h(f) \approx S_h(f_i)$, over the frequency band of the template. This approximation simplifies the form of the inner product as the noise power spectrum appears in the inner product as a weighting function.

In order to estimate how close together the templates must be, we define the distance function $ds_{ij}^2 = 1 - (u_i|u_j)$ corresponding to the mismatch between the two templates u_i and u_j . This interval can be expressed in terms of a metric as $ds^2 = g_{\alpha\beta}dx^\alpha dx^\beta$ where $x^\alpha = (f, Q)^\alpha$ are coordinates on the two dimensional parameter space. Such an expression is only valid for sufficiently close points on parameter space. In the limit of a continuum of templates over parameter space, the metric can be evaluated by $g_{\alpha\beta} = -\frac{1}{2}(u|\partial_\alpha\partial_\beta u)$ where ∂_α is a partial derivative with respect to the coordinate x^α . We find that the mismatch between templates that differ in frequency by df and in quality by dQ is given by

$$ds^2 = \frac{1}{8} \left\{ \frac{3 + 16Q^4}{Q^2(1 + 4Q^2)^2} dQ^2 - 2 \frac{3 + 4Q^2}{fQ(1 + 4Q^2)} dQ df + \frac{3 + 8Q^2}{f^2} df^2 \right\} \quad (8.18.1)$$

$$\approx \frac{1}{8} \frac{dQ^2}{Q^2} - \frac{1}{4} \frac{dQ}{Q} \frac{df}{f} + Q^2 \frac{df^2}{f^2}. \quad (8.18.2)$$

In the approximate metric of equation (8.18.2), we have kept only the dominant term in the limit of high quality. The minimum number of templates, \mathcal{N} , required to span the parameter space such that there is no point on parameter space that is a distance larger than $ds_{\text{threshold}}^2$ from the nearest template can be found by integrating the volume element $\sqrt{\det g_{\alpha\beta}}$ over the parameter space. Using the approximate metric and the parameter ranges $Q \leq Q_{\text{max}}$ and $f \in [f_{\text{min}}, f_{\text{max}}]$, we find that the number of templates required is

$$\begin{aligned} \mathcal{N} &\approx \frac{1}{4\sqrt{2}} (ds_{\text{threshold}}^2)^{-1} Q_{\text{max}} \log(f_{\text{max}}/f_{\text{min}}) \\ &\simeq 2700 \left(\frac{ds_{\text{threshold}}^2}{0.03} \right)^{-1} \left(\frac{Q_{\text{max}}}{100} \right) \left\{ 1 + \frac{1}{\log 100} \left[\log\left(\frac{f_{\text{max}}}{10 \text{ kHz}}\right) - \log\left(\frac{f_{\text{min}}}{100 \text{ Hz}}\right) \right] \right\}. \end{aligned} \quad (8.18.3)$$

The issue of template placement is more difficult than computing the number of templates required. Fortunately, for the problem of quasinormal ringdown template placement, the metric is reasonably simple.

By using the coordinate $\phi = \log f$ rather than f , we see that the metric components depend on Q alone. We can exploit this property for the task of template placement as follows: First, choose a “surface” of constant $Q = Q_{\min}$, and on this surface place templates at intervals in ϕ of $d\phi = d\ell/g_{\phi\phi}$ for the entire range of ϕ . Here, $d\ell = \sqrt{(2ds_{\text{threshold}}^2)}$. Then choose the next surface of constant Q with $dQ = d\ell/g_{QQ}$ 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.

Author: Jolien Creighton, jolien@tapir.caltech.edu

8.19 The close-limit approximation and numerical simulations

For a subset of black hole collisions, where the black holes collide head-on, there exist as of today (Feb 1999) reasonably reliable full numerical simulations of the collision. Because of the lack of an inspiral phase, the waveform profiles of these kind of collisions are completely dominated by the ringdown of the final black hole [for a recent reference see Anninos, Brandt, and Walker [49] (ABW)]. Even for this simple case, there are some discrepancies between various numerical codes.

A separate approach to black hole collisions has been the close-limit approximation (see Khanna et al. gr-qc/9905081 for a description of the close-limit approximation applied to inspiralling black holes), which describes the merger of two black holes as a perturbation of a single black hole; the perturbation is based on a small parameter measuring the separation of the two black holes. This approximation has had an uncanny degree of success in replicating (at least for the head-on case) numerical estimates of the merger waveform, and it provides “a little-bit more” of the merger waveform than the quasinormal ringdown. It is useful to examine how good ringdown filters will work in detecting the more realistic waveforms produced by the close-limit approximation. In this addendum we will describe the use of both close limit and full numerical simulations in the GRASP package.

There are two data sets containing full numerical waveforms for head on collisions of two black holes released initially from rest. These correspond to the two codes described in ABW [49] for a moderate separation of the holes ($\mu_0 = 1.9$ in the ABW [49] notation, about 10 in terms of single black hole radii). These waveforms are shown in figure 54.

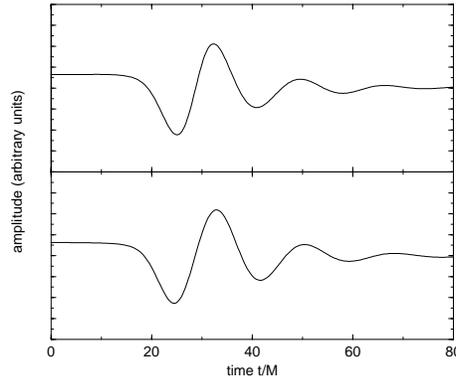


Figure 54: ABW waveforms with $\mu_0 = 1.9$.

These two waveforms correspond to the two codes described in ABW [49], based on two different set of coordinate systems. To a certain extent, they exhibit the limitations of the state of the art in numerical relativity: both waveforms represent the best effort by correct, “convergent” codes, and yet there is some disagreement among them. This disagreement can be settled by comparing with the close approximation (see ABW [49]). But it is also instructive to compare how bad the disagreement is from the point of view of a data analyst.

The fitting factor is defined by

$$\eta = \max_t r(t) = \max_t \alpha^{-1} \Re \int_0^\infty df e^{-2\pi i f t} \frac{\tilde{a}(f) \tilde{b}^*(f)}{S(f)} \quad (8.19.1)$$

where

$$\alpha^2 = \int_0^\infty df \frac{\|\tilde{a}(f)\|^2}{S(f)} \times \int_0^\infty df \frac{\|\tilde{b}(f)\|^2}{S(f)} \quad (8.19.2)$$

with $a(t)$ being the close-limit approximation waveform, $b(t)$ being the ringdown waveform, and $S(f)$ being the detector noise power spectrum. This is computed by the program `corr` (below). For the LIGO-I interferometer noise curve and a $200 M_{\odot}$ black hole, the fitting factors are 91% and 85% for the full numerical waveforms. These factors represent the fraction of the signal-to-noise ratio that the ringdown filter will obtain relative to the optimal filter. As can be seen in the following figure, the moment at which the ABW [49] waveform looks most “ringdown-like” is *not* the moment at which the peak signal-to-noise ratio is obtained.

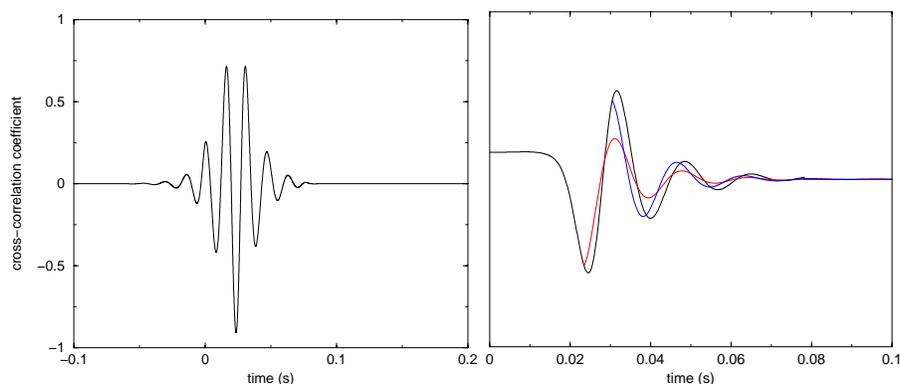


Figure 55: Pure ringdown fits to the ABW waveform. First: the correlation between the ABW waveform and a pure ringdown as a function of time. Second: pure ringdowns superimposed on the ABW waveform for the times of greatest correlation.

From the point of view of source modelling, the difference in head-on collisions between the full numerical waveforms and the close limit ones is not substantial.

Authors: Jolien Creighton (jolien@tapir.caltech.edu) and Jorge Pullin (pullin@phys.psu.edu)

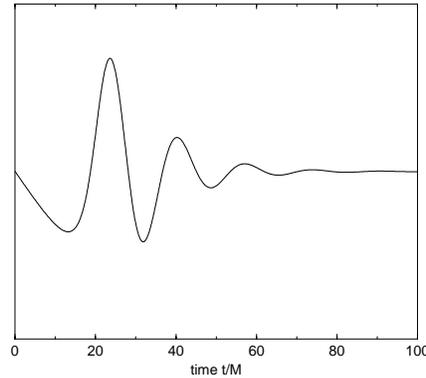


Figure 56: The waveform for the ringdown of an inspiralling collision with Kerr parameter $a = 0.35 M$ and initial separation $L = 0.9 M$. Extrapolations beyond the realm of validity of the close limit suggest that for a “realistic” collision with a/M close to unity about 1% of the mass is radiated.

8.20 Inspiralling collisions

For the collision of inspiralling black holes there are no currently available full numerical simulations. Here the only information available is from the close limit approximation. In this case, one only expects to get correctly one portion of the waveform, since in addition to the final ringdown captured by the close limit approximation, one also will have the “chirp” and “merger” phases of the collision. Nevertheless, it is instructive to see what the use of the “realistic” ringdown part of the waveforms yields. The close-limit approximation is quite limited in validity for inspiralling black hole. The problem is that for large values of the angular momentum, as are expected in realistic collisions, the spacetime departs quite radically from that of a single spinning black holes. Best educated guesses suggest that trustworthy results from the close limit approximation can only be obtained up to values of the Kerr parameter of $a = 0.5 M$. At such values, for separations of a few M , the radiated energy is of the order of 0.3% of the total mass of the system. Of the angular momentum, a similar fraction gets radiated away. By eyeballing the curve showing the energy dependence as a function of angular momentum, one could expect that a “realistic” collision with values of a/M close to unity would probably radiate of the order of 1% of the total mass. This is a significant extrapolation from the “reliable” results, but barring unexpected physics at the last moments of the collision, it is probably right. We present here the analysis of a “close-limit” type waveform for the ringdown of the final moments of an inspiralling collision. The waveform was calculated for $a = 0.35 M$ and separation $L = 0.9 M$ ($\mu_0 = 1.5$). The waveform is shown in figure 56.

For the LIGO-I noise curve and a total black hole mass of $200 M_\odot$, the fitting factor achieved by a quasinormal ringdown template with the “correct” $a = 0.35 M$ is 69% (see figure 57). However, this is not a true fitting factor in the sense that the maximization has been over the time of arrival only—for different values of the ringdown template mass and spin, the fit will improve. In particular, only the $\ell = m = 2$ quasinormal mode is considered in constructing the ringdown template while the close-limit waveform contains excitations from the $\ell = 2, m = 0$ as well. For a rapidly spinning black hole, the $\ell = m = 2$ mode will (eventually) dominate the waveform because it is much longer lived. However, the spin of the black hole in the present case is not particularly large, and the $m = 0$ mode tends to dominate. In fact, the fitting factor obtained by using a ringdown template with $a = 0$ is 84%—much better than the fit obtained using the $\ell = m = 2$ ringdown with $a = 0.35 M$. It would be useful to examine the entire parameter space of quasinormal ringdowns in mass and spin as well as arrival time in order to obtain the “true” fitting factor.

Authors: Jolien Creighton (jolien@tapir.caltech.edu) and Jorge Pullin (pullin@phys.psu.edu)

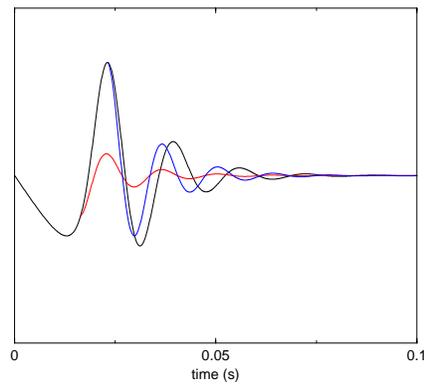


Figure 57: Pure ringdowns superimposed on the waveform for the ringdown of an inspiralling collision with Kerr parameter $a = 0.35 M$ and initial separation $L = 0.9 M$ for the times of greatest correlation.

8.21 Example: ring-corr program

This program computes the maximum weighted cross-correlation of a black hole merger waveforms (contained in data files) with ringdown waveforms. The input data file(s) contain close-limit approximations or numerical relativity results for the merger of two black holes.

To execute the program:

```
corr [options] [file1 [file2]]
```

options:

```
-h          prints a help message
-m mass     specifies the black hole mass (in solar masses)
-s spin     specifies the dimensionless black hole spin [0,1)
-p powfile  file containing the noise power spectrum
```

Here, file1 and file2 are optional filenames containing the waveform data. If two arguments are present, the cross-correlations of the data in the two files is computed; otherwise, the cross-correlation of the data in the single file (or in file close-limit.dat if there are no arguments) with a Schwarzschild ringdown is computed. If a power spectrum filename is not specified, the program looks for ligo-0.dat.

```
#include <assert.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <math.h>

#include "grasp.h"

extern char *optarg;
extern int  optind;
int getopt(int, char * const [], const char *);

int main(int argc, char *argv[])
{
    void usage(char *);
    void realft(float [], unsigned long, int);
    void readNoisePower(FILE *, double [], int, float);
    void readCloseLimitData(FILE *, float [], int, float, float);
    void makeQuasiNormalRing(float [], int, float, float, float);

    const double hscale = 1e21;      /* a convenient scale factor */
    const float  Msun   = 4.89e-6;   /* solar mass (s) */
    const float  srate  = 16384;    /* sample rate (Hz) */
    const int    npoint = 65536;    /* segment length (samples) */

    double scale;
    double *power;                  /* power spectrum S(f) */
    float  *weight;                 /* weighting factor 4/S(f) */
    float  *arrayA;                 /* waveform a(t) */
    float  *arrayB;                 /* waveform b(t) */
    float  *crossCorr;              /*  $\int df e^{2\pi ift} \tilde{a}(f) \tilde{b}^*(f) / S(f)$  */
    float  *autoCorrA;              /*  $\int df e^{2\pi ift} |\tilde{a}(f)|^2 / S(f)$  */
    float  *autoCorrB;              /*  $\int df e^{2\pi ift} |\tilde{b}(f)|^2 / S(f)$  */
    float  spin = 0.35;             /* dimensionless spin of black hole */
    float  mass  = 200;             /* mass of black hole (solar masses) */
    float  norm;
```

```
float    max;
char     powfile[256] = "ligo-0.dat"; /* noise power filename */
FILE     *fp;
int      i;

/* allocate memory to arrays */
assert(power      = (double *)malloc((npoint/2+1)*sizeof(double)));
assert(weight     = (float  *)malloc((npoint/2+1)*sizeof(float)));
assert(arrayA     = (float  *)malloc(npoint*sizeof(float)));
assert(arrayB     = (float  *)malloc(npoint*sizeof(float)));
assert(crossCorr  = (float  *)malloc(npoint*sizeof(float)));
assert(autoCorrA  = (float  *)malloc(npoint*sizeof(float)));
assert(autoCorrB  = (float  *)malloc(npoint*sizeof(float)));

while (1) { /* parse command line options */

    int c;

    /* call the standard C library option parser */
    c = getopt(argc,argv,"hs:m:p:");
    if (c == -1) break;

    switch (c) {

    case 'h': /* print a simple message and exit */
        usage(argv[0]);
        exit(0);

    case 's':
        spin = atof(optarg);
        assert(spin >= 0 && spin < 1);
        break;

    case 'm':
        mass = atof(optarg);
        assert(mass > 0);
        break;

    case 'p':
        strncpy(powfile,optarg,sizeof(powfile));
        break;

    default: /* something went wrong */
        fprintf(stderr,"warning: getopt returned character code 0%o\n",c);
    }

}

switch (argc - optind) { /* process remaining command line arguments */

case 0: /* compare a default data file and a computed ringdown */
    fprintf(stderr,"compare waveform data from file close-limit.dat\n");
    assert(fp = fopen("close-limit-insp.dat","r"));
    readCloseLimitData(fp,arrayA,npoint,srate,mass*Msun);
    fclose(fp);
    fprintf(stderr,"with a computed ringdown waveform\n");
    makeQuasiNormalRing(arrayB,npoint,srate,mass*Msun,spin);
    fprintf(stderr,"  black hole mass: %.2f solar masses\n",mass);
```

```
    fprintf(stderr, "  black hole spin: %05.2f%% of extreme\n", spin*100);
    break;

case 1: /* compare a specified data file and a computed ringdown */
    fprintf(stderr, "compare waveform data from file %s\n", argv[optind]);
    assert(fp = fopen(argv[optind++], "r"));
    readCloseLimitData(fp, arrayA, npoint, srate, mass*Msun);
    fclose(fp);
    fprintf(stderr, "with a computed ringdown waveform\n");
    makeQuasiNormalRing(arrayB, npoint, srate, mass*Msun, spin);
    fprintf(stderr, "  black hole mass: %.2f solar masses\n", mass);
    fprintf(stderr, "  black hole spin: %05.2f%% of extreme\n", spin*100);
    break;

case 2: /* compare two specified data files */
    fprintf(stderr, "compare waveform data from file %s\n", argv[optind]);
    assert(fp = fopen(argv[optind++], "r"));
    readCloseLimitData(fp, arrayA, npoint, srate, mass*Msun);
    fclose(fp);
    fprintf(stderr, "with waveform data from file %s\n", argv[optind]);
    assert(fp = fopen(argv[optind++], "r"));
    readCloseLimitData(fp, arrayB, npoint, srate, mass*Msun);
    fprintf(stderr, "  black hole mass: %.2f solar masses\n", mass);
    fclose(fp);
    break;

default: /* too many arguments */
    usage(argv[0]);
    exit(1);
}

/* get correlation weighting factor */
fprintf(stderr, "using power spectrum from file %s\n", powfile);
assert(fp = grasp_open("GRASP_PARAMETERS", powfile, "r"));
readNoisePower(fp, power, npoint/2, srate);
fclose(fp);
scale = 4/(npoint*srate*hscale*hscale);
weight[0] = weight[npoint/2] = 0;
for (i = 1; i < npoint/2; ++i)
    weight[i] = scale/power[i];

/* FFT waveform arrays */
realft(arrayA-1, npoint, 1);
realft(arrayB-1, npoint, 1);

/* compute cross- and auto-correlations */
autoCorrA[0] = autoCorrA[1] = 0;
autoCorrB[0] = autoCorrB[1] = 0;
crossCorr[0] = crossCorr[1] = 0;
for (i = 1; i < npoint/2; ++i) {
    int ir = i + i;
    int ii = ir + 1;
    float ar = arrayA[ir];
    float ai = arrayA[ii];
    float br = arrayB[ir];
    float bi = arrayB[ii];
    float fac = weight[i];
    autoCorrA[ir] = fac*(ar*ar + ai*ai);
```

```
    autoCorrA[ii] = 0;
    autoCorrB[ir] = fac*(br*br + bi*bi);
    autoCorrB[ii] = 0;
    crossCorr[ir] = fac*(ar*br + ai*bi);
    crossCorr[ii] = fac*(ai*br - ar*bi);
}
realft(autoCorrA-1,npoint,-1);
realft(autoCorrB-1,npoint,-1);
realft(crossCorr-1,npoint,-1);

/* compute fitting factor normalization */
assert(autoCorrA[0] > 0);
assert(autoCorrB[0] > 0);
norm = sqrt(autoCorrA[0]*autoCorrB[0]);
assert(norm > 0);

/* find maximum of cross-correlation and print fitting factor */
max = 0;
for (i = 0; i < npoint; ++i)
    if (fabs(crossCorr[i]) > fabs(max))
        max = crossCorr[i];
printf("fitting factor: %.2f%%\n",100*max/norm);

return 0;
}

/* Print a message describing the usage of this program.
 * Arguments:
 * *program the program name
 */
void usage(char *program)
{
    fprintf(stderr,"usage: %s [options] [file1 [file2]]\n",program);
    fprintf(stderr,"options:\n");
    fprintf(stderr," -h prints this message\n");
    fprintf(stderr," -s spin dimensionless spin of the black hole [0,1]\n");
    fprintf(stderr," -m mass mass of the black hole (solar masses)\n");
    fprintf(stderr," -p file file containing the noise power spectrum\n");
    return;
}

/* Read a data file containing a close-limit waveform.
 * Arguments:
 * *fp the data file
 * *arr array to store the data
 * npoint size of array *arr
 * srate sample rate for data in *arr (in Hz)
 * mass mass of black hole (in seconds)
 */
void readCloseLimitData(FILE *fp, float *arr, int npoint, float srate,
                        float mass)
{
    void spline(float [], float [], int, float, float, float []);
    void splint(float [], float [], float [], int, float, float *);

    const int ninc = 1024;
    const float natural = 1e30;

```

```
float *time, *data, *datapp;
int i, imax, nmax = ninc, n = 0;

/* allocate memory */
assert(data = (float *)malloc(nmax*sizeof(float)));
assert(time = (float *)malloc(nmax*sizeof(float)));

/* read waveform data file */
while (EOF != fscanf(fp, "%e\t%e\n", time+n, data+n))
    if (++n >= nmax) {
        nmax += ninc;
        assert(data = (float *)realloc(data, nmax*sizeof(float)));
        assert(time = (float *)realloc(time, nmax*sizeof(float)));
    }

/* use cubic spline interpolation to generate waveform at
 * the required sample times
 */
assert(datapp = (float *)malloc(n*sizeof(float)));
spline(time-1, data-1, n, natural, natural, datapp-1);
imax = (int)floor((time[n-1] - time[0])*srate*mass);
for (i = 0; i < npoint; ++i)
    if (i > imax)
        arr[i] = 0;
    else
        splint(time-1, data-1, datapp-1, n, (float)i/(srate*mass)+time[0], arr+i);

/* free memory and return */
free(time);
free(data);
free(datapp);
return;
}

/* Generate a quasinormal ringdown waveform.
 * Arguments:
 * *arr    array to store the data
 * *npoint size of array *arr
 * *srate  sample rate for data in *arr (in Hz)
 * *mass   mass of black hole (in seconds)
 * *spin   dimensionless spin of black hole
 */
void makeQuasiNormalRing(float *arr, int npoint, float srate,
                        float mass, float spin)
{
    const float pi = 3.14159265358979;
    const float freq = (1 - 0.63*pow(1-spin, 0.3))/(2*pi*mass);
    const float qual = 2*pow(1-spin, -0.45);
    /* freq and qual are computed using the fits by Echeverria */
    int i;

    for (i = 0; i < npoint; ++i) {
        float time = (float)i/srate;
        arr[i] = exp(-pi*time*freq/qual)*cos(2*pi*time*freq);
    }

    return;
}
}
```

```
/* Read the noise power spectrum.
 * Arguments:
 * *fp      the noise power file
 * *power   array to store the data
 * n       size of array *power
 * srate   sample rate for data (in Hz)
 */
void readNoisePower(FILE *fp, double *power, int n, float srate)
{
    void spline(float [], float [], int, float, float, float []);
    void splint(float [], float [], float [], int, float, float *);
    const int nmax = 65536; /* assumed maximum size of data file */
    const float natural = 1e30;
    float *freq, *ampl, *amplpp;
    char line[100];
    int i, length;

    /* allocate memory */
    assert(freq = (float *)malloc(nmax*sizeof(float)));
    assert(ampl = (float *)malloc(nmax*sizeof(float)));
    assert(amplpp = (float *)malloc(nmax*sizeof(float)));

    /* read power spectrum data file */
    length = 0;
    while (1) {
        if (fgets(line, sizeof(line), fp) == NULL) /* end of file */
            break;
        if (line[0] != '#') {
            assert(length < nmax);
            sscanf(line, "%e\t%e\n", freq+length, ampl+length);
            ++length;
        }
    }

    /* use cubic spline interpolation to get the spectrum
     * at the required frequencies
     */
    spline(freq-1, ampl-1, length, natural, natural, amplpp-1);
    for (i = 0; i < n; ++i) {
        float f = i*srate/(float)n;
        float value;
        double dvalue;
        splint(freq-1, ampl-1, amplpp-1, length, f, &value);
        dvalue = (double)value;
        power[i] = dvalue*dvalue;
    }

    /* free memory and return */
    free(freq);
    free(ampl);
    free(amplpp);
    return;
}

/* The following routines is to be run to display data during
   debugging it's based on the GRASP graph() routine */

void graphSpec(float arr[], int n)
{
```

```
FILE *fp;
int i;
fp = fopen("temp.graph","w");
for (i = 0; i < n/2; ++i) {
    int ir = i + i;
    int ii = ir + 1;
    float re = arr[ir];
    float im = arr[ii];
    float pow = re*re + im*im;
    float arg = atan2(im,re);
    fprintf(fp,"%d\t%e\t%e\n",i,pow,arg);
}
fclose(fp);
system("xmgr -nxy temp.graph 1>/dev/null 2>&1 &");
return;
}
```

Authors: Jolien Creighton (jolien@tapir.caltech.edu) and Jorge Pullin (pullin@phys.psu.edu)

9 GRASP Routines: Template Bank Generation & Searching

For the most part, one of our main interests is the search for signals whose wave-forms are characterized by unknown values of a set of parameters (for binary inspiral, these would be m_1 and m_2). In order to use the matched filtering technique described in the previous section, it is necessary to set up a “bank” of templates, designed so that any expected signal is “close” (in parameter space) to one of the elements of the bank. This section contains a set of routines for setting up such a template bank, in the case where the signals are parameterized by two parameters.

It also contains a (parallel) routine to search for binary inspiral in a bank of templates.

9.1 Structure: struct Template

The structure used to describe the “chirp” signals from coalescing binary systems is: struct Template {

int num: In order to deal with templates “wholesale” it is useful to number them. The numbering system is up to you; we typically give each template a number, starting from 0 and going up to the number of templates minus one!

float f_lo: This is the starting (low) frequency f_0 of template, in units of sec^{-1} .

float f_hi: This is the ending (high) frequency of the template, in units of sec^{-1}

float tau0: The Newtonian time τ_0 to coalescence, in seconds, starting from the moment when the frequency of the waveform is f_lo.

float tau1: First post-Newtonian correction τ_1 to τ_0 .

float tau15: 3/2 PN correction

float tau20: second order PN correction

float pha0: Newtonian phase to coalescence, radians

float pha1: First post-Newtonian correction to pha0

float pha15: 3/2 PN correction

float pha20: second order PN correction

float mtotal: total mass $m_1 + m_2$, in solar masses

float mchirp: chirp mass $\mu\eta^{-2/5}$, in solar masses

float mred: the reduced mass $\mu = m_1 m_2 / (m_1 + m_2)$, in solar masses

float eta: reduced mass/total mass $\eta = m_1 m_2 / (m_1 + m_2)^2$, dimensionless

float m1: the smaller of the two masses, in solar masses.

float m2: the larger of the two masses, in solar masses.

};

One may use the technique of *matched filtering* to search for chirps. The (noisy) signal is compared with templates, each formed from a chirp with a particular values of m_1, m_2 , and a “start frequency” f_0 of the waveform at the time that it enters the bandpass of the gravitational wave detector. Several theoretical studies [4, 5] have shown how the template filtering technique performs when the detector is not ideal, but is contaminated by instrument noise.

In the presence of detector noise, one can never be entirely certain that a given chirp (determined by m_1, m_2) will be detected by a particular template, even one with the exact same mass parameters. However one can make statistical statements about a template, such as “if the masses m_1 and m_2 of the chirp lie in region R of parameter space, then with 97% probability, they will be detected if their amplitude exceeds value h ”. Thus, associated with each chirp, and a specified level of uncertainty, is a region of parameter space.

It turns out that if we use the correct choice of coordinates on the parameter space (m_1, m_2) then these regions R are quite simple. If we demand that the uncertainty associated with each template be fairly small, then these regions are ellipses. Moreover, to a good approximation, the shape of the ellipses is determined only by the noise power spectrum of the detector, and does not change significantly as we move about in the parameter space. These “nice” coordinates (τ_0, τ_1) have units of time, and are defined by

$$\begin{aligned}\tau_0 &= \frac{5}{256} \left(\frac{GM}{c^3} \right)^{-5/3} \eta^{-1} (\pi f_0)^{-8/3} \\ &= \frac{5}{256} \left(\frac{M}{M_\odot} \right)^{-5/3} \eta^{-1} (\pi f_0)^{-8/3} T_\odot^{-5/3}\end{aligned}\tag{9.1.1}$$

and

$$\begin{aligned}\tau_1 &= \frac{5}{192} \left(\frac{c^3}{G\eta M} \right) \left(\frac{743}{336} + \frac{11}{4}\eta \right) (\pi f_0)^{-2} \\ &= \frac{5}{192} \left(\frac{M_\odot}{M} \right) \left(\frac{743}{336}\eta^{-1} + \frac{11}{4} \right) (\pi f_0)^{-2} T_\odot^{-1}.\end{aligned}\tag{9.1.2}$$

The symbol

$$M \equiv m_1 + m_2\tag{9.1.3}$$

denotes the total mass of the binary system, and

$$\eta \equiv \frac{m_1 m_2}{(m_1 + m_2)^2}\tag{9.1.4}$$

is the ratio of the reduced mass to M . Notice that η is always (by definition) less than or equal to $1/4$.

We are generally interested in a region of parameter space corresponding to binary systems, each of whose masses lie in some given range, say from $1/2$ to 3 solar masses. The region of parameter space is determined by a minimum and maximum mass; we show an example of this in Figure 58. Since we may take $m_2 \leq m_1$ without loss of generality, the region of interest is triangular rather than rectangular. The three lines on this diagram are:

- (1) The equal mass line. Along this line $\eta = 1/4$.
- (2) The minimum mass line. Along this line, one of the masses has its smallest value.
- (3) The maximum mass line. Along this line, one of the masses has its largest value.

This triangular region is mapped into the (τ_0, τ_1) plane as shown in Figure 59 In this diagram, the lower curve $\tau_1 \propto \tau_0^{3/5}$ is the equal mass line (1). The upper curve, to the right of the “kink” is the minimum mass line (2). The upper curve, to the left of the “kink” is the maximum mass line (3).

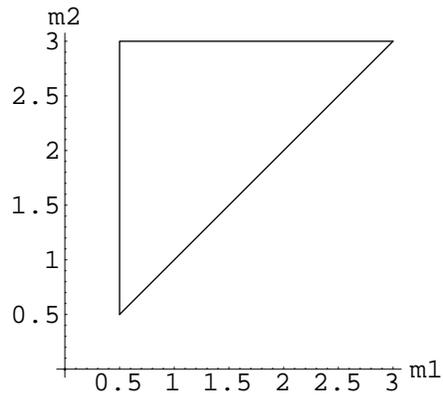


Figure 58: The set of binary stars with masses lying between set minimum and maximum values defines the interior of a triangle in parameter space

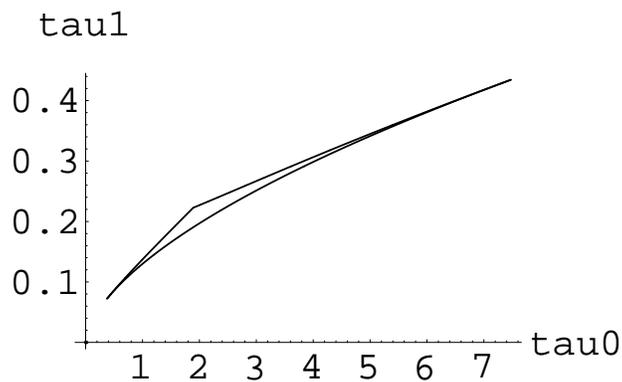


Figure 59: The triangular region of the previous figure is mapped into a distorted triangle in the (τ_0, τ_1) plane. Here f_0 is 120 Hz.

9.2 Structure: struct Scope

The set of templates is described by a structure `struct Scope`. This structure specifies a set of templates covering the mass range in parameter space described above and shown in Figure 59. The fields of this structure are:

```
struct Scope {
```

`int n_tmplt`: This integer is the total number of templates needed to cover the region in parameter space. This is typically computed or set by `template_grid()`.

`float m_mn`: The minimum mass of an object in the binary system, as described above, in solar masses.

`float m_mx`: The maximum mass of an object in the binary system, as described above, in solar masses. Together with the `m_mn`, this describes the region in parameter space covered by the set of templates.

`float theta`: The angle to the axis of the constant ambiguity ellipse whose axis has diameter `dp`. The angle is measured in radians counterclockwise from the τ_0 axis. The range is $\theta \in (-\pi/2, \pi/2)$.

`float dp`: The diameter along the ellipse (in sec). This is twice the radius r_1 given in Table 8. The angle θ is measured to this axis.

`float dq`: The diameter along the ellipse (in sec). This is twice the radius r_2 given in Table 8.

`float f_start`: The frequency f_0 used in the definitions of τ_0 and τ_1 (9.1.1,9.1.2); this is typically the frequency at which a binary chirp first enters the usable bandpass of the detector.

`struct Template* templates`: Pointer to the array of templates. This pointer is typically set by `template_grid()`, when it allocates the memory necessary to store the templates, and creates the necessary templates.

```
};
```

Note that a given constant ambiguity ellipse can be specified in either of two equivalent ways. For example the ellipse defined by

$$\theta = \pi/4, dp = 1 \text{ msec}, dq = 5 \text{ msec} \quad (9.2.1)$$

is completely equivalent to the ellipse

$$\theta = 5\pi/4, dp = 5 \text{ msec}, dq = 1 \text{ msec}. \quad (9.2.2)$$

Either of these is acceptable. The literature frequently uses the second convention (angle measured to the major axis).

9.3 Function: tau_of_mass()

```
void tau_of_mass(double m1, double m2, double pf, double *tau0, double *tau1)
```

This function calculates the coordinates (τ_0, τ_1) associated with particular values of the masses of the objects in the binary system, and a particular value of frequency f_0 .

The arguments are:

m1: Input. The first mass (in solar masses).

m2: Input. The second mass (in solar masses).

pf: Input. The value πf_0 . Here f_0 is the frequency used in defining the τ coordinates (see below). It is often chosen to be at (or below) the frequency at which the chirp first enters the bandpass of the gravitational wave detector.

tau0: Output. Pointer to τ_0 (in seconds).

tau1: Output. Pointer to τ_1 (in seconds).

Although one can think of τ_0 and τ_1 as coordinates in the parameter space defined by (9.1.1) and (9.1.2) they have simple physical meanings. τ_0 is the time to coalescence of the binary system, measured from the time that the waveform passes through frequency f_0 , in the zeroth post-Newtonian approximation. τ_1 is the first-order post-Newtonian correction to this quantity, so that to this order the time to coalescence is $\tau_0 + \tau_1$.

Author: Bruce Allen, ballen@dirac.phys.uwm.edu

Comments: None.

9.4 Function: `m_and_eta()`

`int m_and_eta(double tau0, double tau1, double *M, double *eta, double Mmin, double Mmax, double pf)`

This function takes as inputs the coordinates (τ_0, τ_1) . If these correspond to individual masses m_1 and m_2 each lying in the range from M_{\min} to M_{\max} then the function sets the total mass $M = m_1 + m_2$ and sets $\eta = m_1 m_2 / (m_1 + m_2)^2$ and returns the value 1. Otherwise, the function returns 0 and does not change the values of mass M or η .

The arguments are:

`tau0` Input. The value of τ_0 (positive, sec).

`tau1` Input. The value of τ_1 (positive, sec).

`M` Output. The total mass M (solar masses). Unaltered if no physical mass values are found in the desired range.

`eta` Output. The value of η (dimensionless). Unaltered if no physical mass values are found in the desired range.

`Mmin` Input. Minimum mass of one object in the binary pair, in solar masses (positive).

`Mmax` Input. Maximum mass of one object in the binary pair, in solar masses (positive).

`pf`: Input. The value πf_0 . Here f_0 is the frequency at which the chirp first enters the bandpass of the gravitational wave detector.

The algorithm followed by `m_and_eta()` is as follows. Eliminate η from the equations defining τ_0 (9.1.1) and τ_1 (9.1.2) to obtain the following relation:

$$c_1 + c_2 \left(\frac{M}{M_\odot} \right)^{5/3} - c_3 \left(\frac{M}{M_\odot} \right) = 0, \quad (9.4.1)$$

with the constants given by:

$$\begin{aligned} c_1 &= 1155 T_\odot \\ c_2 &= 47552 (\pi f_0 T_\odot)^{8/3} \tau_0 \\ c_3 &= 16128 (\pi f_0 T_\odot)^2 \tau_1. \end{aligned} \quad (9.4.2)$$

Given (τ_0, τ_1) our goal is to find the roots of equation (9.4.1). It is easy to see that the function on the lhs of (9.4.1) has at most two roots. The function is positive at $M = 0$ but decreasing for small positive M . However it is positive and increasing again as $M \rightarrow \infty$. Hence the function on the lhs of (9.4.1) has at most a single minimum for $M > 0$. Setting the derivative equal to zero and solving, this minimum lies at a value of the total mass M_{crit} which satisfies

$$\frac{M_{\text{crit}}}{M_\odot} = \left(\frac{3}{5} \frac{c_3}{c_2} \right)^{3/2} \quad (9.4.3)$$

Hence the lhs of (9.4.1) has no roots if its value is positive at $M = M_{\text{crit}}$ or it has two roots if that value is negative. (The ‘‘set of measure zero’’ possibility is a single root at M_{crit} .)

If $2M_{\min} < M_{\text{crit}} < 2M_{\max}$ then `m_and_eta()` searches for roots $2M_{\min} < M < M_{\text{crit}}$ and $M_{\text{crit}} < M < 2M_{\max}$ separately, else it looks for a root M in the range $2M_{\min} < M < 2M_{\max}$. If the lhs of (9.4.1) changes sign at the upper and lower boundaries of the interval, then a double-precision routine, similar to

the *Numerical Recipes* routine `rtsafe()`, is used to obtain the root with a combination of “safe” bisection and “rapid” Newton-Raphson.

If a root M is found in the desired range, then η is determined by (9.1.1) to be

$$\eta = \frac{5}{256} \left(\frac{M}{M_{\odot}} \right)^{-5/3} (\pi f_0 T_{\odot})^{-8/3} \frac{T_{\odot}}{\tau_0} \quad (9.4.4)$$

If $\eta \leq 1/4$ then the smaller and larger masses are calculated from

$$m_1 = \frac{M}{2} \left(1 - \sqrt{1 - 4\eta} \right) \quad m_2 = \frac{M}{2} \left(1 + \sqrt{1 - 4\eta} \right). \quad (9.4.5)$$

(If both roots for M correspond to $\eta \leq 1/4$ then an error message is generated and the routine aborts.) If both m_1 and m_2 are in the desired range $M_{\min} < m_1, m_2 < M_{\max}$ then `m_and_eta()` returns 1 and sets M and η appropriately, else it returns 0, leaving M and η unaffected.

Author: Bruce Allen, ballen@dirac.phys.uwm.edu

Comments: Although the arguments to this function are double precision floats, the values of m_1 and m_2 that may be inferred from them can generally only be determined to single precision, particularly in the neighborhood of $m_1 = m_2$. The reason is that in the vicinity of $\eta \sim 1/4$, a fractional error ϵ is the value of η produces a fractional error $\sqrt{\eta}$ in the masses.

9.5 Function: `template_area()`

`float template_area(struct Scope *Grid)`

This function computes the area of the enclosed region of parameter space shown in Figure 59.

The arguments are:

Grid: Input. This function uses only the minimum mass, maximum mass and the cut-off frequency f_0 fields of `Grid`.

The function returns the numerical value of the area in units of sec^2 . See the example in the following subsection.

The function uses an analytic expression for the area obtained by integration of formulae (9.1.1,9.1.2) for τ_0 and τ_1 given earlier. For example, to obtain the area of the trapezoidal region bounded above by the maximum-mass curve and below by the τ_0 axis, we integrate

$$\begin{aligned} A_1 &= \int_{m_{\max}}^{m_{\min}} \tau_1(m_{\min}, m) \frac{d\tau_0(m_{\min}, m)}{dm} dm \\ &= A_0 \left[\frac{m_{\min}}{M_{\odot}} \right]^{8/3} \left\{ \frac{-[3 + 2(4 + 2a)u + (5 + 9a)u^2]}{2u^2(1 + u)^{2/3}} \right. \\ &\quad + \frac{9a - 1}{\sqrt{3}} \arctan \left[\frac{1 + 2(1 + u)^{1/3}}{\sqrt{3}} \right] \\ &\quad \left. + \frac{9a - 1}{6} \log \left[\frac{1 + (1 + u)^{1/3} + (1 + u)^{2/3}}{1 - 2(1 + u)^{1/3} + (1 + u)^{2/3}} \right] \right\}_{u=m_{\min}/m_{\max}}^{u=1} . \end{aligned}$$

Here $a = 924/743$ and A_0 is a quantity with dimensions sec^2 given by

$$A_0 = \frac{18575}{49545216} \frac{M_{\odot}^2}{(\pi M_{\odot} f_0)^{14/3}} \left(\frac{c^3}{G} \right)^{8/3} .$$

The area A_2 under the minimum-mass curve can be obtained from the formula above by interchanging m_{\min} and m_{\max} . (If you wish to use geometrized units in which the solar mass is $4.92 \times 10^{-6} \text{ sec}$ simply set $G = c = 1$.) The area under the equal-mass curve A_3 can be obtained by performing a similar integration along the equal-mass curve

$$\begin{aligned} A_3 &= \int_{m_{\max}}^{m_{\min}} \tau_1(m, m) \frac{d\tau_0(m, m)}{dm} dm \\ &= \frac{60875}{2064384} \frac{M_{\odot}^2}{(\pi f_0 M_{\odot})^{14/3}} \left[\left(\frac{M_{\odot}}{m_{\min}} \right)^{8/3} - \left(\frac{M_{\odot}}{m_{\max}} \right)^{8/3} \right] \left(\frac{c^3}{G} \right)^{8/3} . \end{aligned}$$

These three results can be combined to give the total area enclosed

$$A_{\text{total}} = A_1 + A_2 - A_3 . \tag{9.5.1}$$

Equation (9.5.1) is the basis of `template_area()`; the next example shows an application of this function.

Author: Alan Wiseman, agw@tapir.caltech.edu

Comments: None.

9.6 Example: area program

This example uses the function `template_area()` described in the previous section to compute the area of the specified parameter space. The parameters specifying the region are set: the minimum and maximum mass in solar masses and the cut off frequency in seconds⁻¹. The numerical value of the area is returned and printed.

```
/* GRASP: Copyright 1997,1998 Bruce Allen */
#include "grasp.h"

int main() {
    struct Scope Grid;
    float area;

    /* Specify the parameter space */
    Grid.m_mn=0.8;
    Grid.m_mx=50.0;
    Grid.f_start=140.0;

    /* find area of parameter space */
    area=template_area(&Grid);

    /* and print it */
    printf("The area in parameter space is %f seconds^2.\n",area);
    return 0;
}
```

9.7 The match between two templates

When one performs a search for a gravitational wave signal in noisy instrumental data, one lays a grid of templates out in parameter space. For instance, if one uses τ_0 and τ_1 [see Eqs. (9.1.1) and (9.1.2)] as parameter space coordinates, then one's templates can be described as a set of points (τ_0^i, τ_1^i) (with i ranging from 1 to the total number of templates). One requires these points to be spaced such that no more than some *a priori* fraction of SNR is lost due to the discreteness of the template family.

Suppose one has decided that a set templates can lose no more than 3% SNR in a search. This means that if some arbitrary signal $b(t)$ is dropped onto the template grid, there must exist a template, $a(t)$, such that

$$\max_{t_0} \int_{-\infty}^{\infty} df \frac{\tilde{b}(f)\tilde{a}^*(f)}{S_h(f)} e^{-2\pi i f t_0} \geq .97 \left[\int_{-\infty}^{\infty} df \frac{|\tilde{b}(f)|^2}{S_h(f)} \right]^{1/2} \left[\int_{-\infty}^{\infty} df \frac{|\tilde{a}(f)|^2}{S_h(f)} \right]^{1/2} \quad (9.7.1)$$

("max t_0 " indicates the integral on the left hand side is to be maximized over all possible values of t_0 .) The integral on the left is the SNR obtained when the signal $b(t)$ is measured using the Wiener optimal filter corresponding to the template $a(t)$. The first integral on the right is the SNR obtained when $b(t)$ is measured with the Wiener optimal filter corresponding to a template $b(t)$; the second when the signal and template are both $a(t)$. (The integrals on the right hand side, in other words, describe the situation in which the template exactly matches the signal). For a detailed discussion of Wiener filtering, see Section 6.14.

To simplify this discussion, let us introduce the following inner product:

$$\langle a, b \rangle_{t_0} \equiv \int_{-\infty}^{\infty} df \frac{\tilde{a}^*(f)\tilde{b}(f)}{S_h(f)} e^{-2\pi i f t_0}. \quad (9.7.2)$$

[Note: this inner product is not to be confused with the inner product (a, b) defined in Eq. (6.14.9).] We will use the convention that not including the t_0 subscript on the angle bracket is equivalent to $t_0 = 0$. Eq. (9.7.1) can now be rewritten

$$\max_{t_0} \langle a, b \rangle_{t_0} \geq .97 \sqrt{\langle a, a \rangle \langle b, b \rangle}. \quad (9.7.3)$$

This motivates the definition of the *match* between $a(t)$ and $b(t)$:

$$\mu \equiv \max_{t_0} \frac{\langle a, b \rangle_{t_0}}{\sqrt{\langle a, a \rangle \langle b, b \rangle}}. \quad (9.7.4)$$

The match can be thought of as a distance measure between $a(t)$ and $b(t)$ (it is in fact one of the starting points for the metric that Owen defines in [5]). One uses the match function as a means of determining how one must space templates on the parameter space. If one requires that no more than 3% of possible SNR be lost due to template discreteness, then one must require adjacent templates to have a match $\mu = .97$.

The next few functions described in this manual are tools that can be used for calculating the match function and understanding how it varies over one's parameter space.

9.8 Function: `compute_match()`

```
float compute_match(float m1, float m2, float ch0tilde[], float ch90tilde[],  
float inverse_distance_scale, float twice_inv_noise[], float flo, float s_n0,  
float s_n90, int npoint, float srate, int err_cd_sprs, int order)
```

This function computes and returns the match function between a binary inspiral template that is stored in the arrays `ch0tilde[]` and `ch90tilde` and the binary inspiral template that corresponds to the binary system whose bodies have masses `m1` and `m2`.

The two phases of the “reference chirp”, `ch0tilde[]` and `ch90tilde[]`, are assumed to have been precomputed and run through the function `orthonormalize()`. (The parameters `s_n0` and `s_n90` are assumed to have been found when the reference chirp was orthonormalized.) This allows efficient computation of the match of many different templates with the reference chirp.

The arguments to the function are:

`m1`: Input. Mass of body 1 in the template that is cross-correlated with the reference chirp, solar masses.

`m2`: Input. Mass of Body 2 in the template, solar masses.

`ch0tilde`: Input. The FFT of the 0°-phase reference chirp.

`ch90tilde`: Input. The FFT of the 90°-phase reference chirp.

`inverse_distance_scale`: Input. The inverse distance to the binary system, in 1/Mpc. Because the match is a normalized correlation, this parameter isn’t physically relevant: moving the binary twice as far from the earth has no effect on the match. However, it may be computationally convenient to scale the inner products that go into the match definition by some amount to prevent numerical error.

`twice_inv_noise`: Input. Twice the inverse noise power spectrum, used for optimal filtering. For a more detailed description, see the routine `find_chirp()` (which is used within `compute_match()`).

`flo`: Input. The low-frequency cutoff to impose, in Hz. Within the code, this is used as the starting frequency of the templates; see `make_filters()`.

`s_n0`: Input. The normalization of `ch0tilde[]`, found using `orthonormalize()`.

`s_n90`: Input. The normalization of `ch90tilde[]`, found using `orthonormalize()`. Note that only the ratio `s_n0/s_n90` is physically relevant, because the match is normalized; if both `s_n0` and `s_n90` are multiplied by some constant, the match is unaffected.

`npoint`: Input. Defines the lengths of the various arrays: `ch0tilde[0..npoint-1]`, `ch90tilde[0..npoint-1]`, `twice_inverse_noise[0..npoint/2]`.

`srate`: Input. The sampling rate, in Hz. Used to convert between integer array time-domain subscripts and frequency subscripts. For example this is the sample rate of the 0°- and 90°-phase reference chirps, before they are FFT’d.

`err_cd_sprs`: Input. The error suppression code to be passed to the chirp generator; see `chirp_filters()`.

`order`: Input. Twice the post-Newtonian order; *i.e.*, the power of (v/c) used in the expansion. See `chirp_filters()`.

Author: Scott Hughes, hughes@tapir.caltech.edu

9.9 Function: match_parab()

```
int match_parab(float m1ref, float m2ref, float matchcont, int order, float
srates, float flo, float ftau, char *noisefile, float *semimajor, float *semimi-
nor, float *theta, float mcoef[])
```

This function attempts to find a parabolic fit to the match function near a reference template with masses (m1ref, m2ref). It works in (τ_0, τ_1) coordinates, and can use any noise curve listed in detectors.dat for its inner products when computing the match.

Let the coordinates of the reference chirp be (τ_0^r, τ_1^r) , and define $x \equiv \tau_0 - \tau_0^r, y \equiv \tau_1 - \tau_1^r$. Then, the fit to the match is of the form

$$\begin{aligned} \mu &= 1 + ax^2 + 2bxy + cy^2 \\ &= 1 + \begin{pmatrix} x & y \end{pmatrix} \cdot \begin{pmatrix} a & b \\ b & c \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}. \end{aligned} \quad (9.9.1)$$

Written in the form on the second line, it is easy to show that, if the match is in fact parabolic, it has surfaces of constant value that are ellipses. The (unnormalized) eigenvectors of this matrix are given by

$$\begin{aligned} \vec{v}_0 &= \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} (a-c)/2b + \sqrt{[(a-c)/2b]^2 + 1} \\ 1 \end{pmatrix} \\ \vec{v}_1 &= \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} (a-c)/2b - \sqrt{[(a-c)/2b]^2 + 1} \\ 1 \end{pmatrix}, \end{aligned} \quad (9.9.2)$$

and the eigenvalues are

$$\begin{aligned} \lambda_0 &= \frac{1}{2}(a+c) + \sqrt{\frac{1}{4}(a-c)^2 + b^2} \\ \lambda_1 &= \frac{1}{2}(a+c) - \sqrt{\frac{1}{4}(a-c)^2 + b^2}. \end{aligned} \quad (9.9.3)$$

(Note: because the match is maximal at $x = y = 0$ and falls off as x and y increase, the matrix is negative definite. The eigenvalues are therefore negative, and so $|\lambda_1| > |\lambda_0|$.) From these values, it is simple to construct the equimatch ellipse. If the value of the match on the contour is μ_{cont} , then the semimajor axis of the ellipse has length

$$r_{\text{major}} = \sqrt{\frac{\mu_{\text{cont}} - 1}{\lambda_0}}, \quad (9.9.4)$$

and the semiminor axis has length

$$r_{\text{minor}} = \sqrt{\frac{\mu_{\text{cont}} - 1}{\lambda_1}}. \quad (9.9.5)$$

The counterclockwise angle between the semimajor axis and the τ_0 axis is easily found from \vec{v}_0 :

$$\theta = \text{atan2}(v_{0y}, v_{0x}) = \arctan\left(1 / \left[(a-c)/2b + \sqrt{[(a-c)/2b]^2 + 1}\right]\right). \quad (9.9.6)$$

(Here, $\text{atan2}()$ is the C math library function; using $\text{atan2}()$ insures that the computer points θ to the correct quadrant of the τ_0, τ_1 plane.) If we now define normalized eigenvectors $\vec{e}_0 = \vec{v}_0/|\vec{v}_0|, \vec{e}_1 = \vec{v}_1/|\vec{v}_1|$, the ellipses are then easily constructed using the parametric curve

$$\begin{pmatrix} x \\ y \end{pmatrix} = r_{\text{major}} \cos \phi \vec{e}_0 + r_{\text{minor}} \sin \phi \vec{e}_1, \quad (9.9.7)$$

with ϕ varying from 0 to 2π .

The arguments to the function are:

`m1ref`: Input. Mass of body 1 for the reference chirp (solar masses).

`m2ref`: Input. Mass of body 2 for the reference chirp (solar masses).

`matchcont`: Input. The value of the match contour.

`order`: Input. Twice the post-Newtonian order to be used in computing the templates; *i.e.*, the power of (v/c) used in the post-Newtonian expansion.

`srate`: Input. The sample rate, in Hz. Used to convert between integer array time-domain subscripts and frequency subscripts. For example this is the sample rate of the 0° - and 90° -phase reference chirps, before they are FFT'd.

`flo`: Input. The low-frequency cutoff to impose, in Hz. Within the code, this is used as the starting frequency of the templates; see `make_filters()`.

`ftau`: Input. The frequency used to find τ_0 and τ_1 ; see Eqs. (9.1.1) and (9.1.2). Different authors use different conventions for this frequency—for example, Sathyaprakash uses the seismic wall frequency, whereas Owen uses the frequency at which the noise power is minimum. `ftau` is arbitrary, but should be used consistently: pick a value and stick with it.

`noisefile`: Input. A character string that specifies the name of a data file containing information about the noise power spectrum $P(f)$ of a detector. See `noise_power()` for extended discussion.

`semimajor`: Output. The semimajor axis of the ellipse along which the match has the value `matchcont`.

`semiminor`: Output. The semiminor axis of the ellipse.

`theta`: Output. The counterclockwise angle, in radians, between `semimajor` and the τ_0 axis.

`mcoef`: Output. The array `mcoef[0..2]` contains the coefficients of the parabolic fit to the match:

$$\mu_{\text{fit}} = 1 + \text{mcoef}[0]x^2 + \text{mcoef}[1]xy + \text{mcoef}[2]y^2.$$

The function works by sampling many templates in (τ_0, τ_1) coordinates that are close to the template with masses (`m1ref`, `m2ref`). Periodically, it computes the best parabolic fit to the data it has gathered so far and constructs the elliptical contour corresponding to that fit. It then takes N_{ell} steps around this ellipse and compares the value of the match predicted by the fit with the actual match value at each point. It then computes the following “ χ^2 -like” statistic:

$$\varepsilon = \frac{1}{N_{\text{ell}}} \sum_{i=1}^{N_{\text{ell}}} \left(\frac{\mu_{\text{actual}}^i - \mu_{\text{fit}}^i}{10^{-3}} \right)^2. \quad (9.9.8)$$

If $\varepsilon = 1$, then each fit point differs from the match by 10^{-3} . A “good” fit will have ε of order 1.

This function returns 0 if a good fit is not found (ε is greater than 5 yet more than 250 templates have been used to generate fit data), and 1 otherwise. If a good fit is not found, then the match is not parabolic in the vicinity of the template (`m1ref`, `m2ref`) down to $\mu = \text{matchcont}$. This is typically the case if the masses are large (so that there are few cycles measured, and relativistic effects are very important), and if the value of `matchcont` is too far from 1. For instance, with the LIGO 40-meter prototype, `match_parab()` cannot find a parabolic fit to the .97 match contour for a binary with $m_1 = 1.2M_\odot$, $m_2 = 1.6M_\odot$; but it *does* find a parabolic fit for this binary at the .99 match contour.

Author: Scott Hughes, hughes@tapir.caltech.edu

9.10 Function: match_cubic()

```
int match_cubic(float m1ref, float m2ref, float matchcont, int order, float
srate, float flo, float ftau, char *noisefile, float *semimajor, float *semimi-
nor, float *theta, float mcoef[])
```

This function is almost identical to `match_parab()`, except that it attempts to fit the match to a cubic form:

$$m = 1 + ax^2 + 2bxy + cy^2 + dx^3 + ey^3 + fx^2y + gxy^2 \quad (9.10.1)$$

The arguments to the function are:

`m1ref`: Input. Mass of body 1 for the reference chirp (solar masses).

`m2ref`: Input. Mass of body 2 for the reference chirp (solar masses).

`matchcont`: Input. The value of the match contour.

`order`: Input. Twice the post-Newtonian order to be used in computing the templates; *i.e.*, the power of (v/c) used in the post-Newtonian expansion.

`srate`: Input. The sample rate, in Hz. Used to determine the spacing of frequency bins for the templates.

`flo`: Input. The low-frequency cutoff to impose, in Hz. Within the code, this is used as the starting frequency of the templates; see `make_filters()`.

`ftau`: Input. The frequency used to find τ_0 and τ_1 ; see Eqs. (9.1.1) and (9.1.2). Different authors use different conventions for this frequency—for example, Sathyaprakash uses the seismic wall frequency, whereas Owen uses the frequency at which the noise power is minimum. `ftau` is arbitrary, but should be used consistently: pick a value and stick with it.

`noisefile`: Input. A character string that specifies the name of a data file containing information about the noise power spectrum $P(f)$ of a detector. See `noise_power()` for extended discussion.

`semimajor`: Output. The semimajor axis of the ellipse along which the match has the value `matchcont`.

`semiminor`: Output. The semiminor axis of the ellipse.

`theta`: Output. The counterclockwise angle, in radians, between `semimajor` and the τ_0 axis.

`mcoef`: Output. The array `mcoef[0..6]` contains the coefficients of the parabolic fit to the match:
$$\mu_{\text{fit}} = 1 + \text{mcoef}[0]x^2 + \text{mcoef}[1]xy + \text{mcoef}[2]y^2 + \text{mcoef}[3]x^3 + \text{mcoef}[4]y^3 + \text{mcoef}[5]x^2y + \text{mcoef}[6]xy^2.$$

The function works in almost exactly the same manner as `match_parab()`. In particular, it constructs an ellipse using the parabolic piece of the cubic fit, and checks the goodness of the fit along that ellipse. Because the ellipse is not made from the full functional form of the fit, the fit does not have constant value along the ellipse. Thus, `match_cubic()` does not really find contours with constant match value `matchcont`. The ellipses it finds, however, generally have match values fairly close to `matchcont`; and, more importantly, the match values along the ellipse are never less than `matchcont`.

Author: Scott Hughes, hughes@tapir.caltech.edu

9.11 Example: match_fit program

This program will try to find the fit to the match function about some template. It is called with four arguments: the mass of body 1 (in solar masses), the mass of body 2 (in solar masses), the value of the match for which it tries to fit, and (twice) the order of the post-Newtonian expansion used to compute the templates. For example, `match_fit 1.2 1.8 .98 4` will try to find a fit to the .98 match contour near the template for the $1.2 M_{\odot} - 1.8 M_{\odot}$ using post-2-Newtonian templates.

The program first attempts to find a parabolic fit; if it is unable to do so, it then tries a cubic. If the cubic fails, you are in a region of parameter space where the match is badly behaved. This is typically the case if you ask for masses that are too large—for example, no fit can be found near a $5 M_{\odot} - 5 M_{\odot}$ solar mass binary with the LIGO 40-meter prototype noise curve. When the masses are large, the system radiates very few gravitational-wave cycles in the instrument's frequency band; and, those cycles typically correspond to a strongly relativistic regime of inspiral. If you find yourself in this circumstance, either give up on the large mass binaries, or try to find a fit at a match level closer to 1.

```
/* GRASP: Copyright 1997,1998 Bruce Allen */
#include "grasp.h"

#define DETECTOR_NUM 15      /* Smooth fit to Caltech 40m prototype */
#define FLO 120.            /* Hz - low frequency cut off for filtering */
#define FTAU 140.          /* Hz - frequency used in definitions of
                           tau0, tau1. */

/*#define DETECTOR_NUM 8 Caltech 40m prototype */
/*#define DETECTOR_NUM 1 LIGO initial interferometer */
/*#define DETECTOR_NUM 12 LIGO Advanced interferometer */

int main(int argc, char **argv)
{
    float *pfit, *cfit, semimajor, semiminor, theta;
    float m1, m2, matchcont;
    float srate=50000;
    float site_parameters[9];
    char noise_file[128], whiten_file[128], site_name[128];
    int order, tstp, tstc;

    /* Check that the program is called with the correct number of
       arguments; print out argument information if it's not. */
    if(argc != 5) {
        fprintf(stderr, "4 Arguments: 1. Mass of body 1 (solar masses)\n");
        fprintf(stderr, "          2. Mass of body 2 (solar masses)\n");
        fprintf(stderr, "          3. Match contour match value;\n");
        fprintf(stderr, "          4. Waveform order [power of (v/c)]\n");
        fprintf(stderr, "\nExample: match_fit 1.2 1.6 .97 4\n");
        exit(0);
    }
    /* Assign arguments to variables */
    m1=atof(argv[1]);
    m2=atof(argv[2]);
    matchcont=atof(argv[3]);
    order=atoi(argv[4]);

    /* Get the file names for the desired noise curve */
    detector_site("detectors.dat", DETECTOR_NUM, site_parameters,
                 site_name, noise_file, whiten_file);

    printf("\nEvaluating templates for detector: %s using data from file: \"%s\"\n\n",
```

```
    site_name,noise_file);

/* Allocate memory for the coefficients used in the parabolic fits */
pfit=(float *)malloc(sizeof(float)*3);
cfit=(float *)malloc(sizeof(float)*7);

/* Try to find a parabolic fit */
tstp=match_parab(m1,m2,matchcont,order,srate,FLO,FTAU,noise_file,
                &semimajor,&semiminor,&theta,pfit);
if(tstp) {
    printf("Found a parabolic fit to the match around template with\n");
    printf("m1=%f, m2=%f.\n\n",m1,m2);
    printf("Semimajor axis of best fit ellipse:    %e ms\n",
           semimajor*1.e3);
    printf("Semiminor axis of best fit ellipse:    %e ms\n",
           semiminor*1.e3);
    printf("Angle between semimajor and tau0 axis: %f rad\n",theta);
    printf("Fit: m = 1 + %e x^2 + %e xy + %e y^2\n",
           pfit[0],pfit[1],pfit[2]);
    printf("[where x=dtau0, y=dtaul]\n");
} else
    printf("Unable to find parabolic fit.  Attempting cubic fit.\n");

/* If the parabola failed, try to find a cubic fit */
if(!tstp) {
    tstc=match_cubic(m1,m2,matchcont,order,srate,FLO,FTAU,noise_file,
                    &semimajor,&semiminor,&theta,cfit);
    if(tstc) {
        printf("Found a cubic fit to the match around template with\n");
        printf("m1=%f, m2=%f.\n\n",m1,m2);
        printf("Using ellipse constructed from parabolic part of cubic.\n");
        printf("Semimajor axis of best fit ellipse:    %e ms\n",
               semimajor*1.e3);
        printf("Semiminor axis of best fit ellipse:    %e ms\n",
               semiminor*1.e3);
        printf("Angle between semimajor and tau0 axis: %f rad\n",theta);
        printf("Fit: m = 1 + %e x^2 + %e xy + %e y^2\n",
               cfit[0],cfit[1],cfit[2]);
        printf("      + %e x^3 + %e y^3 + %e x^2 y + %e xy^2\n",
               cfit[3],cfit[4],cfit[5],cfit[6]);
        printf("[where x=dtau0, y=dtaul]\n");
    } else {
        printf("Unable to find a cubic fit.  Try looking for a match\n");
        printf("contour at smaller match value; or, give up on this\n");
        printf("mass regime.\n");
    }
}

return 0;
}
```

Author: Scott Hughes, hughes@tapir.caltech.edu

9.12 Structure: struct cubic_grid

This structure is used to store precomputed coefficients of the cubic fit to the match function, generated by `match_cubic()` on an equally spaced grid in the m_1, m_2 parameter space. The structure stores the coefficients as well as all the information required to generate, retrieve, and interpolate among them. The fields of this structure are:

```
struct cubic_grid {
    int n; The number of points along the side of the grid.

    float m_mn; The minimum mass of an object in the parameter space covered by the grid (solar masses).
    float m_mx; The minimum mass of an object in the parameter space covered by the grid (solar masses).
    float dm; The spacing between grid points (solar masses); equal to  $(m_{mx} - m_{mn}) / (n - 1)$ .
    float match; The match level (between 0 and 1) out to which the cubic fit was made.
    float angle; The angle (radians) counterclockwise from the  $\tau_0$  axis to the  $x$  axis (see below).
    int order; Twice the post-Newtonian order of the chirp templates used to compute the match function.
    float srate; The sampling rate of the chirp templates used to compute the match function.
    float flo; The initial frequency of the chirp templates used to compute the match function.
    float ftau; The reference frequency used to define the  $\tau_0, \tau_1$  coordinates.
    int detector; The index of the detector site in the data file detectors.dat, used to identify a noise curve for computing the match function.

    float ***coef; A pointer to the array of coefficients.
};
```

The `cubic_grid.coef` field points to an array of the form `coef[0..n-1][0..n-1][0..9]`. The first two indices `[i][j]` identify a point in the mass parameter space: $m_1 = m_{mn} + i \times dm$ and $m_2 = m_{mn} + j \times dm$. The third index identifies a particular coefficient computed at that point. The individual coefficients are defined as follows: The first 7 entries `[0..6]` are the actual coefficients of the cubic fit to the match function μ :

$$\mu = 1 + [0]x^2 + [1]xy + [2]y^2 + [3]x^3 + [4]y^3 + [5]x^2y + [6]xy^2, \quad (9.12.1)$$

where x, y are small displacements in directions at an angle `cubic_grid.angle` counterclockwise from the τ_0, τ_1 directions, respectively. If one considers only the quadratic part of this fit, the equation $\mu = \text{cubic_grid.match}$ defines an ellipse in the x, y plane. Entries `[7]` and `[8]` are then the semimajor and semiminor axes of this ellipse, respectively (in units of seconds), and entry `[9]` is the angle (in radians) counterclockwise from the x axis to the semimajor axis. The entries `[7..9]` can be computed without too much difficulty from the entries `[0..2]` and the value of `match`, but it can be useful to have them precomputed.

9.13 Function: generate_cubic

```
void generate_cubic(struct cubic_grid grid, char *detectors_file,  
                  const char *outfile, const char *logfile);
```

This routine computes the coefficients of the cubic fit to the match function on a mesh in parameter space, and writes the results to an ASCII textfile, suitable for reading by the routine `read_cubic()` (section 9.15).

The arguments are:

`grid`: Input/Output. This structure contains the parameters for the computation of the cubic fits. All of the fields except for `grid.dm` and `grid.coef` must be set; those fields are the ones that are computed.

`detectors_file`: Input. The name of a data file containing detector site information, such as `detectors.dat`. This is used to get a noise file for computing the match function.

`outfile`: Input. The name of the output file to which the coefficients and related information will be written.

`logfile`: Input. The name of a log file which tracks the progress of this routine (since it can take several hours to generate a reasonable grid).

The output file is an ASCII textfile containing the fields of the structure `grid`. Each field except the `coef` field is printed on a separate line of the output file. The `coef` data is written as $0.5 \times \text{grid.n} \times (\text{grid.n}+1)$ lines of 10 floating point numbers; each line represents the 10 coefficients `coef[i][j][0..9]` for a given `i, j`. The lines are ordered by increasing `j` from 0 to `i` for each `i` from 0 to `grid.n-1`. Integers are printed exactly; floats are printed in 10-digit precision exponential notation.

One should also note that this routine can take quite a long time to run: on a 100 MHz pentium it typically takes 10 to 15 minutes per point in the grid. This is the reason for creating the log file to track the routine's progress.

Author: Teviet Creighton, teviet@tapir.caltech.edu

9.14 Function: regenerate_cubic

```
int regenerate_cubic(char *detectors_file, const char *infile,  
                    const char *outfile, const char *logfile);
```

It may happen that the routine `generate_cubic()` terminates before completing the entire coefficient grid. Since each grid point takes so long to compute, it would be foolish to discard those already generated. This routine, therefore, reads in a partially-complete data file, and then continues the computation where `generate_cubic()` left off. The results are written to a new data file (leaving the original file incomplete). It returns 0 upon successful completion, or 1 if the data file was absent or corrupt. `regenerate_cubic()` also creates its own log file to track its progress.

The arguments are:

`detectors_file`: Input. The name of a data file containing detector site information, such as `detectors.dat`. This is used to get a noise file for computing the match function.

`infile`: Input. The name of the incomplete data file of coefficients.

`outfile`: Input. The name of the data file where this routine stores its results.

`logfile`: Input. The name of a log file which tracks the progress of this routine.

Author: Teviet Creighton, teviet@tapir.caltech.edu

9.15 Function: read_cubic

```
int read_cubic(struct cubic_grid *grid, char *infile);
```

This routine reads a textfile generated by `generate_cubic()`, stores the data in a variable `grid` of type `struct cubic_grid`, and passes this structure back. `read_cubic()` itself returns 0 after successful completion, or 1 if the data file was absent or corrupt (in which case `grid` is left unchanged).

Note that memory for the coefficient array `grid.coef` is allocated in this routine; to free this memory, call `free_cubic(grid)`. Allocating and de-allocating this array requires some care: since `grid.coef[i][j][k]` is necessarily symmetric in the first two indices, the pointers `grid.coef[i][j]` and `grid.coef[j][i]` have been explicitly set to point to the same memory location, in order to save memory.

The arguments are:

`grid`: Output. The coefficient array and related data read from the data file.

`infile`: Input. The name of the data file.

Author: Teviet Creighton, teviet@tapir.caltech.edu

9.16 Function: `get_cubic`

```
int get_cubic(float m1, float m2, struct cubic_grid grid, float *coef);
```

This routine computes the coefficients of a cubic fit to the match function at a specified point in parameter space, by linear interpolation of precomputed coefficients on a grid in parameter space. It returns 0 if successfully executed, or 1 if the point (m1,m2) lies outside of the grid. In the latter case, `get_cubic()` will compute extrapolated coefficients, but these are unreliable.

The arguments are:

`m1`: Input. One of the binary mass coordinates of the requested point in parameter space.

`m2`: Input. The other binary mass coordinates of the requested point in parameter space.

`grid`: Input. The data structure containing the precomputed coefficients and the information required to retrieve them.

`coef`: Output. The array `coef[0..9]` is filled with the interpolated coefficients.

Author: Teviet Creighton, teviet@tapir.caltech.edu

9.17 Function: free_cubic

```
void free_cubic(struct cubic_grid grid);
```

Frees the memory allocated to the array `grid.coef` by `read_cubic()`.

The argument is:

`grid`: Input. The `cubic_grid` structure whose coefficient array is to be freed.

Author: Teviet Creighton, teviet@tapir.caltech.edu

9.18 Function: transform_cubic

```
void transform_cubic(struct cubic_grid *grid, float angle, float match);
```

This routine applies a rotation to the coefficients stored in *grid, and rescales the equimatch ellipses to a new match level.

The arguments are:

`grid`: Input/Output. The structure containing the coefficients to be transformed.

`angle`: Input. The new value of `grid.angle` (the elements `(*grid).coef[i][j][0..6,9]` will be transformed to fit this new angle).

`match`: Input. The new value of `grid.match` (the elements `(*grid).coef[i][j][7,8]` will be rescaled according to this value).

Author: Teviet Creighton, teviet@tapir.caltech.edu

Comments: The result of this transformation is not quite the same as if the grid were originally generated with the new value of the match. During initial generation of the grid, the match field also specifies the domain over which the cubic fit is made, as well as setting the scale for the equimatch ellipse axes. This routine only rescales the ellipses; it does not regenerate a cubic fit over a new match range.

9.19 Example: make_grid program

This example program uses the function `generate_cubic()` to create a grid of match function coefficients over the mass range of 0.8 to 3.2 solar masses, down to a match level of 0.98, using the smooth fit to the LIGO noise power spectrum in computing the match function. The resulting grid structure is stored in the data file `cubic_coef_40meter_m=0.8-3.2.ascii`. Note that the program can take quite a long time to run: approximately 20 hours on a 100 MHz Pentium computer. The program's progress is tracked in the log file `cubic_coef_40meter_m=0.8-3.2.log`.

The program makes frequent calls to the routine `match_cubic()`, which generates a lot of messages in `stderr`, which are generally unimportant unless things go wrong. You'll probably want to redirect `stderr` to some junk file. The important progress record is stored in `cubic_coef_40meter_m=0.8-3.2.log`, which, when complete, looks like this:

Using Caltech-40 noise curve from the file `noise_40smooth.dat`.
Generating match coefficients at 91 points:

```
.  
..  
...  
....  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....
```

The source code for `make_grid.c` is listed below:

```
/* GRASP: Copyright 1997,1998 Bruce Allen */  
#include "grasp.h"  
  
int main(int argc, char **argv)  
{  
    struct cubic_grid grid;  
  
    /* Set grid parameters. */  
    grid.n=13;  
    grid.m_mn=0.8;  
    grid.m_mx=3.2;  
    grid.match=0.98;  
    grid.angle=0.0;  
    grid.order=4;  
    grid.srate=50000.0;  
    grid.flo=120.0;  
    grid.ftau=140.0;  
    grid.detector=15; /* Smooth fit to Caltech 40m prototype. */  
    /* grid.detector=8; Caltech 40m prototype. */  
    /* grid.detector=1; LIGO initial interferometer. */
```

```
/* grid.detector=12; LIGO advanced interferometer. */  
  
/* Generate grid of cubic-fit coefficients */  
generate_cubic(grid, "detectors.dat",  
               "cubic_coef_40meter_m=0.8-3.2.ascii",  
               "cubic_coef_40meter_m=0.8-3.2.log");  
return 0;  
}
```

Author: Teviet Creighton, teviet@tapir.caltech.edu

9.20 Example: read_grid program

This example program uses the function `read_cubic()` to read the data file `cubic_coef_40meter_m=0.8-3.2.ascii` generated by `make_grid` (section 9.19), and return the coefficients of the cubic fit to the match function for any point m_1, m_2 in mass space. Here are some sample runs of `read_cubic`:

```
% read_grid
Usage: read_grid M1 M2
% read_grid 1.3 1.5
Match coefficients: -5.075e+03  2.977e+04 -4.602e+04
                   -2.703e+04  6.656e+06  7.090e+05 -4.044e+06
Axis lengths:      9.245e-03  6.272e-04
Axis angle:        3.144e-01
% read_grid 1.3 3.5
(1.300,3.500) lies outside of grid.  Extrapolating...
Match coefficients: -4.159e+03  2.058e+04 -2.716e+04
                   2.866e+04  2.770e+06  1.424e+05 -1.620e+06
Axis lengths:      9.631e-03  8.116e-04
Axis angle:        3.688e-01
%
```

Masses are entered in solar masses. However, the coefficients are the cubic fit coefficients in τ_0, τ_1 space, so the first three coefficients have units of s^{-2} , the next four coefficients of s^{-3} , and the axis lengths of s . The angle (counterclockwise from the τ_0 axis to the principle axis of the equimatch ellipse) is in radians.

```
/* GRASP: Copyright 1997,1998 Bruce Allen */
#include "grasp.h"

int main(int argc, char **argv)
{
    struct cubic_grid grid;

    /* Set grid parameters. */
    grid.n=13;
    grid.m_mn=0.8;
    grid.m_mx=3.2;
    grid.match=0.98;
    grid.angle=0.0;
    grid.order=4;
    grid.srate=50000.0;
    grid.flo=120.0;
    grid.ftau=140.0;
    grid.detector=15; /* Smooth fit to Caltech 40m prototype. */
    /* grid.detector=8; Caltech 40m prototype. */
    /* grid.detector=1; LIGO initial interferometer. */
    /* grid.detector=12; LIGO advanced interferometer. */

    /* Generate grid of cubic-fit coefficients */
    generate_cubic(grid, "detectors.dat",
                  "cubic_coef_40meter_m=0.8-3.2.ascii",
                  "cubic_coef_40meter_m=0.8-3.2.log");

    return 0;
}
```

Author: Teviet Creighton, teviet@tapir.caltech.edu

9.21 Function: `template_grid()`

```
void template_grid(struct Scope *Grid)
```

This function evolved from `grid4.f`, a FORTRAN routine written by Sathyaprakash. This function lays down a grid of templates that cover a particular mass range (the region inside the distorted triangle shown in Figure 59).

The arguments are:

`Grid`: Input/Output. This function uses as input all of the fields of `Grid` except for `Grid.n_tmplt` and `Grid.templates`. On return from `template_grid` these latter two fields are set. The function uses `malloc()` to allocate storage space and creates in this space an array containing `Grid.n_tmplt` objects of type `Template`. If you wish to free the memory, call `free(Grid.templates)`.

It is easy to cover the parameter space shown in Figure 59 with ellipses. However each ellipse represents a filter, and filtering takes computer time and memory, so the real problem is to cover the parameter space completely, using the *smallest possible number* of templates. This is a non-trivial *packing problem*; while our solution is certainly not optimal, it is quite close.

The algorithm used to place the templates works in coordinates (x_0, x_1) which are rotated versions of (τ_0, τ_1) , aligned along the minor and major (or major and minor) axes of the template ellipses. The input angle `Grid.theta`, in the range $(-\pi, \pi)$, is the counterclockwise angle through which the (x_0, x_1) axes need to be rotated to bring them into alignment with the principal axis of the template ellipses.

Although each template is an ellipse, the problem of packing templates onto the parameter space can be more easily described in terms of a more familiar packing problem: packing pennies on the plane. One can always transform an ellipse into a circle by merely scaling one coordinate uniformly while leaving the other coordinate unchanged. So we introduce coordinates x_1 along the major diameter and x_0 along the minor diameter of the ellipse, and then “shrinking” the x_1 coordinate by the ratio of major to minor diameters. In this way the ellipses are transformed into circles.

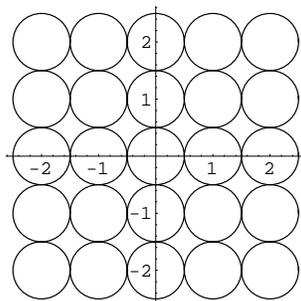


Figure 60: Covering a plane with a square lattice of pennies (or templates) leaves 21% of the area exposed

First, a template is laid down at the point where the equal mass line intersects the maximum mass line. Then additional templates are placed along the equal mass line, at increasing values of x_0 . These templates are staggered up and down in the x_1 direction. After laying down this set of templates, the remaining part of parameter space is covered with additional templates, in columns starting at each of the previously determined template locations. These columns have the same value of x_0 as the previously determined templates but increasing values of x_1 . The columns are continued until the “leading edge” of the final template lies outside the parameter space.

We can describe the packing (and the “efficiency”) of the packing in terms of the penny-packing problem. Suppose we start by setting pennies of radius $1/2$ on all points in the plane with integer coordinates,

as shown in Figure 60. It is easy to show that the fraction of the plane (i.e., parameter space!) which is not covered by any pennies is $\epsilon = 1 - \pi/4 = 0.214\dots$ or about 21%.

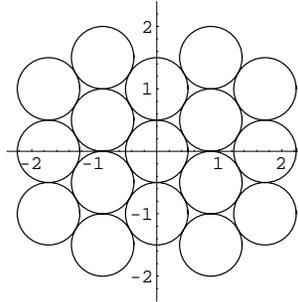


Figure 61: Staggering the pennies (or templates) decreases the uncovered fraction of the plane to 9.3%

Now suppose that we “stagger” the pennies as shown in Figure 61. In this case, the fraction of area not covered is $\epsilon = 1 - \frac{\pi}{2\sqrt{3}} = 0.093\dots$ or about 9.3%. If we wish to completely cover the missing bits of the plane, then we can do so by increasing the radius of each penny by $\sqrt{5/4}$ (or, equivalently, by moving the points at which the pennies lie closer together by that same factor). The resulting diagram is shown in Figure 62. By increasing the number-density of pennies on the plane by 25% we have successfully covered up the remaining 9.3% of the area.

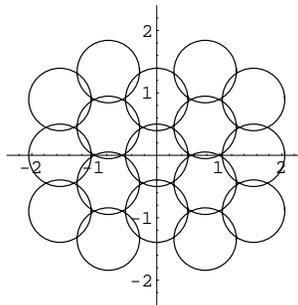


Figure 62: Decreasing the spacings of the pennies (or templates) by a factor of $(5/4)^{1/2} = 1.118\dots$ then covers the entire plane.

Now it is not possible to implement this algorithm exactly, because we are not attempting to cover the entire plane, but rather only a finite region of it. You might think that we could just start laying down templates in the same way as for Figure 62 and stick in a few extra ones for any parts of the parameter space which were not covered, but unfortunately this would then lead us to place templates centered at points in (τ_0, τ_1) space that do not correspond to $\eta \leq 1/4$, and for which the very meaning of a “chirp” is ill-defined.

The code in `template_grid()` thus uses a heuristic method to place templates, trying whenever possible to stagger them in the same way as Figure 62 but then shifting the center locations when necessary to ensure that the template corresponds to physical values of the mass parameters m_1 and m_2 . This is often referred to as “hexagonal packing”. In practice, to see if this placement has been successful or not, the function `plot_template()` can be used to visually examine the template map.

Table 8 gives information about the appropriate template sizes, spacings and orientations as found in the recent literature, and using the `match_fit` example program. The angle θ is the angle to the axis of the ellipse whose radius is r_1 , measured counterclockwise from the τ_0 axis. The other radius (semi-axis)

Author	Detector	f_0 /Hz	θ /rad	radius r_1 (msec)	radius r_2 (msec)
Sathyaprakash	Caltech 40m (Nov 94)	140	0.307	8.0	0.6
Owen	Initial LIGO	200	0.5066	2.109	0.162
Owen	Advanced LIGO	70	0.4524	3.970	0.352

Table 8: Orientation and dimensions of 0.97 ambiguity templates.

of the ellipse has length r_2 . Equation (3.16-18) of reference [5] do not appear to agree with Table 8, but that is because the $r_i = dx_i$ of [5] are defined by $(dx_i)_{\text{Owen}} = dl_i/\sqrt{E_i}$. The dl_i are the edge lengths of a hypercube in dimension N , chosen so that if templates are centered on its vertices, then the templates touch in the center of the cube, so that $(dx_i)_{\text{Owen}} = dl_i/\sqrt{E_i}$. In our $N = 2$ dimensional case, this gives $r_i = dx_i = (dx_i)_{\text{Owen}}/\sqrt{2}$. Note also that in this table, Owen and Sathyaprakash use different definitions of f_0 , so that their results may not be directly compared. In Owen's case, f_0 refers to the frequency of maximum sensitivity of the detector, whereas in Sathyaprakash's case it refers to the frequency at which the chirp first enters the bandpass of the detector. In the case of the November 1994 data set, we quote two different sizes and orientations for the ellipses, depending upon the choice of f_0 .

Author: Bruce Allen, ballen@dirac.phys.uwm.edu

Comments: This routine evolved from `grid4.f`, which was written by Sathyaprakash. The method used to stagger templates is heuristic, and could perhaps be improved. Very small regions of the parameter space along the equal-mass line ($\eta = 1/4$) may not be covered by any templates.

9.22 Function: plot_template()

```
void plot_template(char *filename, struct Scope Grid, int npages, int number)
```

This function generates a PostScript (tm) file that draws a set of templates on top of the region of parameter space which they cover.

The arguments are:

filename: Input. Pointer to a character string. This is used as the name of the output file, into which postscript output is written. We suggest that you use “.ps” as the final three characters of the filename. These files are best viewed using GhostView.

Grid: Input. The mass range specified by Grid is used to draw an outline of the region in (τ_0, τ_1) parameter space covered by the mass range, and an ellipse for each template included in Grid is then drawn on top of this outline.

npages: Input. If there are more than a few templates (and there can be thousands, or more) it is impossible to view this graphical output unless it is spread across many pages. npages specifies the number of pages to spread the output across. We suggest at least one page per hundred templates.

number: Input. Each template specified in Grid is numbered by the field Grid.n_tmplt. If number is set to 1, then when each ellipse is drawn in parameter space, the number of the template is placed inside the ellipse so that the particular template associated with each ellipse may be easily identified. If number is set to 0, then the templates are not identified in this way; each template is simply drawn as an empty ellipse.

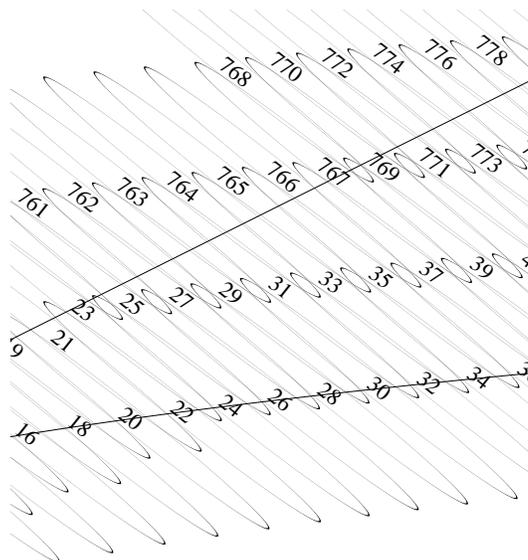


Figure 63: Part of some sample output from plot_template().

Note that the output postscript file is designed to be edited if needed to enable clear viewing of details. Each file is broken into pages. At the beginning of each page are commands that set the magnification scale of each page, and determine if the page will be clipped at the boundaries of the paper or not. You can edit these

lines in the postscript file to enable you to “zoom in” on part of the parameter space, if desired. By turning off the clipping, you can easily move off the boundaries of a given page, if desired. Some sample output from `plot_template()` is shown in Figure 63. (In fact, this is part of the output file produced by the example program, showing a small number of the total of 1001 templates required).

Author: Bruce Allen, ballen@dirac.phys.uwm.edu

Comments: Another option should be added, to print out at the center of each template, the mass parameters m_1 and m_2 associated with the template.

9.23 Example: template program

This example lays down an optimal grid of templates covering parameter space. It also outputs a postscript file (best viewed with GhostView) which shows the elliptical region of parameter space covered by each template.

```
/* GRASP: Copyright 1997,1998 Bruce Allen */
#include "grasp.h"

int main() {
    struct Scope Grid;

    /* Set parameters for the inspiral search in CIT 40 meter */
    Grid.m_mn=0.9;
    Grid.m_mx=3.0;
    Grid.theta=0.307;
    Grid.dp=2*0.008;
    Grid.dq=2*0.0006;
    Grid.f_start=140.0;

    /* construct template set covering parameter space */
    template_grid(&Grid);

    /* create a postscript file showing locations of templates */
    plot_template("templates_40meter.ps",Grid,15,1);
    return 0;
}
```

Part of a typical picture contained in the output file `temp_list.ps` is shown in Figure 63 (though for different parameters than those shown above).

9.24 Example: `multifilter` program

This example implements optimal filtering by a bank of properly-spaced templates. One could do this with trivial modifications of the example `optimal` program given earlier. Here we have shown something slightly more ambitious. The `multifilter` program is an MPI-based parallel-processing code, designed to run on either a network of workstations or on a dedicated parallel machine. It is intended to illustrate a particularly simple division of labor among computing nodes. Each segment of data (of length `NPOINT`) is broadcast to the next available node. That node is responsible for filtering the data through a bank of templates, chosen to cover the mass range from `MMIN` to `MMAX`. The output of each one of these filters is a set of 11 signals, which measure the following quantities:

1. The largest signal-to-noise ratio (SNR) at the output of the filter, for the given segment of data,
2. The distance for an optimally-oriented source, in Mpc, at which the SNR would be unity.
3. The amplitude α of the zero-degree phase chirp matching the observed signal.
4. The amplitude β of the ninety-degree phase chirp matching the observed signal.
5. The offset of the best-fit chirp into the given segment of data
6. The offset of the impulse into the given segment of data, which would produce the observed output.
7. The time of that impulse, measured in seconds from the start of the data segment,
8. The time (in seconds, measured from the start of the data segment) at which an inspiral, best fitting the observed filter output, would have passed through the start frequency `FLO`.
9. The time (in seconds, measured from the start of the data segment) at which an inspiral, best fitting the observed filter output, would have passed through coalescence.
10. The observed average value of the output SNR (should be approximately unity).
11. The probability, using the splitup technique described earlier, that the observed filter output is consistent with a chirp plus stationary detector noise.

For completeness, we give this code in its entirety here. We also show some typical graphs produced by the MPE utility `nupshot` which illustrates the pattern of communication and computation for an analysis run. For these graphs, the analysis run lasted only about four minutes, and analyzed about three minutes of IFO data. We have performed an identical, but longer run, which analyzed about five hours of IFO output in just over three hours, running on a network of eight SUN workstations. The data is analyzed in 6.5 second segments, each of which is processed through a set of 66 filter templates completely covering the mass range from 1.2 to 1.6 solar masses. For the run that we have profiled here, `STORE_TEMPLATES` is set to 1. This means that each slave allocates memory internally for storing the Fourier-transformed chirp signals; the slaves only compute these once. However this does place demands on the internal storage space required - in the run illustrated here each individual process allocated about 34 Mbytes of internal memory. Another version of the code has also been tested; in this version the slave nodes compute the filters and Fourier transform them each time they are needed, for each new segment of data. This code has `STORE_TEMPLATES` set to 0. This is less efficient computationally, but requires only a small amount of internal storage. For a given hardware configuration, the optimal balance between these extremes, and between the amount of redundant broadcasting of data, depends upon the relative costs of communication and computation, and the amount of internal storage space available.

Based on these figures, it is possible to provide a rough table of computation times. These are given in tabular form in Table 9.

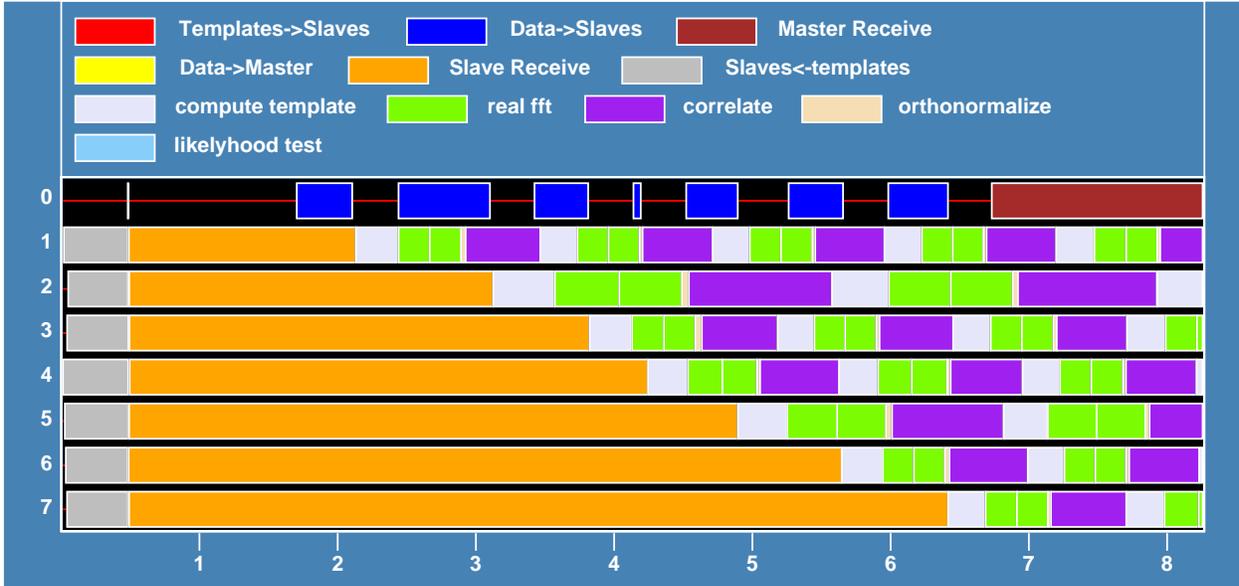


Figure 64: Output of the nupshot profiling tool, showing the behavior of the multifilter program running on a workstation network of 8 machines (the fastest of these are Sparc-20 class processors). This shows the first 8 seconds of operation (time on the horizontal axis). The gray segments show the slave processes receiving the template list. During the orange segments, the slave processes are waiting for data; the blue segments show the master transmitting data to each slave. During the light gray segments, the slaves are computing the templates, during the green segments they are computing the FFT's of those templates, and during the purple segments they are correlating the data against the templates. During the brown segment, the master is waiting to receive data back from the slaves.

Task	Color	Approximate time	Processing done
data → slaves	dark blue	350 msec	transfer 384 kbytes
data → master	yellow	1 msec	transfer 3 kbytes
correlate	purple	500 msec	2 ffts of 64k floats, and search
splitup (likelihood)	light blue	330 msec	several runs through 64k floats
real FFT (one phase)	green	150 msec	1 fft of 64k floats
compute template	gray	350 msec	compute 2 arrays of ≈ 18k floats
orthonormalize templates	wheat	25 msec	several runs through 64k floats

Table 9: Approximate computation times for different elements of the optimal-filtering process.

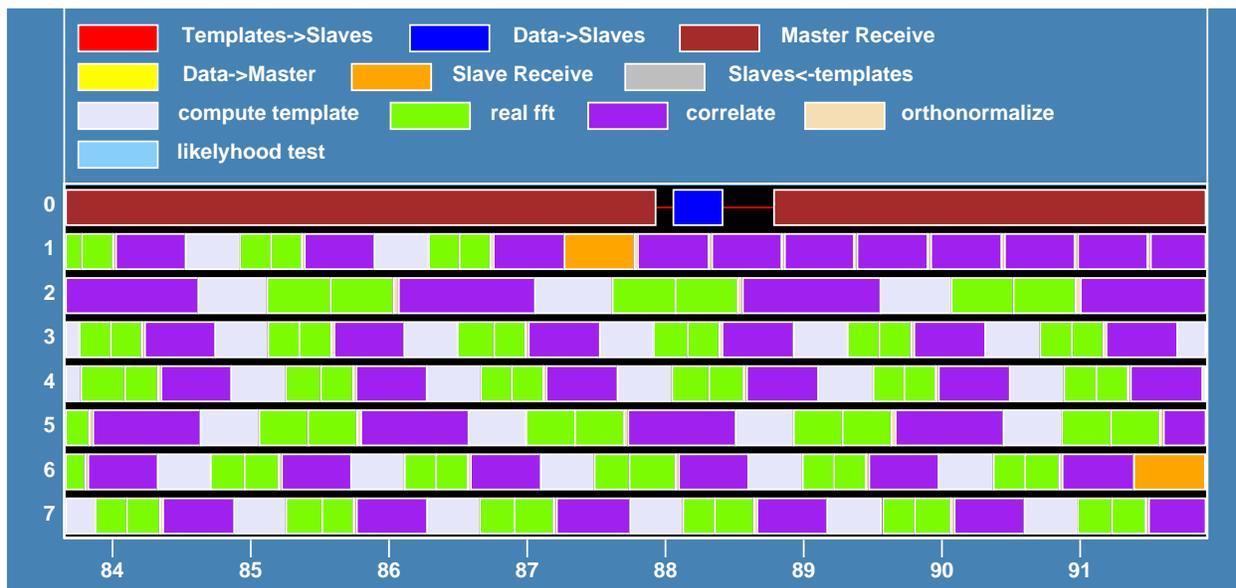


Figure 65: This is a continuation of the previous figure. Slave number 1 has completed its computation of the templates, and during the orange segment, waits to make a connection with the master. This is followed by a (very small) yellow segment, during which the slave transmits data back to the master, and a blue segment during which the master transmits new data to slave number 1. Immediately after this, slave number 1 begins a new (purple) sequence of correlation calculations on the newly received block of data. Notice that because slave 1 has already computed the templates, the light gray and green operations are no longer needed.

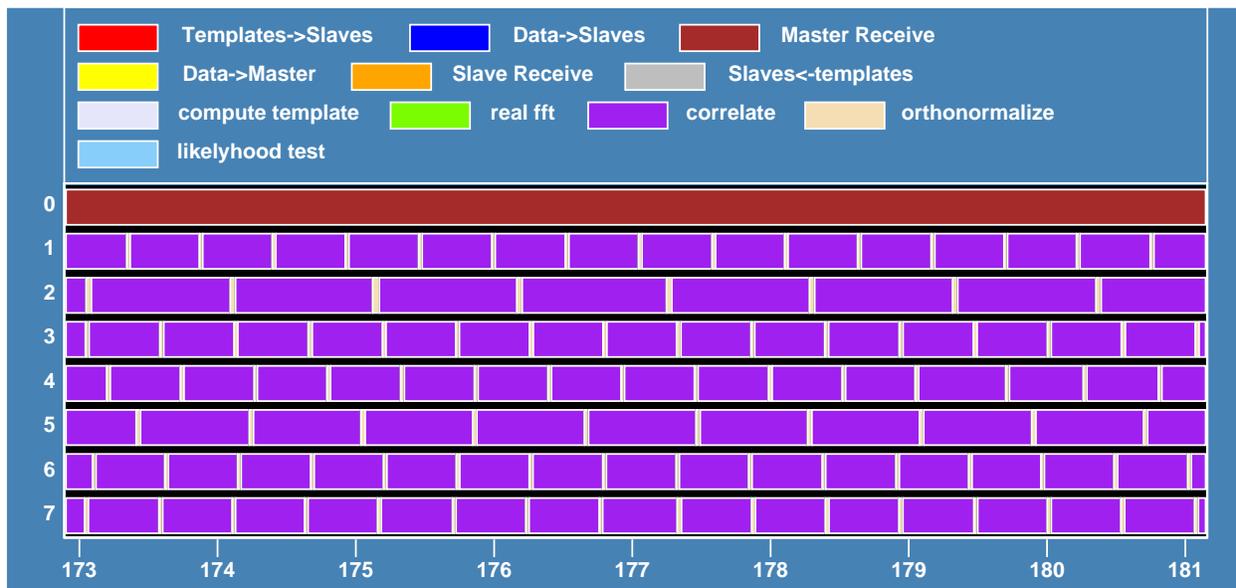


Figure 66: This is a continuation of the previous figure, and represents the “long-term” or “steady-state” behavior of the multiprocessing system. In this state, the different processors are spending all of their time doing correlation measurements of the data, as indicated by the purple segments, and the master is waiting for the results of the analysis (brown segments).

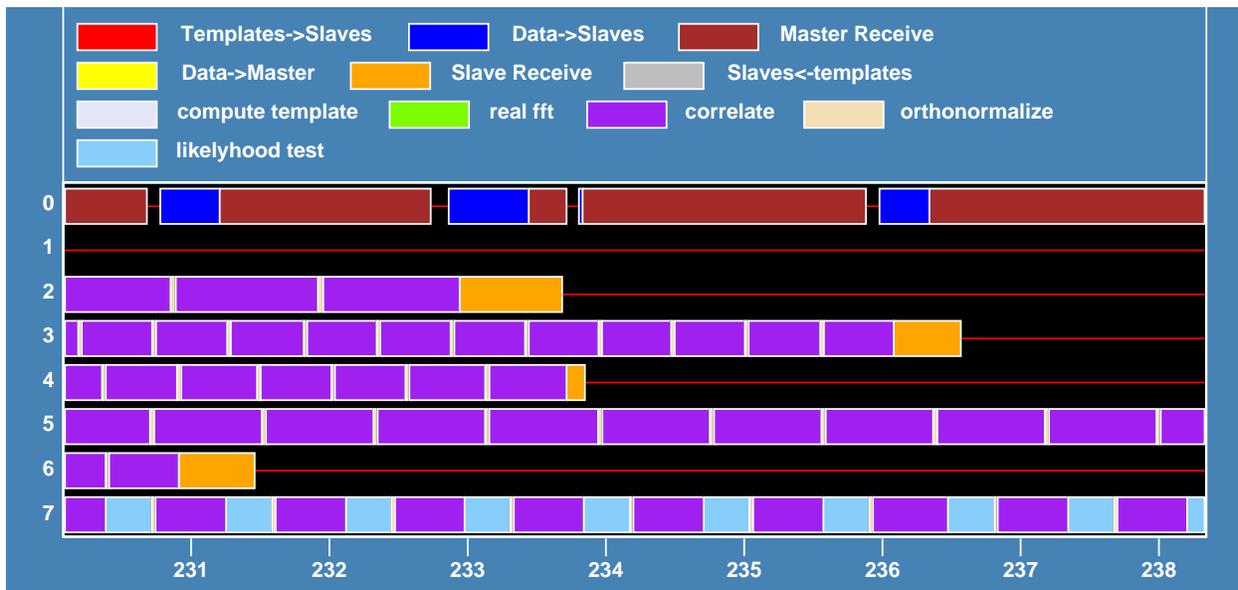


Figure 67: This is a continuation of the previous figure, and shows the termination of some of the slave processes (all the data has been analyzed, and there is no new data remaining). The blue segments (data being sent to slaves) are actually termination messages being sent to the different processes 2,3,4 and 6. Processes 5 and 7 are still computing. In the case of process 7, the data being analyzed contains a non-stationary “spurion” which triggered most of the filters beyond a pre-set threshold level. As a result, process 7 is performing some additional computations (the split-up likelihood test, shown as light blue segments) on the data.

Author: Bruce Allen, ballen@dirac.phys.uwm.edu

Comments: There are many other ways in which this optimal filtering code could be parallelized. This program illustrates one of the possibilities. Other possibilities include: maintaining different templates on different processes, and broadcasting identical IFO data to these different processes, or parallelizing across both data and templates.

```
/* GRASP: Copyright 1997,1998 Bruce Allen */
/* multifilter.c
This code is intended for machines where computation is cheap,
and communication is expensive. The processing is organized as
master/slaves (or manager/workers!). The master process sends out data
chunks to individual slave processes. These slave processes analyze
the data against all templates, then return the largest signal values
obtained for each template, along with other parameters like the time of
coalescence and the phase of coalescence. They then get a new data chunk.
If STORE_TEMPLATES is set to 1, then the filters are computed once,
then stored internally by each slave. This is the correct choice if each
slave has lots of fast memory available to it. If STORE_TEMPLATES is set
to 0, then the slaves recompute the templates each time they use them.
This is the correct choice if each slave has only small amounts of fast
memory available.
*/

#include "mpi.h"
#include "mpe.h"
#include "grasp.h"

#define NPOINT 65536 /* The size of our segments of data (26.2 secs) */
#define FLO 120.0 /* The low frequency cutoff for filtering */
#define HSCALE 1.e21 /* A convenient scaling factor; results independent of it */
#define MIN_INT0_LOCK 3.0 /* Number of minutes to skip into each locked section */
#define SAFETY 200 /* Padding safety factor to avoid wraparound errors */
#define CHIRPLEN 18000 /* length of longest allowed chirp */
#define MMIN 1.2 /* min mass object, solar masses */
#define MMAX 1.6 /* max mass object, solar masses */
#define DATA_SEGMENTS 25 /* largest number of data segments to process */
#define NSIGNALS 11 /* number of signal values computed for each template */
#define STORE_TEMPLATES 0 /* 0: slaves recompute templates. 1: slaves save templates. */

void shiftdata();
void realft(float*, unsigned long, int);
int get_calibrated_data();

struct Saved {
    float tstart;
    int gauss;
};

short *datas;
int npoint, remain=0, needed, diff, gauss_test, num_sent=0, fill_buffer();
float *twice_inv_noise, *htilde, *data, *mean_pow_spec, tstart;
float srate=9868.4208984375, decaytime, datastart, *response;
double norm, decay;
FILE *fpifo, *fpss, *fplock;

int main(int argc, char *argv[])
{
    int *lchirppoints, num_stored;
    float *ltc, *lch0tilde, *lch90tilde;
    int myid, numprocs, i, j, maxi, impulseoff, *chirppoints, indices[8], num_templates;
    int slave, more_data, temp_no, num_recv=0, namelen, completed=0, longest_template=0;
    float *tc, m1, m2, *template_list, *sig_buffer, distance, snr_max, var, timeoff, timestart;
    float n0, n90, inverse_distance_scale, *output90, *output0, *ch0tilde, *ch90tilde;
    float lin0, lin90, varsplit, stats[8], gammq(float, float);
```

```
double prob;
FILE *fpout;
MPI_Status status;
char processor_name[MPI_MAX_PROCESSOR_NAME], logfile_name[64], name[64];
struct Scope Grid;
struct Saved *saveme;

/* start MPI, find number of processes, find process number */
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &myid);
MPI_Get_processor_name(processor_name, &namelen);
MPI_Init_log();

/* number of points to sample and fft (power of 2) */
needed=npoint=NPOINT;

/* Gravity wave signal (frequency domain) & twice inverse noise power */
htilde=(float *)malloc(sizeof(float)*npoint+sizeof(float)*(npoint/2+1));
twice_inv_noise=htilde+npoint;

/* Structure for saving information about data sent to slaves */
saveme=(struct Saved *)malloc(sizeof(struct Saved)*numprocs);

/* MASTER */
if (myid==0) {
    MPE_Describe_state(1,2,"Templates->Slaves", "red:vlines3");
    MPE_Describe_state(3,4,"Data->Slaves", "blue:gray3");
    MPE_Describe_state(5,6,"Master Receive", "brown:light_gray");
    MPE_Describe_state(7,8,"Data->Master", "yellow:dark_gray");
    MPE_Describe_state(9,10,"Slave Receive", "orange:white");
    MPE_Describe_state(13,14,"Slaves<-templates", "gray:black");
    MPE_Describe_state(15,16,"compute template", "lavender:black");
    MPE_Describe_state(17,18,"real fft", "lawn green:black");
    MPE_Describe_state(19,20,"correlate", "purple:black");
    MPE_Describe_state(21,22,"orthonormalize", "wheat:black");
    MPE_Describe_state(23,24,"likelyhood test", "light sky blue:black");

    /* Set parameters for the inspiral search */
    Grid.m_mn=MMIN;
    Grid.m_mx=MMAX;
    Grid.theta=0.964;
    Grid.dp=2*0.00213;
    Grid.dq=2*0.0320;
    Grid.f_start=140.0;

    /* construct template set covering parameter space, m1 m2 storage */
    template_grid(&Grid);
    num_templates=Grid.n_tmplt;
    printf("The number of templates being used is %d\n", num_templates);
    template_list=(float *)malloc(sizeof(float)*2*num_templates);

    /* put mass values into an array */
    for (i=0; i<Grid.n_tmplt; i++) {
        template_list[2*i]=Grid.templates[i].m1;
        template_list[2*i+1]=Grid.templates[i].m2;
        printf("Mass values are m1 = %f m2 = %f\n", Grid.templates[i].m1, Grid.templates[i].m2);
    }
    fflush(stdout);
}
```

```
/* storage for returned signals (NSIGNALS per template) */
sig_buffer=(float *)malloc(sizeof(float)*num_templates*NSIGNALS);

/* broadcast templates */
MPE_Log_event(1,myid,"send");
MPI_Bcast(&num_templates,1,MPI_INT,0,MPI_COMM_WORLD);
MPI_Bcast(template_list,2*num_templates,MPI_FLOAT,0,MPI_COMM_WORLD);
MPE_Log_event(2,myid,"sent");

/* number of points to sample and fft (power of 2) */
needed=npoint=NPOINT;

/* stores ADC data as short integers */
datas=(short*)malloc(sizeof(short)*npoint);

/* stores ADC data in time & freq domain, as floats */
data=(float *)malloc(sizeof(float)*npoint);

/* The response function (transfer function) of the interferometer */
response=(float *)malloc(sizeof(float)*(npoint+2));

/* The autoregressive-mean averaged noise power spectrum */
mean_pow_spec=(float *)malloc(sizeof(float)*(npoint/2+1));

/* Set up noise power spectrum and decay time */
norm=0.0;
clear(mean_pow_spec,npoint/2+1,1);
decaytime=10.0*npoint/srate;
decay=exp(-1.0*npoint/(srate*decaytime));

/* open the IFO output file, lock file, and swept-sine file */
fpifo=grasp_open("GRASP_DATAPATH","channel.0","r");
fplock=grasp_open("GRASP_DATAPATH","channel.10","r");
fpss=grasp_open("GRASP_DATAPATH","swept-sine.ascii","r");

/* get the response function, and put in scaling factor */
normalize_gw(fpss,npoint,srate,response);
for (i=0;i<npoint+2;i++)
    response[i]*=HSCALE/ARMLENGTH_1994;

/* while not finished, loop over slaves */
for (slave=1;slave<numprocs;slave++) {
    if (get_calibrated_data()) {
        /* if new data exists, then send it (nonblocking?) */
        fprintf(stderr,"Master broadcasting data segment %d\n",num_sent+1);
        MPE_Log_event(3,myid,"send");
        MPI_Send(htilde,NPOINT+NPOINT/2+1,MPI_FLOAT,slave,++num_sent,MPI_COMM_WORLD);
        MPE_Log_event(4,myid,"sent");
        saveme[slave-1].gauss=gauss_test;
        saveme[slave-1].tstart=datastart;
        shiftdata();
    }
    else {
        /* tell remaining processes to exit */
        MPE_Log_event(3,myid,"send");
        MPI_Send(htilde,NPOINT+NPOINT/2+1,MPI_FLOAT,slave,0,MPI_COMM_WORLD);
        MPE_Log_event(4,myid,"sent");
    }
}
```

```
}

/* now loop, gathering answers, sending out more data */
while (num_sent!=num_rcv) {
    more_data=get_calibrated_data();

    /* listen for answer */
    MPE_Log_event(5,myid,"receiving...");
    MPI_Recv(sig_buffer,NSIGNALS*num_templates,MPI_FLOAT,MPI_ANY_SOURCE,
             MPI_ANY_TAG,MPI_COMM_WORLD,&status);
    MPE_Log_event(6,myid,"received");
    num_rcv++;

    /* store the answers... */
    sprintf(name,"signals.%05d",status.MPI_TAG-1);
    fpout=fopen(name,"w");
    if (fpout==NULL) {
        fprintf(stderr,"Multifilter: can't open output file %s\n",name);
        MPI_Finalize();
        return 1;
    }
    fprintf(fpout,"# Gaussian %d\n",saveme[status.MPI_SOURCE-1].gauss);
    fprintf(fpout,"# tstart %f\n",saveme[status.MPI_SOURCE-1].tstart);
    fprintf(fpout,"# snr    distance    phase0    phase90    maxi\
impulseoff impulsetime startinspiral coalesce    variance    prob\n");
    for (i=0;i<num_templates;i++) {
        for (j=0;j<NSIGNALS-1;j++)
            fprintf(fpout,"%g\t",sig_buffer[i*NSIGNALS+j]);
        fprintf(fpout,"%f\n",sig_buffer[i*NSIGNALS+j]);

        /* if data stream has no obvious outliers, and chirp prob is high, print */
        if (sig_buffer[i*NSIGNALS+10]>0.03 && saveme[status.MPI_SOURCE-1].gauss) {
            printf("POSSIBLE CHIRP: signal file %d, template %d, SNR = %f, prob = %f\n",
                  status.MPI_TAG-1,i,sig_buffer[i*NSIGNALS],sig_buffer[i*NSIGNALS+10]);
            fflush(stdout);
        }
    }
}
fclose(fpout);

/* if there is more data, send it off */
if (more_data) {
    fprintf(stderr,"Master broadcasting data segment %d\n",num_sent+1);
    MPE_Log_event(3,myid,"send");
    MPI_Send(htilde,NPOINT+NPOINT/2+1,MPI_FLOAT,status.MPI_SOURCE,++num_sent,MPI_COMM_WORLD);
    MPE_Log_event(4,myid,"sent");
    saveme[status.MPI_SOURCE-1].gauss=gauss_test;
    saveme[status.MPI_SOURCE-1].tstart=datastart;
    shiftdata();
}
/* or else tell the process that it can pack up and go home */
else {
    printf("Shutting down slave process %d\n",status.MPI_SOURCE);
    MPE_Log_event(3,myid,"send");
    MPI_Send(htilde,NPOINT+NPOINT/2+1,MPI_FLOAT,status.MPI_SOURCE,0,MPI_COMM_WORLD);
    MPE_Log_event(4,myid,"sent");
}
}
```

```
    /* when all the answers are in, print results */
    printf("This is the master - all answers are in!\n");
}

/* SLAVES */
else {
    printf("Slave %d (%s) just got started...\n",myid,processor_name);
    fflush(stdout);

    /* allocate storage space */
    /* Ouput of matched filters for phase0 and phase pi/2, in time domain, and temp storage */
    output0=(float *)malloc(sizeof(float)*npoint);
    output90=(float *)malloc(sizeof(float)*npoint);

    /* get the list of templates to use */
    MPE_Log_event(13,myid,"receiving...");
    MPI_Bcast(&num_templates,1,MPI_INT,0,MPI_COMM_WORLD);
    sig_buffer=(float *)malloc(sizeof(float)*num_templates*NSIGNALS);
    template_list=(float *)malloc(sizeof(float)*2*num_templates);
    MPI_Bcast(template_list,2*num_templates,MPI_FLOAT,0,MPI_COMM_WORLD);
    MPE_Log_event(14,myid,"received");
    printf("Slave %d (%s) just got template list...\n",myid,processor_name);
    fflush(stdout);

    /* Orthogonalized phase 0 and phase pi/2 chirps, in frequency domain */
    num_stored=STORE_TEMPLATES*(num_templates-1)+1;
    lch0tilde=(float *)malloc(sizeof(float)*npoint*num_stored);
    lch90tilde=(float *)malloc(sizeof(float)*npoint*num_stored);
    lchirppoints=(int *)malloc(sizeof(float)*num_stored);
    ltc=(float *)malloc(sizeof(float)*num_stored);

    if (lch0tilde==NULL || lch90tilde==NULL || lchirppoints==NULL || ltc==NULL) {
        fprintf(stderr,"Node %d on machine %s: could not malloc() memory!\n",
            myid,processor_name);
        MPI_Abort(MPI_COMM_WORLD,1);
    }

    /* now enter an infinite loop, waiting for new inputs */
    while (1) {
        /* listen for data, parameters from master */
        MPE_Log_event(9,myid,"receiving...");
        MPI_Recv(htilde,NPOINT+NPOINT/2+1,MPI_FLOAT,0,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
        MPE_Log_event(10,myid,"received");
        printf("Slave %d (%s) got htilde (and noise spectrum) for segment %d \n",
            myid,processor_name,status.MPI_TAG);
        fflush(stdout);

        /* if this is a termination message, we are done! */
        if (status.MPI_TAG==0) break;

        /* compute signals */
        for (temp_no=0;temp_no<num_templates;temp_no++) {

            ch0tilde=lch0tilde+npoint*temp_no*STORE_TEMPLATES;
            ch90tilde=lch90tilde+npoint*temp_no*STORE_TEMPLATES;
            chirppoints=lchirppoints+temp_no*STORE_TEMPLATES;
            tc=ltc+temp_no*STORE_TEMPLATES;

            /* Compute the template, and store it internally, if desired */

```

```
if (completed!=num_templates) {
    /* manufacture two chirps (dimensionless strain at 1 Mpc distance) */
    m1=template_list[2*temp_no];
    m2=template_list[2*temp_no+1];

    MPE_Log_event(15,myid,"computing");
    make_filters(m1,m2,ch0tilde,ch90tilde,FLO,npoint,srate,chirppoints,tc,4000,4);
    MPE_Log_event(16,myid,"computed");

    if (*chirppoints>longest_template) longest_template=*chirppoints;

    if (*chirppoints>CHIRPLEN) {
        fprintf(stderr,"Chirp m1=%f m2=%f length %d too long!\n",m1,m2,
            *chirppoints);
        fprintf(stderr,"Maximum allowed length is %d\n",CHIRPLEN);
        fprintf(stderr,"Please recompile with larger CHIRPLEN value\n");
        fflush(stderr);
        MPI_Abort(MPI_COMM_WORLD,1);
    }

    /* normalize the chirp template */
    /* normalization of next line comes from GRASP (5.6.3) and (5.6.4) */
    inverse_distance_scale=2.0*HSCALE*(TSOLAR*C_LIGHT/MPC);
    for (i=0;i<*chirppoints;i++){
        ch0tilde[i]*=inverse_distance_scale;
        ch90tilde[i]*=inverse_distance_scale;
    }

    /* and FFT the chirps */
    MPE_Log_event(17,myid,"starting fft");
    realft(ch0tilde-1,npoint,1);
    MPE_Log_event(18,myid,"ending fft");
    MPE_Log_event(17,myid,"starting fft");
    realft(ch90tilde-1,npoint,1);
    MPE_Log_event(18,myid,"ending fft");

    if (STORE_TEMPLATES) completed++;

    /* print out the length of the longest template */
    if (completed==num_templates)
        printf("Slave %d: templates completed. Longest is %d points\n",
            myid,longest_template);
    fflush(stdout);
} /* done computing the template */

/* orthogonalize the chirps: we never modify ch0tilde, only ch90tilde */
MPE_Log_event(21,myid,"starting");
orthonormalize(ch0tilde,ch90tilde,twice_inv_noise,npoint,&n0,&n90);
MPE_Log_event(22,myid,"done");

/* distance scale Mpc for SNR=1 */
distance=sqrt(1.0/(n0*n0)+1.0/(n90*n90));

/* find the moment at which SNR is a maximum */
MPE_Log_event(19,myid,"searching");
find_chirp(htilde,ch0tilde,ch90tilde,twice_inv_noise,n0,n90,output0,output90,
    npoint,CHIRPLEN,&maxi,&snr_max,&lin0,&lin90,&var);
MPE_Log_event(20,myid,"done");
```

```
/* identify when an impulse would have caused observed filter output */
impulsoff=(maxi+*chirppoints)%npoint;
timeoff=impulsoff/srate;
timestart=maxi/srate;

/* collect interesting signals to return */
sig_buffer[temp_no*NSIGNALS]=snr_max;
sig_buffer[temp_no*NSIGNALS+1]=distance;
sig_buffer[temp_no*NSIGNALS+2]=lin0;
sig_buffer[temp_no*NSIGNALS+3]=lin90;
sig_buffer[temp_no*NSIGNALS+4]=maxi;
sig_buffer[temp_no*NSIGNALS+5]=impulsoff;
sig_buffer[temp_no*NSIGNALS+6]=timeoff;
sig_buffer[temp_no*NSIGNALS+7]=timestart;
sig_buffer[temp_no*NSIGNALS+8]=timestart+*tc;
sig_buffer[temp_no*NSIGNALS+9]=var;

prob=0.0;
if (snr_max>5.0) {
    MPE_Log_event(23,myid,"testing");
    varsplit=splitup_freq2(lin0*n0/sqrt(2.0),lin90*n90/sqrt(2.0),ch0tilde,
                          ch90tilde,2.0/(n0*n0),twice_inv_noise,npoint,maxi,8,
                          indices,stats,output0,htilde);
    prob=gammap(4.0,4.0*varsplit);
    MPE_Log_event(24,myid,"done");
}
sig_buffer[temp_no*NSIGNALS+10]=prob;
} /* end of loop over the templates */

/* return signals to master */
MPE_Log_event(7,myid,"send");
MPI_Send(sig_buffer,NSIGNALS*num_templates,MPI_FLOAT,0,status.MPI_TAG,MPI_COMM_WORLD);
MPE_Log_event(8,myid,"sent");

} /* end of loop over the data */
}

/* both slaves and master exit here */
printf("%s preparing to shut down (process %d)\n",processor_name,myid);
sprintf(logfile_name,"multifilter.%d.%d.log",numprocs,DATA_SEGMENTS);
MPE_Finish_log(logfile_name);
MPI_Finalize();
printf("%s shutting down (process %d)\n",processor_name,myid);
return 0;
}

/* This routine gets the data set, overlapping the data buffer as needed */
int get_calibrated_data() {
    int i,code;

    if (num_sent>=DATA_SEGMENTS)
        return 0;

    while (remain<needed) {
        code=get_data(fpifo,fplock,&tstart,MIN_INTO_LOCK*60*srate,
                    datas,&remain,&srate,1);
        if (code==0) return 0;
    }
}
```

```
/* Get the next needed samples of data */
diff=npoint-needed;
code=get_data(fpifo,fplock,&tstart,needed,datas+diff,&remain,&srates,0);
datastart=tstart-diff/srates;

/* copy integer data into floats */
for (i=0;i<npoint;i++) data[i]=datas[i];

/* find the FFT of data*/
realft(data-1,npoint,1);

/* normalized delta-L/L tilde */
product(htilde,data,response,npoint/2);

/* update the inverse of the auto-regressive-mean power-spectrum */
avg_inv_spec(FLO,srates,npoint,decay,&norm,htilde,mean_pow_spec,twice_inv_noise);

/* see if the data has any obvious outliers */
gauss_test=is_gaussian(datas,npoint,-2048,2047,0);

return 1;
}

/* this function shifts data by CHIRPLEN points in buffer */
void shiftdata() {
    int i;

    /* shift ends of buffer to the start */
    needed=npoint-CHIRPLEN+1;
    for (i=0;i<CHIRPLEN-1;i++) datas[i]=datas[i+needed];

    /* reset if not enough points remain to fill the buffer */
    if (remain<needed) needed=npoint;

    return;
}
```

9.25 Optimization and computation-speed considerations

The previous subsection describes the `multifilter` program, which filters data through a bank of templates. We have experimented with the optimization of this code on several platforms, and here recount some of that experience.

The first comment is that the *Numerical Recipes* routine `realft()` is not as efficient as possible. In order to produce a production version of the GRASP code, we suggest replacing this function with a more-optimal version. For example, on the Intel Paragon, the CLASSPACK library provides optimized real-FFT functions. To replace the `realft()` routine, we provide a replacement routine by the same name, which calls the CLASSPACK library. This routine may be found in the `src/optimization/paragon` directory of GRASP. By including the object file for this routine in the linking path, before the *Numerical Recipes* library, it replaces the `realft()` routine. (Note: GRASP currently contains optimized replacement routines for the FFT on SGI/Cray, Sun, Paragon, DEC and Intel Linux machines; see the `src/optimization/*` directories of GRASP, described in Section 2.6.9).

The second comment is related to inspiral-chirp template generation. The binary inspiral chirps may be saved in the `multifilter` program, but one is then limited by the available memory space, as well as incurring the overhead of frequent disk accesses if that memory space is swapped onto and off the disk. To avoid this, it is attractive to generate templates “on the fly”, then dispose of them after each segment of data is analyzed. This corresponds to setting `STORE_TEMPLATES` to 0 in `multifilter`. In this instance, the computational cost of computing binary chirp templates may become quite high, relative to the cost of the remaining computation (FFT’s, orthogonalization, searching for the maximum SNR).

To cite a specific example, on the Intel Paragon, we found that the template generation was almost a factor of ten more time-consuming than the rest of the searching procedure. Some profiling revealed that the two culprits were the cube-root operation and the calculations of sines and cosines. Because the floating point hardware on the Paragon only does add, subtract and multiply, these operations required expensive library calls. In both cases, a small amount of work serves to eliminate most of this computation time. In the case of the cube root function, we have provided (through an `ifdef` `INLINE_CUBEROOT` in the code) an inline computation of cuberoot in 15 FLOPS, which only uses add, subtract and multiply. This routine shifts x into the range from $1 \rightarrow 2$, then uses a fifth-order Chebyshev approximation of $x^{-2/3}$ then make one pass of Newton-Raphson to clean up to float precision, and returns $x^{1/3} = x^{-2/3}x$. In the case of the trig functions we have provided (through an `ifdef` `INLINE_TRIGS` in the code) inline routines to calculate the sine and cosine as well. After reducing the range of the argument to $x \in [-\pi, \pi]$, these use a 6th order Chebyshev polynomial to approximate the sine and cosine. These techniques speed up the template generation to the point where it is approximately as expensive as the remaining computations. While there is some small loss of computational accuracy, we have not found it to be significant. Shown in Figure 68 is a timing diagram illustrating the relative computational costs of these operations.

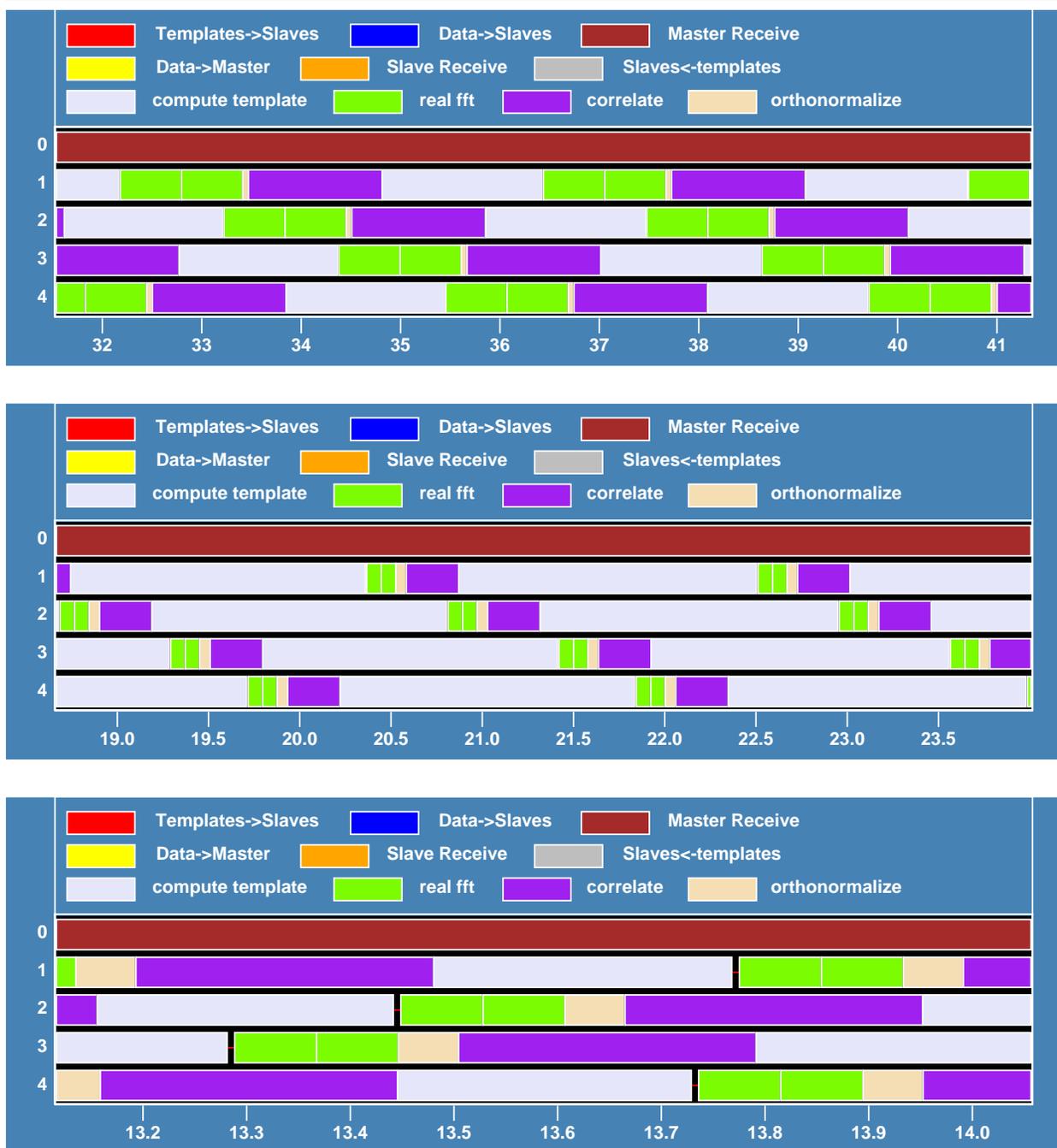


Figure 68: This shows the performance of an “on the fly” template search on the Intel Paragon, with different levels of optimization. The top diagram uses the *Numerical Recipes* FFT routine `realft()`, and takes about 4.2 seconds to process 6 seconds of data. The middle diagram shows identical code using the *CLASSPACK* optimized FFT routine, and takes about 2.1 seconds. Note that the template generation process is now becoming expensive. The bottom diagram shows identical code which includes inline functions for cube-root and sine/cosine functions to speed up the template generation process. The template generation takes about 325 msec, and the entire search procedure (including template generation) takes 780 msec per template per processor per 6-second stretch of data. Relative to the top diagram, this represents a speed-up factor of more than 5. Running on 256 nodes, it is possible to filter 5 hours of data through 66 templates (representing the mass range from 1.2 to 1.6 solar masses) in $5 \times 3600 \times 66 \times (0.780) / (256 \times 6)$ seconds = 10.1 minutes.

9.26 Template Placement

As mentioned in preceding sections, when detecting signals using a discrete bank of templates, some loss in signal strength will always occur due to imperfect matching of the signal with the closest template in the bank. The *match function* μ measures this signal loss; the *mismatch* $1 - \mu$ can then be thought of as a proper distance interval on the template parameter space. The goal of template bank construction is to place templates with a sufficient density in parameter space that the fractional signal loss is reduced to less than some specified amount. However, since each template in the bank represents a computational investment, one would like to do this with as few templates as possible.

Conceptually this can be thought of as a tiling problem — one attempts to cover the parameter space completely with “tiles”, each representing a template. Each tile is small enough to fit entirely within the equimatch contour at the specified match level, drawn about the tile’s centre. For match levels close to 1, the match function μ drops off quadratically with parameter offsets, and the equimatch contours are ellipses. However, the sizes and orientations of these ellipses will in general vary over the parameter space, as the behaviour of the match function changes. This complicates the problem significantly. For instance, while hexagonal tiling can be shown to be the most efficient tiling scheme when tile sizes are constant, it is not at all clear how to implement such a scheme when the sizes and shapes of the hexagons are allowed to vary.

For this reason, a somewhat less efficient but algorithmically simpler tiling scheme has been adopted. We lay out on the parameter space a rectilinear coordinate system with some arbitrary rotation, take our tiles to be rectangles whose sides are aligned with the coordinate axes. The width and height of each tile may vary so that it exactly inscribes the equimatch ellipse, but its orientation is fixed by the coordinate system. As figure 69(a) shows, aligning the coordinate axes with the principle axes of the ellipse results in the largest tile areas, and hence the fewest number of templates. When the ellipse have changing orientations, one must choose some appropriate averaged angle for the coordinate system. Several guesses are often required before an optimal orientation is found.

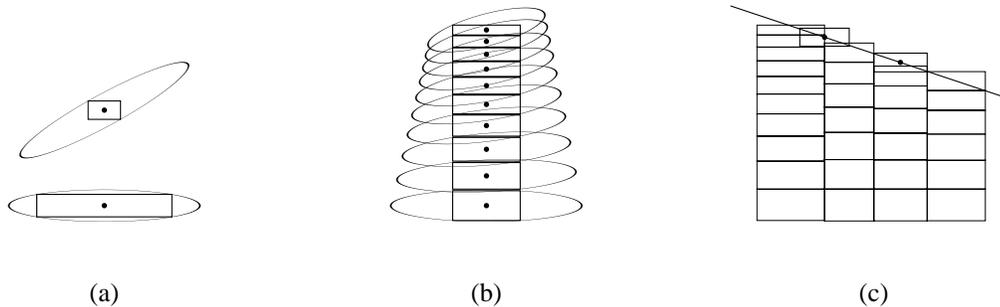


Figure 69: Some aspects of template placement using rectangular tiling: (a) By applying a coordinate rotation, the axes of the equimatch ellipse can be alligned with the axes of the rectangular tiles, maximizing the tile area. (b) Rectangular tiles can easily be stacked into columns, even when the match contours are varying. (c) At the ends of a column, extra overlapping tiles may be required to cover the edges of the parameter space.

Once a coordinate system is chosen, the parameter space can be divided into columns, into which the rectangular tiles are stacked. The height of each tile is chosen so as to stay within its equimatch ellipse, as shown in figure 69(b), and the column widths may vary between columns so as to maximize the resulting

tile areas. At the boundaries of the parameter space, extra tiles may have to be added at the corners of a column to provide complete coverage, as shown in figure 69(c).

The function `tiling_2d()` (section 9.28) is a generic implementation of such a rectangular tiling scheme. It works on nearly any parameter space on which a proper distance metric can be defined. The function `get_chirp_templates()` (section 9.37) implements the `tiling_2d()` algorithm for the specific case of binary inspiral templates, using the quadratic terms of a fit to the chirp template match function (equation 9.12.1) to define the distance metric on the space.

9.27 Structure: struct tile

For routines which lay out a mesh of tiles, a convenient data structure for keeping track of these templates is a linked list. Such a structure allows new tiles to be added or inserted into the list, without knowing in advance how many tiles are going to be generated. The following structure stores data for a rectangular tile inscribed within an ellipse; this is suitable for many tiling problems in which the goal is to fill a two-dimensional space with some minimum density of tiles.

```
struct tile {  
  
    int flag; Error codes generated while processing this tile.  
  
    double x; The horizontal position of the patch.  
  
    double y; The vertical position of the patch.  
  
    double dx; The horizontal width of the patch.  
  
    double dy; The vertical height of the patch.  
  
    double r1; The semimajor axis of the circumscribing ellipse.  
  
    double r2; The semiminor axis of the circumscribing ellipse.  
  
    double theta; The angle between the x and semimajor axes.  
  
    struct tile *next; A pointer to the next tile in the list.  
  
};
```

9.28 Function: `tiling_2d`

```
int tiling\_2d(double *x_bound, double *y_bound, int npts,  
             int (*metric)(double , double , double *),  
             struct tile **tail, int *n_tiles);
```

This is a generic routine for laying out a mesh of overlapping rectangular tiles in a (relatively) arbitrary two-dimensional parameter space. The tiles are sized such that no point on the tile is more than one unit of proper distance from its centre, where the proper distance is computed with a metric function which can vary arbitrarily over the parameter space. Thus the size and shape of the tiles can and will vary over the space. The tiles are rectangular, aligned with the coordinate axes, and are laid out in columns, with extra overlapping tiles on the edges to ensure complete coverage of the space. The lattice of tile positions is stored as a linked list.

Note that the routine can be easily modified to lay out tiles with non-unit proper radius. To scale the tiles by a factor d , simply multiply the metric function by a factor d^{-2} .

Upon successful execution, `tiling_2d()` attaches the new linked list to `(**tail).next`, updates `*tail` to point to the new tail of the list, and returns a value of 0. It returns an error code of 1 if it suspects that some of the columns may not be properly filled. It returns 2 if at any point the width of the parameter space was more than `N_COLS` times the computed column width. It returns 3 if the routine terminated prematurely for any other reason (usually because the algorithm accidentally stepped out of the parameter space, due to imprecise interpolation of the boundary). In the case of error codes 2 and 3, `tiling_2d()` still attaches the generated list onto `(**tail).next` (up to the point where the error occurred), but does not update the position of `*tail`. `tiling_2d()` will also write appropriate error messages indicating where any errors took place, and the `flag` field of the tile on the list (at the time of the error) is set to the error code. A `flag` value of -1 on any tile is a warning flag; the tile was placed correctly, but not all of the fields were calculated.

The arguments are:

`x_bound`: Input. An array `[0..npts]` storing the x -coordinates of `npts` points along the boundary of the parameter space. The array has length `npts+1`, but the index `[npts]` should refer to the same point as `[0]`.

`y_bound`: Input. As above, but the y -coordinates.

`npts`: Input. The number of points used to specify the boundary.

`metric()`: Input. This function computes the three independent components of the distance metric matrix at a given point in parameter space. The first two arguments are the x and y coordinates of the requested point, the third passes back the metric components in a three-element array. The `[0]`, `[1]`, and `[2]` metric components are defined in terms of the proper interval as follows:

$$ds^2 = [0]dx^2 + [1]dxdy + [2]dy^2.$$

`metric()` itself should return 0, or 1 if the metric is undefined or not computable at the specified location.

`tail`: Input/Output. Initially points to the “tail” of a pre-existing list; the generated list is attached to `(**tail).next`. Upon successful completion, `**tail` is updated to the new tail of the list.

`n_tiles`: Input/Output. A running tally of the number of tiles in the mesh; it is incremented each time a tile is added (and decremented whenever a tile is removed).

The `tiling_2d()` routine makes very few assumptions about the parameter space. The most stringent is the assumption that the parameter space boundary can be expressed as bivalued functions of both x and y ; that is, both vertical and horizontal lines intersect the boundary at no more than two points. If a vertical line intersects at more than two points, the routine may come to a point where it cannot determine the location or width of a column of tiles, and will terminate. If a horizontal line intersects at more than two points, the routine may not completely cover the edges of the parameter space. Appropriate warning or error messages are generated in these cases.

Author: Teviet Creighton, teviet@tapir.caltech.edu

9.29 Function: plot_list

```
int plot_list(double *x_bound, double *y_bound, int npts,  
             struct tile *head, int n_tiles, double angle,  
             double magnification, int plot_boundary,  
             int plot_tiles, int plot_ellipses, int plot_flags,  
             const char *psfile);
```

This routine generates a postscript file displaying a parameter space, the brickwork mesh of tiles covering it, and an overlapping mesh of elliptical contours of unit proper radius, circumscribing each tile. The function returns the number of pages of postscript output.

The arguments are:

x_bound: Input. The array `x_bound[0..npts]` contains the *x* components of a set of `npts` boundary points. Note that the array is of length `npts+1`; the `[0]` and `[npts]` index values refer to the same point, so the array explicitly describes a closed boundary; however, this is irrelevant to the current routine.

y_bound: Input. The array `y_bound[0..npts]` contains the *y* components of the boundary points, as above.

npts: Input. The number of points along the boundary.

head: Input. The head of the linked list of tiles to be plotted.

n_tiles: Input. The number of tiles to be plotted from the list.

angle: Input. The angle counterclockwise from the *x*-axis of the parameter space to the horizontal axis of the plot.

magnification: Input. The scale factor of points ($1/72$ of an inch) per unit coordinate distance in the parameter space.

plot_boundary: Input. 1 if boundary is to be shown, 0 otherwise.

plot_tiles: Input. 1 if the tile brickwork is to be shown, 0 otherwise.

plot_ellipses: Input. 1 if the overlapping ellipses are to be shown, 0 otherwise.

plot_flags: Input. If nonzero, indicates the size of dot used to mark flagged tiles (in points = $1/72$ inches). If zero, flags are ignored.

psfile: Input. The name of the postscript file created.

Author: Teviet Creighton, teviet@tapir.caltech.edu

9.30 Constants in `tiling_2d.c`

The following constants are `#defined` in the module `tiling_2d.c`, and are used by the routines `tiling_2d()` and `plot_list()`. There may be circumstances in which they might need to be changed.

`N_COLS = 1 000 000`: The maximum number of columns of tiles allowed in the parameter space.

`N_ROWS = 1 000 000`: The maximum number of rows of tiles allowed in any one column.

`X_MARGIN = 36 points = 0.5"`: The horizontal separation between the left and right borders of the plot and the edge of the page.

`Y_MARGIN = 36 points = 0.5"`: The vertical separation between the top and bottom borders of the plot and the edge of the page.

`X_SIZE = 612 points = 8.5"`: The width of the page.

`Y_SIZE = 792 points = 11"`: The height of the page.

`N_OBJ = 797`: The maximum number of objects which may be placed into a single PostScript macro. This limitation is required in order to prevent errors in certain PostScript interpreters (such as Ghostscript). This value may be set smaller: the only effect is to generate additional PostScript macros containing fewer objects each.

9.31 Structure: struct chirp_space

The following data structure, when fully assigned, contains complete information about a region of the parameter space of chirp signals: it describes the extent of the region, the coordinates used on it, the behaviour of the match function over it, and the number and location of chirp templates covering it. It is assumed that the templates will be placed using coordinates x, y which are related to the τ_0, τ_1 coordinates by a simple rotation; for the best results, the x, y axes should be roughly aligned with the principle axes of the elliptical equimatch contours discussed in section 9.9. The fields of this structure are:

```
struct chirp_space {  
  
    float m_mn; The minimum mass of a binary component in the parameter space (solar masses).  
  
    float m_mx; The maximum mass of a binary component in the parameter space (solar masses).  
  
    float ftau; The reference frequency used to define the  $\tau_0, \tau_1$  coordinates.  
  
    float angle; The angle (radians) counterclockwise from the  $\tau_0$  axis to the  $x$  coordinate axis used in  
        placing the template patches.  
  
    float match; The minimum match level of the covering template patches.  
  
    int n_bound; The number of points used to define the boundary of the region.  
  
    double *x_bound; An array [0..n_bound] containing the  $x$  coordinates of the points defining the  
        boundary. The array is of length n_bound+1, but the index values [0] and [n_bound] should  
        refer to the same point, so as to define an explicitly closed polygon.  
  
    double *y_bound; An array [0..n_bound] as above, but the  $y$  coordinates.  
  
    struct cubic_grid grid; A data structure containing coefficients of a cubic fit to the match func-  
        tion, evaluated on a grid of points in the parameter space, plus related information. See the documen-  
        tation for the struct cubic_grid data structure in section 9.12.  
  
    int n_templates; The number of template patches covering the space.  
  
    struct chirp_template *templates; An array [0..n_templates-1] of data structures  
        describing the positions of the covering templates. See the next section for a description of these  
        data structures.  
  
};
```

9.32 Structure: struct chirp_template

The following data structure is used to carry information about the position of a chirp template in a variety of coordinate systems, from the most specific computational parameters (its index in an enumerated list) to the most general physical parameters (the masses of its binary components). In many cases the transformations among these coordinates depend on additional parameters, such as the reference frequency for the τ_0, τ_1 coordinates, or the angle between the τ_0 and x axes; this global information is typically stored in a data structure of type `struct chirp_space` (section 9.31). In addition, the following structure contains some information about the size of the coordinate patch covered by the template. The fields are:

```
struct chirp_template {
```

`int flag`; An indicator of any errors which occurred while generating or placing this template. At present, the following codes are recognized:

- 1: Template is incomplete; some fields could not be filled.
- 0: No errors occurred.
- 1: Possible uncovered region of parameter space near the corners of this template.
- 2: Template placement terminated prematurely at this template, due to a metric singularity.
- 3: Template placement terminated prematurely at this template, for some other reason.

`int num`; The index of the template in an enumerated list, normally ranging from 0 to the number of templates - 1.

`double x`; The x coordinate of the template in a computational Cartesian coordinate system. Normally this system is related to the τ_0, τ_1 coordinate system by a simple rotation, so it has units of seconds.

`double y`; As above, but the y coordinate.

`double dx`; The width in the x direction of a rectangular patch about the template, which is inscribed within an equimatch ellipse.

`double dy`; The height in the y direction of the rectangular patch.

`double semimajor`; The length of the semimajor axis of the equimatch ellipse circumscribing the template patch.

`double semiminor`; The length of the semiminor axis of the equimatch ellipse circumscribing the template patch.

`double theta`; The angle counterclockwise from the x axis to the semimajor axis (radians).

`double tau0`; The 0th order post-Newtonian time to coalescence (seconds).

`double tau1`; The 1st order post-Newtonian correction to the time to coalescence (seconds).

`double mtotal`; The total mass of the binary system (solar masses).

`double mchirp`; The chirp mass of the system (solar masses).

`double mred`; The reduced mass of the system (solar masses).

`double eta`; The ratio of the reduced mass to the total mass.

double m1; The mass of one of the binary components (by convention, the larger), in solar masses.
double m2; The mass of the other binary component (by convention, the smaller), in solar masses.
};

9.33 Function: set_chirp_space

```
void set_chirp_space(struct chirp_space space);
```

This routine sets a global parameter `global_space` in the `chirp_templates.c` module, which is required for the `chirp_metric()` routine to function — see section 9.34. This data must be passed to `chirp_metric()` as a global parameter, rather than an argument, in order for `chirp_metric()` to have a “generic” argument list as required by the routine `tiling_2d()` (section 9.28).

The argument is:

`space` : Input. The structure that `global_space` is set to equal.

9.34 Function: chirp_metric

```
int chirp_metric(double x, double y, double *m);
```

This routine computes the coefficients of a local distance metric on the space of chirp templates, at a specified location in that space. It returns 0 upon successful completion, or 1 if the metric could not be computed at that point.

The arguments are:

x: Input. The x coordinate of the specified point.

y: Input. The y coordinate of the specified point.

m: Output. The array `m[0..2]` contains the three independent components of the local distance metric; see below.

Note that this routine uses the global parameter `global_space`, defined at the top of the `chirp_templates.c` module. This parameter must be set by the routines `set_chirp_space()` or `get_chirp_templates()` before `chirp_metric()` can be called. The x, y coordinate system used is the one defined in the `global_space` structure: it is to a (counterclockwise) rotation of the τ_0, τ_1 coordinate system by an angle of `global_space.angle`. The metric is computed by interpolating the grid of precomputed quadratic coefficients of the match function, stored in `global_space.grid`.

The local distance function is defined so that the match decreases to the value `global_space.match` at a proper distance of 1. That is, the unit of proper distance is defined to be the maximum template patch radius. The metric coefficients `m[0..2]` are related to the proper interval ds^2 by:

$$ds^2 = m[0]dx^2 + m[1]dxdy + m[2]dy^2 . \quad (9.34.1)$$

Note that $ds^2 = 1$ corresponds to the match being equal to $\mu = \text{global_space.match}$ in equation 9.12.1. We ignore the cubic components of the match function when dealing with the local distance metric. So we have:

$$\text{global_space.match} = 1 + \text{coef}[0]dx^2 + \text{coef}[1]dxdy + \text{coef}[2]dy^2 \quad (9.34.2)$$

when $ds^2 = 1$. So the metric components `m[]` are related to the match function coefficients `coef[]` via:

$$m[i] = \frac{\text{coef}[i]}{\text{global_space.match}} , i = 0 \dots 2 . \quad (9.34.3)$$

This routine also checks to make sure that the resulting metric is positive definite, which should always be the case if the match function is locally paraboloidal (rather than a saddle).

Author: Teviet Creighton, teviet@tapir.caltech.edu

9.35 Function: get_chirp_boundary

```
void get_chirp_boundary(struct chirp_space *space);
```

This routine computes the boundary of the triangular parameter space of chirp signals, setting the fields `x_bound` and `y_bound` of `*space`. Memory for these arrays is allocated in this routine; to free it, call `free((*space).x_bound)` and `free((*space).y_bound)`.

The argument is:

`space`: Input/Output. The data structure describing the parameter space of chirps. This routine uses as input the fields `m_mn`, `m_mx`, `f_tau`, `angle`, and `n_bound`, and assigns the fields `x_bound` and `y_bound`.

Author: Teviet Creighton, teviet@tapir.caltech.edu

Comments: The boundary is actually made to lie just inside of the triangular region specified by $m_{mn} < m_2 < m_1 < m_{mx}$. Particular care is taken along the equal mass line to insure that when one connects the computed points, the resulting line segments lie entirely inside the true boundary: the points must be shifted inwards to account for the slight concavity along this curve. Such caution is required because we must occasionally convert points back into mass space, and bad things happen if the points lie even slightly past the equal mass line.

9.36 Function: get_chirp_grid

```
int get_chirp_grid(struct chirp_space *space, const char *gridfile);
```

This routine sets the field `(*space).grid`, which contains pre-computed coefficients of the cubic fit to the match function at various points over the parameter space. It returns 0 if no warnings were generated, 1 if parameters used to generate the coefficient grid differed in some nontrivial way from those of the parameter space, or 2 if the coefficient grid could not be read in; in the latter case, `(*space).grid` is unchanged. Otherwise, this routine will allocate memory for the coefficient grid; to free this memory, call `free_cubic((*space).grid)`.

The arguments are:

`space`: Input/Output. The parameter space over which the coefficient grid is being assigned. The fields `m_mn` and `m_mx` are used only to check that the grid covers the space. The fields `ftau`, `angle`, and `match` are used to check, rotate, and rescale the grid's coordinate system (repectively). The field `grid` is the one which is set by this routine.

`gridfile`: Input. The name of a file containing the pre-computed coefficients of the match function; see the routine `generate_cubic()` in section 9.13.

Author: Teviet Creighton, teviet@tapir.caltech.edu

9.37 Function: get_chirp_templates

```
int get_chirp_templates(struct chirp_space *space);
```

This routine computes the positions of a mesh of chirp templates on a parameter space, using the generic tiling routine `tiling_2d()`. See section 9.28 for documentation of this routine. The function `get_chirp_templates()` passes to `tiling_2d()` the boundary polygon defined by `(*space).x_bound`, `(*space).y_bound`, and `(*space).n_bound`, as well as the template space metric function `chirp_metric()`, then converts the returned list of templates into the array `(*space).templates`. The `chirp_metric()` routine requires the parameter space to be passed to it as a global static variable named `global_space`; this variable is set equal to `*space`. The routine `get_chirp_templates()` itself returns the error code generated by the call to `tiling_2d()`; see the documentation of that routine. If a fatal error occurs before `tiling_2d()` is even called, `get_chirp_templates()` returns an error code of 3, `(*space).n_templates` is set to 0 and `(*space).templates` to NULL. Otherwise, memory for the template array will be allocated; to free this memory, call `free((*space).templates)`.

The argument is:

`space`: Input/Output. The data structure of the parameter space being filled with templates. This routine uses as input the fields `n_bound`, `x_bound`, `y_bound`, and `grid`, and assigns the fields `n_templates` and `templates`.

Author: Teviet Creighton, teviet@tapir.caltech.edu

9.38 Function: plot_chirp_templates

```
int plot_chirp_templates(struct chirp_space space,  
                        double magnification, int plot_boundary,  
                        int plot_patches, int plot_ellipses,  
                        int plot_flags, const char *psfile);
```

This routine plots a postscript file of a chirp parameter space and the templates covering it, using the generic template-plotting routine `plot_list()`. See section 9.29 for documentation on this routine. The input parameters for `plot_list()` are passed directly from the parameters for `plot_chirp_templates()`, except as follows: the boundary is taken from `space.x_bound`, `space.y_bound`, and `space.n_bound`, the linked list of templates is generated from `space.templates` and `space.n_templates`, and the rotation angle of the plot is such that τ_0 runs down the length of the page. The routine `plot_chirp_templates()` itself returns the number of pages of postscript output.

The arguments are:

`space`: Input. The data structure of the parameter space and its templates. This routine makes use of the fields `n_bound`, `x_bound`, `y_bound`, and `angle`; if template patches or flags are to be plotted, the fields `n_templates` and `templates` are also used.

`magnification`: Input. The scale factor of points ($1/72$ of an inch) per unit coordinate distance in the parameter space.

`plot_boundary`: Input. 1 if boundary is to be shown, 0 otherwise.

`plot_patches`: Input. 1 if rectangular brickwork of patches is to be shown, 0 otherwise.

`plot_ellipses`: Input. 1 if the overlapping match contours are to be shown, 0 otherwise.

`plot_flags`: Input. If nonzero, indicates the size of dot used to mark flagged templates (in points = $1/72$ inches). If zero, flags are ignored.

`psfile`: Input. The name of the postscript file created.

Author: Teviet Creighton, teviet@tapir.caltech.edu

9.39 Example: make_mesh program

This example program generates a template space and template bank for a binary inspiral search. It uses the functions `get_chirp_boundary()` to compute the perimeter of the template space, `get_chirp_grid()` to read in precomputed match coefficients over that space, `get_chirp_templates()` to place the template bank, and `plot_chirp_templates()` to plot a PostScript file of the template space.

Note that the `get_chirp_grid()` function requires the existence of a match coefficient data file named `cubic_coef_40meter_m=0.8-3.2.ascii`, as generated by the example program `make_grid` in section 9.19.

This program prints log messages to `stdout`, a list of the perimeter points (in x, y coordinates) to `boundary.ascii`, a list of the template positions (in mass coordinates) to `templates.ascii`, and the PostScript image to `templates.ps`. Here is the standard output produced by this program when everything functions normally:

```
make_mesh: get_chirp_templates generated 687 templates and exited with code 0.
make_mesh: plot_chirp_templates generated 8 pages of postscript output.
```

Here are the first few lines of the file `templates.ascii`, which lists the positions of the templates in mass space. The numbers in the two columns are the m_1, m_2 coordinates, in solar masses.

```
2.963451 2.980348
2.941810 2.960199
2.986875 2.999635
2.918038 2.932353
2.896092 2.911018
2.895651 2.999934
2.869409 2.885002
2.847899 2.862208
...
```

Figure 70 shows a portion of the PostScript file `templates.ps` generated by `plot_chirp_templates()`. The figure is magnified by an extra factor of 2 for clearer display.

```
/* GRASP: Copyright 1997,1998 Bruce Allen */
#include "grasp.h"

int main(int argc, char **argv)
{
    struct chirp_space space;
    double mag;
    int i,code,plot_boundary,plot_patches,plot_ellipses,plot_flags;
    FILE *fpout;

    /* Set the chirp space parameters. */
    space.m_mn=1.0;
    space.m_mx=3.0;
    space.angle=0.318;
    space.ftau=140.0;
    space.match=0.98;
    space.n_bound=600;

    /* Set the plotting parameters. */
    mag=4200.0;          /* Magnification factor. */
```

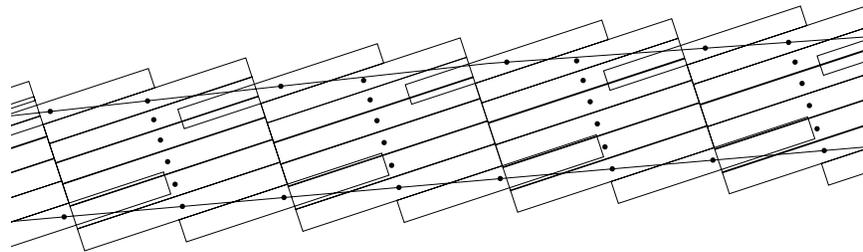


Figure 70: A portion of the chirp template parameter space generated by the program `make_mesh`, showing the template patches which cover the space to a match level of 0.98. This figure illustrates how the templates are stacked in columns, with additional overlapping patches along the boundary. Less obvious is the fact that the rectangles are not of uniform size, reflecting the changing behaviour of the match function over the space.

```
plot_boundary=1;    /* Do show the boundary. */
plot_patches=1;    /* Do show the patches. */
plot_ellipses=0;   /* Don't show the circumscribed ellipses. */
plot_flags=1;     /* Do indicate any flagged templates. */

/* Generate the parameter space boundary. */
get_chirp_boundary(&space);

/* Print out the boundary points. */
fpout=fopen("boundary.ascii", "w");
for(i=0; i<=space.n_bound; i++)
    fprintf(fpout, "%f %f\n", space.x_bound[i], space.y_bound[i]);
fclose(fpout);

/* Get the match function parameters over the space. */
if(get_chirp_grid(&space, "cubic_coef_40meter_m=0.8-3.2.ascii"))
    return 1;

/* Generate the template mesh. */
code=get_chirp_templates(&space);
fprintf(stdout, "%s: get_chirp_templates generated %i templates and"
        " exited with code %i.\n", argv[0], space.n_templates, code);

/* Print a list of template positions in mass space. */
fpout=fopen("templates.ascii", "w");
for(i=0; i<space.n_templates; i++)
    fprintf(fpout, "%f %f\n", space.templates[i].m1,
            space.templates[i].m2);
fclose(fpout);

/* Flag every template, so that their centres will be marked with
   dots on the plot. Normally, a template will be flagged only if
   an error occurred while generating it. */
for(i=0; i<space.n_templates; i++)
    space.templates[i].flag=1;
```

```
/* Plot a postscript diagram of the parameter space. */
code=plot_chirp_templates(space,mag,plot_boundary,plot_patches,
                          plot_ellipses,plot_flags,"templates.ps");
fprintf(stdout,"%s: plot_chirp_templates generated %i pages of"
        " postscript output.\n",argv[0],code);

return 0;
}
```

Author: Teviet Creighton, teviet@tapir.caltech.edu

10 GRASP Routines: Time-Frequency Methods

This section describes the use of software written to extract signals in noise via time frequency methods. Such methods will be useful in extracting signals from poorly modeled or unmodeled sources. The basic procedure we adopt involves the following steps,

- Construction of two dimensional ‘time-frequency’ (TF) maps of the time series data,
- Search for one-dimensional structures in the map.
- Use a statistic based on the length and/or the intensity of the line to determine whether the structure is due to a signal.

Each of the above defined steps can be accomplished by various algorithms. For a detailed discussion of one implementation of this strategy, please refer to [33] and references therein. In this implementation we use the Wigner-Ville distribution to construct the TF map and the Steger’s line detection algorithm to detect the line features in the map and a simple length threshold to determine whether we have detected a signal.

In order to improve the computation/communication ratio during code parallelization we introduce the concepts of segment and subsegment. The master process sends out large chunks of data called data segments. Each data segment contains many subsegments and the slave processes compute the TF map for each subsegment in turn. The sizes of the segments and the subsegments are user defined. Also some points at the beginning and the end of each segments are not analysed. This can, of course, be compensated for by padding. The number of points skipped at the beginning and end are termed PRESAFETY and POSTSAFETY respectively and are user defined.

10.1 Construction of the TF map

As mentioned earlier, there exist many algorithms to construct a “time-frequency” map of a data stream. We have currently implemented three algorithms for the construction of the map. We describe each of these below.

10.1.1 Wigner-Ville Distribution

The Wigner-Ville distribution (WVD) $\rho(t, f)$ is defined by the relation,

$$\rho(t, f) = \int_{-\infty}^{\infty} h\left(t - \frac{\tau}{2}\right) h^*\left(t + \frac{\tau}{2}\right) e^{2\pi i f \tau} d\tau, \quad (10.1.1)$$

where $h(t)$ is the time series data. In practice we use the discrete analog of the previous equation,

$$\rho_{jk} = \sum_{\ell=-N/2}^{N/2} h_{(j-\ell/2)} h_{(j+\ell/2)} e^{2\pi i k \ell / N}. \quad (10.1.2)$$

This appears to presents a minor dilemma, since (10.1.1) contains expressions of the form $h(t - \tau/2)$, which when discretized become $h_{j-k/2}$. This implies that the original data has to be oversampled by at least a factor of 2. We resample our simulated data so that $h_j \equiv h(2j\Delta t)$ accordingly. Among the various algorithms to generated a TF map, we find that the WVD is most suitable for our purpose. We show in figures 71 and 72 WVD distributions of timeseries data containing only noise and timeseries data containing noise and an injected signal. The signal has been injected at an optimal filter signal to noise ration of 8.

Two versions of the Wigner-Ville transform are implemented. Equation (10.1.1) contains expressions of the form $h(t - \tau/2)$, which when discretized become $h_{j-k/2}$. This implies that for $j = 0$ we require

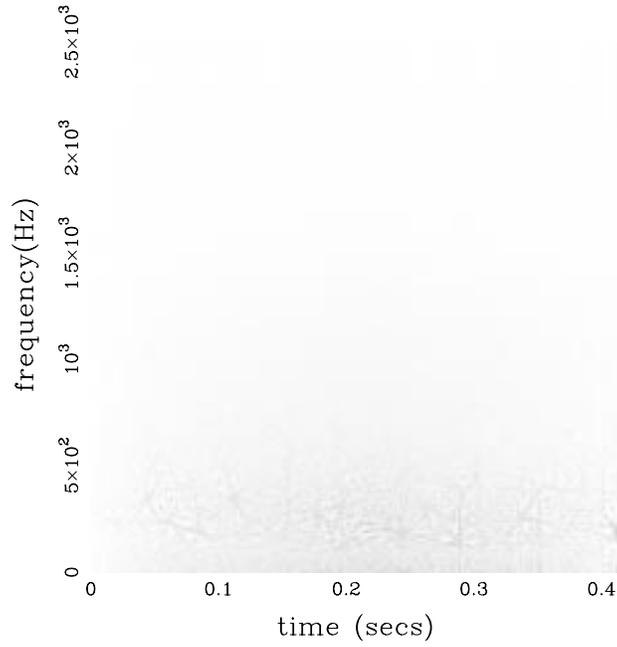


Figure 71: The WVD of simulated initial LIGO noise.

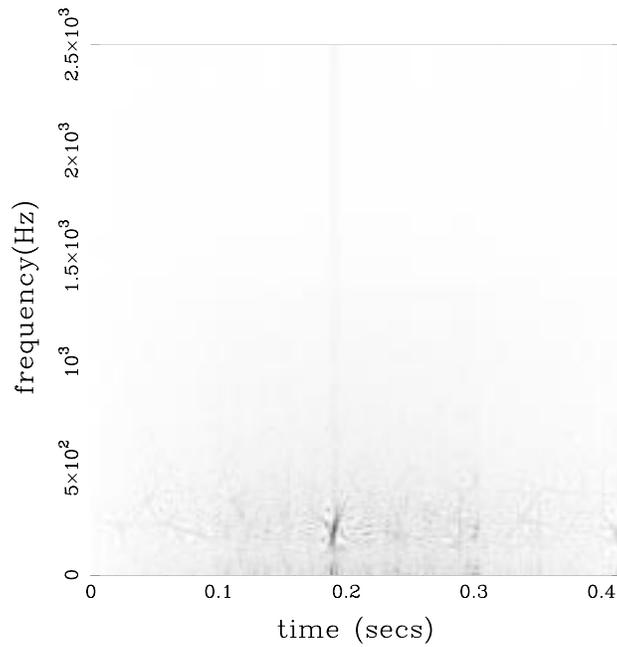


Figure 72: The WVD of simulated initial LIGO noise and a signal embedded at an SNR of 8.

array points with negative indices. There are two ways of handling this problem. One can assume that these array points have a value of zero or one can assume that these array points refer to data points before h_0 . We implement both versions of the Wigner transform. Please note that we have used the former one in [33].

10.1.2 Windowed Fourier transform

The basic idea here is to multiply the data train with a window function and compute the Fourier transform. We use the Welch Window for our Windowed Fourier transforms (WFT). The WFT is defined by the relation,

$$\rho(t, f) = \int_{-\infty}^{\infty} h(\tau) w(\tau - t) e^{2\pi i f \tau} d\tau,$$

where, $w(t, \tau)$ is given as,

$$w(\tau) = 1.0 - \frac{4.0\tau^2}{d^2} \quad : \|\tau\| < d/2 \quad (10.1.3)$$

$$= 0.0 \quad : \|\tau\| \geq d/2, \quad (10.1.4)$$

where d is the width of the window.

10.1.3 Choi-William's distribution

The Choi Williams distribution (CWD), is given by the relation,

$$\rho(t, f) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} h\left(u - \frac{\tau}{2}\right) h^*\left(u + \frac{\tau}{2}\right) \sqrt{\frac{\pi\sigma}{\tau^2}} e^{-\pi^2\sigma\frac{(u-t)^2}{\tau^2} - 2\pi i \tau f} d\tau du, \quad (10.1.5)$$

where σ is the width of the window. As described in section 10.1.1 we again need to oversample the data train at least by a factor of two. Also it is to be noted that the Choi-Williams transform is quite expensive to compute.

10.2 Steger's Line Detection Routines

A gravitational wave signal in interferometer data $h(t)$ should produce a ridge in the TFD $\rho(t, f)$. Therefore, to detect GWs, a ridge detection algorithm (or equivalently line detection algorithm if $\rho(t, f)$ is represented as a gray-scale map as in Fig. 72) is required. Fortunately, there are a number of ridge detection algorithms in the digital image processing and computer vision literature from which to choose.

We use Steger's second-derivative hysteresis-threshold algorithm [35]. The essential idea of this scheme is simple. A ridge in a surface will have high curvature (second derivative of $\rho(t, f)$) in the direction perpendicular to the ridge. Furthermore, the first derivative will vanish at the top of the ridge, since it is a local maximum. Thus, ridges are identified as contiguous sets of point at which $\rho(t, f)$ has a high-curvature local maximum. We have included Stegers code [34] in the GRASP package with minor changes. The main aim of making the the changes was to enable Steger's routines to take a map whose pixels are of type `float` rather than `unsigned char`. For details about this algorithm please see [33, 35]

10.3 Structure: struct struct_tfparam

The structure `struct_tfparam` is the main structure used by the time-frequency routines and the program calling these routines. Some of the fields of this structure are used by the time-frequency routines while others are useful for book keeping. (We will use the abbreviation BK to denote the fields used for Book Keeping.)

The fields of this structure are:

```
struct_tfparam{
```

```
int run_number: BK. An identification label.
```

```
float f_lower: BK. The lower frequency cutoff for the signals injected.
```

```
int start_segment: BK. Used by the calling program to identify the first data segment to analyse.
```

```
int transformtype: The type of the time-frequency transform to be used to construct the map. Should be set to any one of the three Macros defined in file grasp.h, namely, WIGNERTF, WIGNERTF_NP, WFFTWTF, or CHOIWILLIAMS which correspond to the currently implemented transforms namely, the Wigner-Ville transform with zero padding, Wigner-Ville transform without zero padding, the windowed Fourier transform and the Choi-Williams transform respectively.
```

```
int windowwidth: The width (in number of data points) of the window to use for the Choi-Williams and the windowed Fourier transform.
```

```
int offset_step_size: This variable governs the resolution at which the TF map is computed and should normally be set to unity. If this variable is not set to unity then the TF distributions are not computed for every value of time. For example if this variable is set to 2 then the TF distribution is computed at half the resolution.
```

```
int num_of_segments: BK. The number of data segments to analyse.
```

```
float maxpixelval: Can be used to set a threshold on the values of the pixels in the TF map. This value can be computed using the routine compute_scalefactor().
```

```
int DIM: The data dimension of the data segment array.
```

```
int ND: The data dimension of the subsegment array.
```

```
int PD: The dimension of the TF map. Must be less than ND/4.
```

```
int PRE: The number of data points to skip at the beginning of each segment.
```

```
int POST: The number of data points to skip at the end of each segment.
```

```
int TD: Has to be set to ND/PD.
```

```
int FD: Has to be set to ND/(4*PD)
```

```
float rescale_factor: Used to compute maxpixelval via the routine compute_scale_factor(). Has to be set to a number greater than unity.
```

```
float hscale: BK. An arbitrary scaling number.
```

```
int noisetype: BK. The type of noise.
```

```
float srate: The sampling rate.
```

```
}
```

10.4 Function: time_freq_map()

```
void time_freq_map(float *htilde, struct_tfparam *tfs, int index, float **tf-  
general, float **pic)
```

This function is the main routine responsible for creating the time-frequency map. Currently three algorithms are implemented for creating TF maps from time series data, namely, the Wigner-Ville distribution, the Windowed Fourier transform and the Choi-Williams distribution. There are two versions of the Wigner-Ville transform which have both been implemented. This function calls the appropriate transform routine based on the variable (`*tfs`).transformtype.

The arguments are:

float *htilde: Input. The pointer to the beginning of the data segment.

struct_tfparam *tfs: Input. A pointer to a structure (defined in the previous subsection).

int index: Input. The subsegment for which the map is to be computed.

float **tfgeneral: Input. A temporary work space. Memory must be allocated by the calling program. tfgeneral must point to an array of size (`(*tfs).TD * sizeof(*float)`). Each of these pointers must be allocated a space of (`(*tfs).ND * sizeof(float)`) bytes.

float **pic: Output. The computed TF map. Memory must be allocated by the calling program. pic must point to an array of size (`(*tfs).PD * sizeof(*float)`). Each of these pointers must be allocated a space of (`(*tfs).PD * sizeof(float)`) bytes.

In addition to the arguments the structure variable (`*tfs`) contains all the additional parameters for the construction of the map. This structure is defined in the previous subsection.

Author: R. Balasubramanian, bala@chandra.phys.uwm.edu

10.5 Function compute_scalefactor()

```
float compute_scalefactor(float **pic, float rescale, int pdim)
```

This function is used to compute the rescale value for the TF maps. Such a rescaling is required for instance if the line recognition algorithm requires input maps whose pixels are of type unsigned char or short. It returns the maximum pixel value in the TF map multiplied by the argument rescale. The arguments are:

float **pic: Input. The pointer to the TF map.

float rescale: Input. The factor by which to multiply the maximum pixel value.

int pdim: Input. The size of the TF map.

Author: R. Balasubramanian, bala@chandra.phys.uwm.edu

10.6 Function rescale()

```
void rescale(float **pic, int pdim, float rescale)
```

This function is used to rescale and threshold TF maps so that the pixel values lie between zero and unity. The routine simply divides each pixel by the argument rescale and sets any value greater than unity to unity. Since the TF maps contain noise, it is possible for the pixels to attain values which are much higher than the average. To avoid losing information the argument rescale should be obtained by averaging the maximum pixel value in a large number of maps. The routine compute_scalefactor() can be used for this purpose. The arguments are:

float **pic: Input/Output. The pointer to the TF map.

int pdim: Input. The size of the TF map.

float rescale: Input. The factor by which to divide each pixel of the TF map.

Author: R. Balasubramanian, bala@chandra.phys.uwm.edu

10.7 Function normalize_picture()

```
void normalize_picture(float **pic, int pdim)
```

This function is used to normalize the TF map so that the pixel values lie between zero and unity. In other words the maximum value is set to unity and the minimum to zero and the rest of the values are scaled uniformly. This function is useful since the routine plottf() requires the pixels of the TF maps to lie between zero and unity. The arguments are:

float **pic: Input/Output. The pointer to the TF map.

int pdim: Input. The size of the TF map.

Author: R. Balasubramanian, bala@chandra.phys.uwm.edu

10.8 Function gen_quasiperiodic_signal()

```
void gen_quasiperiodic_signal(float *arr, int n, float fa, float fs, float  
pind, float ampind, float timfrac, float freqfrac, int *filled)
```

This routine generates a quasiperiodic signal with both frequency and amplitude increasing in time as power laws. The arguments are:

float *arr: Output. The array to contain the signal points.

int n: Input. The size of the data array.

float fa: Input. The initial frequency of the signal.

float fs: Input. The sampling frequency.

float pind: Input. The exponent for the power law increase in frequency.

float ampind: Input. The exponent for the power law increase in amplitude.

float timfrac: Input. The fraction of the length of the data array for which the signal lasts.

float freqfrac: Input. The fraction of the sampling frequency to be used as the upper cutoff frequency. Typically this should be around 15% of the sampling frequency.

`int *filled` Output. On return `*filled` contains the length of the signal.

Author: R. Balasubramanian, bala@chandra.phys.uwm.edu

10.9 Function: ppmprint()

```
void ppmprint(float **pic, char *file, int pdim)
```

This function is used to print the TF map as a PPM format file. The arguments are:

float **pic: Input. The pointer to the TF map. The pixel values must be between zero and unity.

char *file: Input. A pointer to the name of the PPM file.

int pdim: Input. The size of the TF map.

Author: R. Balasubramanian, bala@chandra.phys.uwm.edu

10.10 Function: pgmprint()

```
void pgmprint(float **pic, char *file, int pdim)
```

This function is used to print the TF map as a PGM format file. The arguments are:

float **pic: Input. The pointer to the TF map. The pixel values must be between zero and unity.

char *file: Input. A pointer to the name of the PGM file.

int pdim: Input. The size of the TF map.

Author: R. Balasubramanian, bala@chandra.phys.uwm.edu

10.11 Function: plottf()

```
void plottf(float **pic, int pdim)
```

This function is used to print the TF map on the screen using GL/MESA calls. The arguments are:

float **pic: Input. The pointer to the TF map. The pixel values must be between zero and unity.

int pdim: Input. The size of the TF map.

Author: R. Balasubramanian, bala@chandra.phys.uwm.edu

10.12 Function: get_line_lens()

```
int get_line_lens(double sigma, double high, double low, int rows, int cols,  
float **pic, char *file)
```

This function detects one dimensional structures in the TF map. The arguments are:

double sigma: Input. The width of the line to be detected in pixels.

double high: Input. The upper threshold on the second directional derivative.

double low: Input. The lower threshold on the second directional derivative.

int rows: Input. The number of rows in the TF map.

int cols: Input. The number of columns. in the TF map.

float **pic: Input. The pointer to the TF map.

char *file: Input. The name of the file to which the output is to be written. The file is written to disk only if at least one line is detected in the map. The first entry in the file is the number of lines detected. Then for each line detected the number of pixels in the line and the sum of the TF map pixel values along the line is recorded. A typical output file looks as shown below:

```
3  
57 44.564186  
24 10.698793  
20 9.513889
```

Author: Warren G. Anderson, warren@ricci.phys.uwm.edu This routine is an interface to the routines by Carsten Steger to detect one dimensional structures in two dimensional images, [34, 35].

10.13 Function: get_lines()

```
int get_lines(double sigma, double high, double low, int rows, int cols,  
float **pic, float **out_img)
```

This function is almost identical with the function get_line_lens() defined in the previous subsection. The only difference being the last argument of the function. The routine returns a map (out_img) whose pixels have the value of either unity if the pixel corresponds to a line point and zero otherwise.

Author: Warren G. Anderson, warren@ricci.phys.uwm.edu This routine is an interface to the routines by Carsten Steger to detect one dimensional structures in two dimensional images, [34, 35].

10.14 Example: `tfmain` program

This program was used to test the algorithm described in the Introduction (see section 10). We generate coloured Gaussian noise and compute the TF map of each data subsegment and then search for one dimensional structures in the map. The overall structure of the code is as follows. We use MPI code to set up a master slave operation. We have a single master which generates data segments containing simulated Gaussian noise with a signal embedded in the segment at an SNR as specified in the input file (`tfmain.in`). Data segments containing only noise are generated by setting $\text{SNR} = 0.0$. These segments are then passed on to the slaves who compute the TF maps, detect the lines in each map and write the output files directly to disk. In order to reduce the communication overheads, the slaves compute many TF maps before they request for more data. Each segment of data contains many subsegments of data. The slave computes the TF map and detects the lines therein for every subsegment in the segment successively.

The program is divided into the following files:

`tfmain.h`: A header file containing function prototypes and parameters.

`tfmain.c`: The main program containing MPI code and organizing the data flow.

`tf_get_data.c`: Routines to generate Gaussian random noise and insert a signal at a given SNR.

`tf_misc.c`: Miscellaneous routines.

`randomseeds`: Contains a single column of random number seeds.

`tfmain.in`: An input file containing various parameters read in by the program.

`MergeSig.dat`: This file contains a single column of floating point numbers and these are equally spaced samples of the coalescence waveform for a pair of $30M_{\odot}$ blackholes. The sampling frequency is 9868.4209 Hz. In addition we also include coalescence waveforms for binaries in the mass range $45M_{\odot} - 70M_{\odot}$. These files are called `MergeSig.dat.*` where `*` is a wildcard denoting the total mass of the binary.

`combine.c`: A program to combine the various output files produced during a run. The original output files are deleted and all the output information is written to a single file.

`readertf.c`: A program to interpret the file produced by `combine.c`.

10.14.1 Environment variables used by `tfmain`

The `tfmain` program uses the environment variable `GRASP_PARAMETERS`, the directory path which contains information to construct the power spectrum for various detectors.

10.14.2 File: tfmain.h

```
/*GRASP: Copyright, 1997,1998 Bruce Allen */

#define DATADIM 65536 /* size of segment (must be power of 2)*/
#define NDIM 4096 /* size of the subsegment to be transformed (must be power of 2)*/
#define PDIM 512 /* dimension of time-freq map; (max value NDIM/4) */
#define PRESAFETY DATADIM/8 /* number of points to ignore at beginning of correlation */
#define POSTSAFETY DATADIM/8 /* number of points to ignore at end of correlation */
#define FDIM (NDIM/(4*PDIM)) /* number of bins to average over in frequency space*/
#define TDIM (NDIM/PDIM) /* number of bins to average over in time space */

/* THE TYPE OF NOISE */
#define NOISE_WHITE 1 /* white noise */
#define NOISE_LIGO_INI 2 /* initial ligo noise curve */

/* THE TYPE OF SIGNAL */
#define INSERT_INSPIRAL 1 /* the inspiral waveform */
#define INSERT_QUAS_PER 2 /* a waveform with power law increase of freq and amp*/
#define INSERT_COALESCENCE 3 /* the full coalescence waveform */

#define RESCALE_FACTOR 2.0 /* used to decide the rescale number*/
#define DEBUG 0 /* For debugging purposes */
#define DEBUG1 0 /* temporary debugging */
#define DEBUG2 0 /* temporary debugging */
#define NOISE_TYPE NOISE_LIGO_INI /* To determine the noise_type */

#ifndef SRATE
#define SRATE 9868.4208984375 /* sample rate in HZ of IFO_DMRO channel */
#endif

/* Function declarations */
void fill_data_with_signal(int, float *, float *, float);
void get_coalescence(float *, int, float, float, int *);
int get_time_series_data();
int gethtilde();
void getshf(int, float *, float);
void gettfparameters();
void master();
float mygasdev(long *);
void noise_gau_col(long *, unsigned long, float *, float *);
void noise_gau_col_fr(long *, unsigned long, float *, float *);
void normalizehtilde(float *, int, float *);
void overwhiten_filter(float *, long, float *);
void picturerawprint(float **);
void realft(float *, unsigned long, int);
void slave();
void timstat(int, FILE *, float *htilde);
void whiten_filter(float *, long, float *);

/* Structure definitions */
typedef struct
{
    int signaltype; /* the type of signal to be used */
    float m1; /* the mass of one of the stars; used if signaltype = 1 */
    float m2; /* the mass of the other star; used if signaltype = 1 */
    float pind; /* the exponent for the power law freq. increase; used if signaltype = 2 */
    float ampind; /* the exponent for the power law amplitude. increase; used if signaltype=2 */
}
```

```
float timfrac;      /* the fraction of time for which the signal lasts in each subsegment
                   ;used if signalttype=2 */
float freqfrac;    /* the bandwidth of the signal as a fraction of the sampling rate;
                   used if signalttype=2 */
float cons_noise_pow; /* an arbitrary scaling factor for the power spectrum */
float snr;         /* the signal to noise ratio */
int signaloffset; /* the offset at which the signal begins in each subsegment */
int addsignal;    /* flag; TO ADD signal 1; if signal is not to be added 0 */
} struct_signalparameters;
```

10.14.3 File: tfmain.c

```
/* To investigate the Time Frequency distribution of chirps */
/* using MPI and GL calls */
/* prototype program */

#include "mpi.h"
#include "grasp.h"
#include "tfmain.h"

/*****
GLOBAL VARIABLES TO BE USED ACROSS FILES
*****/

struct_tfparam tfparam; /* the parameters of the timefrequency program defined in file timefreq.h */
dl_options dlopt; /* the parameters for the linerecognition algorithms*/
struct_signalparameters snpar; /* the parameters of the signal and noise; defined in tfmain.h */
MPI_Comm comm = 91;
long longn=DATADIM;
float *htilde,srate=SRATE;
int reply=1,count=0,counter;
int npoint=DATADIM,numprocs,myid,new_lock=0;

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

char processor_name[256],command[256];
int namelen,mypid;

/* initialize the MPI processes*/
MPI_Init ( &argc, &argv);
MPI_Comm_size ( comm, &numprocs );
MPI_Comm_rank ( comm, &myid );
MPI_Get_processor_name(processor_name,&namelen);

/* Renice the processes */
mypid = getpid();
sprintf(command,"renice 10 %d &\n",mypid);
system(command);
/* For debugging purposes */
#if(DEBUG==1)
if(myid==0){
sprintf(command,"xxgdb tf %d &\n",mypid);
system(command);
}
sleep(20);
#endif

/*allocate space various common arrays */
htilde = (float *) malloc(sizeof(float)*npoint);
/* get the parameters from the file tfmain.in */
gettfparameters();
/* branch off to either the master or the slaves */
if(myid==0)
master();
else
slave();
/* exit */
MPI_Finalize();
```

```
    return 0;
}

void master()
{
    int slaves,recv=0,from,mdata=0,i;
    MPI_Status status;
    char tmp_str[256];
    float *scale;
    FILE *fp,*fp1,*fp2;

    /* create the output directory and copy the input file there */
    sprintf(tmp_str,"mkdir run%02d\n",tfparam.run_number);
    system(tmp_str);
    sprintf(tmp_str,"cp tfmain.in run%02d\n",tfparam.run_number);
    system(tmp_str);
    sprintf(tmp_str,"cp tfmain.h run%02d\n",tfparam.run_number);
    system(tmp_str);
    sprintf(tmp_str,"run%02d/timstat",tfparam.run_number);
    fp = fopen(tmp_str,"w");
    sprintf(tmp_str,"run%02d/rescale",tfparam.run_number);
    fp1 = fopen(tmp_str,"w");
    if(snpar.signaltype==INSERT_COALESCENCE){
        sprintf(tmp_str,"cp MergeSig.dat run%02d\n",tfparam.run_number);
        system(tmp_str);
    }
    sprintf(tmp_str,"run%02d/segments",tfparam.run_number);
    fp2 = fopen(tmp_str,"w");
    sprintf(tmp_str,"printenv > run%02d/environment\n",tfparam.run_number);
    system(tmp_str);
    sprintf(tmp_str,"echo number of processes = %d >> run%02d/prog",numprocs,tfparam.run_number);
    system(tmp_str);
    /* set the rescale parameter */
    snpar.addsignal = 0;
    scale = (float *) malloc(sizeof(float)*numprocs);
    for (slaves=1;slaves<numprocs;slaves++){
        mdata = get_time_series_data();
        MPI_Send(&count, 1, MPI_INT, slaves, 1001, comm);
        MPI_Send(htilde,npoint,MPI_FLOAT,slaves,1002,comm);
    }
    for(i=0;i<tfparam.start_segment;i++){
        mdata = get_time_series_data();
        printf("Skipping segment %d, mdata = %d\n",i,mdata);
        fflush(stdout);
    }
    for(slaves=1;slaves<numprocs;slaves++){
        MPI_Recv(scale+slaves, 1,MPI_FLOAT, slaves, 1004, comm, &status);
        fprintf(fp1,"%d %f\n",slaves,scale[slaves]);
    }
    fclose(fp1);
    /* Compute average of the rescale values */
    tfparam.maxpixelval = 0.0;
    for(slaves=1;slaves<numprocs;slaves++) tfparam.maxpixelval += scale[slaves];
    tfparam.maxpixelval /= (numprocs-1);
    printf(" The average rescale value : %f\n",tfparam.maxpixelval);
    fflush(stdout);
    free(scale);
    /* send the rescale value to all the slaves */
    MPI_Bcast(&tfparam.maxpixelval, 1, MPI_FLOAT, 0, comm);
}
```

```
for(slaves=1;slaves<numprocs;slaves++){
    MPI_Recv(&reply, 1,MPI_INT, slaves , 1003, comm, &status);
    MPI_Recv(&counter , 1,MPI_INT, slaves , 1003, comm, &status);
}

count= tfparam.start_segment;
recv = tfparam.start_segment;
snpar.addsignal = 1;
/* all set to start the actual simulation loop over
   slaves and send the datasegments to the slaves */
sprintf(tmp_str,"date >> run%02d/prog\n",tfparam.run_number);
system(tmp_str);
for (slaves=1;slaves<numprocs;slaves++){
    /* get the data */
    mdata = get_time_series_data();
#ifdef DEBUG1
    graph(htilde,npoint,1);
    sleep(5);
#endif
    timstat(count, fp, htilde);
    MPI_Send(&count, 1, MPI_INT, slaves , 1001, comm);
    MPI_Send(htilde,npoint,MPI_FLOAT,slaves,1002,comm);
    printf("master   : sent segment %d to slave %d mdata = %d\n",count,slaves,mdata);
    fflush(stdout);
    count++;
}
/* wait for the slaves to send the done message and send fresh datasegments */
while(recv<count){
    MPI_Recv(&reply , 1, MPI_INT, MPI_ANY_SOURCE, 1003, comm, &status);
    from = status.MPI_SOURCE;
    MPI_Recv(&counter , 1,MPI_INT, from , 1003, comm, &status);
    printf("master   : Received reply for segment %d back from slave %d\n",counter,from);
    fflush(stdout);
    recv++;
    if(mdata) mdata = get_time_series_data();
    printf("master   :count=%d,recv=%d,mdata=%d\n",count,recv,mdata);
    if((count<tfparam.num_of_segments)&&(mdata)) {
        fprintf(fp2,"%d %d %d \n",count, from,new_lock);
        MPI_Send(&count, 1, MPI_INT, from , 1001, comm);
        MPI_Send(htilde, npoint, MPI_FLOAT, from , 1002, comm);
        printf("master   : sent segment %d to slave %d\n",count,from);
        fflush(stdout);
        timstat(count, fp,htilde);
        count++;
    }
    /* send slave the termination message */
    else{
        counter=-1;
        printf("master   :Sending termination signal to slave %d \n",from);
        MPI_Send(&counter, 1, MPI_INT, from , 1001, comm);
    }
}
fclose(fp);
fclose(fp2);
sprintf(tmp_str,"date >> run%02d/prog\n",tfparam.run_number);
system(tmp_str);
return;
}
```

```
void slave()
{
    MPI_Status status;
    float *tfgeneral[TDIM],*pic[PDIM],*out_img[PDIM],scalefactor=0.0;
    int i,rows=PDIM,cols=PDIM,noofsub,ind;
    char filen[256];
    static int first = 1;

    /* the number of subsegments in each segment of data */
    noofsub = (DATADIM-(POSTSAFETY+PRESAFETY))/NDIM;
    /* allocate memory of the timefrequency block and for the picture matrices*/
    for(i=0;i<TDIM/tfparam.offset_step_size;i++)
        tfgeneral[i] = (float *)malloc(sizeof(float)*NDIM);
    for(i=0;i<PDIM;i++){
        pic[i] = (float *)malloc(sizeof(float)*PDIM);
        out_img[i] = (float *)malloc(sizeof(float)*PDIM);
    }

    /* loop for receiving the data */
    while(1){
        MPI_Recv(&count,1,MPI_INT,0,1001,comm,&status);
        /* Terminate if count is less than 0*/
        if(count<0) return;
        /* Recv the htilde array from the master*/
        MPI_Recv(htilde, npoint, MPI_FLOAT, 0, 1002, comm, &status);
        printf("slave %d: received segment %d from master\n",myid,count);
        fflush(stdout);
        /* compute the time-frequency maps for the data segment */
        for(ind=0;ind<noofsub;ind++){
            /* compute the time-frequency maps for each subsegment */
            time_freq_map(htilde, &tfparam, ind,tfgeneral,pic);
            if(first){
                scalefactor += compute_scalefactor(pic,tfparam.rescale_factor,PDIM);
#ifdef DEBUG1
                printf("slave %d: nofsub = %d  scalefactor = %f\n",myid,ind,compute_scalefactor(pic,t
                fflush(stdout);
#endif
                if(ind==(noofsub-1)){
                    first = 0;
                    scalefactor /= noofsub;
                    MPI_Send(&scalefactor,1,MPI_FLOAT, 0, 1004, comm);
                    MPI_Bcast(&tfparam.maxpixelval, 1, MPI_FLOAT, 0, comm);
                }
            }
            else{
                rescale(pic,PDIM,tfparam.maxpixelval);
                sprintf(filen, ". /run%02d/out_%d.%02d", tfparam.run_number, count, ind);
                get_line_lens(dlopt.sigma,dlopt.high,dlopt.low,rows,cols,pic,filen);
#ifdef DEBUG2
                if((ind==0)&&(count==0))
                    ppmprint(pic,"picture.ppm",PDIM);
#endif
#ifdef HAVE_GL
                /* if you want to display the TF map on the screen */
                if(myid==1)
                    plottf(pic,PDIM);
#endif
                printf("slave %d: nofsub = %d\n",myid,ind);
                fflush(stdout);
            }
        }
    }
}
```

```
    }  
  }  
  /* inform master that you have finished the current segment */  
  MPI_Send(&reply, 1, MPI_INT, 0, 1003, comm);  
  /* send the counter back to the master as a check */  
  MPI_Send(&count, 1, MPI_INT, 0, 1003, comm);  
}  
}
```

10.14.4 File: tf_get_data.c

```
#include "grasp.h"
#include "tfmain.h"

float *twice_inv_noise,*ch0,*ch1,*buff,*shf,*shf_root,deltaf;
extern float *htilde,srate;
extern int npoint;
extern long longn;
extern struct_tfparam tfparam;
extern struct_signalparameters snpar;
long idummy=-89473884; /* initialized to a random seed*/

int get_time_series_data()
{
    return gethtilde();
}

int gethtilde()
{
    int i,order=1,err_cd_sprs=4000,filled;
    static int first=1;
    float t_coal,tempfloat;
    FILE *randfp;

    if(first){
        first = 0;
        /* read in the random number seed */
        if((randfp = fopen("randomseeds","r"))==NULL){
            printf("the randomseeds file is not present\n");
            printf("Please create a file called randomseeds which contains\n");
            printf("a column of negative random numbers seeds \n");
            exit(-1);
        }
        for(i=0;i<tfparam.run_number;i++){
            if(fscanf(randfp,"%ld\n",&idummy)<0){
                printf("the randomseeds file does not contain enough random numbers\n");
                exit(-1);
            }
        }
        fclose(randfp);
        /* allocate memory for the inverse power spectrum, signal arrays and buffers*/
        twice_inv_noise = (float *) malloc(sizeof(float)*(npoint/2 + 1));
        ch0 = (float *) malloc(sizeof(float)*npoint);
        ch1 = (float *) malloc(sizeof(float)*npoint);
        buff = (float *) malloc(sizeof(float)*npoint);
        shf = (float *)malloc (sizeof(float)*(npoint/2+1));
        shf_root = (float *)malloc (sizeof(float)*(npoint/2+1));
        /* initialize the signal arrays to zero */
        for(i=0;i<npoint;i++) ch0[i]=ch1[i]=0.0;
        /* switch between the signal types */
        switch(snpar.signaltype){
            case INSERT_QUAS_PER:
                gen_quasiperiodic_signal(ch1,NDIM,tfparam.f_lower,srate,snpar.pind, snpar.ampind,snpar.ampind,snpar.freqfrac,&filled);
                break;
            case INSERT_INSPIRAL:
                make_filters(snpar.m1,snpar.m2,ch0,ch1,tfparam.f_lower,npoint,
                    srate,&filled,&t_coal, err_cd_sprs, order);
        }
    }
}
```

```

        break;
    case INSERT_COALESCENCE:
        get_coalescence(ch1,npoint,tfparam.f_lower,srate,&filled);
        break;
}
/* offset the signal to the right by tfparam.signaloffset points */
for(i=filled-1;i>=0;i--) ch1[i+snpar.signaloffset] = ch1[i];
for(i=0;i<snpar.signaloffset;i++) ch1[i] = 0.0;
/* copy ch1 to ch0 */
for(i=0;i<npoint;i++)
    ch0[i] = ch1[i];
/* take the Fourier transform of the signal*/
realft(ch0-1, longn, 1);
/* get the power spectrum for noise */
deltaf = srate/npoint;
getshf(npoint/2+1, shf, deltax);
for(i=0;i<=npoint/2;i++) twice_inv_noise[i] = 1./shf[i];
for(i=0;i<=npoint/2;i++) shf_root[i] = sqrt(shf[i]);
/* normalize the signal to a particular SNR*/
correlate(buff, ch0, ch0, twice_inv_noise, npoint);
tempfloat = snpar.snr/sqrt(buff[0]);
fill_data_with_signal(npoint, ch1, ch0, tempfloat);
realft(ch0-1, longn, 1);
}
noise_gau_col_fr(&idummy, npoint, htilde, shf_root);
/* add the signal to the noise */
if(snpar.addsignal)
    for(i=0;i<npoint;i++) htilde[i] += ch0[i];
over_whiten_filter(htilde, npoint, twice_inv_noise);
/* zero out the higher frequency to avoid aliasing */
if((tfparam.transformtype==WIGNERTF)||((tfparam.transformtype==CHOIWILLIAMS))
    for(i=npoint/2;i<npoint;i++) htilde[i] = 0.0;
realft(htilde - 1, longn, -1);
return DATADIM;
}

```

10.14.5 File: tfmain.in

This file acts as the input to the tfmain program. The routine `gettfparameters()` defined in the file `tf_misc.c` reads in this input file. The file contains dummy strings describing the parameters followed by that parameter. We list below the various input parameters. The default values for the various parameters are the ones used by us in investigating the efficiency of the algorithm described in this section and is documented in [33]. A run with these parameters reproduces the point corresponding to the $60M_{\odot}$ curve at an SNR of 10 in Figure 5 of [33]. For convenience we reproduce the figure here, 73. Please note that the parameters defined in file `tfmain.h` must also be unchanged to reproduce the results of our paper.

`run_number` : This is used when you need to make multiple runs of the code for different parameters. The name of the directory to write the output files is set to be `./run$(run_number)` where `$(run_number)` is the value of the parameter `run_number`. Also the random number seed used is determined by this number. The file `randomseeds` contains a single column of user generated random numbers which are used as seeds and the `run_number` determines which random seed to use and is essentially the random seed on the `run_numberth` line in the file `randomseeds`.

`flo` : This is the lower frequency cutoff for the generated signal.

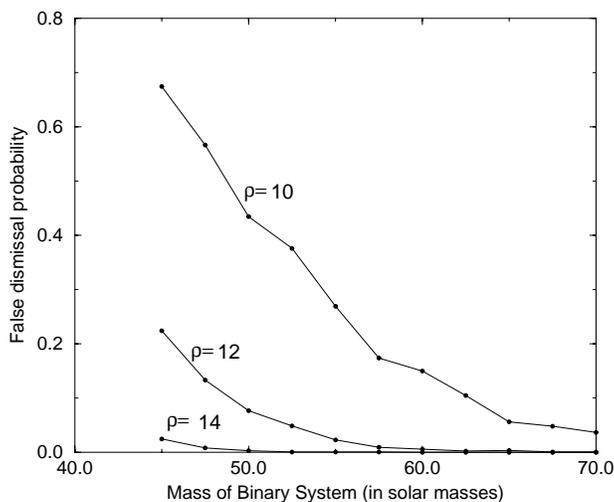


Figure 73: False dismissal probability as a function of mass. The three curves correspond to three different values the optimal filter signal-to-noise ratio. With the parameters we have chosen, our method tends to work better for higher mass binaries, where the energy is more localized in the TF map.

`start_segment` : the first segment to start analysing; segments are numbered 0 onwards.

`transform_type` : To select between the Wigner-Ville(1), windowed Fourier transform(2) and Choi-Williams distribution (3), and Wigner-Ville with no zero padding (4).

`window_width` : the size of the window used in the windowed Fourier transform.

`offset_step_size` : to be set to unity.

`signal_type` : the type of signal to insert in the subsegments, inspiral waveform (1), quasiperiodic waveform with power law increase in frequency and amplitude (2), coalescence waveform(3).

`signal_offset` : the offset at which to insert the signal in each subsegment.

`m1` : mass of the star; used if `signal_type` = 1

`m2` : mass of the other star; used if `signal_type` = 1

`pinde` : the exponent for the power law increase in frequency; used if `signal_type` = 2

`ampinde` : the exponent for the power law increase in amplitude; used if `signal_type` = 2

`timfrac` : the fraction of the subsegment for which the signal lasts; used if `signal_type` = 2

`snr` : the signal to noise ratio at which to insert the signal

`number_of_segments` : the number of segments to analyse

`dl_sigma` : the value of the thickness of the lines expected in the map in pixels.

`dl_high` : used as a threshold to determine whether a pixel is part of a curve.

10.14.6 File: `combine.c`

Usage: `combine directory_name nseg nsubseg`

where `directory_name` is the name of the directory containing the output files of the `tfmain` program *e.g.* `run01`. The argument `nseg` is the number of segments which have been analysed. The argument `nsubseg` is the number of subsegments in each segment of data.

The program outputs a single file called `allcurv.dat` in the output directory. The format of the file is as follows. There is one line for every subsegment analysed. The first number in each line is simply an index from 0 to `nseg*nsubseg - 1`. The next number in each line is the number of lines detected and the subsequent numbers are the number of pixels and the strength of each line in a alternating sequence. The original output files are deleted.

10.14.7 File: `readertf.c`

Usage: `readertf directory_name`

where `directory_name` is the name of the directory containing the file `allcurv.dat`. This program reads this file and writes a new file called `curves.dat`. The basic purpose is to select the longest line in each analysed subsegment. The format of the file is described below. There is again one line for every analysed data subsegment. The first column is an index ranging from 0 to the total number of data subsegments analysed. The subsequent columns are, the number of curves found, the length of the longest curve, the strength of the curve, and the average strength for that curve respectively. The average strength is just the strength divided by the length.

11 GRASP Routines: Stochastic background detection

11.1 Data File: detectors.dat

This file contains site location and orientation information, a convenient name for the detector, and filenames for the detector noise power spectrum and whitening filter, for 11 different detector sites. These site are:

- (1) Hanford, Washington LIGO site,
- (2) Livingston, Louisiana LIGO site,
- (3) VIRGO site,
- (4) GEO-600 site,
- (5) Garching site,
- (6) Glasgow site,
- (7) MIT 5 meter interferometer,
- (8) Caltech 40 meter interferometer,
- (9) TAMA-300 site,
- (10) TAMA-20 site,
- (11) ISAS-100 site.

As explained below, information for additional detector sites can be added to `detectors.dat` as needed. [In fact, there are many additional sites currently in the file – look at it to see for yourself. For example, this file and the referenced parameter files now include the 7 distinct stages of “enhanced LIGO”. Thus a given site, for example the Hanford Washington LIGO site, has a number of different entries, corresponding to different noise power spectra.]

The data contained within this file is formatted as follows: Any line beginning with a # is regarded as a comment. All other lines are assumed to begin with an integer (which is the site identification number) followed by five floating point numbers and three character strings, each separated by *white* space (i.e., one or more spaces, which may include tabs). The first two floating point numbers specify the location of the central station (the central vertex of the two detector arms) on the earth’s surface: The first number is the latitude measured in degrees North of the equator; the second number is the longitude measured in degrees West of Greenwich, England. The third floating point number specifies the orientation of the first arm of the detector, measured in degrees counter-clockwise from true North. The fourth floating point number specifies the orientation of the second arm of the detector, also measured in degrees counter-clockwise from true North. The fifth floating point number is the arm length, in cm. The three character strings specify: (i) a convenient name (e.g., VIRGO or GEO-600) for the detector site, (ii) the name of a data file that contains information about the noise power spectrum of the detector, and (iii) the name of a data file that contains information about the spectrum of the whitening filter of the detector. (We will say more about the content and format of these two data files in Secs. 11.4 and 11.5.) The information currently contained in `detectors.dat` is shown below:

```
#
# Hanford, Washington LIGO Site (initial detector)
# Fred Raab fjr@ligo.caltech.edu
1 46.45236 119.40753 36.8 126.8 4.e5 LIGO-WA_init noise_ligo_init.dat whiten_ligo_init.dat
#
# Livingston, Louisiana LIGO Site (initial detector)
# Fred Raab fjr@ligo.caltech.edu
2 30.56277 90.77425 108.0 198.0 4.e5 LIGO-LA_init noise_ligo_init.dat whiten_ligo_init.dat
#
# VIRGO Site
# Biplab Bhawal biplab@iucaa.iucaa.ernet.in
```

```
# 3 43.3 -10.1 71.5 341.5
# Raffaele Flaminio flaminio@lapphp0.in2p3.fr
# Carlo Bradaschia BRADASCHIA@VAXPIA.PI.INFN.IT
# Rosa Poggiani POGGIANI@pisa.infn.it
3 43.6333 -10.5 71.5 341.5 3.e5 VIRGO noise_virgo.dat whiten_virgo.dat
#
# GEO-600 as of April 1995
# Albrecht Ruediger atr@mpg.mpg.de
4 52.2467 -9.82167 25.94 291.61 6.e4 GEO-600 noise_geo.dat whiten_geo.dat
#
# Garching 30 Meter Interferometer
# Albrecht Ruediger atr@mpg.mpg.de
5 48.244 -11.675 329.0 239.0 3.e3 Garching-30 XXXXX XXXXX
#
# Glasgow 10 Meter Interferometer
# Albrecht Ruediger atr@mpg.mpg.de
# 6 55.86 4.23 77.0 167.0
# Jim Hough hough@physics.gla.ac.uk
6 55.8667 4.28333 62.0 152.0 1.e3 Glasgow-10 XXXXX XXXXX
#
# MIT 5 Meter Interferometer
# Gabriela Gonzalez gg@tristan.mit.edu
7 42.3667 71.1 34.5 304.5 5.e2 MIT-5 XXXXX XXXXX
#
# Caltech 40 Meter Interferometer NEEDS CORRECTION
# Fred Raab fjr@ligo.caltech.edu
8 34.1667 118.133 180.0 270.0 4.e3 Caltech-40 noise_40.dat whiten_40.dat
#
# TAMA 300 Meter
# Masa-Katsu Fujimoto fujimoto@gravity.mtk.nao.ac.jp
9 35.6766 -139.536 90.0 180.0 3.0e4 TAMA-300 noise_tama.dat whiten_tama.dat
#
# TAMA 20 Meter
# Masa-Katsu Fujimoto fujimoto@gravity.mtk.nao.ac.jp
10 35.6751 -139.536 45.0 315.0 2.0e3 TAMA-20 XXXXX XXXXX
#
# ISAS 100 Meter delay line
# Hide Mizuno hide@pleiades.sci.isas.ac.jp
11 35.5678 -139.467 42.0 135.0 1.0e4 ISAS-100 XXXXX XXXXX
#
```

Site information for new (or hypothetical) detectors can be added to `detectors.dat` by simply adhering to the above data format. For example, as the noise in the LIGO detectors improves, one can accommodate these changes in `detectors.dat` by adding additional lines that have the same site location and orientation information as the “old” detectors, but refer to different noise power spectra and whitening filter data files. The only other requirement is that the site identification numbers for these “new and improved” detectors be different from the old site identification numbers, so as to avoid any ambiguity. Explicitly, one could add the following lines to `detectors.dat` to include information about the advanced LIGO detectors:

```
#
# Hanford, Washington LIGO Site (advanced detector)
# Fred Raab fjr@ligo.caltech.edu
12 46.45236 119.40753 36.8 126.8 4.e5 LIGO-WA_adv noise_ligo_adv.dat whiten_ligo_adv.dat
#
# Livingston, Louisiana LIGO Site (advanced detector)
```

```
# Fred Raab fjr@ligo.caltech.edu
13 30.56277 90.77425 108.0 198.0 4.e5 LIGO-LA_adv noise_ligo_adv.dat whiten_ligo_adv.dat
#
```

The file `detectors.dat` currently resides in the `parameters` subdirectory of GRASP. In order for the stochastic background routines and example programs that are defined in the following sections to be able to access the information contained in this file, the user must set the environment variable `GRASP_PARAMETERS` to point to this directory. For example, a command like:

```
setenv GRASP_PARAMETERS /usr/local/GRASP/parameters
```

should do the trick. If, however, you want to modify this file (e.g., to add another detector or to add another noise curve), then just copy the `detectors.dat` file to your own home directory, modify it, and set the `GRASP_PARAMETERS` environment variable to point to this directory.

Comment: If you happen to find an error in the `detectors.dat` file, *please* communicate it to the caretakers of GRASP.

11.2 Function: `detector_site()`

```
void detector_site(char *detectors_file, int site_choice, float site_parameters[9],
char *site_name, char *noise_file, char *whiten_file)
```

This function calculates the components of the position vector of the central station, and the components of the two vectors that point along the directions of the detector arms (from the central station to each end station), for a given choice of detector site, using information contained in an input data file. This function can also be used to obtain the latitude, longitude, arm orientations, and arm length of a detector site. This function also outputs three character strings that specify the site name, the name of a data file containing the detector noise power information, and the name of a data file containing information about the detector whitening filter, respectively.

The arguments of `detector_site()` are:

`detectors_file`: Input. A character string that specifies the name of a data file containing detector site information. This file is most likely the `detectors.dat` data file described in Sec. 11.1. If the file is different from `detectors.dat`, it must have the same data format as `detectors.dat`, and it must reside in the directory pointed to by the `GRASP_PARAMETERS` environment variable (which you may set as you wish). If you want to use the `detectors.dat` file distributed with GRASP, use a command like:

```
setenv GRASP_PARAMETERS /usr/local/GRASP/parameters
```

to point to the directory containing this file. If you want to modify this file (e.g., to add another detector or to add another noise curve), then just copy the `detectors.dat` file to your own home directory, modify it, and set the `GRASP_PARAMETERS` environment variable to point to this directory.

`site_choice`: Input. An integer value used as an index into the input data file. The absolute value of `site_choice` should be chosen to match the site identification number for one of the detectors contained in this file. The integer can be positive or negative depending on whether the user wants the positions of the end stations (positive), or simply the latitude, longitude, arm orientation and length (negative).

`site_parameters`: Output. If `site_choice` was positive, `site_parameters[0..8]` is an array of nine floating point variables that define the position of the central station of the detector site and the orientation of its two arms. The three-vector `site_parameters[0..2]` are the (x, y, z) components (in cm) of the position vector of the central station, as measured in a reference frame with the origin at the center of the earth, the z -axis exiting the North pole, and the x -axis passing out the line of 0° longitude. The three-vector `site_parameters[3..5]` are the (x, y, z) components (in cm) of a vector pointing along the direction of the first arm (from the central station to the end station). The three-vector `site_parameters[6..8]` are the (x, y, z) components (in cm) of a vector pointing along the direction of the second arm (from the central station to the end station). If `site_choice` was negative, `site_parameters[0]` contains the site latitude (degrees north), `site_parameters[1]` contains the site longitude (degrees west), `site_parameters[2]` contains the orientation of the first arm (degrees CCW from North), `site_parameters[3]` contains the orientation of the second arm (degrees CCW from North), and `site_parameters[4]` contains the armlength (in cm). In this case, the unused elements `site_parameters[5..7]` are unchanged.

`site_name`: Output. A character string that specifies a convenient name (e.g., VIRGO or GEO-600) for the chosen detector site.

`noise_file`: Output. A character string that specifies the name of a data file containing information

about the noise power spectrum of the detector. (See Sec. 11.4 for more details regarding the content and format of this data file.)

`whiten_file`: Output. A character string that specifies the name of a data file containing information about the spectrum of the whitening filter of the detector. (See Sec. 11.5 for more details regarding the content and format of this data file.)

`detector_site()` reads input data from the file specified by `detectors_file`. This file is searched (linearly from top to bottom) until the absolute value of `site_choice` matches the site identification number for one of the detectors contained in this file. The site location and orientation information for the chosen detector site are then read into variables local to `detector_site()`. If `site_choice` was negative, this information is returned in the array `site_parameters[]`; otherwise the values contained in the array `site_parameters[]` are calculated from these input variables using standard equations from spherical analytic geometry. (A correction *is* made, however, for the oblateness of the earth, using information contained in Ref. [37].) The `site_name`, `noise_file`, and `whiten_file` character strings are simply copied from input data file. If `site_choice` does not match any of the site identification numbers, `detector_site()` prints out an error message and aborts execution.

Authors: Bruce Allen, ballen@dirac.phys.uwm.edu, and Joseph Romano, romano@csd.uwm.edu

Comments: None.

11.3 Comment: noise power spectra for “advanced” LIGO & the Cutler-Flanagan model

During the years 1992-96, the upcoming LIGO experiment was planned in two stages, an “initial” and an “advanced” stage. These words were taken from an important article in *Science* [38] which described the LIGO plans. (The “advanced” stage has now been supplanted by a series of seven enhancements and is now described as “enhanced” LIGO).

During this period 1992-96, research on data analysis algorithms made use of detector noise curves taken from the *Science* article. Unfortunately two of the figures in the article (Figs. 7 and 10) which gave the noise curve were inconsistent, and also inconsistent with the parameters given in the article. The GRASP parameters/ directory contains a noise curve corresponding to Fig. 7, and another noise curve, generally called the “Cutler and Flanagan” approximation, which is an approximation to the curve used in Fig. 10.

Note that the noise level in Fig. 7 in the *Science* article [38] is a factor of 3 in h_{rms} [or a factor of ~ 10 in $S_n(f)$] lower than that of Fig. 10 in between ~ 10 Hz and ~ 70 Hz. Kip Thorne has informed us that Fig. 10 is the correct figure and Fig. 7 is in error, and that the error does not appear in the corresponding figure V.4 of the 1989 LIGO proposal. The error can be seen by inserting the parameter values $m = 1000$ kg, $f_0 = 1$ Hz, and $Q_0 = 10^9$ given in [38] into the standard equation for suspension thermal noise due to viscous damping, as given in, e.g., Eq. (4.3) of Reference [15]. The resulting noise level is a factor of 3 higher than the noise level shown in Fig. 7, and agrees with the noise level of Fig. 10. Note however that the noise curve of Fig. 7 has been adopted and used by several researchers as the “advanced ligo noise curve”, and that the GRASP “advanced” advanced noise curve is that of Fig. 7.

The Cutler and Flanagan approximation to the advanced ligo noise curve is

$$S_n(f) = \frac{1}{5}S_0 \left[\frac{f_0^4}{f^4} + 2 \left(1 + \frac{f^2}{f_0^2} \right) \right] \quad (11.3.1)$$

for $f \geq 10$ Hz, and $S_n(f) = \infty$ for $f < 10$ Hz, where $S_0 = 3 \times 10^{-48} \text{ Hz}^{-1}$ and $f_0 = 70$ Hz. This is Eq. (2.1) of Reference [21] with S_0 replaced by $S_0/5$ to correct a typo in the published paper. The noise curve (11.3.1) is an approximate analytic fit to the advanced noise curve shown in Fig. 10 (not Fig. 7) of the LIGO *Science* article [38]. It is the GRASP noise curve `noise_cutler_flanagan.dat`. The accuracy of the fit is fairly good but not a perfect fit – in particular the noise curve (11.3.1) is slightly larger than the noise curve in [38] at high frequencies. A slightly more accurate fit has been obtained by Scott Hughes and Kip Thorne (quoted in Reference [6]), which uses the same functional form but the slightly different parameter values $f_0 = 75$ Hz and $S_0 = 2.3 \times 10^{-48}$ with a lower shutoff frequency of 12 Hz.

11.4 Function: noise_power()

```
void noise_power(char *noise_file, int n, float delta_f, double *power)
```

This function calculates the noise power spectrum $P(f)$ of a detector at a given set of discrete frequency values, using information contained in a data file.

The arguments of `noise_power()` are:

`noise_file`: Input. A character string that specifies the name of a data file containing information about the noise power spectrum $P(f)$ of a detector. Like the `detectors.dat` file described in Sec. 11.1, the noise power data file should reside in the directory pointed to by the `GRASP_PARAMETERS` environment variable (which you may set as you wish). If you want to use the noise power spectrum data files distributed with GRASP, use a command like:

```
setenv GRASP_PARAMETERS /usr/local/GRASP/parameters
```

to point to the directory containing these files. If you want to use your own noise power spectrum data files, then simply set the `GRASP_PARAMETERS` environment variable to point to the directory containing these files. Note, however, that if a program needs to access *both* detector site information and noise power spectrum data, then all of the files containing this information should reside in the *same* directory. (A similar remark applies for the whitening filter data files described in Sec. 11.5.)

`n`: Input. The number N of discrete frequency values at which the noise power spectrum $P(f)$ is to be evaluated.

`delta_f`: Input. The spacing Δf (in Hz) between two adjacent discrete frequency values: $\Delta f := f_{i+1} - f_i$.

`power`: Output. `power[0..n-1]` is an array of double precision variables containing the values of the noise power spectrum $P(f)$. These variables have units of $\text{strain}^2/\text{Hz}$ (or seconds). `power[i]` contains the value of $P(f)$ evaluated at the discrete frequency $f_i = i\Delta f$, where $i = 0, 1, \dots, N - 1$.

The input data file specified by `noise_file` contains information about the noise power spectrum $P(f)$ of a detector. The data contained in this file is formatted as follows: Any line beginning with a # is regarded as a comment. All other lines are assumed to consist of two floating point numbers separated by white space. The first floating point number is a frequency f (in Hz); the second floating point number is the square root of the *one-sided* noise power spectrum $P(f)$, evaluated at f . $P(f)$ is defined by equation (3.18) of Ref. [36]:

$$\langle \tilde{n}^*(f) \tilde{n}(f') \rangle =: \frac{1}{2} \delta(f - f') P(f). \quad (11.4.1)$$

Here $\langle \rangle$ denotes ensemble average, and $\tilde{n}(f)$ is the frequency spectrum (i.e., Fourier transform) of the strain $n(t)$ produced by the noise intrinsic to the detector. $P(f)$ is a non-negative real function, having units of $\text{strain}^2/\text{Hz}$ (or seconds). It is defined with a factor of 1/2 to agree with the standard definition used by instrument builders. The total noise power is the integral of $P(f)$ over all frequencies from 0 to ∞ (not from $-\infty$ to ∞). Hence the name *one-sided*.

Since the frequency values contained in the input data file need not agree with the desired frequencies $f_i = i\Delta f$, `noise_power()` must determine the desired values of the noise power spectrum by doing an interpolation/extrapolation on the input data. `noise_power()` performs a cubic spline interpolation, using the *Numerical Recipes in C* routines `spline()` and `splint()`. `noise_power()` assumes that the length of the input data is ≤ 65536 , and it uses boundary conditions for a natural spline (i.e., with zero second derivative on the two boundaries). `noise_power()` also squares the output of the `splint()` routine, since the desired values are $P(f)$ —and not their square roots (which are contained in the input data file).

Authors: Bruce Allen, `ballen@dirac.phys.uwm.edu`, and Joseph Romano, `romano@csd.uwm.edu`

Comments: In order for the cubic spline interpolation routines to yield approximations to $P(f)$ that are not contaminated by spurious DC or low frequency (e.g., approximately 1 Hz) components, it is important that the input data file specified by `noise_file` contain noise power information down to, and including, zero Hz. This information can be added in “by hand,” for example, if the experimental data for the noise power spectrum only goes down to 1 Hz. In this case, setting the values of $\sqrt{P(f)}$ at 0.0, 0.1, 0.2, \dots , 0.9 Hz equal to its 1 Hz value seems to be sufficient. (See Sec. 11.5 for a similar comment regarding `whiten()`.)

11.5 Function: whiten()

void whiten(char *whiten_file, int n, float delta_f, double *whiten_out)
This function calculates the real and imaginary parts of the spectrum $\tilde{W}(f)$ of the whitening filter of a detector at a given set of discrete frequency values, using information contained in a data file.

The arguments of whiten() are:

whiten_file: Input. A character string that specifies the name of a data file containing information about the spectrum $\tilde{W}(f)$ of the whitening filter of a detector. Like the `detectors.dat` and noise power spectrum data files described in Secs. 11.1 and 11.4, the whitening filter data file should reside in the directory pointed to by the `GRASP_PARAMETERS` environment variable (which you may set as you wish). If you want to use the whitening filter data files distributed with GRASP, use a command like:

```
setenv GRASP_PARAMETERS /usr/local/GRASP/parameters
```

to point to the directory containing these files. If you want to use your own whitening filter data files, then simply set the `GRASP_PARAMETERS` environment variable to point to the directory containing these files. Note, however, that if a program also needs to access either detector site information or noise power spectrum data, then all of the files containing this information should reside in the *same* directory.

n: Input. The number N of discrete frequency values at which the real and imaginary parts of the spectrum $\tilde{W}(f)$ of the whitening filter are to be evaluated.

delta_f: Input. The spacing Δf (in Hz) between two adjacent discrete frequency values: $\Delta f := f_{i+1} - f_i$.

whiten_out: Output. `whiten_out[0..2*n-1]` is an array of double precision variables containing the values of the real and imaginary parts of the spectrum $\tilde{W}(f)$ of the whitening filter. These variables have units rHz/strain (or $\text{sec}^{-1/2}$), which are inverse to the units of the square root of the noise power spectrum $P(f)$. `whiten_out[2*i]` and `whiten_out[2*i+1]` contain, respectively, the values of the real and imaginary parts of $\tilde{W}(f)$ evaluated at the discrete frequency $f_i = i\Delta f$, where $i = 0, 1, \dots, N - 1$.

The input data file specified by `whiten_file` contains information about the spectrum $\tilde{W}(f)$ of the whitening filter of a detector. The data contained in this file is formatted as follows: Any line beginning with a # is regarded as a comment. All other lines are assumed to consist of three floating point numbers, each separated by white space. The first floating point number is a frequency f (in Hz). The second and third floating point numbers are, respectively, the real and imaginary parts of the spectrum $\tilde{W}(f)$, evaluated at f . These last two numbers have units of rHz/strain (or $\text{sec}^{-1/2}$). This is because the whitening filter is, effectively, the inverse of the amplitude $\sqrt{P(f)}$ of the noise power spectrum.

Since the frequency values contained in the input data file need not agree with the desired frequencies $f_i = i\Delta f$, `whiten()` must determine the desired values of the real and imaginary parts of the spectrum of the whitening filter by doing an interpolation/extrapolation on the input data. Similar to `noise_power()` (see Sec. 11.4), `whiten()` performs a cubic spline interpolation, using the `spline()` and `splint()` routines from *Numerical Recipes in C*. Like `noise_power()`, `whiten()` assumes that the length of the input data is ≤ 65536 , and it uses boundary conditions for a natural spline. Unlike `noise_power()`, `whiten()` does not have to square the output of the `splint()` routine, since the data contained in the input file and the desired output data both have the same form (i.e., both involve just the real and imaginary parts of $\tilde{W}(f)$).

Authors: Bruce Allen, ballen@dirac.phys.uwm.edu, and Joseph Romano, romano@csd.uwm.edu

Comments: In order for the cubic spline interpolation routines to yield approximations to $\tilde{W}(f)$ that are not contaminated by spurious DC or low frequency (e.g., approximately 1 Hz) components, it is important that the input data file specified by `whiten_file` contain information about the whitening filter down to, and including, zero Hz. This information can be added in “by hand,” for example, if the experimental data for the spectrum of the whitening filter only goes down to 1 Hz. In this case, setting the values of $\tilde{W}(f)$ at 0.0, 0.1, 0.2, \dots , 0.9 Hz equal to their 1 Hz values seems to be sufficient. (See Sec. 11.4 for a similar comment regarding `noise_power()`.)

Also, for the initial and advanced LIGO detector noise models, the spectra $\tilde{W}(f)$ of the whitening filters contained in the input data files were constructed by simply inverting the square roots of the corresponding noise power spectra $P(f)$. The spectra of the whitening filters thus constructed are *real*. Although this method of obtaining information about the spectra of the whitening filters is fine for simulation purposes, the data contained in the actual whitening filter input data files will be obtained *independently* from that contained in the noise power spectra data files, and the spectra $\tilde{W}(f)$ will in general be complex. The function `whiten()` described above—and all other stochastic background routines—allow for this more general form of whitening filter data.

11.6 Function: overlap()

```
void overlap(float *site1_parameters, float *site2_parameters, int n, float
delta_f, double *gamma12)
```

This function calculates the values of the overlap reduction function $\gamma(f)$, which is the averaged product of the response of a pair of detectors to an isotropic and unpolarized stochastic background of gravitational radiation.

The arguments of `overlap()` are:

`site1_parameters`: Input. `site1_parameters[0..8]` is an array of nine floating point variables that define the position of the central station of the first detector site and the orientation of its two arms. The three-vector `site1_parameters[0..2]` are the (x, y, z) components (in cm) of the position vector of the central station of the first site, as measured in a reference frame with the origin at the center of the earth, the z -axis exiting the North pole, and the x -axis passing out the line of 0° longitude. The three-vector `site1_parameters[3..5]` are the (x, y, z) components (in cm) of a vector pointing along the direction of the first arm of the first detector (from the central station to the end station). The three-vector `site1_parameters[6..8]` are the (x, y, z) components (in cm) of a vector pointing along the direction of the second arm of the first detector (from the central station to the end station).

`site2_parameters`: Input. `site2_parameters[0..8]` is an array of nine floating point variables that define the position of the central station of the second detector site and the orientation of its two arms, in exactly the same format as the previous argument.

`n`: Input. The number N of discrete frequency values at which the overlap reduction function $\gamma(f)$ is to be evaluated.

`delta_f`: Input. The spacing Δf (in Hz) between two adjacent discrete frequency values: $\Delta f := f_{i+1} - f_i$.

`gamma12`: Output. `gamma12[0..n-1]` is an array of double precision variables containing the values of the overlap reduction $\gamma(f)$ for the two detector sites. These variables are dimensionless. `gamma12[i]` contains the value of $\gamma(f)$ evaluated at the discrete frequency $f_i = i\Delta f$, where $i = 0, 1, \dots, N - 1$.

The values of $\gamma(f)$ calculated by `overlap()` are defined by equation (3.9) of Ref. [36]:

$$\gamma(f) := \frac{5}{8\pi} \int_{S^2} d\hat{\Omega} e^{2\pi i f \hat{\Omega} \cdot \Delta \vec{x} / c} \left(F_1^+ F_2^+ + F_1^\times F_2^\times \right). \quad (11.6.1)$$

Here $\hat{\Omega}$ is a unit-length vector on the two-sphere, $\Delta \vec{x}$ is the separation vector between the two detector sites, and $F_i^{+, \times}$ is the response of detector i to the $+$ or \times polarization. For the first detector ($i = 1$) one has

$$F_1^{+, \times} = \frac{1}{2} \left(\hat{X}_1^a \hat{X}_1^b - \hat{Y}_1^a \hat{Y}_1^b \right) e_{ab}^{+, \times}(\hat{\Omega}), \quad (11.6.2)$$

where the directions of the first detector's arms are defined by \hat{X}_1^a and \hat{Y}_1^a , and $e_{ab}^{+, \times}(\hat{\Omega})$ are the spin-two polarization tensors for the "plus" and "cross" polarizations, respectively. (A similar expression can be written down for the second detector.) The normalization of $\gamma(f)$ is determined by the following statement: For coincident and coaligned detectors (i.e., for two detectors located at the same place, with both pairs of arms pointing in the same directions), $\gamma(f) = 1$ for all frequencies.

Authors: Bruce Allen, ballen@dirac.phys.uwm.edu, and Joseph Romano, romano@csd.uwm.edu

Comments: None.

11.7 Example: overlap program

The following example program shows one way of combining the functions `detector_site()` and `overlap()` to calculate the overlap reduction function $\gamma(f)$ for a given pair of detectors. In particular, we calculate $\gamma(f)$ for the Hanford, WA and Livingston, LA LIGO detector sites. The resulting overlap reduction function data is stored as two columns of double precision numbers (f_i and $\gamma(f_i)$) in the file `LIGO_overlap.dat`. Here $f_i = i\Delta f$ with $i = 0, 1, \dots, N - 1$. The values of N and Δf are input parameters to the program, which the user can change if he/she desires. (See the `#define` statements listed at the beginning of the program.) Also, by changing the site location identification numbers and the output file name, the user can calculate and save the overlap reduction function for *any* pair of detectors—e.g., the Hanford, WA LIGO detector and the GEO-600 detector; the GEO-600 and VIRGO detector; the Garching and Glasgow detectors; etc. The overlap reduction function data that is stored in the file can then be displayed with `xmgr`, for example. (See Fig. 74.)

```
/* GRASP: Copyright 1997,1998 Bruce Allen */
/* main program to illustrate the function overlap() */

#include "grasp.h"

#define DETECTORS_FILE "detectors.dat" /* file containing detector info */
#define SITE1_CHOICE 1 /* 1=LIGO-Hanford site */
#define SITE2_CHOICE 2 /* 2=LIGO-Livingston site */
#define N 10000 /* number of frequency points */
#define DELTA_F 1.0 /* frequency spacing (in Hz) */
#define OUT_FILE "LIGO_overlap.dat" /* output filename */

int main(int argc, char **argv)
{
    int i;
    double f;

    float site1_parameters[9], site2_parameters[9];
    char site1_name[100], noise1_file[100], whiten1_file[100];
    char site2_name[100], noise2_file[100], whiten2_file[100];

    double *gamma12;

    FILE *fp;
    fp=fopen(OUT_FILE, "w");

    /* ALLOCATE MEMORY */
    gamma12=(double *)malloc(N*sizeof(double));

    /* CALL DETECTOR_SITE() TO GET SITE PARAMETER INFORMATION */
    detector_site(DETECTORS_FILE, SITE1_CHOICE, site1_parameters, site1_name,
                 noise1_file, whiten1_file);
    detector_site(DETECTORS_FILE, SITE2_CHOICE, site2_parameters, site2_name,
                 noise2_file, whiten2_file);

    /* CALL OVERLAP() AND WRITE DATA TO THE FILE */
    overlap(site1_parameters, site2_parameters, N, DELTA_F, gamma12);

    for (i=0; i<N; i++) {
        f=i*DELTA_F;
        fprintf(fp, "%e %e\n", f, gamma12[i]);
    }
}
```

```
fclose(fp);  
return 0;  
}
```

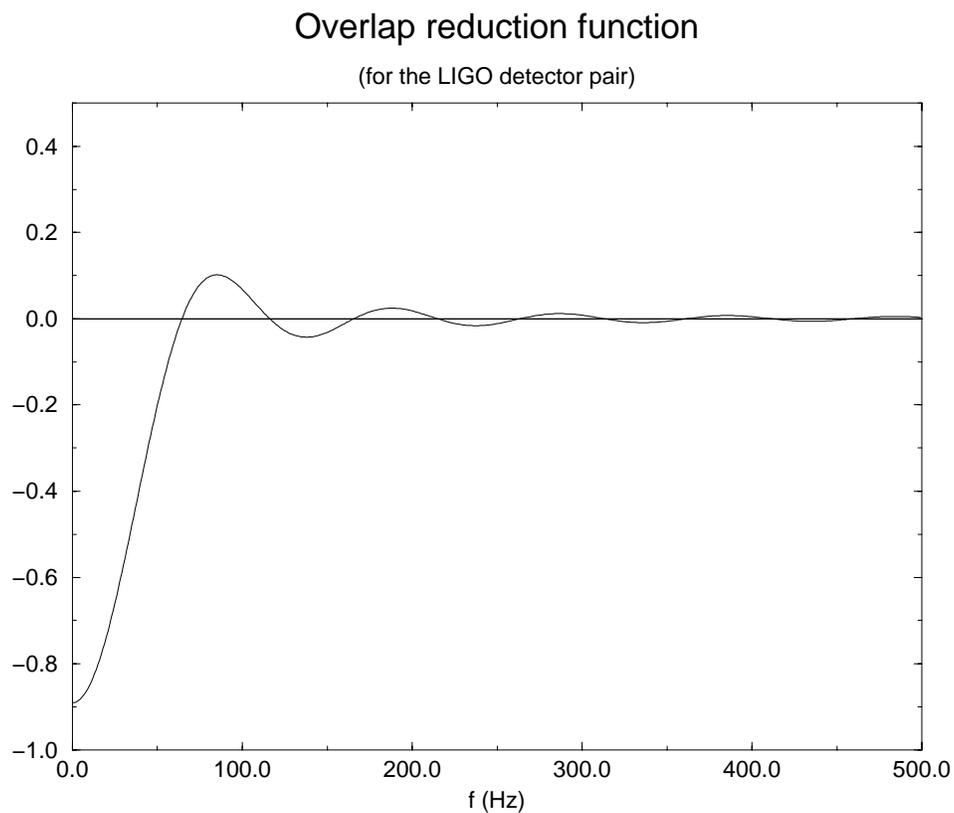


Figure 74: The overlap reduction function $\gamma(f)$ for the Hanford, WA and Livingston, LA LIGO detector pair.

11.8 Function: `get_IFO12()`

```
get_IFO12(FILE *fp1, FILE *fp2, FILE *fp1lock, FILE *fp2lock, int n, float  
*out1, float *out2, float *srate1, float *srate2)
```

This function gets real interferometer output (IFO) data from two detector sites.

The arguments of `get_IFO12()` are:

`fp1`: Input. A pointer to a file that contains the interferometer output (IFO) data produced by the first detector.

`fp2`: Input. A pointer to a file that contains the interferometer output (IFO) data produced by the second detector.

`fp1lock`: Input. A pointer to a file that contains the TTL lock signal for the interferometer output produced by the first detector.

`fp2lock`: Input. A pointer to a file that contains the TTL lock signal for the interferometer output produced by the second detector.

`n`: Input. The number N of data points to be retrieved.

`out1`: Output. `out1[0..n-1]` is an array of floating point variables containing the values of the interferometer output produced by the first detector. These variables have units of ADC counts. `out1[i]` contains the value of the whitened data stream $o_1(t)$ evaluated at the discrete time $t_i = i\Delta t_1$, where $i = 0, 1, \dots, N - 1$ and Δt_1 is the sampling period of the first detector, defined below.

`out2`: Output. `out2[0..n-1]` is an array of floating point variables containing the values of the interferometer output produced by the second detector, in exactly the same format as the previous argument.

`srate1`: Output. The sample rate Δf_1 (in Hz) of the first detector. $\Delta t_1 := 1/\Delta f_1$ (in sec) is the corresponding sampling period of the first detector.

`srate2`: Output. The sample rate Δf_2 (in Hz) of the second detector. $\Delta t_2 := 1/\Delta f_2$ (in sec) is the corresponding sampling period of the second detector.

`get_IFO12()` consists effectively of two calls to `get_data()`, which is described in detail in Sec. 3.6. It prints out a warning message if no data remains for one or both detectors. For that case, both `out1[]` and `out2[]` are set to zero.

Authors: Bruce Allen, ballen@dirac.phys.uwm.edu, and Joseph Romano, romano@csd.uwm.edu

Comments: Currently, `get_IFO12()` calls `get_data()` and `get_data2()`, where `get_data2()` is simply a copy of the `get_data()` routine. `get_data()` should eventually be modified so that it can handle simultaneous requests for data from more than one detector. After this change is made, the function `get_data2()` should be removed.

11.9 Function: simulate_noise()

```
void simulate_noise(int n, float delta_t, double *power, double *whiten_out,
float *out, int *pseed)
```

This function simulates the generation of noise intrinsic to a detector. The output is a (not necessarily continuous-in-time) whitened data stream $o(t)$ representing the detector output when only detector noise is present.

The arguments of `simulate_noise()` are:

n: Input. The number N of data points corresponding to an observation time $T := N \Delta t$, where Δt is the sampling period of the detector, defined below. N should equal an integer power of 2.

delta_t: Input. The sampling period Δt (in sec) of the detector.

power: Input. `power[0..n/2-1]` is an array of double precision variables containing the values of the noise power spectrum $P(f)$ of the detector. These variables have units of $\text{strain}^2/\text{Hz}$ (or seconds). `power[i]` contains the value of $P(f)$ evaluated at the discrete frequency $f_i = i/(N\Delta t)$, where $i = 0, 1, \dots, N/2 - 1$.

whiten_out: Input. `whiten_out[0..n-1]` is an array of double precision variables containing the values of the real and imaginary parts of the spectrum $\tilde{W}(f)$ of the whitening filter of the detector. These variables have units rHz/strain (or $\text{sec}^{-1/2}$), which are inverse to the units of the square root of the noise power spectrum $P(f)$. `whiten_out[2*i]` and `whiten_out[2*i+1]` contain, respectively, the values of the real and imaginary parts of $\tilde{W}(f)$ evaluated at the discrete frequency $f_i = i/(N\Delta t)$, where $i = 0, 1, \dots, N/2 - 1$.

out: Output. `out[0..n-1]` is an array of floating point variables containing the values of the whitened data stream $o(t)$ representing the output of the detector when only detector noise is present. $o(t)$ is the convolution of detector whitening filter $W(t)$ with the noise $n(t)$ intrinsic to the detector. The variables `out[]` have units of rHz (or $\text{sec}^{-1/2}$), which follows from the definition of $n(t)$ as a strain and $\tilde{W}(f)$ as the “inverse” of the square root of the noise power spectrum $P(f)$. `out[i]` contains the value of $o(t)$ evaluated at the discrete time $t_i = i\Delta t$, where $i = 0, 1, \dots, N - 1$.

pseed: Input. A pointer to a seed value, which is used by the random number generator routine.

`simulate_noise()` simulates the generation of noise intrinsic to a detector in the following series of steps:

- (i) It first constructs random variables $\tilde{n}(f_i)$ in the frequency domain that have zero mean and satisfy:

$$\langle \tilde{n}^*(f_i) \tilde{n}(f_j) \rangle = \frac{1}{2} T \delta_{ij} P(f_i), \quad (11.9.1)$$

where $\langle \rangle$ denotes ensemble average. The above equation is just the discrete frequency version of Eq. (11.4.1). This equation can be realized by setting

$$\tilde{n}(f_i) = \frac{1}{2} \sqrt{T} P^{1/2}(f_i) (u_i + iv_i), \quad (11.9.2)$$

where u_i and v_i are statistically independent (real) Gaussian random variables, each having zero mean and unit variance. These Gaussian random variables are produced by calls to the *Numerical Recipes in C* random number generator routine `gasdev()`.

- (ii) `simulate_noise()` then whitens the data in the frequency domain by multiplying $\tilde{n}(f_i)$ by the frequency components $\tilde{W}(f_i)$ of the whitening filter of the detector:

$$\tilde{o}(f_i) := \tilde{n}(f_i) \tilde{W}(f_i). \quad (11.9.3)$$

This (complex) multiplication in the frequency domain corresponds to the convolution of $n(t)$ and $W(t)$ in the time domain. By convention, the DC (i.e., zero frequency) and Nyquist critical frequency components of $\tilde{o}(f_i)$ are set to zero.

- (iii) The final step consists of Fourier transforming the frequency components $\tilde{o}(f_i)$ into the time domain to obtain the whitened data stream $o(t_i)$. Here $t_i = i\Delta t$ with $i = 0, 1, \dots, N - 1$.

Authors: Bruce Allen, ballen@dirac.phys.uwm.edu, and Joseph Romano, romano@csd.uwm.edu

Comments: In the context of stochastic background simulations, it would be more efficient to simulate the noise at *two* detectors simultaneously. Since the time-series data are real, the two Fourier transforms that would need to be performed in step (iii) could be done simultaneously. However, for modularity of design, and to simulate noise for “single-detector” gravity-wave searches, we decided to write the above routine instead.

11.10 Function: simulate_sb()

```
void simulate_sb(int n, float delta_t, float omega_0, float f_low, float f_high,
double *gamma12, double *whiten1, double *whiten2, float *out1, float *out2,
int *pseed)
```

This function simulates the generation of an isotropic and unpolarized stochastic background of gravitational radiation having a constant frequency spectrum: $\Omega_{\text{gw}}(f) = \Omega_0$ for $f_{\text{low}} \leq f \leq f_{\text{high}}$. The outputs are (not necessarily continuous-in-time) whitened data stream $o_1(t)$ and $o_2(t)$ representing the detector outputs when only a stochastic background signal is present.

The arguments of `simulate_sb()` are:

`n`: Input. The number N of data points corresponding to an observation time $T := N \Delta t$, where Δt is the sampling period of the detectors, defined below. N should equal an integer power of 2.

`delta_t`: Input. The sampling period Δt (in sec) of the detectors.

`omega_0`: Input. The constant value Ω_0 (dimensionless) of the frequency spectrum $\Omega_{\text{gw}}(f)$ for the stochastic background:

$$\Omega_{\text{gw}}(f) = \begin{cases} \Omega_0 & f_{\text{low}} \leq f \leq f_{\text{high}} \\ 0 & \text{otherwise.} \end{cases}$$

Ω_0 should be greater than or equal to zero.

`f_low`: Input. The frequency f_{low} (in Hz) below which the spectrum $\Omega_{\text{gw}}(f)$ of the stochastic background is zero. f_{low} should lie in the range $0 \leq f_{\text{low}} \leq f_{\text{Nyquist}}$, where f_{Nyquist} is the Nyquist critical frequency. (The Nyquist critical frequency is defined by $f_{\text{Nyquist}} := 1/(2\Delta t)$, where Δt is the sampling period of the detectors.) f_{low} should also be less than or equal to f_{high} .

`f_high`: Input. The frequency f_{high} (in Hz) above which the spectrum $\Omega_{\text{gw}}(f)$ of the stochastic background is zero. f_{high} should lie in the range $0 \leq f_{\text{high}} \leq f_{\text{Nyquist}}$. It should also be greater than or equal to f_{low} .

`gamma12`: Input. `gamma12[0..n/2-1]` is an array of double precision variables containing the values of the overlap reduction function $\gamma(f)$ for the two detector sites. These variables are dimensionless. `gamma12[i]` contains the value of $\gamma(f)$ evaluated at the discrete frequency $f_i = i/(N\Delta t)$, where $i = 0, 1, \dots, N/2 - 1$.

`whiten1`: Input. `whiten1[0..n-1]` is an array of double precision variables containing the values of the real and imaginary parts of the spectrum $\tilde{W}_1(f)$ of the whitening filter of the first detector. These variables have units rHz/strain (or $\text{sec}^{-1/2}$), which are inverse to the units of the square root of the noise power spectrum $P_1(f)$. `whiten1[2*i]` and `whiten1[2*i+1]` contain, respectively, the values of the real and imaginary parts of $\tilde{W}_1(f)$ evaluated at the discrete frequency $f_i = i/(N\Delta t)$, where $i = 0, 1, \dots, N/2 - 1$.

`whiten2`: Input. `whiten2[0..n-1]` is an array of double precision variables containing the values of the real and imaginary parts of the spectrum $\tilde{W}_2(f)$ of the whitening filter of the second detector, in exactly the same format as the previous argument.

`out1`: Output. `out1[0..n-1]` is an array of floating point variables containing the values of the whitened data stream $o_1(t)$ representing the output of the first detector when only a stochastic background signal is present. $o_1(t)$ is the convolution of detector whitening filter $W_1(t)$ with the gravitational strain $h_1(t)$. The variables `out1[]` have units of rHz (or $\text{sec}^{-1/2}$), which follows from the

definition of $h_1(t)$ as a strain and $\tilde{W}_1(f)$ as the “inverse” of the square root of the noise power spectrum $P_1(f)$. `out1[i]` contains the value of $o_1(t)$ evaluated at the discrete time $t_i = i\Delta t$, where $i = 0, 1, \dots, N - 1$.

`out2`: Output. `out2[0..n-1]` is an array of floating point variables containing the values of the whitened data stream $o_2(t)$ representing the output of the second detector when only a stochastic background signal is present, in exactly the same format as the previous argument.

`pseed`: Input. A pointer to a seed value, which is used by the random number generator routine.

`simulate_sb()` simulates the generation of an isotropic and unpolarized stochastic background of gravitational radiation having a constant frequency spectrum $\Omega_{\text{gw}}(f) = \Omega_0$ for $f_{\text{low}} \leq f \leq f_{\text{high}}$ in the following series of steps:

- (i) It first constructs random variables $\tilde{h}_1(f_i)$ and $\tilde{h}_2(f_i)$ in the frequency domain that have zero mean and satisfy:

$$\langle \tilde{h}_1^*(f_i) \tilde{h}_1(f_j) \rangle = \frac{1}{2} T \delta_{ij} \frac{3H_0^2}{10\pi^2} f_i^{-3} \Omega_0 \quad (11.10.1)$$

$$\langle \tilde{h}_2^*(f_i) \tilde{h}_2(f_j) \rangle = \frac{1}{2} T \delta_{ij} \frac{3H_0^2}{10\pi^2} f_i^{-3} \Omega_0 \quad (11.10.2)$$

$$\langle \tilde{h}_1^*(f_i) \tilde{h}_2(f_j) \rangle = \frac{1}{2} T \delta_{ij} \frac{3H_0^2}{10\pi^2} f_i^{-3} \Omega_0 \gamma(f_i), \quad (11.10.3)$$

where $\langle \rangle$ denotes ensemble average. Here $\tilde{h}_1(f_i)$ and $\tilde{h}_2(f_i)$ are the Fourier components of the gravitational strains $h_1(t)$ and $h_2(t)$ at the two detectors. The above equations are the discrete frequency versions of equation (3.17) of Ref. [36], with $\Omega_{\text{gw}}(f) = \Omega_0$ for $f_{\text{low}} \leq f \leq f_{\text{high}}$. They can be realized by setting

$$\tilde{h}_1(f_i) = \frac{1}{2} \sqrt{T} \left(\frac{3H_0^2}{10\pi^2} \right)^{1/2} f_i^{-3/2} \Omega_0^{1/2} (x_{1i} + iy_{1i}) \quad (11.10.4)$$

$$\tilde{h}_2(f_i) = \tilde{h}_1(f_i) \gamma(f_i) + \quad (11.10.5)$$

$$\frac{1}{2} \sqrt{T} \left(\frac{3H_0^2}{10\pi^2} \right)^{1/2} f_i^{-3/2} \Omega_0^{1/2} \sqrt{1 - \gamma^2(f_i)} (x_{2i} + iy_{2i}), \quad (11.10.6)$$

where x_{1i}, y_{1i}, x_{2i} , and y_{2i} are statistically independent (real) Gaussian random variables, each having zero mean and unit variance. (Note: The x_{1i}, y_{1i}, x_{2i} , and y_{2i} random variables are statistically independent of the u_i and v_i random variables defined in Sec. 11.9.) These Gaussian random variables are produced by calls to the *Numerical Recipes in C* random number generator routine `gasdev()`. Note also that the second term of $\tilde{h}_2(f_i)$ (which is proportional to $\sqrt{1 - \gamma^2(f_i)}$) is needed to obtain equation (11.10.2). Without this term, $\langle \tilde{h}_2^*(f_i) \tilde{h}_2(f_j) \rangle$ would include an additional (unwanted) factor of $\gamma^2(f_i)$.

- (iii) `simulate_sb()` then whitens the data in the frequency domain by multiplying $\tilde{h}_1(f_i)$ and $\tilde{h}_2(f_i)$ by the frequency components $\tilde{W}_1(f_i)$ and $\tilde{W}_2(f_i)$ of the whitening filters of the two detectors:

$$\tilde{o}_1(f_i) := \tilde{h}_1(f_i) \tilde{W}_1(f_i) \quad (11.10.7)$$

$$\tilde{o}_2(f_i) := \tilde{h}_2(f_i) \tilde{W}_2(f_i). \quad (11.10.8)$$

This (complex) multiplication in the frequency domain corresponds to the convolution of $h_1(t)$ and $W_1(t)$, and $h_2(t)$ and $W_2(t)$ in the time domain. By convention, the DC (i.e., zero frequency) and Nyquist critical frequency components of $\tilde{o}_1(f_i)$ and $\tilde{o}_2(f_i)$ are set to zero.

(iii) The final step consists of Fourier transforming the frequency components $\tilde{o}_1(f_i)$ and $\tilde{o}_2(f_i)$ into the time domain to obtain the whitened data streams $o_1(t_i)$ and $o_2(t_i)$. Here $t_i = i\Delta t$ with $i = 0, 1, \dots, N - 1$. Since $\tilde{o}_1^*(f_i)$ and $\tilde{o}_2^*(f_i)$ are the Fourier transforms of real data sets, the two Fourier transforms can be performed simultaneously.

Authors: Bruce Allen, ballen@dirac.phys.uwm.edu, and Joseph Romano, romano@csd.uwm.edu

Comments: Although it is possible and more efficient to write a single function to simulate the generation of a stochastic background and intrinsic detector noise simultaneously, we have chosen—for the sake of modularity—to write separate functions to perform these two tasks separately. (See also the comment at the end of Sec. 11.9.)

11.11 Function: combine_data()

void combine_data(int which, int n, float *in1, float *in2, float *out)
This low-level function takes two arrays as input, shifts them by half their length, and combines them with one another and with data stored in an internally-defined static buffer to produce output data that is continuous from one call of combine_data() to the next.

The arguments of combine_data() are:

which: Input. An integer variable specifying which internally-defined static buffer should be used when combining the input arrays with data saved from a previous call. The allowed values are $1 \leq \text{which} \leq 16$.

n: Input. The number N of data points contained in the input and output arrays. N is assumed to be even.

in1: Input. in1[0..n-1] is an array of floating point variables containing the values of the first input array.

in2: Input. in2[0..n-1] is an array of floating point variables containing the values of the second input array.

out: Output. out[0..n-1] is an array of floating point variables containing the output data, which is continuous from one call of combine_data() to the next.

combine_data() produces continuous output data by modifying the appropriately chosen static buffer buf[0..3*n/2-1] as follows:

$$\begin{aligned} \text{buf}[i] + &= \sin[i * \text{M_PI}/n] * \text{in1}[i] && \text{for } 0 \leq i \leq n/2 - 1 \\ \text{buf}[i] + &= \sin[i * \text{M_PI}/n] * \text{in1}[i] + \sin[(i - n/2) * \text{M_PI}/n] * \text{in2}[i - n/2] && \text{for } n/2 \leq i \leq n - 1 \\ \text{buf}[i] + &= \sin[(i - n/2) * \text{M_PI}/n] * \text{in2}[i - n/2] && \text{for } n \leq i \leq 3 * n/2 - 1. \end{aligned}$$

The values of the output array out[0..n-1] are taken from the first two-thirds of the buffer, while the last one-third of the buffer is copied to the first third of the buffer in preparation for the next call. When this is complete, the last two-thirds of the buffer is cleared.

One nice feature of combining the data with a sine function (rather than with a triangle function, for example) is that if the input data represent statistically independent, stationary random processes having zero mean and the same variance, then the output data will also have zero mean and the same variance. This is a consequence of the trigonometric identity

$$\sin^2[i * \text{M_PI}/n] + \sin^2[(i - n/2) * \text{M_PI}/n] = 1. \quad (11.11.1)$$

Thus, combine_data() preserves the first and second-order statistical properties of the input data when constructing the output.

Authors: Bruce Allen, ballen@dirac.phys.uwm.edu, and Joseph Romano, romano@csd.uwm.edu

Comments: In the context of stochastic background simulations, the two input arrays would represent two whitened data streams produced by a single detector, which are then time-shifted and combined to simulate *continuous-in-time* detector output. Since the contents of the internally-defined static buffer are equal to zero when combine_data() is first called, the amplitude of the output array initially builds up from zero to its nominal value over the course of the first $N/2$ data points. This corresponds to an effective “turn-on” transient, with turn-on time equal to $N \Delta t/2$ (Δt being the time between successive data samples).

11.12 Function: monte_carlo()

```
void monte_carlo(int fake_sb, int fake_noise1, int fake_noise2, int n, float
delta_t, float omega_0, float f_low, float f_high, double *gamma12, double
*power1, double *power2, double *whiten1, double *whiten2, float *out1, float
*out2, int *pseed)
```

This high-level function simulates (if desired) the generation of noise intrinsic to a pair of detectors, and an isotropic and unpolarized stochastic background of gravitational radiation having a constant frequency spectrum: $\Omega_{\text{gw}}(f) = \Omega_0$ for $f_{\text{low}} \leq f \leq f_{\text{high}}$. The outputs are two continuous-in-time whitened data streams $o_1(t)$ and $o_2(t)$ representing the detector outputs in the presence of a stochastic background signal plus noise.

The arguments of monte_carlo() are:

fake_sb: Input. An integer variable that should be set equal to 1 if a simulated stochastic background is desired.

fake_noise1: Input. An integer variable that should be set equal to 1 if simulated detector noise for the first detector is desired.

fake_noise2: Input. An integer variable that should be set equal to 1 if simulated detector noise for the second detector is desired.

n: Input. The number N of data points corresponding to an observation time $T := N \Delta t$, where Δt is the sampling period of the detector, defined below. N should equal an integer power of 2.

delta_t: Input. The sampling period Δt (in sec) of the detector.

omega_0: Input. The constant value Ω_0 (dimensionless) of the frequency spectrum $\Omega_{\text{gw}}(f)$ for the stochastic background:

$$\Omega_{\text{gw}}(f) = \begin{cases} \Omega_0 & f_{\text{low}} \leq f \leq f_{\text{high}} \\ 0 & \text{otherwise.} \end{cases}$$

Ω_0 should be greater than or equal to zero.

f_low: Input. The frequency f_{low} (in Hz) below which the spectrum $\Omega_{\text{gw}}(f)$ of the stochastic background is zero. f_{low} should lie in the range $0 \leq f_{\text{low}} \leq f_{\text{Nyquist}}$, where f_{Nyquist} is the Nyquist critical frequency. (The Nyquist critical frequency is defined by $f_{\text{Nyquist}} := 1/(2\Delta t)$, where Δt is the sampling period of the detector.) f_{low} should also be less than or equal to f_{high} .

f_high: Input. The frequency f_{high} (in Hz) above which the spectrum $\Omega_{\text{gw}}(f)$ of the stochastic background is zero. f_{high} should lie in the range $0 \leq f_{\text{high}} \leq f_{\text{Nyquist}}$. It should also be greater than or equal to f_{low} .

gamma12: Input. gamma12[0..n/2-1] is an array of double precision variables containing the values of the overlap reduction function $\gamma(f)$ for the two detector sites. These variables are dimensionless. gamma12[i] contains the value of $\gamma(f)$ evaluated at the discrete frequency $f_i = i/(N\Delta t)$, where $i = 0, 1, \dots, N/2 - 1$.

power1: Input. power1[0..n/2-1] is an array of double precision variables containing the values of the noise power spectrum $P_1(f)$ of the first detector. These variables have units of strain²/Hz (or seconds). power1[i] contains the value of $P_1(f)$ evaluated at the discrete frequency $f_i = i/(N\Delta t)$, where $i = 0, 1, \dots, N/2 - 1$.

`power2`: Input. `power2[0..n/2-1]` is an array of double precision variables containing the values of the noise power spectrum $P_2(f)$ of the second detector, in exactly the same format as the previous argument.

`whiten1`: Input. `whiten1[0..n-1]` is an array of double precision variables containing the values of the real and imaginary parts of the spectrum $\tilde{W}_1(f)$ of the whitening filter of the first detector. These variables have units rHz/strain (or $\text{sec}^{-1/2}$), which are inverse to the units of the square root of the noise power spectrum $P_1(f)$. `whiten1[2*i]` and `whiten1[2*i+1]` contain, respectively, the values of the real and imaginary parts of $\tilde{W}_1(f)$ evaluated at the discrete frequency $f_i = i/(N\Delta t)$, where $i = 0, 1, \dots, N/2 - 1$.

`whiten2`: Input. `whiten2[0..n-1]` is an array of double precision variables containing the values of the real and imaginary parts of the spectrum $\tilde{W}_2(f)$ of the whitening filter of the second detector, in exactly the same format as the previous argument.

`out1`: Output. `out1[0..n-1]` is an array of floating point variables containing the values of the continuous-in-time whitened data stream $o_1(t)$ representing the output of the first detector. $o_1(t)$ is the convolution of detector whitening filter $W_1(t)$ with the data stream $s_1(t) := h_1(t) + n_1(t)$, where $h_1(t)$ is the gravitational strain and $n_1(t)$ is the noise intrinsic to the detector. These variables have units of rHz (or $\text{sec}^{-1/2}$), which follows from the definition of $s_1(t)$ as a strain and $\tilde{W}_1(f)$ as the “inverse” of the square root of the noise power spectrum $P_1(f)$. `out1[i]` contains the value of $o_1(t)$ evaluated at the discrete time $t_i = i\Delta t$, where $i = 0, 1, \dots, N - 1$.

`out2`: Output. `out2[0..n-1]` is an array of floating point variables containing the values of the continuous-in-time whitened data stream $o_2(t)$ representing the output of the second detector, in exactly the same format as the previous argument.

`pseed`: Input. A pointer to a seed value, which is used by the random number generator routine.

`monte_carlo()` is a very simple function, consisting of calls to `simulate_sb()`, `simulate_noise()`, and `combine_data()`. If `fake_sb=1`, `monte_carlo()` calls `simulate_sb()` twice, producing two sets of data that are time-shifted and combined by `combine_data()` to simulate continuous-in-time detector output. Similar statements apply when either `fake_noise1` or `fake_noise2` equals 1.

Authors: Bruce Allen, ballen@dirac.phys.uwm.edu, and Joseph Romano, romano@csd.uwm.edu

Comments: None.

11.13 Example: monte_carlo program

The following example program is a simple demonstration of the function `monte_carlo()`, which was defined in the previous section. It produces animated output representing time-series data for simulated detector noise and for a simulated stochastic background having a constant frequency spectrum: $\Omega_{\text{gw}}(f) = \Omega_0$ for $f_{\text{low}} \leq f \leq f_{\text{high}}$. The output from this program must be piped into `xmgr`. The parameters that were chosen for the example program shown below produce whitened time-series data for a stochastic background having $\Omega_{\text{gw}}(f) = 1.0 \times 10^{-3}$ for $5 \text{ Hz} \leq f \leq 5000 \text{ Hz}$. For this particular example, the noise intrinsic to the detectors was set to zero. A sample “snapshot” of the animation is shown in Fig. 75.

By modifying the parameters listed at the top of the example program, one can also simulate an unwhitened stochastic background signal (Fig. 76), and whitened and unwhitened data streams corresponding to the noise intrinsic to an initial LIGO detector (Figs. 77 and 78). Other combinations of signal, noise, whitening, and unwhitening are of course also possible. To produce the animated output, simply enter the command:

```
monte_carlo | xmgr -pipe &
```

after compilation.

Note: The amplitude of the animated output initially builds up from zero to its nominal value over (approximately) the first 1.5 seconds. This “turn-on” transient is a consequence of the overlapping technique used by `combine_data()` to produce continuous-in-time detector output. (See the comment at the end of Section 11.11.)

```
/* GRASP: Copyright 1997,1998 Bruce Allen */
/* main program to illustrate monte_carlo() */

#include "grasp.h"
void graphout(float,float,int);

#define DETECTORS_FILE "detectors.dat" /* file containing detector info */
#define SITE1_CHOICE 1 /* identification number for site 1 */
#define SITE2_CHOICE 2 /* identification number for site 2 */
#define FAKE_SB 1 /* 1: simulate stochastic background */
/* 0: no stochastic background */
#define FAKE_NOISE1 0 /* 1: simulate detector noise at site 1 */
/* 0: no detector noise at site 1 */
#define FAKE_NOISE2 0 /* 1: simulate detector noise at site 2 */
/* 0: no detector noise at site 2 */
#define WHITEN_OUT1 1 /* 1: whiten output at site 1 */
/* 0: don't whiten output at site 1 */
#define WHITEN_OUT2 1 /* 1: whiten output at site 2 */
/* 0: don't whiten output at site 2 */

#define N 65536 /* number of data points */
#define DELTA_T (5.0e-5) /* sampling period (in sec) */
#define OMEGA_0 (1.0e-3) /* omega_0 */
#define F_LOW (5.0) /* minimum frequency (in Hz) */
#define F_HIGH (5.0e3) /* maximum frequency (in Hz) */
#define NUM_RUNS 5 /* number of runs */

int main(int argc,char **argv)
{
    int i,j,last=0,seed= -17;
    float delta_f,tstart=0.0,time_now;

    float site1_parameters[9],site2_parameters[9];
```

```
char site1_name[100],noise1_file[100],whiten1_file[100];
char site2_name[100],noise2_file[100],whiten2_file[100];

double *power1,*power2,*whiten1,*whiten2,*gamma12;
float *out1,*out2;

/* ALLOCATE MEMORY */
power1=(double *)malloc((N/2)*sizeof(double));
power2=(double *)malloc((N/2)*sizeof(double));
whiten1=(double *)malloc(N*sizeof(double));
whiten2=(double *)malloc(N*sizeof(double));
gamma12=(double *)malloc((N/2)*sizeof(double));
out1=(float *)malloc(N*sizeof(float));
out2=(float *)malloc(N*sizeof(float));

/* IDENTITY WHITENING FILTERS (IF WHITEN_OUT1=WHITEN_OUT2=0) */
for (i=0;i<N/2;i++) {
    whiten1[2*i]=whiten2[2*i]=1.0;
    whiten1[2*i+1]=whiten2[2*i+1]=0.0;
}

/* CALL DETECTOR_SITE() TO GET SITE PARAMETER INFORMATION */
detector_site(DETECTORS_FILE,SITE1_CHOICE,site1_parameters,site1_name,
             noise1_file,whiten1_file);
detector_site(DETECTORS_FILE,SITE2_CHOICE,site2_parameters,site2_name,
             noise2_file,whiten2_file);

/* CONSTRUCT NOISE POWER SPECTRA, OVERLAP REDUCTION FUNCTION, AND */
/* (NON-TRIVIAL) WHITENING FILTERS, IF DESIRED */
delta_f=(float)(1.0/(N*DELTA_T));
noise_power(noise1_file,N/2,delta_f,power1);
noise_power(noise2_file,N/2,delta_f,power2);
overlap(site1_parameters,site2_parameters,N/2,delta_f,gamma12);
if (WHITEN_OUT1==1) whiten(whiten1_file,N/2,delta_f,whiten1);
if (WHITEN_OUT2==1) whiten(whiten2_file,N/2,delta_f,whiten2);

/* SIMULATE STOCHASTIC BACKGROUND AND/OR DETECTOR NOISE */
for (j=0;j<NUM_RUNS;j++) {
    monte_carlo(FAKE_SB,FAKE_NOISE1,FAKE_NOISE2,N,DELTA_T,OMEGA_0,F_LOW,F_HIGH,
              gamma12,power1,power2,whiten1,whiten2,out1,out2,&seed);

    /* DISPLAY OUTPUT USING XMGR */
    for (i=0;i<N;i++) {
        time_now=tstart+i*DELTA_T;
        printf("%e\t%e\n",time_now,out1[i]);
    }
    if (j==NUM_RUNS-1) last=1;
    graphout(tstart,tstart+N*DELTA_T,last);

    /* UPDATE TSTART */
    tstart+=N*DELTA_T;
} /* end for (j=0;j<NUM_RUNS;j++) */

return 0;
}

void graphout(float xmin,float xmax,int last)
```

```
{
static int first=1;
printf("&\n");

if (first) {

    /* first time we draw plot */
    printf("@doublebuffer true\n"); /* keep display from flashing */
    printf("@focus off\n");
    printf("@world xmin %e\n",xmin);
    printf("@world xmax %e\n",xmax);
    printf("@autoscale yaxes\n");
    printf("@xaxis label \"t (sec)\"\n");
    printf("@title \"Simulated Detector Ouput\"\n");
    printf("@subtitle \"(stochastic background--whitened)\"\n");
    printf("@redraw \n");
    if (!last) printf("@kill s0\n"); /* kill set; ready to read again */

    first=0;
}
else {

    /* other timeOAs we draw plot */
    printf("@world xmin %e\n",xmin);
    printf("@world xmax %e\n",xmax);
    printf("@autoscale yaxes\n");
    if (!last) printf("@kill s0\n"); /* kill set; ready to read again */

}

return;
}
```

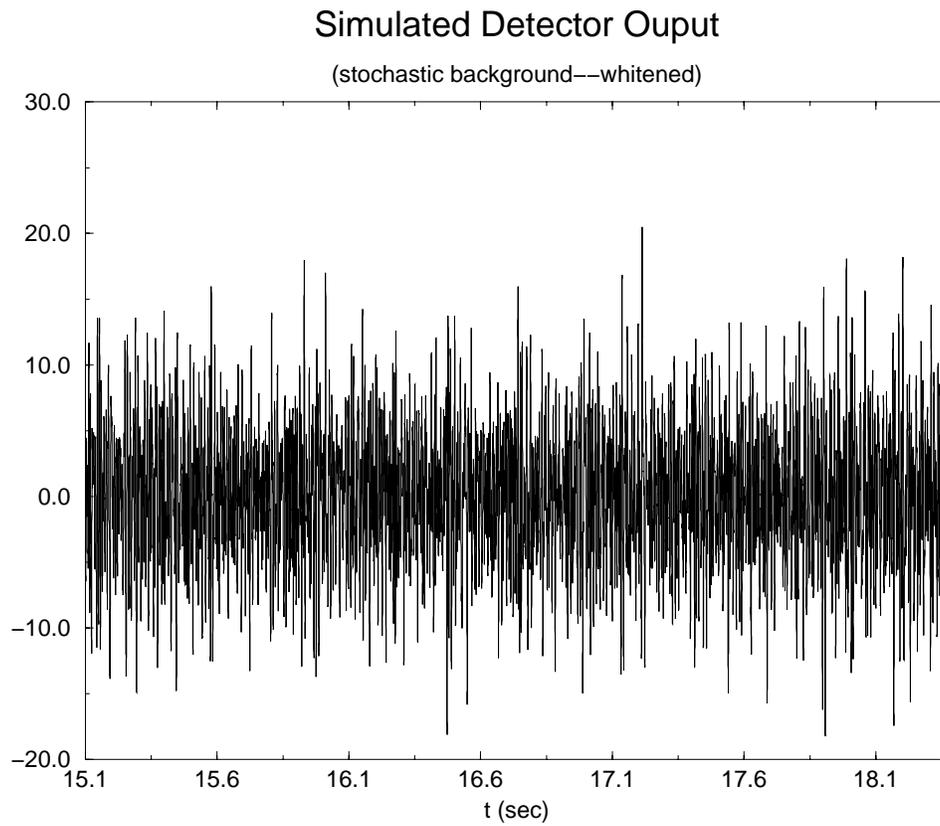


Figure 75: Time-series data (whitened) for a stochastic background having a constant frequency spectrum: $\Omega_{\text{gw}}(f) = 1.0 \times 10^{-3}$ for $5 \text{ Hz} \leq f \leq 5000 \text{ Hz}$.

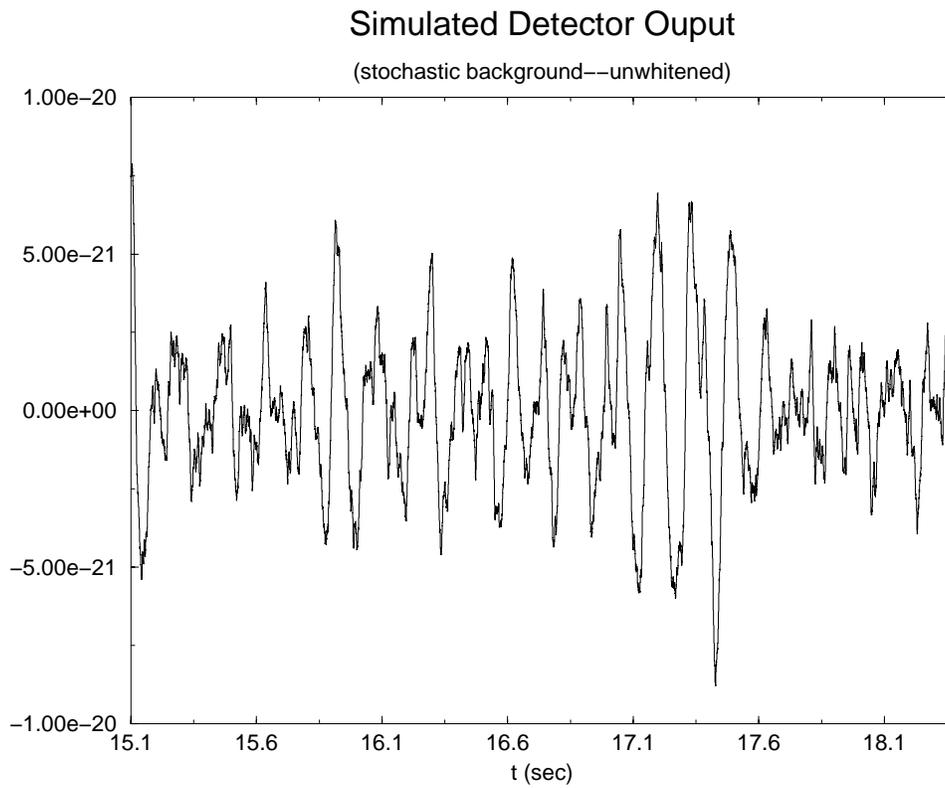


Figure 76: Time-series data (unwhitened) for a stochastic background having a constant frequency spectrum: $\Omega_{\text{gw}}(f) = 1.0 \times 10^{-3}$ for $5 \text{ Hz} \leq f \leq 5000 \text{ Hz}$.

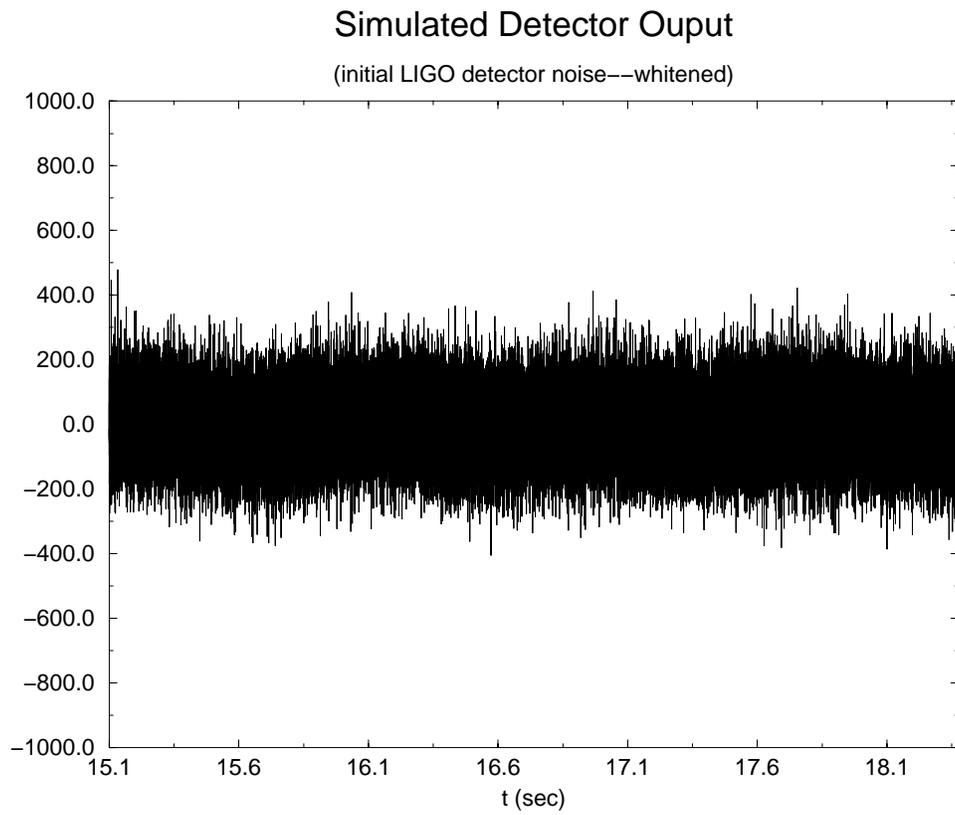


Figure 77: Time-series data (whitened) for the noise intrinsic to an initial LIGO detector.

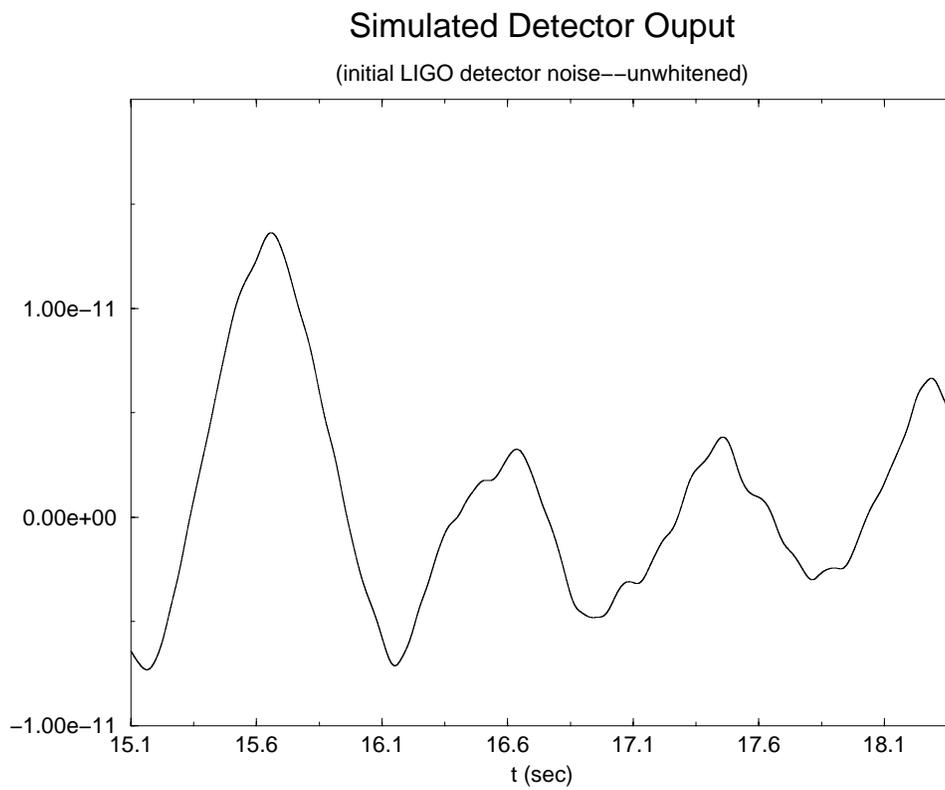


Figure 78: Time-series data (unwhitened) for the noise intrinsic to an initial LIGO detector.

11.14 Function: test_data12()

```
int test_data12(int n, float *data1, float *data2)
```

This function tests two data sets to see if they have probability distributions consistent with a Gaussian normal distribution.

The arguments of test_data12() are:

n: Input. The number N of data points contained in each of the input arrays.

data1: Input. data1[0..n-1] is an array of floating point variables containing the values of the first array to be tested.

data2: Input. data2[0..n-1] is an array of floating point variables containing the values of the second array to be tested.

test_data12() is a simple function that makes use of the is_gaussian() utility routine. (See Sec. 16.5 for more details.) test_data12() prints a warning message if either of the data sets contain a value too large to be stored in 16 bits. (The actual maximum value was chosen to be 32765.) It returns 1 if both data sets pass the is_gaussian() test. It returns 0 if either data set fails, and prints a message indicating the bad set.

Authors: Bruce Allen, ballen@dirac.phys.uwm.edu, and Joseph Romano, romano@csd.uwm.edu

Comments: In the context of stochastic background simulations, data1[] and data2[] contain the values of the whitened data streams $o_1(t)$ and $o_2(t)$ that are output by the two detectors.

11.15 Function: `extract_noise()`

```
void extract_noise(int average, int which, float *in, int n, float delta_t,
double *whiten_out, double *power)
```

This function calculates the real-time noise power spectrum $P(f)$ of a detector, using a Hann window and averaging the spectrum for two overlapped data sets, if desired.

The arguments of `extract_noise()` are:

average: Input. An integer variable that should be set equal to 1 if the values of the real-time noise power spectra corresponding to two overlapped data sets are to be averaged.

which: Input. An integer variable specifying which internally-defined static buffer should be used when overlapping the new input data set with data saved from a previous call. The allowed values are $1 \leq \text{which} \leq 16$.

in: Input. `in[0..n-1]` is an array of floating point variables containing the values of the assumed continuous-in-time whitened data stream $o(t)$ produced by the detector. $o(t)$ is the convolution of detector whitening filter $\tilde{W}(t)$ with the data stream $s(t) := h(t) + n(t)$, where $h(t)$ is the gravitational strain and $n(t)$ is the noise intrinsic to the detector. The variables `in[]` have units of rHz (or $\text{sec}^{-1/2}$), which follows from the definition of $s(t)$ as a strain and $\tilde{W}(f)$ as the “inverse” of the square root of the noise power spectrum $P(f)$. `in[i]` contains the value of $o(t)$ evaluated at the discrete time $t_i = i\Delta t$, where $i = 0, 1, \dots, N - 1$.

n: Input. The number N of data points corresponding to an observation time $T := N \Delta t$, where Δt is the sampling period of the detector, defined below. N should equal an integer power of 2.

delta_t: Input. The sampling period Δt (in sec) of the detector.

whiten_out: Input. `whiten_out[0..n-1]` is an array of double precision variables containing the values of the real and imaginary parts of the spectrum $\tilde{W}(f)$ of the whitening filter of the detector. These variables have units rHz/strain (or $\text{sec}^{-1/2}$), which are inverse to the units of the square root of the noise power spectrum $P(f)$. `whiten_out[2*i]` and `whiten_out[2*i+1]` contain, respectively, the values of the real and imaginary parts of $\tilde{W}(f)$ evaluated at the discrete frequency $f_i = i/(N\Delta t)$, where $i = 0, 1, \dots, N/2 - 1$.

power: Output. `power[0..n/2-1]` is an array of double precision variables containing the values of the real-time noise power spectrum $P(f)$ of the detector. Explicitly,

$$P(f) := \frac{2}{T} \tilde{s}^*(f) \tilde{s}(f), \quad (11.15.1)$$

where $\tilde{s}(f)$ is the Fourier transform of the unwhitened data stream $s(t)$ produced by the detector. These variables have units of $\text{strain}^2/\text{Hz}$ (or seconds). `power[i]` contains the value of $P(f)$ evaluated at the discrete frequency $f_i = i/(N\Delta t)$, where $i = 0, 1, \dots, N/2 - 1$.

`extract_noise()` calculates the real-time noise power spectrum $P(f)$ as follows:

- (i) It first stores the input data stream $o(t)$ in the last two-thirds of an appropriately chosen static buffer `buf[0..3*n/2-1]`. The first one-third of this buffer contains the input data left over from the previous call.

(ii) It then multiplies the first two-thirds of this buffer by the Hann window function:

$$w(t) := \sqrt{\frac{8}{3}} \cdot \frac{1}{2} \left[1 - \cos\left(\frac{2\pi t}{T}\right) \right]. \quad (11.15.2)$$

The factor $\sqrt{8/3}$ is the “window squared-and-summed” factor described in *Numerical Recipes in C*, p.553. It is needed to offset the reduction in power that is introduced by the windowing.

(iii) The windowed data is then Fourier transformed into the frequency domain, where it is unwhitened by dividing by the (complex) spectrum $\tilde{W}(f)$ of the whitening filter of the detector. The resulting unwhitened frequency components are denoted by ${}^{(1)}\tilde{s}(f)$; the superscript (1) indicates that we are analyzing the first of two overlapped data sets.

(iv) The real-time noise power spectrum is then calculated according to:

$${}^{(1)}P(f) := \frac{2}{T} {}^{(1)}\tilde{s}^*(f) {}^{(1)}\tilde{s}(f). \quad (11.15.3)$$

(v) The data contained in the last two-thirds of the buffer is then copied to the first two-thirds of the buffer, and steps (ii)-(iv) are repeated, yielding a second real-time noise power spectrum ${}^{(2)}P(f)$.

(vi) If `average=1`, $P(f)$ is given by:

$$P(f) := \frac{1}{2} \left[{}^{(1)}P(f) + {}^{(2)}P(f) \right]. \quad (11.15.4)$$

Otherwise, $P(f) = {}^{(2)}P(f)$.

(vii) Finally, the data contained in the last two-thirds of the buffer is again copied to the first two-thirds, in preparation for the next call to `extract_noise()`. The data saved in the first one-third of this buffer will match onto the next input data stream if the input data from one call of `extract_noise()` to the next is continuous.

Note: One should call `extract_noise()` with `average ≠ 1`, when one suspects that the current input data is *not* continuous with the data that was saved from the previous call. This is because a discontinuity between the “old” and “new” data sets has a tendency to introduce spurious large frequency components into the real-time noise power spectrum, which should not be present. Since a single input data stream by itself is continuous, the noise power spectrum ${}^{(2)}P(f)$ (which is calculated on the second pass through the data) will be free of these spurious large frequency components. This is why we set $P(f)$ equal to ${}^{(2)}P(f)$ —and not equal to ${}^{(1)}P(f)$ —when `average ≠ 1`.

Authors: Bruce Allen, ballen@dirac.phys.uwm.edu, and Joseph Romano, romano@csd.uwm.edu

Comments: In the context of stochastic background simulations, it would be more efficient to extract the real-time noise power spectra at *two* detectors simultaneously. However, for modularity of design, and to allow this function to be used possibly for “single-detector” gravity-wave searches, we decided to write the above routine instead.

11.16 Function: extract_signal()

```
void extract_signal(int average, float *in1, float *in2, int n, float delta_t,
double *whiten1, double *whiten2, double *signal12)
```

This function calculates the real-time cross-correlation spectrum $\tilde{s}_{12}(f)$ of the unwhitened data streams $s_1(t)$ and $s_2(t)$, using a Hann window and averaging the spectrum for two overlapped data sets, if desired.

The arguments of `extract_signal()` are:

average: Input. An integer variable that should be set equal to 1 if the values of the real-time cross-correlation spectra corresponding to two overlapped data sets are to be averaged.

in1: Input. `in1[0..n-1]` is an array of floating point variables containing the values of the assumed continuous-in-time whitened data stream $o_1(t)$ produced by the first detector. $o_1(t)$ is the convolution of detector whitening filter $W_1(t)$ with the data stream $s_1(t) := h_1(t) + n_1(t)$, where $h_1(t)$ is the gravitational strain and $n_1(t)$ is the noise intrinsic to the detector. The variables `in1[]` have units of rHz (or $\text{sec}^{-1/2}$), which follows from the definition of $s_1(t)$ as a strain and $\tilde{W}_1(f)$ as the “inverse” of the square root of the noise power spectrum $P_1(f)$. `in1[i]` contains the value of $o_1(t)$ evaluated at the discrete time $t_i = i\Delta t$, where $i = 0, 1, \dots, N - 1$.

in2: Input. `in2[0..n-1]` is an array of floating point variables containing the values of the assumed continuous-in-time whitened data stream $o_2(t)$ produced by the second detector, in exactly the same format as the previous argument.

n: Input. The number N of data points corresponding to an observation time $T := N \Delta t$, where Δt is the sampling period of the detectors, defined below. N should equal an integer power of 2.

delta_t: Input. The sampling period Δt (in sec) of the detectors.

whiten1: Input. `whiten1[0..n-1]` is an array of double precision variables containing the values of the real and imaginary parts of the spectrum $\tilde{W}_1(f)$ of the whitening filter of the first detector. These variables have units rHz/strain (or $\text{sec}^{-1/2}$), which are inverse to the units of the square root of the noise power spectrum $P_1(f)$. `whiten1[2*i]` and `whiten1[2*i+1]` contain, respectively, the values of the real and imaginary parts of $\tilde{W}_1(f)$ evaluated at the discrete frequency $f_i = i/(N\Delta t)$, where $i = 0, 1, \dots, N/2 - 1$.

whiten2: Input. `whiten2[0..n-1]` is an array of double precision variables containing the values of the real and imaginary parts of the spectrum $\tilde{W}_2(f)$ of the whitening filter of the second detector, in exactly the same format as the previous argument.

signal12: Output. `signal12[0..n/2-1]` is an array of double precision variables containing the values of the real-time cross-correlation spectrum

$$\tilde{s}_{12}(f) := (\tilde{s}_1^*(f) \tilde{s}_2(f) + \text{c.c.}) , \quad (11.16.1)$$

where $\tilde{s}_1(f)$ and $\tilde{s}_2(f)$ are the Fourier transforms of the unwhitened data streams $s_1(t)$ and $s_2(t)$ produced by the two detectors. These variables have units of $\text{strain}^2 \cdot \text{sec}^2$ (or simply sec^2). `signal12[i]` contains the value of $\tilde{s}_{12}(f)$ evaluated at the discrete frequency $f_i = i/(N\Delta t)$, where $i = 0, 1, \dots, N/2 - 1$.

`extract_signal()` calculates the real-time cross-correlation spectrum $\tilde{s}_{12}(f)$ as follows:

- (i) It first stores the input data streams $o_1(t)$ and $o_2(t)$ in the last two-thirds of internally-defined static buffers `buf1[0..3*n/2-1]` and `buf2[0..3*n/2-1]`. The first one-third of these buffers contains the input data left over from the previous call.
- (ii) It then multiplies the first two-thirds of these buffers by the Hann window function:

$$w(t) := \sqrt{\frac{8}{3}} \cdot \frac{1}{2} \left[1 - \cos\left(\frac{2\pi t}{T}\right) \right]. \quad (11.16.2)$$

The factor $\sqrt{8/3}$ is the “window squared-and-summed” factor described in *Numerical Recipes in C*, p.553. It is needed to offset the reduction in power that is introduced by the windowing.

- (iii) The windowed data is then Fourier transformed into the frequency domain, where it is unwhitened by dividing by the (complex) spectra $\bar{W}_1(f)$ and $\bar{W}_2(f)$, which represent the whitening filters of the two detectors. The resulting unwhitened frequency components are denoted by $^{(1)}\tilde{s}(f)$ and $^{(2)}\tilde{s}(f)$; the superscript (1) indicates that we are analyzing the first of two overlapped data sets.
- (iv) The real-time cross-correlation spectrum is then calculated according to:

$$^{(1)}\tilde{s}_{12}(f) := \left[^{(1)}\tilde{s}_1^*(f) ^{(1)}\tilde{s}_2(f) + \text{c.c.} \right]. \quad (11.16.3)$$

- (v) The data contained in the last two-thirds of the buffers is then copied to the first two-thirds of the buffers, and steps (ii)-(iv) are repeated, yielding a second real-time cross-correlation spectrum $^{(2)}\tilde{s}_{12}(f)$.
- (vi) If `average=1`, $\tilde{s}_{12}(f)$ is given by:

$$\tilde{s}_{12}(f) := \frac{1}{2} \left[^{(1)}\tilde{s}_{12}(f) + ^{(2)}\tilde{s}_{12}(f) \right]. \quad (11.16.4)$$

Otherwise, $\tilde{s}_{12}(f) = ^{(2)}\tilde{s}_{12}(f)$.

- (vii) Finally, the data contained in the last two-thirds of the buffers is again copied to the first two-thirds, in preparation for the next call to `extract_sb()`. The data saved in the first one-third of these buffers will match onto the next input data streams if the input data from one call of `extract_sb()` to the next is continuous.

Note: One should call `extract_sb()` with `average ≠ 1`, when one suspects that the current input data is *not* continuous with the data that was saved from the previous call. This is because a discontinuity between the “old” and “new” data sets has a tendency to introduce spurious large frequency components into the real-time cross-correlation spectrum, which should not be present. Since a single input data stream by itself is continuous, the cross-correlation spectrum $^{(2)}\tilde{s}_{12}(f)$ (which is calculated on the second pass through the data) will be free of these spurious large frequency components. This is why we set $\tilde{s}_{12}(f)$ equal to $^{(2)}\tilde{s}_{12}(f)$ —and not equal to $^{(1)}\tilde{s}_{12}(f)$ —when `average ≠ 1`.

Authors: Bruce Allen, ballen@dirac.phys.uwm.edu, and Joseph Romano, romano@csd.uwm.edu

Comments: Although it is possible and more efficient to write a single function to extract the real-time detector noise power and cross-correlation signal spectra simultaneously, we have chosen—for the sake of modularity—to write separate functions to perform these two tasks separately. (See also the comment at the end of Sec. 11.15.)

11.17 Function: `optimal_filter()`

```
void optimal_filter(int n, float delta_f, float f_low, float f_high, double
*gamma12, double *power1, double *power2, double *filter12)
```

This function calculates the values of the spectrum $\tilde{Q}(f)$ of the optimal filter function, which maximizes the cross-correlation signal-to-noise ratio for an isotropic and unpolarized stochastic background of gravitational radiation having a constant frequency spectrum: $\Omega_{\text{gw}}(f) = \Omega_0$ for $f_{\text{low}} \leq f \leq f_{\text{high}}$.

The arguments of `optimal_filter()` are:

`n`: Input. The number N of discrete frequency values at which the spectrum $\tilde{Q}(f)$ of the optimal filter is to be evaluated.

`delta_f`: Input. The spacing Δf (in Hz) between two adjacent discrete frequency values: $\Delta f := f_{i+1} - f_i$.

`f_low`: Input. The frequency f_{low} (in Hz) below which the spectrum $\Omega_{\text{gw}}(f)$ of the stochastic background—and hence the optimal filter $\tilde{Q}(f)$ —is zero. f_{low} should lie in the range $0 \leq f_{\text{low}} \leq f_{\text{Nyquist}}$, where f_{Nyquist} is the Nyquist critical frequency. (The Nyquist critical frequency is defined by $f_{\text{Nyquist}} := 1/(2\Delta t)$, where Δt is the sampling period of the detectors.) f_{low} should also be less than or equal to f_{high} .

`f_high`: Input. The frequency f_{high} (in Hz) above which the spectrum $\Omega_{\text{gw}}(f)$ of the stochastic background—and hence the optimal filter $\tilde{Q}(f)$ —is zero. f_{high} should lie in the range $0 \leq f_{\text{high}} \leq f_{\text{Nyquist}}$. It should also be greater than or equal to f_{low} .

`gamma12`: Input. `gamma12[0..n-1]` is an array of double precision variables containing the values of the overlap reduction function $\gamma(f)$ for the two detector sites. These variables are dimensionless. `gamma12[i]` contains the value of $\gamma(f)$ evaluated at the discrete frequency $f_i = i\Delta f$, where $i = 0, 1, \dots, N - 1$.

`power1`: Input. `power1[0..n-1]` is an array of double precision variables containing the values of the noise power spectrum $P_1(f)$ of the first detector. These variables have units of $\text{strain}^2/\text{Hz}$ (or seconds). `power1[i]` contains the value of $P_1(f)$ evaluated at the discrete frequency $f_i = i\Delta f$, where $i = 0, 1, \dots, N - 1$.

`power2`: Input. `power2[0..n-1]` is an array of double precision variables containing the values of the noise power spectrum $P_2(f)$ of the second detector, in exactly the same format as the previous argument.

`filter12`: Output. `filter12[0..n-1]` is an array of double precision variables containing the values of the spectrum $\tilde{Q}(f)$ of the optimal filter function for the two detectors. These variables are dimensionless for our choice of normalization $\langle S \rangle = \Omega_0 T$. (See the discussion below.) `filter12[i]` contains the value of $\tilde{Q}(f)$ evaluated at the discrete frequency $f_i = i\Delta f$, where $i = 0, 1, \dots, N - 1$.

The values of $\tilde{Q}(f)$ calculated by `optimal_filter()` are defined by equation (3.32) of Ref. [36]:

$$\tilde{Q}(f) := \lambda \frac{\gamma(f)\Omega_{\text{gw}}(f)}{f^3 P_1(f) P_2(f)}. \quad (11.17.1)$$

Such a filter maximizes the cross-correlation signal-to-noise ratio $\text{SNR} := \mu/\sigma$, where

$$\mu := \langle S \rangle = T \frac{3H_0^2}{20\pi^2} \int_{-\infty}^{\infty} df \gamma(|f|) |f|^{-3} \Omega_{\text{gw}}(|f|) \tilde{Q}(f) \quad (11.17.2)$$

$$\sigma^2 := \langle S^2 \rangle - \langle S \rangle^2 \approx \frac{T}{4} \int_{-\infty}^{\infty} df P_1(|f|) P_2(|f|) |\tilde{Q}(f)|^2. \quad (11.17.3)$$

(T corresponds to the observation time of the measurement.) We are working here under the assumption that the magnitude of the noise intrinsic to the detectors is much larger than the magnitude of the signal due to the stochastic background. If this assumption does not hold, Eq. 11.17.3 for σ^2 needs to be modified, as discussed in Sec. 11.19.

Note that we have explicitly included a normalization constant λ in the definition of $\tilde{Q}(f)$. The choice of λ does not affect the value of the signal-to-noise ratio, since μ and σ are both multiplied by the same factor of λ . For a stochastic background having a constant frequency spectrum

$$\Omega_{\text{gw}}(f) = \begin{cases} \Omega_0 & f_{\text{low}} \leq f \leq f_{\text{high}} \\ 0 & \text{otherwise,} \end{cases}$$

it is convenient to choose λ so that

$$\mu = \Omega_0 T. \quad (11.17.4)$$

From equations (11.17.1) and (11.17.2), it follows that

$$\lambda = \left[\frac{3H_0^2}{10\pi^2} \Omega_0 \int_{f_{\text{low}}}^{f_{\text{high}}} df \frac{\gamma^2(f)}{f^6 P_1(f) P_2(f)} \right]^{-1} \quad (11.17.5)$$

will do the job. With this choice of λ , $\tilde{Q}(f)$ is dimensionless and independent of the value of Ω_0 . This is why Ω_0 does not have to be passed as a parameter to `optimal_filter()`.

Authors: Bruce Allen, ballen@dirac.phys.uwm.edu, and Joseph Romano, romano@csd.uwm.edu

Comments: None.

11.18 Example: optimal_filter program

The following example program shows one way of combining the functions `detector_site()`, `noise_power()`, `overlap()`, and `optimal_filter()` to calculate the spectrum $\tilde{Q}(f)$ of the optimal filter function for a given pair of detectors. Below we explicitly calculate $\tilde{Q}(f)$ for the initial Hanford, WA and Livingston, LA LIGO detectors. (We also choose to normalize the magnitude of the spectrum $\tilde{Q}(f)$ to 1, for later convenience when making plots of the output data.) Noise power information for these two detectors is read from the input data file `noise_init.dat`. This file is specified by the information contained in `detectors.dat`. (See Sec. 11.1 for more details.) The resulting optimal filter function data is stored as two columns of double precision numbers (f_i and $\tilde{Q}(f_i)$) in the file `LIGO_filter.dat`, where $f_i = i\Delta f$ and $i = 0, 1, \dots, N - 1$. A plot of this data is shown in Fig. 79.

As usual, the user can modify the parameters in the `#define` statements listed at the beginning of the program to change the number of frequency points, the frequency spacing, etc. used when calculating $\tilde{Q}(f)$. Also, by changing the site location identification numbers and the output file name, the user can calculate and save the spectrum of the optimal filter function for *any* pair of detectors. For example, Fig. 80 is a plot of the optimal filter function for the advanced LIGO detectors.

```
/* GRASP: Copyright 1997,1998 Bruce Allen */
/* main program to illustrate the function optimal_filter() */

#include "grasp.h"

#define DETECTORS_FILE "detectors.dat" /* file containing detector info */
#define SITE1_CHOICE 1 /* 1=LIGO-Hanford site */
#define SITE2_CHOICE 2 /* 2=LIGO-Livingston site */
#define N 500 /* number of frequency points */
#define DELTA_F 1.0 /* frequency spacing (in Hz) */
#define F_LOW 0.0 /* minimum frequency (in Hz) */
#define F_HIGH (1.0e+4) /* maximum frequency (in Hz) */
#define OUT_FILE "LIGO_filter.dat" /* output filename */

int main(int argc, char **argv)
{
    int i;
    double f;
    double abs_value, max;

    float site1_parameters[9], site2_parameters[9];
    char site1_name[100], noise1_file[100], whiten1_file[100];
    char site2_name[100], noise2_file[100], whiten2_file[100];

    double *power1, *power2;
    double *gamma12;
    double *filter12;

    FILE *fp;
    fp=fopen(OUT_FILE, "w");

    /* ALLOCATE MEMORY */
    power1=(double *)malloc(N*sizeof(double));
    power2=(double *)malloc(N*sizeof(double));
    gamma12=(double *)malloc(N*sizeof(double));
    filter12=(double *)malloc(N*sizeof(double));

    /* CALL DETECTOR_SITE() TO GET SITE PARAMETER INFORMATION */
```

```
detector_site(DETECTORS_FILE, SITE1_CHOICE, site1_parameters, site1_name,
              noise1_file, whiten1_file);
detector_site(DETECTORS_FILE, SITE2_CHOICE, site2_parameters, site2_name,
              noise2_file, whiten2_file);

/* CALL NOISE_POWER() AND OVERLAP() */
noise_power(noise1_file, N, DELTA_F, power1);
noise_power(noise2_file, N, DELTA_F, power2);
overlap(site1_parameters, site2_parameters, N, DELTA_F, gamma12);

/* CALL OPTIMAL_FILTER() AND DETERMINE MAXIMUM ABSOLUTE VALUE */
optimal_filter(N, DELTA_F, F_LOW, F_HIGH, gamma12, power1, power2, filter12);

max=0.0;
for (i=0; i<N; i++) {
    abs_value=fabs(filter12[i]);
    if (abs_value>max) max=abs_value;
}

/* WRITE FILTER FUNCTION (NORMALIZED TO 1) TO FILE */
for (i=0; i<N; i++) {
    f=i*DELTA_F;
    fprintf(fp, "%e %e\n", f, filter12[i]/max);
}

fclose(fp);

return 0;
}
```

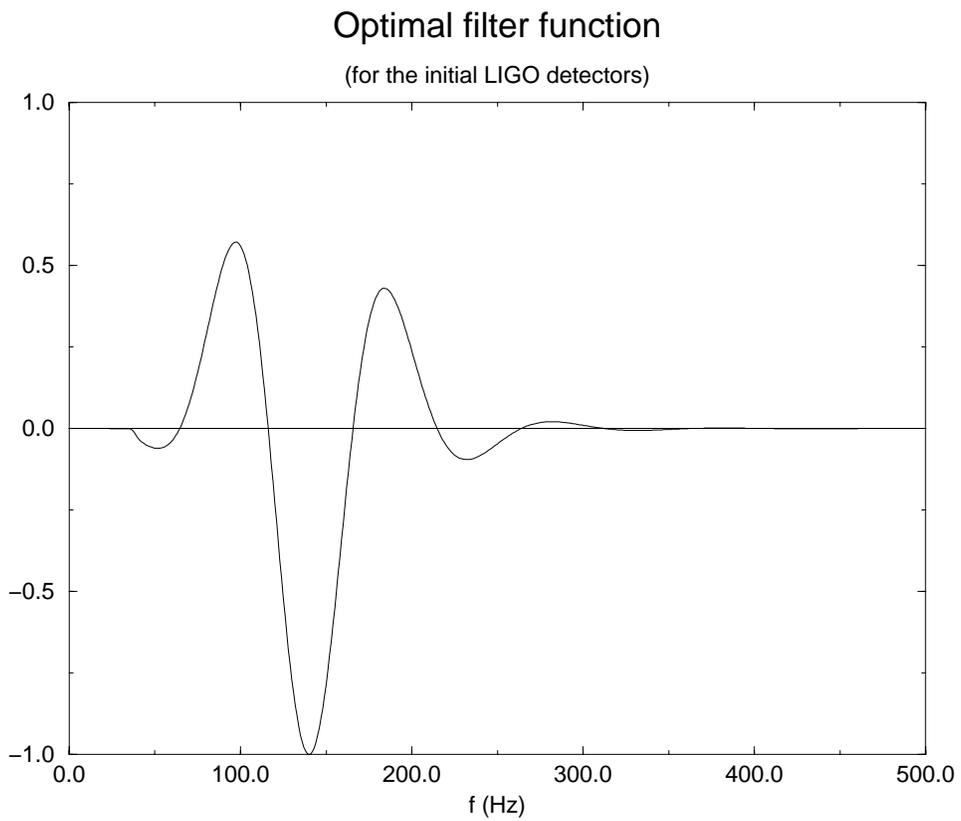


Figure 79: Optimal filter function $\tilde{Q}(f)$ (normalized to 1) for the initial LIGO detectors.

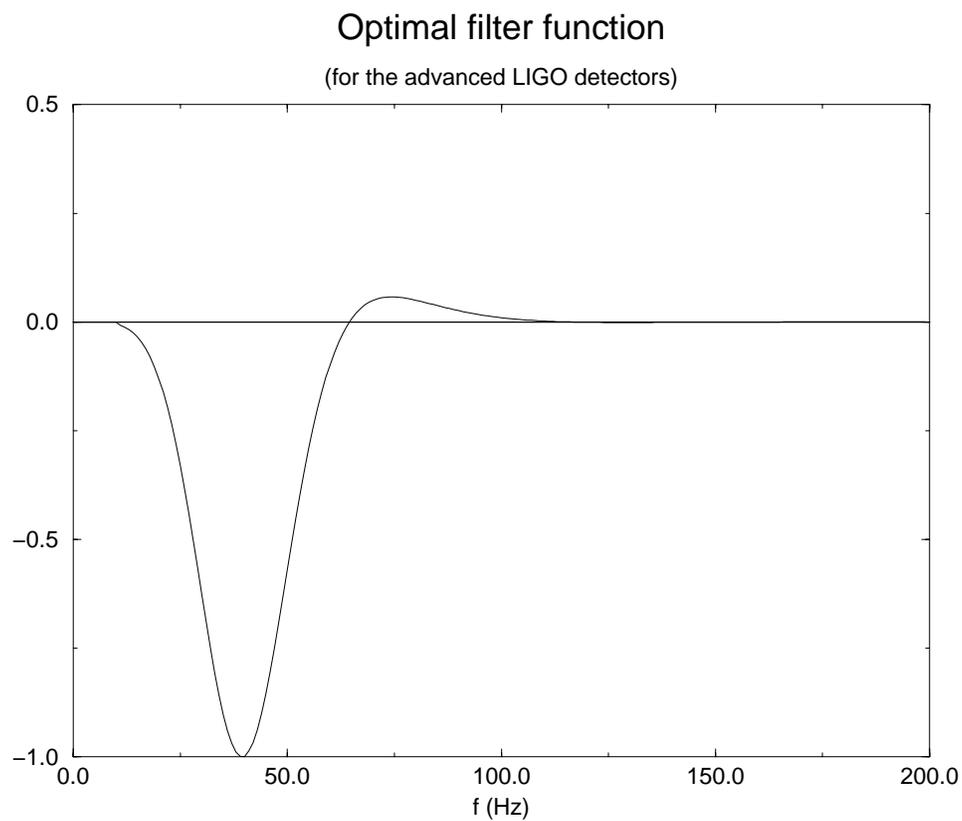


Figure 80: Optimal filter function $\tilde{Q}(f)$ (normalized to 1) for the advanced LIGO detectors.

11.19 Discussion: Theoretical signal-to-noise ratio for the stochastic background

In order to reliably detect a stochastic background of gravitational radiation, we will need to be able to say (with a certain level of confidence) that an observed positive mean value for the cross-correlation signal measurements is not the result of detector noise alone, but rather is the result of an incident stochastic background. This leads us naturally to consider the signal-to-noise ratio, since the larger its value, the more confident we will be in saying that the observed mean value of our measurements is a valid estimate of the true mean value of the stochastic background signal. Thus, an interesting question to ask in regard to stochastic background searches is: “What is the theoretically predicted signal-to-noise ratio after a total observation time T , for a given pair of detectors, and for a given strength of the stochastic background?” In this section, we derive the mathematical equations that we need to answer this question. Numerical results will be calculated by example programs in Secs. 11.21 and 11.22.

To answer the above question, we will need to evaluate both the mean value

$$\mu := \langle S \rangle \quad (11.19.1)$$

and the variance

$$\sigma^2 := \langle S^2 \rangle - \langle S \rangle^2 \quad (11.19.2)$$

of the stochastic background cross-correlation signal S . The signal-to-noise ratio SNR is then given by

$$\text{SNR} := \frac{\mu}{\sigma}. \quad (11.19.3)$$

As described in Sec. 11.17, if the magnitude of the noise intrinsic to the detectors is much larger than the magnitude of the signal due to the stochastic background, then

$$\mu = T \frac{3H_0^2}{20\pi^2} \int_{-\infty}^{\infty} df \gamma(|f|) |f|^{-3} \Omega_{\text{gw}}(|f|) \tilde{Q}(f) \quad (11.19.4)$$

$$\sigma^2 \approx \frac{T}{4} \int_{-\infty}^{\infty} df P_1(|f|) P_2(|f|) |\tilde{Q}(f)|^2, \quad (11.19.5)$$

where $\tilde{Q}(f)$ is an arbitrary filter function. The choice

$$\tilde{Q}(f) := \lambda \frac{\gamma(f) \Omega_{\text{gw}}(f)}{f^3 P_1(f) P_2(f)} \quad (11.19.6)$$

maximizes the signal-to-noise ratio (11.19.3). It is the *optimal* filter for stochastic background searches. As also described in Sec. 11.17, if the stochastic background has a constant frequency spectrum

$$\Omega_{\text{gw}}(f) = \begin{cases} \Omega_0 & f_{\text{low}} \leq f \leq f_{\text{high}} \\ 0 & \text{otherwise,} \end{cases}$$

it is convenient to choose the normalization constant λ so that

$$\mu = \Omega_0 T. \quad (11.19.7)$$

For such a λ ,

$$\sigma^2 \approx \frac{T}{2} \left(\frac{10\pi^2}{3H_0^2} \right)^2 \left[\int_{f_{\text{low}}}^{f_{\text{high}}} df \frac{\gamma^2(f)}{f^6 P_1(f) P_2(f)} \right]^{-1}, \quad (11.19.8)$$

which leads to the *squared* signal-to-noise ratio

$$(\text{SNR})^2 = T \Omega_0^2 \frac{9H_0^4}{50\pi^4} \int_{f_{\text{low}}}^{f_{\text{high}}} df \frac{\gamma^2(f)}{f^6 P_1(f) P_2(f)}. \quad (11.19.9)$$

This is equation (3.33) in Ref. [36].

But suppose that we do *not* assume that the noise intrinsic to the detectors is much larger in magnitude than that of the stochastic background. Then Eq. (11.19.5) for σ^2 needs to be modified to take into account the non-negligible contributions to the variance brought in by the stochastic background signal. (Equation (11.19.4) for μ is unaffected.) This change in σ^2 implies that Eq. (11.19.6) for $\tilde{Q}(f)$ is no longer optimal. But to simplify matters, we will leave $\tilde{Q}(f)$ as is. Although such a $\tilde{Q}(f)$ no longer maximizes the signal-to-noise ratio, it at least has the nice property that, for a stochastic background having a constant frequency spectrum, the normalization constant λ can be chosen so that $\tilde{Q}(f)$ is independent of Ω_0 . The expression for the actual optimal filter function, on the other hand, would depend on Ω_0 .

So keeping Eq. (11.19.6) for $\tilde{Q}(f)$, let us consider a stochastic background having a constant frequency spectrum as described above. Then we can still choose λ so that

$$\mu = \Omega_0 T, \tag{11.19.10}$$

(the same λ as before works), but now

$$\begin{aligned} \sigma^2 = & \frac{T}{2} \left[\int_{f_{\text{low}}}^{f_{\text{high}}} df \frac{\gamma^2(f)}{f^6 P_1(f) P_2(f)} \right]^{-2} \left\{ \left(\frac{10\pi^2}{3H_0^2} \right)^2 \int_{f_{\text{low}}}^{f_{\text{high}}} df \frac{\gamma^2(f)}{f^6 P_1(f) P_2(f)} \right. \\ & + \Omega_0 \left(\frac{10\pi^2}{3H_0^2} \right) \int_{f_{\text{low}}}^{f_{\text{high}}} df \frac{\gamma^2(f)}{f^9 P_1^2(f) P_2(f)} + \Omega_0 \left(\frac{10\pi^2}{3H_0^2} \right) \int_{f_{\text{low}}}^{f_{\text{high}}} df \frac{\gamma^2(f)}{f^9 P_1(f) P_2^2(f)} \\ & \left. + \Omega_0^2 \int_{f_{\text{low}}}^{f_{\text{high}}} df \frac{\gamma^2(f)}{f^{12} P_1^2(f) P_2^2(f)} (1 + \gamma^2(f)) \right\}. \end{aligned} \tag{11.19.11}$$

The new squared signal-to-noise ratio is $\Omega_0^2 T^2$ divided by the above expression for σ^2 .

Note the three additional terms that contribute to the variance σ^2 . Roughly speaking, they can be thought of as two “signal+noise” cross-terms and one “pure signal” variance term. These are the terms proportional to Ω_0 and Ω_0^2 , respectively. When Ω_0 is small, the above expression for σ^2 reduces to the pure noise variance term (11.19.8). This is what we expect to be the case in practice. But for the question that we posed at the beginning of the section, where no assumption is made about the relative strength of the stochastic background and detector noise signals, the more complicated expression (11.19.11) for σ^2 should be used. The function `calculate_var()`, which is defined in the following section, calculates the variance using this equation.

11.20 Function: calculate_var()

```
double calculate_var(int n, float delta_f, float omega_0, float f_low, float
f_high, float t, double *gamma12, double *power1, double *power2)
```

This function calculates the theoretical variance σ^2 of the stochastic background cross-correlation signal S .

The arguments of `calculate_var()` are:

`n`: Input. The number N of discrete frequency values at which the spectra are to be evaluated.

`delta_f`: Input. The spacing Δf (in Hz) between two adjacent discrete frequency values: $\Delta f := f_{i+1} - f_i$.

`omega_0`: Input. The constant value Ω_0 (dimensionless) of the frequency spectrum $\Omega_{\text{gw}}(f)$ for the stochastic background:

$$\Omega_{\text{gw}}(f) = \begin{cases} \Omega_0 & f_{\text{low}} \leq f \leq f_{\text{high}} \\ 0 & \text{otherwise.} \end{cases}$$

Ω_0 should be greater than or equal to zero.

`f_low`: Input. The frequency f_{low} (in Hz) below which the spectrum $\Omega_{\text{gw}}(f)$ of the stochastic background is zero. f_{low} should lie in the range $0 \leq f_{\text{low}} \leq f_{\text{Nyquist}}$, where f_{Nyquist} is the Nyquist critical frequency. (The Nyquist critical frequency is defined by $f_{\text{Nyquist}} := 1/(2\Delta t)$, where Δt is the sampling period of the detector.) f_{low} should also be less than or equal to f_{high} .

`f_high`: Input. The frequency f_{high} (in Hz) above which the spectrum $\Omega_{\text{gw}}(f)$ of the stochastic background is zero. f_{high} should lie in the range $0 \leq f_{\text{high}} \leq f_{\text{Nyquist}}$. It should also be greater than or equal to f_{low} .

`t`: Input. The observation time T (in sec) of the measurement.

`gamma12`: Input. `gamma12[0..n-1]` is an array of double precision variables containing the values of the overlap reduction function $\gamma(f)$ for the two detector sites. These variables are dimensionless. `gamma12[i]` contains the value of $\gamma(f)$ evaluated at the discrete frequency $f_i = i\Delta f$, where $i = 0, 1, \dots, N - 1$.

`power1`: Input. `power1[0..n-1]` is an array of double precision variables containing the values of the noise power spectrum $P_1(f)$ of the first detector. These variables have units of $\text{strain}^2/\text{Hz}$ (or seconds). `power1[i]` contains the value of $P_1(f)$ evaluated at the discrete frequency $f_i = i\Delta f$, where $i = 0, 1, \dots, N - 1$.

`power2`: Input. `power2[0..n-1]` is an array of double precision variables containing the values of the noise power spectrum $P_2(f)$ of the second detector, in exactly the same format as the previous argument.

The double precision value returned by `calculate_var()` is the theoretical variance σ^2 given by Eq. (11.19.11) of Sec. 11.19. As discussed in that section, Eq. (11.19.11) for σ^2 makes no assumption about the relative strengths of the stochastic background and detector noise signal, but it does use Eq. (11.19.6) for the filter function $\tilde{Q}(f)$, which is optimal only for the large detector noise case. For stochastic background simulations, Ω_0 is usually chosen to equal some known non-zero value. This is the value that should be passed as a parameter to `calculate_var()`. For stochastic background searches (where Ω_0 is not known a priori) the value of the parameter Ω_0 should be set to zero. The variance for this case is given by Eq. (11.19.8).

Section	GRASP Routines: Stochastic background detection	Page
11.20	Function: calculate_var()	411

Authors: Bruce Allen, ballen@dirac.phys.uwm.edu, and Joseph Romano, romano@csd.uwm.edu

Comments: None.

11.21 Example: snr program

As mentioned in Sec. 11.19, an interesting question to ask in regard to stochastic background searches is: “What is the theoretically predicted signal-to-noise ratio after a total observation time T , for a given pair of detectors, and for a given strength of the stochastic background?” The following example program shows how one can combine the functions `detector_site()`, `noise_power()`, `overlap()`, and `calculate_var()` to answer this question for the case of a stochastic background having a constant frequency spectrum: $\Omega_{\text{gw}}(f) = \Omega_0$ for $f_{\text{low}} \leq f \leq f_{\text{high}}$. Specifically, we calculate and display the theoretical SNR after approximately 4 months of observation time ($T = 1.0 \times 10^7$ seconds), for the initial Hanford, WA and Livingston, LA LIGO detectors, and for $\Omega_0 = 3.0 \times 10^{-6}$ for $5 \text{ Hz} \leq f \leq 5000 \text{ Hz}$. (The answer is $\text{SNR} = 1.73$, which means that we could say, with greater than 95% confidence, that a stochastic background has been detected.) By changing the parameters in the `#define` statements listed at the beginning of the program, one can calculate and display the signal-to-noise ratios for different observation times T , for different detector pairs, and for different strengths Ω_0 of the stochastic background.

Note: Values of N and Δf should be chosen so that the whole frequency range (from DC to the Nyquist critical frequency) is included, and that there are a reasonably large number of discrete frequency values for approximating integrals by sums. The final answer, however, is independent of the choice of N and Δf , for N sufficiently large and Δf sufficiently small.

```
/* GRASP: Copyright 1997,1998 Bruce Allen */
/* main program to calculate the theoretical snr */

#include "grasp.h"

#define DETECTORS_FILE "detectors.dat" /* file containing detector info */
#define SITE1_CHOICE 1 /* 1=LIGO-Hanford site */
#define SITE2_CHOICE 2 /* 2=LIGO-Livingston site */
#define OMEGA_0 (3.0e-6) /* Omega_0 (for initial detectors) */
#define F_LOW 0.0 /* minimum frequency (in Hz) */
#define F_HIGH (1.0e+4) /* maximum frequency (in Hz) */
#define T (1.0e+7) /* total observation time (in sec) */
#define N 40000 /* number of frequency points */
#define DELTA_F 0.25 /* frequency spacing (in Hz) */

int main(int argc, char **argv)
{
    double mean, variance, stddev, snr;

    float site1_parameters[9], site2_parameters[9];
    char site1_name[100], noise1_file[100], whiten1_file[100];
    char site2_name[100], noise2_file[100], whiten2_file[100];

    double *power1, *power2;
    double *gamma12;

    /* ALLOCATE MEMORY */
    power1=(double *)malloc(N*sizeof(double));
    power2=(double *)malloc(N*sizeof(double));
    gamma12=(double *)malloc(N*sizeof(double));

    /* CALL DETECTOR_SITE() TO GET SITE PARAMETER INFORMATION */
    detector_site(DETECTORS_FILE, SITE1_CHOICE, site1_parameters, site1_name,
                 noise1_file, whiten1_file);
    detector_site(DETECTORS_FILE, SITE2_CHOICE, site2_parameters, site2_name,
                 noise2_file, whiten2_file);
```

```
/* CALL NOISE_POWER() AND OVERLAP() */
noise_power(noise1_file,N,DELTA_F,power1);
noise_power(noise2_file,N,DELTA_F,power2);
overlap(site1_parameters,site2_parameters,N,DELTA_F,gamma12);

/* CALCULATE MEAN, VARIANCE, STDDEV, AND SNR */
mean=OMEGA_0*T;
variance=calculate_var(N,DELTA_F,OMEGA_0,F_LOW,F_HIGH,T,gamma12,
                      power1,power2);
stddev=sqrt(variance);
snr=mean/stddev;

/* DISPLAY RESULTS */
printf("\n");
printf("Detector site 1 = %s\n",site1_name);
printf("Detector site 2 = %s\n",site2_name);
printf("Omega_0 = %e\n",OMEGA_0);
printf("f_low  = %e Hz\n",F_LOW);
printf("f_high = %e Hz\n",F_HIGH);
printf("Observation time T = %e sec\n",T);
printf("Theoretical S/N = %e\n",snr);
printf("\n");

return 0;
}
```

11.22 Example: omega_min program

The example program described in the previous section calculates the theoretical signal-to-noise ratio after a total observation time T , for a given pair of detectors, and for a given strength Ω_0 of the stochastic background. A related—and equally important—question is the *inverse*: “What is the minimum value of Ω_0 required to produce a given SNR after a given observation time T ?” For example, if $\text{SNR} = 1.65$, then the answer to the above question is the minimum value of Ω_0 for a stochastic background that is detectable with 95% confidence after an observation time T . The following example program calculates and displays this 95% confidence value of Ω_0 for the initial Hanford, WA and Livingston, LA LIGO detectors, for approximately 4 months ($T = 1.0 \times 10^7$ seconds) of observation time. (The answer is $\Omega_0 = 2.87 \times 10^{-6}$.) Again, we are assuming in this example program that the stochastic background has a constant frequency spectrum: $\Omega_{\text{gw}}(f) = \Omega_0$ for $5 \text{ Hz} \leq f \leq 5000 \text{ Hz}$. By modifying the parameters in the #define statements listed at the beginning of the program, one can calculate and display the minimum required Ω_0 's for different detector pairs, for different signal-to-noise ratios, and for different observation times T .

Note: As shown in Sec. 11.19, the squared signal-to-noise ratio can be written in the following form:

$$(\text{SNR})^2 = \frac{T \Omega_0^2}{A + B \Omega_0 + C \Omega_0^2}, \quad (11.22.1)$$

where A , B , and C are complicated expressions involving integrals of the the overlap reduction function and the noise power spectra of the detectors, but are independent of T and Ω_0 . Thus, given SNR and T , Eq. (11.22.1) becomes a quadratic for Ω_0 :

$$a \Omega_0^2 + b \Omega_0 + c = 0, \quad (11.22.2)$$

which we can easily solve. It is this procedure that we implement in the following program.

The omega_min example program can be run in two ways. Without any arguments:

```
machine-prompt> omega_min
```

uses the detectors defined by SITE1_CHOICE and SITE2_CHOICE. The program can also be run with two command line arguments which specify alternative detector site choices, for example:

```
machine-prompt> omega_min 23 31
```

which produces the output:

```
Detector site 1 = LIGO-WA_enh7
Detector site 2 = LIGO-LA_enh7
S/N ratio = 1.650000e+00
f_low   = 0.000000e+00 Hz
f_high  = 1.000000e+04 Hz
Observation time T = 1.000000e+07 sec
Minumum Omega_0 = 5.290809e-09
```

```
/* GRASP: Copyright 1997,1998 Bruce Allen */
/* main program to calculate the minimum detectable omega_0 */
```

```
#include "grasp.h"
```

```
#define DETECTORS_FILE "detectors.dat" /* file containing detector info */
#define SITE1_CHOICE 1 /* 1=LIGO-Hanford site */
#define SITE2_CHOICE 2 /* 2=LIGO-Livingston site */
#define SNR 1.65 /* 1.65=SNR for 95% confidence */
#define F_LOW 3.0 /* minimum frequency (in Hz) */
#define F_HIGH (1.0e+4) /* maximum frequency (in Hz) */
```

```
#define T (1.0e+7)          /* total observation time (in sec) */
#define N 40000            /* number of frequency points */
#define DELTA_F 0.25       /* frequency spacing (in Hz) */
#define PLOT_WANTED 0

#if PLOT_WANTED
FILE *fp;
#endif

int main(int argc, char **argv)
{
    int    i;
    float  f;

    double factor, f3, f6, f9, f12, p1, p2, g2;
    double int1, int2, int3, int4;
    double a, b, c, omega_0;
    int    site1_choice=SITE1_CHOICE, site2_choice=SITE2_CHOICE;

    float  site1_parameters[9], site2_parameters[9];
    char   site1_name[100], noise1_file[100], whiten1_file[100];
    char   site2_name[100], noise2_file[100], whiten2_file[100];

    double *power1, *power2;
    double *gamma12;

    /* ALLOCATE MEMORY */
    power1=(double *)malloc(N*sizeof(double));
    power2=(double *)malloc(N*sizeof(double));
    gamma12=(double *)malloc(N*sizeof(double));

    /* Use detector sites specified on command line */
    if (argc==3) {
        site1_choice=atoi(argv[1]);
        site2_choice=atoi(argv[2]);
    }

    /* CALL DETECTOR_SITE() TO GET SITE PARAMETER INFORMATION */
    detector_site(DETECTORS_FILE, site1_choice, site1_parameters, site1_name,
                 noise1_file, whiten1_file);
    detector_site(DETECTORS_FILE, site2_choice, site2_parameters, site2_name,
                 noise2_file, whiten2_file);

#if PLOT_WANTED
    /* output file of integrand if needed */
    fp=fopen(site1_name, "w");
#endif

    /* CALL NOISE_POWER() AND OVERLAP() */
    noise_power(noise1_file, N, DELTA_F, power1);
    noise_power(noise2_file, N, DELTA_F, power2);
    overlap(site1_parameters, site2_parameters, N, DELTA_F, gamma12);

    /* CALCULATE INTEGRALS FOR VARIANCE */
    int1=int2=int3=int4=0.0;

    for (i=1; i<N; i++) { /* start sum at i=1 to avoid possible division */
                          /* by 0 (e.g., if f_low=0) */
```

```
f=i*DELTA_F;
if (F_LOW<=f && f<=F_HIGH) {
    f3=f*f*f;
    f6=f3*f3;
    f9=f6*f3;
    f12=f9*f3;
    g2=gamma12[i]*gamma12[i];
    p1=power1[i];
    p2=power2[i];

#if PLOT_WANTED
fprintf(fp, "%e\t%e\n", f, 1.e-80*g2/(f6*p1*p2));
#endif

    int1+=DELTA_F*g2/(f6*p1*p2);
    int2+=DELTA_F*g2/(f9*p1*p1*p2);
    int3+=DELTA_F*g2/(f9*p1*p2*p2);
    int4+=DELTA_F*g2*(1.0+g2)/(f12*p1*p1*p2*p2);
}
}

/* CALCULATE COEFFICIENTS OF QUADRATIC EQUATION */
factor=10.0*M_PI*M_PI/(3.0*HUBBLE*HUBBLE);

a=(int4/int1-2.0*T*int1/(SNR*SNR))/(factor*factor);
b=(int2+int3)/(int1*factor);
c=1.0;

/* SOLVE THE QUADRATIC */
omega_0=0.5*(-b-sqrt(b*b-4*a*c))/a;

/* DISPLAY RESULTS */
printf("\n");
printf("Detector site 1 = %s\n",site1_name);
printf("Detector site 2 = %s\n",site2_name);
printf("S/N ratio = %e\n",SNR);
printf("f_low = %e Hz\n",F_LOW);
printf("f_high = %e Hz\n",F_HIGH);
printf("Observation time T = %e sec\n",T);
printf("Minimum Omega_0 h_100^2 = %e\n",omega_0);
printf("(This corresponds to false alarm rate 5%% and false dismissal rate 50%%.)\n");
printf("With a 5%% false alarm rate and a 5%% false dismissal rate:\n");
printf("minimum Omega_0 h_100^2 = %e\n",2.0*omega_0);
printf("See Allen & Romano, PRD59 (1999) 102001 eqn (4.38) for details.\n");
printf("\n");

#if PLOT_WANTED
fclose(fp);
#endif

return 0;
}
```

Authors: Bruce Allen, ballen@dirac.phys.uwm.edu, and Joseph Romano, romano@csd.uwm.edu

Comments: None.

11.23 Function: analyze()

```
void analyze(int average, float *in1, float *in2, int n, float delta_t, float
f_low, float f_high, double *gamma12, double *whiten1, double *whiten2, int
realtime_noise1, int realtime_noise2, double *power1, double *power2, dou-
ble *signal, double *variance)
```

This high-level function performs the optimal data processing for the detection of an isotropic and unpolarized stochastic background of gravitational radiation having a constant frequency spectrum: $\Omega_{\text{gw}}(f) = \Omega_0$ for $f_{\text{low}} \leq f \leq f_{\text{high}}$. It calculates the cross-correlation signal value S and theoretical variance σ^2 , taking as input the continuous-in-time whitened data streams $o_1(t)$ and $o_2(t)$ produced by two detectors.

The arguments of `analyze()` are:

average: Input. An integer variable that should be set equal to 1 if the values of the real-time cross-correlation and/or noise power spectra corresponding to two overlapped data sets are to be averaged.

in1: Input. `in1[0..n-1]` is an array of floating point variables containing the values of the continuous-in-time whitened data stream $o_1(t)$ produced by the first detector. $o_1(t)$ is the convolution of detector whitening filter $W_1(t)$ with the data stream $s_1(t) := h_1(t) + n_1(t)$, where $h_1(t)$ is the gravitational strain and $n_1(t)$ is the noise intrinsic to the detector. These variables have units of rHz (or $\text{sec}^{-1/2}$), which follows from the definition of $s_1(t)$ as a strain and $\bar{W}_1(f)$ as the “inverse” of the square root of the noise power spectrum $P_1(f)$. `in1[i]` contains the value of $o_1(t)$ evaluated at the discrete time $t_i = i\Delta t$, where $i = 0, 1, \dots, N - 1$.

in2: Input. `in2[0..n-1]` is an array of floating point variables containing the values of the continuous-in-time whitened data stream $o_2(t)$ produced by the second detector, in exactly the same format as the previous argument.

n: Input. The number N of data points corresponding to an observation time $T := N \Delta t$, where Δt is the sampling period of the detectors, defined below. N should equal an integer power of 2.

delta_t: Input. The sampling period Δt (in sec) of the detectors.

f_low: Input. The frequency f_{low} (in Hz) below which the spectrum $\Omega_{\text{gw}}(f)$ of the stochastic background is assumed to be zero. f_{low} should lie in the range $0 \leq f_{\text{low}} \leq f_{\text{Nyquist}}$, where f_{Nyquist} is the Nyquist critical frequency. (The Nyquist critical frequency is defined by $f_{\text{Nyquist}} := 1/(2\Delta t)$, where Δt is the sampling period of the detectors.) f_{low} should also be less than or equal to f_{high} .

f_high: Input. The frequency f_{high} (in Hz) above which the spectrum $\Omega_{\text{gw}}(f)$ of the stochastic background is assumed to be zero. f_{high} should lie in the range $0 \leq f_{\text{high}} \leq f_{\text{Nyquist}}$. It should also be greater than or equal to f_{low} .

gamma12: Input. `gamma12[0..n/2-1]` is an array of double precision variables containing the values of the overlap reduction function $\gamma(f)$ for the two detector sites. These variables are dimensionless. `gamma12[i]` contains the value of $\gamma(f)$ evaluated at the discrete frequency $f_i = i/(N\Delta t)$, where $i = 0, 1, \dots, N/2 - 1$.

whiten1: Input. `whiten1[0..n-1]` is an array of double precision variables containing the values of the real and imaginary parts of the spectrum $\bar{W}_1(f)$ of the whitening filter of the first detector. These variables have units rHz/strain (or $\text{sec}^{-1/2}$), which are inverse to the units of the square root of the noise power spectrum $P_1(f)$. `whiten1[2*i]` and `whiten1[2*i+1]` contain, respectively, the values of the real and imaginary parts of $\bar{W}_1(f)$ evaluated at the discrete frequency $f_i = i/(N\Delta t)$, where $i = 0, 1, \dots, N/2 - 1$.

`whiten2`: Input. `whiten2[0..n-1]` is an array of double precision variables containing the values of the real and imaginary parts of the spectrum $\tilde{W}_2(f)$ of the whitening filter of the second detector, in exactly the same format as the previous argument.

`real_time_noise1`: Input. An integer variable that should be set equal to 1 if the real-time noise power spectrum $P_1(f)$ of the first detector should be calculated and used when performing the data analysis.

`real_time_noise2`: Input. An integer variable that should be set equal to 1 if the real-time noise power spectrum $P_2(f)$ for the second detector should be calculated and used when performing the data analysis.

`power1`: Input/Output. `power1[0..n/2-1]` is an array of double precision variables containing the values of the noise power spectrum $P_1(f)$ of the first detector. These variables have units of $\text{strain}^2/\text{Hz}$ (or seconds). `power1[i]` contains the value of $P_1(f)$ evaluated at the discrete frequency $f_i = i/(N\Delta t)$, where $i = 0, 1, \dots, N/2 - 1$. If `real_time_noise1` = 1, the values of `power1[0..n/2-1]` are changed to

$$P_1(f) := \frac{2}{T} \tilde{s}_1^*(f) \tilde{s}_1(f), \quad (11.23.1)$$

where $\tilde{s}_1(f)$ is the Fourier transform of the unwhitened data stream $s_1(t)$ at the first detector site. If `real_time_noise1` \neq 1, the values of `power1[0..n/2-1]` are unchanged.

`power2`: Input/Output. `power2[0..n/2-1]` is an array of double precision variables containing the values of the noise power spectrum $P_2(f)$ of the second detector, in exactly the same format as the previous argument.

`signal`: Output. A pointer to a double precision variable containing the value of the cross-correlation signal

$$S := \int_{f_{\text{low}}}^{f_{\text{high}}} df \tilde{s}_{12}(f) \tilde{Q}(f), \quad (11.23.2)$$

where $\tilde{s}_{12}(f)$ is the real-time cross-correlation spectrum and $\tilde{Q}(f)$ is the spectrum of the optimal filter function. S has units of seconds.

`variance`: Output. A pointer to a double precision variable containing the value of the theoretical variance σ^2 of the cross-correlation signal S . σ^2 has units of sec^2 .

`analyze()` is very simple function, consisting primarily of calls to other more basic functions. If `real_time_noise1` or `real_time_noise2` = 1, `analyze()` calls `extract_noise()` to obtain the desired real-time noise power spectra. It then calls `extract_signal()` and `optimal_filter()` to obtain the values of $\tilde{s}_{12}(f)$ and $\tilde{Q}(f)$, which are needed to calculate the cross-correlation signal S , according to Eq. (11.23.2). Finally, `analyze()` calls `calculate_var()` to obtain the theoretical variance σ^2 associated with S .

Note: One should call `analyze()` with `average` \neq 1, when one suspects that the current input data `in1[]` and `in2[]` are *not* continuous with the data from the previous call to `analyze()`. This is because a discontinuity between the “old” and “new” data sets has a tendency to introduce spurious large frequency components into the real-time cross-correlation and/or noise power spectra, which should not be present. (See the discussion at the end of Secs. 11.15 and 11.16 for more details.)

Authors: Bruce Allen, ballen@dirac.phys.uwm.edu, and Joseph Romano, romano@csd.uwm.edu

Comments: None.

11.24 Function: `prelim_stats()`

`prelim_stats(float omega_0, float t, double signal, double variance)`

This function calculates and displays the theoretical and experimental mean value, standard deviation, and signal-to-noise ratio for a set of stochastic background cross-correlation signal measurements, weighting each measurement by the inverse of the theoretical variance associated with that measurement.

The arguments of `prelim_stats()` are:

`omega_0`: Input. The constant value Ω_0 (dimensionless) of the frequency spectrum $\Omega_{\text{gw}}(f)$ for the stochastic background:

$$\Omega_{\text{gw}}(f) = \begin{cases} \Omega_0 & f_{\text{low}} \leq f \leq f_{\text{high}} \\ 0 & \text{otherwise.} \end{cases}$$

Ω_0 should be greater than or equal to zero.

`float t`: Input. The observation time T (in sec) of an individual measurement.

`double signal`: Input. The value S of the current cross-correlation signal measurement. This variable has units of seconds.

`double variance`: Input. The value σ^2 of the theoretical variance associated with the current cross-correlation signal measurement. This variable has units of sec^2 .

`prelim_stats()` calculates the theoretical and experimental mean value, standard deviation, and signal-to-noise ratio, weighting each measurement S_i by the inverse of the theoretical variance σ_i^2 associated with that measurement. This choice of weighting maximizes the theoretical signal-to-noise, allowing for possible drifts in the detector noise power spectra over the course of time. More precisely, if we let S_i ($i = 1, 2, \dots, n$) denote a set of n statistically independent random variables, each having the same mean value

$$\mu := \langle S_i \rangle, \tag{11.24.1}$$

but different variances

$$\sigma_i^2 := \langle S_i^2 \rangle - \langle S_i \rangle^2, \tag{11.24.2}$$

then one can show that the weighted-average

$$\bar{S} := \frac{\sum_{i=1}^n \lambda_i S_i}{\sum_{j=1}^n \lambda_j} \tag{11.24.3}$$

has maximum signal-to-noise ratio when $\lambda_i = \sigma_i^{-2}$. Roughly speaking, the above averaging scheme assigns more weight to signal values that are measured when the detectors are “quiet,” than to signal values that are measured when the detectors are “noisy.”

The values calculated and displayed by `prelim_stats()` are determined as follows:

- (i) The total observation time is

$$T_{\text{tot}} := n T, \tag{11.24.4}$$

where n is the total number of measurements, and T is the observation time of an individual measurement.

- (ii) The theoretical mean is given by the product

$$\mu_{\text{theory}} = \Omega_0 T. \tag{11.24.5}$$

This follows from our choice of normalization constant for the optimal filter function. (See Sec. 11.17 for more details.)

(iii) The theoretical variance is given by

$$\sigma_{\text{theory}}^2 = \frac{n}{\sum_{i=1}^n \sigma_i^{-2}} . \quad (11.24.6)$$

Note that when the detector noise power spectra are constant, $\sigma_i^2 =: \sigma^2$ for $i = 1, 2, \dots, n$ and $\sigma_{\text{theory}}^2 = \sigma^2$. This case arises, for example, if we do *not* calculate real-time noise power spectra, but use noise power information contained in data files instead.

(iv) The theoretical signal-to-noise ratio (for n measurements) is given by

$$\text{SNR}_{\text{theory}} = \sqrt{n} \frac{\mu_{\text{theory}}}{\sigma_{\text{theory}}} . \quad (11.24.7)$$

The factor of \sqrt{n} comes from our assumption that the n individual measurements are statistically independent.

(v) The experimental mean is the weighted-average

$$\mu_{\text{expt}} := \frac{\sum_{i=1}^n \sigma_i^{-2} S_i}{\sum_{j=1}^n \sigma_j^{-2}} . \quad (11.24.8)$$

(vi) The experimental variance is given by

$$\sigma_{\text{expt}}^2 := \frac{\sum_{i=1}^n \sigma_i^{-2} S_i^2}{\sum_{j=1}^n \sigma_j^{-2}} - \mu_{\text{expt}}^2 . \quad (11.24.9)$$

When the weights σ_i^{-2} are constant, the above formula reduces to the usual expression

$$\sigma_{\text{expt}}^2 = \frac{1}{n} \sum_{i=1}^n S_i^2 - \left(\frac{1}{n} \sum_{i=1}^n S_i \right)^2 \quad (11.24.10)$$

for the variance of n measurements S_i .

(vii) The experimental signal-to-noise ratio is given by

$$\text{SNR}_{\text{expt}} = \sqrt{n} \frac{\mu_{\text{expt}}}{\sigma_{\text{expt}}} . \quad (11.24.11)$$

(viii) The relative error in the signal-to-noise ratios is

$$\text{relative error} := \left| \frac{\text{SNR}_{\text{theory}} - \text{SNR}_{\text{expt}}}{\text{SNR}_{\text{theory}}} \right| \cdot 100\% . \quad (11.24.12)$$

The value of this quantity should be on the order of $(1/\text{SNR}_{\text{theory}}) \cdot 100\%$.

Note: `prelim_stats()` has internally-defined static variables which keep track of the number of times that it has been called, the sum of the weights, the sum of weights times the signal values, and the sum of the weights times the signal values squared.

Authors: Bruce Allen, ballen@dirac.phys.uwm.edu, and Joseph Romano, romano@csd.uwm.edu

Comments: None.

11.25 Function: `statistics()`

```
void statistics(float *input, int n, int num_bins)
```

This function calculates and displays the mean value, standard deviation, signal-to-noise ratio, and confidence intervals for an input array of (assumed) statistically independent measurements x_i of a random variable x . This function also write output data to two files: `histogram.dat` and `gaussian.dat`. The first file contains a histogram of the input data x_i ; the second file contains the Gaussian probability distribution that best matches this histogram. (See Sec. 11.23 for more details.)

The arguments of `statistics()` are:

`input`: Input. `input[0..n-1]` is an array of floating point variables containing the values of a set of (assumed) statistically independent measurements x_i of a random variable x .

`n`: Input. The length N of the input data array. If $N < 2$, `statistics()` prints out an error message and aborts execution.

`num_bins`: Input. The number of bins to be used when constructing a histogram of the input data x_i .

`statistics()` calculates and displays the mean value and standard deviation of the input data x_i . It also calculates and displays the signal-to-noise ratio and 68%, 90%, and 95% confidence intervals for the input data, assuming that the x_i are statistically independent measurements of a random variable x . `statistics()` also writes output data to two files:

- (i) `histogram.dat` is a two-column file of floating point numbers containing a histogram of the input data x_i . The length of each column of data is equal to `num_bins`, and the histogram is normalized so that it has unit area.
- (ii) `gaussian.dat` is a two-column file of floating point numbers containing the Gaussian probability distribution function that best matches the histogram of the input data x_i . Each column of `gaussian.dat` has a length equal to 8192. There are also three *markers* included in the Gaussian probability distribution data: One marker for the mean, and two for the \pm one standard deviation values of x .

Authors: Bruce Allen, ballen@dirac.phys.uwm.edu, and Joseph Romano, romano@csd.uwm.edu

Comments: In the context of the stochastic background routines, `statistics()` is used to perform a statistical analysis of the cross-correlation signal values S_i calculated by the function `analyze()`.

11.26 Example: simulation program

By combining all of the functions defined in the previous sections, one can write a program to simulate the generation and detection of a stochastic background of gravitational radiation having a constant frequency spectrum: $\Omega_{\text{gw}}(f) = \Omega_0$ for $f_{\text{low}} \leq f \leq f_{\text{high}}$. The following example program is a simulation for the initial Hanford, WA and Livingston, LA LIGO detectors. The parameters chosen for this particular simulation are contained in the #define statements listed at the beginning of the program. By changing these parameters, one can simulate the generation and detection of a stochastic background for different stochastic backgrounds (i.e., for different values of Ω_0 , f_{low} , and f_{high}) and for different detector pairs. The number of data points, the sampling period of the detectors, and the total observation time for the simulation, etc. can also be modified. Preliminary statistics are displayed during the simulation. In addition, a histogram and the best-fit Gaussian probability distribution for the output data are stored in two files: histogram.dat and gaussian.dat. Sample output produced by the simulation and a plot of the histogram and best-fit Gaussian data are given in Sec. 11.27.

```
/* GRASP: Copyright 1997,1998 Bruce Allen */
/* main program for stochastic background simulation */

#include "grasp.h"

#define DETECTORS_FILE "detectors.dat" /* file containing detector info */
#define SITE1_CHOICE 1 /* identification number for site 1 */
#define SITE2_CHOICE 2 /* identification number for site 2 */
#define FAKE_SB 1 /* 1: simulate stochastic background */
/* 0: stochastic background from real data */
#define FAKE_NOISE1 1 /* 1: simulate detector noise at site 1 */
/* 0: detector noise from real data at site 1 */
#define FAKE_NOISE2 1 /* 1: simulate detector noise at site 2 */
/* 0: detector noise from real data at site 2 */

#define N 65536 /* number of data points */
#define DELTA_T (5.0e-5) /* sampling period (in sec) */
#define OMEGA_0 (1.0e-3) /* omega_0 */
#define F_LOW (0.0) /* minimum frequency (in Hz) */
#define F_HIGH (1.0e4) /* maximum frequency (in Hz) */
#define REAL_TIME_NOISE1 0 /* 1: use real-time noise at site 1 */
/* 0: use noise information from data file */
#define REAL_TIME_NOISE2 0 /* 1: use real-time noise at site 2 */
/* 0: use noise information from data file */

#define NUM_RUNS 2500 /* number of runs (for simulation) */
#define NUM_BINS 200 /* number of bins (for statistics) */

int main(int argc, char **argv)
{
    int i, pass_test=0, previous_test, runs_completed=0, seed= -17;
    float delta_f;
    double signal, variance;

    float site1_parameters[9], site2_parameters[9];
    char site1_name[100], noise1_file[100], whiten1_file[100];
    char site2_name[100], noise2_file[100], whiten2_file[100];

    double *generation_power1, *generation_power2;
    double *analysis_power1, *analysis_power2;
    double *whiten1, *whiten2;
    double *gamma12;
    float *out1, *out2;
```

```
float *stats;

/* ALLOCATE MEMORY */
generation_power1=(double *)malloc((N/2)*sizeof(double));
generation_power2=(double *)malloc((N/2)*sizeof(double));
analysis_power1=(double *)malloc((N/2)*sizeof(double));
analysis_power2=(double *)malloc((N/2)*sizeof(double));
whiten1=(double *)malloc(N*sizeof(double));
whiten2=(double *)malloc(N*sizeof(double));
gamma12=(double *)malloc((N/2)*sizeof(double));
out1=(float *)malloc(N*sizeof(float));
out2=(float *)malloc(N*sizeof(float));
stats=(float *)malloc(NUM_RUNS*sizeof(float));

/* INITIALIZE OUTPUT ARRAYS TO ZERO */
for (i=0;i<N;i++) out1[i]=out2[i]=0.0;

/* CALL DETECTOR_SITE() TO GET SITE PARAMETER INFORMATION */
detector_site(DETECTORS_FILE,SITE1_CHOICE,site1_parameters,site1_name,
              noise1_file,whiten1_file);
detector_site(DETECTORS_FILE,SITE2_CHOICE,site2_parameters,site2_name,
              noise2_file,whiten2_file);

/* DISPLAY STOCHASTIC BACKGROUND SIMULATION PARAMETERS */
printf("\n");
printf("STOCHASTIC GRAVITATIONAL WAVE BACKGROUND SIMULATION\n");
printf("\n");
printf("PARAMETERS:\n");
printf("Simulated stochastic background (0=no,1=yes): %d\n",FAKE_SB);
printf("Simulated detector noise at site 1 (0=no,1=yes): %d\n",FAKE_NOISE1);
printf("Simulated detector noise at site 2 (0=no,1=yes): %d\n",FAKE_NOISE2);
printf("Real-time noise at site 1 (0=no,1=yes): %d\n", REAL_TIME_NOISE1);
printf("Real-time noise at site 2 (0=no,1=yes): %d\n", REAL_TIME_NOISE2);
printf("Detector site 1 = %s\n",site1_name);
printf("Detector site 2 = %s\n",site2_name);
printf("Sampling period = %e seconds\n",DELTA_T);
printf("Number of data points = %d\n",N);
printf("Omega_0 = %e\n",OMEGA_0);
printf("f_low = %e Hz\n",F_LOW);
printf("f_high = %e Hz\n",F_HIGH);
printf("Number of runs (for simulation) = %d\n",NUM_RUNS);
printf("Number of bins (for statistics) = %d\n",NUM_BINS);
printf("\n");

/* CONSTRUCT NOISE POWER (FOR SIGNAL GENERATION), WHITENING FILTER */
/* AND THE OVERLAP REDUCTION FUNCTION */
delta_f=(float)(1.0/(N*DELTA_T));
noise_power(noise1_file,N/2,delta_f,generation_power1);
noise_power(noise2_file,N/2,delta_f,generation_power2);
whiten(whiten1_file,N/2,delta_f,whiten1);
whiten(whiten2_file,N/2,delta_f,whiten2);
overlap(site1_parameters,site2_parameters,N/2,delta_f,gamma12);

/* CONSTRUCT NOISE_POWER (FOR SIGNAL ANALYSIS) IF REAL-TIME NOISE */
/* IS NOT DESIRED */
if (REAL_TIME_NOISE1!=1) {
    for (i=0;i<N/2;i++) analysis_power1[i]=generation_power1[i];
}
if (REAL_TIME_NOISE2!=1) {
```

```
    for (i=0;i<N/2;i++) analysis_power2[i]=generation_power2[i];
}

/* PERFORM THE SIMULATION */
for (i=1;i<=NUM_RUNS;i++) {

    /* SIMULATE STOCHASTIC BACKGROUND AND/OR DETECTOR NOISE, IF DESIRED */
    if (FAKE_SB==1 || FAKE_NOISE1==1 || FAKE_NOISE2==1) {
        monte_carlo(FAKE_SB,FAKE_NOISE1,FAKE_NOISE2,N,DELTA_T,OMEGA_0,
                    F_LOW,F_HIGH,gamma12,
                    generation_power1,generation_power2,
                    whiten1,whiten2,out1,out2,&seed);
    }

    /* TEST DATA TO SEE IF GAUSSIAN */
    previous_test=pass_test;
    pass_test=test_data12(N,out1,out2);

    if (pass_test==1) {

        /* ANALYZE DATA */
        analyze(previous_test,out1,out2,N,DELTA_T,OMEGA_0,F_LOW,F_HIGH,
                gamma12,whiten1,whiten2,
                REAL_TIME_NOISE1,REAL_TIME_NOISE2,
                analysis_power1,analysis_power2,&signal,&variance);

        /* DISPLAY PRELIMINARY STATISTICS */
        prelim_stats(OMEGA_0,N*DELTA_T,signal,variance);

        /* UPDATE RUNS COMPLETED AND STATS ARRAY FOR FINAL STATISTICS */
        runs_completed++;
        stats[runs_completed-1]=signal;
    }
} /* end for (i=1;i<+NUM_RUNS;i++) */

/* FINAL STATISTICS */
printf("\n");
statistics(stats,runs_completed,NUM_BINS);

return 0;
}
```

11.27 Some output from the simulation program

Below is a sample of the output that is produced during the execution of the stochastic background simulation program described in Sec. 11.26. Also shown, in Fig. 81, is a plot of the histogram and best-fit Gaussian probability distribution that were stored in data files by the function `statistics()`. For this particular simulation, the total number of runs was equal to 1271 and the number of bins for the histogram was equal to 200.

```
total number of runs completed=815
total observation time =2.670592e+03 seconds
signal value=2.659629e-03
experimental mean=3.360998e-03
experimental stddev=1.214569e-02
experimental SNR=7.899961e+00
theoretical mean=3.276800e-03
theoretical stddev=1.112916e-02
theoretical SNR=8.405551e+00
relative error in SNR=6 percent
experimental omega_0=1.025695e-03
theoretical omega_0=1.000000e-03
theoretical omega_0 for detection with 95 percent confidence=1.962989e-04
```

```
total number of runs completed=816
total observation time =2.673869e+03 seconds
signal value=-3.592409e-03
experimental mean=3.352476e-03
experimental stddev=1.214068e-02
experimental SNR=7.888017e+00
theoretical mean=3.276800e-03
theoretical stddev=1.112916e-02
theoretical SNR=8.410706e+00
relative error in SNR=6 percent
experimental omega_0=1.023095e-03
theoretical omega_0=1.000000e-03
theoretical omega_0 for detection with 95 percent confidence=1.961785e-04
```

```
total number of runs completed=817
total observation time =2.677146e+03 seconds
signal value=-7.967954e-03
experimental mean=3.338620e-03
experimental stddev=1.213970e-02
experimental SNR=7.860860e+00
theoretical mean=3.276800e-03
theoretical stddev=1.112916e-02
theoretical SNR=8.415858e+00
relative error in SNR=6 percent
experimental omega_0=1.018866e-03
theoretical omega_0=1.000000e-03
theoretical omega_0 for detection with 95 percent confidence=1.960585e-04
```

Data segment 1 failed Gaussian test!

```
total number of runs completed=818
total observation time =2.680422e+03 seconds
signal value=1.447747e-02
experimental mean=3.352238e-03
experimental stddev=1.213852e-02
```

experimental SNR=7.898519e+00
theoretical mean=3.276800e-03
theoretical stddev=1.112916e-02
theoretical SNR=8.421007e+00
relative error in SNR=6 percent
experimental omega_0=1.023022e-03
theoretical omega_0=1.000000e-03
theoretical omega_0 for detection with 95 percent confidence=1.959386e-04

total number of runs completed=819
total observation time =2.683699e+03 seconds
signal value=3.647211e-03
experimental mean=3.352598e-03
experimental stddev=1.213111e-02
experimental SNR=7.909022e+00
theoretical mean=3.276800e-03
theoretical stddev=1.112916e-02
theoretical SNR=8.426153e+00
relative error in SNR=6 percent
experimental omega_0=1.023132e-03
theoretical omega_0=1.000000e-03
theoretical omega_0 for detection with 95 percent confidence=1.958189e-04

total number of runs completed=820
total observation time =2.686976e+03 seconds
signal value=-5.958459e-03
experimental mean=3.341243e-03
experimental stddev=1.212807e-02
experimental SNR=7.889026e+00
theoretical mean=3.276800e-03
theoretical stddev=1.112916e-02
theoretical SNR=8.431295e+00
relative error in SNR=6 percent
experimental omega_0=1.019666e-03
theoretical omega_0=1.000000e-03
theoretical omega_0 for detection with 95 percent confidence=1.956995e-04

total number of runs completed=821
total observation time =2.690253e+03 seconds
signal value=1.057661e-02
experimental mean=3.350056e-03
experimental stddev=1.212331e-02
experimental SNR=7.917764e+00
theoretical mean=3.276800e-03
theoretical stddev=1.112916e-02
theoretical SNR=8.436435e+00
relative error in SNR=6 percent
experimental omega_0=1.022356e-03
theoretical omega_0=1.000000e-03
theoretical omega_0 for detection with 95 percent confidence=1.955803e-04

Data segment 2 failed Gaussian test!

total number of runs completed=822
total observation time =2.693530e+03 seconds
signal value=6.683305e-03
experimental mean=3.354111e-03
experimental stddev=1.211649e-02
experimental SNR=7.936639e+00

```
theoretical mean=3.276800e-03  
theoretical stddev=1.112916e-02  
theoretical SNR=8.441571e+00  
relative error in SNR=5 percent  
experimental omega_0=1.023593e-03  
theoretical omega_0=1.000000e-03  
theoretical omega_0 for detection with 95 percent confidence=1.954613e-04
```

Histogram and Gaussian Probability Distribution

(for the initial LIGO detectors simulation)

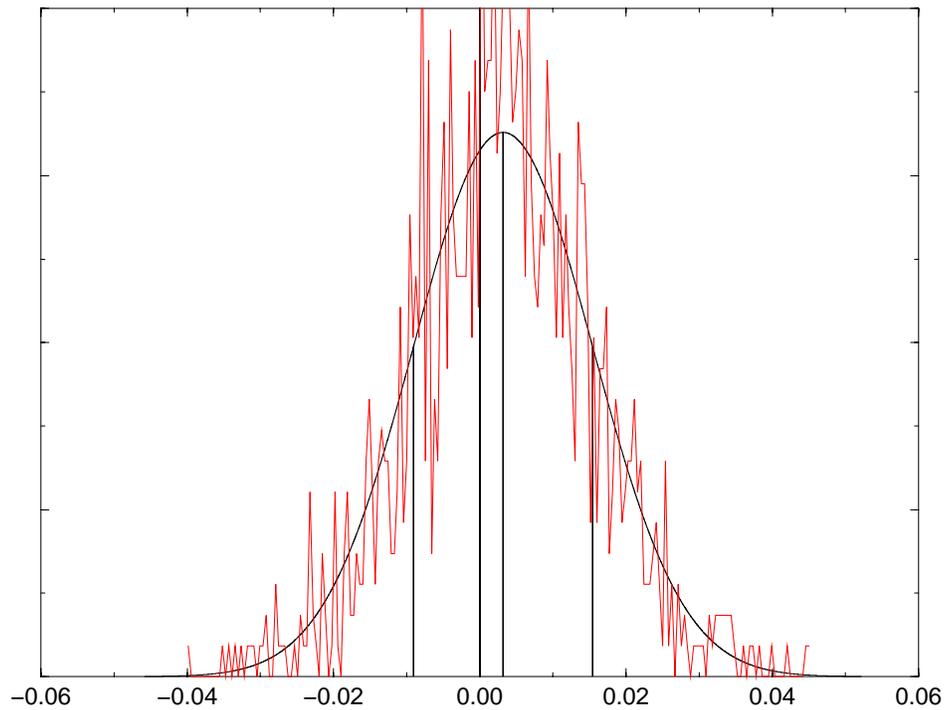


Figure 81: Histogram of the measured cross-correlation signal values, and the corresponding best-fit Gaussian probability distribution for the stochastic background simulation.

12 Galactic Modelling

Large parts of GRASP were developed using data from the Caltech 40-meter prototype detector. This detector is typically sensitive to distances of tens of kiloparsecs, roughly the length scale of our galaxy. It was therefore useful to develop a model of the galactic distributions of expected sources, for example, of the galactic distribution of binary star systems. This section contains a package of routines which can be used to simulate the expected distribution of binary inspiral sources in our galaxy. Because of the earth's motion about its axis and about the sun, the distribution of expected sources (direction, distance, orientation) changes as a function of time, with roughly a 24-hour period. This section contains routines which model the distribution of these parameters as a function of the earth's motion.

12.1 Function: `local_sidereal_time()`

```
float local_sidereal_time(time_t time, float longitude)
```

Returns the local sidereal time, in decimal hours, for a given calendar time and detector longitude. The arguments are:

`time`: Input. The time as an integer number of seconds since 0h 1 January 1970. This number is returned by the time routines in `<time.h>`.

`longitude`: Input. The longitude of the detector in degrees West.

The local sidereal time is calculated as follows. Let JD be the Julian date of 0h on the desired calendar day. (This is computed using the Numerical Recipes routine `julday()`, but a value of 0.5 must be subtracted from this routine to give the JD at 0h rather than at 12h.) The Universal Time, UT, is computed using the `gmtime()` function. The Greenwich Sidereal Time, GST, is

$$\text{GST} = T_0 + 1.002\,737\,909 \times \text{UT}$$

(modulo 24 hours) where

$$T_0 = 6.697\,374\,558 + 2\,400.051\,336 \times T + 0.000\,025\,862 \times T^2$$

and

$$T = \frac{\text{JD} - 2\,451\,545}{36\,525}.$$

The *local* sidereal time is obtained by subtracting the longitude of the detector expressed as decimal hours West of Greenwich.

Author: Jolien Creighton, jolien@tapir.caltech.edu

Comments: This routine is adapted from the method given in: Peter Duffet-Smith *Practical Astronomy with Your Calculator*, 3rd edition, (Cambridge University Press, 1988).

12.2 Example: caltech_lst program

This program is an example of how to use the function `local_sidereal_time()` to compute the local sidereal time at the Caltech 40-meter laboratory. The program determines the present sidereal time at Caltech if it receives no arguments. Instead, the program can receive a single long integer specifying the number of seconds since 0h UTC 1 Jan 1970, and it returns the Caltech sidereal time at that time. Also returned are the number of seconds since 0h UTC 1 Jan 1970, the local and coordinated universal times, and the Greenwich sidereal time. Some examples:

1. caltech_lst

```
881889773 seconds since 0h UTC 1 Jan 1970
17:22:53 PST Thu 11 Dec 1997
01:22:53 UTC Fri 12 Dec 1997
22:53:30 (22.891788 h) Local Sidereal Time
06:46:02 (06.767323 h) Greenwich Sidereal Time
```

(guess when I wrote this example!)

2. caltech_lst 0

```
0 seconds since 0h UTC 1 Jan 1970
16:00:00 PST Wed 31 Dec 1969
00:00:00 UTC Thu 01 Jan 1970
22:48:23 (22.806442 h) Local Sidereal Time
06:40:55 (06.681976 h) Greenwich Sidereal Time
```

The program can be simply modified to correspond to *your* local sidereal time by changing the `#define LON-
GITUDE` command to your local longitude (in degrees West of Greenwich).

author: Jolien Creighton, jolien@tapir.caltech.edu

```
/* GRASP: Copyright 1997,1998 Bruce Allen */
#include "grasp.h"

#define LONGITUDE 118.133 /* degrees West: Caltech */

int main(int argc, char *argv[])
{
    time_t now;
    double lst_h,lst_m,lst_s;
    double gst_h,gst_m,gst_s;
    float lst,gst,longitude=LONGITUDE;
    char loc[128],utc[128];

    if (argc==2) now = (time_t)atoi(argv[1]);
    else time(&now);

    strftime(loc,128,"%H:%M:%S %Z %a %d %b %Y",localtime(&now));
    strftime(utc,128,"%H:%M:%S UTC %a %d %b %Y",gmtime(&now));
    lst = local_sidereal_time(now,longitude);
    gst = local_sidereal_time(now,0);
    lst_s = 60*modf(60*modf(lst,&lst_h),&lst_m);
    gst_s = 60*modf(60*modf(gst,&gst_h),&gst_m);

    printf("%ld seconds since 0h UTC 1 Jan 1970\n%s\n%s\n", (long)now,loc,utc);
    printf("%02u:%02u:%02u (%09.6f h) Local Sidereal Time\n",
           (char)lst_h,(char)lst_m,(char)lst_s,lst);
    printf("%02u:%02u:%02u (%09.6f h) Greenwich Sidereal Time\n",
           (char)gst_h,(char)gst_m,(char)gst_s,gst);

    return 0;
}
```

12.3 Function: galactic_to_equatorial()

```
void galactic_to_equatorial(float l, float b, float *alpha, float *delta)
```

This routine converts the coordinates of an object from the Galactic system—Galactic longitude ℓ and latitude b —to the equatorial system—right ascension α and declination δ . The arguments are:

l: Input. The Galactic longitude ℓ (radians).

b: Input. The Galactic latitude b (radians).

alpha: Output. The right ascension α_{1950} (radians).

delta: Output. The declination δ_{1950} (radians).

The transformation is the following:

$$\delta = \arcsin[\cos b \cos \delta_{\text{NGP}} \sin(\ell - \ell_{\text{ascend}}) + \sin b \sin \delta_{\text{NGP}}] \quad (12.3.1)$$

and

$$\alpha = \arctan \left[\frac{\cos b \cos(\ell - \ell_{\text{ascend}})}{\sin b \cos \delta_{\text{NGP}} - \cos b \sin \delta_{\text{NGP}} \sin(\ell - \ell_{\text{ascend}})} \right] + \alpha_{\text{NGP}} \quad (12.3.2)$$

where

$$\begin{aligned} \alpha_{\text{NGP}} = 192^{\circ}25 & \text{ is the right ascension (1950) of the North Galactic Pole} \\ \delta_{\text{NGP}} = 27^{\circ}4 & \text{ is the declination (1950) of the North Galactic Pole} \\ \ell_{\text{ascend}} = 33^{\circ} & \text{ is the ascending node of the Galactic plane on the equator.} \end{aligned}$$

Author: Jolien Creighton, jolien@tapir.caltech.edu

Comments: This routine is adapted from the method given in: Peter Duffet-Smith *Practical Astronomy with Your Calculator*, 3rd edition, (Cambridge University Press, 1988). The values used in this routine should be updated to epoch 2000 coordinates.

12.4 Example: galactic2equatorial program

This is a simple implementation of the function `galactic_to_equatorial()` that converts a specified Galactic longitude and latitude to right ascension and declination (epoch 1950). The Galactic coordinates are entered in decimal degrees as command line arguments. Some examples:

1. The North Galactic pole is $b = 90^\circ$, or $\alpha_{1950} = 12^{\text{h}}49^{\text{m}}$ and $\delta_{1950} = +27^\circ24'$. The output of the command `galactic2equatorial 0 90` is

```
l   (deg):  0.00
b   (deg): +90.00
RA  (hms): 12 48 59
Dec (dms): +27 23 59
```

2. The Galactic center is $\ell = b = 0$, or $\alpha_{1950} = 17^{\text{h}}42^{\text{m}}24^{\text{s}}$ and $\delta_{1950} = -28^\circ55'$. The output of the command `galactic2equatorial 0 0` is

```
l   (deg):  0.00
b   (deg): +0.00
RA  (hms): 17 42 26
Dec (dms): -28 55 00
```

3. The Large Magellanic Cloud is $\ell = 280^\circ.5$ and $b = -32^\circ.9$, or $\alpha_{2000} = 05^{\text{h}}23^{\text{m}}34^{\text{s}}.598$ and $\delta_{2000} = -69^\circ45'22''.33$. The output of the command `galactic2equatorial 280.5 -32.9` is

```
l   (deg): 280.50
b   (deg): -32.90
RA  (hms): 05 23 48
Dec (dms): -69 49 36
```

These examples show that the output is correct to the accuracy of the input coordinates.

author: Jolien Creighton, jolien@tapir.caltech.edu

```
/* GRASP: Copyright 1997,1998 Bruce Allen */
#include "grasp.h"
#define DEG_TO_RAD ((float)(M_PI/180))
#define RAD_TO_DEG ((float)180/M_PI)
#define RAD_TO_HOUR ((float)12/M_PI)

int main(int argc, char *argv[])
{
    float l,b,alpha,delta;
    double hra,mra,sra,ddec,mdec,sdec;

    if (argc==3) {
        l = atof(argv[1]);
        b = atof(argv[2]);
    } else {
        fprintf(stderr,"usage: %s l b\n",argv[0]);
        fprintf(stderr," l: Galactic longitude (degrees)\n");
        fprintf(stderr," b: Galactic latitude (degrees)\n");
        return 1;
    }

    printf("l (deg): %6.2f\n",l);
    printf("b (deg): %6.2f\n",b);

    l *= DEG_TO_RAD;
    b *= DEG_TO_RAD;

    galactic_to_equatorial(l,b,&alpha,&delta);

    alpha *= RAD_TO_HOUR;
    delta *= RAD_TO_DEG;

    sra = 60*modf(60*modf(alpha,&hra),&mra);
    sdec = 60*modf(fabs(60*modf(delta,&ddec)),&mdec);

    printf("RA (hms): %02u %02u %02u\n", (char)hra, (char)mra, (char)sra);
    printf("Dec (dms): %+02d %02u %02u\n", (int)ddec, (char)mdec, (char)sdec);

    return 0;
}
```

12.5 Function: equatorial_to_horizon()

```
void equatorial_to_horizon(float alpha, float delta, float time, float lat,  
float *azi, float *alt)
```

This routine converts the coordinates of an object from the equatorial system—right ascension α and declination δ —to the horizon system—azimuth A and altitude a —for a given time and latitude. The arguments are:

`alpha`: Input. The right ascension α (radians).

`delta`: Input. The declination δ (radians).

`time`: Input. The time of day (sidereal seconds).

`lat`: Input. The latitude λ North (radians).

`azi`: Output. The azimuth A (radians clockwise from North).

`alt`: Output. The altitude a (radians up from the horizon).

The transformation is the following:

$$a = \arcsin[\sin \delta \sin \lambda + \cos \delta \cos \lambda \cos h] \quad (12.5.1)$$

and

$$A = \arctan \left[\frac{-\cos \delta \cos \lambda \sin h}{\sin \delta - \sin \lambda \sin a} \right] \quad (12.5.2)$$

where λ is the latitude and $h = (\text{local sidereal time}) - \alpha$ is the hour angle.

Author: Jolien Creighton, jolien@tapir.caltech.edu

Comments: This routine is adapted from the method given in: Peter Duffet-Smith *Practical Astronomy with Your Calculator*, 3rd edition, (Cambridge University Press, 1988). I have assumed that the user either (a) has correctly precessed the equatorial coordinates, or (b) doesn't care.

12.6 Function: beam_pattern()

```
void beam_pattern(float theta, float phi, float psi, float *plus, float *cross)
```

This routine computes the beam pattern functions, F_+ and F_\times , for some specified angles θ , ϕ , and ψ . The arguments are:

`theta`: Input. The polar angle θ (radians from zenith).

`phi`: Input. The azimuthal angle ϕ (radians counter-clockwise from the first arm).

`psi`: Input. The polarization angle ψ (radians).

`plus`: Output. The detector response function F_+ .

`cross`: Output. The detector response function F_\times .

The beam pattern functions are calculated according to the following formulae:

$$F_+ = \frac{1}{2}(1 + \cos^2 \theta) \cos 2\phi \cos 2\psi - \cos \theta \sin 2\phi \sin 2\psi \quad (12.6.1)$$

and

$$F_\times = \frac{1}{2}(1 + \cos^2 \theta) \cos 2\phi \sin 2\psi + \cos \theta \sin 2\phi \cos 2\psi. \quad (12.6.2)$$

Author: Jolien Creighton, jolien@tapir.caltech.edu

Comments: The beam pattern formulae, as well as a precise definition of the angles can be found in: Kip Thorne in *300 Years of Gravitation*, S. Hawking and W. Israel editors (Cambridge University Press, 1987). The formulae are suitable for detectors in which the arms are perpendicular; they are not suitable for the GEO-600 site because the opening angle is approximately 94° .

12.7 Function: `mc_chirp()`

```
void mc_chirp(float time, float latitude, float orientation, long *seed,  
             float *invMpc, float *c0, float *c1)
```

This routine makes a random chirp from a hypothetical distribution of Galactic binary neutron stars. The arguments are:

`time`: Input. The time of arrival of the chirp in sidereal seconds.

`latitude`: Input. The latitude of the detector in radians North.

`orientation`: Input. The orientation of the arm of the detector in radians counter clockwise from North.

`seed`: Input/Output. A random number seed for use in the Numerical Recipes random number generator `ran1()`.

`invMpc`: Output. The effective distance of the chirp, i.e., the distance of the binary system if it were optimally oriented.

`c0`: Output. The cosine of the random phase of the chirp.

`c1`: Output. The sine of the random phase of the chirp.

The resulting chirp waveform is $h(t) = \text{invMpc} \times [c0 \times h_c(t) + c1 \times h_s(t)]$ where $h_c(t)$ is the cosine-phase chirp waveform and $h_s(t)$ is the sine-phase chirp waveform normalized to one megaparsec.

The hypothetical number of Galactic binary neutron star systems between the Galactocentric radii R and $R + dR$, and between disk heights z and $z + dz$, is taken to be

$$dN \propto R dR dz e^{-R^2/2R_0^2} e^{-|z|/h_z} \quad (12.7.1)$$

with $R_0 = 4.8$ kpc and $h_z = 1$ kpc. Only the disk population is considered in this calculation. The Galactocentric radius of the Sun is taken to be $R_\odot = 8.5$ kpc.

Author: Jolien Creighton, jolien@tapir.caltech.edu

Comments: The hypothetical distribution was adapted from the distribution used by S. J. Curran and D. R. Lorimer, *Mon. Not. R. Astron. Soc.* **276** 347 (1995). The scale height is a guestimate. Very little is known about the actual distribution of Galactic neutron star binaries! The routine assumes that the arms of the detector are perpendicular; it is not suitable for the GEO-600 site because the opening angle is approximately 94° .

12.8 Example: `inject` program

This program produces a list of random Galactic neutron star-neutron star binary inspiral parameters for injection into the interferometer data to test the data analysis software.

The inspiral waveforms correspond to two neutron star companions, each with a mass distribution that is uniform between two cutoffs given by the parameters `MLO` and `MHI` [for example see L S Finn, Phys Rev Lett **73** 1878 (1994)].

The amplitude distribution corresponds to the (time dependent) model described for the GRASP routine `mc_chirp()`. The initial phase is uniformly distributed. The parameters `LAT`, `LON`, and `ARM` correspond to the latitude, longitude, and arm orientation of the detector, and are required for the model.

The injection time is either at a fixed intervals given by parameter `INV_RATE` (when parameter `FIXED=1`), or at random intervals corresponding to a Poisson process with inverse rate `INV_RATE` (when `FIXED=0`). Injection times are between the start and the end of the data run specified by the environment variable `GRASP_DATAPATH`. The start and end times of this data run are obtained from code resembling that in program `locklist`. If two chirps potentially occur within the same data segment (with length given by parameter `NPOINT`), a warning message is printed.

The results are output to `stdout` in a list containing the arrival time (double), the two masses (floats), the amplitude—inverse Mpc distance (float), and the initial phase (float), separated by spaces. This is the same format as required for the file `insert.ascii` which is read by the `binary_get_data()` routine in the `binary_search` code.

Author: Jolien Creighton, jolien@tapir.caltech.edu

```
/* GRASP: Copyright 1997,1998 Bruce Allen */  
/* Program inject.c
```

Produces a list of random Galactic NS-NS binary inspiral parameters for injection into the interferometer data to test the data analysis software.

The inspiral waveforms correspond to two NS companions, each with a mass distribution that is uniform between two cutoffs MLO and MHI [for example see L S Finn, Phys Rev Lett 73 1878 (1994)].

The amplitude distribution corresponds to the (time dependent) model described for the GRASP routine mc_chirp(). The initial phase is uniformly distributed. The variables LAT, LON, and ARM correspond to the latitude, longitude, and arm orientation of the detector, and are required for the model.

The injection time is either at a fixed intervals INV_RATE (FIXED=1), or at random intervals corresponding to a Poisson process with inverse rate INV_RATE (FIXED=0).

Injection times are between the start and the end of the data run specified by the environment variable GRASP_DATAPATH. The start and end times of this data run are obtained from code resembling that in program locklist.c. If two chirps potentially occur within the same data segment (of length NPOINT points), a warning message is printed.

The results are output to stdout in a list containing the arrival time (double), the two masses (floats), the amplitude—inverse Mpc distance (float), and the initial phase (float), separated by spaces. This is the same format as required for the file insert.ascii which is read by the binary_get_data() routine in the binary search code.

```
*/  
#include "grasp.h"  
  
#define OFFSET 15.0 /* offset in secs of injected chirps arrival times */  
#define SEED -101 /* initial seed value for random #, <0 */  
#define LAT 34.1667 /* detector latitude in degrees North */  
#define LON 118.133 /* detector longitude in degrees West */  
#define ARM 180.0 /* detector arm orientation in degrees CCW from North */  
#define MLO 1.29 /* low NS mass limit in solar masses */  
#define MHI 1.45 /* high NS mass limit in solar masses */  
#define INV_RATE 30.0 /* inverse of event rate in seconds */  
#define FIXED 1 /* 0: Events occur in Poisson-distributed intervals  
1: Events occur at the fixed rate RATE */  
#define NPOINT 262144 /* number of points in each data segment (used to warn  
when two events may exist in the same segment) */  
#define KNOWN_START_END 1 /* use the starting times below rather than from data */  
#define TSTART 784880277  
#define TEND 785388428  
float ran1(long *);  
  
int known_time(double *time, long *seed){  
static int first=1;  
if (first) {  
*time=TSTART+OFFSET;  
first=0;  
}  
else {  
if (FIXED)  
*time+=INV_RATE;
```

```
    else
        *time+=INV_RATE*log(ran1(seed));
    }
    return (*time<TEND);
}

int main()
{
    float local_sidereal_time(time_t, float);
    void mc_chirp(float, float, float, long *, float *, float *, float *);
    int next_time(double *, long *);
    double time;
    long seed=SEED;

#ifdef KNOWN_START_END
    while (known_time(&time,&seed))
#else
    while (next_time(&time,&seed))
#endif
    {
        float lst,m1,m2,c0,c1,phase,invMpc;

        /* compute the local sidereal time in seconds */
        lst = 3600*local_sidereal_time((time_t)time,LON);

        /* obtain the random chirp parameters */
        mc_chirp(lst,LAT,ARM,&seed,&invMpc,&c0,&c1);

        /* the random phase in radians */
        phase = atan2(c1,c0);

        /* the mass of the first and second NSs */
        m1 = MLO + (MHI - MLO)*ran1(&seed);
        m2 = MLO + (MHI - MLO)*ran1(&seed);

        /* print the parameters */
        printf("%f %f %f %f %f\n",time,m1,m2,invMpc,phase);
    }
    return 0;
}

/*
Calculates the next time of a binary inspiral and returns 1 if this time
occurs before the end of the data run or 0 if it does not.
*/
int next_time(double *time, long *seed)
{
    float ran1(long *);

    static int first=1;
    static double end;
    static float srate;
    float dt;

    if (first) { /* obtain the start and the end of the data run on first call */

        time_t begin;
```

```
float st,et,slowrate;
int soff,sblk,eoff,eblk;
FILE *fplock;

first = 0;

/* read ld_mainheader to get begin time */
{
    struct ld_mainheader mh;
    struct ld_binheader bh;
    short *datas=NULL;
    FILE *fpifo;
    int alloc=0,n=0;
    fpifo = grasp_open("GRASP_DATAPATH","channel.0","r");
    read_block(fpifo,&datas,&n,&st,&srate,1,&alloc,1,&bh,&mh);
    fclose(fpifo);
    begin = (time_t)mh.epoch_time_sec;
}
/* start of the first locked segment */
fplock = grasp_open("GRASP_DATAPATH","channel.10","r");
find_locked(fplock,&soff,&sblk,&eoff,&eblock,&st,&et,&slowrate);
*time = (double)(begin) + (double)(soff/srate) + OFFSET;

/* end of the last locked segment */
while (find_locked(fplock,&soff,&sblk,&eoff,&eblock,&st,&et,&slowrate));
fclose(fplock);
end = (double)(begin) + (double)(et + eoff/srate);

/* print the start and end times and the duration of the run */
fprintf(stderr,"start: %f, end: %f, duration: %f\n",*time,end,end - *time);
}
if (FIXED) /* fixed rate intervals */
    dt = INV_RATE;
else /* intervals for a Poisson process */
    dt = -INV_RATE*log(ran1(seed));

if (dt*srate<NPOINT) /* print warning message when chirps too close */
    fprintf(stderr,"Warning: potentially two chirps in same segment\n");

/* increment the time */
*time += dt;

if (*time<end) return 1;
else return 0;
}
```

13 Binary Inspiral Search on November 1994 Data

This section includes and documents the code that was used to perform a binary inspiral search of the Caltech 40-meter data from November 1994. The goal of this work was to place an upper limit on the event rate for binary inspiral in our galaxy. This section includes the following items:

- Source code for the search.
- Source code for generating simulated signals with “galactic” distribution.
- Scripts for running the code on a Beowulf-type system (see www.lsc-group/phys.uwm.edu/~www/docs/beowulf for an example).
- Other related materials.

The results of this search are described in the paper, *Observational limit on gravitational waves from binary neutron stars in the Galaxy*[48], which may be obtained from the following web site:

<http://xxx.lanl.gov/abs/gr-qc/9903108>.

13.1 The Statistical Theory of Reception

The reception process is simply the construction of some statistic of the data followed by some test of the hypotheses “there is a signal present in the data” and “there is no signal present in the data.” When these hypotheses are considered to be exclusive, the reception process reduces to the comparison of the statistic with some pre-assigned threshold. Thus, there are two issues: first, what is the *optimal* statistic to construct, and second, how is the threshold to be determined.

13.1.1 Maximum Likelihood Receiver

Suppose that the detector output, h , contains either noise alone, $h = n$, or both a signal and noise, $h = s + n$. The maximum likelihood receiver returns the quantity

$$\Lambda = \frac{P(h | s)}{P(h | \neg s)} \quad (13.1.1)$$

where $P(h | s)$ is the probability of obtaining the output given that there is a signal present and $P(h | \neg s)$ is the probability of obtaining the output given that there is no signal present. The likelihood ratio Λ can be viewed as the factor which relates the *a priori* probability of a signal being present with the *a posteriori* probability of a signal being present given the detector output:

$$\frac{P(s | h)}{P(\neg s | h)} = \Lambda \frac{P(s)}{P(\neg s)}. \quad (13.1.2)$$

In general, there is no universal way of deciding on the *a priori* probabilities $P(s)$ and $P(\neg s) = 1 - P(s)$, so one is limited to the construction of the likelihood ratio Λ . However, as Λ grows larger, the probability of a signal increases, so we can use it to test our hypotheses as follows:

- If $\Lambda \geq \Lambda_*$ then decide that there is a signal present.
- If $\Lambda < \Lambda_*$ then decide that there is no signal present.

Here, Λ_* is some threshold. Lacking any *a priori* information about whether there is a signal present, the threshold Λ_* should be determined by setting a desired probability for a false alarm and/or false dismissal.

13.1.2 A Receiver for a Known Signal

Consider the case in which the signal has an exactly known form, $s(t)$. In general, signals can occur with a variety of amplitudes; we let $s(t)$ be the known signal with some fiducial normalization and we write the actual signal (if present) as $As(t)$ where A is the amplitude of the signal. Further, we assume that the noise samples are drawn from a stationary Normal distribution, though there may be correlations amongst the noise events (colored noise). The noise correlations may be expressed in terms of the one-sided noise power spectrum, $\frac{1}{2}S_h(|f|)\delta(f - f') = \langle \tilde{n}(f)\tilde{n}^*(f') \rangle$, where $\tilde{n}(f)$ is the Fourier transform of the noise $n(t)$, and $*$ denotes complex conjugation. The probability of obtaining an instant of noise, $n(t)$, is $p(n) \propto \exp[-\frac{1}{2}(n | n)]$, where the inner product $(\cdot | \cdot)$ is given by

$$(a | b) = \int_{-\infty}^{\infty} df \frac{\tilde{a}^*(f)\tilde{b}(f) + \tilde{a}(f)\tilde{b}^*(f)}{S_h(|f|)}. \quad (13.1.3)$$

The likelihood ratio is the ratio of the probabilities $P(h | s) = p(h - As)$ (since $h(t) = As(t) + n(t)$ given the signal is present) and $P(h | \neg s) = p(h)$:

$$\Lambda = e^{Ax - A^2\sigma^2/2} \quad (13.1.4)$$

where $x = (h | s)$ and $\sigma^2 = (s | s)$. Since the likelihood ratio is a monotonically increasing function of x , and since the output h appears only in the construction of x , we can set a threshold on the value of x rather than on Λ .

It is straightforward to compute the false alarm and true detection probabilities for any choice of threshold x_* . Under the assumption that there is no signal present, the false alarm probability is the probability that $|x| > x_*$:

$$P(\text{false alarm}) = P(|x| > x_* | \neg s) = \text{erfc}[x_*/\sqrt{(2\sigma^2)}]. \quad (13.1.5)$$

Notice that we have used the absolute value of x since the actual signal may have either positive or negative amplitude. Similarly, the true detection probability (the converse of the false dismissal probability) is the probability that $|x| > x_*$ when a signal is present:

$$\begin{aligned} P(\text{true detection}) &= P(|x| > x_* | s) \\ &= \frac{1}{2} \text{erfc}[(x_* - A\sigma^2)/\sqrt{(2\sigma^2)}] + \frac{1}{2} \text{erfc}[(x_* + A\sigma^2)/\sqrt{(2\sigma^2)}]. \end{aligned} \quad (13.1.6)$$

Here, the complementary error function is defined by $\text{erfc}(x) = (2/\sqrt{\pi}) \int_x^\infty e^{-t^2} dt$. Using these equations, the threshold can be set for any desired false alarm probability, and then the probability of a true detection can be computed.

13.1.3 A Receiver for a Signal of Unknown Phase

Suppose that the signal is known up to an arbitrary phase, $As(t) = A \cos \theta \times s_0(t) + A \sin \theta \times s_1(t)$ where $s_0(t)$ and $s_1(t)$ are known waveforms and θ is the unknown phase. For simplicity, suppose further that $(s_0 | s_0) = (s_1 | s_1) = \sigma^2$ and $(s_0 | s_1) = 0$, i.e., the waveforms are orthogonal. Again assume that the noise is a stationary (but colored) normal process. Then the likelihood ratio, averaged over a uniform distribution of the unknown phase, is

$$\bar{\Lambda} = e^{-A^2\sigma^2/2} I_0(Az) \quad (13.1.7)$$

where $z^2 = x^2 + y^2$ with $x = (h | s_0)$ and $y = (h | s_1)$. Here, $I_0(x) = (2\pi)^{-1} \int_0^{2\pi} e^{x \sin \theta} d\theta$ is the modified Bessel function of the first kind of order zero. Since this function is monotonic, a threshold level can be set on the value of the statistic z rather than $\bar{\Lambda}$.

When there is no signal present, the statistic z assumes the Rayleigh distribution

$$p(z | \neg s) = \frac{z}{\sigma^2} \exp[-z^2/2\sigma^2] \quad (13.1.8)$$

and, in the presence of a signal, the statistic assumes a Rice distribution

$$p(z | s) = \frac{z}{\sigma^2} \exp[-(z^2/\sigma^2 + A^2\sigma^2)/2] I_0(Az). \quad (13.1.9)$$

Thus, the false alarm probability for a threshold of z_* is

$$P(\text{false alarm}) = P(z > z_* | \neg s) = \exp(-z_*^2/2\sigma^2) \quad (13.1.10)$$

and the true detection probability is

$$P(\text{true detection}) = P(z > z_* | s) = Q(A\sigma, z_*/\sigma). \quad (13.1.11)$$

The Q -function, which is defined by the integral

$$Q(\alpha, \beta) = \int_\beta^\infty dx x e^{-(x^2+\alpha^2)/2} I_0(\alpha x), \quad (13.1.12)$$

has the properties $Q(\alpha, 0) = 1$, $Q(0, \beta) = e^{-\beta^2/2}$, and the asymptotic forms

$$\begin{aligned} Q(\alpha, \beta) &\sim 1 - \frac{1}{\alpha - \beta} \sqrt{\frac{\beta}{2\pi\alpha}} e^{-(\alpha - \beta)^2/2} \quad \alpha \gg \beta \gg 1, \\ Q(\alpha, \beta) &\sim \frac{1}{\beta - \alpha} \sqrt{\frac{\beta}{2\pi\alpha}} e^{-(\beta - \alpha)^2/2} \quad \beta \gg \alpha \gg 1. \end{aligned} \tag{13.1.13}$$

For any desired value of the false alarm probability, we can calculate the threshold $z_* = \sqrt{[-2\sigma \log P(\text{false alarm})]}$. Then the probability of true detection can be evaluated for that threshold using the Q -function.

13.1.4 Reception of a Signal with Unknown Arrival Time

In general, the expected signals will be much shorter than the observation time, and we do not know when the signal will occur. We wish not only to detect these signals but also to measure their arrival time. To do this, we construct the time series

$$x(t) = \int_{-\infty}^{\infty} df e^{-2\pi ift} \frac{\tilde{h}^*(f)\tilde{s}_0(f) + \tilde{h}(f)\tilde{s}_0^*(f)}{S_h(|f|)} \tag{13.1.14}$$

and

$$y(t) = \int_{-\infty}^{\infty} df e^{-2\pi ift} \frac{\tilde{h}^*(f)\tilde{s}_1(f) + \tilde{h}(f)\tilde{s}_1^*(f)}{S_h(|f|)} \tag{13.1.15}$$

for the case of an unknown phase, or just the quantity $x(t)$ with $s_0(t) = s(t)$ if the signal is known completely (up to the arrival time). For each observation period, we compute the mode statistic,

$$\rho = \sigma^{-1} \max_t \begin{cases} |x(t)| & \text{known phase} \\ z(t) = \sqrt{x^2(t) + y^2(t)} & \text{unknown phase,} \end{cases} \tag{13.1.16}$$

and set some threshold ρ_* for this statistic.

In general, it is difficult to compute the false alarm probability for a given threshold because the values of $x(t)$ and $y(t)$ are correlated. However, in the limit of large thresholds (small false alarm probability for a large observation time), each sample $x(t_i)$ and $y(t_i)$ becomes effectively independent in the sense that if the threshold is exceeded at a time t_i , then it is unlikely that it will be exceeded again at any time within the correlation timescale of time t_i . The false alarm probability is then the probability computed for a single time receiver (above) multiplied by the total number of samples in the observation time. Stated another way, the *rate* of false alarms is

$$\nu(\text{false alarm}) \simeq \Delta^{-1} \begin{cases} \text{erfc}(\rho_*/\sqrt{2}) & \text{known phase} \\ \exp(-\rho_*^2/2) & \text{unknown phase} \end{cases} \quad (\rho_* \gg 1) \tag{13.1.17}$$

where Δ^{-1} is the sampling rate. The probability of a false alarm in the observation time T is $P(\text{false alarm}) = T \times \nu(\text{false alarm})$. Although this false alarm rate is good only in the limit of high thresholds, it provides an overestimate for more modest thresholds.

The true detection probability is even more difficult to compute than the false alarm probability. However, a quick estimate would be to assume that the detection will always be made at the correct time, and then the correct detection probability will be the same as was computed in the previous two sections. This will provide an underestimate of the correct detection probability.

13.1.5 Reception of a Signal with Additional Unknown Parameters

In the case that the signal waveform possesses parameters, other than the time of arrival that we wish to measure, then we must consider a *bank* of filters, $\{x_i(t; \lambda_i)\}$ (and $\{y_i(t; \lambda_i)\}$ if there is also an unknown phase), corresponding to a correlation of the output $h(t)$ with all of the possible signals $s(t; \lambda_i)$. Here, the set of N_λ parameters is $\{\lambda_i\}$. A set of statistics $\{\rho_i\}$ is then created according to the methods given in the previous section. The largest of these statistics is compared to the threshold to determine if a signal is present or not. If it is decided that a signal is present, then the estimate of the parameters of the signal are those parameters λ_i corresponding to the largest statistic ρ_i .

The filters in the filter bank will typically be densely packed into the parameter space in order to accurately match any expected signal. Thus, these filters will likely be highly correlated with one another. To estimate the rate of false alarms, we appeal to the high threshold limit in which each filter is effectively independent. Then, the rate of false alarms is

$$\nu(\text{false alarm}) \simeq N_\lambda \Delta^{-1} \begin{cases} \text{erfc}(\rho_*/\sqrt{2}) & \text{known phase} \\ \exp(-\rho_*^2/2) & \text{unknown phase} \end{cases} \quad (\rho_* \gg 1). \quad (13.1.18)$$

Similarly, we assume that if a signal is detected, then it will be detected with the correct parameters. The probability of true detection under this assumption is

$$P(\text{true detection}) \simeq \begin{cases} \frac{1}{2} \text{erfc}[(\rho_* - A\sigma)/\sqrt{2}] + \frac{1}{2} \text{erfc}[(\rho_* + A\sigma)/\sqrt{2}] & \text{known phase} \\ Q(A\sigma, \rho) & \text{unknown phase.} \end{cases} \quad (13.1.19)$$

These expressions provide an overestimate of the false alarm rate and an underestimate of the probability of true detection; thus, they may be used to set a threshold for an upper limit to the desired false alarm rate, and the false dismissal probability should be no greater than the value computed from that threshold.

13.2 Details of Normalization

The Fourier transform conventions of Numerical Recipes are used here. In particular, suppose that the time series $a(t)$ is sampled at intervals of $\Delta t = 1/\text{srate}$ and these samples are stored in the array `array[0..n-1]` where n is the number of samples taken (thus, the total observation time is $T = n\Delta t$). Then, the Fourier transform $\tilde{a}(f) = \int dt e^{2\pi ift} a(t)$ is related to the FFT of `array` by

$$\tilde{a}(f_j) = \Delta t \times (\text{atilde}[2j] + i \text{atilde}[2j+1]) \quad (13.2.1)$$

where `atilde[0..n-1]` is produced from `array[0..n-1]` by `realft(...,1)`. Note that the frequency $f_j = j\Delta f$ where $\Delta f = (n\Delta t)^{-1} = \text{srate}/n$. Define the one-sided mean power spectrum of $a(t)$ by

$$S_a(|f|) = \frac{2}{T} \langle |\tilde{a}(f)|^2 \rangle \quad (13.2.2)$$

and similarly define

$$\text{power}[j] = 2 \langle (\text{atilde}[2j])^2 + (\text{atilde}[2j+1])^2 \rangle. \quad (13.2.3)$$

Notice that $S_a(|f|)$ has dimensions of time while `power[j]` is, of course, dimensionless. These two power spectra are related by

$$S_a(f_j) = \frac{1}{n \times \text{srate}} \text{power}[j]. \quad (13.2.4)$$

A well known “feature” of the inverse FFT produced by `realft(...,-1)` is that

$$\text{realft}(\text{atilde}, n, -1) \Rightarrow \frac{n}{2} \times \text{array}[0..n-1]. \quad (13.2.5)$$

We can express the correlation between two time series, $a(t)$ and $b(t)$, weighted by twice the inverse power spectrum, as

$$\begin{aligned} c(t_k) &= \frac{1}{2} \int_{-\infty}^{\infty} df e^{-2\pi ift_k} \tilde{a}(f) \tilde{b}^*(f) \frac{2}{S(|f|)} \\ &\simeq \frac{1}{2} \frac{\text{srate}}{n} \left\{ \sum_{j=0}^{n/2-1} e^{-2\pi ijk/n} \right. \\ &\quad \times \frac{\text{atilde}[2j] + i \text{atilde}[2j+1]}{\text{srate}} \\ &\quad \times \frac{\text{btilde}[2j] - i \text{btilde}[2j+1]}{\text{srate}} \\ &\quad \left. \times \frac{2 \times n \times \text{srate}}{\text{power}[j]} \right\} + \text{cc} \\ &\Leftarrow \text{correlate}(c, \text{atilde}, \text{btilde}, \text{weight}, n) \end{aligned} \quad (13.2.6)$$

[cf. equation (6.16.1)] where `weight[j] = 2/power[j]` and $c(t_k) = c[k]$. Notice that all the factors of 2, n , and `srate` are accounted for. However, the correlation defined in the first line is one half of the correlation defined in equation (13.1.14).

13.3 Function: `strain_spec()`

```
void strain_spec(float flo, float srate, int n, float *adc_spec,  
float *response, float *mean_pow_spec, float *twice_inv_noise)
```

This routine computes the strain spectrum $\alpha S_h(f)$ and twice the inverse noise $2/(\alpha S_h(f))$ from the ADC spectrum $S_{\text{ADC}}(f)$ and the strain response function $R(f)/\ell$. The arguments are:

`flo`: Input. The low frequency cutoff (Hz) for computing the strain spectrum and twice inverse noise. These are set to zero for frequencies below `flo`.

`srate`: Input. The sampling rate in Hertz.

`n`: Input. Sets the size of the following arrays.

`adc_spec`: Input. The vector `adc_spec[0..n/2]` of the ADC power spectrum $S_{\text{ADC}}(f)$.

`response`: Input. The vector `response[0..n+1]` of the detector *strain* response function $R(f)/\ell$. Here, ℓ is the armlength of the detector, so that `response[]` has dimensions of (ADC counts)⁻¹.

`mean_pow_spec`: Output. The vector `mean_pow_spec[0..n/2]` containing the strain power spectrum $\alpha S_h(f)$.

`twice_inv_noise`: Output. The vector `twice_inv_noise[0..n/2]` containing twice the inverse strain noise power spectrum $2/(\alpha S_h(f))$.

The strain power spectrum is $S_h(f) = |R(f)/\ell|^2 \times S_{\text{ADC}}(f)$, and the normalization factor $\alpha = n \times \text{srate}$ is present for agreement with the output of the routine `avg_inv_spec()`.

Author: Jolien Creighton, jolien@tapir.caltech.edu

Comments: Note that the strain power spectrum differs from $S_h(f)$ by the low frequency cutoff, and by the factor α .

13.4 Function: `corr_coef()`

```
void corr_coef(float *a0, float *a1, float *r, int n,  
              float *r00, float *r11, float *r01)
```

This routine computes the correlation coefficients, r_{00} , r_{11} , and r_{01} , of two vectors of data, a_0 and a_1 :

$$r_{ij} = \begin{bmatrix} (a_0, a_0) & (a_1, a_0) \\ (a_0, a_1) & (a_1, a_1) \end{bmatrix}. \quad (13.4.1)$$

The arguments are:

`a0`: Input. The vector $\tilde{a}_0(f)$.

`a1`: Input. The vector $\tilde{a}_1(f)$.

`r`: Input. Twice the inverse noise $2/(\alpha S_h(f))$, as returned by, e.g., `strain_spec()`.

`n`: Input. The length of the arrays `a0[1..n-1]`, `a1[1..n-1]`, and `r[0..n/2]`.

`r00`: Output. The correlation coefficient $r_{00} = (a_0, a_0)$.

`r00`: Output. The correlation coefficient $r_{11} = (a_1, a_1)$.

`r00`: Output. The correlation coefficient $r_{01} = (a_0, a_1)$.

Author: Jolien Creighton, jolien@tapir.caltech.edu

Comments: The inner product (a_0, a_1) is equal to the value `c[0]` of the output of `correlate(c, a0, a1, r, n)`; thus, it differs by a factor of two from the Cutler and Flanagan inner product. The constant $\alpha = n \times \text{srate}$ is explained in section 13.3.

13.5 Function: receiver1()

```
int receiver1(float *input, float *filter, float *twice_inv_noise, float var,  
             float *output, int n, int presafe, int postsafe, float threshold,  
             float *snr, int *ind)
```

This function computes the maximum signal-to-noise ratio ρ for a matched filter of known form (including phase), but unknown arrival time, and returns the number of times in the data segment that the signal-to-noise ratio exceeded some specified threshold ρ_* . The arguments are:

input: Input. The vector `input[0..n-1]` containing the input data $\tilde{h}(f)$, in frequency domain, to be filtered.

signal: Input. The vector `signal[0..n-1]` containing the expected waveform $\tilde{s}(f)$, in frequency domain.

twice_inv_noise: Input. The vector `twice_inv_noise[0..n/2]`, containing twice the inverse power spectrum $2/(\alpha S_h(f))$ of the noise.

var: Input. The “variance,” (s, s) , of the expected waveform.

output: Output. The vector `output[0..n-1]` corresponding to the result of correlating the input with the expected waveform. This quantity is computed by the call:
`correlate(output, input, signal, twice_inv_noise, n).`

n: Input. The integer that defines the lengths of the previous arrays.

presafe: Input. The number of points to skip at the beginning of the correlation in order to avoid wrap-around errors.

postsafe: Input. The number of points to skip at the end of the correlation in order to avoid wrap-around errors. This should be longer than the length of the signal $s(t)$ in the time domain.

threshold: Input. The signal-to-noise ratio threshold ρ_* used in counting the number of times that the filter output exceeded ρ_* .

snr: Output. The vector `snr[0..n-1]` corresponding to the signal-to-noise ratios

$$\text{snr}[i] = |\text{output}[i]| \times \sqrt{2/\text{var}}.$$

ind: Output. A table of indices `ind[0..n-presafe-postsafe-1]` giving the offsets between `presafe` and `n - postsafe` of the signal-to-noise ratios sorted into decreasing order. Thus, `snr[ind[0]]` is the largest signal-to-noise ratio (between the pre- and post-safety margins), `snr[ind[1]]` is the second largest, etc.

The number of threshold crossings returned is the number of times that the threshold signal-to-noise ratio is exceeded between the pre- and post-safety margins.

Author: Jolien Creighton, jolien@tapir.caltech.edu

Comments: The constant $\alpha = n \times \text{srate}$ is explained in section 13.3. The extra factor of $\sqrt{2}$ in the signal-to-noise ratio arises from the factor of two difference between the inner product of Cutler and Flanagan and the value of `output[0]`.

13.6 Function: receiver2()

```
int receiver2(float *input, float *filter0, float *filter1,
             float *twice_inv_noise, float var,
             float *output0, float *output1, int n, int presafe, int postsafe,
             float threshold, float *snrsq, int *ind)
```

This function computes the maximum signal-to-noise ratio ρ for a matched filter of known form but unknown phase and arrival time, and returns the number of times in the data segment that the signal-to-noise ratio exceeded some specified threshold ρ_* . The arguments are:

input: Input. The vector `input[0..n-1]` containing the input data $\tilde{h}(f)$, in frequency domain, to be filtered.

signal0: Input. The vector `signal0[0..n-1]` containing the expected waveform $\tilde{s}_0(f)$, in frequency domain.

signal1: Input. The vector `signal1[0..n-1]` containing the expected waveform $\tilde{s}_1(f)$, in frequency domain.

twice_inv_noise: Input. The vector `twice_inv_noise[0..n/2]`, containing twice the inverse power spectrum $2/(\alpha S_h(f))$ of the noise.

var: Input. The “variance,” $\frac{1}{2}(r_{00} + r_{11}) = \frac{1}{2}[(s_0, s_0) + (s_1, s_1)]$, of the expected waveform.

output0: Output. The vector `output0[0..n-1]` corresponding to the result of correlating the input with the expected waveform $s_0(t)$. This quantity is computed by the call:
`correlate(output0, input, signal0, twice_inv_noise, n)`.

output1: Output. The vector `output1[0..n-1]` corresponding to the result of correlating the input with the expected waveform $s_1(t)$. This quantity is computed by the call:
`correlate(output1, input, signal1, twice_inv_noise, n)`.

n: Input. The integer that defines the lengths of the previous arrays.

presafe: Input. The number of points to skip at the beginning of the correlation in order to avoid wrap-around errors.

postsafe: Input. The number of points to skip at the end of the correlation in order to avoid wrap-around errors. This should be longer than the length of the signal $s(t)$ in the time domain.

threshold: Input. The signal-to-noise ratio threshold ρ_* used in counting the number of times that the filter output exceeded ρ_* .

snrsq: Output. The vector `snrsq[0..n-1]` corresponding to the squared signal-to-noise ratios

$$\text{snrsq}[i] = 2 \times \frac{(\text{output0}[i])^2 + (\text{output1}[i])^2}{\text{var}}$$

ind: Output. A table of indices `ind[0..n-presafe-postsafe-1]` giving the offsets between `presafe` and `n - postsafe` of the signal-to-noise ratios sorted into decreasing order. Thus, `snrsq[ind[0]]` is the largest squared signal-to-noise ratio (between the pre- and post-safety margins), `snr[ind[1]]` is the second largest, etc.

The peak signal-to-noise ratio, $\sqrt{\text{snrsq}[\text{ind}[0]]}$, is the signal-to-noise ratio of the maximum output of a matched filter corresponding to the measured signal $\hat{s}(t) = c_0 s_0(t) + c_1 s_1(t)$ where

$$\begin{aligned} c_0 &= \frac{r_{11} \times \text{output0}[\text{ind}[0]] - r_{01} \times \text{output1}[\text{ind}[0]]}{r_{00}r_{11} - r_{01}r_{01}} \\ c_1 &= \frac{r_{00} \times \text{output1}[\text{ind}[0]] - r_{01} \times \text{output0}[\text{ind}[0]]}{r_{00}r_{11} - r_{01}r_{01}} \end{aligned} \tag{13.6.1}$$

The number of threshold crossings returned is the number of times that the threshold signal-to-noise ratio is exceeded between the pre- and post-safety margins.

Author: Jolien Creighton, jolien@tapir.caltech.edu

Comments: The constant $\alpha = n \times \text{srate}$ is explained in section 13.3. The extra factor of $\sqrt{2}$ in the signal-to-noise ratio arises from the factor of two difference between the inner product of Cutler and Flanagan and the value of, e.g., `output0[0]`.

13.7 Example: `binary_search` program

This program was used to filter the November 1994 Caltech 40-meter data run in search of binary neutron star inspirals.

The program is a modified version of the `multifilter` program. The template bank used is stored in the file `templates.ascii`, which must be generated prior to running the program. Each line of `templates.ascii` contains the masses (separated by a space) of a template. (The file `templates.ascii` generated by `make_mesh`, section 9.39, may be used for this purpose.) The program also reads a file `insert.ascii` if binary inspirals are to be injected. (This file and the injection procedure is described below.) If injection occurs, the program logs the injections in the file `insert.log`. The output of the program is written in binary files in the directory set by the environment variable `GRASP_MFOUT`. The file `signal.header` contains a header file which describes the format of the binary files (including itself), the number of filters used and the filter parameters, the low frequency cutoff used in the filters, and the sampling rate of the IFO. The other files, `signal.00000`, `signal.00001`, etc., contain the output signals for each filter for the segments processed. (The number in the filename indicates the segment.) These files are read using the program `binary_reader`, described in Section 13.9.

At present, there are eight signals generated for each filter. These are:

1. The distance, in Mpc, at which an optimally oriented inspiral would produce a signal-to-noise ratio of one. This describes the sensitivity of the instrument at that given time, and allows one to estimate the distance of a potential signal given its signal-to-noise ratio. An optimally-oriented inspiral is one for which the orbital plane of the inspiral is parallel to the orbital plane defined by the two arms of the detector.
2. The maximum signal-to-noise ratio output by the filter for the data segment. The maximization is made over all possible filter-output offsets in the data segment, but a number of points `PRESAFETY` are omitted from the beginning of the segment and a number `POSTSAFETY` from the end.
3. The maximum signal-to-noise ratio renormalized by a power correction factor. If the estimated power spectrum were wrong by some overall normalization constant, this corrected signal-to-noise ratio represents what would have been obtained if the “correct” power spectrum had been used.
4. The maximum signal-to-noise ratio renormalized by a median correction factor. This factor is the ratio of the expected median of the signal-to-noise ratio distribution (for stationary Gaussian noise) to the median of the observed distribution (which is obtained from every fourth time offset in the time series within the pre- and post-safety margins).
5. The “impulse offset”: the offset at which the filter would peak if it were triggered by an impulse. For an inspiral waveform, this is roughly the offset of the waveform end (modulo the length of the data segment). Thus:

$$\text{impulse offset} = (\text{filter peak offset} + \text{filter length}) \bmod (\text{segment length}). \quad (13.7.1)$$

Alternatively with the macro `COMPARE` set to one, the filter peak offset can be recorded. The difference between the peak and impulse offsets is discussed in Section 6.19. Filter peak offsets are recorded as negative integers so that the reader program can identify the offset as either the impulse offset or as the filter peak offset.

6. The estimated initial phase of the potential inspiral waveform. The phase is

$$\text{phase} = \arctan(\pi/2\text{-phase coefficient}, 0\text{-phase coefficient}).$$

7. The value of the r^2 discriminant described above. This test is only applied if the signal-to-noise ratio exceeds a value of `THRESHOLD`; otherwise, the value of this signal is zero.
8. The number of offsets in which the filter signal-to-noise ratio exceeded the threshold `THRESHOLD`.

In addition to these signals (which are given for each filter), each signal file also contains the time of the start of the data segment (in seconds since 0h 1 January 1970 UTC), and an integer which indicates whether the segment contains significant numbers of outliers. Note that (as described in Section 4.1) these time-stamps have only a few minutes of accuracy. This is good enough to determine the relative orientation of the galaxy and the detector, in order to determine the expected rates of inspiral from galactic source distributions, but may not be good enough for pulsar searches.

The program is divided into the following files:

`binary_params.h`: a header file containing the parameter definitions

`binary_string.h`: a header file containing declarations for the strings `*comment` and `*description`

`binary_search.c`: the main code for analyzing the data and outputting an event list

`binary_get_data.c`: the routines used for data acquisition

`binary_routines.c`: extra miscellaneous routines (which will eventually be documented in the GRASP manual)

`binary_params.ascii`: the file `binary_params.h`, converted into strings, to be included into the string `*description`

Some of these files are described below.

Authors: Bruce Allen (ballen@dirac.phys.uwm.edu), Patrick Brady (patrick@tapir.caltech.edu), Jolien Creighton (jolien@tapir.caltech.edu)

13.7.1 Environment variables used by `binary_search`

The `binary_search` program reads and uses the following environment variables:

- `GRASP_DATAPATH` The directory path in which to search for “old-format” data.
- `GRASP_MFPATH` The directory path in which to write `signal.*` output files and other output.
- `GRASP_INSERT` The directory path in which to search for an `insert.ascii` file, and in which to write an `insert.log` file. Documentation about how to create the file `insert.ascii` and the details of its format may be found in Section 12.8.
- `GRASP_STARTSEGMENT` The segment number at which to start analyzing the data set. Set to ≤ 0 to analyze all data. This environment variable is particularly useful if a run has not terminated properly. It can be used to re-start the analysis run at the point where it previously finished.
- `GRASP_TEMPLATE` The directory in which to search for a `templates.ascii` file containing the template list. To find the code which can be used to generate this file, and documentation about its format, please see Section 9.39.
- `GRASP_KILLSCRIPT` The directory in which to write the file `killscript`. This file contains a `csh` script which will terminate all the `binary_search` processes on a multiprocessor machine or network. It’s useful for “cleaning up the damage” when a job does not terminate correctly and leaves hanging processes.

13.7.2 File: `binary_params.h`

```
/* GRASP: Copyright 1997,1998 Bruce Allen */
/* Parameters that describe the size of the segments of data analyzed */
#define NPOINT 262144 /* The size of our segments of data (~26 seconds) */
#define CHIRPLEN 24000 /* length of longest allowed chirp 23293 points */
#define PRESAFETY 65536 /* ignore PRESAFETY points at beginning of correlation */
#define POSTSAFETY (PRESAFETY + CHIRPLEN) /* ignore POSTSAFETY at end of correlation */
#define SPEC_TRUNC PRESAFETY
/* #define SPEC_TRUNC 0 */

/* Parameters that describe the data acquisition */
#define MIN_INT0_LOCK 3.0 /* Number of minutes to skip into each locked section */
#define DATA_SEGMENTS 16384 /* largest number of data segments to process */

/* Parameter that defines format of data being used */
/* 0: old format data */
/* 1: new (frame) format data */
/* 2: simulated data */
#define DATA_FORMAT 1
#define RANDOMIZE 0 /* Randomizes the phases */

/* Parameters that describe the interferometer */
#define FLO 120.0 /* The low frequency cutoff for filtering (Hz) */
#define HSCALE 1.e21 /* A convenient scaling factor; results independent of it */
#define SRATE 9868.4208984375 /* sample rate in HZ of IFO_DMRO channel */

/* Parameters that describe the filter bank and the filter signals */
#define NUM_TEMPLATES 687 /* number of templates read from an ascii file */
#define STORE_TEMPLATES 0 /* 0: slaves recompute templates. 1: slaves save templates. */
#define NSIGNALS 8 /* number of signal values computed for each template */
#define THRESHOLD 5.0 /* threshold above which a splitup veto test is done */
#define PBINS 20 /* Number of frequency bands for the r^2 discriminator */

/* Parameters controlling optional MPI and MPE code */
#define DEBUG_COMM 0 /* print lots of statements to help debug communication */
#define DBX 0 /* start up dbx windows for each process */
#define ISMPE 0 /* 0: no mpe calls. 1: mpe calls */
#define SMALL_MPELOG 1 /* 0: detailed MPE logging. 1: minimal MPE logging */
#define RENICE 1 /* Priority. 1: processes run NICED 0: not NICED */
#define NPERSLAVE 15 /* The number of segments analyzed in parallel by slave */

/* Parameters for analysing a particular segment and a particular template */
#define PART_TEMPLATE -1 /* template to be investigated (numbered 0,1,...) */
#define COMPARE 0 /* 0: records impulse offset. 1: records peak offset */

/* parameters for type of filtering */
#define INSERT_CHIRP 0 /* 0: don't insert a chirp. 1: insert chirp into data */
#define REMOVE_LINE_BINS 0 /* implement Whitcomb's idea of dropping bins near lines */
#define INJECT_TIME_REVERSE 0 /* inject time-reversed chirps */
#define REVERSE_FILTERS 0 /* analyze data with time-reversed filters */
#define WINDOW 2 /* power spectrum: 0=rectangular 1=Hann 2=Welch 3=Bartlett */
#define NORM_CF 1 /* 0: GRASP snr/distance norm 1: Cutler/Flanagan norms */
#define TWO_PHASE_R2 1 /* 1: Use two-phase r^2 test, 0: use single-phase test */

/* Utility parameters */
#define HEADER_COMMENT_SIZE 16384 /* bytes in the comment part of signal.header file */
```

These parameters are defined by comment text within the file itself. However a few additional comments about these parameters may be helpful.

- `NPOINT` This is a power of two, used to define the segment lengths used in the filtering process.
- `CHIRPLEN` The length of the longest chirp, in samples
- `PRESAFETY` and `POSTSAFETY` should both be set reasonably large. Although the impulse response of the instrument only lasts a few milliseconds (see Section 3.15) the convolution with (the time-domain version of) $\frac{1}{S_h(f)}$ in the optimal filter can stretch this out considerably. When convolved with the wraparound discontinuity at the ends of the segment, this can produce large spurious signals near the beginning and end of a segment. For this reason, we ignore the first `PRESAFETY` points and the final `POSTSAFETY` points at the output of each filter. This eliminates end effects from the wraparound point (the start/end of each segment) arising from the impulse response of the filters.
- `SPEC_TRUNC` This truncates the time-domain version of $1/S_h(f)$ to a pre-determined length, guaranteeing that the impulse response in the time domain is finite and not as long as the data segments. The impulse response is $2 \times \text{SPEC_TRUNC}$ points long (where our counting includes both positive and negative lags).
- `MIN_INTLO` of about three minutes seems to be enough to allow the different pendulum and violin modes to quiet down after coming into lock.
- `DATA_SEGMENTS` The maximum number of segments of data to filter.
- Defining `DATA_FORMAT` to be either 0, 1 or 2 permits one to use this code with either old-format (0) as described in Section 3 or `FRAME`-format data (1) as described in Section 4 or simulated data (2).
- Defining `RANDOMIZE` to be either 0 or 1 determines if the data set (not including an injected signal) has its phase in the frequency domain set to a random value between 0 and 2π , independently in each frequency bin. For 0, no randomization is done. For 1, each bin is randomized in phase. This creates a signal with pseudo-stationary characteristics but the same spectrum as the true signal.
- `FLO` is the gravity wave frequency at which the inspiral chirps begin, and also serves as the low-frequency cutoff for filtering. It should be set well below the most sensitive frequency of the detector, in the seismic- or thermal-noise dominated region.
- `HSCALE` is a constant used to rescale the strain internally in the code to make quantities of interest be of order unity. All results of the code are (and should be) independent of its value.
- `SRATE` The sample rate, in Hz, of the gravitational-wave signal.
- `NUM_TEMPLATES` is the maximum number of templates that can be used in filtering. The templates are defined by pairs of masses which are read from a file defined by the environment variable `GRASP_TEMPLATE`.
- `STORE_TEMPLATES` should be set to 1 to instruct the slave processes to save their templates; it should be set to 0 to instruct them to re-compute the templates each time they are used. The first option saves time but uses memory and can cause significant swapping. The second option increases the CPU usage by a factor of

$$1 + \frac{1}{\text{NPERSLAVE}} \quad (13.7.2)$$

but reduces the memory requirements enough to prevent swapping on any reasonable-sized machine. Note that if `NPERSLAVE = 1` then the factor above is two, so that storing templates halves the execution time.

- `PBINS` is the number of frequency bins used in the time/frequency χ^2 test defined in Section 6.24. The choice of single or two-phase test is made by setting `TWO_PHASE_R2` to 1 for a two-phase test, and to zero otherwise.
- One can instrument the MPI code with the Message Passing Extension (MPE) library by setting `ISMPE` to 1. The amount of detail in the log files is then controlled by the `SMALL_MPELOG` macro.
- `RENICE` If this is set to 1, the master and slave processes are “reniced” to run at low priority. This is useful if you are doing processing on a shared workstation network and don’t want to slow down the other users.
- `NPERSLAVE` The processing is done in parallel on the slaves: each slave acquires then analyzes `NPERSLAVE` segments of data, by computing a filter, FFTing it, then correlating all `NPERSLAVE` segments with it. This is beneficial if the filter FFT’s are not being stored on the slaves, since by making `NPERSLAVE` sufficiently large, the time spent computing and FFTing a template can be made vanishingly small. This may be seen from equation (13.7.2).
- `PART_TEMPLATE` can be used to analyse the correlated output of one template in detail. The rectified output of the template is recorded in a file called `template_(PART_TEMPLATE).(myid)` where (XX) denotes the numerical value of XX. The variable `myid` refers to the process id when the program is run in a multiprocessor environment. The file contains two columns namely the offset and the SNR at that offset. Moreover the `signal.*` files contain an entry only for the particular template `PART_TEMPLATE`. This macro is to be set to -1 if one wants the entire template bank to be used.
- `COMPARE` should be set to 1 to record the filter peak offsets and 0 to record the impulse offsets. Please refer to equation 13.7.1 for a definition of these quantities. Note: to recover the behavior of `binary_search` before this macro was added, set it to 0.
- `REMOVE_LINE_BINS` should be set to 1 to remove certain frequency bins that appear to be strongly contaminated with line-like resonances, else to 0.
- `INJECT_TIME_REVERSE` should be set to 1 to inject time-reversed chirps, 0 otherwise
- `DEBUG_COMM` should be set to 1 to print statements that are helpful in debugging MPI communication problems.
- `DBX` Should be set to 1 to start up debuggers on each different process, 0 otherwise.
- `REVERSE_FILTERS` Should be set to 1 to time-reverse the filter bank, 0 otherwise.

13.7.3 File: `binary_search.c` (MPI multiprocessor code)

This is the slightly-ugly production code used in the binary inspiral search on the November 1994 data.

```
/* GRASP: Copyright 1997,1998 Bruce Allen */
/* binary_search.c
 * (based on multifilter.c)
 * JC: 25 July 1997, PRB: 19 August 1997, JC: 27 October 1997, 29 November 1997
 * BA: 11-13 April 1998
 */

#include <stdlib.h>
#include "binary_params.h"
#include "binary_string.h"
#include "mpi.h"
#include "grasp.h"
#include "assert.h"
#include <unistd.h>
#if (ISMPE)
#include "mpe.h"
#endif
static char *rcsid="$Id: binary_search.c,v 1.23 1999/07/07 18:04:43 ballen Exp $\n$Name: RELEASE_1_9_0";

#ifndef M_LN2
#define M_LN2          0.69314718055994530942          /* log e2 */
#endif

/* global variables for passing data from get_calibrated_data() */
double datastart;
float *n_inv_noise,*htilde,*pow_renorm,srate=9868.4208984375;
int num_templates,npoint=NPOINT,new_lock,gauss_test,insert_chirp=INSERT_CHIRP;

/* global variables for passing between master and slave in non mpi mode */
float *template_list,*sig_buffer;

/* MPI global variables */
int numprocs,myid,namelen;
char processor_name[MPI_MAX_PROCESSOR_NAME],logfile_name[256],*cmd_name;
void export_environment(int,const char*);
pid_t process_id;
#if (RENICE)
char commandstring[256];
int nicevalue;
#endif

/* define prototypes */
void realft(float *,unsigned long ,int);
void corr_coef(float *, float *, float *, int, float *, float *, float *);
/* the following routine is the Numerical Recipes routine select().
   However its name has been changed to NRselect. See section 2.5.2 of the GRASP
   manual for details!
 */
float NRselect(unsigned long, unsigned long, float [ ]);
int get_calibrated_data();

/* time-reversed filters */
void make_retlifs(float m1, float m2, float *ch1, float *ch2, float fstart,
                 int n, float srate, int *filled, float *t_coal,
```

```
        int err_cd_sprs, int order);

/* information to save about segments of data */
struct Saved {
    double tstart;
    int gauss;
    int segmentno;
};

int main(int argc, char *argv[]) {
    /* prototypes */
    void master();
    void slave();

    /* Get the name of this program */
    cmd_name = argv[0];
    /* start MPI, find number of processes, find process number */
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    MPI_Get_processor_name(processor_name, &namelen);

    /* export any environment variables from master to slaves (repeat as many times as needed) */
    export_environment(myid, "GRASP_MFPATH");
    export_environment(myid, "GRASP_TEMPLATE");
    export_environment(myid, "GRASP_KILLSCRIPT");
    #if (DBX)
        export_environment(myid, "DISPLAY");
    #endif /* DBX */
    process_id=getpid();
    #if (RENICE)
        /* set priority to 19 for slaves, 17 for master */
        if (myid)
            nicevalue=19;
        else
            nicevalue=17;
    #if (!DBX)
        sprintf(commandstring, "renice %d -p %d", nicevalue, process_id);
        system(commandstring);
        printf("Processor %s (number %d) process number %d set NICE to %d\n",
            processor_name, myid, process_id, nicevalue);
        fflush(stdout);
    #endif /* !DBX */
    #endif /* (RENICE) */

    /* create a file containing a kill script for this job */
    {
        int i;
        FILE *fpkill;
        char filemode[2];
        filemode[1]='\0';
        for (i=0; i<numprocs; i++) {
            MPI_Barrier(MPI_COMM_WORLD);
            if (i==myid) {
                filemode[0]=(myid==0)?'w':'a';
                fpkill=grasp_open("GRASP_KILLSCRIPT", "killscript", filemode);
                if (myid==0) fprintf(fpkill, "#\n");
                fprintf(fpkill, "rsh %s kill -9 %d\n", processor_name, process_id);
                fflush(fpkill);
            }
        }
    }
}
```

```
        fclose(fpkill);
    }
}

/* if desired, pause to attach debugger */
/* if (myid==0) sleep(30); */

#if (DBX)
    sprintf(commandstring, "xterm -sb -e gdb %s %d &", cmd_name, process_id);
    if (myid==0) system(commandstring);
    sleep(15);
    MPI_Barrier(MPI_COMM_WORLD);
#endif

#if (ISMPE)
    MPE_Init_log();
#endif /* (ISMPE) */

/* Gravity wave signal (frequency domain),
   twice inverse noise power,
   and power renormalization factor */
{
    int factor=1;
    factor=(myid==0)?1:NPERSLAVE;
    htilde = (float *)malloc(factor*(sizeof(float)*npoint
        +sizeof(float)*(npoint/2+1)+sizeof(float)));
}
n_inv_noise = htilde + npoint;
pow_renorm = n_inv_noise + npoint/2 + 1;

/* In the MPI version of the code, call the master or slave */
if (myid==0)
    master();
else
    slave();

/* shutdown process */
printf("%s preparing to shut down (process %d)\n", processor_name, myid);
fflush(stdout);
#if (ISMPE)
    sprintf(logfile_name, "%s.%d.%d.log", cmd_name, numprocs, DATA_SEGMENTS);
    MPE_Finish_log(logfile_name);
#endif
printf("%s waiting at MPI_Barrier... (process %d)\n", processor_name, myid);
MPI_Barrier(MPI_COMM_WORLD);
MPI_Finalize();
printf("%s shutting down (process %d)\n", processor_name, myid);
fflush(stdout);
return 0;
}

/* Function executed by the master node */
void master()
{
    MPI_Status status;
    struct Saved *saveme;
    int i, islave, num_sent=0, num_recv=0, *tot_per_slave, nperslave, *slave_shutdown;
    int startsegment=0;
```

```
char *startsegenv;

startsegenv=getenv("GRASP_STARTSEGMENT");
if (startsegenv==NULL)
    startsegment=0;
else {
    printf("Environment variable GRASP_STARTSEGMENT set to %s\n",startsegenv);
    startsegment=atoi(startsegenv);
    if (startsegment<0 || startsegment > 2048) startsegment=0;
    printf("Master starting with segment # %d\n",startsegment);
}

#if(ISMPE)
MPE_Describe_state(1,2,"Templates->Slaves","red:vlines3");
MPE_Describe_state(3,4,"Data->Slaves","blue:gray3");
MPE_Describe_state(5,6,"Master Receive","brown:light_gray");
MPE_Describe_state(7,8,"Data->Master","yellow:dark_gray");
MPE_Describe_state(9,10,"Slave Receive","orange:white");
MPE_Describe_state(13,14,"Slaves<-templates","gray:black");
MPE_Describe_state(15,16,"compute template","lavender:black");
MPE_Describe_state(17,18,"real fft","lawn green:black");
MPE_Describe_state(19,20,"correlate","purple:black");
MPE_Describe_state(21,22,"correlation coefficients","wheat:black");
MPE_Describe_state(23,24,"likelihood test","light sky blue:black");
#endif

{ /* read in template list */

    float m1,m2;
    FILE *fptemplates;

    num_templates = NUM_TEMPLATES;
    template_list = (float *)malloc(sizeof(float)*2*num_templates);
    fptemplates=grasp_open("GRASP_TEMPLATE","templates.ascii","r");

    for (i=0;i<num_templates;i++) {
        if (EOF==fscanf(fptemplates,"%f %f\n",&m1,&m2)) {
            fprintf(stderr,"Warning: template file ended at template number %d\n%s\n",i,rcsid);
            fflush(stderr);
            num_templates = i;
            break;
        }
        template_list[2*i] = m1;
        template_list[2*i+1] = m2;
    }
    fclose(fptemplates);
}

/* print out the header */
{
    time_t translate_time;
    float flo=FLO;
    int bytes=0,one=1;
    FILE *fpheader;

    fpheader = grasp_open("GRASP_MFPATH","signal.header","w");
    bytes += fprintf(fpheader,
                    "%d\n",HEADER_COMMENT_SIZE);
    bytes += fprintf(fpheader,
```

```
        "This is output from a first pass filtering of the Nov 1994 data set.\n");
time(&translate_time);
bytes += fprintf(fpheader, "%s\n", ctime(&translate_time));
bytes += fprintf(fpheader, "%s", comment);
bytes += fprintf(fpheader, "%s", description);
if (bytes > HEADER_COMMENT_SIZE) {
    fprintf(stderr, "HEADER_COMMENT_SIZE = %d must be increased to >= %d\n", HEADER_COMMENT_SIZE, bytes);
    fflush(stderr);
    MPI_Abort(MPI_COMM_WORLD, 1);
}
while (bytes < HEADER_COMMENT_SIZE) bytes += fprintf(fpheader, "\n");
fwrite(&one, 4, 1, fpheader);
fwrite(&num_templates, 4, 1, fpheader);
fwrite(&flo, 4, 1, fpheader);
fwrite(&srates, 4, 1, fpheader);
for (i=0; i < 2*num_templates; i++) fwrite(template_list+i, 4, 1, fpheader);
fclose(fpheader);
}

/* storage for returned signals (NSIGNALS per template) */
sig_buffer = (float *)malloc(sizeof(float)*num_templates*NSIGNALS*NPERSLAVE);

/* Structure for saving information about data sent to slaves */
saveme = (struct Saved *)malloc(sizeof(struct Saved)*numprocs*NPERSLAVE);
tot_per_slave = (int *)malloc(sizeof(int)*numprocs);
slave_shutdown = (int *)malloc(sizeof(int)*numprocs);
{
    int k;
    for (k=0; k < numprocs; k++) tot_per_slave[k] = slave_shutdown[k] = 0;
}

/* broadcast templates */
#if(ISMPE)
    MPE_Log_event(1, myid, "send");
#endif
    MPI_Bcast(&num_templates, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(template_list, 2*num_templates, MPI_FLOAT, 0, MPI_COMM_WORLD);
#if(ISMPE)
    MPE_Log_event(2, myid, "sent");
#endif

/* Skip segments 0 to startsegment-1 */
{
    int i;
    i = startsegment;
    while(--i >= 0) get_calibrated_data();
    num_sent = num_rcv = startsegment;
}

/* while not finished, loop over slaves */
for (islave=1; islave < numprocs; islave++)
    for (nper_slave=0; nper_slave < NPERSLAVE; nper_slave++)
        if (get_calibrated_data()) { /* if new data exists, then send it (nonblocking?) */
            num_sent++;
            printf("Master broadcasting data segment %d to slave %d\n", num_sent, islave);
            fflush(stdout);
        }
#if(ISMPE)
    MPE_Log_event(3, myid, "send");
#endif
#endif /* ISMPE */
```

```
MPI_Send(htilde,NPOINT+NPOINT/2+1+1,MPI_FLOAT,islave,num_sent,MPI_COMM_WORLD);
#if(ISMPE)
    MPE_Log_event(4,myid,"sent");
#endif /* ISMPE */
    saveme[NPERSLAVE*(islave-1)+nperslave].gauss = gauss_test;
    saveme[NPERSLAVE*(islave-1)+nperslave].tstart = datastart;
    saveme[NPERSLAVE*(islave-1)+nperslave].segmentno = num_sent;
    tot_per_slave[islave]++;
} else { /* tell remaining processes to exit */
    printf("Failed to get data segment %d\n",num_sent+1);
    fflush(stdout);
#if(ISMPE)
    MPE_Log_event(3,myid,"send");
#endif
    printf("Master - sent shutdown message to process %d\n",islave);
    fflush(stdout);
    MPI_Send(htilde,NPOINT+NPOINT/2+1+1,MPI_FLOAT,islave,0,MPI_COMM_WORLD);
    saveme[NPERSLAVE*(islave-1)+nperslave].segmentno=0;
    slave_shutdown[islave]=1;
#if(ISMPE)
    MPE_Log_event(4,myid,"sent");
#endif
}

/* now loop, gathering answers, sending out more data */
printf("Entering loop to gather answers\n");
fflush(stdout);
while (num_sent!=num_rcv) {

    FILE *fpout;
    char fname[256];
    int more_data;

    /* listen for answer */
#if(ISMPE)
    MPE_Log_event(5,myid,"receiving...");
#endif

    /* This next block retrieves signals from the slaves */
    {
        int flag=0;
        while (flag==0) {
            /* check to see if a signal is waiting for pickup */
            MPI_Iprobe(MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD,&flag,&status);
            if (flag) {
                /* if it is, then get it and exit */
                MPI_Recv(sig_buffer,NSIGNALS*num_templates*NPERSLAVE,MPI_FLOAT,
                    MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
            }
            else
                sleep(1);
        }
    }
#if (DEBUG_COMM)
    printf("Master just got a response from slave %d (should be %d segments)\n",
        status.MPI_SOURCE,tot_per_slave[status.MPI_SOURCE]);
    fflush(stdout);
#endif
    if (status.MPI_TAG != tot_per_slave[status.MPI_SOURCE]) {
        fprintf(stderr,"SERIOUS PROBLEM: slave node %d returned data for: %d data segments\n",
```



```
        MPE_Log_event(4,myid,"sent");
#endif /* ISMPE */
    }
}

} /* end of while (num_sent!=num_rcv) loop */

/* when all the answers are in, print results */
printf("This is the master - all answers are in!\n");
fflush(stdout);

/* shut down any remaining slaves */
for (islave=1;islave<numprocs;islave++)
    if (slave_shutdown[islave]==0) {
        MPI_Send(htilde,NPOINT+NPOINT/2+1+1,MPI_FLOAT,islave,0,MPI_COMM_WORLD);
        printf("Master - sent shutdown message to process %d (don't know why still running!)\n",
            islave);
        fflush(stdout);
    }
free(saveme);
return;
}

void slave()
{
    void realft(float *, unsigned long, int);
    static float *output0=NULL,*output90=NULL,*snrdata=NULL;
    static float *lhc=NULL,*lch0tilde=NULL,*lch90tilde=NULL;
    static int *lchirppoints,completed=0;
    int i,num_stored,temp_no,nskip=4,validdata=0,nperslave,location;
    float *htildel,*n_inv_noisel,*pow_renorml;
    MPI_Status status;
    int segno[NPERSLAVE];

#ifdef PART_TEMPLATE>-1
    char templatefile[256];
    FILE *fp_pt[NPERSLAVE];
#endif

    printf("Slave %d (%s) just got started...\n",myid,processor_name);
    fflush(stdout);

    /* allocate storage space */
    /* Ouput of matched filters for phase0 and phase pi/2, in time domain, and temp storage */
    snrdata = (float *)realloc(snrdata,sizeof(float)*npoint/nskip);
    output0 = (float *)realloc(output0,sizeof(float)*npoint);
    output90 = (float *)realloc(output90,sizeof(float)*npoint);

    /* get the list of templates to use */
#ifdef ISMPE
    MPE_Log_event(13,myid,"receiving...");
#endif
    MPI_Bcast(&num_templates,1,MPI_INT,0,MPI_COMM_WORLD);
    sig_buffer = (float *)malloc(sizeof(float)*num_templates*NSIGNALS*NPERSLAVE);
    template_list = (float *)malloc(sizeof(float)*2*num_templates);
    MPI_Bcast(template_list,2*num_templates,MPI_FLOAT,0,MPI_COMM_WORLD);
#ifdef ISMPE
    MPE_Log_event(14,myid,"received");
#endif
#endif
```

```
printf("Slave %d (%s) just got template list...\n",myid,processor_name);
fflush(stdout);

/* Phase 0 and phase pi/2 chirps, in frequency domain */
num_stored = STORE_TEMPLATES*(num_templates - 1) + 1;
lch0tilde = (float *)realloc(lch0tilde,sizeof(float)*npoint*num_stored);
lch90tilde = (float *)realloc(lch90tilde,sizeof(float)*npoint*num_stored);
lchirppoints = (int *)realloc(lchirppoints,sizeof(float)*num_stored);
ltc = (float *)realloc(ltc,sizeof(float)*num_stored);

if (lch0tilde==NULL || lch90tilde==NULL || lchirppoints==NULL || ltc==NULL) {
    fprintf(stderr,"Node %d on machine %s: could not malloc() memory!\n%s\n",
            myid,processor_name,rctsid);
    fflush(stderr);
    MPI_Abort(MPI_COMM_WORLD,5);
}

/* now enter an infinite loop, waiting for new inputs */
while (1) {
    /* listen for data, parameters from master */
#ifdef ISMPE
    MPE_Log_event(9,myid,"receiving...");
#endif
    validdata=0;
    for (nperslave=0;nperslave<NPERSLAVE;nperslave++) {
        MPI_Recv(htilde+nperslave*(NPOINT+NPOINT/2+1),NPOINT+NPOINT/2+1,MPI_FLOAT,0,MPI_ANY_TAG,
                MPI_COMM_WORLD,&status);
        segno[nperslave]=status.MPI_TAG;
#ifdef (DEBUG_COMM)
        printf("Slave %s node %d loop: message tag is %d\n",processor_name,myid,status.MPI_TAG);
        fflush(stdout);
#endif /* DEBUG COMM */
        if (segno[nperslave]!=0)
            validdata++;
        else
            break;
    }
#ifdef ISMPE
    MPE_Log_event(10,myid,"received");
#endif
    /* if this is a termination message, we are done! */
    if (validdata==0)
        break;

    printf("Slave %d (%s) got htilde (and noise spectrum) for %d segments: %d to %d\n",
           myid,processor_name,validdata,segno[0],segno[validdata-1]);
    fflush(stdout);

    /* compute signals */
    for (temp_no=0;temp_no<num_templates;temp_no++) {

        float *ch0tilde,*ch90tilde,*tc,r00,r11,r01,sigma2,distance;
        float snr_max,c0,c90,varsplit,stats[2*PBINS],med_renorm;
        int *chirppoints,num_crossing,maxi,impulseoff,indices[PBINS];

        /* To skip all templates except a particular one */
#ifdef (PART_TEMPLATE)>-1
        for (nperslave=0;nperslave<validdata;nperslave++) {
            snr_max=0.0;
```

```
varsplit=0.0;
num_crossing=0;

if((temp_no)!=PART_TEMPLATES) {
    sig_buffer[nperslave*num_templates*NSIGNALS+temp_no*NSIGNALS] = 0.0;
    sig_buffer[nperslave*num_templates*NSIGNALS+temp_no*NSIGNALS+1] = 0.0;
    sig_buffer[nperslave*num_templates*NSIGNALS+temp_no*NSIGNALS+2] = 0.0;
    sig_buffer[nperslave*num_templates*NSIGNALS+temp_no*NSIGNALS+3] = 0.0;
    ((int *)sig_buffer)[nperslave*num_templates*NSIGNALS+temp_no*NSIGNALS+4] = 0.0;
    sig_buffer[nperslave*num_templates*NSIGNALS+temp_no*NSIGNALS+5] = 0.0;
    continue; /* THIS IS NO LONGER RIGHT?? */
}
else{
    sprintf(templatefile,"template_%d.%d",PART_TEMPLATES,segno[nperslave]);
    fp_pt[nperslave] = grasp_open("GRASP_MFPATH",templatefile,"w");
}
}
#endif

ch0tilde = lch0tilde + npoint*temp_no*STORE_TEMPLATES;
ch90tilde = lch90tilde + npoint*temp_no*STORE_TEMPLATES;
chirppoints = lchirppoints + temp_no*STORE_TEMPLATES;
tc = ltc + temp_no*STORE_TEMPLATES;

/* Compute the template, and store it internally, if desired */
if (completed!=num_templates) {
    float m1,m2;
    int longest_template=0;

    /* manufacture two chirps (dimensionless strain at 1 Mpc distance) */
    m1 = template_list[2*temp_no];
    m2 = template_list[2*temp_no+1];

#ifdef ISMPE
    if (!SMALL_MPELOG) MPE_Log_event(15,myid,"computing");
#endif
#ifdef REVERSE_FILTERS
    make_retlifs(m1,m2,ch0tilde,ch90tilde,FLO,npoint,srate,chirppoints,tc,4000,4);
#else
    make_filters(m1,m2,ch0tilde,ch90tilde,FLO,npoint,srate,chirppoints,tc,4000,4);
#endif

#ifdef ISMPE
    if (!SMALL_MPELOG) MPE_Log_event(16,myid,"computed");
#endif

    if (*chirppoints>longest_template) longest_template = *chirppoints;

    if (*chirppoints>CHIRPLEN) {
        fprintf(stderr,"Chirp m1=%f m2=%f length %d too long!\n",m1,m2,
            *chirppoints);
        fprintf(stderr,"Maximum allowed length is %d\n",CHIRPLEN);
        fprintf(stderr,"Please recompile with larger CHIRPLEN value\n");
        fprintf(stderr,"%s\n",rcsid);
        fflush(stderr);
        MPI_Abort(MPI_COMM_WORLD,6);
    }

    /* normalize the chirp template */
    /* normalization of next line comes from GRASP (5.6.3) and (5.6.4) */
```

```
{
    float inverse_distance_scale=2.0*HSCALE*(TSOLAR*C_LIGHT/MPC);
    for (i=0;i<*chirppoints;i++){
        ch0tilde[i] *= inverse_distance_scale;
        ch90tilde[i] *= inverse_distance_scale;
    }
}

/* zero out the unused elements of the tilde arrays */
for (i=(*chirppoints);i<npoint;i++)
    ch0tilde[i]=ch90tilde[i]=0.0;

/* and FFT the chirps */
#if(ISMPE)
    if (!SMALL_MPELOG) MPE_Log_event(17,myid,"starting fft");
#endif
    realft(ch0tilde-1,npoint,1);
#if(ISMPE)
    if (!SMALL_MPELOG) MPE_Log_event(18,myid,"ending fft");
    if (!SMALL_MPELOG) MPE_Log_event(17,myid,"starting fft");
#endif
    realft(ch90tilde-1,npoint,1);
#if(ISMPE)
    if (!SMALL_MPELOG) MPE_Log_event(18,myid,"ending fft");
#endif
    if (STORE_TEMPLATES) completed++;

/* print out the length of the longest template */
if (completed==num_templates)
    fprintf(stderr,"Slave %d: templates completed. Longest is %d points\n",
        myid,longest_template);
fflush(stdout);
} /* done computing the template */

/* correlation coefficients */
for (nperslave=0;nperslave<validdata;nperslave++) {
    n_inv_noise=n_inv_noise+nperslave*(NPOINT+NPOINT/2+1+1);
    htilde=htilde+nperslave*(NPOINT+NPOINT/2+1+1);
    pow_renorml=pow_renorm+nperslave*(NPOINT+NPOINT/2+1+1);
#if(ISMPE)
    if (!SMALL_MPELOG) MPE_Log_event(21,myid,"starting");
#endif
    corr_coef(ch0tilde,ch90tilde,n_inv_noise,npoint,&r00,&r11,&r01);
#if(ISMPE)
    if (!SMALL_MPELOG) MPE_Log_event(22,myid,"done");
#endif
    /* sigma squared and optimal distance scale Mpc for SNR=1 */
    sigma2 = 0.5*(r00 + r11);
    distance = sqrt(sigma2);

/* find the moment at which SNR is a maximum */
#if(ISMPE)
    if (!SMALL_MPELOG) MPE_Log_event(19,myid,"searching");
#endif
    {
        int count=0;
        float x,y,amp,medsnr,expect=sqrt(M_LN2);
        float x2py2,x2py2max,x2py2_thresh;
```

```
correlate(output0,htilde1,ch0tilde,n_inv_noise1,npoint);
correlate(output90,htilde1,ch90tilde,n_inv_noise1,npoint);

x2py2max=0.0;
num_crossing=0;
x2py2_thresh=THRESHOLD*THRESHOLD*sigma2;
maxi=-1;
for (i=PRESAFETY;i<NPOINT-POSTSAFETY;i++) {
    x = output0[i];
    y = output90[i];
    x2py2 = x*x + y*y;
#if (PART_TEMPLATES>-1)
    fprintf(fp_pt[nperslave],"%d %f\n",i,sqrt(x2py2/sigma2));
#endif
    if (x2py2>x2py2max) {
        x2py2max = x2py2;
        maxi = i;
    }
    if (x2py2>x2py2_thresh) num_crossing++;
    if (!(i%nskip)) snrdata[count++] = x2py2;
}
snr_max=sqrt(x2py2max/sigma2);

#if (PART_TEMPLATES>-1)
    fclose(fp_pt[nperslave]);
#endif

/* check that we did correctly find a maximum offset */
assert(PRESAFETY <= maxi && maxi <= NPOINT-POSTSAFETY);

c0 = (r11*output0[maxi] - r01*output90[maxi]);
c90 = (r00*output90[maxi] - r01*output0[maxi]);
amp = sqrt(c0*c0 + c90*c90);

c0 /= amp;
c90 /= amp;
/* the following routine is the Numerical Recipes routine select().
   However its name has been changed to NRselect. See section 2.5.2 of the GRASP
   manual for details!
*/
medsnr = NRselect(count/2,count,snrdata-1);
med_renorm = expect/sqrt(medsnr/sigma2);
}

#if (ISMPE)
    if (!SMALL_MPELOG) MPE_Log_event(20,myid,"done");
#endif

/* identify when an impulse would have caused observed filter output */
impulseoff = (maxi + *chirppoints)%npoint;

/* collect interesting signals to return */
location=NSIGNALS*num_templates*nperslave+temp_no*NSIGNALS;
sig_buffer[location] = distance;
sig_buffer[location+1] = snr_max;
sig_buffer[location+2] = snr_max*(pow_renorm);
sig_buffer[location+3] = snr_max*med_renorm;
#if (COMPARE)
    ((int *)sig_buffer)[location+4] = -maxi;
#else
```

```
((int *)sig_buffer)[location+4] = impulseoff;
#endif
sig_buffer[location+5] = atan2(c90,c0);

varsplit=0.0;
if (snr_max>THRESHOLD) {
#if(ISMPE)
    if (!SMALL_MPELOG) MPE_Log_event(23,myid,"testing");
#endif
#if (TWO_PHASE_R2)
    /* two-phase r^2 test */
    varsplit = splitup_freq5(sqrt(0.5),sqrt(0.5),ch0tilde,ch90tilde,r00,
                            n_inv_noisel,npoint,
                            maxi,PBINS,indices,stats,output0,htildel);
#else
    /* Single phase r^2 test: */
    varsplit = splitup_freq2(c0,c90,ch0tilde,ch90tilde,r00,
                            n_inv_noisel,npoint,
                            maxi,PBINS,indices,stats,output0,htildel);
#endif
    varsplit /= sigma2;
#if(ISMPE)
    if (!SMALL_MPELOG) MPE_Log_event(24,myid,"done");
#endif
    }
    sig_buffer[location+6] = varsplit;
    ((int *)sig_buffer)[location+7] = num_crossing;
    } /* end of loop over the templates */
} /* end of loop over nperslave */

/* return signals to master */
#if(ISMPE)
    MPE_Log_event(7,myid,"send");
#endif
#if (DEBUG_COMM)
    printf("Slave %s node %d sending message to master: tag is %d\n",processor_name,myid,
          validdata);
    fflush(stdout);
#endif /* DEBUG_COMM */
    MPI_Send(sig_buffer,NSIGNALS*num_templates*NPERSLAVE,MPI_FLOAT,0,validdata,MPI_COMM_WORLD);
#if(ISMPE)
    MPE_Log_event(8,myid,"sent");
#endif
    if (validdata<NPERSLAVE) break;
} /* end of loop over the data */

printf("Node %d on machine %s: received a shutdown message\n",
       myid,processor_name);
fflush(stdout);
return;
}

/* This routine exports environment variables to the nodes */
void export_environment(int node_number,const char *name) {
    char *environment,*var=NULL;
```

```
int length;

/* if this is the master, get environment variable */
if (node_number==0) {
    if (name==NULL) {
        fprintf(stderr,"Argument to export_environment() can't be null!\n");
        fflush(stderr);
        MPI_Abort(MPI_COMM_WORLD,2);
    }
    var=getenv(name);
    if (var==NULL) {
        fprintf(stderr,"Environment variable %s MUST be set!\n",name);
        fflush(stderr);
        MPI_Abort(MPI_COMM_WORLD,3);
    }
    /* calculate the length of the string to pass */
    length=strlen(name)+strlen(var)+2;
}

/* broadcast the length of the string */
MPI_Bcast(&length,1,MPI_INT,0,MPI_COMM_WORLD);

/* allocate storage for the string */
if (NULL==(environment=(char *)malloc(length))) {
    fprintf(stderr,"Export_environment() node %d couldn't allocate memory!\n",
            node_number);
    fflush(stderr);
    MPI_Abort(MPI_COMM_WORLD,4);
}

/* print environment variable into string */
if (node_number==0) {
    sprintf(environment,"%s=%s",name,var);
    fprintf(stderr,"Exporting environment variable %s\n",environment);
    fflush(stderr);
}

/* then broadcast it to the other processes */
MPI_Bcast(environment,length,MPI_CHAR,0,MPI_COMM_WORLD);

/* and put it into the local environment */
if (node_number) {
    putenv(environment);
    fprintf(stderr,"Node %d receiving environment variable %s\n",
            node_number,environment);
    fflush(stderr);
}
return;
}
```

13.7.4 File: binary_get_data.c

```
#include <string.h>
#include "grasp.h"
#include "binary_params.h"

#define SIM_VARIANCE 16384.0
#define SIM_SITE 8

char *rcsid="$id";

static int count_chunks,count_locked,count_segments;
static double count_tinlock,lastlock,discarded,lastdiscard=0;
static FILE *fp_statistics;

struct Chunk {
    double time;
    short *data;
    float *spec;
    int cont;
    int is_gaussian;
    int counter;
    int used_in_spectrum;
};

void make_retlifs(float m1, float m2, float *ch1, float *ch2, float fstart,
                int n, float srate, int *filled, float *t_coal,
                int err_cd_sprs, int order);

int get_calibrated_data();
#if (RANDOMIZE)
float ran2(long *);
long randomize=-12345;
#endif

void nullout(float freqmin,float freqmax,int npoint,float srate,float* n_inv_noise) {
    int imin,imax,i;

    imin=freqmin*npoint/srate;
    imax=freqmax*npoint/srate;
    /* set to zero in those bins */
    if (imin<npoint/2)
        for (i=imin;((i<imax) && (i<npoint/2));i++)
            n_inv_noise[i]=0.0;
    return;
}

/* Routine to compute window function for some specified window type.
 *
 * Input is the window type:
 * type = 0: rectangular (no) window,
 * type = 1: Hann window,
 * type = 2: Welch window,
 * type = 3: Bartlett window.
 *
 * Output are *wss (sum of window function values squared) and the window
 * function values window[0..n-1]. */
void compute_window(float *wss, float window[], int n, int type)
```

```
{
  int i;
  float fac=2.0/n;

  *wss = 0;
  for (i=0;i<n;i++) {
    float win;
    switch (type) {
      case 0: /* rectangular (no) window */
        win = 1;
        break;
      case 1: /* Hann window */
        win = 0.5*(1 - cos(i*fac*M_PI));
        break;
      case 2: /* Welch window */
        win = i*fac - 1;
        win = 1 - win*win;
        break;
      case 3: /* Bartlett window */
        win= 1 - fabs(i*fac - 1);
        break;
      default: /* unrecognized window type */
        GR_start_error("avg_spec()",rcsid, __FILE__, __LINE__);
        GR_report_error("don't recognize windowtype=%d\n", type);
        GR_end_error();
        abort();
        break;
    }
    *wss += win*win;
    window[i] = win;
  }
  return;
}

/* Routine to calculate the response function using the old-style
 * data acquisition routines */
void recalibrate_old(float *response, int npoint)
{
  int i;
  FILE *fpss;
  fpss = grasp_open("GRASP_DATAPATH", "swept-sine.ascii", "r");
  normalize_gw(fpss, npoint, SRATE, response);
  fclose(fpss);
  for (i=0;i<npoint;i++) response[i] *= HSCALE/ARMLENGTH_1994;
  return;
}

/* Are we building a frame-compatible version? */
#if (DATA_FORMAT == 1)

/* Global frame variables */
struct fgetoutput fgetoutput={0};
struct fgetinput fgetinput={0};
int frame_init=0;
int time_last_calibrated=-1;

/* Routine to initialize global frame variables and to allocate memory */
void initialize_frame()
```

```
{
    frame_init = 1;
    fgetinput.nchan = 1;
    fgetinput.files = framefiles;
    fgetinput.calibrate = 1;
    fgetinput.chnames = (char **)malloc(fgetinput.nchan*sizeof(char *));
    fgetinput.locations = (short **)malloc(fgetinput.nchan*sizeof(short *));
    fgetinput.chnames[0] = "IFO_DMRO";
    fgetinput.inlock = 1;
    fgetoutput.npoint = (int *)malloc(fgetinput.nchan*sizeof(int));
    fgetoutput.ratios = (int *)malloc(fgetinput.nchan*sizeof(int));
    return;
}

/* Routine to calculate the response function using the frame
 * data acquisition routines. This routine should only be called
 * after some data has been read, so that the frame output variables
 * have been set. */
void recalibrate_frame(float *response, int npoint)
{
    float srate=SRATE;
    int i;

    if (!frame_init) {
        GR_start_error("recalibrate_frame()", rcsid, __FILE__, __LINE__);
        GR_report_error("must get some data before calling this routine!\n");
        GR_end_error();
        abort();
    }

    /* print message */
    GR_start_error("recalibrate_frame()", rcsid, __FILE__, __LINE__);
    GR_report_error("Note: data calibration carried out at time %d.\n",
                    fgetoutput.tcalibrate);
    GR_report_error(" previous frame calibration was from time %d.\n",
                    time_last_calibrated);
    GR_end_error();

    /* do the calibration */
    GRnormalize(fgetoutput.fri, fgetoutput.frinum, npoint, srate, response);
    for (i=0; i<npoint; i++) response[i] *= HSCALE/ARMLENGTH_1994;

    /* record time that we did calibration */
    time_last_calibrated=fgetoutput.tcalibrate;

    return;
}
#endif /* end of frame-compatible version */

/* Routine to calculate the response function for simulated data */
void recalibrate_sim(float *response, int npoint)
{
    double *power, *tmp;
    float parameters[9], srate=SRATE;
    float fac=HSCALE*sqrt(0.5*srate/SIM_VARIANCE);
    char site_name[256], noise_file[256], whiten_file[256];
    int m=npoint/2;
    tmp = power = (double *)malloc(m*sizeof(double));
```

```
if (power == NULL) {
    GR_start_error("recalibrate_sim()", rcsid, __FILE__, __LINE__);
    GR_report_error("failed to allocate %d doubles\n", m);
    GR_end_error();
    abort();
}
detector_site("detectors.dat", SIM_SITE, parameters,
             site_name, noise_file, whiten_file);
noise_power(noise_file, m, srate/npoint, power);
while (m-- > 0) {
    *response++ = fac*sqrt(*tmp++);
    *response++ = 0;
}
free(power);
}

/* Routine to calculate the response function. The response function
 * is only updated if needed — otherwise, it is unchanged.
 *
 * Input is the number of points in the response function, npoint.
 *
 * Output is the response function array response[0..npoint-1]. */
void recalibrate(float *response, int npoint)
{
# if (DATA_FORMAT == 0)
    static int first = 1;
    /* old format data */
    if (first) {
        first = 0;
        recalibrate_old(response, npoint);
    }
# elif (DATA_FORMAT == 1)
    /* frame format data */
    if (time_last_calibrated != fgetoutput.tcalibrate)
        recalibrate_frame(response, npoint);
# elif (DATA_FORMAT == 2)
    static int first = 1;
    /* simulated data */
    if (first) {
        first = 0;
        recalibrate_sim(response, npoint);
    }
# else
    /* not a recognized format! */
    GR_start_error("recalibrate()", rcsid, __FILE__, __LINE__);
    GR_report_error("unrecognized DATA_FORMAT %d\n", DATA_FORMAT);
    GR_end_error();
    abort();
# endif
    return;
}

/* open the IFO output file and read the mainheader to get runstart time */
double get_runstart_old()
{
    struct ld_mainheader mh;
    struct ld_binheader bh;
    float tstart, srate;

```

```
int zero=0,npt=NPOINT;
FILE *fpifo;
short *here;
fpifo = grasp_open("GRASP_DATAPATH","channel.0","r");
read_block(fpifo,&here,&zero,&tstart,&srates,0,&npt,1,&bh,&mh);
fclose(fpifo);
return (double)mh.epoch_time_sec + (double)mh.epoch_time_msec*0.001;
}

/* Routine to get a chunk of data using the old data acquisition routines.
 *
 * Input is the number of points in the chunk, npoint.
 *
 * Output is the data in the array here[0..npoint-1]
 * and the time, *time, of the start of the data.
 *
 * Returned is the code:
 * code = 0: no more data;
 * code = 1: beginning of a locked section
 * (no data acquired, but we have skipped MIN_INT0_LOCK
 * minutes into the locked segment);
 * code = 2: continuing a locked section (data acquired). */
int get_chunk_data_old(short *here, int npoint, double *time)
{
    static FILE *fpifo,*fplock;
    static int remain=0,first=1;
    static float srates=SRATE;
    static double runstart;
    float tstart;
    int code;

    if (first) { /* initialization */
        first = 0;
        runstart = get_runstart_old();
        /* open the IFO output file, lock file, and swept-sine file */
        fpifo = grasp_open("GRASP_DATAPATH","channel.0","r");
        fplock = grasp_open("GRASP_DATAPATH","channel.10","r");
    }

    if (remain < npoint) { /* if new locked section */
        int nskip=(int)(60*MIN_INT0_LOCK*SRATE);
        /* skip forward MIN_INT0_LOCK minutes */
        code = get_data(fpifo,fplock,&tstart,nskip,here,&remain,&srates,1);
        if (code == 0) return 0;
        else return 1;
    }

    /* get the needed data */
    code = get_data(fpifo,fplock,&tstart,npoint,here,&remain,&srates,0);

    /* compute the start time */
    *time = runstart + tstart;

    return code;
}

/* are we building a frame compatible version? */
#if (DATA_FORMAT == 1)
```

```
/* Routine to get a chunk of data using the frame data acquisition routines.
 *
 * Input is the number of points in the chunk.
 *
 * Output is the data in the array here[0..npoint-1]
 * and the time, *time, of the start of the data.
 *
 * Returned is the code:
 * code = 0: no more data;
 * code = 1: beginning of a locked section
 * (no data acquired, but we have skipped MIN_INT0_LOCK
 * minutes into the locked segment—WARNING... the data
 * array here will be filled with junk, which should be
 * ignored!)
 * code = 2: continuing a locked section (data acquired). */
int get_chunk_data_frame(short *here, int npoint, double *time, int *counter)
{
    static float srate=SRATE;
    int code, retcode;
    static int private_count=0;

    private_count++;
    fprintf(fp_statistics, "get_chunk_data_frame: getting chunk %d\n", private_count);

    if (!frame_init) initialize_frame();

                                /* we want... */
    fgetinput.npoint = npoint;    /* ...npoints of data... */
    fgetinput.locations[0] = here; /* ...in array here[ ]... */
    fgetinput.seek = 0;          /* ...no seek... */
    fgetinput.inlock = 1;        /* ...only when in lock */

    /* get npoints of data */
    retcode = code = fget_ch(&fgetoutput, &fgetinput);
    fprintf(fp_statistics, "Current frame file: %s\n", fgetoutput.filename);

    while (code == 1) {
        /* seek to a total of MIN_INT0_LOCK minutes into locked section ... */
        fgetinput.seek = 1;
        /* ... we have already gone npoint, so we need to subtract this off ... */
        fgetinput.npoint = MIN_INT0_LOCK*60*srate - npoint;
        /* ... subtracting npoint is an inconvenience! */
        if (fgetinput.npoint < 1) {
            GR_start_error("get_chunk_data_frame()", rcsid, __FILE__, __LINE__);
            GR_report_error("set MIN_INT0_LOCK to a larger value\n");
            GR_end_error();
            abort();
        }
        code = fget_ch(&fgetoutput, &fgetinput);
        fprintf(fp_statistics, "Current frame file: %s\n", fgetoutput.filename);

        if (code == 0) return 0; /* no more data */
        if (code == 1) { /* oops... skiped into yet another locked section ... */
            /* ... so we need to skip forward the npoint offset we subtracted above */
            fgetinput.npoint = npoint;
            code = fget_ch(&fgetoutput, &fgetinput);
            fprintf(fp_statistics, "Current frame file: %s\n", fgetoutput.filename);
        }
    }
}
```

```
        if (code == 0) return 0;
    }
}

lastlock=fgetoutput.lastlock;
discarded=(double)fgetoutput.discarded/srate;
*time = fgetoutput.tstart;
*counter=private_count;
return retcode;
}
#endif

/* Routine to generate a chunk of simulated white noise.
 *
 * Input is the number of points in the chunk, npoint.
 *
 * Output is the data in the array here[0..npoint-1]
 * and the time, *time, of the start of the data.
 *
 * Returned is the code:
 * code = 0: no more data;
 * code = 1: beginning of a locked section
 * (no data acquired, but we have skipped MIN_INTO_LOCK
 * minutes into the locked segment);
 * code = 2: continuing a locked section (data acquired). */
int get_chunk_data_sim(short *here, int npoint, double *time)
{
    float ran2(long *);
    static double local_time=0,srate=SRATE;
    static long seed=-100;
    int m=npoint/2;

    if (seed < 0) {
        ran2(&seed);
        return 1;
    }
    local_time += (double)npoint/srate;
    *time = local_time;
    while (m-- > 0) {
        float x,y,rr,fac;
        do {
            x = 2*ran2(&seed) - 1;
            y = 2*ran2(&seed) - 1;
            rr = x*x + y*y;
        } while (rr > 1 || rr == 0);
        fac = sqrt(-2*SIM_VARIANCE*log(rr)/rr);
        *here++ = floor(fac*x);
        *here++ = floor(fac*y);
    }
    return 2;
}

/* Routine to compute the spectrum of a chunk of data.
 *
 * Input is the number of points of data, npoint,
 * and the array of data, data[0..npoint-1].
 *
 * Output is spectrum, spec[0..npoint/2]. */
```

```
void get_spectrum(short *data, float *spec, int npoint)
{
    void realft(float [], unsigned long, int);
    static float *work=NULL,*win=NULL,fac;
    static int nkeep=-1;
    int i,n=npoint,m=npoint/2;

    if (npoint != nkeep) {
        float wss;
        nkeep = npoint;
        work = realloc(work,npoint*sizeof(float));
        if (work == NULL) {
            GR_start_error("get_spectrum()",rcsid,__FILE__,__LINE__);
            GR_report_error("failed to allocate %d floats\n",npoint);
            GR_end_error();
            abort();
        }
        win = realloc(win,npoint*sizeof(float));
        if (win == NULL) {
            GR_start_error("get_spectrum()",rcsid,__FILE__,__LINE__);
            GR_report_error("failed to allocate %d floats\n",npoint);
            GR_end_error();
            abort();
        }
        compute_window(&wss,win,npoint,WINDOW);
        fac = 2/(wss*SRATE);
    }
    /* copy data to workspace and apply window */
    for (i=0;i<n;i++) work[i] = win[i]*data[i];

    /* FFT workspace */
    realft(work-1,npoint,1);

    /* DC and Nyquist terms */
    spec[0] = fac*work[0];
    spec[m] = fac*work[1];

    /* other terms */
    for (i=1;i<m;i++) {
        int ir=i+i,ii=ir+1;
        float re=work[ir],im=work[ii];
        spec[i] = fac*(re*re + im*im);
    }
    return;
}

/* === Patrick Brady === Modified: 4 December 1998 ===
*
* Routine to fill a chunk with data.
*
* Input is the number of points, npoint, in the chunk of data.
*
* Output is the struct chunk with
* chunk.data[0..npoint-1]: the data
* chunk.spec[0..npoint/2]: the power spectrum of the data
* chunk.time: the time of the start of the data in the chunk
* chunk.cont: a flag indicating if the data is continuous from the
* last chunk filled (1) or not (0).
* chunk.is_gaussian: a flag to indicate if the data has outliers (0) or not (1)
```

```
*
* Returned is the code:
* code = 0: no more data;
* code = 1: beginning of a locked section
* (no data acquired, but we have skipped MIN_INT0_LOCK
* minutes into the locked segment);
* code = 2: continuing a locked section (data acquired). */
int fill_chunk(struct Chunk *chunk, int npoint)
{
    int code;

#ifdef DATA_FORMAT == 0
    /* old format data */
    code = get_chunk_data_old(chunk->data, npoint, &chunk->time);
#elif DATA_FORMAT == 1
    /* frame format data */
    code = get_chunk_data_frame(chunk->data, npoint, &chunk->time, &chunk->counter);
#elif DATA_FORMAT == 2
    /* simulated data */
    code = get_chunk_data_sim(chunk->data, npoint, &chunk->time);
#else
    /* not a recognized request! */
    GR_start_error("fill_chunk()", rcsid, __FILE__, __LINE__);
    GR_report_error("unrecognized DATA_FORMAT %d\n", DATA_FORMAT);
    GR_end_error();
    abort();
#endif

    switch (code) {
        case 0: /* no more data */
            return code;
        case 1: /* starting a new locked section */
            chunk->cont = 0;
            break;
        case 2: /* continuing a locked section */
            chunk->cont = 1;
            break;
        default:
            GR_start_error("fill_chunk()", rcsid, __FILE__, __LINE__);
            GR_report_error("unrecognized code %d\n", code);
            GR_end_error();
            abort();
    }

    chunk->is_gaussian = is_gaussian(chunk->data, npoint, -2048, 2047, 0);
    chunk->used_in_spectrum = 0;
    get_spectrum(chunk->data, chunk->spec, npoint);

    return code;
}

/* Bruce Allen, Modified Jan 18, 1998
   The new algorithm works as follows:
   (1) go through all the kk current chunks
   (2) Count the number which are outlier free
   (3) replace the oldest of the kk saved spectra with these ones
   (4) compute the average spectrum
   (5) return
```

Comment: the only place this routine might "misbehave" is at the start of a newly-calibrated section of data (if the calibration curve has significantly changed), or if the first `kk` segments of data all have outliers.

```
*/
void average_spectrum(struct Chunk *chunk, int kk, int npoint, float *spec, int new_lock)
{
    static int num_stored=0,oldest=0;
    static float *stored_spectras=NULL;
    static int *stored_chunks=NULL;

    float factor;
    int i,j,m=npoint/2;

    /* allocate memory if first time */
    if (!stored_spectras)
    {
        stored_spectras=(float *)malloc(kk*m*sizeof(float));
        stored_chunks=(int *)malloc(kk*sizeof(int));
    }

    /* loop over outlier-free chunks not already used in spectrum */
    for (i=0;i<kk;i++)
        if (chunk[i].is_gaussian && !chunk[i].used_in_spectrum)
        {
            float *oldest_spec;
            /* mark this spectrum as used */
            chunk[i].used_in_spectrum=1;
            /* keep track of chunk being used to produce spectrum */
            stored_chunks[oldest]=chunk[i].counter;
            /* copy outlier-free spectra to oldest saved spectra */
            oldest_spec=stored_spectras+oldest*m;
            memcpy((void *) (oldest_spec), (const void *) (chunk[i].spec), (size_t)m*sizeof(float));
            /* make circular buffer point back if needed */
            oldest=(oldest+1)%kk;
            /* increment number of stored spectra */
            if (num_stored<kk) num_stored++;
            /* print some useful diagnostic information */
            fprintf(fp_statistics,"Incorporating chunk %d into power spectra",chunk[i].counter);
            for (j=0;j<num_stored-1;j++)
                fprintf(fp_statistics," %d",stored_chunks[j]);
            fprintf(fp_statistics," %d\n",stored_chunks[num_stored-1]);
        }

    /* check that we have SOMETHING stored! */
    if (!num_stored)
    {
        fprintf(stderr,"average_spectrum: no outlier free data to compute spectrum with!\n");
        fflush(stderr);
        abort();
    }

    /* Now compute the sum of the stored spectra */
    clear(spec,m,1);
    for (j=0;j<num_stored;j++)
    {
        float *current_spec;
        current_spec=stored_spectras+j*m;
        for (i=0;i<m;i++)
```

```
        spec[i]+=current_spec[i];
    }

    /* Normalize to get average spectrum */
    factor=1.0/num_stored;
    for (i=0;i<m;i++)
        spec[i]*=factor;

    /* print out some information about which chunks are used */

    return;
}

/* === Patrick Brady === Modified: 4 December 1998 ===

Routine to average the spectra of several chunks of data.

chunk: Input. The array of chunks

kk: Input. The number of chunks

npoint: Input. The number of points of data in each chunk

spec: Ouput. Pointer to the array of (npoint/2 +1) floats

new_lock: Input. Flag to indicate if all the chunks are continuous (new_lock=0)
           or if we have new locked data in the buffer (new_lock=1).

Authors: Jolien Creighton

*/
void average_spectrum_brady(struct Chunk *chunk, int kk, int npoint, float *spec, int new_lock)
{
    static float *stored_spec;
    static int first=1;
    float fac=1/(float)kk,norm=0;
    int i,j,m=npoint/2;

    if (first){ /* Allocate memory the first time in */
        stored_spec=(float *)malloc((npoint/2 + 1)*sizeof(float));
        clear(stored_spec,m,1);
        first=0;
    }

    /* Note: ignore the Nyquist component for some reason... */
    /* If the data is continuous ... */
    if (new_lock==0){
        clear(spec,m,1);
        for (j=0;j<kk;j++){
            /* If data is outlier free ... */
            if (chunk[j].is_gaussian){
                /* ..... add in spectrum from this chunk */
                norm+=1;
                for (i=0;i<m;i++)
                    spec[i] += chunk[j].spec[i];
            }
        }
        /* If all spectra have outliers ... */
        if (!norm){
            GR_start_error("average_spectrum()", rcsid, __FILE__, __LINE__);
        }
    }
}
```

```

        GR_report_error("%i contiguous chunks with outliers\n",kk);
        GR_end_error();
        /* return the spectrum computed previously */
        for (i=0;i<m;i++) spec[i] = stored_spec[i];
    }
    /* otherwise ... */
    else{
        fac= (1/norm);
        for (i=0;i<m;i++){
            spec[i] *= fac;          /* normalize the spectrum */
            stored_spec[i]= spec[i]; /* keep this spectrum for when we encounter new locked data */
        }
    }
}
/* otherwise the data is not continuous */
else
    for (i=0;i<m;i++)
        spec[i] = stored_spec[i]; /* return the spectrum computed previously */
return;
}
*/ === Patrick Brady === 4 December 1998 ===

```

This function initialises the array of chunks, and returns an integer code:

code = 0: no more data;
code = 2: continuing a locked section.

If it returns code = 1 something is wrong.

The arguments are:

chunk: Output. The array of chunks.

npoint: Input. The number of points in chunk.data[0..npoint-1]

chunks_filled: Input. Number of chunks that are already filled.

start_filled: Input. First chunk with good data

kk: Input: total number of chunks

*/

```

int initialise_chunks(struct Chunk *chunk, int npoint, int chunks_filled, int start_filled, int kk){
    int i=0,code=2;
    float srate=SRATE;

    while (i < chunks_filled){
        /* copy the data */
        memcpy((void *)chunk[i].data,(void *)chunk[start_filled+i].data,(size_t)npoint*sizeof(short))
        /* copy the spectrum */
        memcpy((void *)chunk[i].spec,(void *)chunk[start_filled+i].spec,(size_t)(npoint/2+1)*sizeof(float))
        /* and the time, and continuity flag */
        chunk[i].time=chunk[start_filled+i].time;
        chunk[i].cont=chunk[start_filled+i].cont;
        chunk[i].counter=chunk[start_filled+i].counter;
        chunk[i].is_gaussian=chunk[start_filled+i].is_gaussian;
        chunk[i].used_in_spectrum=chunk[start_filled+i].used_in_spectrum;
    }
}

```

```
        /* increment the counter */
        i++;
    }

    while (i < kk){
        code = fill_chunk(chunk + i,npoint);
        switch (code) {
            case 0: /* no more data */
                return 0;
            case 1: /* entering new locked set */
                fprintf(fp_statistics,"Message from initialise_chunks():\n");
                fprintf(fp_statistics," Locked section was too short to get %i chunks\n",kk);
                fflush(fp_statistics);
                i= -1;
                break;
            case 2: /* continuing a locked set */
                break;
            default: /* unrecognized code */
                GR_start_error("initialise_chunks()",rcsid,__FILE__,__LINE__);
                GR_report_error("unrecognized code %d\n",code);
                GR_end_error();
                abort();
        }
        /* increment the counter */
        i++;
    }

    /* If we have a locked section > MIN_INT0_LOCK + kk*npoint/srate */
    if(code==2){
        fprintf(fp_statistics,"Message from initialise_chunks():\n");
        fprintf(fp_statistics," Last locked section had %f secs of data, ",MIN_INT0_LOCK*60.0 +
            (float)(count_chunks)*npoint/srate);
        fprintf(fp_statistics," and %f secs discarded\n",discarded-lastdiscard);
        lastdiscard=discarded;
        count_locked++;
        count_chunks=kk;
        count_tinlock+=(MIN_INT0_LOCK*60.0 + (double)(kk*npoint)/srate);
        fprintf(fp_statistics," Starting locked section %i at time %f with segment %d\n",count_locked,
            time(NULL),segment);
        fflush(fp_statistics);
    }

    return code;
}

/* === Patrick Brady === Modified: 4 December 1998 ===
```

This function gets more data for the `binary_search` code. It returns an integer code:

- code = 0: no more data;
- code = 1: beginning of (well... `MIN_INT0_LOCK` minutes into) a locked section of data
- code = 2: continuing a locked section.

The arguments are:

need: Input. The number of points to skip ahead between calls.

data: Output. The `IFO_DMRO` is returned in this array

spec: Output. The average spectrum is returned in `spec[0..npoint/2]`

*time: Output. Time at the start of the data

Authors: Jolien Creighton.

```
*/
int get_more_data(int need, short data[], float spec[], float response[],
                 double *time)
{
    static short *buffer;           /* pointer to the beginning of the buffered data */
    static short *end_buffer;       /* pointer to the end of the buffered data */
    static short *here;            /* pointer to the position of the data to return next */
    static struct Chunk *chunk;     /* buffered data also stored in chunks */
    static int lastfill_chunkNumber=-1; /* last chunk that was filled with data */
    static int first=1, npoint=NPOINT, chunk_needed=0, here_position=0, new_lock=0;
    const int kk=8, k=4;           /* kk is the (even) number of chunks; k is half of kk */
    int here_chunkNumber=0, out_code=2; /* chunk that here points to, and the return code */

    if (first) { /* on first call */
        int i;
        first = 0;
        /* Open the file to record the statistics of the run */
        fp_statistics = grasp_open("GRASP_MFPATH", "run.stats", "w");
        count_chunks=count_locked=count_segments=0;
        count_tinlock=0.0;
        /* allocate memory to the buffer: extra npoints is to insure continuity in the data */
        here = buffer = (short *)malloc((kk + 1)*npoint*sizeof(short));
        /* end_buffer points to the beginning of the duplicated data at the end of the buffer */
        end_buffer = buffer + kk*npoint;
        /* allocate memory for the chunks of data */
        chunk = (struct Chunk *)malloc(kk*sizeof(struct Chunk));
        for (i=0; i<kk; i++) { /* partition the buffered data into chunks */
            chunk[i].data = buffer + i*npoint;
            chunk[i].spec = (float *)malloc((npoint/2 + 1)*sizeof(float));
        }
        /* fill the chunks with data */
        if(!(initialise_chunks(chunk, npoint, 0, 0, kk))) return 0;
    }
    else { /* on other calls */
        here += need; /* advance here by the needed amount of data */
        chunk_needed += need; /* increment flag for getting another chunk */
        here_position += need; /* increment counter to position of here in buffer */
        here_chunkNumber = (here_position/npoint)%kk; /* chunk that here points to */
    }

    /* If here is in duplicated data, move to equivalent place at start of buffer */
    if ((here_position/npoint) == kk) {
        here_position = here_position%npoint;
        here = buffer + here_position;
    }

    /* Does next chunk contain continuous data? */
    if(here_position%npoint > 0){
        int next_chunk=(here_chunkNumber + 1)%kk;
        /* If not ..... */
        if (chunk[next_chunk].cont == 0 && new_lock == 1){
            int code;

```

```
code = initialise_chunks(chunk, npoint, 0, 0, kk);
switch (code) {
case 0: /* no more data */
return 0;
case 2: /* continuing a locked set */
break;
default: /* unrecognized code */
GR_start_error("get_more_data()", rcsid, __FILE__, __LINE__);
GR_report_error("Should never get code %d here\n", code);
GR_end_error();
abort();
}
/* PATRICK: check this with Jolien and Bruce. Does the calling routine need
to know about this? Or should it be told code=2?
*/
out_code=1; /* let calling routine know this is new locked data */
here=buffer; /* point here to start of buffer */
chunk_needed=here_position=here_chunkNumber=new_lock=0; /* Reset all counters */
lastfill_chunkNumber=-1; /* fill buffer from the beginning */
}
}

/* If we need more data, and the last call did not return a new locked chunk */
if( chunk_needed > k*npoint && new_lock == 0){
int code;
chunk_needed -= npoint;
/* advance lastfill_chunkNumber (modulo kk)... */
lastfill_chunkNumber = (lastfill_chunkNumber + 1) % kk;
/* ...and fill the data in the chunk */
code = fill_chunk(chunk + lastfill_chunkNumber, npoint);
switch (code) {
case 0: /* no more data */
return 0;
case 1: /* entering new locked set */
new_lock = 1;
break;
case 2: /* continuing a locked set */
count_chunks++;
count_tinlock+=((double)npoint/SRATE);
break;
default: /* unrecognized code */
GR_start_error("get_more_data()", rcsid, __FILE__, __LINE__);
GR_report_error("unrecognized code %d\n", code);
GR_end_error();
abort();
}
/* If data was put at beginning of buffer */
if (lastfill_chunkNumber == 0)
/* place a copy of it at the end to maintain continuity */
memcpy((void *)end_buffer, (void *)buffer, (size_t)npoint*sizeof(short));
}

/* compute the average spectrum for the kk chunks of buffered data */
average_spectrum(chunk, kk, npoint, spec, new_lock);

/* recalibrate (response only modified if needed) */
recalibrate(response, npoint);

/* the time of the returned data */
```

```
*time = chunk[here_chunkNumber].time + (double)(here_position%npoint)/SRATE;

/* copy the data to the returned array. */
memcpy((void *)data, (void *)here, (size_t)npoint*sizeof(short));

count_segments++;
fprintf(fp_statistics, "Returning segment %d, based on chunks", count_segments);
{ int i;
for (i=0; i<kk-1; i++)
    fprintf(fp_statistics, " %d", chunk[i].counter);
}
fprintf(fp_statistics, " %d\n", chunk[kk-1].counter);

return out_code;
}

/* global variables for passing data from get_calibrated_data() */
extern double datastart;
extern float *n_inv_noise, *htilde, *pow_renorm, srate;
extern int npoint, new_lock, gauss_test;

int get_calibrated_data()
{
    void reallft(float [], unsigned long, int);
#ifdef INSERT_CHIRP
    void ins_chirp(int);
#endif

    static float *work, *spec, *response, *ave_spec;
    static short *data;
    static int first=1, npoint=NPOINT, num_sent=0;
    double fac=(double)npoint*SRATE;
    int i, cut, code, need=npoint-(POSTSAFETY+PRESAFETY);
    static int nomoredata=0;

    if (first) {
        first = 0;
        data = (short *)malloc(npoint*sizeof(short));
        work = (float *)malloc(npoint*sizeof(float));
        spec = (float *)malloc((npoint/2 + 1)*sizeof(float));
        ave_spec = (float *)malloc((npoint/2 + 1)*sizeof(float));
        response = (float *)malloc((npoint+2)*sizeof(float));
        if (data == NULL || work == NULL || spec == NULL
            || ave_spec == NULL || response == NULL) {
            GR_start_error("get_calibrated_data()", rcsid, __FILE__, __LINE__);
            GR_report_error("could not allocate memory\n");
            GR_end_error();
            abort();
        }
    }

    /* return 0 if the required number of segments (or more!) have been sent */
    /* fprintf(stderr, "Just sent segment number %d\n", num_sent); */
    if ((num_sent++) >= DATA_SEGMENTS) return 0;
    if (nomoredata) return 0;

    code = get_more_data(need, data, ave_spec, response, &datastart);
```

```
switch (code) {
case 0:
    fprintf(fp_statistics,"Message from get_calibrated_data():\n");
    fprintf(fp_statistics,"  Last locked section had %f secs of data, ",MIN_INT0_LOCK*60.0 +
            (float)(count_chunks)*npoint/srate);
    fprintf(fp_statistics," and %f secs discarded\n",discarded-lastdiscard);
    fprintf(fp_statistics,"No more data\n");
    fprintf(fp_statistics,"  Total locked data = %f secs, acquired lock %i times, ",
            count_tinlock,count_locked);
    fprintf(fp_statistics," and analyzed %i segments\n",count_segments);
    fflush(fp_statistics);
    nomoredata=1;
    return 0;
case 1:
    new_lock = 1;
    break;
case 2:
    new_lock = 0;
    break;
default:
    GR_start_error("get_calibrated_data()",rcsid,__FILE__,__LINE__);
    GR_report_error("unrecognized code %d\n",code);
    GR_end_error();
    abort();
}

for (i=0;i<npoint;i++) work[i] = data[i];
realft(work-1,npoint,1);
#if (RANDOMIZE)
/* do everything BUT DC and Nyquist */
for (i=1;i<npoint/2;i++) {
    int ir=i+i,ii=ir+1;
    double mag=sqrt(work[ir]*work[ir]+work[ii]*work[ii]);
    float phase=2.0*M_PI*ran2(&randomize);
    work[ir]=mag*cos(phase);
    work[ii]=mag*sin(phase);
}
#endif
product(htilde,work,response,npoint/2);
#if (INSERT_CHIRP)
/* insert a chirp if desired */
ins_chirp(num_sent);
#endif

/* lower cutoff frequency */
cut = npoint*FLO/SRATE;
if (cut<1) cut = 1;

/* set n_inv_noise to zero at low frequencies and Nyquist */
n_inv_noise[0] = 0;
n_inv_noise[npoint/2] = 0;
for (i=1;i<cut;i++) n_inv_noise[i] = 0;

/* compute remaining n_inv_noise elements */
for (i=cut;i<npoint/2;i++) {
    int ir=i+i,ii=ir+1;
    double re=response[ir],im=response[ii],tmp=ave_spec[i];
#if (NORM_CF)
    n_inv_noise[i] = 4.0/(fac*tmp*(re*re + im*im));
#endif
}
```

```
#else
    n_inv_noise[i] = 2.0/(fac*tmp*(re*re + im*im));
#endif
}

# if (SPEC_TRUNC > 0) /* truncate time-domain version of n_inv_noise[] */
{
    float out0,norm = 2/(float)npoint; /* normalization factor for iFFT */
    int spec_zero=(SPEC_TRUNC/2); /* where to start zeroing */
    for(i=0;i<npoint;i++) work[i]=0; /* clear out work array */
    for(i=0;i<npoint/2;i++) work[i+i]=sqrt(n_inv_noise[i]); /* fill the array */
    realft(work-1,npoint,-1); /* iFFT it */
    for(i=spec_zero;i<npoint-spec_zero;i++) work[i]=0; /* truncate work[] */
    realft(work-1,npoint,1); /* FFT to freq domain */
    for(i=0;i<npoint/2;i++){
        out0=norm*work[i+i]; /* reconstruct n_inv_noise[] */
        n_inv_noise[i]=out0*out0;
    }
    if ((cut = npoint*FLO/SRATE) < 1) cut = 1; /* low-frequency cutoff */
    for (i=0;i<cut;i++) n_inv_noise[i]=0.0; /* clear low-frequency components */
    n_inv_noise[npoint/2] = 0; /* make absolutely sure Nyquist is zero */
}
# endif

#if (REMOVE_LINE_BINS)
{
    int harmonic;
    float freq;
    /* step through all frequency harmonics of 60 Hz. Note: freq=(srate*i)/npoint */
    for (harmonic=1;harmonic<85;harmonic++) {
        float freqmin,freqmax;
        /* find the freq bins +- 0.5 N Hz above/below center */
        freqmin=harmonic*59.5;
        freqmax=harmonic*60.5;
        nullout(freqmin,freqmax,npoint,srate,n_inv_noise);
    }
    /* Now do the same for the violin resonances & other lines: */
    /* Bruce's list */
    nullout( 79.0, 81.0,npoint,srate,n_inv_noise);
    nullout( 109.0, 110.0,npoint,srate,n_inv_noise);
    nullout( 139.0, 140.0,npoint,srate,n_inv_noise);
    nullout( 245.0, 246.0,npoint,srate,n_inv_noise);
    nullout( 487.0, 490.0,npoint,srate,n_inv_noise);
    nullout( 499.0, 501.0,npoint,srate,n_inv_noise);
    nullout( 571.0, 572.0,npoint,srate,n_inv_noise);
    nullout( 576.0, 585.0,npoint,srate,n_inv_noise);
    nullout( 592.0, 602.0,npoint,srate,n_inv_noise);
    nullout( 603.0, 607.0,npoint,srate,n_inv_noise);
    nullout( 998.0, 1001.0,npoint,srate,n_inv_noise);
    nullout(1155.0, 1159.0,npoint,srate,n_inv_noise);
    nullout(1208.0, 1214.0,npoint,srate,n_inv_noise);
    nullout(1740.0, 1750.0,npoint,srate,n_inv_noise);
    nullout(3500.0, 3520.0,npoint,srate,n_inv_noise);

    /* Stan's list */
    nullout( 77.0, 83.0,npoint,srate,n_inv_noise);
    nullout( 105.0, 115.0,npoint,srate,n_inv_noise);
    nullout( 569.0, 607.0,npoint,srate,n_inv_noise);
    nullout(1138.0, 1214.0,npoint,srate,n_inv_noise);
}
```

```
    nullout(1707.0, 1821.0,npoint,srate,n_inv_noise);
    nullout(3500.0, 3520.0,npoint,srate,n_inv_noise);
}
#endif /* (REMOVE_LINE_BINS) */

/* compute outlier statistic */
gauss_test = is_gaussian(data,npoint,-2048,2047,0);

/* compute power statistic—don't include DC term */
get_spectrum(data,spec,npoint);
*pow_renorm = 0;
for (i=1;i<npoint/2;i++) *pow_renorm += spec[i]/ave_spec[i];
*pow_renorm *= 2.0/npoint;
return 1;
}

#if (INSERT_CHIRP)
/* routine to determine if a chirp is to be inserted, and to insert it */
void ins_chirp(int segment)
{
    void realft(float *, unsigned long, int);
    static FILE *fpinsert,*fpinslog;
    static double instime=0;
    static float *chirp0,*chirp1,m1,m2,invMpc,c0,c1,phase;
    static int first=1,end=0,npoint=NPOINT;
    int offset;

    /* no more chirps to insert */
    if (end) return;
    if (first) {
        first = 0;

        /* open the insert.ascii file for input */
        fpinsert = grasp_open("GRASP_INSERT","insert.ascii","r");

        /* open the insert.log file for output */
        fpinslog = grasp_open("GRASP_INSERT","insert.log","w");

        /* allocate memory to chirp arrays */
        chirp0 = (float *)malloc(npoint*sizeof(float));
        chirp1 = (float *)malloc(npoint*sizeof(float));
    }

    /* scan through the file until the next chirp is found */
    /* note: assume that only one chirp will be present in any data segment! */
    while (instime<datastart) {

        float tc,scale=2*HSCALE*M_SOLAR/MPC;
        int i,code,chpts;

        /* read the next injected chirp time */
        code = fscanf(fpinsert,"%lf %f %f %f %f\n",
                    &instime,&m1,&m2,&invMpc,&phase);
        /* if we have reached the end of the file */
        if (code==EOF) {
            end = 1;
            fclose(fpinsert);
            fclose(fpinslog);
        }
    }
}
#endif
```

```
    free(chirp0);
    free(chirp1);
    return;
}

/* If injection is well before the segment start, try again */
if (instime < (datastart - 2 * npoint / srates)) continue;

/* coefficients of the injected chirp */
c0 = cos(phase);
c1 = sin(phase);

/* construct the chirp to be injected */
#if (INJECT_TIME_REVERSE)
    make_retlifs(m1, m2, chirp0, chirp1, FLO, npoint, srates, &chpts, &tc, 4000, 4);
#else
    make_filters(m1, m2, chirp0, chirp1, FLO, npoint, srates, &chpts, &tc, 4000, 4);
#endif
for (i = 0; i < chpts; i++) {
    chirp0[i] *= scale;
    chirp1[i] *= scale;
}
for (i = chpts; i < npoint; i++) chirp0[i] = chirp1[i] = 0;
realft(chirp0 - 1, npoint, 1);
realft(chirp1 - 1, npoint, 1);
}
/* compute offset; return if after the end of the data segment */
if (instime > (datastart + npoint / srates)) return;
offset = srates * (instime - datastart);

/* inject the chirp */
freq_inject_chirp(c0, c1, offset, invMpc, chirp0, chirp1, htilde, npoint);

/* write an entry into the log file */
fprintf(fpinslog, "%d %d %f %f %f %f %f\n",
        segment, offset, instime, m1, m2, invMpc, phase);
fflush(fpinslog);
return;
}
#endif
```

13.8 Scripts for running `binary_search`

In order to analyze the November 1994 data, in Frame format, we used a pair of scripts. The script `main-script.noinject` analyzes the data, and `mainscript.inject` analyzes the data with simulated signals added in. We normally run the scripts like this:

```
beowulf> mainscript.noinject >&! mainscript.out &
```

to redirect the output to a file and run in the background. The number of processors used is set in the scripts.

The `mainscript.noinject` script:

```
#!/bin/tcsh

unsetenv GRASP_DATAPATH
setenv GRASP_FRAMEPATH /data/frames
set rundir='pwd'/frameoutput
if ( ! -d $rundir ) then
    mkdir $rundir
endif

setenv GRASP_MFPATH $rundir
touch $GRASP_MFPATH/run_output
setenv GRASP_INSERT $rundir
setenv GRASP_STARTSEGMENT 0

setenv GRASP_TEMPLATE `pwd`
setenv GRASP_KILLSCRIPT $rundir
touch $rundir/environment_values
setenv | grep GRASP >! $rundir/environment_values
echo Starting Frame Run at time `date` | mail ballen@dirac.phys.uwm.edu
mpirun -np 48 -machinefile ./machines ./examples_binary-search/binary_search >&! $rundir/run_output
```

The `mainscript.inject` script:

```
#!/bin/tcsh -v

unsetenv GRASP_DATAPATH
setenv GRASP_FRAMEPATH /data/frames
set rundir='pwd'/frameoutput-inject
if ( ! -d $rundir ) then
    mkdir $rundir
endif

setenv GRASP_MFPATH $rundir
touch $GRASP_MFPATH/run_output
setenv GRASP_INSERT $rundir
setenv GRASP_STARTSEGMENT 6000
if ( ! -f $GRASP_INSERT/insert.ascii ) then
    cp -f /home/ballen/insert.ascii-midmass.2 $GRASP_INSERT/insert.ascii
endif

setenv GRASP_TEMPLATE `pwd`
setenv GRASP_KILLSCRIPT $rundir
touch $rundir/environment_values
setenv | grep GRASP >! $rundir/environment_values
echo Starting Frame Run at time `date` | mail ballen@dirac.phys.uwm.edu
mpirun -np 48 -machinefile ./machines ./examples_binary-search/binary_search >&! $rundir/run_output
```

13.9 Example: `binary_reader` program

This function reads the output of the `binary_search` program as implemented to analyse the 40m data taken in November 1994. The data was broken into segments of approximately 20 seconds in length (plus approximately 6 seconds to avoid wrap-around problems in the discrete correlations). This program, with no options implemented, prints out the maximum signal to noise achieved over the bank of filters for each segment. The function requires the environment variable `GRASP_MFPATH` to point to the directory containing the output files written by `binary_reader`. You can set this with a command such as

```
setenv GRASP_MFPATH /usr/local/GRASP/data/mfout/
```

The `binary_reader` program does not assume that the data files which it is reading were written on a machine with the same byte-order as the machine that it is running one. This is helpful if one does production runs on a little-endian machine (say a DEC alpha machine) but subsequently analyzes the data on a big-endian machine (for example a SUN sparcstation). The `binary_reader` program does any byte-swapping that is needed.

The program has several optional flags, some of which require a numerical value to be supplied. The flags are:

- h : prints a summary of the command options
- m : prints the list of templates to standard out. The output format is *template m₁ m₂*
- f *fmt* : specifies that the format *fmt* should be used in output (see below)
- l : specifies that field labels should be included in output
- o : only print segments without outliers
- r *rmax* : reject filters with r^2 less than *rmax* in maximization over template bank
- t *threshold* : only print segments with maximum SNR greater than *threshold*
- L *lower* : only maximize over templates numbers greater than *lower*
- T *tempno* : consider only template number *tempno*—do not maximize over template bank
- M : maximize over the template bank using the median-renormalized signal-to-noise ratio.
- c : Print the filter peak offsets. These are defined in equation 13.7.1. The peak offsets are only stored in the data files created by `binary_search` and used by `binary_reader` if the macro `COMPARE = 1` has been set in the file `binary_params.h`. If peak offsets are requested when the impulse offsets have been recorded, an error message is printed and vice versa.
- s *segno* : print the results for all the templates for the segment *segno* to the file *segment.segno*.

The format specifier *fmt* is a string of upto 16 characters which are replaced by the corresponding output field:

t : segment start time

s : segment number

- O** : segment outlier test result (1: outliers / 0: no outliers)
- T** : number of maximum template
- M** : masses of maximum template
- D** : distance of SNR=1 for maximum template
- S** : max SNR of maximum template
- X** : max SNR of maximum template with power renormalization
- Y** : max SNR of maximum template with median renormalization
- O** : offset of maximum template
- P** : phase of maximum template
- R** : r^2 test statistic value of maximum template
- N** : number of threshold crossings of maximum template

Examples

1. To print the SNR and segment number of each segment, simply type

```
binary_reader
```

2. To print the segment start time and offset for every data segment that passes an outlier test and has a maximum SNR greater than 10 where the maximization is over all templates that have r^2 less than 2.5:

```
binary_reader -o -t 10.0 -r 2.5 -f tO
```

3. To produce a sorted list of signal-to-noise ratio—maximized over templates that have r^2 less than 2.5—and rank for data segments that pass an outlier test:

```
binary_reader -o -r 2.5 -f S | sort -n | awk ' { print $1"\t"NR } '
```

4. To print the information about all the templates for a particular segment (for example segment number 10) to a file segment.10:

```
binary_reader -s 10
```

5. To interpret the stored offsets as filter peak offsets and not as impulse offsets:

```
binary_reader -c -f O
```

Authors: Patrick Brady (patrick@tapir.caltech.edu), Jolien Creighton (jolien@tapir.caltech.edu), R. Balasubramanian (bala@chandra.phys.uwm.edu) and Bruce Allen (ballen@dirac.phys.uwm.edu).

Comments: The routine could be modified to keep the SNR in an internal array, and then to provide both the SNR and the fractional number of events exceeding that SNR in two columns. This is then directly comparable to the false alarm probability.

13.10 Identification of Spurious Events

The non-stationary and non-Gaussian noise in the detector leads to a large number of spurious detections, where by a detection we mean that the Signal-to-Noise Ratio (SNR) maximized over the template bank has crossed a certain preset threshold. In order to differentiate between a false alarm and a ‘correct’ detection, we must make use of several independent discrimination techniques. The r^2 test is one such technique and is described in the GRASP manual in section 6.24.

It is also useful to investigate whether the outputs of the templates in the template bank can be used collectively as a discriminator between a false alarm and a correct detection. This would be different from a discriminator constructed using only the template which maximises the correlation between the detector output and the template or in other words the SNR. Such a technique might be a robust indicator of the presence of a chirp signal in the detector output.

One possibility is to measure the times-of-arrival (t_a) at each template. The time-of-arrival at each template is the time when the correlated output reaches a maximum in that template. This maximization is carried out after the phase parameter is maximised over. The basic expectation here is that the t_a measured at templates in the neighbourhood of the template which maximises the SNR will follow a certain pattern if the signal is actually present in the detector output. If the distribution of the t_a (as a function of the template number) for a spurious detection can be shown to be very different from the case when a true signal is actually present, then this can be quantified and used as a discriminator.

In order to test this idea, experiments were carried out. Chirp signals were artificially injected into the 40-m data and filtered through the bank of templates. High signal to noise ratio events were identified. Some of these detections correctly corresponded with the injected signal. In particular, for these events the estimated values of the masses and the time of arrival tallied well with those of the actual signal injected.

In Figure 82 we plot the measured value of the times-of-arrival at each template, contrasting the case where a ‘real’ chirp signal is present (black points) and where no signal is injected but there is a large SNR observed (red points). The black points are obtained for segment number 2 (in the 14nov94.1 data) where by convention segment number 0 is the first segment of data. (Note: The length of the data segments were taken to be 262144 and the parameters PRESAFETY and POSTSAFETY were set to be 16384 and 49152 respectively. Three minutes of the data was skipped at the beginning of each locked stretch of the data.) The maximum signal to noise ratio is obtained for template number 383 with a SNR of 13.62. Here again we use the index 0 to denote the first template. The masses corresponding to the template number 383 tally well with the those of the injected signal (1.4,1.4 solar mass binary). We observe a well-defined pattern for the t_a across the template bank. The templates are roughly arranged in the order of decreasing masses and consequently of larger chirp times. The red points are obtained for segment 33. The maximum SNR obtained in this case is 8.48. In this segment no chirp signal had been injected. Again we plot the t_a obtained at various templates. The behaviour is remarkably similar to the case where there is actually a signal present in the detector output. The arrows point to the template which maximises the correlation.

A possible explanation for this phenomenon is as follows. Consider first the case where a signal has been injected into the data (the black points on the graph). In this case the template and signal achieve a good correlation if the large amplitudes parts of their waveforms are well aligned. In other words the template matches well with the injected signal if the time of coalescence is the same for both the template and the injected signal. The sum of the chirp times $t_{chirp} = \tau_0 + \tau_1 + \tau_{1.5} + \tau_2$ is almost equal to the length of the chirp waveform and the time of coalescence is the sum, $t_C = t_a + t_{chirp}$. The templates are evenly placed in the τ_0, τ_1 parameter space and consequently the total chirp time increases almost linearly with the template number. Since t_C is being held constant, t_a must decrease linearly with the template number.

Consider now the case when there is no injected signal. Nevertheless, a large SNR (8.48) has been obtained. We assume that the high SNR is caused by a short burst of noise in the data. The correlation between any template and a noise burst will be maximised for a time-of-arrival for which the last few cycles

of the template coincide with the noise burst. Now since the the chirp times increase roughly linearly as the template number increases we again have the time of arrival decreasing linearly. Thus, we conclude that it is difficult to distinguish between a correct detection and a false alarm using this test.

Another possible discriminator could be the variation of the maximum correlation (maximized over the phase and the arrival times) as a function of template number. In Figure 83 we plot the maximum SNR obtained for a template against the template number. The black curve represents segment number 2 corresponding to a true detection and the red curve represents segment number 33 corresponding to a spurious detection. Again the variation of the SNR with template number in the two cases does not help in distinguishing a true detection from a false alarm.

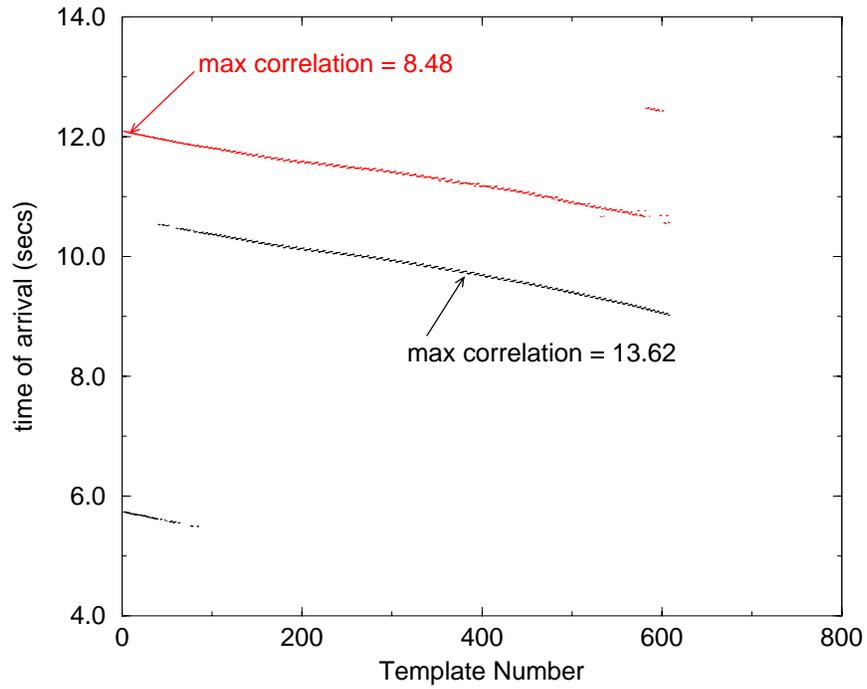


Figure 82: Variation of time of arrival across template bank.

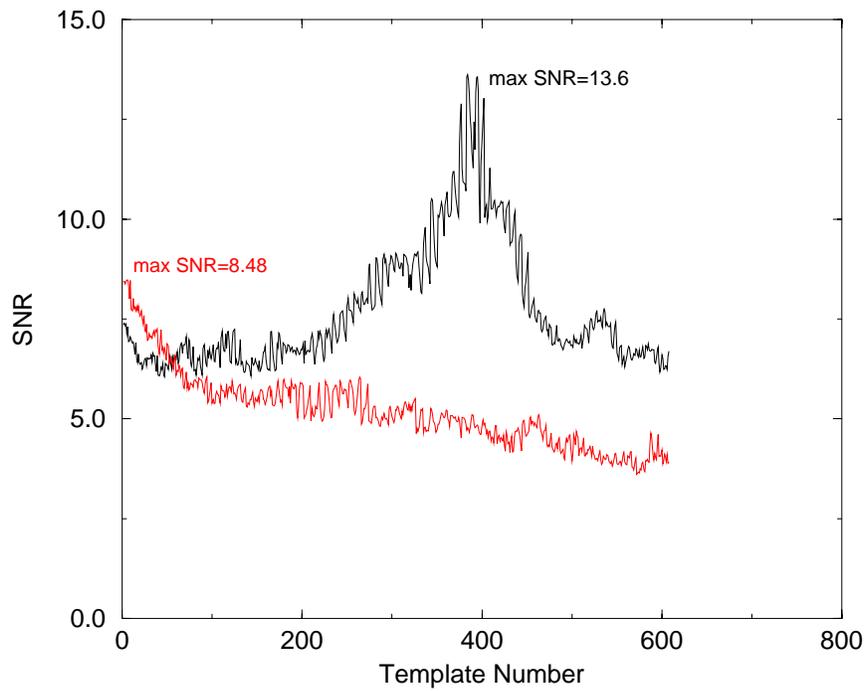


Figure 83: Variation of SNR across the template bank

14 GRASP Routines: Supernovae and other transient sources

In any classification scheme for natural phenomena, the boundaries between classifications run the risk of becoming broad and blurry. While previous sections have avoided this by considering very narrowly defined classes of gravitational wave sources, this section considers a much broader class of sources, and as such is a kind of “catch all” for a variety of sources. These sources, however, all share in common two properties which unite them in terms of practical detection strategies:

1. They produce gravitational radiation in the Ligo frequency and sensitivity band for a short time. The definition of short time is, of course, relative. However, if the time scale over which a source is visible to LIGO is of the order of days, as opposed to seconds, then it will likely be more suited to the analysis tools for quasi-periodic and periodic sources in section 15.
2. Astrophysical models of these sources are somewhat crude. Those short lived sources from which the gravitational radiation can be modelled accurately enough to permit matched filtering, may be dealt with using the tools of earlier sections (cf. section 6).

At the time of this writing, the sources which have these properties are thought to be primarily related to supernova events.

14.1 Centrifugal Hang-up of Core Collapse Supernovae

Massive stars which have burnt their nuclear fuel will collapse under the pull of their gravitational self attraction, sometimes triggering a supernova event. This collapse can lead to situations in which significant gravitational radiation is produced. In particular, there are various scenarios in which the collapsing core is thought to “hang-up” in a non-axisymmetric configuration and radiate this assymetry away through gravitational waves.

A promising scenario of this type was explored by Lai and Shapiro [42] and proceeds as follows: a stellar core with some initial angular momentum collapses. As the collapse proceeds, the ratio $\beta = T/W$ of the rotational energy (T) to the gravitational potential energy (W) will vary inversely with the core radius. If β becomes large enough, the evolution of non-axisymmetric modes of the core become unstable. Such instabilities are well known, being first described by Chandrasekhar [43]. There are two possibilities. If β lies in a critical band of values between ~ 0.17 and ~ 0.27 , the non-axisymmetric bar mode ($\ell = m = 2$) of the core will be *secularly* unstable and grow due to either radiation reaction or viscosity. This long lasting instability is expected to produce significant gravitational radiation. If β exceeds the upper limit of this band ($\beta > 0.27$) then non-axisymmetric modes become *dynamically* unstable. The transition through dynamical instability is rapid and results in a nearly axisymmetric configuration with $\beta < 0.27$ but still much larger than 0.17. The core therefore again enters the regime of secular instability. In either scenario the onset of the secular instability occurs at approximately neutron star densities. If $\beta < 0.17$ for the entire collapse, then no significant gravitational radiation is expected from this mechanism.

Lai and Shapiro have calculated the waveform of the gravitational radiation emitted by a secularly unstable core based on crude Newtonian fluid ellipsoids models without viscosity. They find [44]:

$$\left. \begin{aligned} h_+(t) &= A(f_{max}, r, M, d) u(t)^{2.1} \sqrt{1-u(t)} (1 + \cos^2 i) \cos \Phi(t), \\ h_\times(t) &= 2 A(f_{max}, r, M, d) u(t)^{2.1} \sqrt{1-u(t)} \cos i \sin \Phi(t). \end{aligned} \right\} \quad (14.1.1)$$

where:

f_{max} is the initial frequency of the gravitational wave (typically $\lesssim 800$ Hz).

r is the radius of the core (typically ~ 10 km).

M is the mass of the core (typically $\sim 1.4 M_\odot$).

d is the distance from the core (source).

i inclination angle (see section 6).

$A(f_{max}, r, M, d)$ is the amplitude function

$$A = \frac{1}{2} \frac{M^2}{rd} \left\{ \begin{aligned} &\left(\frac{\overline{f_{max}}}{1756} \right)^{2.7}, & \overline{f_{max}} \leq 415 \text{Hz}, \\ &\left(\frac{\overline{f_{max}}}{1525} \right)^{3.0}, & \overline{f_{max}} > 415 \text{Hz}. \end{aligned} \right. \quad (14.1.2)$$

$\overline{f_{max}} = f_{max} \sqrt{r_{10}^3 / M_{1.4}}$ where $r_{10} = r/10\text{km}$ and $M_{1.4} = M/(1.4M_\odot)$.

$u(t) = f(t)/f_{max}$ is given by the frequency evolution equation

$$\frac{du}{dt} = \left(\frac{M}{r} \right)^{5/2} \left(\frac{A}{0.16} \right)^2 \sqrt{\frac{M_{1.4}}{r_{10}}} u^{5.2} (u - 1). \quad (14.1.3)$$

$\Phi(t)$ is the phase, given implicitly by

$$f(t) = \frac{1}{2\pi} \frac{d\Phi}{dt}(t). \quad (14.1.4)$$

This model is implemented below.

14.2 Structure: LS_physical_constants

The physical parameters describing the hung-up core are passed in a structure `struct LS_physical_constants`. The fields are:

```
struct LS_physical_constants {
```

`float mass`: The mass of the stellar core in solar masses (typically $\sim 1.4M_{\odot}$).

`float radius`: The radius of the stellar core in km (typically ~ 10 km).

`float distance`: The distance from the stellar core to the detector in Megaparsecs.

`float fmax`: The maximum frequency of gravitational wave emitted by the stellar core (typically $\lesssim 800$ Hz).

`inclination angle`: angle between spin axis of stellar core and line of sight to detector in radians.

`Phi_0`: the initial phase of the gravitational wave.

```
}
```

14.3 Function: LS_freq_deriv()

void LS_freq_deriv(float t, float u[], float dudt[]) This function gives the derivative of frequency (actually, u which is the frequency divided by the maximum frequency) as a function of time, (see (14.1.3)). The amplitude $\left(\frac{M}{r}\right)^{5/2} \left(\frac{A}{0.16}\right)^2 \sqrt{\frac{M_{1.4}}{r_{10}}}$ must be declared and assigned a value as an external variable called A.

The arguments are:

t: Input. The time at which the derivative is to be taken.

u[]: Input. An array of initial values (in this case, containing exactly one element) of the dependent variables (in this case, only u).

dudt[]: Output. An array of derivatives (in this case, containing exactly one element) of derivatives of the dependent variables (in this case, $\partial u / \partial t$).

Authors: Warren G. Anderson, warren@ricci.phys.uwm.edu

Comments: This function is required by numerical recipes odeint. See the numerical recipes manual for more information.

14.4 Function: LS_phas_and_freq()

```
void LS_phas_and_freq(double Phi[], float u[], float A, float fmax, float dt, int n_samples)
```

This function integrates Lai and Shapiro's frequency and phase evolution equations, (14.1.3) and (14.1.4), for a hung-up collapsed core. We use numerical recipes odeint, an adaptive step size 4th order Runge-Kutta integrator for the frequency integration and a simple trapezoidal integration for the phase.

The arguments are:

`Phi[]`: Output. An array which holds the phase of the gravitational wave in radians at equally spaced time intervals. `Phi[]` must be allocated sufficient memory before being passed to `LS_phas_and_freq()`.

`u[]`: Output. An array which holds the reduced frequency (frequency divided by f_{max}) of the gravitational wave at equally spaced time intervals. `u[]` must be allocated sufficient memory before being passed to `LS_phas_and_freq()`.

`A`: Input. The Amplitude A as calculated in (14.1.2)

`fmax`: Input. The maximum frequency, f_{max} . Usually taken from the appropriate field of a `LS_physical_constants` structure.

`dt`: Input. The time interval (in seconds) at which the phase and frequency values should be output.

`n_samples`: Input. The number of phase and frequency values to be output (i.e. the number of elements in the arrays `Phi[]` and `u[]`).

Authors: Warren G. Anderson, warren@ricci.phys.uwm.edu and Patrick Brady, patrick@tapir.caltech.edu

14.5 Function: LS_waveform()

```
void LS_waveform(float h[], struct LS_physical_constants phys_const, float sky_theta, float sky_phi, float polarization, float dt, int n_samples)
```

This function calculates the stress as measured by a LIGO like detector due to a hung-up collapsing core. It uses the routines LS_phas_and_freq to obtain the wave phase and reduced frequency at regular time intervals, calculates h_+ and h_\times at each interval according to (14.1.1), and then converts these into a single detector stress at each interval using beam pattern factors calculated with beam_pattern. Note that the external variable A is assigned a value here for use in LS_freq_deriv (see 14.3).

The arguments are:

`h[]`: Output. An array which holds the detector stress due the gravitational wave of the hung-up core at equally spaced time intervals. `h[]` must be allocated sufficient memory before being passed to LS_waveform().

`phys_const`: Input. A structure of type LS_physical_constants (see subsection 14.2) which contains the physical parameters of the hung-up core.

`sky_theta`: Input. The polar angle from zenith in radians.

`sky_phi`: Input. The azimuthal angle (measured counter clockwise from first arm) in radians.

`polarization`: Input. The polarization angle in radians.

`dt`: Input. The time interval (in seconds) at which the detector stress values should be output.

`n_samples`: Input. The number of detector stress values to be output (i.e. the number of elements in the array `h[]`).

Authors: Warren G. Anderson, warren@ricci.phys.uwm.edu and Patrick Brady, patrick@tapir.caltech.edu

Comments: This function currently calculates the stress at a user defined distance from the hung-up core. For realistic distances, this leads to small values (of the order of $10e-22$ at 10 Mpc). If one wishes to have numbers of the order of unity, simply set the distance in the structure `phys_const` to be $10e-22$.

15 GRASP Routines: Periodic and quasi-periodic sources

16 GRASP Routines: General purpose utilities

This section includes general purpose utility functions for a variety of purposes. For example, these include functions for error handling, to calculate time-averaged power spectra, and functions to graph data, listen to data, etc.

16.1 GRASP Error Handling

GR_error is the GRASP error reporting module. It has two abstract interfaces which insulate the GRASP library and the programs which use it from the details of how GRASP will report errors and how the program will handle them. The internal interface provides GRASP itself with a standard method to report errors. The external interface allows programs which use GRASP to specify exactly how error reports are to be handled. In addition, a default set of handling routines is provided for printing error messages in a standard format to stderr or a log file.

If you are writing a package for GRASP, here is an example of how to call the error handler. The following sections contain more detailed information, but this example is a useful summary of “how to use it”. Please note that you *must* include “newline” characters `\n` in your `GR_report_error()` calls. If you fail to do this the lines of error messages will all be strung together on a single line!

```
...
/* first call GR_start_error() to begin the error message */
GR_start_error("trouble()",rcsid,__FILE__,__LINE__);
/* then call GR_report_error() as many times as desired */
GR_report_error("The GR_report_error() function is like printf().\n");
GR_report_error("It can have no arguments.\n");
GR_report_error("It can have a float argument %f\n",x1);
GR_report_error("Or any mixture of valid types %d %d %s\n",i1,i2,stringptr);
/* finally, call GR_end_error() to terminate the error message */
GR_end_error();
...
```

The use of these three routines as shown will print out, for example:

```
GRASP: Message from function trouble() at line number 123 of file "source.c".
The GR_report_error() function is like printf().
It can have no arguments.
It can have a float argument 5.43210
Or any mixture of valid types -2 17 the string pointed to by the pointer
$Id: man_utility.tex,v 1.38 1999/07/11 21:22:18 ballen Exp $
$Name: RELEASE_1_9_8 $
```

In particular, the line number, release number, and file name are all filled in automatically.

16.1.1 Reporting Errors In GRASP Code

Reporting errors from within GRASP code is simple: every legal GRASP error report consists of exactly one call to `GR_start_error()`, zero or more calls to `GR_report_error()`, and exactly one call to `GR_end_error()`. Failing to call the functions in exactly this order will cause an assert to fail and the program to abort; this allows the code in the handler routines to safely assume that they will be called in the correct order.

`GR_start_error()` takes four arguments specifying the name of the function in which the error occurred, the RCS ID of the GRASP file which contains the function’s source, the name of that file, and the line number on which the error report began. In practice providing this data is easy. Every GRASP file should declare a static string ‘rcsid’ containing the ID, and ANSI C defines the macros `__FILE__` and `__LINE__` which expand to the file name and line number, respectively. Within a given file only the function name parameter will change from call to call.

The arguments to `GR_report_error()` are exactly the same as those to `printf()`: a familiar `printf`-style format string followed by a variable number of arguments. There is no file pointer as in `fprintf()` because the errors may not be printed to a file or the screen but handled in some completely different way determined by the calling program. Repeated calls are a convenient way to build up the complete message just as with printing to the screen with `printf()`. Finally, a call to `GR_end_error()` ends the report. It requires no arguments.

The GRASP source code provides many examples of the use of these functions, and they are documented by example in Section 16.1. Because of the stereotyped calling sequence and arguments involved, an efficient technique for GRASP library programmers is to paste in the calls from another location in the source file or from a handy template kept in a scratch file so that only the function name and actual message needs editing for each case.

16.1.2 How GRASP Error Reports Are Handled

Programs vary widely in how they notify the user of errors. A simple command-line program will probably print the messages to the screen or a file, while a program with a graphical interface will likely send them to an error log window or to a dialog window. GRASP allows arbitrary error handlers to be specified, but the default provided will probably suffice for most GRASP application programmers.

Calling `GR_set_errors_enabled()` with an argument of false (zero) will suppress all GRASP error messages regardless of the actual handlers used, while a true argument will enable them again. `GR_errors_enabled()` returns true if errors are currently enabled and false otherwise. Errors always start out enabled unless the environment variable 'GRASP_NODEBUG' exists (its value is irrelevant), in which case they are disabled by default. Errors may be disabled within a handler, but the change will not take place until the next `GR_start_error()` call.

By default, the default error handlers print errors to `stderr`. The function, file, and line number are reported in a standard header format, followed by the message itself printed as though each call to `GR_report_error(format, args)` was a call to `fprintf(stderr, format, args)`, and ending with the RCS ID of the file containing the function source. If the environment variable 'GRASP_ERRORFILE' exists and its value is the pathname of a file which can be opened for appending that filename is used instead of `stderr`. Finally, if in addition 'GRASP_ERRORFILE_OVERWRITE' exists (its value is irrelevant) the file is overwritten rather than appended to.

16.1.3 Customizing The Default Handlers

GRASP error reports may be customized by modifying the behavior of the default handler functions or by replacing them entirely. By default, when `GR_start_error()` is called it checks to see if there is a default file name set; if so, that file is opened for appending. If not (that is, if the filename has been set to `NULL`), or if the file cannot be opened, it then checks to see if an error `FILE*` has been set. If so, that file is used. If not, then the handler fails since it has no way to report the error. The default `GR_report_error()` prints to whatever file `GR_start_error()` chose to use, and `GR_end_error()` prints the RCS ID. If the file was opened by name in `GR_start_error()` then `GR_end_error()` also closes it.

The default `FILE*` can be set and examined with `GR_set_error_file()` and `GR_get_error_file()`, and the default filename can be set and examined with `GR_set_error_file_name()` and `GR_get_error_file_name()`. The default behavior can be restored with `GR_set_error_file(stderr)` and/or `GR_set_error_file_name(NULL)`. These functions may be called at any time except during an error report (between the calls to `GR_start_error()` and `GR_end_error()`), when an `assert` would fail. (This is a non-issue in practice because the file/filename could only be changed during a report if the file or filename is set in GRASP code itself, if the error reporting functions are called in user code as well as in GRASP, or if a custom handler changes the file/filename and

then calls the default handler. The first possibility is strictly forbidden, and the others are discouraged because they can lead to confusion and subtle bugs.)

The reason for this somewhat complex system is safety. If a specified file name cannot be opened errors can still be reported to the FILE*, probably stderr. If an error file name has been set, the file is opened and closed for each report so that if the program crashes as much of the error log as possible is preserved on disk.

When using GR_set_error_file(), the calling program is responsible for opening the file for writing before the call and closing it (if necessary); GRASP will simply assume that the file is always available for output. A NULL file pointer is allowed, in which case calls to the default error handler will cause an assert to fail unless an error file name has been set.

When a file name is set with GR_set_error_file_name(), GRASP immediately attempts to open the file. If the erasefile parameter is TRUE (nonzero) the file will be opened for writing, if it is false (zero) it will be opened for appending. If the open succeeds, the name is copied and stored for future use (this means that the function can be safely called with locally allocated storage), an identifying start-up message and the time is written to the log file (even if error reporting is disabled), the file is closed, and the function returns true. When appending to a non-empty file it also writes a separator line so that reports from different runs are more easily distinguished. If the open fails, the filename is left unchanged (if a previous one existed), an error is reported in the usual way (unless error messages are suppressed), and the function returns false. Setting a NULL filename means that the FILE* should be used instead; the erasefile parameter is ignored in this case and the call always succeeds. As with GR_set_error_file(), the filename may not be changed during a report.

It is easy to see how the default behavior is obtained using these handlers. The default FILE* is stderr; conceptually a call to GR_set_error_file(stderr) occurs before the program begins. Similarly, a conceptual call to GR_set_error_file_name(filename, erase) occurs before program execution, with filename having the value of the environment variable GRASP_ERRORFILE if it exists and NULL otherwise, and erase true if the environment variable GRASP_ERRORFILE_OVERWRITE exists and false otherwise.

16.1.4 Writing Custom Error Handlers

Internally, GR_error keeps three pointers of type GR_start_error_type, GR_report_error_type, and GR_end_error_type (defined in grasp.h) which point to the current start_error, report_error, and end_error handlers, respectively. When the three error handlers GR_start_error(), GR_report_error(), and GR_end_error() are called, they in turn check to see that they are called in the proper order and then call the function pointed to by the corresponding function pointer if (and only if) the function pointer is non-NULL (so that for convenience if a particular handler is not necessary a dummy routine is not required) and if errors are currently enabled (so disabling errors works regardless of the handler). By default the handler pointers simply reference GR_default_start_error(), GR_default_report_error(), and GR_default_end_error(), which actually implement the default behavior described above. For convenience they may be restored with GR_restore_default_handlers() as well as GR_set_error_handlers().

The functions GR_set_error_handlers() and GR_get_error_handlers() set and examine the handler's current values, so that by writing the proper functions GRASP's error reports can be customized in any way desired. The GR_..._error_type typedef's in grasp.h illustrate the proper function prototypes. Note that GR_report_error_type functions take a va_list as their second argument rather than '...'; for convenience GR_report_error() creates the list and calls va_start() beforehand and calls va_end() afterwards so the handler need only deal with the list itself. In the common case where the message will simply be printed to a file, the va_list may be passed directly to one of the v...printf functions. The only restriction is that the handlers may not be changed between calls to GR_start_error() and GR_end_error(), but just as when changing the default error files this should not be a problem in practice.

The default handlers in `src/utility/GR_error.c` and their supporting routines are good examples of how GRASP error handlers are written. They are special only in that they are initialized and set automatically; otherwise, they use only features available to any handler. Most of their code provides the ability to switch files easily and safely; writing a custom handler that does not need this generality is quite straightforward. `GR_is_reporting()` returns true if a report is in progress and is sometimes useful when writing custom handlers.

16.1.5 Functions: GR_start_error(), GR_report_error(), GR_end_error()

These three functions are the GRASP error handlers. Their use is best illustrated by example. A typical usage is shown below – a fragment taken from a fictitious routine called “trouble()”.

```
...
GR_start_error("trouble()", rcsid, __FILE__, __LINE__);
GR_report_error("The GR_report_error() function is like printf().\n");
GR_report_error("It can have no arguments.\n");
GR_report_error("It can have a float argument %f\n", x1);
GR_report_error("Or any mixture of valid types %d %d %s\n", i1, i2, stringptr);
GR_end_error();
...
```

The use of these three routines as shown will print out, for example:

```
GRASP: Message from function trouble() at line number 123 of file "source.c".
The GR_report_error() function is like printf().
It can have no arguments.
It can have a float argument 5.43210
Or any mixture of valid types -2 17 the string pointed to by the pointer
$Id: man_utility.tex,v 1.38 1999/07/11 21:22:18 ballen Exp $
$Name: RELEASE_1_9_8 $
```

In particular, the line number, release number, and file name are all filled in automatically. The environment variables that govern the behavior of the default error handler are shown in Table 10 below.

Environment variable.	How to set, and effect obtained.
GRASP_NODEBUG	setenv GRASP_NODEBUG turns off error messages.
GRASP_ERRORFILE	setenv GRASP_ERRORFILE thisfile sends errors to file “thisfile”.
GRASP_ERRORFILE_OVERWRITE	setenv GRASP_ERRORFILE_OVERWRITE errors don’t accumulate in file.

Table 10: The behavior of the error handler is determined by three environment variables, which can be set and un-set using the shell commands setenv and unsetenv. These permit the error messages to be turned off, saved in a file, and control the file name and its over-write properties.

16.2 Function: grasp_open()

FILE* grasp_open(const char *environment_variable, const char *shortpath, const char *mode)

This routine provides a simple mechanism for obtaining the pointer to a data or parameter file. It is called with two character strings. One of these is the name of an environment variable, for example GRASP_DATAPATH or GRASP_PARAMETERS. The second argument is the "tail end" of a path name. The routine then constructs a path name whose leading component is determined by the environment variable and whose tail end is determined by the short path name. grasp_open() opens the file (printing useful error messages if this is problematic) and returns a pointer to the file.

The arguments are:

environment_variable: Input. Pointer to a character string containing the name of the environment variable.

shortpath: Input. Pointer to a character string containing the remainder of the path to the file.

mode: Input. File mode. Pointer to a character string containing "r" if you want to read the file, "w" if you want to write to the file, and so on. A list of the possible modes is:

r or rb	open file for reading
w or wb	truncate to zero length or create file for writing
a or ab	append; open or create file for writing at end-of-file
r+ or rb+ or r+b	open file for update (reading and writing)
w+ or wb+ or w+b	truncate to zero length or create file for update
a+ or ab+ or a+b	append; open or create file for update, writing at end-of-file

As a simple example, if the environment variable GRASP_PARAMETERS is set to /usr/local/data/14nov94.2 and one calls grasp_open("GRASP_PARAMETERS", "channel.0", "r") then the routine opens the file /usr/local/data/14nov94.2/channel.0 for reading and returns a pointer to it.

16.3 Function: avg_spec ()

```
void avg_spec(float *data, float *average, int npoint, int *reset, float srate, float decaytime, int windowtype, int overlap)
```

This routine calculates the power spectrum of the (time-domain) input stream `data[]`, averaged over time with a user-set exponential decay, several possible choices of windowing and the possibility to overlap data in subsequent calls.

The arguments are:

`data`: Input. The time domain input samples are contained in `data[0 . . N-1]`, with the data sample at time $t = n\Delta t$ contained in `data[n]`.

`average`: Output. The one sided power spectrum is returned in `average[0, . . N/2-1]`. The value of `average[m]` is the average power spectrum at frequency

$$f = \frac{m \times \text{srate}}{N}. \quad (16.3.1)$$

We do not output the value of the average at the Nyquist frequency, which would be the (non-existent) array element `average[N]`. The units of `average[]` are `data[]2/Hz`. Note: the elements of `average[]` must not be changed in between successive calls to `avg_spec ()`.

`npoint`: Input. The number of points `npoint = N` input. This must be an integer power of two.

`reset`: Input. If set to zero, then any past contribution to the average power spectrum is initialized to zero, and a new average is begun with the current input data.

`srate`: Input. The sample rate $1/\Delta t$ of the input data, in Hz.

`decaytime`: Input. The characteristic (positive) decay time τ in seconds, to use for the moving (exponentially-decaying) average described below. If no averaging over time is wanted, simply set `decaytime` to be small compared to $N\Delta t$.

`windowtype`: Input. Sets the type of window used in power spectrum estimation. Rectangular windowing (i.e., no windowing) is `windowtype=0`, Hann windowing is `windowtype=1`, Welch windowing is `windowtype=2` and Bartlett windowing is `windowtype=3`. See [1] for a discussion of windowing and the definitions of these window types.

`overlap`: Input. Must be either zero or unity. If set to unity, then the data is overlapped by $N/2$ points with previous data (see below for a description of the overlapping procedure). When set to zero no overlapping is performed.

The methods used in this routine are quite similar to those used in the *Numerical Recipes* [1] routine `spectrm ()`, and the reader interested in the details of this routine should first read the corresponding section of [1]. Note that to reproduce (exactly) the procedure described in *Numerical Recipes* [1] one must have `npoint=2×M` where `M` is the variable used in the procedure `spectrm ()`, and the decay time must be very large (so that the two successive spectra are equally weighted). If the data being passed is not continuous from one call to the next, set `overlap=0`.

One frequently wants to do a moving-time average of power spectra, for example to see how the noise spectral properties of an interferometer are changing with time. This is accomplished in `avg_spec ()` by averaging the spectrum with an exponentially-decaying average. Let $A_t(f)$ denote the average power

spectrum as a function of frequency f , at time t . Then the exponentially-decaying average $\langle A(f) \rangle_t$ at time t is defined by

$$\langle A(f) \rangle_t = \frac{\int_{-\infty}^t dt' A_{t'}(f) e^{-(t-t')/\tau}}{\int_{-\infty}^t dt' e^{-(t-t')/\tau}}, \quad (16.3.2)$$

where τ is the characteristic decay time over which an impulse in the power spectrum would decay. In our case, we wish to average the power spectra obtained in the n th pass through the averaging routine. The discrete analog of the previous equation (16.3.2) is

$$\langle A(f) \rangle_N = \frac{\sum_{n=0}^N A_n(f) e^{-\alpha(N-n)}}{\sum_{n=0}^N e^{-\alpha(N-n)}}. \quad (16.3.3)$$

Here,

$$\alpha = \frac{\text{npoint}}{\text{srate} \times \text{decaytime}} \quad (16.3.4)$$

is determined by the averaging time desired. The average defined by (16.3.3) can be easily determined by a recursion relation. We denote the the normalization factor by

$$\mathcal{N}_N = \sum_{n=0}^N e^{-\alpha(N-n)}. \quad (16.3.5)$$

It obeys the (stable) recursion relation $\mathcal{N}_N = 1 + e^{-\alpha} \mathcal{N}_{N-1}$ together with the initial condition $\mathcal{N}_{-1} = 0$. The exponentially-decaying average then satisfies the (stable) recursion relation

$$\langle A(f) \rangle_N = e^{-\alpha} \frac{\mathcal{N}_{N-1}}{\mathcal{N}_N} \langle A(f) \rangle_{N-1} + \frac{A_N(f)}{\mathcal{N}_N} \quad \text{for } N = 0, 1, 2, \dots \quad (16.3.6)$$

(no initial condition is needed). The routine `avg_spec()` computes the exponentially decaying average by implementing these recursion relations for $\langle A(f) \rangle_N$ and \mathcal{N}_N .

The units of the output array `average[]` are the square of the units of the input array `data[]` per Hz, i.e.

$$\text{units}(\text{average}[]) = (\text{units}(\text{data}[]))^2 / \text{Hz}. \quad (16.3.7)$$

The example program `calibrate` described earlier makes use of the routine `avg_spec()`.

Authors: Bruce Allen, ballen@dirac.phys.uwm.edu and Patrick Brady patrick@tapir.caltech.edu

Comments: See comments for `calibrate`. *Warning:* If `overlap` is turned on, and you pass `avg_spec()` sets of points that are not continuous, you will introduce discontinuous jumps between the data sets, and add lots of peculiar high-frequency garbage to the spectrum.

16.4 Function: binshort()

```
void binshort(short *input,int ninput,double *bins,int offset)
```

This function performs the “binning” which is needed to study the statistics of an array of short integers, such as the output of a 12 or 16 bit analog-to-digital converter. Its output is a histogram showing the number of times that a particular value occurred in an input array. Note that this routine *increments* the output histogram, so that you can use it for accumulating statistics of a particular variable.

The arguments are:

`input`: Input. This routine makes a histogram of the values `input[0..ninput-1]`.

`ninput`: Input. The number of elements in the previous array.

`bins`: Output. Upon return from the function, this array contains a histogram showing the probability distribution of the values `input[0..ninput-1]`. The array element `bins[offset]` is incremented by the number of elements x of `input[]` that had value $x = 0$. The array element `bins[offset+i]` is incremented by the number of elements x of `input[]` that had value $x = i$. If the output of your 16 bit ADC ranges from -32,768 to +32,767 and `nbins` has value $2^{16} = 65,536$ then you would want `offset = 32,768`. For a 12-bit ADC you would probably want `nbins = 2^{12} = 4096`, and depending upon the sign conventions either `offset = 2047` or `offset = 2048`.

`offset`: Input. The offset defined above.

Note that in the interests of speed and efficiency this routine does *not* check that your values lie within range. So if you try to bin a value that lies outside of the range $-\text{offset}, -\text{offset} + 1, \dots, \text{offset} - 1$ you may end up over-writing another array! You’ll then spend unhappy hours trying to locate the source of bizarre unpredictable behavior in your code, when you could be doing better things, like seeing if your ADC has dynamic range problem (reaches the end-point values too often) or has a mean value of zero (even with AC-coupled inputs the ADC may have substantial DC offset).

Authors: Bruce Allen, ballen@dirac.phys.uwm.edu

Comments: None.

16.5 Function: `is_gaussian()`

```
int is_gaussian(short *array, int n, int min, int max, int print)
```

This is a quick and robust test to see if a collection of values has a probability distribution that is consistent with a Gaussian normal distribution (“normal IFO operation”), or if the collection of values contains “outlier” points, indicating that the set of values contains “pulses”, “blips” and other “obvious” exceptional events that “stick out above the noise” (caused by bad cabling, alignment problems, or other short-lived transient events).

The arguments are:

`array`: Input. The values whose probability distribution is examined are `array[0..n-1]`.

`n`: Input. The length of the previous array.

`min`: Input. The minimum value that the input values *might* assume. For example, if `array[]` contains the output of a 12-bit analog-to-digital converter, one might set `min=-2048`. Of course the minimum value in the input array might be considerably larger than this (i.e., closer to zero!) as it should be if the ADC is being operated well within its dynamic range limits. If you’re not sure of the smallest value produced in `array[]`, set `min` smaller (i.e., more negative) than needed; the only cost is storage, not computing time.

`max`: Input. The maximum value that the input values *might* assume. For example, if `array[]` contains the output of a 12-bit analog-to-digital converter, one might set `max=2047`. The previous comments apply here as well: set `max` larger than needed, if you are not sure about the largest value contained in `array[]`.

`print`: Input. If this is non-zero, then the routine will print some statistical information about the distribution of the points.

The value returned by `is_gaussian()` is 1 if the distribution of points is consistent with a Gaussian normal distribution with no outliers, and 0 if the distribution contains outliers.

The way this is determined is as follows (we use x_i to denote the array element `array[i]`):

- First, the mean value \bar{x} of the distribution is determined using the standard estimator:

$$\bar{x} = \frac{1}{n} \sum_{i=0}^{n-1} x_i. \quad (16.5.1)$$

- Next, the points are binned into a histogram $N[v]$. Here $N[v]$ is the number of points in the array that have value v . The sum over the entire histogram is the total number of points: $\sum_i N[i] = n$.
- Then the standard deviation s is estimated in the following robust way. It is the smallest integer s for which

$$\sum_{i=-s}^s N[i + \bar{x}] > n \operatorname{erf}(1/\sqrt{2}) = n \frac{1}{\sqrt{2\pi}} \int_{-1}^1 e^{-x^2/2} dx. \quad (16.5.2)$$

This value of s is a robust estimator of the standard deviation; the range of $\pm s$ about the mean includes 68% of the samples. (Note that since the values of x_i are integers, we replace \bar{x} by the closest integer to it, in the previous equation).

- Next, the number of values in the range from one standard deviation to three standard deviations is found, and the number of values in the range from three to five standard deviations is found. This is compared to the expected number:

$$n(\operatorname{erfc}(3/\sqrt{2}) - \operatorname{erfc}(5/\sqrt{2})). \quad (16.5.3)$$

- If there are points more than five standard deviations away from the mean, or significantly more points in the 3 to 5 standard deviation range than would be expected for a Gaussian normal distribution, then `is_gaussian()` returns 0. If the numbers of points in each range is consistent with a Gaussian normal distribution, then `is_gaussian()` returns 1.

Authors: Bruce Allen, ballen@dirac.phys.uwm.edu

Comments: This function should be generalized in the obvious way, to look at one sigma wide bins in a more systematic way. It can eventually be replaced by a more rigorously characterized test to see if the distribution of sample values is consistent with the normal IFO operation.

16.6 Function: clear()

```
void clear(float *array,int n,int spacing)
```

This routine clears (sets to zero) entries in an array.

The arguments are:

array: Ouput. This routine clears elements `array[0]`, `array[spacing]`, ..., `array[(n-1)*spacing]`.

n: Input. The number of array elements that are set to zero.

spacing: Input. The spacing in the array between successive elements that are set to zero.

Authors: Bruce Allen, ballen@dirac.phys.uwm.edu

Comments: None.

16.7 Function: product()

void product(float *c, float *a, float *b, int ncomplex) This routine takes as input a pair of arrays a and b containing complex numbers. It multiplies a with b , placing the result in c , so that $c = a \times b$. The arguments are:

- a: Input. An array of N complex numbers $a[0..2N-1]$ with $a[2j]$ and $a[2j+1]$ respectively containing the real and imaginary parts.
- b: Input. An array of N complex numbers $b[0..2N-1]$ with $b[2j]$ and $b[2j+1]$ respectively containing the real and imaginary parts.
- c: Output. The array of N complex numbers $c[0..2N-1]$ with $c[2j]$ and $c[2j+1]$ respectively containing the real and imaginary parts of $a \times b$.

ncomplex: Input. The number N of complex numbers in the arrays.

Note that the two input arrays $a[]$ and $b[]$ can be the same array; or the output array $c[]$ can be the same as either or both of the inputs. For example, the following are all valid:

- product(c, a, a, n), which performs the operation $a^2 \rightarrow c$.
- product(a, a, b, n), which performs the operation $a \times b \rightarrow a$.
- product(a, b, a, n), which performs the operation $a \times b \rightarrow a$.
- product(a, a, a, n), which performs the operation $a^2 \rightarrow a$.

Note also that this routine does not allocate any memory itself - your input and output arrays must be allocated before calling product().

Author: Bruce Allen, ballen@dirac.phys.uwm.edu

Comments: None.

16.8 Function: `productc()`

`void productc(float *c, float *a, float *b, int ncomplex)` This routine takes as input a pair of arrays a and b containing complex numbers. It multiplies a with the complex-conjugate of b , placing the result in c , so that $c = a \times b^*$. The arguments are:

- a: Input. An array of N complex numbers $a[0..2N-1]$ with $a[2j]$ and $a[2j+1]$ respectively containing the real and imaginary parts.
 - b: Input. An array of N complex numbers $b[0..2N-1]$ with $b[2j]$ and $b[2j+1]$ respectively containing the real and imaginary parts.
 - c: Output. The array of N complex numbers $c[0..2N-1]$ with $c[2j]$ and $c[2j+1]$ respectively containing the real and imaginary parts of $a \times b^*$.
- `ncomplex`: Input. The number N of complex numbers in the arrays.

Note that the two input arrays $a[]$ and $b[]$ can be the same array; or the output array $c[]$ can be the same as either or both of the inputs. For example, the following are all valid:

- `productc(c, a, a, n)`, which performs the operation $|a|^2 \rightarrow c$.
- `productc(a, a, b, n)`, which performs the operation $a \times b^* \rightarrow a$.
- `productc(a, b, a, n)`, which performs the operation $a^* \times b \rightarrow a$.
- `productc(a, a, a, n)`, which performs the operation $|a|^2 \rightarrow a$.

Note also that this routine does not allocate any memory itself - your input and output arrays must be allocated before calling `productc()`.

Author: Bruce Allen, ballen@dirac.phys.uwm.edu

Comments: None.

16.9 Function: ratio()

`void ratio(float *c, float *a, float *b, int ncomplex)` This routine takes as input a pair of arrays a and b containing complex numbers. It divides a by b , placing the result in c , so that $c = a/b$. The arguments are:

- a: Input. An array of N complex numbers $a[0..2N-1]$ with $a[2j]$ and $a[2j+1]$ respectively containing the real and imaginary parts.
 - b: Input. An array of N complex numbers $b[0..2N-1]$ with $b[2j]$ and $b[2j+1]$ respectively containing the real and imaginary parts.
 - c: Output. The array of N complex numbers $c[0..2N-1]$ with $c[2j]$ and $c[2j+1]$ respectively containing the real and imaginary parts of a/b .
- `ncomplex`: Input. The number N of complex numbers in the arrays.

Note that the two input arrays $a[]$ and $b[]$ can be the same array; or the output array $c[]$ can be the same as either or both of the inputs. For example, the following are all valid:

`ratio(c, a, a, n)`, which (very inefficiently) sets every element of c to $1 + 0i$.

`ratio(a, a, b, n)`, which performs the operation $a/b \rightarrow a$.

`ratio(a, b, a, n)`, which performs the operation $b/a \rightarrow a$.

`ratio(a, a, a, n)`, which (very inefficiently) sets every element of a to $1 + 0i$.

This routine is particularly useful when you want to reconstruct the raw interferometer output $\widetilde{C}_0(f)$ that would have produced a particular interferometer displacement $\widetilde{\Delta}l(f)$ (see for example `normalize_gw()` in Section 3.12). This occurs for example if you are “injecting” chirps into the raw interferometer output; they first need to be deconvolved with the response function of the instrument. One can invert this equation using `ratio()` since $\widetilde{\Delta}l(f) = R(f)\widetilde{C}_0(f) \Rightarrow \widetilde{C}_0(f) = \widetilde{\Delta}l(f)/R(f)$.

Author: Bruce Allen, ballen@dirac.phys.uwm.edu

Comments: None.

16.10 Function: reciprocal()

```
void reciprocal(float *b, float *a, int ncomplex)
```

This routine computes the array $b = 1/a$ for an input array a containing complex numbers. The arguments are:

a: Input. The array of N complex numbers $a[0..2N-1]$ with $a[2j]$ and $a[2j+1]$ respectively containing the real and imaginary parts.

b: Output. The array of N complex numbers $b[0..2N-1]$ with $b[2j]$ and $b[2j+1]$ respectively containing the real and imaginary parts of $1/a$.

ncomplex: Input. The number N of complex numbers in the arrays.

Note that the arrays $a[]$ and $b[]$ can be the same.

In order to reduce the potential for overflows (since floating point arithmetic was used), the reciprocals of the complex numbers were computed according to the following formula:

$$b = \frac{1}{a} = \frac{1}{x + iy} = \begin{cases} \frac{1 - i(y/x)}{x + y(y/x)} & |x| > |y| \\ \frac{(x/y) - 1}{x(x/y) + y} & |x| \leq |y|. \end{cases} \quad (16.10.1)$$

Author: Jolien Creighton, jolien@tapir.caltech.edu

Comments: None.

16.11 Function: graph()

```
void graph(float *array,int n,int spacing)
```

This is a useful function for debugging. It pops up a graph on the computer screen (using the graphing program `xmgr`) showing a graph of some array which you happen to want to look at.

The arguments are:

`array`: Input. The array that you want a graph of.

`n`: Input. The number of array elements that you want to graph.

`spacing`: Input. The spacing of the array elements that you want to graph. The elements graphed are `array[0]`, `array[spacing]`, `array[2*spacing]`, ..., `array[(n-1)*spacing]`.

This function is a handy way to get a quick look at the contents of some array. It writes the output to a temporary file and then starts up `xmgr`, reading the input from the file. The x values are evenly spaced integers from 0 to $n-1$. The y values are the (subset of) points in `array[]`. If your array contains real data, you might want to use `spacing=1`. If your array contains complex data (with real and imaginary parts interleaved) you will use `spacing=2`, and make separate calls to see the real and imaginary parts. For example if `complex[0..2047]` contains 1024 complex numbers, then:

```
graph(complex,1024,2) (view 1024 real values)
```

```
graph(complex+1,1024,2) (view 1024 imaginary values)
```

Note that in order not to produce too much garbage on the screen, any output or error messages from `xmgr` are tossed into `/dev/null`!

Authors: Bruce Allen, ballen@dirac.phys.uwm.edu

Comments: None.

16.12 Function: graph_double()

```
void graph_double(double *array,int n,int spacing)
```

This is a useful function for debugging, and exactly like the function `graph()`, except that it's intended for double precision floating point numbers. It pops up a graph on the computer screen (using the graphing program `xmgr`) showing a graph of some array which you happen to want to look at.

The arguments are:

`array`: Input. The array that you want a graph of.

`n`: Input. The number of array elements that you want to graph.

`spacing`: Input. The spacing of the array elements that you want to graph. The elements graphed are `array[0]`, `array[spacing]`, `array[2*spacing]`, ..., `array[(n-1)*spacing]`.

This function is a handy way to get a quick look at the contents of some array. It writes the output to a temporary file and then starts up `xmgr`, reading the input from the file. The x values are evenly spaced integers from 0 to $n-1$. The y values are the (subset of) points in `array[]`. If your array contains real data, you might want to use `spacing=1`. If your array contains complex data (with real and imaginary parts interleaved) you will use `spacing=2`, and make separate calls to see the real and imaginary parts. For example if `complex[0..2047]` contains 1024 complex numbers, then:

```
graph(complex,1024,2) (view 1024 real values)
```

```
graph(complex+1,1024,2) (view 1024 imaginary values)
```

Note that in order not to produce too much garbage on the screen, any output or error messages from `xmgr` are tossed into `/dev/null`!

Authors: Bruce Allen, ballen@dirac.phys.uwm.edu

Comments: None.

16.13 Function: graph_short()

```
void graph_short(short *array, int n)
```

This is a useful function for debugging, and exactly like the function `graph()`, except that it's intended for short integer values. It pops up a graph on the computer screen (using the graphing program `xmgr`) showing a graph of some array which you happen to want to look at.

The arguments are:

`array`: Input. The array that you want a graph of.

`n`: Input. The number of array elements that you want to graph. The elements graphed are `array[0..n-1]`.

This function is a handy way to get a quick look at the contents of some array. It writes the output to a temporary file and then starts up `xmgr`, reading the input from the file. The x values are evenly spaced integers from 0 to $n-1$. The y values are the points in `array[]`.

Note that in order not to produce too much garbage on the screen, any output or error messages from `xmgr` are tossed into `/dev/null!`

Authors: Bruce Allen, ballen@dirac.phys.uwm.edu

Comments: None.

16.14 Function: sgraph()

sgraph(short *array,int n,char *name,int filenumber)

This routine writes the elements of a short array into a file so that they may be viewed later with a graphing program like xmgr.

The arguments are:

array: Input. The array that you want to graph.

n: Input. The number of array elements that you want to graph. The elements used are array[0..n-1].

name: Input. Used to construct the output file name.

filenumber: Input. The value of *y* used to construct the output file name.

This function produces an output file with two columns, containing:

```
0    array[0]
1    array[1]
...
n-1  array[n-1]
```

The name of this file is: name.y where *y* is the integer specified by filenumber. Note that if *y* < 1000 then *y* is "expanded" or "padded" to three digits. For example, calling

```
sgraph(array,1024,"curious",9)
```

will produce the file

```
curious.009
```

containing 1024 lines.

Authors: Bruce Allen, ballen@dirac.phys.uwm.edu

Comments: None.

16.15 Function: audio()

```
void audio(short *array, int n)
```

Makes a workstation play music! This is guaranteed to work on SUN machines, and may also work on others.

The arguments are:

`array`: Input. The array that you want to hear.

`n`: Input. The number of array elements that you want to hear. The elements used are `array[0..n-1]`.

It doesn't take much experience before you find out that an interferometer can do funny things that you can't see in the data stream, if you just graph the numbers. However in many cases you can *hear* the peculiar events. This function works only on Sun workstations with a CD-sound quality chipset, that can handle 16 bit linear PCM audio. It creates a temporary file, then pipes it though the Sun utility `audioplay`. The sample rate is assumed to be 9600 Hz.

Note that `audio()` *adjusts the volume* so that the loudest event (largest absolute value) in the data stream has a (previously fixed, by us!) maximum amplitude. So the "background level" of the sound will depend upon the amplitude of the most obnoxious pings, blips, bumps, scrapes or howlers in the data set.

On a machine not equipped with the correct sound chip (for example a SparcStation 2) you can listen to the file, if you first convert it to a format that the chipset can handle. This can be done by taking the output of `audio()`, which is a file called `temp.au` and converting it to "voice" format. To do this, use the command:

```
audioconvert -f voice -o temp2.au temp.au
```

You can then listen to the sound using the command:

```
audioplay temp2.au
```

Warning: If you share your office with others, they will find the first few events that you listen to highly entertaining. After the first day however they will stop asking what you're listening to. After a few more days, their suggestions that you buy headphones will become more pointed. Respect this request.

Authors: Bruce Allen, ballen@dirac.phys.uwm.edu

Comments: This routine could be modified to permit a bit more freedom in setting the volume and/or the sample rate.

16.16 Example: makesounds program

The program `makesounds` demonstrates the `graph_short()` function, which uses `xmgr` to create graphs of an array of short integers, and the `audio()` function, which plays a waveform, expressed as an array of shorts, on a sound-capable workstation. This program generates three waveforms with which to demonstrate these utilities:

- a constant frequency Cosine wave with a quadratically varying amplitude, $t(T-t)\cos(2\pi*200*t)$, where T is the duration of the signal,
- a binary inspiral chirp for two 5.0 solar mass objects, starting from a frequency of 64 Hz and chirping to coalescence,
- and a Lai-Shapiro waveform from the supernova hang-up with a 1.4 solar mass core and rotating with an initial frequency of 1000 Hz.

Each waveform is displayed with `graph_short()` and played on your workstation with `audio()`.

```
/* Demonstrates the audio() and graph_short() with (1) 200 Hz Cosine wave
with a quadratically varying amplitude (2) 5.0-5.0 solar mass inspiral
chirp (3) Lai-Shapiro 1.4 solar mass: Displays graph and plays audio
for each. */

#include "grasp.h"
#include <unistd.h>
#define S_RATE 9600
#define NUM_PTS 10000
int i, chirp_pts;
float *wave, *dummy, wavemax, t_coal;
short *snd;
struct LS_physical_constants phys_const;

int main(){
    /* Allocate arrays */
    wave=(float *)malloc(NUM_PTS*sizeof(float));
    dummy=(float *)malloc(NUM_PTS*sizeof(float));
    snd=(short *)malloc(NUM_PTS*sizeof(float));

    /* Generate a 200 Hz Cosine wave with quadratically varying amplitude */
    for (i=0;i<NUM_PTS;i++) wave[i]=i*(NUM_PTS-i)*cos(2.0*M_PI*200.0*i/9600.0)/NUM_PTS;
    /* Convert wave to shorts, rescaling maximum amplitude to SHRT_MAX-2 */
    wavemax=0;
    for (i=0;i<NUM_PTS;i++) if (fabs(wave[i])>wavemax) wavemax=fabs(wave[i]);
    for (i=0;i<NUM_PTS;i++) snd[i]=(short)((SHRT_MAX-2)*wave[i]/wavemax);
    /* Graph and play waveform, then pause briefly */
    graph_short(snd, NUM_PTS);
    audio(snd, NUM_PTS);
    sleep(2);

    /* Same procedure for an inspiral chirp for a 5.0-5.0 solar mass system. */
    make_filters(5.0, 5.0, wave, dummy, 64.0, NUM_PTS, S_RATE, &chirp_pts, &t_coal, 4000, 2);
    wavemax=0;
    for (i=0;i<NUM_PTS;i++) if (fabs(wave[i])>wavemax) wavemax=fabs(wave[i]);
    for (i=0;i<NUM_PTS;i++) snd[i]=(short)((SHRT_MAX-2)*wave[i]/wavemax);
    graph_short(snd, NUM_PTS);
    audio(snd, NUM_PTS);
    sleep(2);
```

```
/* Define parameters for Lai-Shapiro waveform */
phys_const.mass=1.4;
phys_const.radius=10.0;
phys_const.distance=1.0;
phys_const.fmax=1000.0;
phys_const.inclination=phys_const.Phi_0=0;
/* Then generate, display and play the Lai-Shapiro waveform. */
LS_waveform(wave,phys_const,0.0,0.0,0.125*M_PI,1.0/S_RATE,NUM_PTS);
wavemax=0;
for (i=0;i<NUM_PTS;i++) if (fabs(wave[i])>wavemax) wavemax=fabs(wave[i]);
for (i=0;i<NUM_PTS;i++) snd[i]=(short)((SHRT_MAX-2)*wave[i]/wavemax);
graph_short(snd,NUM_PTS);
audio(snd,NUM_PTS);

return 0;
}
```

Author: Warren Anderson, warren@ricci.phys.uwm.edu

Comments: If the `audio()` function does not play sound on your workstation, replace it with `sound()` below, and use that to create sound files which you should be able to listen to in some way.

16.17 Function: sound()

sound(short *array,int n,char *name,int filenumber)

This is just like the function audio() except that it writes the sound data into a file of the form *.au.
The arguments are:

array: Input. The array that you want to hear.

n: Input. The number of array elements that you want to hear. The elements used are array[0..n-1].

name: Input. Used to construct the output file name.

filenumber: Input. The value of *y* used to construct the output file name.

This function produces an output file with 16-bit PCM linear coding, containing sound data. The name of the file is: name.y.au where *y* is the integer specified by filenumber. Note that if *y* < 1000 then *y* is "expanded" or "padded" to three digits. For example, calling

```
sound(array,4800,"growl",9)
```

will produce the file

```
growl.009.au
```

containing 1/2 second of sound.

Note: see the *Warning* that goes with audio().

Authors: Bruce Allen, ballen@dirac.phys.uwm.edu

Comments: This routine could be modified to permit a bit more freedom in setting the volume and/or the sample rate.

16.18 Example: translate

This example may be found in the `src/examples/examples_utility` directory of GRASP, and contains an example program which translates data from the “old 1994” Caltech 40-meter format described earlier, to the new LIGO/VIRGO frame format. Because this code provides an example of how the data is encoded in this new format, we have included the text of the translation code here. The frames produced by this translation contain about 5 seconds of data each, and are about half a megabyte in length. The number of frames in each data file is set by the

```
# define FRAMES_PER_FILE
```

at the top of the code. To run the utility, use the command

```
translate directory-name
```

where `directory-name` is the name of the directory in which the files `channel.0` to `channel.15` may be found. The FRAME format files produced by `translate` are labelled uniquely by the time at which the first data point in the first frame was taken. The choice of file names depends upon the version of the Frame library.

- For Frame library versions ≤ 3.30 , the files are labeled by their time (in Coordinated Universal Time or UTC). An example of such a file (produced by `translate`) is:

```
C1-94_10_15_06_18_02.
```

In the file name, 94 denotes the year (we will use 01 for 2001, etc.) and 10 denotes the month (labelled from 1 to 12). The hour ranges from 0 to 23 and in this examples is 06. The minutes (18) ranges from 0 to 59 and the seconds (02) ranges from 0 to 61 to include leap seconds but is normally in the range from 0 to 59. This naming convention will be used for any data files containing one second or more of data.

- For Frame library versions > 3.30 , the files are labeled by their GPS time. The relationship between different time coordinates is explained in Section 17. An example of the corresponding file (produced by `translate`) is:

```
C1-468915467.F
```

where the suffix means “frame”, and the integer is the number of seconds after Jan 6, 1980 00:00:00 UTC.

Here C1 denotes the Caltech 40-meter prototype. The names that will be used for the other sites are: H2 Hanford LIGO detectors (these will be stored in a single frame, together, with channel names to distinguish the 2km and 4km detectors, L for the Livingston LIGO detector V1 for the Virgo detector, G1 for the GEO detector, T for the Tama detector, S for the Glasgow detector, M for the Max-Planck detector, and A for the AIGO detector. In some cases (for example at Livingston and Hanford) additional frame files containing trend data, with typically one sample per channel per second, and one minute of data per frame, will be stored in files of the form `H-609637094.T`, with the suffix denoting “trend”.

```
/* GRASP: Copyright 1997,1998 Bruce Allen */
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "FrameL.h"
#include "grasp.h"
#define OLDNAMES 0 /* set to zero to use new channel names, 1 for old names */
#define LOCKLO 1
#define LOCKHI 10
#define CORRECTTIMESTAMPS 1 /* set to 1 to correct loss of timestamp resolution */
/* The compression method: None = 0; GZIP = 1; Diff = 2; Diff+GZIP = 3 */
```

```
#define COMPRESSION 3
/* The Level of GZIP compression used; Values between 1 and 9 allowed */
#define GZIP_LEVEL 1
/* the latest version of the frame lib that this code has been tested with */
#define FRAMELIB_TESTED 3.85

/* Each block of old-format data contains 5.07 secs of data. This
parameter determines how many of these old-format blocks (now a frame)
end up in each FRAME file. */
#define FRAMES_PER_FILE 32
/* earth's equatorial radius, in meters */
#define EQUATORIAL (6.37814e+06)
/* earth's ellipticity or flattening due to rotation */
#define FLAT (3.35281e-3)

/* the conversion from ADC counts to volts is: */
static char units[]="Units are 10 volts per 2048 counts. Range -2048 to +2047";

#if (OLDNAMES)
/* channel assignments before Nov 15, 1994 */
static char *prenov15[]={
    "IFO output", "", "", "microphone", "dc strain", "mode cleaner pzt",
    "seismometer", "", "", "", "TTL locked", "arm 1 visibility", "arm 2 visibility",
    "mode cleaner visibility", "slow pzt", "arm 1 coil driver"};

/* channel assignments after Nov 15, 1994 */
static char *postnov15[]={
    "IFO output", "magnetometer", "microphone", "", "dc strain", "mode cleaner pzt",
    "seismometer", "slow pzt", "power stabilizer", "",
    "TTL locked", "arm 1 visibility", "arm 2 visibility", "mode cleaner visibility",
    "", "arm 1 coil driver"};
#else
/* channel assignments before Nov 15, 1994 */
static char *prenov15[]={
    "IFO_DMRO", "", "", "IFO_Mike", "IFO_DCDM", "PSL_MC_V",
    "IFO_Seis_1", "", "", "", "IFO_Lock", "IFO_EAT", "IFO_SAT",
    "IFO_MCR", "IFO_SPZT", "SUS_EE_Coil_V"};

/* channel assignments after Nov 15, 1994 */
static char *postnov15[]={
    "IFO_DMRO", "IFO_Mag_x", "IFO_Mike", "", "IFO_DCDM", "PSL_MC_V",
    "IFO_Seis_1", "IFO_SPZT", "PSL_PSS", "",
    "IFO_Lock", "IFO_EAT", "IFO_SAT", "IFO_MCR",
    "", "SUS_EE_Coil_V"};
#endif

/* Program's only argument is the name of the directory containing old-format data */
int main(int argc, char* argv[] ) {
    char filename[256], name[256], hist[1024], *histnew, *buff, **chan_name;
    static char machinename[256]="";
    int i, code=1, num, large=50000, small=5000, n, first=1, firsttime=0, nlines;
    long buffSize;
    float fastrate=9868.4208984375, tblock, slowrate=986.84208984375, *real, *imag, *freq;
    double firstmsec=0.0, first_estimate, second_estimate, diff, dt, dtslow;
    float starttime=-100.0, guesstime;
    double currenttime=-200;
    int blockcount=0, channelsopen=0, expected;
    struct FrFile *outputfile=NULL;
    struct FrameH *frame;
```

```
struct FrAdcData *adc[16];
struct FrDetector *frdetect;
struct FrVect *framevec;
struct FrVect *framevecS;
struct FrStatData *staticdata;
struct FrStatData *staticdataS;
struct ld_binheader bin_header;
struct ld_mainheader main_header;
struct tm timetm,*gtime,gts;
time_t translate_time,calendartime;
FILE *fp[16],*fpsweptsine,*pipe;
void unhappyexit(int i);
int get_run_number(int firsttime);

/* print out some information about the library being used */
fprintf(stderr,"translate compiled with Frame header file version: FRAMELIB_VERSION=%.2f\n",
        FRAMELIB_VERSION);
#if (FRAMELIB_VERSION_INT>=370)
fprintf(stderr,"translate linked with Frame library archive version: FrLibVersion=%.2f\n",
        FrLibVersion(NULL));
if ((int)(1000*(FRAMELIB_VERSION-FrLibVersion(NULL))))
    fprintf(stderr,
            "WARNING: translate code linked to different run-time library than header file version!\n");
#endif
if (FRAMELIB_VERSION_INT!=100*FRAMELIB_VERSION)
    fprintf(stderr,
            "WARNING: in building this code FRAMELIB_VERSION_INT=%d != 100 x (FRAMELIB_VERSION=%.2f)\n",
            FRAMELIB_VERSION_INT,FRAMELIB_VERSION);
if (FRAMELIB_VERSION>FRAMELIB_TESTED)
    fprintf(stderr,"Warning: translate has only been tested with FRAMELIB_VERSION <= %.2f\n",
            FRAMELIB_TESTED);

/* initialize the frame system */
FrLibIni(NULL,NULL,2);
buffSize=1000000;
buff=malloc(buffSize);

/* create a frame */
frame=FrameHNew("C1");

/* assign detector structure: site location and orientation information */
#if (FRAMELIB_VERSION_INT<=237)
frame->detectRec=FrDetectorNew("real");
frdetect=frame->detectRec;
frdetect->latitude=34.1667;
frdetect->longitude=118.133;
frdetect->arm1Angle=180.0;
frdetect->arm2Angle=270.0;
frdetect->arm1Length=38.5;
frdetect->arm2Length=38.1;
#else
frame->detectProc=FrDetectorNew("real");
frdetect=frame->detectProc;
frdetect->latitudeD=34;
frdetect->latitudeM=10;
frdetect->latitudeS=0;
frdetect->longitudeD=118;
frdetect->longitudeM=8;
frdetect->longitudeS=0;
```

```
frdetect->armXazimuth=180.0;
frdetect->armYazimuth=270.0;
frdetect->armLength=38.3;
#endif

/* Correct for oblateness of earth, use reference spheroid with
flattening FLAT; EQUATORIAL is earth equatorial radius in meters.
Reference: eqns (4.13-14) in "Spacecraft attitude determination and
control", Ed. James R. Wortz, D. Reidel Publishing Co., Boston, 1985.
Note: this SHOULD be corrected to add in the height of Caltech above
sea level. */
/* angle measured down from the North pole */

#if (FRAMELIB_VERSION_INT<=237)
{
float theta;
theta=(M_PI/180.0)*(90.0-frdetect->latitude);
frdetect->altitude=EQUATORIAL*(1.0-FLAT*cos(theta)*cos(theta));
}
#else
frdetect->elevation=0.0 /*FILL IN THE CORRECT VALUE */;
#endif

/* now open files containing 40 meter data */
if (!argv[1] || argc!=2) unhappyexit(1);

/* step through all possible channels, seeing which channels have data */
for (i=0;i<16;i++) {
printf(name, "%s/channel.%d", argv[1], i);
fp[i]=fopen(name, "r");
if (fp[i]==NULL)
fprintf(stderr, "File %s unavailable. Skipping it...\n", name);
else
channelsopen++;
}

/* if there are no open files, then please exit with a warning message */
if (channelsopen==0) unhappyexit(1);

/* the sample times for the fast/slow channels */
dt=1.0/fastrate;
dtslow=1.0/slowrate;

/* Define 4 fast, 12 slow ADC channels (long strings of blanks needed - see below) */
for (i=0;i<16;i++)
if (fp[i]!=NULL)
if (i<4)
/* sample rates differ from fastrate, slowrate - see GRASP manual for details */
adc[i]=FrAdcDataNew(frame, " ", 50000.0*15.0/76.0, large, 16);
else
adc[i]=FrAdcDataNew(frame, " ", 5000.0*15.0/76.0, small, 16);

/* now loop over the input data, creating blocks of output data */
while (code>0) {
/* read a block of data */
for (i=0;i<16;i++) {
/* set size of data block */
n=(i<4)?large:small;
/* read data into frame short array */
```

```
    if (i<4 && fp[i]!=NULL)
        code=read_block(fp[i],&(adc[i]->data->dataS),&num,&tblock,&fastrate,0,&n,0,
            &bin_header,&main_header);
    else if (fp[i]!=NULL)
        code=read_block(fp[i],&(adc[i]->data->dataS),&num,&tblock,&slowrate,0,&n,0,
            &bin_header,&main_header);
}

/* if no data remains, we have found an error */
if (code==0) {
    fprintf(stderr,"Error in translation: unexpected end of data!\n");
    abort();
}

/* check the various sample times */
if (dt!=1.0/fastrate) fprintf(stderr,"Fast sample rates don't match!\n");
if (dtslow!=1.0/slowrate) fprintf(stderr,"Slow sample rates don't match!\n");

/* set time stamps for this block of data */
/* Put the local California time-of-day into a structure for later use */
timetm.tm_sec=main_header.tod_second;
timetm.tm_min=main_header.tod_minute;
timetm.tm_hour=main_header.tod_hour;
timetm.tm_mday=main_header.date_day;
timetm.tm_mon=main_header.date_month;
timetm.tm_year=main_header.date_year;
timetm.tm_wday=main_header.date_dow;
timetm.tm_yday=-1; /* info not available, but filled in by mktime */
timetm.tm_isdst=-1; /* info not available, but filled in by mktime */

/* Now convert the stored Calendar time into the right data type */
calendartime=main_header.epoch_time_sec;

/* Put the UTC time-of-day into a structure for later use */
gtime=gmtime(&calendartime);
gts=*gtime;

/* set the time stamp for the first data sample (more precise than header time) */
if (first) {
    firsttime=main_header.epoch_time_sec;
    firstmsec=0.001*main_header.epoch_time_msec;
    printf("UTC (gmtime) start time: %s",asctime(&gts));
    printf("          CA start time: %s\n",asctime(&timetm));

    /* assign the run number from 1,..,11 to the frame. */
    frame->run=get_run_number(firsttime);
    if (frame->run<1 || frame->run>11) unhappyexit(2);

    /* assign proper name to adc channel (to overwrite long blank space above) */
    if (frame->run<=2) {
        chan_name=prenov15;
        expected=11;
    }
    else {
        chan_name=postnov15;
        expected=13;
    }

    if (channelsopen!=expected) {
```

```
    fprintf(stderr, "Only found %d channels.  Expected %d\n", channelsopen, expected);
    exit(1);
}

for (i=0; i<16; i++)
    if (fp[i]!=NULL) {
        /* verify that name is correct */
        if (strcmp(chan_name[i], "")==0) {
            fprintf(stderr, "Channel %d is not recognized and has no name!\n", i);
            exit(1);
        }

        /* point to the correct channel name for this particular date, channel */
        strcpy(adc[i]->name, chan_name[i]);

        /* put in the physical volts/counts conversion */
#ifdef FRAMELIB_VERSION_INT<=237
        adc[i]->data->unit[0]=(char *)malloc((strlen(units)+1)*sizeof(char));
        strcpy(adc[i]->data->unit[0], units);
#else
        adc[i]->data->unitY=(char *)malloc((strlen(units)+1)*sizeof(char));
        strcpy(adc[i]->data->unitY, units);
#endif

        /* which ADC "crate" was this */
        adc[i]->crate=i;
    }
}

if (CORRECTTIMESTAMPS) {
    guesstime=currenttime+76.0/15.0;
    if (fabs(guesstime-tblock)>1.0) {
        starttime=tblock;
        blockcount=0;
    }
    currenttime=(blockcount+)*((double)76.0/15.0)+starttime;

    /* put the time stamp into the frame structure */
    currenttime+=firstmsec;
#ifdef FRAMELIB_VERSION_INT<330
    frame->UTimeS=firsttime+(int)currenttime;
    frame->UTimeN=(int)(1.e9*(currenttime-(int)currenttime));
    frame->dt=76.0/15.0;
#else
    frame->GTimeS=firsttime+(int)currenttime-UTCTOGPS;
    frame->GTimeN=(int)(1.e9*(currenttime-(int)currenttime));
    /*JKB: should be INT(TAI-UTC) */
    /* BA - for Nov 1994 - see GRASP manual on time defs */
    frame->ULeapS=29;
    frame->dt=(76.0/15.0);
#endif
}
else {
    /* put the time stamp into the frame structure */
    tblock+=firstmsec;
#ifdef FRAMELIB_VERSION_INT<330
    frame->UTimeS=firsttime+(int)tblock;
    frame->UTimeN=(int)(1.e9*(tblock-(int)tblock));
    frame->dt=num/slowrate;

```

```
#else
    frame->GTimeS=firsttime+(int)tblock-UTCTOGPS;
    frame->GTimeN=(int)(1.e9*(tblock-(int)tblock));
                                                    /*JKB: should be INT(TAI-UTC)*/
    frame->ULeapS=29;
                                                    /* BA - for Nov 1995 - see GRASP manual on time defs */
    frame->dt=(num/slowrate);
#endif

}

/* Localtime - UTC time in seconds */
frame->localTime=-8*3600;

/* frame->type[0]=0; */

/* put in the history information (only once per translation) */
if (first) {
    first=0;
    histnew=hist;
    time(&translate_time);

    /* get the name of the local machine */
    pipe=popen("hostname","r");
    if (pipe==NULL) {
        /* if we can't open the pipe, then list machine name as unknown */
        strcpy(machinename,"hostname undetermined");
    }
    else
        fscanf(pipe,"%s",machinename);

    histnew+=sprintf(histnew,"\nTranslation carried out by:\n");
    histnew+=sprintf(histnew,"    login: %s\n",getenv("LOGNAME"));
    histnew+=sprintf(histnew,"    user: %s\n",getenv("USER"));
    histnew+=sprintf(histnew,"    translation machine name: %s\n",machinename);
    histnew+=sprintf(histnew,"    directory: %s\n",getenv("PWD"));
    histnew+=sprintf(histnew,"    datapath: %s\n",argv[1]);
    histnew+=sprintf(histnew,"    translation program name: %s\n",argv[0]);
    histnew+=sprintf(histnew,"    source code name: %s\n","translate.c");
    histnew+=sprintf(histnew,"    Frame library header file (compile) version: %.2f\n",FRAMELIB_VERSION);
}
#if (FRAMELIB_VERSION_INT)>=370
    histnew+=sprintf(histnew,"    Frame library archive (link) version: %.2f\n",FrLibVersion(N));
#endif
histnew+=sprintf(histnew,"    translation date: %s\n",ctime(&translate_time));
FrHistoryAdd(frame,hist);

/* read the swept sine calibration files (only once per run) */
sprintf(name,"%s/swept-sine.ascii",argv[1]);
fpsweptsine=fopen(name,"r");
read_sweptsine(fpsweptsine,&nlines,&freq,&real,&imag);

/* copy swept sine calibration data into vector; see below for packing style */
#if (FRAMELIB_VERSION_INT)<=237)
    framevec=FrVectNew(FR_VECT_F,1,3*nlines,1.0,"Vifo/Vcoil");
#else
    framevec=FrVectNew(FR_VECT_4R,1,3*nlines,1.0,"Vifo/Vcoil");
#endif
for (i=0;i<nlines;i++) {
    framevec->dataF[i]=freq[i];
    framevec->dataF[i+nlines]=real[i];
}
```

```
        framevec->dataF[i+2*nlines]=imag[i];
    }

    /* then link the calibration data into the history structure */
#if (FRAMELIB_VERSION_INT<330)
    staticdata=FrStatDataNew("sweptsine",
        "swept sine calibration:\npacking: freq[i], real[i], imaginary[i]",
        frame->UTimeS,INT_MAX,1,framevec);
#else
    staticdata=FrStatDataNew("sweptsine",
        "swept sine calibration:\npacking: freq[i], real[i], imaginary[i]",
        frame->GTimeS,INT_MAX,1,framevec);
#endif

#if (FRAMELIB_VERSION_INT<=237)
    FrStatDataAdd(&frame->detectRec->sData,staticdata);
#elif (FRAMELIB_VERSION_INT<=330)
    FrStatDataAdd(&frame->detectProc->sData,staticdata);
#else
    FrStatDataAdd(frame->detectProc,staticdata);
#endif

    /* put in lock range (INCLUSIVE low->high) Rolf: if 0=unlock and 1=lock
       then you need LOCKLO=LOCKHI=1
    */

#if (FRAMELIB_VERSION_INT<=237)
    framevecS=FrVectNew(FR_VECT_S,1,2,1.0,"adcCounts");
#else
    framevecS=FrVectNew(FR_VECT_2S,1,2,1.0,"adcCounts");
#endif

    framevecS->dataS[0]=LOCKLO; /* smallest value at which we are still in lock */
    framevecS->dataS[1]=LOCKHI; /* largest value at which we are still in lock */

    /* then link the lockrange data into the history structure */
#if (FRAMELIB_VERSION_INT<330)
    staticdataS=FrStatDataNew("locklo/lockhi",
        "lock range:\npacking: array[0]=locklo array[1]=lockhi",
        frame->UTimeS,INT_MAX,1,framevecS);
#else
    staticdataS=FrStatDataNew("locklo/lockhi",
        "lock range:\npacking: array[0]=locklo array[1]=lockhi",
        frame->GTimeS,INT_MAX,1,framevecS);
#endif

#if (FRAMELIB_VERSION_INT<=237)
    FrStatDataAdd(&frame->detectRec->sData,staticdataS);
#elif (FRAMELIB_VERSION_INT<=330)
    FrStatDataAdd(&frame->detectProc->sData,staticdataS);
#else
    FrStatDataAdd(frame->detectProc,staticdataS);
#endif
}

    /* is the time stamp for this data block consistent with start time+offset? */
#if (FRAMELIB_VERSION_INT<330)
    first_estimate=frame->UTimeS+1.e-9*frame->UTimeN;
#else
```

```
    first_estimate=frame->GTimeS+UTCTOGPS+1.e-9*frame->GTimeN;
#endif

    second_estimate=main_header.epoch_time_sec+1.e-3*main_header.epoch_time_msec;
    diff=first_estimate-second_estimate;
    if (fabs(diff)>0.002)
        fprintf(stderr,"Time stamps have drifted by %f msec!\n",diff);

    /* Increment frame counter (set to 1 for first frame of each run) */
    frame->frame++;

    /* Open Frame file (one file per FRAMES_PER_FILE frames) */
    if ((frame->frame%FRAMES_PER_FILE)==1) {
        /* set file name. Note than month=1 to 12 not 0 to 11! */
/* Obsolete as of Aug 1998 - new file name is GPS time */
#if (FRAMELIB_VERSION_INT<330)
        sprintf(filename,"C1-%02d_%02d_%02d_%02d_%02d",gts.tm_year,gts.tm_mon+1,
            gts.tm_mday,gts.tm_hour,gts.tm_min,gts.tm_sec);
#else
        sprintf(filename,"C1-%d.F",frame->GTimeS);
#endif

        printf("Filename: %s\n",filename);
#if (FRAMELIB_VERSION_INT<330)
        outputfile=FrFileONew(filename, NO, buff, buffSize);
#else
        outputfile=FrFileONew(filename, COMPRESSION, buff, buffSize);
        if (GZIP_LEVEL>0)
            {
                printf("Building frames with compression gzip level = %d\n",GZIP_LEVEL);
            }
        FrFileOSetGzipLevel(outputfile,GZIP_LEVEL);
#endif
    }

    /* un-comment to print a short snippet of each Frame onto the screen */
    /* FrameDump(frame, stdout, 2); */

    /* Write frame to file, */
    FrameWrite(frame, outputfile);

    /* Close file if finished with FRAMES_PER_FILE or no remaining data */
    if ((frame->frame%FRAMES_PER_FILE)==0 || code==-1)
        FrFileOEnd(outputfile);
}

/* Free frame memory and return */
FrameFree(frame);
return(0);
}

/* this routine is called if something is wrong */
void unhappyexit(int i) {
switch (i) {
    case 1:
        fprintf(stderr,
            "Syntax: \ntranslate directory\nwhere channel.* files may be found in directory\n");
        exit(1);
    case 2:
```

```
    fprintf(stderr,
    "The UTC does not appear to lie in the range of any data set!\n");
    exit(1);
default:
    abort();
}
return;
}

/* number of secs after Jan 1 1970 UTC at which Nov 1994 runs began */
static int stimes[]={784880277,784894763,785217574,785233119,785250938,785271063,
                    785288073,785315747,785333880,785351969,785368428,785388248};

/* This routine looks at the epoch time (sec) and returns the run number (1-11) */
int get_run_number(int firsttime) {
    int i;

    for (i=0;i<12;i++)
        if (firsttime<stimes[i]) break;

    return i;
}
```

Author: Bruce Allen, ballen@dirac.phys.uwm.edu

Comments: The technique used to time-stamp this data is an attempt to correct the poor resolution of the original data – please see the remarks in 4.1 for additional detail. Also notice that because the sample rates of the slow/fast channels differ by a ratio of 10, we can not easily reformat the frames with sample sizes of length 2^n . We expect that the FRAME format will continue to evolve, so that this translator (and the FRAME format data) may require periodic updates. Should the year have four digits (eg, 1994) for easier sorting?

16.19 Multi-taper methods for spectral analysis

Since the early 1980's there has been a revolution in the spectral analysis, due largely to a seminal paper by Thomson [39]. There is now a standard textbook on the subject, by Percival and Walden [40], to which we will frequently refer.

Among the most useful of these techniques are the so-called "multitaper" methods. These make use of a special set of windowing functions, called Slepian tapers. For discretely-sampled data sets, these are discrete prolate spheroidal sequences, and are related to prolate spheroidal functions. The GRASP package contains (a modified version of) a public domain package by Lees and Park, which is described in [41]. Further details of this package may be found at <http://love.geology.yale.edu/mtm/>. Note however that we have already included this package in GRASP; there is no need to hunt it down yourself.

For those who are unfamiliar with these techniques, we suggest reading Chapter 7 of [40]. The sets of tapered windows are defined by three parameters. These are, in the notation of Percival and Walden:

N : The length of the discretely-sampled data-set, typically denoted by the integer `npoints` in the GRASP routines.

$NW\Delta t$: The product of total observation time $N\Delta t$ and the resolution bandwidth W . This dimensionless (non-integer) quantity is denoted `nwdt` in the GRASP routines. Note that for a conventional FFT, the frequency resolution would be $W = \Delta f = 1/N\Delta t$. This corresponds to having $NW\Delta t = 1$. The multitaper techniques are typically used with values of W which are several times larger, for example $W = 3\Delta f$, which corresponds to $NW\Delta t = 3$.

K : The number of Slepian tapers (or window functions) used, typically denoted `nwin` in the GRASP routines. Note that it is highly recommended (see page 334 of [40] and the final two figures on page 339) that the number of tapers $K < 2NW\Delta t$.

In addition to providing better spectral estimation tools, the multi-taper methods also provide nice techniques for spectral line parameter estimation and removal. When the sets of harmonic coefficients are generated for different choices of windows, one can perform a regression test to determine if the signal contains a sinusoid of fixed amplitude and phase, consistent across the complete set of tapers. The GRASP package uses this technique (the F-test described on page 499, and the worked-out example starting on page 504 of [40]) to estimate and remove spectral lines from a data-set. This can be used both for diagnostic purposes (i.e., track contamination of the data set by the 5th line harmonic at 300Hz) or to "clean up" the data (i.e., remove the pendulum resonance at 590 Hz).

As an aid in understanding these techniques, we have included with GRASP a section of the data-set from the Willamette River appearing on pg 505 of Percival and Walden [40], and an example program which repeats and reproduces the results in Section 10.13 of that textbook. This demonstrates the use of multi-taper methods in removing "spectral lines" from a data set.

16.20 Function: slepian_tapers()

```
int slepian_tapers(int num_points, int nwin, double *lam, float nwdt, double *tapers, double *tapsum)
```

This function computes and returns properly-normalized Slepian tapers. These tapers are normalized so that

$$\sum_{t=1}^N h_t^2 = N, \quad (16.20.1)$$

which is the *Numerical Recipes* convention for tapers (rather than the Percival and Walden convention for which the rhs is 1). It uses the method described in Percival and Walden [40] pages 386-387, finding the eigenvectors and eigenvalues of a tri-diagonal matrix. The arguments are:

`num_points`: Input. The number of points N in the taper.

`nwin`: Input. The number of tapers computed.

`lam`: Output. Upon return, `lam[0..nwin-1]` contains the eigenvalues λ of the tapers. Note that $0 < \lambda < 1$.

`nwdt`: Input. The (total sample time) \times (frequency resolution bandwidth) product.

`tapers`: Output. Upon return: `tapers[0..num_points-1]` contains the first taper, `tapers[num_points..2*num_points-1]` contains the second taper, and so on.

`tapsum`: Output. On return `tapsum[0]` contains the sum of the `num_points` values of the first taper, `tapsum[1]` contains the sum of the values of the second taper, and so on. Note that because the odd-index Slepian taper functions are odd, `tapsum[1, 3, 5, ...]` would vanish if it were not for round-off and other numerical error.

This function will print a warning message if the condition $K < 2NW\Delta t$ is not satisfied (see Section 16.19).
Author: Adapted from the original code (Lees and Park) by Bruce Allen (ballen@dirac.phys.uwm.edu) and Adrian Ottewill (ottewill@relativity.ucd.ie).

Comments: There are a number of techniques for calculating the Slepian tapers. We have not extensively tested these routines, but they appear to work well. They make use of the standard EISPACK routines, translated from FORTRAN into C using `f2c`.

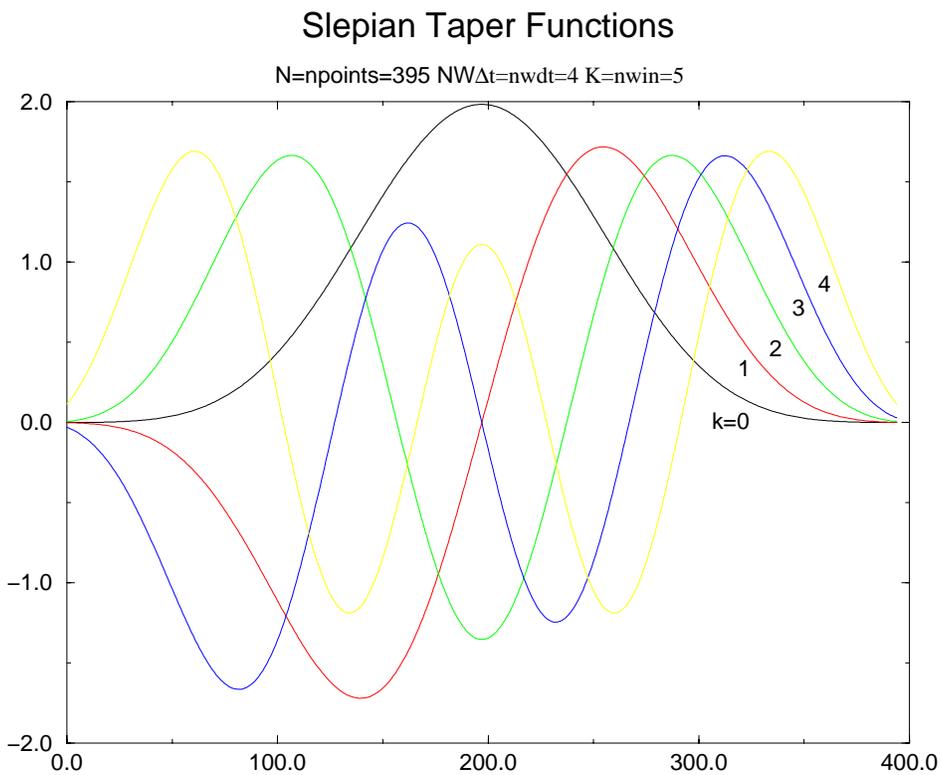


Figure 84: Here are five Slepian tapers computed with `slepian_tapers()`. The parameters are `npoints=395,nwdt=4.0` and `nwin=5`.

16.21 Function: multitaper_spectrum()

multitaper_spectrum(float *data, int npoints, int kind, int nwin, float nwdt, int inorm, float dt, float *ospec, float *dof, float *fvalues, int padded_length, float *cest, int dospec)

This function calculates the multi-taper estimate of the amplitude spectrum $\sqrt{S(f)}$:

$$\sqrt{\hat{S}^{(mt)}(f_j)} := \sqrt{\frac{1}{K} \sum_{k=0}^{K-1} \frac{1}{\lambda_k} \left| \sum_{m=0}^{N-1} x_m h_{k,m} e^{-i2\pi m j / N_p} \right|^2}. \quad (16.21.1)$$

Here x_m is the m th element of the input data array, $h_{k,m}$ is the m th element of the k th order $NW\Delta t$ discrete prolate spheroidal sequence data taper, λ_k is its associated eigenvalue, and f_j is the discrete frequency $f_j := j/N_p\Delta t$, where $j = 0, 1, \dots, N_f - 1 = N_p/2$. The above multi-taper estimate differs slightly from Equation (333) in Percival and Walden: (i) there is the square root; (ii) the data tapers are normalized so that $\sum_{n=0}^{N-1} h_{k,n}^2 = N$; (iii) there is no factor of Δt ; (iv) the individual estimates are weighted by $1/\lambda_k$. (But for $K < 2NW\Delta t$, $1/\lambda_k \approx 1$).

For the sake of efficiency, multitaper_spectrum() computes, and then stores internally, the Slepian taper functions, so that if it is called a second time (and needs the same tapers) they do not need to be re-computed. If called with different parameters it recomputes the Slepian tapers for the new parameters.

If you want to obtain the same normalization as that used in the avg_spec() routine described by equation (16.3.7), the output array ospec[] should first be squared, and then multiplied by a factor of $2\Delta t/N$.

The arguments are:

data: Input. Pointer to the time-domain data array, data[0..npoints-1].

npoints: Input. The total number N of data points in the data array.

kind: Input. If set to 1, compute the normal (i.e., “high resolution”) multi-taper estimate of the amplitude spectrum. If set to 2, compute the “adaptive” multi-taper estimate $\sqrt{\hat{S}^{(amt)}(f_j)}$ of the amplitude spectrum, defined by Percival and Walden Equation (370a).

nwin: Input. The number of tapers to use.

nwdt: Input. The (total sample time) \times (frequency resolution bandwidth) product.

inorm: Input. Determines choice of normalization. Possible values are

- 1: Divide spectrum by N^2 .
- 2: Divide spectrum by Δt^2 .
- 3: Divide spectrum by N .
- 4: Divide spectrum by 1.

dt: Input. Sample interval (only used for normalization).

ospec: Output. The output spectrum, including both DC and Nyquist frequency bins. The array range is ospec[0..padded_length/2]. Warning -this is an *odd* number of entries. The user must provide a pointer to sufficient storage space.

`dof`: Output. The effective number of degrees of freedom of the spectral estimator at a given frequency, defined by Percival and Walden eqn (370b). The number of degrees of freedom is the constant $n_{win}-1$ for `kind=1` above, and only useful in the adaptive case where `kind=2`. The array range is `dof[0..padded_length/2]`. Warning - this is an *odd* number of entries. The user must provide a pointer to sufficient storage space.

`fvalues`: Output. The value of the F-statistic in each frequency bin spectrum, including both DC and Nyquist. This is defined by Percival and Walden equation (499c), and roughly speaking is the ratio of the energy explained by the hypothesis that one has a fixed-amplitude spectral line at that frequency to the energy not explained by this hypothesis. The array range is `fvalues[0..padded_length/2]`. Warning -this is an *odd* number of entries. The user must provide a pointer to sufficient storage space.

`padded_length`: Input. The padded length N_p is an integer power of 2, greater than (or equal to) `npoints`. The (tapered) data is zero-padded out to this length. You generally want N_p to be around four to eight times larger than the length of your data set, to get decent frequency resolution. The number of frequency bins (including DC and Nyquist) in the output spectrum is $N_f = 1 + N_p/2$.

`cest`: Output. The estimated Fourier coefficients of any spectral lines in the data. The real and imaginary parts at DC are contained in `cest[0],cest[1]`. The next higher frequency bin has its real/imaginary parts contained in `cest[2],cest[3]`, and so on. This pattern continues up to and including the Nyquist frequency. The length of the array is `cest[0..padded_length+1]`. The normalization/sign conventions are identical to Percival and Walden eqns (499a) and the example on line 20 of page 513, except that the sign of the imaginary part is reversed, because the Percival and Walden FFT conventions eqn (65ab) are opposite to Numerical Recipes. The user must provide a pointer to sufficient storage space.

`dospec`: Input. If set non-zero, then the power spectrum (pointed to by `ospec`) is calculated. If set to zero, then to save time in situations where all that is needed is `cest`, the power spectrum `ospec` is *not* calculated.

Author: Adapted from the original code (Lees and Park) by Bruce Allen (ballen@dirac.phys.uwm.edu) and Adrian Ottewill (ottewill@relativity.ucd.ie).

Comments: None.

16.22 Function: multitaper_cross_spectrum()

```
void multitaper_cross_spectrum(float *o1, float *o2, int npoints, int padded_length,
float delta_t, int nwin, float nwdt, double *ReImSpec12)
```

This function calculates the high resolution multitaper estimate of the (complex-valued) cross-correlation spectrum $\tilde{o}_1^*(f)\tilde{o}_2(f)$ of the two input time-series $o_1(t)$ and $o_2(t)$.

The arguments are:

o1: Input. `o1[0..npoints-1]` is an array of floating point variables containing the values of the input time-series $o_1(t)$. `o1[i]` contains the value of $o_1(t)$ evaluated at the discrete time $t_i := i\Delta t$, where $i = 0, 1, \dots, N - 1$.

o2: Input. `o2[0..npoints-1]` is an array of floating point variables containing the values of the input time-series $o_2(t)$, in exactly the same format as the previous argument.

npoints: Input. The total number N of data points contained in the two input time-series.

padded_length: Input. The padded length N_p is an integer power of 2, greater than (or equal to) the total number of data points. The tapered data sets are zero-padded out to this length. The total number of frequency bins (including DC and Nyquist) in the output cross-correlation spectrum is $N_f = 1 + N_p/2$.

delta_t: Input. The sampling period Δt (in sec).

nwin: Input. The total number K of data tapers used when forming the multitaper spectral estimate.

nwdt: Input. The (total sample time) \times (frequency resolution bandwidth) product $N\Delta t \cdot W$.

ReImSpec12: Output. `ReImSpec12[0..padded_length+1]` is an array of double precision variables containing the values of the high resolution multitaper spectral estimate of the (complex-valued) cross-correlation spectrum $\tilde{o}_1^*(f)\tilde{o}_2(f)$. `ReImSpec12[2*j]` and `ReImSpec12[2*j+1]` contain, respectively, the values of the real and imaginary parts of

$$\Delta t^2 \frac{1}{K} \sum_{k=0}^{K-1} \frac{1}{\lambda_k} \left(\sum_{m=0}^{N-1} o_1(t_m) h_{k,m} e^{-i2\pi m j / N_p} \right) \left(\sum_{n=0}^{N-1} o_2(t_n) h_{k,n} e^{+i2\pi n j / N_p} \right) \quad (16.22.1)$$

where $h_{k,n}$ is the n th element of the k th order $NW\Delta t$ discrete prolate spheroidal sequence data taper, and λ_k is its associated eigenvalue. (Here $j = 0, 1, \dots, N_f - 1 = N_p/2$.) The data tapers are normalized so that $\sum_{n=0}^{N-1} h_{k,n}^2 = N$.

If you want to obtain the same normalization as that used in the `avg_spec()` routine described by equation (16.3.7) for the case where $o_1(t) = o_2(t)$ then the output array `ReImSpec12` should be multiplied by a factor of $2/N\Delta t$.

Author:

Adapted from the original code (Lees and Park) by Bruce Allen (ballen@dirac.phys.uwm.edu), Adrian Ottewill (ottewill@relativity.ucd.ie), and Joseph Romano (romano@csd.uwm.edu).

Comments: None.

16.23 Structure: struct removed_lines

This is a structure used to keep track of spectral lines as they are removed. Its primary use is in the function `remove_spectral_lines()`. The structure contains the following:

```
struct removed_lines{
    int index;
    float fvalue;
    float re;
    float im;
};
```

The different quantities are:

index: The subscript (frequency bin) occupied by the spectral line in an array of length N_f (defined in the previous section). Note that in typical use index runs over a range of $2^n + 1$ possible values, including DC and Nyquist.

fvalue The value of the F-statistic, defined by Percival and Walden eqn. (499c).

re: The real part of the line's complex amplitude. The normalization/sign conventions are identical to Percival and Walden eqns (499a) and the example on line 20 of page 513.

im: The imaginary part of the line's complex amplitude. The normalization conventions are identical to Percival and Walden eqns (499a) and the example on line 20 of page 513, but the sign is reversed, because the Percival and Walden FFT conventions eqn (65ab) are opposite to Numerical Recipes.

16.24 Function: fvalue_cmp()

```
int fvalue_cmp(const void *f1, const void *f2)
```

This is a function which may be used to compare the \hat{f} values of two different objects of type `struct removed_lines`. It is used for example as an argument to the standard-C library routine `qsort` for sorting lists of removed lines into decreasing order of `fvalue`.

This function is supplied with pointers to two structures. It returns -1 if the first structure has the larger `fvalue`, +1 if the first structure has the smaller `fvalue`, and 0 if the `fvalues` are equal.

The arguments are:

f1: Input. Pointer to the first structure of type `struct removed_lines` (cast to `void *` so that your compiler does not complain).

f2: Input. Pointer to the second structure of type `struct removed_lines` (cast to `void *` so that your compiler does not complain).

As an example, if `line_list[0..n-1]` is an array of `struct removed_lines`, then the function call:

```
qsort(line_list,n,sizeof(struct significant_values),fvalue_cmp)
```

will sort that array into decreasing `fvalue` order. (Note: you may have to cast the arguments to prevent your compiler from complaining.)

Author: Bruce Allen (ballen@dirac.phys.uwm.edu) and Adrian Ottewill (ottewill@relativity.ucd.ie).

Comments: None.

16.25 Function: index_cmp()

```
int index_cmp(const void *f1, const void *f2)
```

This is a function which may be used to compare the indexes of two different objects of type `struct removed_lines`. It is used for example as an argument to the standard-C library routine `qsort` for sorting lists of removed lines into increasing order in frequency.

This function is supplied with pointers to two structures. It returns -1 if the first structure has the smaller index, +1 if the first structure has the larger index, and 0 if the indexes are equal.

The arguments are:

f1: Input. Pointer to the first structure of type `struct removed_lines` (cast to `void *` so that your compiler does not complain).

f2: Input. Pointer to the second structure of type `struct removed_lines` (cast to `void *` so that your compiler does not complain).

As an example, if `line_list[0..n-1]` is an array of `struct removed_lines`, then the function call:

```
qsort(line_list,n,sizeof(struct significant_values),index_cmp)
```

will sort that array into increasing index (frequency!) order. (Note: you may have to cast the arguments to prevent your compiler from complaining.)

Author: Bruce Allen (ballen@dirac.phys.uwm.edu) and Adrian Ottewill (ottewill@relativity.ucd.ie).

Comments: None.

16.26 Function: remove_spectral_lines()

```
void remove_spectral_lines(float *data, int npoints, int padded_length, float
nwdt, int nwin, int max_lines, int maxpass, int *num_removed, struct removed_lines
*line_list,
float *mtap_spec_init, float *mtap_spec_final, int dospecs, int fimin, int
fimax)
```

This routine automatically identifies and removes “spectral lines” from a time series. The procedure followed is described in Percival and Walden Chapter 10. A worked example may be found in Section 10.13 of that book, and the next subsection of the GRASP manual includes two example programs which use `remove_spectral_lines()`. Upon return, `remove_spectral_lines()` provides both an “initial” multi-taper amplitude spectrum, of the original data, and a “final” multi-taper amplitude spectrum, after line removal. Upon return, the data set has the spectral lines subtracted. This routine also returns a list of the lines removed. For each line it provides the frequency bin (for the padded data set) in which the line falls, the value of the F-test for that line, and the complex coefficient \hat{C}_i defined by Percival and Walden eqn (499a) which defines the line.

The arguments are:

`data`: Input. A pointer to the time-series array `data[0..npoints-1]`.

`npoints`: Input. The number of points in the previous array.

`padded_length`: Input. The number of points N_p of zero-padded data that will be analyzed. Note that N_p must be an integer power of two greater than or equal to `npoints`. We recommend that you use at least a factor of four greater, to obtain sufficient frequency resolution to accurately identify/remove spectral lines.

`nwdt`: Input. The (total sample time) \times (frequency resolution bandwidth) product.

`nwin`: Input. Number of Slepian tapers. See previous sections.

`max_lines`: Input. The maximum number of spectral lines that you want removed. The array `line_list[0..max_lines-1]` must have at least this length.

`maxpass`: Input. The maximum number of iterations or passes through the line-removal loop described below. Set to a large number to make as many passes as needed to remove all the spectral lines.

`num_removed`: Output. The actual number of spectral lines subtracted from the data.

`line_list`: Output. A list of structures `line_list[0..num_removed-1]` containing the frequency bin, real and imaginary parts of the removed line, and the F-test significance value associated with the *first* removal of the line. Upon return from this function, the elements of `line_list[]` are sorted into increasing frequency-bin order.

`mtap_spec_init`: Output. The multi-taper estimate of the amplitude spectrum of the *initial* `data[]` array, including both DC and Nyquist frequency bins. The array range is `mtap_spec_init[0..padded_length/2-1]`. Warning -this is an *odd* number of entries. The user must provide a pointer to sufficient storage space.

`mtap_spec_final`: Output. The multi-taper estimate of the amplitude spectrum of the *final* `data[]` array, with the spectral lines subtracted, including both DC and Nyquist frequency bins. The array range is `mtap_spec_final[0..padded_length/2-1]`. Warning -this is an *odd* number of entries. The user must provide a pointer to sufficient storage space.

`dospecs`: Input. If set non-zero, then the initial/final amplitude spectra (pointed to by `mtap_spec_init` and `mtap_spec_final`) are calculated. If set to zero, then to save time in situations where all that is needed a list of spectral lines and their amplitudes and phases, then neither of the amplitude spectra are calculated.

`fimin`: Input. In situations where all that is needed is a list of spectral lines and their amplitudes, and it is desired to limit the search to a restricted range of frequencies, then `fimin` defines the lower bound of the range of (padded) frequency bins which are searched for spectral lines. The range of `fimin` is $0.\text{padded_length}/2$. Also, $fimin < fimax$.

`fimax`: Input. In situations where all that is needed is a list of spectral lines and their amplitudes, and it is desired to limit the search to a restricted range of frequencies, then `fimax` defines the upper bound of the range of (padded) frequency bins which are searched for spectral lines. The range of `fimax` is $0.\text{padded_length}/2$. Also, $fimin < fimax$.

The algorithm used by `remove_spectral_lines()` is an automated version of the procedure illustrated in Percival and Walden Section 10.13. The steps followed are:

1. The mean value is subtracted from the data-set, and it is zero padded to the specified length.
2. The set of Fourier coefficients for the tapered data sets are determined.
3. From these coefficients the F-statistic is determined for each frequency bin (Percival and Walden eqn (499c)). If the confidence level (that the frequency bin contains a spectral line) exceeds $1 - 1/\text{npoints}$ (Percival and Walden pg 513), an estimator of the spectral line coefficients is constructed, and the line is placed onto a working list. If no frequency bins exceed this level of confidence, we are finished.
4. The working list is now sorted into order of decreasing F-values.
5. To ensure that we do not remove the same line twice, the spectral line associated with each spectral line on the working list is subtracted from the data-set, provided that it does not lie within a frequency width of $\pm W$ of a stronger (larger F-value) line.
6. We return to step 1 above, iterating this procedure, provided that the number of times that we have passed by step 1 is less than or equal to `maxpass`.

Author: Bruce Allen (ballen@dirac.phys.uwm.edu) and Adrian Ottewill (ottewill@relativity.ucd.ie).

Comments: If `max_lines` is not large enough, then the `line_list[]` array may not contain all of the possible spectral lines, which exceed the confidence level above. This may even be the case if `num_removed` is less than `max_lines`. We suggest that you make `max_lines` somewhat larger than `num_removed`. One ought to be able to improve on this routine, by using the array of F-values generated internally and interpolating to find the frequency of the lines more precisely. One might also be able to fit to a model of two closely separated lines to better remove certain “split” features, or to fit to an exponentially-decaying model to remove other broadened features.

16.27 Example: river

This is an example program which uses the function `remove_spectral_lines()` to repeat the analysis of data from the Willamette River given by Percival and Walden in section 10.13 of their textbook.

It displays graphs of the river flow data (which is distributed with GRASP) and spectrum before and after automatic removal of the two significant spectral lines (whose frequencies are 1/year and 2/year). These graphs are also shown here. Before running this program, be sure to set the environment variable giving the path to the river data, for example:

```
setenv GRASP_PARAMETERS /usr/local/GRASP/parameters
```

The text output of the program is as follows:

```
Total number of lines removed: 2
Removed line of amplitude -0.291175 + i 0.312209 at freq 1.005848 cycles/year
(F-test value 48.455242)
Removed line of amplitude 0.023220 + i 0.098357 at freq 2.000000 cycles/year
(F-test value 15.224311)
```

```
/* GRASP: Copyright 1997,1998 Bruce Allen */
#include "grasp.h"
#include <unistd.h> /* need the header for the sleep() function */

int main() {
    int i,num_points,num_win,num_freq,padded_length,max_lines,num_removed;
    float nwdt,*data,*mtap_spec_init,*mtap_spec_final,freq,fnyquist;
    struct removed_lines *line_list;
    FILE *fpriver;

    /* data length, padded length, num frequencies including DC, Nyquist */
    num_points=395;
    padded_length=1024;
    num_freq=1+padded_length/2;

    /* number of taper windows to use, and time-freq bandwidth */
    num_win=5;
    nwdt=4.0;

    /* maximum number of lines to remove */
    max_lines=8;

    /* allocate arrays */
    data= (float *)malloc(sizeof(float)*num_points);
    mtap_spec_init=(float *)malloc(sizeof(float)*num_freq);
    mtap_spec_final=(float *)malloc(sizeof(float)*num_freq);
    line_list=(struct removed_lines *)malloc(sizeof(struct removed_lines)*max_lines);

    /* Read Willamette River data from Percival & Walden example, pg 505 */
    fpriver=grasp_open("GRASP_PARAMETERS","willamette_river.dat","r");
    for (i=0;i<395;i++) fscanf(fpriver,"%f",data+i);
    fclose(fpriver);

    /* Since the data is sampled once per month, fnyquist = 6 cyles/year */
    fnyquist=0.5*12;

    /* pop up a graph of the original data */
    graph(data,num_points,1); sleep(5);
```

```
/* now remove the spectral lines from the data set */
remove_spectral_lines(data,num_points,padded_length,nwdt,num_win,
    max_lines,500,&num_removed,line_list,mtap_spec_init,mtap_spec_final,1,0,num_freq);

/* pop up a graph of the original multitapered spectrum */
graph(mtap_spec_init,num_freq,1); sleep(5);

/* pop up a graph of the line-removed data and multitapered spectrum */
graph(data,num_points,1); sleep(5);
graph(mtap_spec_final,num_freq,1); sleep(5);

/* print out a list of lines removed */
printf("Total number of lines removed: %d\n",num_removed);
for (i=0;i<num_removed;i++) {
    freq=line_list[i].index*fnyquist/num_freq;
    printf("Removed line of amplitude %f + i %f at freq %f cycles/year\t",
        line_list[i].re,line_list[i].im,freq);
    printf("(F-test value %f)\n",line_list[i].fvalue);
}
return 0;
}
```

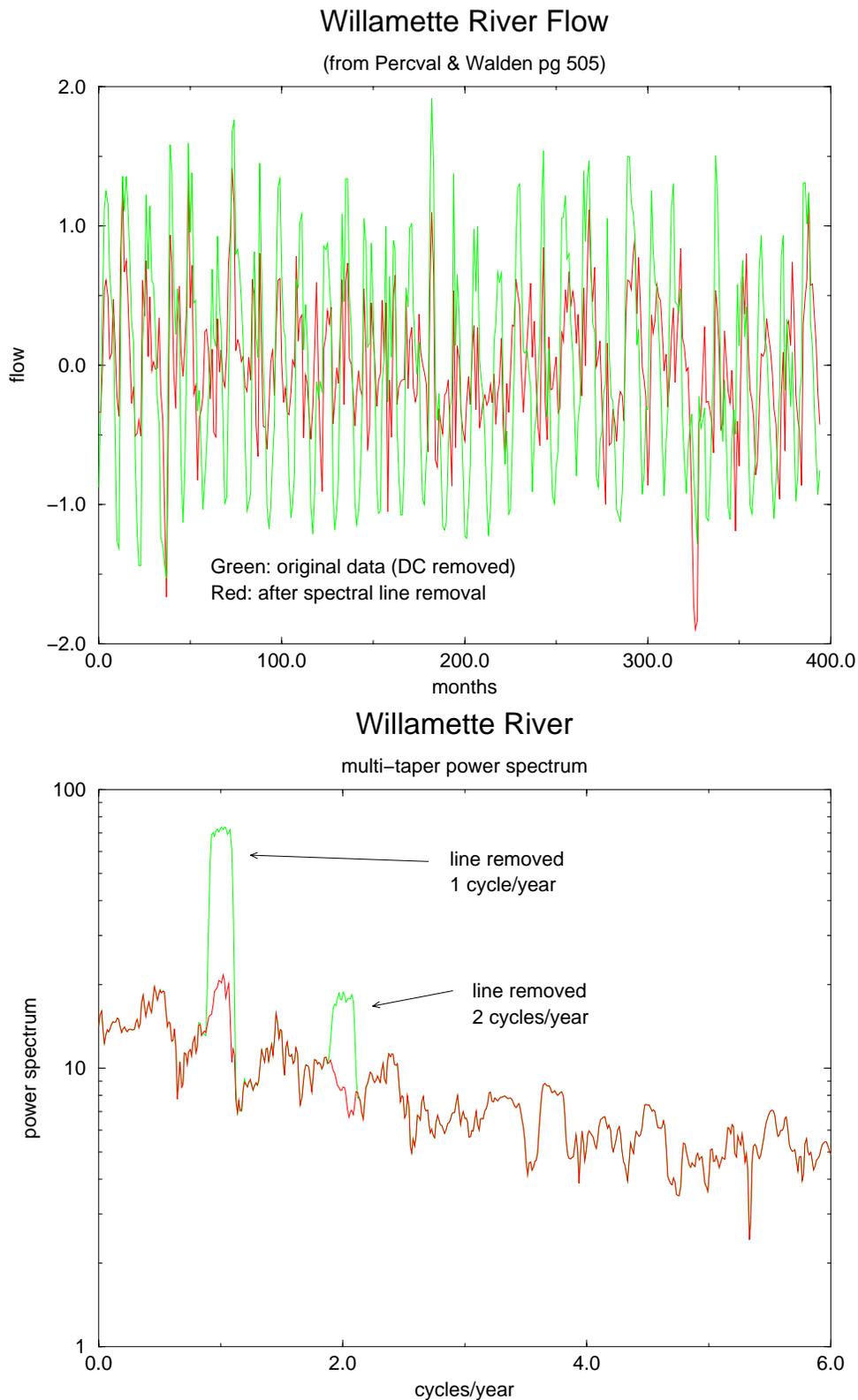


Figure 85: Output of the example program `river`, making use of `remove_spectral_lines()` to automatically find and remove two “spectral line” features from a data set. This is the same example treated by Percival and Walden in Section 10.13 of their textbook.

16.28 Example: ifo_clean

This example program uses `remove_spectral_lines()` to automatically identify and remove “spectral lines” from the output of the 40-meter IFO. To run this program, be sure to set the data path environment variable, for example:

```
setenv GRASP_DATAPATH /usr/local/GRASP/data/19nov94.3
```

The program outputs graphs in a two files called `ifo_clean_data.out` and `ifo_clean_spec.out`, containing the before/after time series and spectra. These may be viewed with `xmgr` by typing:

```
xmgr -nxy ifo_clean_data.out
```

and

```
xmgr -nxy ifo_clean_spec.out
```

to start up the `xmgr` graphing program.

The output of this program is a list of lines removed:

```
Total number of lines removed: 39
```

```
Removed line frequency 30.717 Hz amplitude 0.78 phase 15.54 (F-test 68.6)
Removed line frequency 79.203 Hz amplitude 0.55 phase -157.41 (F-test 52.5)
Removed line frequency 80.257 Hz amplitude 0.12 phase -101.84 (F-test 39.3)
Removed line frequency 109.318 Hz amplitude 4.52 phase 10.21 (F-test 75.5)
Removed line frequency 120.009 Hz amplitude 0.46 phase 5.01 (F-test 537.9)
Removed line frequency 139.584 Hz amplitude 0.29 phase -163.57 (F-test 304.5)
Removed line frequency 179.938 Hz amplitude 21.91 phase -43.22 (F-test 3635.0)
Removed line frequency 239.867 Hz amplitude 0.45 phase 130.25 (F-test 42.2)
Removed line frequency 245.438 Hz amplitude 0.21 phase -116.94 (F-test 51.9)
Removed line frequency 279.167 Hz amplitude 0.31 phase 0.52 (F-test 47.2)
Removed line frequency 299.947 Hz amplitude 15.37 phase -135.82 (F-test 9712.5)
Removed line frequency 359.876 Hz amplitude 1.17 phase 61.64 (F-test 134.8)
Removed line frequency 419.955 Hz amplitude 4.48 phase -39.58 (F-test 356.1)
Removed line frequency 488.768 Hz amplitude 0.19 phase 165.56 (F-test 50.5)
Removed line frequency 500.212 Hz amplitude 0.64 phase 129.38 (F-test 34.5)
Removed line frequency 539.964 Hz amplitude 5.09 phase 119.38 (F-test 425.2)
Removed line frequency 571.585 Hz amplitude 4.01 phase 120.03 (F-test 50.6)
Removed line frequency 578.662 Hz amplitude 34.97 phase -149.12 (F-test 429.8)
Removed line frequency 582.426 Hz amplitude 107.36 phase 15.64 (F-test 1129.7)
Removed line frequency 597.936 Hz amplitude 58.72 phase 63.27 (F-test 558.6)
Removed line frequency 605.314 Hz amplitude 17.21 phase -140.57 (F-test 489.7)
Removed line frequency 659.822 Hz amplitude 2.20 phase -152.53 (F-test 121.0)
Removed line frequency 779.831 Hz amplitude 3.95 phase -39.18 (F-test 502.4)
Removed line frequency 839.760 Hz amplitude 2.75 phase -172.15 (F-test 468.2)
Removed line frequency 899.840 Hz amplitude 3.40 phase 113.05 (F-test 529.6)
Removed line frequency 959.919 Hz amplitude 0.80 phase 178.70 (F-test 43.2)
Removed line frequency 999.822 Hz amplitude 1.01 phase 67.74 (F-test 114.8)
Removed line frequency 1019.698 Hz amplitude 1.46 phase -156.72 (F-test 146.6)
Removed line frequency 1079.777 Hz amplitude 3.00 phase 51.82 (F-test 128.9)
Removed line frequency 1157.023 Hz amplitude 2.99 phase -76.14 (F-test 129.4)
Removed line frequency 1210.778 Hz amplitude 2.12 phase 128.39 (F-test 69.5)
Removed line frequency 1319.644 Hz amplitude 3.02 phase -105.29 (F-test 146.2)
Removed line frequency 1499.582 Hz amplitude 1.31 phase 141.94 (F-test 50.5)
```

```
Removed line frequency 1559.662 Hz amplitude 2.79 phase 107.12 (F-test 60.0)
Removed line frequency 1746.978 Hz amplitude 1.81 phase 50.38 (F-test 112.0)
Removed line frequency 2039.697 Hz amplitude 1.65 phase 165.82 (F-test 62.3)
Removed line frequency 2279.413 Hz amplitude 2.12 phase -25.06 (F-test 163.0)
Removed line frequency 3509.465 Hz amplitude 0.11 phase 43.89 (F-test 60.1)
Removed line frequency 4609.720 Hz amplitude 0.03 phase 24.61 (F-test 39.4)
```

Virtually all of these lines can be identified as either 60 Hz line harmonics, or as specific suspension and pendulum modes. The removal of these lines makes a dramatic difference to the appearance (and sound of) the signal, as shown in Figure 86. Note that the amplitudes of the lines above are properly normalized (in ADC units). For example the 180 Hz line harmonic is well described by $A(t) = 21.91 \sin(360\pi t/\text{sec})$. By far the largest amplitude lines are the three violin modes at 578.662, 582.426, and 597.936 Hz, and the 180 and 300 Hz line harmonics. Most of the structure visible in Figure 86 is the result of these five harmonics.

```
/* GRASP: Copyright 1997,1998 Bruce Allen */
#include "grasp.h"

int main() {
    short *datas;
    int i,num_points,num_win,num_freq,padded_length,max_lines,num_removed,remain;
    float nwdt,*data,*mtap_spec_init,*mtap_spec_final,freq,tstart,srate,*initial_data,amp,phi;
    struct removed_lines *line_list;
    FILE *fpifo,*fplock,*fpout1,*fpout2;

    /* open the IFO output file and lock file */
    fpifo =grasp_open("GRASP_DATAPATH","channel.0","r");
    fplock=grasp_open("GRASP_DATAPATH","channel.10","r");

    /* data length, padded length, num frequencies including DC, Nyquist */
    num_points=8192;
    padded_length=65536;
    num_freq=1+padded_length/2;

    /* number of taper windows to use, and time-freq bandwidth */
    num_win=5;
    nwdt=3.0;

    /* maximum number of lines to remove */
    max_lines=100;

    /* allocate arrays */
    datas=(short *)malloc(sizeof(short)*num_points);
    data=(float *)malloc(sizeof(float)*num_points);
    mtap_spec_init=(float *)malloc(sizeof(float)*num_freq);
    mtap_spec_final=(float *)malloc(sizeof(float)*num_freq);
    line_list=(struct removed_lines *)malloc(sizeof(struct removed_lines)*max_lines);
    initial_data=(float *)malloc(sizeof(float)*num_points);

    /* get a section of data... */
    get_data(fpifo,fplock,&tstart,num_points,datas,&remain,&srate,0);

    /* copy short data to float data,and save initial data set */
    for (i=0;i<num_points;i++) initial_data[i]=data[i]=datas[i];

    /* remove the spectral lines from the data set */
    remove_spectral_lines(data,num_points,padded_length,nwdt,num_win,
```

```
max_lines,500,&num_removed,line_list,mtap_spec_init,mtap_spec_final,1,0,num_freq);

/* print out a list of lines removed */
printf("Total number of lines removed: %d\n",num_removed);
for (i=0;i<num_removed;i++) {
    freq=0.5*line_list[i].index *srate/num_freq;
    amp=2.0*sqrt(line_list[i].re*line_list[i].re+line_list[i].im*line_list[i].im);
    phi=180*atan2(line_list[i].im,line_list[i].re)/M_PI;
    printf("Removed line frequency %.3f Hz amplitude %.2f phase %.2f (F-test %.1f)\n",
          freq,amp,phi,line_list[i].fvalue);
}

/* now output a file containing the initial and final data... */
fpout1=fopen("ifo_clean_data.out","w");
fprintf(fpout1,"# Three columns are:\n# Time (sec) Initial data Final Data\n");
for (i=0;i<num_points;i++)
    fprintf(fpout1,"%f\t%f\t%f\n",i/srate,initial_data[i],data[i]);
fclose(fpout1);

/* ... and the initial and final spectra, for graphing by xmgr */
fpout2=fopen("ifo_clean_spec.out","w");
fprintf(fpout2,"# Three columns are:\n# Freq (Hz) Initial spectrum Final spectrum\n");
for (i=0;i<num_freq;i++)
    fprintf(fpout2,"%f\t%f\t%f\n",0.5*i*srate/num_freq,mtap_spec_init[i],mtap_spec_final[i]);
fclose(fpout2);

return 0;
}
```

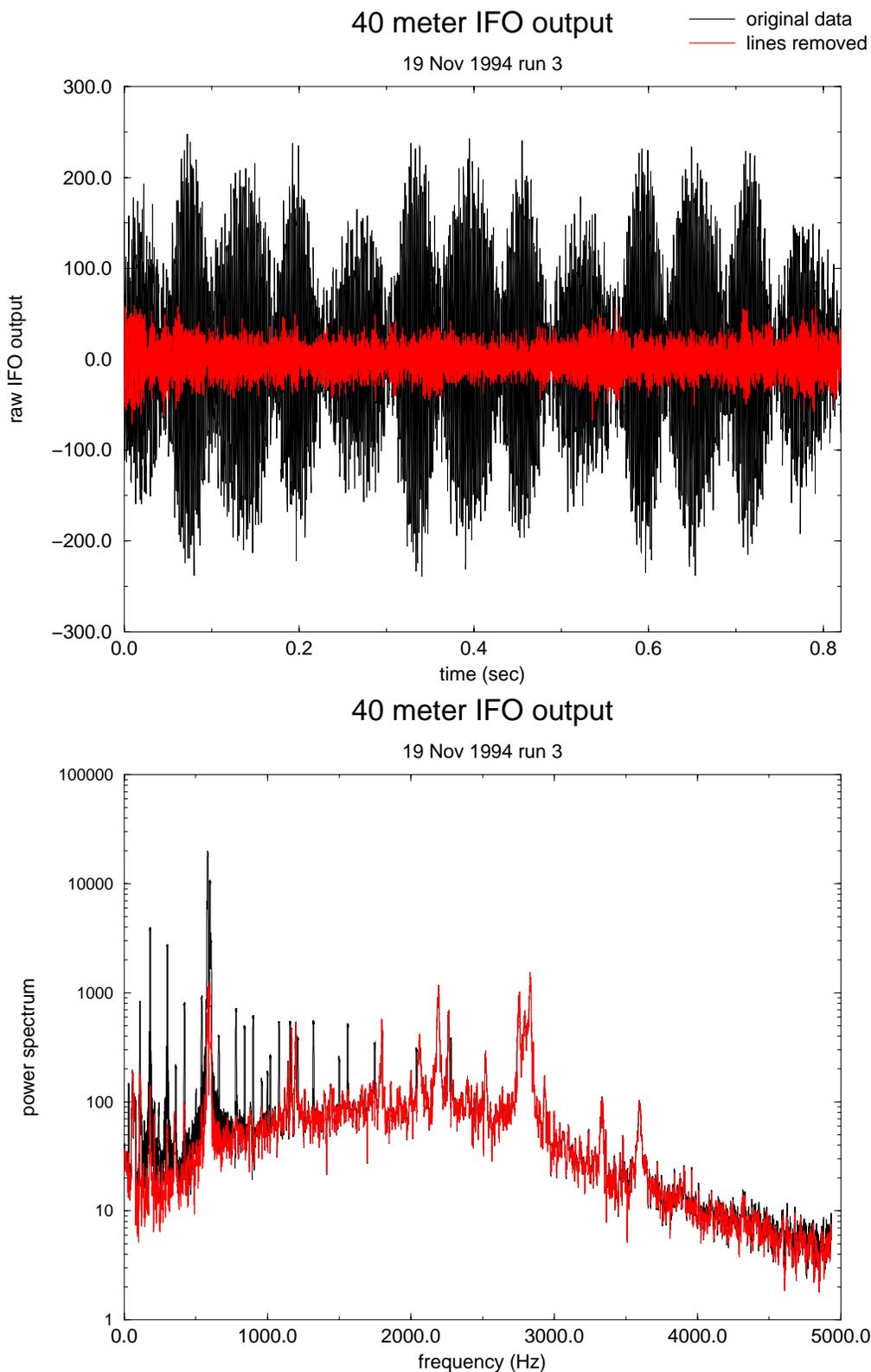


Figure 86: Output of the example program `ifo_clean`, making use of `remove_spectral_lines()` to automatically identify and remove “spectral line” features from the (whitened) output of the Caltech 40-meter interferometer. The black curve and the red curve show the before/after time series and spectra. We have deliberately chosen a stretch of data immediately after the IFO locks, so that the suspension and pendulum modes are excited.

16.29 Example: tracker

This program produces an animated display which tracks the amplitude and phase of selected line features in the spectrum. It has a number of user-settable options which determine how the line is tracked. To run this program, type

```
tracker | xmgr -pipe
```

and an animated display will start up. In normal use, the parameters should be set as follows:

`num_points`: a power of two. A single phase/amplitude point is printed for each set of `num_points` samples.

`padding_factor`: a power of two. This determines the amount of padding done on the data set, and thus the ultimate frequency resolution of the line discrimination.

`fpreset`: your best guess for the frequency that you want to track. If the actual frequency of the spectral line differs from this value, then the phase will slowly drift as a linear function of time. (The `tracker` program does a robust best linear fit to this slope, and uses it to report a best frequency estimate.)

`estimate`: if set to zero, then the phase of the line found is always compared with the frequency preset above. If set non-zero, then `tracker` will make a “best estimate” of the true frequency, and compare the phase of the line found with the phase appropriate to that sinusoid.

`nbins`: the number of (padded) frequency bins adjacent to the one of interest in which the line will be searched for. The frequency range covered is thus given by

$$\Delta f = \pm \frac{\text{nbins}}{\Delta t(\text{num_points} \times \text{padding_factor} + 2)} \quad (16.29.1)$$

`num_display`: the number of points displayed by `tracker`. The total amount of time covered by the output is `num_display` \times `num_points` \times Δt where Δt is the sample interval.

`num_win`, `nwdt`: these parameters are described in the section on multi-taper methods.

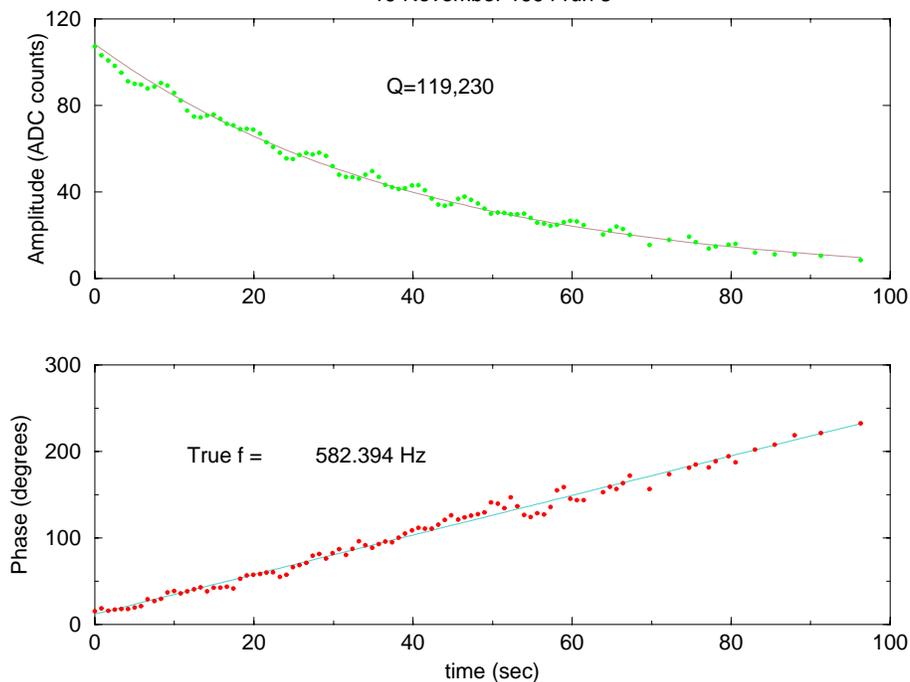
`maxpass`: the number of passes to make within `remove_spectral_lines()`. This number should be set as small as possible, provided that you still “catch” the line of interest.

Authors: Bruce Allen (ballen@dirac.phys.uwm.edu) and Adrian Ottewill (ottewill@relativity.ucd.ie).

Comments: None.

582.4 Hz violin mode

19 November 1994 run 3



Line Tracker

best estimate f=179.973053 Hz

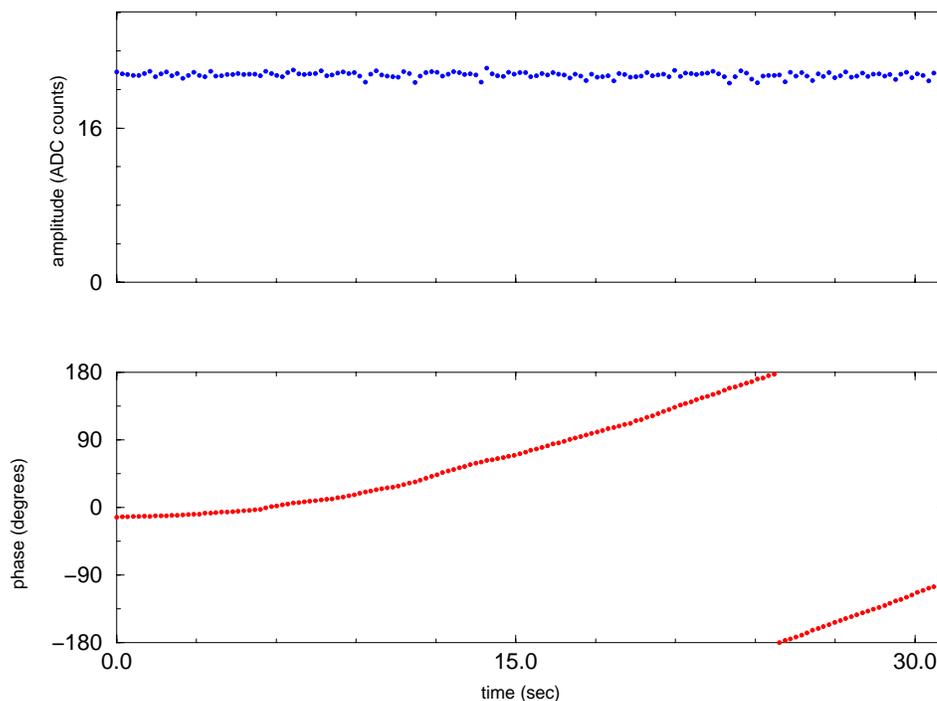


Figure 87: Output of the example program `tracker`, making use of `remove_spectral_lines()` to track the amplitude and phase of a selected “spectral line” features from the (whitened) output of the Caltech 40-meter interferometer. The upper two graphs show the approximately exponential decay of the 582.396 Hz violin mode; the lower two graphs show the amplitude and phase of the third harmonic of the 60Hz line noise (note the remarkable amplitude stability).

16.30 Example: trackerF

This example program is identical to the `tracker` program just described, which tracks spectral lines, but with one crucial difference: it reads its data from FRAME files rather than from the old format data stream.

To run this program, type

```
setenv GRASP_FRAMEPATH /usr/local/GRASP/18nov94.1frame
```

```
trackerF | xmgr -pipe
```

and an animated display will start up.

To run this example in real-time on data coming out at the 40-meter lab, type

```
setenv GRASP_REALTIME
```

```
trackerF | xmgr -pipe
```

and an animated display will start up.

```
/* GRASP: Copyright 1997,1998 Bruce Allen */
#include "grasp.h"

/* macros to define the standard mathematical forms of mod */
#define MOD(X) ((X)>=0?((X)%num_display):(num_display-1+((X+1)%num_display)))
#define FMOD2PI(X) ((X)>=0?(fmod((X),2.0*M_PI):(2.0*M_PI+fmod((X),2.0*M_PI)))

/* numerical recipes routine for robust linear fit */
void medfit(float x[],float y[],int npoints,float *a ,float *b,float *dev);
void graphout(float,float,float,float);

int main() {
    short *datas;
    int npass=1,num_points,num_win,num_freq,padded_length,max_lines,num_removed,code;
    int i,top=0,estimate,nbins,padding_factor,num_display,nprint,index,new=0,maxpass=1,minbin,maxbin;
    float nwdt,*data,*mtap_spec_init,*mtap_spec_final,srate,creal,cimag;
    float *phase,phase1=0.0,amp1,phase2,*times,*linfitx,*linfity,offset,binpreset;
    float displaytime,t1,*amp,dbin,ffit,intercept,slope,deviation,maxamp,displayamp=1.0;
    double time,fpreset;
    struct removed_lines *line_list;
    struct fgetinput fgetinput;
    struct fgetoutput fgetoutput;

    /* ----- USER DEFINABLE ----- */
    /* data length, padded length (powers of 2!) */
    num_points=2048;
    padding_factor=8;

    /* your best guess for the line frequency you want to track */
    fpreset=582.395;

    /* set non-zero if you want us to estimate the best-fit frequency */
    estimate=0;

    /* number of (padded) frequency bins (either side) to search near fpreset */
    nbins=5;

    /* the number of phase/amplitudes to display */
    num_display=150;

    /* number of taper windows to use, and time-freq bandwidth */
    num_win=5;
```

```
nwdt=3.0;

/* the number of passes to make within the line removal algorithm */
maxpass=1;

/* num_points=2048; padding_factor=8;fpreset=582.395; */
num_points=4096; padding_factor=4;fpreset=582.395;
num_points=4096; padding_factor=4;fpreset=180.0;
/* ----- END OF USER DEFINABLE -----*/

/* number of channels */
fgetinput.nchan=1;
fgetinput.inlock=0;
fgetinput.npoint=num_points;

/* source of files */
fgetinput.files=framefiles;

fgetinput.chnames=(char **)malloc(fgetinput.nchan*sizeof(char *));
fgetinput.locations=(short **)malloc(fgetinput.nchan*sizeof(short *));
fgetoutput.npoint=(int *)malloc(fgetinput.nchan*sizeof(int));

/* channel name */
fgetinput.chnames[0]="IFO_DMRO";

/* number of points to get */
fgetinput.seek=0;
fgetinput.calibrate=0;

padded_length=padding_factor*num_points;

/* num frequencies including DC, Nyquist */
num_freq=1+padded_length/2;

/* max number of lines to report on */
max_lines=64;

/* allocate storage */
datas=(short *)malloc(sizeof(short)*num_points);
data=(float *)malloc(sizeof(float)*num_points);
mtap_spec_init=(float *)malloc(sizeof(float)*num_freq);
mtap_spec_final=(float *)malloc(sizeof(float)*num_freq);
line_list=(struct removed_lines *)malloc(sizeof(struct removed_lines)*max_lines);
amp=(float *)malloc(sizeof(float)*num_display);
phase=(float *)malloc(sizeof(float)*num_display);
times=(float *)malloc(sizeof(float)*num_display);
linfitx=(float *)malloc(sizeof(float)*num_display);
linfity=(float *)malloc(sizeof(float)*num_display);

fgetinput.locations[0]=datas;

while (npass>0) {
    /* get a section of data... */
    code=fget_ch(&fgetoutput,&fgetinput);
    time=fgetoutput.dt;

    if (code==0) return 0;
    new+=code;
}
```

```
srate=fgetoutput.srate;
if (new==1) {
    fprintf(stderr, "\aTracker: New Locked Segment at time %f\n",time);
    ffit=fpreset;
    npass=1;
    top=0;
    time=0.0;
}

binpreset=fpreset*2.0*num_freq/srate;
minbin=binpreset-nbins;
if (minbin<0) minbin=0;
maxbin=binpreset+nbins;
if (maxbin>num_freq) maxbin=num_freq;

/* copy short data to float data */
for (i=0;i<num_points;i++) data[i]=datas[i];

/* remove the spectral lines from the data set */
remove_spectral_lines(data,num_points,padded_length,nwdt,num_win,max_lines,
                    maxpass,&num_removed,line_list,mtap_spec_init,mtap_spec_final,0,minbin,maxbin);

/* if we fail to remove a line, amplitude set to zero, phase RETAINS PRIOR VALUE */
amp1=0.0;
/* look in the list of removed lines for the right one */
for (i=0;i<num_removed;i++) {
    /* the closest bin to our estimated frequency */
    dbin=binpreset-line_list[i].index;
    if (fabs(dbin)<=nbins) {
        creal=line_list[i].re+dbin*line_list[i].dcdb2r+
            0.5*dbin*dbin*line_list[i].d2cdb2r;
        cimag=line_list[i].im+dbin*line_list[i].dcdbi+
            0.5*dbin*dbin*line_list[i].d2cdb2i;
        amp1=2.0*sqrt(creal*creal+cimag*cimag);
        phase1=atan2(cimag,creal)+2.0*M_PI*fmod(fpreset*time,1.0);
        break;
    }
}

/* save data in a circular buffers *[0..num_display-1] */
amp[top]=amp1;
phase[top]=FMOD2PI(phase1);
times[top]=time;

/* how many values are we going to output to the graph? */
nprint=(npass<num_display)?npass:num_display;

/* cut out a piece for the linear fit */
if (npass>=2) {
    /* adjust the phases to avoid boundary jumps */
    offset=0.0;
    index=MOD(top-nprint+1);
    linitx[0]=times[index];
    linfty[0]=phase[index];
    for (i=1;i<nprint;i++) {
        index=MOD(top+i-nprint+1);
        linitx[i]=times[index];
        if (phase[index]-phase[MOD(index-1)]>M_PI)
```

```
        offset-=2.0*M_PI;
    else if (phase[index]-phase[MOD(index-1)]<=-M_PI)
        offset+=2.0*M_PI;
    linfity[i]=phase[index]+offset;
}

/* do a robust linear fit */
medfit(linfitx-1,linfity-1,nprint,&intercept,&slope,&deviation);

/* now see what frequency the best fit corresponds to */
ffit=fpreset-slope/(2.0*M_PI);

/* if we are assuming a fixed frequency (not adapting) */
if (!estimate) {
    slope=intercept=0.0;
}

/* print out amplitude if non-zero */
maxamp=0.0;
for (i=0;i<nprint;i++) {
    index=MOD(top+i-nprint+1);
    if (amp[index]>0.0)
        printf("%e\t%e\n",linfitx[i],amp[index]);
    else
        /* won't appear on the graph - out of bounds */
        printf("%e\t%f\n",linfitx[i],-1.0);

    if (amp[index]>maxamp) maxamp=amp[index];
}
/* separate data sets */
printf("&\n");
/* print out phase if non-zero amplitude */
for (i=0;i<nprint;i++) {
    phase2=linfity[i];
    phase2=FMOD2PI((phase2-slope*linfitx[i]-intercept));
    if (phase2>M_PI)
        phase2-=2.0*M_PI;
    phase2=(180.0/M_PI)*phase2;
    index=MOD(top+i-nprint+1);
    if (amp[index]>0.0)
        printf("%.8e\t%.8f\n",linfitx[i],phase2);
    else
        /* won't appear on the graph - out of bounds */
        printf("%.8e\t%f\n",linfitx[i],-500.0);
}
/* set up scale of the x-axis */
t1=linfitx[0];
displaytime=num_display*(num_points/srate);
/* set up scale of the amplitude graph y-axis */
if (maxamp>0.9*displayamp) {
    displayamp=1.3*maxamp;
    fprintf(stderr,"\aTracker: Line at %f Hz, amplitude just increased\n",fpreset);
}
else if (maxamp<0.4*displayamp && maxamp>0.0)
    displayamp=1.3*maxamp;

graphout(t1,t1+displaytime,ffit,displayamp);
fflush(stdout);
}
```

```
        /* now display set, then kill set */
        npass++;
        top=MOD(top+1);
    }

    return 0;
}

void graphout(float t1,float t2,float freq,float displayamp) {
    static int count=0;
    int xmaj,xmin;
    float ymaj=0.0,ymin=1.0;
    int amprec;

    xmin=(t2-t1)/10.0;
    xmaj=5*xmin;

    if (ymin<=displayamp/10.0)
        while (ymin<=displayamp/10.0) {
            ymin*=2.0;
            ymaj=4.0*ymin;
        }
    else
        while (ymin>displayamp/10.0) {
            ymin/=10.0;
            ymaj=5.0*ymin;
        }
    amprec=(int)log10(ymaj);
    if (amprec>1)
        amprec=0;
    else
        amprec=1-amprec;

    /* end of set marker */
    printf("&\n");

    if (count==0) {
        /* first time we draw the plot */
        printf("@doublebuffer true\n");
        printf("@focus off\n");
    }
    printf("@with g0\n");
    printf("@move g0.s1 to g1.s0\n");
    printf("@title \"\\-Line Tracker\"\n");
    printf("@subtitle \"best estimate f=%f Hz\"\n",freq);
    printf("@s0 linestyle 0\n");
    printf("@s0 symbol color 4\n");
    printf("@s0 symbol 2\n");
    printf("@s0 symbol size 0.28\n");
    printf("@s0 symbol fill 1\n");
    printf("@view 0.15, 0.53, 0.95, 0.90\n");
    /* set up x-axis for amplitude */
    printf("@world xmin %e\n",t1);
    printf("@world xmax %e\n",t2);
    printf("@xaxis tick major %d\n",xmaj);
    printf("@xaxis tick minor %d\n",xmin);
    printf("@xaxis ticklabel prec 1\n");
    printf("@xaxis ticklabel off\n");
}
```

```
printf("@yaxis label \\\"\\-amplitude (ADC counts)\\\"\\n");
printf("@world ymin %e\\n",0.0);
printf("@world ymax %e\\n",displayamp);
printf("@yaxis tick major %e\\n",ymaj);
printf("@yaxis tick minor %e\\n",ymin);
if (amprec<4)
    printf("@yaxis ticklabel prec %d\\n",amprec);
else {
    printf("@yaxis ticklabel format general\\n");
    printf("@yaxis ticklabel prec %d\\n",1);
}
/* now do phase plot */
printf("@with g1\\n");
printf("@s0 linestyle 0\\n");
printf("@s0 linewidth 0\\n");
printf("@s0 symbol color 2\\n");
printf("@s0 symbol 2\\n");
printf("@s0 symbol size 0.28\\n");
printf("@s0 symbol fill 1\\n");
printf("@view 0.15, 0.1, 0.95, 0.47\\n");
/* set up x-axis for phase */
printf("@world xmin %e\\n",t1);
printf("@world xmax %e\\n",t2);
printf("@xaxis tick major %d\\n",xmaj);
printf("@xaxis tick minor %d\\n",xmin);
printf("@xaxis ticklabel prec 1\\n");
printf("@xaxis label \\\"\\-time (sec)\\\"\\n");
/* set up y-axis for phase */
printf("@world ymin %e\\n",-180.0);
printf("@world ymax %e\\n",180.0);
printf("@yaxis tick major 90\\n");
printf("@yaxis tick minor 45\\n");
printf("@yaxis ticklabel prec 0\\n");
printf("@yaxis label \\\"\\-phase (degrees)\\\"\\n");
printf("@xaxis label \\\"\\-time (sec)\\\"\\n");
/* draw plot */
printf("@redraw\\n");
printf("@kill s0\\n");
printf("@with g0\\n");
printf("@kill s0\\n");
    count++;
return;
}
```

Authors: Bruce Allen (ballen@dirac.phys.uwm.edu) and Adrian Ottewill (ottewill@relativity.ucd.ie).

Comments: None.

17 Time Standards: UTC, GPS, TAI, and Unix-C times.

The relationship between different time-standards and labels is somewhat complex. The following web sites contains further information about these different time standards:

`ftp://tycho.usno.navy.mil/pub/series/ser14.txt`

`ftp://maia.usno.navy.mil/ser7/tai-utc.dat`

`http://tycho.usno.navy.mil/time.html` .

What is presented here is merely a summary for the perplexed. The situation is complicated slightly because on standard Unix machines (Solaris 2.6, Linux 2.0*, etc) the time functions in the “ANSI-standard C library” do not behave according to the rather vaguely-defined ANSI/POSIX standards. (In particular, the `gmtime()` function in the C standard library does not know about leap-seconds – although the documentation generally suggests that it should!)

Let us begin by defining a physical time t , which is the time coordinate used in physics lectures. It advances linearly, completing each unit step at the same instant that a perfect pendulum with a frequency of 1 Hz completes a new oscillation. Without loss of generality we will set $t = 0$ at the stroke of midnight, January 1, 1970 UTC. This time is also denoted Jan 1 00:00:00 UTC. Note that we will not discuss *local* time in this section, which differs from UTC by a fixed number of hours that depend upon your time-zone and upon whether or not daylight savings time is in effect. We will assume that once you know the UTC time, you can do the necessary addition or subtraction yourself, to determine local time at any point of interest on the earth.

Universal Coordinated Time (UTC), Global Positioning System (GPS) and International Atomic Time (TAI) are all well-defined global standards. By Unix-C time we mean here the value of t . This is what is returned by the Standard C-library `time()` function, which advances its output by one every second, starting from Jan 1 00:00:00 UTC, *provided that the computer is started at Jan 1 00:00:00 UTC and runs continuously and without error after that.* (See below for an explanation of why this qualification is required!)

UTC should be thought of as a system for attaching “human readable” labels such as “March 11, 1983, at 12:10” to particular moments in time. (We use “military” or 24-hour time to distinguish am and pm.) This is done by advancing in the standard pattern of 1 hour every 3600 seconds, one day every 24 hours, and so on, with two exceptions. These exceptions are necessary because if they were not made, eventually people in the northern hemisphere would be suffering snowfalls in August.

The first of the two exceptions is easy. Once every four years, according to the pattern 1972, 1976, ... there is a leap year, which is a year containing one extra day. In these years, February has 29 days not 28. This affects the pattern by which UTC time follows t , and is a minor nuisance. However, because it follows a regular deterministic pattern, leap years with their extra day present no real problems.

The real complications arise because the earth is gradually slowing in its rotation about its axis, from the effects of earth-moon and earth-sun tidal friction, and because the sunspot cycles affects the heating of the earth and thus its mass distribution and moment of inertia. These effects are not easily predicted in advance, and thus on a regular basis (between once every two years and twice a year) an extra second is inserted into the UTC label. The decision about when to do this is made by the International Earth Rotation Service (IERS) `http://hpiers.obspm.fr/`. This one extra second is generally added right after what would be the final second of a month (generally December or June). Thus, at these times, the normal pattern of ..., 23:59:58, 23:59:59, 00:00:00 ... is broken and replaced by ..., 23:59:58, 23:59:59, 23:59:60, 00:00:00, The next leap second will be inserted into UTC in this way at the end of December 1998. In principle, a leap second can be either positive or negative, although there have not yet (as of September 1998) been any negative leap seconds.

Unfortunately, there is one additional small complication. Until Jan 1, 1972 UTC, the duration of one UTC second was not equal to the duration of one (physical, TAI=GPS=CTime) second. This can be seen most easily from `ftp://maia.usno.navy.mil/ser7/tai-utc.dat` . Here is a small extract

from that file.

```

...
1966 JAN 1 =JD 2439126.5 TAI-UTC= 4.3131700 S + (MJD - 39126.) X 0.002592 S
1968 FEB 1 =JD 2439887.5 TAI-UTC= 4.2131700 S + (MJD - 39126.) X 0.002592 S
1972 JAN 1 =JD 2441317.5 TAI-UTC= 10.0 S + (MJD - 41317.) X 0.0 S
1972 JUL 1 =JD 2441499.5 TAI-UTC= 11.0 S + (MJD - 41317.) X 0.0 S
1973 JAN 1 =JD 2441683.5 TAI-UTC= 12.0 S + (MJD - 41317.) X 0.0 S
...

```

Here, the Modified Julian Day (MJD) is defined as follows: $MJD = JD - 2400000.5$, where the Julian Day increments by one at noon every day. Notice that these relationships imply that, before January 1, 1972 UTC, the difference between atomic time (TAI) and UTC time is *not* an integer number of seconds!

The values of JD and MJD at some interesting times are given below

```

1968 Feb 1 00:00:00 UTC JD=2439887.5 MJD=39887
1969 Feb 1 00:00:00 UTC JD=2440253.5 MJD=40253
1970 Jan 1 00:00:00 UTC JD=2440587.5 MJD=40587
1970 Feb 1 00:00:00 UTC JD=2440618.5 MJD=40618
1971 Jan 1 00:00:00 UTC JD=2440952.5 MJD=40952
1972 Jan 1 00:00:00 UTC JD=2441317.5 MJD=41317

```

In particular, on Jan 1 00:00:00 1970 UTC, the formula above gives:

$$TAI-UTC = 4.2131700 \text{ sec} + (40587 - 39126) \times 0.002592 \text{ sec} = 8.000082 \text{ sec.} \quad (17.0.1)$$

We will ignore the 82 microseconds in what follows.

Much of the complication arises because the standard Unix C-library function `gmtime()`, which takes as its argument the number of seconds after Jan 1, 1970 UTC, and should return the UTC time, *does not return the correct UTC time. The function gmtime() fails to take account of leap seconds.* The relationship between t , Unix-C time, and UTC time is demonstrated in the table below, which shows the effects of the leap seconds and the erroneous behavior of `gmtime()`. Note that in this table, the definition of “leap second” is not made precise until on or after Jan 1, 1972 UTC. Until that time, the number of leap seconds can be non-integer: here we have assumed that it is integer. (This is of no consequence provided that we only study the relationship between our different time standards for times after Jan 1, 1972 UTC).

You can see that UTC time and the quantity returned by `gmtime()` move gradually out of synch with one another. Currently (October 1998) the two quantities have drifted 23 seconds out of sync.

The most easily used time standard today is GPS time, because GPS receivers are cheap, accurate, and available. GPS time is equal to t plus a constant. GPS was set to be zero on Jan 6 00:00:00 1980 UTC. Hence $t = \text{GPS} + 315964811 \text{ sec}$. This is because, up to the 82 microseconds mentioned in equation (17.0.1), one has

$$\text{Jan 6 00:00:00 1980 UTC} - \text{Jan 1 00:00:00 1970} = 315964811 \text{ sec.} \quad (17.0.2)$$

This number of seconds is obtained from:

$$315964811 \text{ sec} = 3600 \text{ sec/hour} \times 24 \text{ hours/day} \times \\ (365 \text{ days/year} \times 8 \text{ years} + 366 \text{ days/year} \times 2 \text{ years} + 5 \text{ days}) + 11 \text{ leap seconds} \quad (17.0.3)$$

You can see that the relationship between GPS time and Unix-C time would be

$$\text{Unix-C} = \text{GPS} + 315964811 \text{ sec} \quad \text{for a computer started Jan 1, 1970 UTC.} \quad (17.0.4)$$

t physical (sec)	GPS time (sec)	Unix-C time ()	UTC time label	gmtime () function of time ()	Leap sec TAI-UTC
0	-315964811	0	Jan 1 00:00:00 1970 UTC	Jan 1 00:00:00 1970	8
1	-315964810	1	Jan 1 00:00:01 1970 UTC	Jan 1 00:00:01 1970	8
...					
33350399	-282614412	33350399	Jan 21 23:59:59 1971 UTC	Jan 21 23:59:59 1971	8
33350400	-282614411	33350400	Jan 21 23:59:60 1971 UTC	Jan 22 00:00:00 1971	8
33350401	-282614410	33350401	Jan 22 00:00:00 1972 UTC	Jan 22 00:00:01 1971	9
...					
63071999	-252892812	63071999	Dec 31 23:59:58 1971 UTC	Dec 31 23:59:59 1971	9
63072000	-252892811	63072000	Dec 31 23:59:59 1971 UTC	Jan 1 00:00:00 1972	9
63072001	-252892810	63072001	Dec 31 23:59:60 1971 UTC	Jan 1 00:00:01 1972	9
63072002	-252892809	63072002	Jan 1 00:00:00 1972 UTC	Jan 1 00:00:02 1972	10
...					
78796800	-237168011	78796800	Jun 30 23:59:58 1972 UTC	Jul 1 00:00:00 1972	10
78796801	-237168010	78796801	Jun 30 23:59:59 1972 UTC	Jul 1 00:00:01 1972	10
78796802	-237168009	78796802	Jun 30 23:59:60 1972 UTC	Jul 1 00:00:02 1972	10
78796803	-237168008	78796803	Jul 1 00:00:00 1972 UTC	Jul 1 00:00:03 1972	11
...					
315964810	-1	315964810	Jan 5 23:59:59 1980 UTC	Jan 6 00:00:10 1980	19
315964811	0	315964811	Jan 6 00:00:00 1980 UTC	Jan 6 00:00:11 1980	19
315964812	1	315964812	Jan 6 00:00:01 1980 UTC	Jan 6 00:00:12 1980	19
...					
784880276	468915465	784880276	Nov 15 06:17:35 1994 UTC	Nov 15 06:17:56 1994	29
784880277	468915466	784880277	Nov 15 06:17:36 1994 UTC	Nov 15 06:17:57 1994	29
784880278	468915467	784880278	Nov 15 06:17:37 1994 UTC	Nov 15 06:17:58 1994	29
...					
911110676	595145865	911110676	Nov 15 06:17:33 1998 UTC	Nov 15 06:17:56 1998	31
911110677	595145866	911110677	Nov 15 06:17:34 1998 UTC	Nov 15 06:17:57 1998	31
911110678	595145867	911110678	Nov 15 06:17:35 1998 UTC	Nov 15 06:17:58 1998	31
...					

Table 11: Relationship between different time coordinates. The UTC time should be thought of as a “human readable” label that gets attached to the time quantities. The number of leap seconds is often denoted by TAI-UTC: it is equal to the amount by which the Unix notion of UTC differs from the actual time: $TAI-UTC = \text{gmtime}(\text{time}()) - \text{UTC} + 8 \text{ sec}$. The most severe problem with the Unix notion of time is that in actually setting the time on an individual machine, one sets the `time()` function to return the wrong value. Thus, at physical time $t = 63072002$ if you correctly set the machine time to Jan 1 00:00:00 1972 UTC, and then immediately call the `time()` function, it will (incorrectly) return the value 63072000. (Note that this table incorrectly treats TAI-UTC as an integer before Jan 1, 1972 UTC.)

if you started a perfect computer with a perfect internal clock running on Jan 1, 1970 UTC and left it running forever. On the other hand, if you started this computer off on Jan 1, 1980 its internal Unix-C time would be set to 315964800 and the relationship would be instead

$$\text{Unix-C} = \text{GPS} + 315964800 \text{ sec} \quad \text{for a computer started Jan 1, 1980 UTC.} \quad (17.0.5)$$

For a computer started at some other time, some other relationship will hold.

The final time coordinate in general use is $\text{TAI} = \text{GPS} + 19 \text{ sec}$. This obeys the relationship

$$t = 315964811 \text{ sec} + \text{TAI} - 19 = 315964792 \text{ sec} + \text{TAI.} \quad (17.0.6)$$

The GRASP library contains a pair of utility functions `utctime()` and `gpstime()` which are similar to `gmtime()`, except that they correctly generate the UTC label for Unix-C time and GPS time (respectively).

Because the number of leap seconds is not known in advance (we can not predict how the earth's rotation rate will change in the future) it is often useful to store in time records both a physical time label (for example GPS time) and in addition the number of leap seconds, in the form (TAI-UTC). *Because* the Unix-C library function `gmtime()` is broken, we can use it to print the correct "human-readable" UTC times *almost* all of the time, by using:

$$\begin{aligned} \text{UTC} &= \text{gmtime}*(\text{Unix-C} - (\text{TAI-UTC}) + 8) \\ &= \text{gmtime}*(\text{GPS} + 315964811 - (\text{TAI-UTC}) + 8) \\ &= \text{gmtime}*(\text{GPS} + 315964800 - (\text{TAI-UTC}) + 19) \end{aligned} \quad (17.0.7)$$

where `*` means "pointer to". This will work properly, except that it will return identical values for two consecutive seconds, when one occurs directly before an additional leap second is added, and the second occurs the first second of the new leap second. This is because the `gmtime()` function will never return a time of the form `XX:XX:60`. The range of seconds returned by this function is `0, ..., 59`.

17.1 Function: utctime()

```
struct tm *utctime(const time_t *tp)
```

This is a “convenience” function which will do what the standard Unix C library function `gmtime()` is supposed to do, but fails to do. That is, it prints out the UTC time. The arguments and value returned by this function are exactly the same as for the Unix standard C library function `gmtime()`.

`tp`: Input. Pointer to an object of type `time_t` which is the number of seconds after Jan 1, 1970 00:00:00 UTC.

The function returns a pointer to a `struct tm` structure. Thus `asctime(utctime(*tp))` is an ascii string showing the UTC time `tp`. This function will warn the user if the `tp` argument is before or after the range of validity of the function.

Author: Bruce Allen (ballen@dirac.phys.uwm.edu).

Comments: This function contains a table of all known leap seconds. It will need to be revised when the next leap second after Dec 31, 1998 is added. Note that if there is a negative leap second the coding will need to be modified. If it is a positive leap second then one additional entry needs to be made to an internal table.

17.2 Function: `gpstime()`

```
struct tm *gpstime(const time_t *tp)
```

This is a “convenience” function which returns the same quantity as the previous function `utctime()` but takes as its argument the GPS time, which is the number of seconds after Jan 6, 1980 00:00:00 UTC. The value returned by this function are exactly the same as for the Unix standard C library function `gmtime()`. Thus `asctime(utctime(*tp))` is an ascii string showing the UTC time.

`tp`: Input. Pointer to an object of type `time_t` which is the number of seconds after Jan 6, 1980 00:00:00 UTC.

The function returns a pointer to a `struct tm` structure. It will warn the user if the `tp` argument is before or after the range of validity of the function.

Author: Bruce Allen (ballen@dirac.phys.uwm.edu).

Comments: This function calls `utctime()`, so please see the comments for `utctime()`.

17.3 Example: testutctime

This example program demonstrates functions `utctime()` and `gpstime()`, and shows the relationship between the UTC time and the number of seconds after a given time. It was used to generate parts of the Table 11, and shows exactly how the standard Unix C library function `gmtime()` is broken.

Here is some output:

```
C-time:      0 GPS Time -315964811 UTC: Thu Jan  1 00:00:00 1970 GPS: Thu Jan  1 00:00:00 1970 gmtime: Thu Jan  1 00:00:00 1970
C-time:      1 GPS Time -315964810 UTC: Thu Jan  1 00:00:01 1970 GPS: Thu Jan  1 00:00:01 1970 gmtime: Thu Jan  1 00:00:01 1970
C-time:      2 GPS Time -315964809 UTC: Thu Jan  1 00:00:02 1970 GPS: Thu Jan  1 00:00:02 1970 gmtime: Thu Jan  1 00:00:02 1970

C-time: 33350399 GPS Time -282614412 UTC: Thu Jan 21 23:59:59 1971 GPS: Thu Jan 21 23:59:59 1971 gmtime: Thu Jan 21 23:59:59 1971
C-time: 33350400 GPS Time -282614411 UTC: Thu Jan 21 23:59:60 1971 GPS: Thu Jan 21 23:59:60 1971 gmtime: Fri Jan 22 00:00:00 1971
C-time: 33350401 GPS Time -282614410 UTC: Fri Jan 22 00:00:00 1971 GPS: Fri Jan 22 00:00:00 1971 gmtime: Fri Jan 22 00:00:01 1971

C-time: 63071999 GPS Time -252892812 UTC: Fri Dec 31 23:59:58 1971 GPS: Fri Dec 31 23:59:58 1971 gmtime: Fri Dec 31 23:59:59 1971
C-time: 63072000 GPS Time -252892811 UTC: Fri Dec 31 23:59:59 1971 GPS: Fri Dec 31 23:59:59 1971 gmtime: Sat Jan  1 00:00:00 1972
C-time: 63072001 GPS Time -252892810 UTC: Fri Dec 31 23:59:60 1971 GPS: Fri Dec 31 23:59:60 1971 gmtime: Sat Jan  1 00:00:01 1972

C-time: 78796800 GPS Time -237168011 UTC: Fri Jun 30 23:59:58 1972 GPS: Fri Jun 30 23:59:58 1972 gmtime: Sat Jul  1 00:00:00 1972
C-time: 78796801 GPS Time -237168010 UTC: Fri Jun 30 23:59:59 1972 GPS: Fri Jun 30 23:59:59 1972 gmtime: Sat Jul  1 00:00:01 1972
C-time: 78796802 GPS Time -237168009 UTC: Fri Jun 30 23:59:60 1972 GPS: Fri Jun 30 23:59:60 1972 gmtime: Sat Jul  1 00:00:02 1972

C-time: 94694401 GPS Time -221270410 UTC: Sun Dec 31 23:59:58 1972 GPS: Sun Dec 31 23:59:58 1972 gmtime: Mon Jan  1 00:00:01 1973
C-time: 94694402 GPS Time -221270409 UTC: Sun Dec 31 23:59:59 1972 GPS: Sun Dec 31 23:59:59 1972 gmtime: Mon Jan  1 00:00:02 1973
C-time: 94694403 GPS Time -221270408 UTC: Sun Dec 31 23:59:60 1972 GPS: Sun Dec 31 23:59:60 1972 gmtime: Mon Jan  1 00:00:03 1973

C-time: 126230402 GPS Time -189734409 UTC: Mon Dec 31 23:59:58 1973 GPS: Mon Dec 31 23:59:58 1973 gmtime: Tue Jan  1 00:00:02 1974
C-time: 126230403 GPS Time -189734408 UTC: Mon Dec 31 23:59:59 1973 GPS: Mon Dec 31 23:59:59 1973 gmtime: Tue Jan  1 00:00:03 1974
C-time: 126230404 GPS Time -189734407 UTC: Mon Dec 31 23:59:60 1973 GPS: Mon Dec 31 23:59:60 1973 gmtime: Tue Jan  1 00:00:04 1974

C-time: 315964810 GPS Time      -1 UTC: Sat Jan  5 23:59:59 1980 GPS: Sat Jan  5 23:59:59 1980 gmtime: Sun Jan  6 00:00:10 1980
C-time: 315964811 GPS Time      0 UTC: Sun Jan  6 00:00:00 1980 GPS: Sun Jan  6 00:00:00 1980 gmtime: Sun Jan  6 00:00:11 1980
C-time: 315964812 GPS Time      1 UTC: Sun Jan  6 00:00:01 1980 GPS: Sun Jan  6 00:00:01 1980 gmtime: Sun Jan  6 00:00:12 1980

C-time: 784880276 GPS Time 468915465 UTC: Tue Nov 15 06:17:35 1994 GPS: Tue Nov 15 06:17:35 1994 gmtime: Tue Nov 15 06:17:56 1994
C-time: 784880277 GPS Time 468915466 UTC: Tue Nov 15 06:17:36 1994 GPS: Tue Nov 15 06:17:36 1994 gmtime: Tue Nov 15 06:17:57 1994
C-time: 784880278 GPS Time 468915467 UTC: Tue Nov 15 06:17:37 1994 GPS: Tue Nov 15 06:17:37 1994 gmtime: Tue Nov 15 06:17:58 1994

C-time: 911110676 GPS Time 595145865 UTC: Sun Nov 15 06:17:33 1998 GPS: Sun Nov 15 06:17:33 1998 gmtime: Sun Nov 15 06:17:56 1998
C-time: 911110677 GPS Time 595145866 UTC: Sun Nov 15 06:17:34 1998 GPS: Sun Nov 15 06:17:34 1998 gmtime: Sun Nov 15 06:17:57 1998
C-time: 911110678 GPS Time 595145867 UTC: Sun Nov 15 06:17:35 1998 GPS: Sun Nov 15 06:17:35 1998 gmtime: Sun Nov 15 06:17:58 1998
```

Author: Bruce Allen (ballen@dirac.phys.uwm.edu).

Comments: None.

18 Matlab Interface: Gravitational Radiation Toolbox

The Gravitational Radiation Toolbox provides a Matlab interface to both GRASP and the Frame Library. The toolbox consists of Matlab m-files and mex-files. The m-files are written in Matlab's native language and are interpreted. The mex-files are written in C and are compiled. The mex-files take input from the user via Matlab and then format it and pass it to the corresponding GRASP routine. The GRASP output is then formatted and passed back into Matlab. The function names and calling methods have been made to resemble those of their corresponding GRASP functions with the main difference being that the Matlab user need not be concerned with memory management and pointers. These functions can be used from the command line, within other m-files, or via a GUI front end called `GRTool`. Currently the toolbox addresses only the detection and simulation of gravitational radiation from binary inspirals.

When installing GRASP you have the option of installing the Gravitational Radiation Toolbox as well. If you have not done this see sections 2.6.8 and 2.6.9 for installation instructions.

18.1 Using GRtool

The easiest way to test out the toolbox is by using the GUI GRtool. Once you have successfully installed the toolbox, typing GRtool at the Matlab prompt should produce the window shown in figure 88. At the left of the figure are axes upon which time and frequency domain data can be displayed. At start-up the only options are to Read or Simulate data.

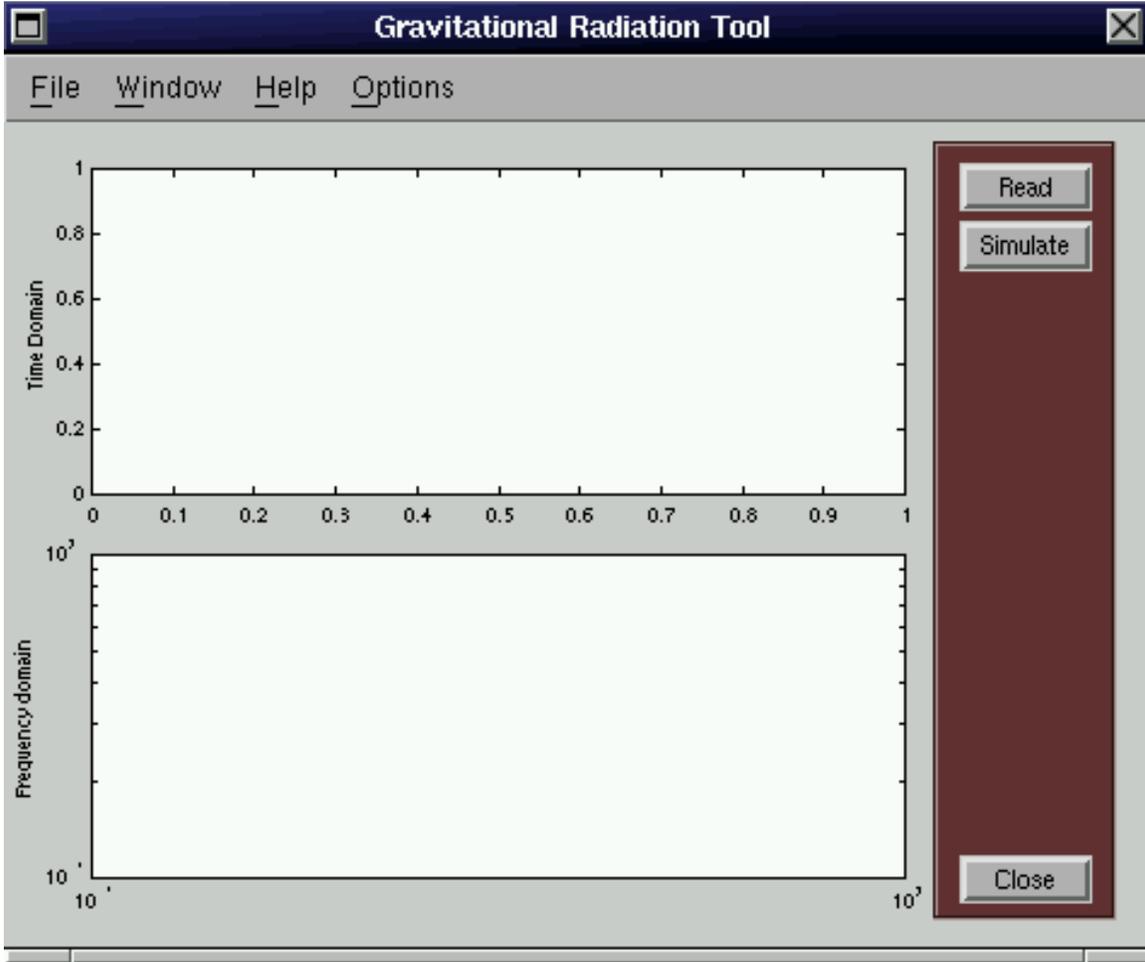


Figure 88: The GRtool start-up window

If you press Simulate you will enter the simulation GUI from which you can simulate inspirals up to second post-Newtonian order approximations as described in [7] and [8]. (Note: You can always run the simulation GUI alone by entering GRtool('simulate_build') at the Matlab prompt.) All simulations are done via a function mxMake_filters which is a Matlab version of the function make_filters. The operation of the simulation GUI is straightforward. You can plot the time and frequency domain simulations, play them as sound, or export them to either the Matlab workspace or *.au sound files.

If you press Read you will be asked a series of questions. At first you will be asked if you would like to read from local disks or from a URL. If you choose local disk, you will then be asked what file you want to open. Only frame formatted data files with the LIGO gravity wave channel (IFO_DMRO) can be read—this goes for data read from a local disk or a URL. If you choose URL you will be prompted for a URL address. After providing the URL you will be asked where you would like to write the contents of the URL. When

the program has read the data you will be presented with more buttons on the panel to the right. Figure 89 shows the results of a successful read of a frame file. The data plotted will only appear if

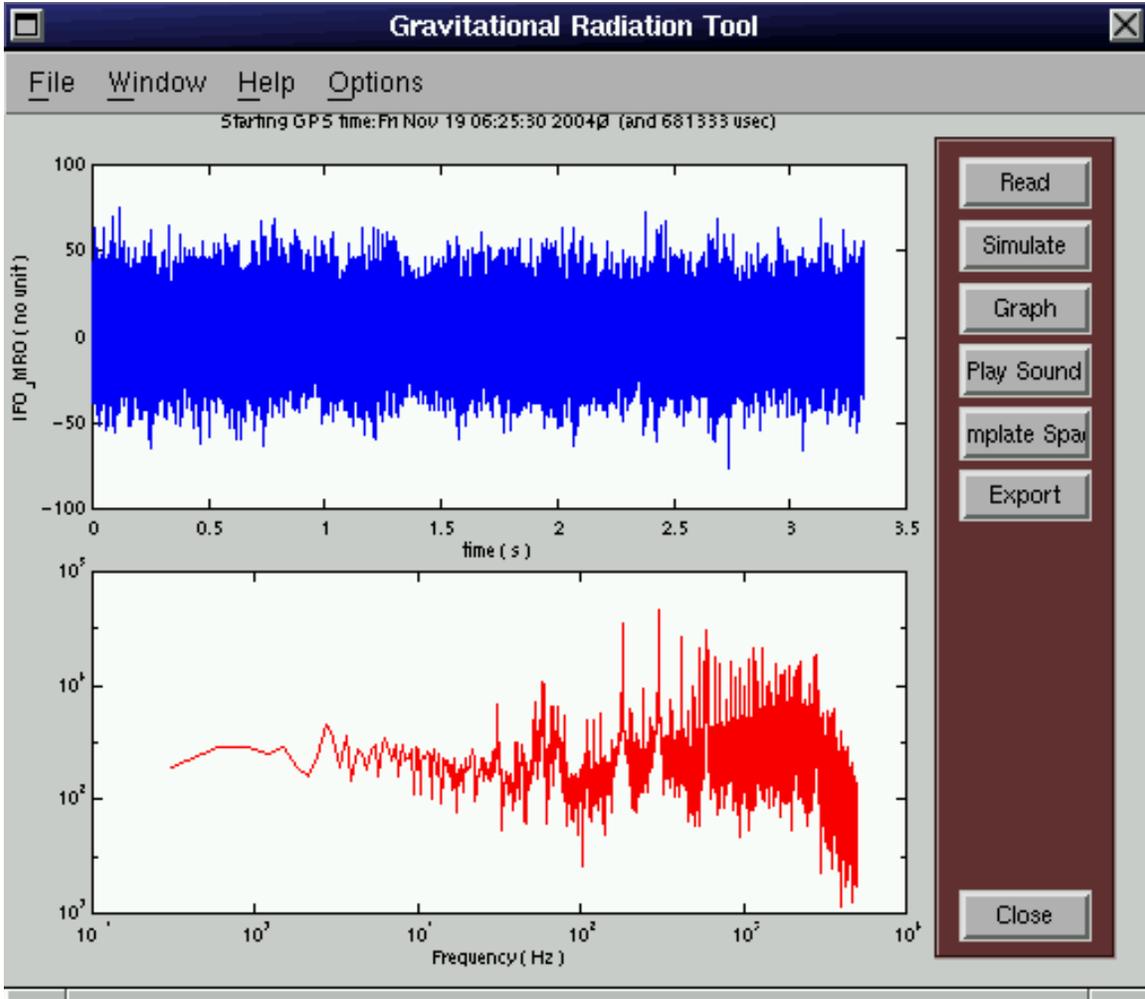


Figure 89: Viewing data in the time and frequency domain

you press `Graph` and then enter a time-span for which you want to look at data. The `Play Sound` button will playback the data from your speakers. The `Export` button allows you to export time domain data to a *.au sound file or the Matlab workspace. If you export the data to the workspace it will be preserved there even if you close `GRtool`. Then you can do with it what you like as it will behave as any other normal Matlab variable.

Pressing `Template Space` will change your view from time/frequency space to template space. Your window will change to look like figure 90.

From template space you can observe the effect of the data set on a grid of matched filters. You can always go back to see the data in time/frequency space by pressing `t/f Space`. Similarly from time/frequency space, you can always go to template space by pressing `Template Space`.

When you enter template space you will also see the template space control panel shown in figure 91.

With this window you control the files and parameters used for any filtering you do. You can always change a parameter or file by pressing it's corresponding `Update` button. The calibration file must be a text

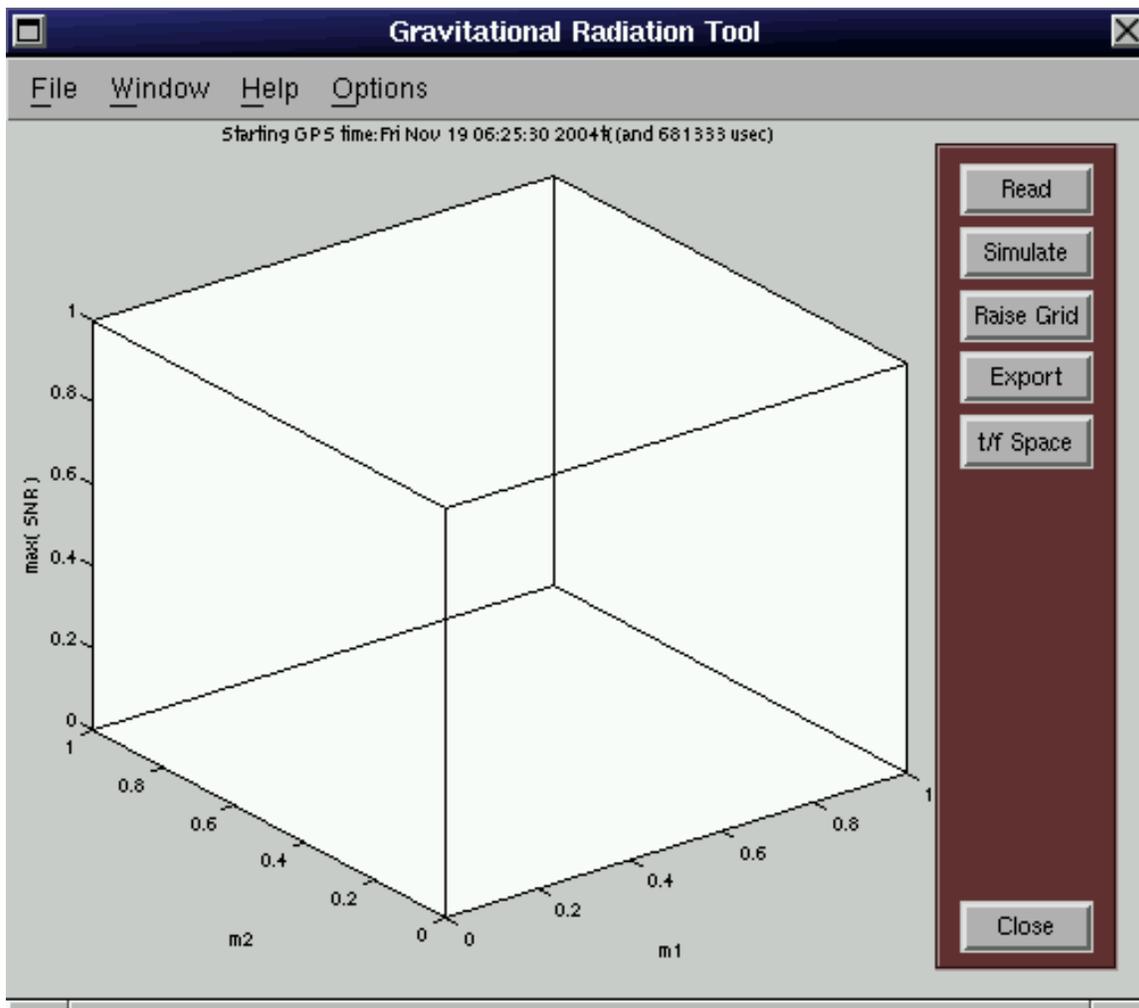


Figure 90: The initial template space window

file with one column of numbers which are the swept sine calibration information contained in the `fri` array returned by the function `fget_ch` or it's Matlab counterpart `mxFget_ch`. You can generate the calibration files using the `getfri` function as described in section 18.2.2. The templates file must be a two column text file. The two columns are the (m_1, m_2) pairs that you wish to include in your grid of filters. There is a very detailed discussion of what (m_1, m_2) pairs to use in section 13. There is a sample templates file in `$GRASP/src/examples/examples_binary-search/templates.ascii` where `$GRASP` is your GRASP root directory. The Low frequency cutoff is the lowest frequency at which the detector you are interested in can operate. This value must be entered in Hz.

After you have specified the necessary files and low frequency cutoff you can filter your data by pressing `Filter Data`. As the filters are being generated and compared against the data you will see lots of text streaming by in the workspace. This text will say something similar to the following:

```
GRASP (page-173.caltech.edu): Message from function phase_frequency()  
at line number 536 of file 'pN_chirp.c'.  
Frequency evolution no longer monotonic.  
Phase evolution terminated at frequency and step: 466.659027 4347
```

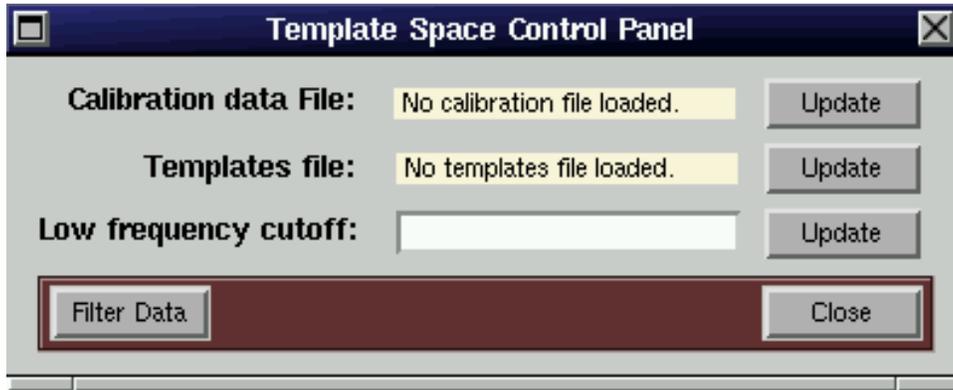


Figure 91: The template space control panel

```
Terminating chirp. Termination code set to: 1201
Returning to calling routine.
$Id: man_GRtoolbox.tex,v 1.3 1999/09/06 17:37:17 ballen Exp $
$Name: RELEASE_1_9_8 $
max snr: 2.56 offset: 24659 variance: 0.95574
max snr: 2.56 offset: 2376 variance: 0.96000
max snr: 2.70 offset: 10504 variance: 0.95999
Done filtering 32768 data points through 20 filters
```

The final line will always tell you how many data points and how many filters were used.

After you filter the data you can visualize the response of the filters by pressing `Raise Grid` in the main window. A typical response is shown in figure 92. The color of the plotted points is based on the time at which that specific filter had the greatest SNR. The colors range from pure blue to pure black. The darker the color the earlier the time.

Figure 93 shows the response of the filters when a simulated inspiral of $m_1 = m_2 = 1.4M_\odot$ has been injected into the data stream.

You can clearly see the filters responding. Notice also that the colors have a well defined pattern in this image. If you press `Export` in this window you will be able to export the arrays $m_1, m_2, \max(\text{SNR})$ and `timestart` to the workspace. The `timestart` array contains the time in seconds (relative to the first time stamp) at which the corresponding filter had a maximum. A more detailed description of the array `timestart` can be found in the description of the function `find_chirp` in section 6.20.

18.2 Functions

Bellow is a list of the functions available to you for use within Matlab. The majority of the functions are ports of other GRASP functions. These have names that are identical to their GRASP versions except their first letter is capitalized and they are prefixed by `mx`. For example the Matlab version of the GRASP function `make_filters` is called `mxMake_filters`. These functions differ from their GRASP counterparts only in their calling methods. **Warning:** you must be careful that you have set any environment variables required by the GRASP functions *before* starting Matlab. If you have not, calling functions that require these variables will abort Matlab and you will lose any data you had stored in the workspace! Use these functions cautiously!

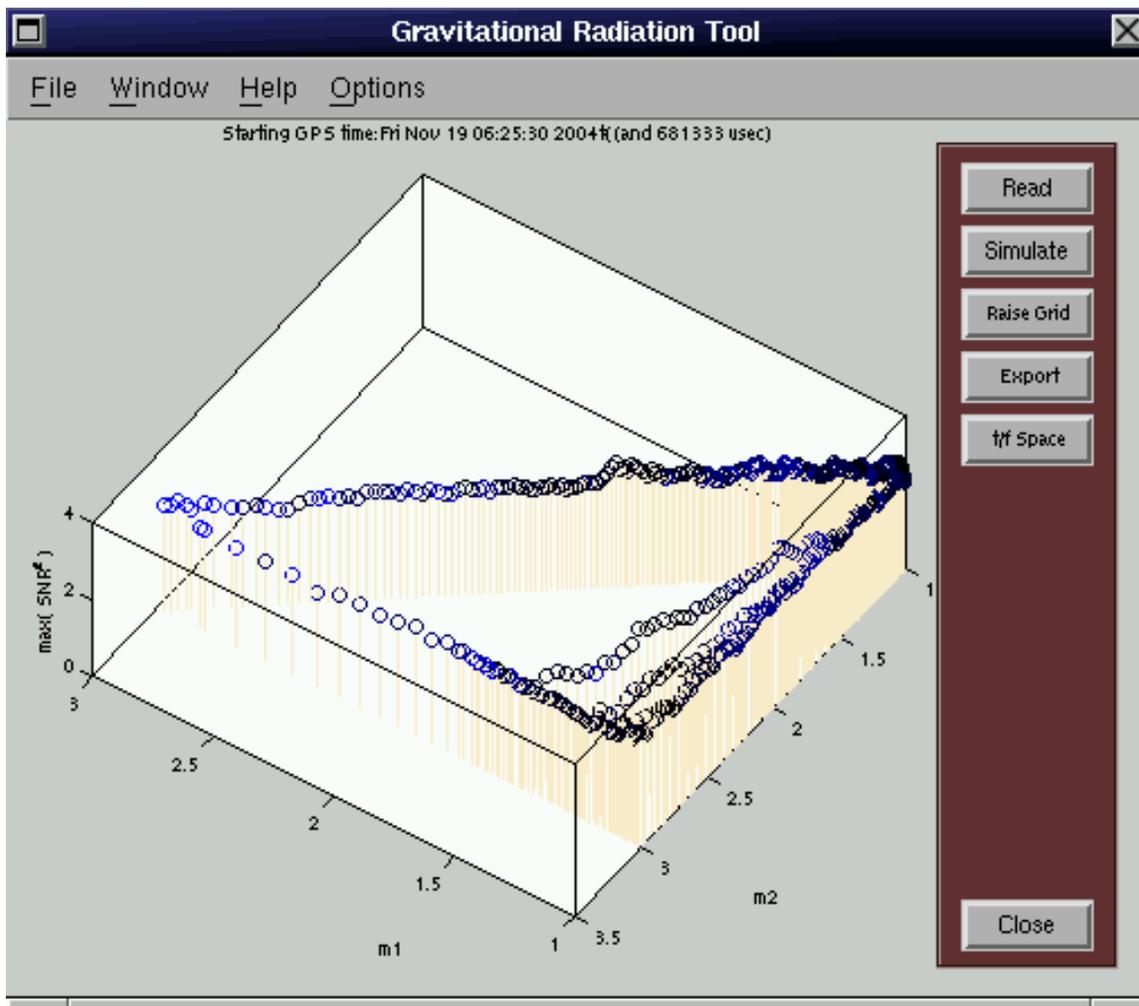


Figure 92: Viewing the data in template space

18.2.1 Function: `frextract`

```
[a,t,f,t0,t0s,c,u,more]=frextract(file,channel,firstframe,nframes)
```

This function is part of the standard frame library distribution. Extracts data from a frame file. See Section 6 of [47] for further documentation.

18.2.2 Function: `getfri`

`getfri('fri.ascii')` Generates calibration files for use with `GRtool`. Will create a file called `fri.ascii`. This function requires that you have set the environment variable `GRASP_FRAMEPATH` to point to the data file from which you want calibration data. This function calls the `mxFget_ch` function once. See the warning about its use in section 18.2.9.

Author: Steve Drasco, steve.drasco@cornell.edu

Comments: None.

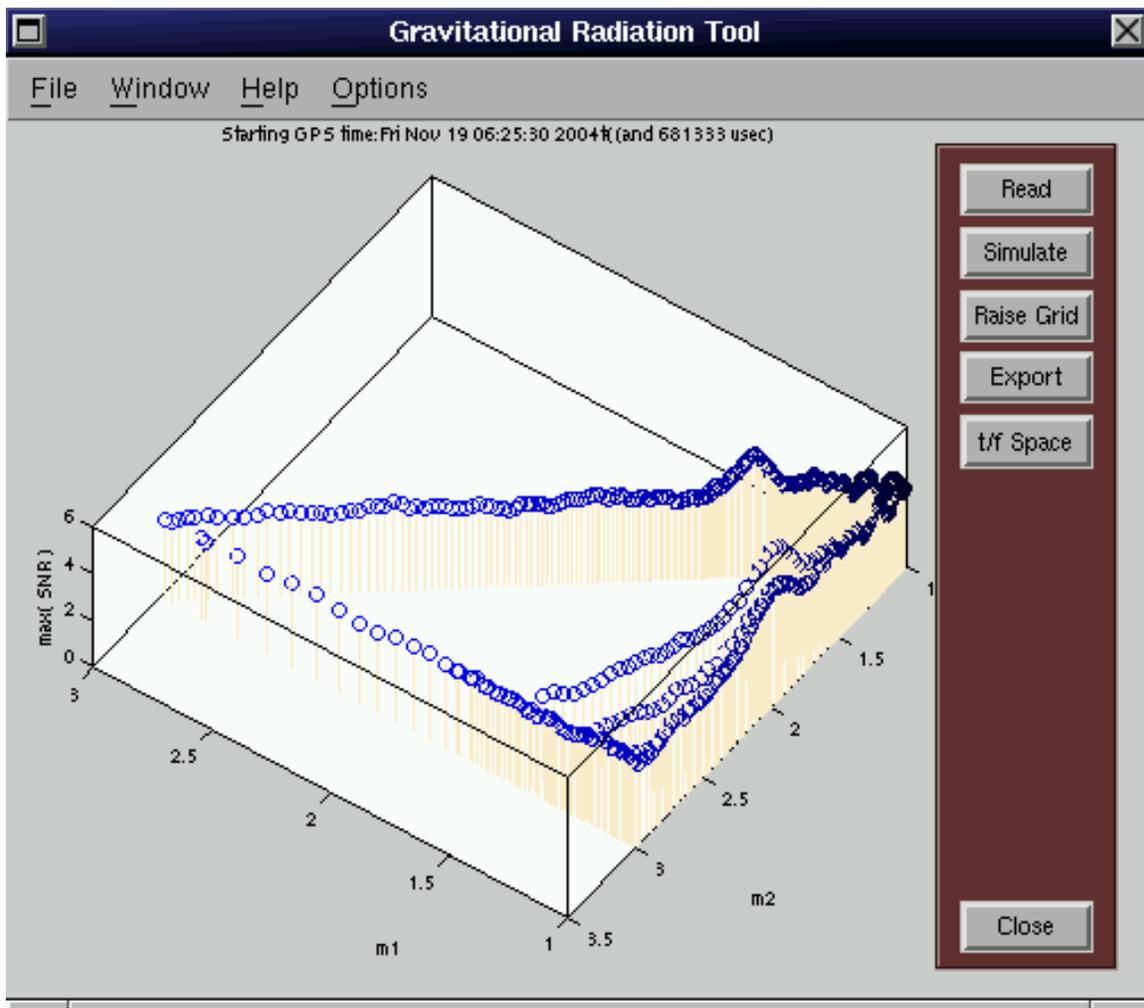


Figure 93: An artificial inspiral detection of $m_1 = m_2 = 1.4M_\odot$

18.2.3 Function: `inspfilt`

```
[snr_max, timestart]=inspfilt(m1,m2,data,srate,flo,fri)
```

Filters data through a grid of matched filters.

`m1` and `m2`: equal length vectors specifying the filters to use.

`data`: vector containing the data to be filtered.

`srate`: sampling frequency of the data in Hz.

`flo`: low frequency cutoff for the detector in Hz.

`fri`: vector containing the calibration data from the instrument.

`snr_max`: vector with the same length as `m1` and `m2`. It contains the maximum value of the SNR for each filter.

`timestart`: vector with the same length as `snr_max` which specifies the time in seconds (relative to the first time stamp) at which the corresponding filter reached the value contained in `snr_max`

Author: Steve Drasco, steve.drasco@cornell.edu

Comments: None.

18.2.4 Function: `mxAvg_inv_spec`

`[mean_pow_spec, twice_inv_noise, norm]=mxAvg_inv_spec(flo, srate, n, decay, norm, htilde)`
All variables are identical to their description in section 6.17.

Author: Steve Drasco, steve.drasco@cornell.edu

Comments: None.

18.2.5 Function: `mxChirp_filters`

`[Max_Freq_Actual, h_c, h_s, steps_filld, clscnc_time]=mxChirp_filters(m1, m2, spin1, spin2, n_phaseterms, phaseterms, Initial_Freq, Max_Freq_Rqst, Sample_Time, steps_alloc, err_cd_sprs)`
All variables are identical to their description in section 6.5.

Author: Steve Drasco, steve.drasco@cornell.edu

Comments: None.

18.2.6 Function: `mxCompute_match`

`[outvalue]=mxCompute_match(m1, m2, ch0tilde, ch90tilde, inverse_discance_scale, twice_inv_noise, flo, s_n0, s_n90, npoint, srate, err_cd_sprs, order)`
All variables are identical to their description in section 9.8.

Author: Steve Drasco, steve.drasco@cornell.edu

Comments: None.

18.2.7 Function: `mxCorrelate`

`[s]=mxCorrelate(h, c, r, n)`
All variables are identical to their description in section 6.16.

18.2.8 Function: `mxDetector_site`

`[site_parameters, site_name, noise_file, whiten_file]=mxDetector_site(detectors_file, site_choice)`
All variables are identical to their description in section 11.2.

Author: Steve Drasco, steve.drasco@cornell.edu

Comments: None.

18.2.9 Function: mxFget_ch

```
[fgetoutput]=mxFget_ch(fgetinput)
```

This function does not behave properly. GRASP treats Matlab as the calling function so it has no way of knowing when you want to reset your search. Be *especially* cautious if you use this function. Most of the input and output structures are the same—however since there are slight difference the full descriptions are given below.

`fgetinput.npoint`: the number of data points you want to get

`fgetinput.inlock`: 1 means get only locked data 0 means get both locked and unlocked data

`fgetinput.seek`: 1 means operate in seek mode (do not return data) 0 means return data

`fgetinput.calibrate`: 1 means return calibration data 0 means do not return calibration data

`fgetinput.nchan`: number of channels to read from

`fgetinput.chnames`: cell array whose elements are strings containing the channel names

`fgetoutput.tstart`: time stamp of the first point output in channel `chnames{1}`

`fgetoutput.srate`] sample rate at which data was recorded

`fgetoutput.npoint`: `npoint(i)` is the number of points returned for channel `chnames{i}`

`fgetoutput.ratios`: `ratios(i)` is the sample rate of channel `chnames{1}` divided by the sample rate of channel `chnames{i}`

`fgetoutput.discarded`: number of points discarded from channel `chnames{1}`

`fgetoutput.tfirist`: the time stamp of the first point returned in the first call to `mxFget_ch`

`fgetoutput.dt`: `tstart-tfirist`

`fgetoutput.lostlock`: time at which we lost lock (if searching for locked segments only)

`fgetoutput.lastlock`: time at which we last regained lock (if searching for locked segments only)

`fgetoutput.returnval`: 0 if unable to satisfy the request, 1 if request is satisfied by beginning new locked or continuous-in-time section, 2 if the data returned is part of an ongoing locked or continuous-in-time sequence

`fgetoutput.frinum`: three times the number of frequency values for which re are returning static calibration information.

`fgetoutput.fri`: calibration data `fri(1) = f0`, `fri(2) = r0`, `fri(3) = i0`, `fri(4) = f1`, `fri(5) = r1`, `fri(6) = i1` ... (see section 4.7)

`fgetoutput.tcalibrate`: time at which current calibration data became valid

`fgetoutput.locklow`: minimum value (inclusive) for “in-lock” in the lock channel, set only if `fgetinput.inlock` is nonzero

`fgetoutput.lockhi`: maximum value (inclusive) for “in-lock” in the lock channel, set only if `fgetinput.inlock` is nonzero

`fgetoutput.data`: a cell array. `fgetoutput.data{i}` is a vector containing the returned data for channel `chnames{i}`

Author: Steve Drasco, steve.drasco@cornell.edu

Comments: A possible fix to this function would be to write a reset function for `fget_ch`. The reset function would tell `fget_ch` that we are going to start a new data input run. Another possible fix would be to add a field `fgetinput.reset` which could reset the function.

18.2.10 Function: `mxFind_chirp`

`[output0,output90,offset,snr_max,c0,c90,var]=mxFind_chirp(htilde, ch0tilde,ch90tilde,twice_inv_noise,n0,n90,n,chirplen)`

All variables are identical to their description in section 6.20.

Author: Steve Drasco, steve.drasco@cornell.edu

Comments: None.

18.2.11 Function: `mxFreq_inject_chirp`

`[htilde]=mxFreq_inject_chirp(c0,c90,offset,invMpc,ch0tilde,ch90tilde,n)`

All variables are identical to their description in section 6.21.

Author: Steve Drasco, steve.drasco@cornell.edu

Comments: None.

18.2.12 Function: `mxGRcalibrate`

`[complex]=mxGRcalibrate(fri,frinum,num,srate,method,order)`

All variables are identical to their description in section 4.8.

Author: Steve Drasco, steve.drasco@cornell.edu

Comments: None.

18.2.13 Function: `mxGRnormalize`

`[response]=mxGRnormalize(fri,frinum,npoint,srate)`

All variables are identical to their description in section 4.10.

Author: Steve Drasco, steve.drasco@cornell.edu

Comments: None.

18.2.14 Function: `mxM_and_eta`

`[m,eta]=mxM_and_eta(tau0,tau1,Mmin,Mmax,pf)`

All variables are identical to their description in section 9.4.

Author: Steve Drasco, steve.drasco@cornell.edu

Comments: None.

18.2.15 Function: mxMake_filters

```
[ch1, ch2, filled, t_coal]=mxMake_filters(m1, m2, fstart, length, srate,  
err_cd_sprs, order)
```

All variables are identical to their description in section 6.10.

Author: Steve Drasco, steve.drasco@cornell.edu

Comments: None.

18.2.16 Function: mxMatch_cubic

```
[semimajor, semiminor, theta, mcoef, outval]=mxMatch_cubic(m1ref, m2ref,  
mathcont, order, srate, flo, ftau, noisefile)
```

All variables are identical to their description in section 9.10.

Author: Steve Drasco, steve.drasco@cornell.edu

Comments: None.

18.2.17 Function: mxOrthonormalize

```
[n0, n90, ch90tilde]=mxOrthonormalize(ch0tilde, ch90tilde,  
twice_inv_noise, n)
```

All variables are identical to their description in section 6.18.

Author: Steve Drasco, steve.drasco@cornell.edu

Comments: None.

18.2.18 Function: mxPhase_frequency

```
[Max_Freq_Actual, phase, frequency, steps_filld, clscnc_time]=mxPhase_frequency  
(m1, m2, spin1, spin2, n_phaseterms, phaseterms,  
Initial_Freq, Max_Freq_Rqst, Sample_Time, steps_alloc, err_cd_sprs)
```

All variables are identical to their description in section 6.2.

Author: Steve Drasco, steve.drasco@cornell.edu

Comments: None.

18.2.19 Function: mxSp_filters

```
[ch1, ch2, f_c]=mxSp_filters(m1, m2, fstart, n, srate, t_c, order)
```

All variables are identical to their description in section 6.12.

Author: Steve Drasco, steve.drasco@cornell.edu

Comments: None.

18.2.20 Function: mxSplitup

```
[indices]=mxSplitup(working,template,r,n,total,p)
```

All variables are identical to their description in section 6.26.

Author: Steve Drasco, steve.drasco@cornell.edu

Comments: None.

18.2.21 Function: mxSplitup_freq

```
[indices,stats,working,htilde]=mxSplitup_freq(c0,c90,chirp0,chirp90,  
norm,twice_inv_noise,n,offset,p)
```

All variables are identical to their description in section 6.27.

Author: Steve Drasco, steve.drasco@cornell.edu

Comments: None.

18.2.22 Function: mxSplitup_freq2

```
[indices,stats,working,htilde]=mxSplitup_freq2(c0,c90,chirp0,chirp90,  
norm,twice_inv_noise,n,offset,p)
```

All variables are identical to their description in section 6.28.

Author: Steve Drasco, steve.drasco@cornell.edu

Comments: None.

18.2.23 Function: mxTau_of_mass

```
[tau0,tau1]=mxTau_of_mass(m1,m2,pf)
```

All variables are identical to their description in section 9.3.

Author: Steve Drasco, steve.drasco@cornell.edu

Comments: None.

18.2.24 Function: mxTemplate_area

```
[Grid,area]=mxTemplate_area(Grid)
```

An empty scope structure can be made with the command `Grid=scope_structure`. All fields are identical to their description in section 9.5.

18.2.25 Function: mxTime_inject_chirp

```
[data,work]=mxTime_inject_chirp(c0,c90,offset,invMpc,chirp0,chirp90,  
response,data,n)
```

All variables are identical to their description in section 6.22.

Author: Steve Drasco, steve.drasco@cornell.edu

Comments: None.

18.2.26 Function: mxUrlopen

```
mxUrlopen('URL', 'filename')
```

This function gets the file found at URL and writes it to a local file filename.

Authors: Steve Drasco, steve.drasco@cornell.edu

Comments: This program is an adaptation of the program `urlopen.c` by Mark Niedengard, Roy Williams, and George Kremenek.

18.3 Examples

The code for the following examples can be found in the directory
\$GRASP/src/examples/examples_GRtoolbox.

18.3.1 Example: print_ssF

A Matlab version of the `print_ssF` example found in section 4.9.

```
function print_ssF();
% PRINT_SSF
% A Matlab version of the GRASP example: print_ssF.
% Example run: print_ssF
%
% Steve Drasco
% Summer 1998

% get some data
fgetinput.npoint = 256;
fgetinput.nchan = 1;
fgetinput.chnames = {'IFO_DMRO'};
fgetinput.inlock = 0;
fgetinput.seek = 1;
fgetinput.calibrate = 1;
fgetoutput = mxFget_ch(fgetinput);

% call mxGRcalibrate
srate = fgetoutput.srate;
npoint = 4096;
cplx=mxGRcalibrate(fgetoutput.fri, fgetoutput.frinum, npoint, srate, 2, 0);

% plot output
freq=1:npoint/2;
freq = freq*srate/npoint;
imaginary = cplx(2*(1:npoint/2));
real = cplx(2*(1:npoint/2)+1);
plot(freq, real, 'b', freq, imaginary, 'r');
```

18.3.2 Example: power_spectrumF

A Matlab version of the power_spectrumF example in section 4.11.

```
function [fgetoutput, response]=power_spectrumF()
% POWER_SPECTRUMF
% A Matlab version of the GRASP example: power_spectrumF.
% Example run: [fgetoutput, response]=power_spectrumF;
%
% Steve Drasco
% Summer 1998

npoint = 65536;
fgetinput.nchan = 1;
fgetinput.chnames = {'IFO_DMRO'};
fgetinput.inlock = 1;
fgetinput.npoint = npoint;
fgetinput.calibrate = 1;

% I won't skip any data
fgetinput.seek = 0;
fgetoutput = mxFget_ch(fgetinput);
srate = fgetoutput.srate;
data = fft(fgetoutput.data{1});
response=mxGRnormalize(fgetoutput.fri, fgetoutput.frinum, npoint, srate);

% one-sided power-spectrum normalization, to get meters/rHz
factor = sqrt(2/(srate*npoint));

% frequency
freq = (1:(npoint/2)-1)*srate/npoint;

% real and imaginary parts of tilde c0
c0_real = real( data( 2:npoint/2 ) );
c0_imag = imag( data( 2:npoint/2 ) );

% real and imaginary parts of R
res_real = response( 1 + 2*( 2:npoint/2 ) );
res_imag = response( 2 + 2*( 2:npoint/2 ) );

% real and imaginary parts of tilde dl
dl_real = c0_real.*res_real - c0_imag.*res_imag;
dl_imag = c0_real.*res_imag + c0_imag.*res_real;
spectrum = factor * sqrt(dl_real.^2 + dl_imag.^2);

% plot the results
loglog(freq, spectrum);
xlabel('frequency (Hz)');
ylabel('noise m/(Hz^1/2)');
```

18.3.3 Example: phase_evoltn

A Matlab version of the phase_evoltn example found in section 6.3.

```
function phase_evoltn()
% PHASE_EVOLUTION
% A Matlab version of the GRASP example: phase_evolution.
% Example run: phase_evolution
%
% Steve Drasco
% Summer 1998

m1 = 1.4;
m2 = 1.4;
spin1 = 0;
spin2 = 0;
n_phaseterms = 5;
Initial_Freq = 60;
Max_Freq_Rqst = 2000;
Sample_Time = 1/9868.4208984375;
err_cd_sprs = 0;
phaseterms = [1 0 1 1 1];

[Max_Freq_Actual, phase, frequency, steps_filld, ...
 clscnc_time]=mxPhase_frequency(m1,m2,spin1,spin2,n_phaseterms,phaseterms,...
 Initial_Freq,Max_Freq_Rqst,Sample_Time,[],err_cd_sprs);

time = (1:steps_filld) * Sample_Time;

subplot(2,1,1)
plot(time, phase, 'b');
xlabel('time (s)');
ylabel('phase');
subplot(2,1,2)
plot(time, frequency, 'r');
    xlabel('time (s)');
    ylabel('frequency (Hz)');
```

18.3.4 Example: filters

A Matlab version of the filters example found in section 6.7.

```
function filters()
% FILTERS
% A Matlab version of the GRASP example: filters.
% Example run: filters
%
% Steve Drasco
% Summer 1998
```

```
m1 = 1.4;
m2 = 1.4;
spin1 = 0;
spin2 = 0;
n_phaseterms = 5;
Initial_Freq = 60;
Max_Freq_Rqst = 2000;
Sample_Time = 1/9868.4208984375;
err_cd_sprs = 0;
phaseterms = [1 0 1 1 1];

[Max_Freq_Actual,h_c,h_s,steps_filld,clscnc_time]=mxChirp_filters(m1,m2,...
spin1,spin2,n_phaseterms,phaseterms,Initial_Freq,Max_Freq_Rqst,Sample_Time,...
[],err_cd_sprs);

time = (1:steps_filld) * Sample_Time;

plot(time, h_c,'b', time, h_s,'r');
xlabel('time (s)');
ylabel('h_c and h_s');
```

18.3.5 Example: area

A Matlab version of the area example found in section 9.5.

```
function [Gridout, out] = area()
% AREA
% A Matlab version of the example: area in GRASP.
% Example run: [Gridout, out] = area;
%
% Steve Drasco
% Summer 1998

Grid = scope_structure;

Grid.m_mn = 0.8;
Grid.m_mx = 50.0;
Grid.f_start = 140.0;

[Gridout, out] = mxTemplate_area(Grid);
```

18.3.6 Example: match_fit

A Matlab version of the match_fit example found in section 9.11.

```
function [semimajor,semiminor,theta,mcoef,tstp]=match_fit(m1,m2,matchcont,...
order)
% MATCH_FIT
```

```
% A Matlab version of the GRASP example: match_fit.
% Example run: [semimajor,semiminor,theta,mcoef,tstp]=match_fit(m1,m2,...
%     matchcont,order)
%
% Steve Drasco
% Summer 1998

srate = 50000;
detector_num = 15;
flo = 120;
ftau = 140;

[site_parameters,site_name,noise_file,...
whiten_file]=mxDetector_site('detectors.dat',detector_num);

[semimajor,semiminor,theta,mcoef,tstp]=mxMatch_parab(m1,m2,matchcont,...
order,srate,flo,ftau,noise_file);
if tstp
semimajor
semiminor
theta
mcoef
elseif ~tstp
[semimajor,semiminor,theta,mcoef,tstp]=mxMatch_parab(m1,m2,matchcont,...
srate,flo,ftau,noise_file);
semimajor
    semiminor
    theta
    mcoef
end
```

18.3.7 Example: readfri

Reads the swept sine calibration information from a frame and returns it in an fgetoutput structure.

```
function fgetoutput=readfri()
% READFRI
% This program reads the calibration data from a frame file.
% Example run: fgetoutput=readfri
%
% Steve Drasco
% Summer 1998

fgetinput.npoint=65536;
fgetinput.inlock=1;
fgetinput.seek=0;
fgetinput.calibrate=1;
```

```
fgetinput.nchan=1;
fgetinput.chnames={'IFO_DMRO'};

%fgetinput.chnames={'IFO_DMRO','IFO_DCDM'};
%fgetinput.nchan=2;

fgetoutput=mxFget_ch(fgetinput);
```

18.3.8 Example: oneFget

Calls the GRASP function `fget_ch` once to read data from a file. You can edit the comments to retrieve data from one or two channels.

```
function fgetoutput=oneFget()
% ONEFGET
% This Program uses the GRASP function fget_ch() once and returns the output
%
% Steve Drasco
% Summer 1998

fgetinput.npoint=296000;
fgetinput.inlock=1;
fgetinput.seek=0;
fgetinput.calibrate=1;

%fgetinput.nchan=1;
%fgetinput.chnames={'IFO_DMRO'}

fgetinput.chnames={'IFO_DMRO','IFO_DCDM'};
fgetinput.nchan=2;

fgetoutput=mxFget_ch(fgetinput);
```

18.3.9 Example: twoFget

This example works just like `oneFget` except that it uses the GRASP function `fget_ch` twice.

```
function fgetoutput=twoFget()
% TWOFGET
% This program uses the GRASP function fget_ch() twice and returns the output.
%
% Steve Drasco
% Summer 1998

fgetinput.npoint = 296000;
fgetinput.inlock = 1;
fgetinput.seek = 0;
fgetinput.calibrate = 1;
```

```
% uncomment these to get one channel
fgetinput.nchan=1;
fgetinput.chnames={'IFO_DMRO'}

% uncomment these to get two channels
%fgetinput.chnames={'IFO_DMRO','IFO_DCDM'};
%fgetinput.nchan=2;

for i = 1:2
fgetoutput = mxFget_ch(fgetinput);
end
```

19 REFERENCES

- [1] W.H. Press, B.P. Flannery, S.A. Teukolsky and W.T. Vetterling, *Numerical recipes in C: the art of scientific computing*, 2nd edition, Cambridge University Press. 13, 16, 47, 48, 49, 49, 86, 86, 87, 87, 164, 189, 226, 516, 516, 516, 516
- [2] Message Passing Interface Forum, *MPI: a message passing interface standard*, International Journal of Supercomputer Applications **8** 3/4 1994. 13
- [3] W. Gropp and E. Lusk, *User's Guide for mpich, a portable implementation of MPI*, Technical Report ANL/MCS-TM-ANL-96/6, Argonne National Laboratory. 13, 17
- [4] R. Balasubramanian, B.S. Sathyaprakash, and S.V. Dhurandhar, *Gravitational waves from coalescing binaries: detection strategies and monte carlo estimation of parameters*, Phys. Rev. **D53** (1996) 3033-3055; Erratum in Phys. Rev. **D54** (1996) 1860. Originally posted as gr-qc/9508011. 280
- [5] B.J. Owen, *Search templates for gravitational waves from inspiraling binaries: choice of template spacing*, Phys. Rev. **D53** (1996) 6749-6761. Originally posted as gr-qc/9511032. 266, 266, 280, 288, 309, 309
- [6] B.J. Owen and B.J. Sathyaprakash, *Matched filtering of gravitational waves from inspiraling compact binaries: Computational cost and template placement*, gr-qc/9808076. 373
- [7] L. Blanchet, B.R. Iyer, C.M. Will, and A.G. Wiseman, *Gravitational waveforms from inspiralling compact binaries to second-post-Newtonian order*, Class. Quantum Grav. **13**, 575-584 (1996). 131, 137, 578
- [8] C.M. Will and A.G. Wiseman, *Gravitational radiation from compact binary systems: Gravitational waveforms and energy loss to second post-Newtonian order*, Phys. Rev. **D54** (1996) 4813-4848. 131, 141, 141, 146, 578
- [9] L. Blanchet, Phys. Rev. **D54**, 1417 (1996). 131, 149, 152
- [10] C. Cutler *et al.*, *The Last Three Minutes: Issues in Gravitational-Wave Measurements of Coalescing Binaries*, Phys. Rev. Lett. **70** (1993) 2984-2987. 131
- [11] C.W. Lincoln and C.M. Will, *Coalescing binary systems of compact objects to (post)^{5/2}-Newtonian order: Late-time evolution and gravitational-radiation emission*, Phys. Rev. **D42** (1990) 1123-1143. 131
- [12] L. Blanchet, T. Damour, B.R. Iyer, C.M. Will, and A.G. Wiseman, *Gravitational-Radiation Damping of Compact Binary Systems to Second Post-Newtonian Order*, Phys. Rev. Lett. **74** (1995) 3515-3518. 137, 147
- [13] E. Flanagan and S. Hughes, *Measuring gravitational waves from binary black hole coalescences. I. Signal to noise for inspiral, merger, and ringdown*, Phys. Rev. **D57**, 4535-4565, (1998). 204, 206, 207
- [14] E. Flanagan and S. Hughes, *Measuring gravitational waves from binary black hole coalescences. II. The waves' information and its extraction, with and without templates*. Phys. Rev. **D57**, 4566-4587, (1998).

- [15] L. S. Finn and D. Chernoff, *Observing binary inspiral in gravitational radiation: one interferometer*. Phys. Rev. **D47**, 2198-2219, (1993). 204, 204, 373, 597
- [16] K. S. Thorne, Private communication to Scott Hughes. 204
- [17] Thorne's estimate of 1.04 is off by 6% because it uses a fit to the probability distribution $P(\Theta^2 > x^2)$ (defined in [15]) which is not accurate enough to reliably compute the x^2 moment of that distribution. 204
- [18] D. Marković, *Possibility of determining cosmological parameters from measurements of gravitational waves emitted by coalescing compact binaries*, Phys. Rev. **D48** 4738-4756 (1993). 206
- [19] D.W. Hogg, notes, *Manual on Cosmological Distance Measures*, available from <http://www.sns.ias.edu/~hogg>. 206
- [20] E.W. Kolb and M.S. Turner, *The Early Universe*, published by Addison Wesley, 1990. 206
- [21] C. Cutler and É.E. Flanagan *Gravitational waves from merging compact binaries: How accurately can one extract the binary's parameters from the inspiral waveform?* Phys. Rev. **D49** 2658–2697 (1994). 152, 159, 266, 373
- [22] S.Droz, D.J. Knapp, E. Poisson, B.J. Owen available at <http://xxx.lanl.gov/abs/gr-qc/9901076>. 152
- [23] E. Poisson and C. M. Will, Phys. Rev. **D52**, 848 (1995). 152
- [24] J. Mathews and R.L. Walker, *Mathematical Methods of Physics*, Second Edition, Addison-Wesley, 1970. 182
- [25] F. Echeverria *Gravitational-wave measurements of the mass and angular momentum of a black hole*, Phys. Rev. **D40** 3194–3203 (1989). 254, 265
- [26] J.N. Goldberg, A.J. Macfarlane, E.T. Newman, F. Rohrlich, and E.C.G. Sudarshan *Spin-s spherical harmonics and $\bar{\partial}$* , J. Math. Phys. **8** 2155–2161 (1967). 212, 244, 245, 245, 245, 245
- [27] E.W. Leaver *An analytic representation for the quasi-normal modes of Kerr black holes*, Proc. Roy. Soc. Lond. **A402** (1985). 241, 244
- [28] H. Onozawa *Detailed study of quasinormal frequencies of the Kerr black hole*, Phys. Rev. **D55** 3593–3602 (1997). 242, 242
- [29] W.H. Press and S.A. Teukolsky (Oct 1973) *Perturbations of a rotating black hole. II. Dynamical stability of the Kerr metric*, Astrophys. J. **185** 649–673 (1973). 244
- [30] S.A. Teukolsky *Perturbations of a rotating black hole. I. Fundamental equations for gravitational, electromagnetic, and neutrino-field perturbations*, Astrophys. J. **185** 635–647 (1973). 211, 239, 239, 239, 241
- [31] A.D. Gillespie, *Thermal Noise in the Initial LIGO Interferometers*, Caltech PhD thesis, 1995. 30
- [32] T.T. Lyons, *An optically recombined laser interferometer for gravitational wave detection*, Caltech PhD thesis, 1997. 30, 31, 33, 33

- [33] Warren G. Anderson and R. Balasubramanian, *Time-Frequency Detection of Gravitational Waves*, Submitted for publication, <http://xxx.lanl.gov/abs/gr-qc/9905023/>. 347, 349, 349, 365, 365
- [34] C. Steger, computer code DETECT-LINES 1.2, Technische Universitat München, München Germany, 1996, available from <ftp://ftp9.informatik.tu-muenchen.de/pub/detect-lines/detect-lines-1.2.tar.gz>. 349, 355, 355
- [35] C. Steger, IEEE Transactions on Pattern Analysis and Machine Intelligence, **20**, 113, (1998). 349, 349, 355, 355
- [36] B. Allen, *The stochastic gravity-wave background: sources and detection*, in Relativistic Gravitation and Gravitational Radiation, Proceedings of the Les Houches School on Astrophysical Sources of Gravitational Radiation, eds. J-A. Marck and J-P. Lasota, (Cambridge University Press, Cambridge, England, 1997) pages 373-417. 374, 378, 385, 402, 409
- [37] See equations (4.13) and (4.14) in *Spacecraft attitude determination and control*, Ed. James R. Wertz, (D. Reidel Publishing Co., Boston, 1985). 372
- [38] A. Abramovici et al., *Science* **256**, 325 (1992). 373, 373, 373, 373, 373
- [39] D.J. Thomson, *Spectrum estimation and harmonic analysis*, Proceedings of the IEEE, **70**, 1055-96, (1982). 55, 94, 544
- [40] D.B. Percival and A.T. Walden, *Spectral analysis for physical applications*, first edition, Cambridge University Press, (1993). 55, 94, 544, 544, 544, 544, 544, 545
- [41] J.M. Lees and J. Park, *Multiple-taper spectral analysis: A stand-alone C subroutine*, Computers and Geology **21**, 199-236. 544
- [42] D. Lai and S. L. Shapiro, *Astrophys. J.* **442**, 259-272, (1995). 501
- [43] S. Chandrasekhar, *Ellipsoidal Figures of Equilibrium*, (Yale University Press, New Haven, 1969). 501
- [44] Dong Lai, Personal Communication. 501
- [45] The data files for the amplitudes $A_{lm}(v)$ and the luminosity function $P(v)$ were kindly provided by Eric Poisson (poisson@physics.uoguelph.ca). 211, 211
- [46] S. Droz and E. Poisson, *Gravitational waves from inspiraling compact binaries: Second post Newtonian waveforms as search templates*, Phys. Rev. **D57**, 4449 (1997). 211
- [47] Benoit Mours, *Frame Library User's Manual*, version 3.60, <http://www.lapp.in2p3.fr/virgo/FrameL/>, email: mours@lapp.in2p3.fr, VIR-MAN-LAP-5400-103. 582
- [48] B. Allen, K. Blackburn, P. Brady, J. Creighton, T. Creighton, S. Droz, A. Gillespie, S. Hughes, S. Kawamura, T. Lyons, J. Mason, B.J. Owen, F. Raab, M. Regehr, B. Sathyaprakash, R.L. Savage, Jr., S. Whitcomb, A.G. Wiseman, *Observational limit on gravitational waves from binary neutron stars in the Galaxy*, Phys. Rev. Lett. **83**, 1498 (1999). Also available from: <http://xxx.lanl.gov/abs/gr-qc/9903108>. 442
- [49] Anninos, Brandt, and Walker, *New orthogonal body-fitting coordinates for colliding black-hole spacetimes*, Phys. Rev. D **57** 6158-6167 (1998). 268, 268, 268, 268, 268, 269