LIGO-T970256-00-D

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

June 2, 1997

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.ligo.caltech.edu/LIGO_web/Collaboration/manual.pdf`, or at `http://www.ligo.caltech.edu/LIGO_web/Collaboration/lsc_interm.html`. The software package is available on request.

GRASP RELEASE 1.4.1
manual version 1.4.0

[*]`ballen@dirac.phys.uwm.edu`

Contents

1 Acknowlegements

This work has been partially supported by National Science Foundation grants to: the University of Wisconsin - Milwaukee PHY95-07740, the LIGO project PHY92-10038 and the LIGO visitors program PHY96-03177. 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. 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 Interferometric Gravitational Observatory (LIGO) project in the United States, and contains a collection of software tools for this purpose.

In order that this package 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 with only one condition: GRASP and its author must be acknowledged or referenced in any work or publications to which GRASP made a contribution. This citation must specify the *version number* (for example, 1.0.0) of GRASP. In addition, if the code has been modified in any way, this must also be stated. While the GRASP package is available in the public domain, we do intend to regulate its distribution. You may request a copy of GRASP for your personal use, or for use at your own institution, but you must not distribute it outside that group. In addition, one person at each institution must 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 "cookbook" 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 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/19nov94.3frame
```

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 `gwouput` from that line (but leave a space after the colon). Then type:

```
make gwouput
```

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.

2.3 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 10.15).

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://lapphp.in2p3.fr/virgo/FrameL`. Contact Benoit Mours `mours@lapp.in2p3.fr` for further information.

2.4 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 2.6 of the routines from "*Numerical Recipes in C: the art of scientific computing*" [1]. 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://cfata2.harvard.edu/numerical-recipes/. These routines are extremely useful and beautifully-documented; if you don't already have them available for your use, you should!

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).

2.5 GRASP Installation

As we have just explained, GRASP requires access to *Numerical Recipes in C* libraries and to MPI and MPE 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.

2.5.1 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:

`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.

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

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

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

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

`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.

`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.
- `inspiral/` Binary inspiral analysis (including optimal filtering and vetoing).
- `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.
- `utility/` General purpose utility routines.
- `examples/` The source code for all of the examples given in this manual (organized by section).

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

`lib/` Object libraries.

`bin/` Executable applications and programs.

2.5.2 Accessing *Numerical Recipes in C* libraries

GRASP makes use of many of the functions and subroutines from *Numerical Recipes in C* [1]. 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 simplify 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. We suggest that you 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 occurence of `select(` to `NRselect(`. You will have to do this in (respectively) three 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 elminate the `select()` routine from the *Numerical Recipes* library, replacing it with a routine called `NRselect()`, and eliminating any possible naming conflict from the library.

2.5.3 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 given above, 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.

2.5.4 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 2.3). 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://lapphp.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.

2.5.5 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.

2.5.6 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.

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_LIB=-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 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.

10. 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.

11. 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 containig 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 (#).

12. 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!)

13. 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 three 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.
    - 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://theory.lcs.mit.edu/~fftw`
      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*.

    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.

14. Now, in the top level GRASP directory, execute the shell script `InstallGRASP`, by typing the commands:
    ```
    chmod +x InstallGRASP
    ./InstallGRASP
    ```
    From here on, the remainder of the installation should proceed automatically. The `InstallGRASP` script takes information contained in the `SiteSpecific` file, uses it to create `Makefile`'s in each `src` subdirectory, and runs `make` in each of those directories.

The `Makefile` in each directory is constructed by concatenating the file `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. 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 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 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.

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

There is a good archive of data from the Caltech 40-meter prototype interferometer. Although the interferometer is only sensitive enough to detect events like binary inspiral within $\approx$ 10kpc (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 [18] and Lyons [19].

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 preferrable 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

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			$1024 \times n$	1024			$1024 \times n$	···

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

the table). This is 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 [19], 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` $\rightarrow$ `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 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.

block 0				block 1				block 2				block 3				···
mh	bh	0's	data	mh	bh	0's	data	mh	bh	0's	data	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 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 millseconds which should be added to the previous quantity.

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

`int tod_minute`: Minutes after hour 0-59.

`int tod_hour`: Hour since midnight 0-23.

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

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

`int date_year`: Years since 1900.

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

`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.

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 [19] 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 ss:timestamp).

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 [19] 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.

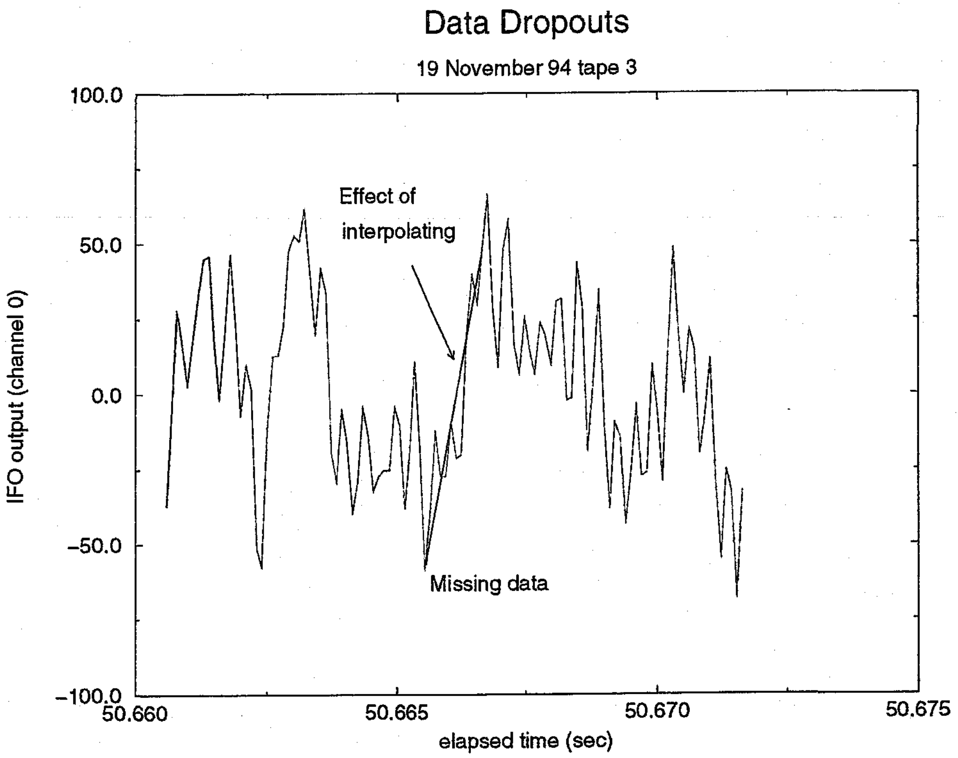


Figure 1: This shows the appearance of `channel.0` before and after the `extract` program was repaired (on 14 November 1996) to correctly extract data from the Exabyte data tapes. The old version of `extract` dropped the ten data points directly above the words "missing data"; in effect these were interpolated 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 10.1. 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, 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");

   /* 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_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.

`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.

`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.

`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).

`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, Bruce Allen */
#include "grasp.h"

int main() {
   float tstart,tend,srate,totaltime,begin,end;
   int start_offset,start_block,end_offset,end_block,points;
   FILE *fplock;

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

   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;
      totaltime=end-begin;

      /* print out info for lock intervals > 30 seconds */
      if (totaltime>30.0) {
         printf("Locked from t = %f to %f for %f sec\n",begin,end,totaltime);
         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.

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, Bruce Allen */
#include "grasp.h"

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");
   fplock=grasp_open("GRASP_DATAPATH","channel.10");

   /* 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   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.

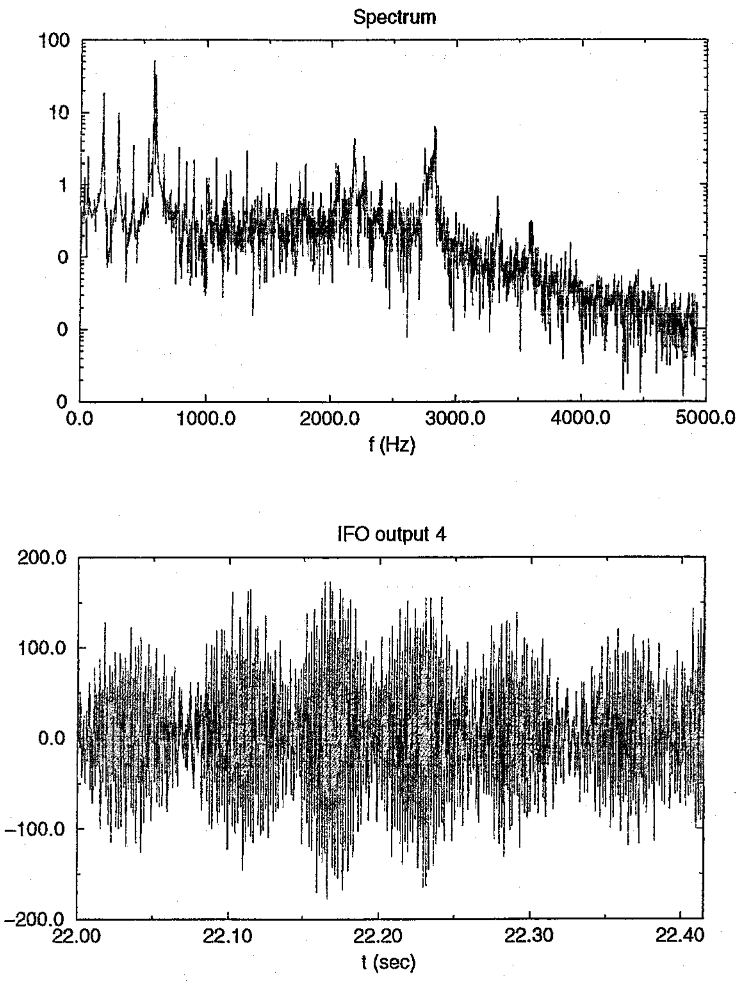


Figure 2: Snapshot of output from `animate`. This shows the (whitened) CIT 40-meter IFO a few seconds after acquiring lock, before the violin modes have damped down

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,

```
        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.

in order for the frequency scale of the Fourier transform to be correct. The corrected version of xmgr is shown in Figure 3.

```
/* GRASP: Copyright 1997, 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;
   int remain,i,seq=0,code,npoint=4096,seek;
   FILE *fp,*fplock;
   short *data;

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

     /* 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:

$$\begin{array}{ccc} f_1 & r_1 & i_1 \\ f_2 & r_2 & i_2 \\ & \cdots & \\ f_m & r_m & i_m \end{array}$$

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. 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:

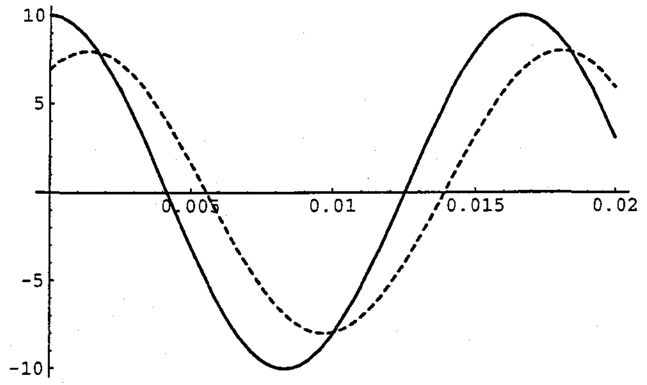


Figure 4: This shows a driving voltage $V_{\rm coil}$ (solid curve) and the response voltage $V_{\rm IFO}$ (dotted curve) as functions of time (in sec). Both are 60 Hz sinusoids; the relative amplitude and phase of the in-phase and out-of-phase components of $V_{\rm IFO}$ are contained in the swept-sine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

```
                 ...
60.000         0.6930        -0.40000
                 ...
```

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 $S(60) = 0.8 \ {\rm e}^{-i\pi/6} = 0.693 - 0.400i$. 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: $S(-60) = S^*(60) = 0.8 \ {\rm e}^{i\pi/6} = 0.693 + 0.400i$.

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 `channel.0` files to reconstruct the differential motion $\Delta l(t)$ (in meters) of the interferometer arms? Here we address the closely related question: given $V_{\rm IFO}$, how do we reconstruct $V_{\rm coil}$? 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

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

The inverse Fourier transform is

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

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

$$\tilde V_{\rm coil}(60) = 5 \quad \text{and} \quad \tilde V_{\rm coil}(-60) = 5, \tag{3.9.5}$$

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

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

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

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

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

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

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 a 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 $i = 0, \cdots, N/2$.

`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, 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");

        /* 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\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;
}
```

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[2`i`]` and `response[2`i`+1]` contain the real and imaginary parts of $R(f)$ at frequency $f = i$`srate`$/N$. 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

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

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

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

We have substituted in equation (3.9.8) which relates $\tilde{V}_{\text{IFO}}$ and $\tilde{V}_{\text{coil}}$. 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 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 `channel.0`, the swept sine file, and the quantities given in Table 4 by

$$\widetilde{\Delta l} = R(f) \times \widetilde{C_0} \qquad \text{with} \quad R(f) = \frac{Q \times \text{ADC}}{-4\pi^2 f^2 S^*(f)}, \tag{3.12.3}$$

where the $\tilde{}$ 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

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	$\mathrm{V_{IFO}}$ (ADC counts)$^{-1}$
Swept sine calibration	S(f)	from file	$\mathrm{V_{IFO}}\,(\mathrm{V_{coil}})^{-1}$
Calibration constant	Q	1.428×10^{-4}	meter $\mathrm{Hz}^2\,(\mathrm{V_{coil}})^{-1}$

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

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

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 FFT[Δl], 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

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

   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.

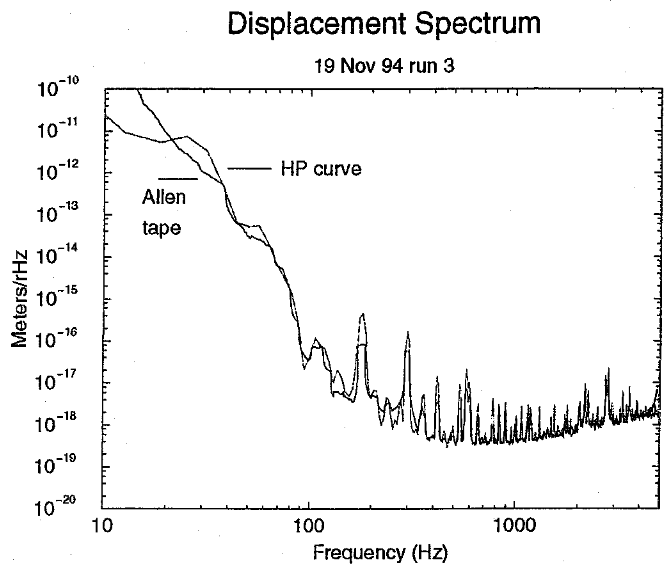


Figure 5: An example of a power spectrum curve produced with `power_spectrum`. The spectrum produced off a data tape (with 100 point smoothing) is compared to that produced by the HP spectrum analyzer in the lab.

```
/* GRASP: Copyright 1997, Bruce Allen */
#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");
   fplock=grasp_open("GRASP_DATAPATH","channel.10");
   fpss=grasp_open("GRASP_DATAPATH","swept-sine.ascii");

   /* 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. Thompson [23], or the textbook by Percival and Walden [24]). 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 implemen them, may be found in Section 10.16.

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`).

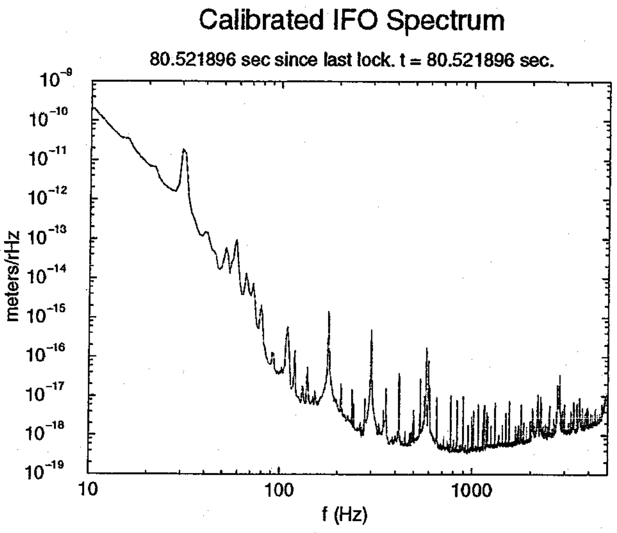


Figure 6: This shows a snapshot of the output from the program `calibrate` which displays an animated average power spectrum (Welch windowed, 30-second decay time).

```
/* GRASP: Copyright 1997, 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],response[2*NPOINT+4];
   float spec,decaytime;
   float srate,tstart=0,freq,tlock;
   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");
    fplock=grasp_open("GRASP_DATAPATH","channel.10");
    fpss=grasp_open("GRASP_DATAPATH","swept-sine.ascii");

    /* 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,&srate,0))) {
         /* put data into floats */
       for (i=0;i<npoint;i++) data[i]=datas[i];
       /* get the normalization */
       if (!pass++)
          normalize_gw(fpss,2*npoint,srate,response);
       /* Reset if just locked */
       if (code==1) {
          reset=0;
          tlock=tstart;
       }
       /* track average power spectrum, with Welch windowing. */
       avg_spec(data,average,npoint,&reset,srate,decaytime,2);
       /* loop over all frequencies except DC (j=0) & Nyquist (j=npoint/2) */
       for (j=1;j<npoint;j++) {
          /* subscripts of real, imaginary parts */
          ii=(ir=j+j)+1;
          /* frequency of the point */
          freq=0.5*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 \"%.2f sec since last lock. t = %.2f sec.\"\n",time-tlock,time);
        if (!last) printf("@kill s0\n");        /* kill graph; ready to read agai */
    }
    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: `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/examples_40meter` 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 $\Delta S(f) >$ `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 $\Delta S(f) <$ `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 7 and 8. 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).

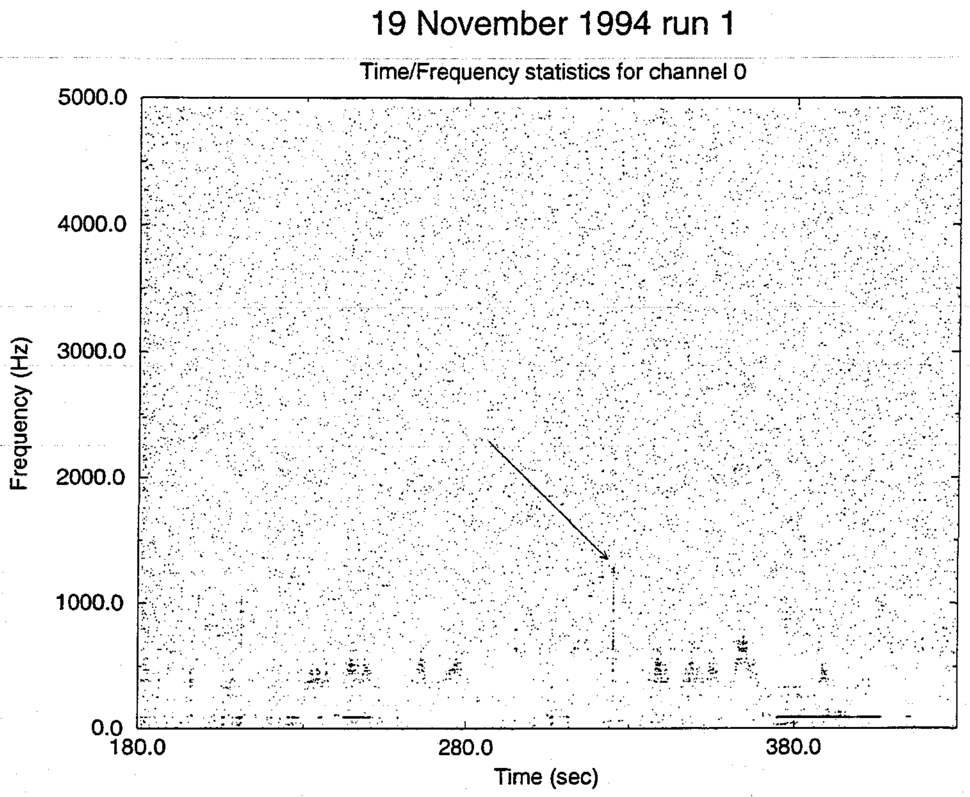


Figure 7: A time-frequency diagnostic graph produced by `diag`. The vertical line pointed to by the arrow is a non-stationary noise event in the IFO output, 325 seconds into the locked section. It sounds like a "drip" and might be due to off-axis modes in the interferometer optical cavities.

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 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()`).

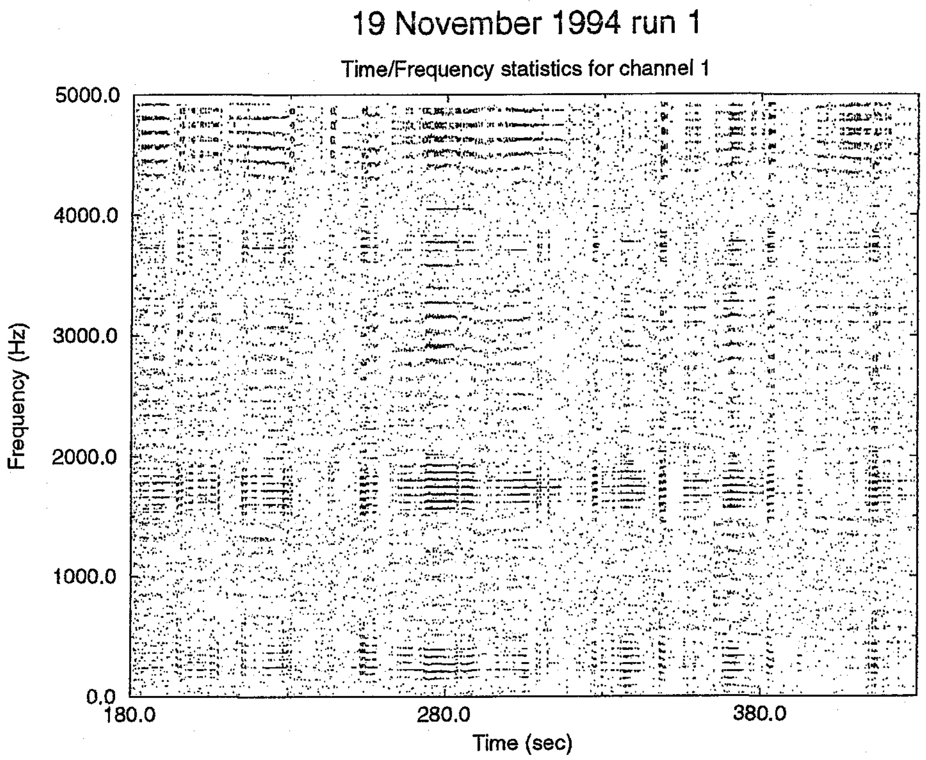


Figure 8: A time-frequency diagnostic graph produced by `diag`. This shows the identical period as the previous graph, but for the magnetometer output. Notice that the spurious event was not caused by magnetic field fluctuations.

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://lapphp.in2p3.fr/virgo/FrameL`. 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 10.15) 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 recomend 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 or 2.30 of the FRAME library. Only two files in the GRASP package (`src/utility/frameinterface.c` and `src/examples/examples_utility/transla` depend upon the version of the FRAME library. We distribute GRASP with versions of these files appropriate for different releases (currently 2.20, 2.30, and 2.33) of the FRAME library. The version 2.30 FRAME library data format is compatible with versions 2.30 and 2.33 of the FRAME library.

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 will attempt 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 $5.06666\cdots$ 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	PSL_SPZT_V
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	IFO_SPZT
15	arm 1 coil driver	SUS_EE_Coil_V	arm 1 coil driver	SUS_EE_Coil_V

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 μsec. Ideally, the time-stamps of the individual blocks should be recorded with a precision which is substantially greater than this, i.e. a few μsec at the most. However the November 1994 time stamps are recorded in two ways: as an integer number of seconds and msec (with 1000 μsec resolution) and as a floating point elapsed time. This latter quantity has a resolution of less than one μsec at early times, but a resolution of about 2000 μsec 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 $76/15 = 5.0666666\cdots$ 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 precison 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 anaysis 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 inlock;
            int seek;
            int calibrate;
};
```

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.

`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. The name of 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.)

`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.)

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 srate;
          int *npoint;
          int *ratios;
          int discarded;
          double tfirst;
          double dt;
          double lostlock;
          double lastlock;
          int returnval;
          int frinum;
          float *fri;
          int tcalibrate;
          int locklow;
          int lockhi;
};
```

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.

`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()`.

`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).

`lastlock`: Output. The time at which we last regained lock (if searching only for locked segments).

`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.

```
        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 9.

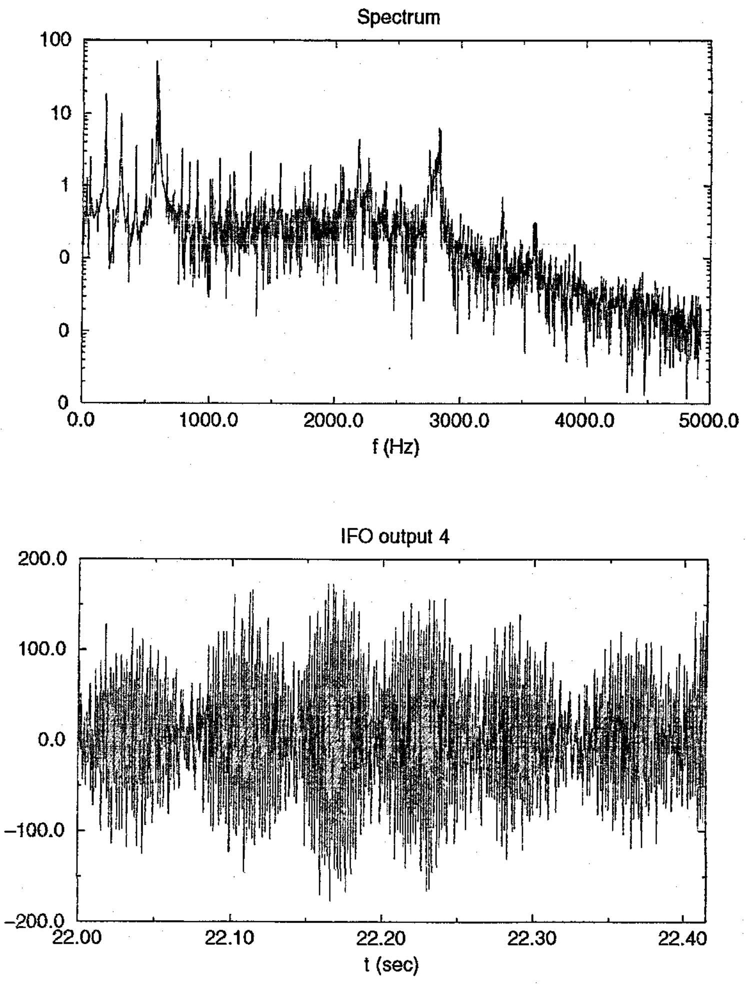


Figure 9: Snapshot of output from `animate`. This shows the (whitened) CIT 40-meter IFO a few seconds after acquiring lock, before the violin modes have damped down

After compilation, to run the program type:

```
    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

```
        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 10: 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.

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.

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 10.

```
/* GRASP: Copyright 1997, 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;
   int i,seq=0,code,npoint=4096;
   short *data;
   struct fgetinput fgetinput;
   struct fgetoutput fgetoutput;

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

   /* 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_REALTIME")) {
    /* 40 meter lab */
    fgetinput.chnames[0]=getenv("GRASP_REALTIME");
    fgetinput.inlock=0;
}
else {
    /* Nov 1994 data set */
    fgetinput.inlock=1;
}

/* 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));
            }
            /* 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) {
   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;
}
```

4.7 Swept-sine calibration information

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

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

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

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

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)

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

This is shown in Figure 11. An electrical engineer would describe this situation by saying that the phase of the response $V_{\rm IFO}$ is lagging the phase of the driving signal $V_{\rm coil}$ by 30°. The corresponding line in the swept sine calibration file would read:

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

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.

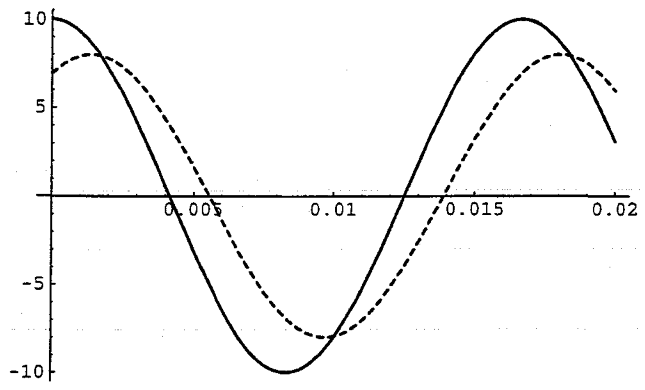


Figure 11: This shows a driving voltage $V_{\rm coil}$ (solid curve) and the response voltage $V_{\rm IFO}$ (dotted curve) as functions of time (in sec). Both are 60 Hz sinusoids; the relative amplitude and phase of the in-phase and out-of-phase components of $V_{\rm IFO}$ are contained in the swept-sine calibration files.

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 $V_{\rm IFO}$, how do we reconstruct $V_{\rm coil}$? 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

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

The inverse Fourier transform is

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

With these conventions, the signals (4.7.1) and (4.7.2) shown in in Figure 11 have Fourier components:

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

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

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

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

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

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

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

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 a 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]`=f_0, `fri[1]`=r_0, `fri[2]`=i_0, `fri[3]`=f_1, `fri[4]`=r_1, `fri[5]`=i_1,... 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 $f_{\rm Nyquist}$. See for example equation (12.1.5) of reference [1]. The frequencies are $f_i = \frac{i}{N} F_{\rm sample}$ for $i = 0, \cdots, N/2$.

`srate`: Input. The sample rate $F_{\rm sample}$ (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 $F_{\rm sample}/2$.

`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, 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;
}
```

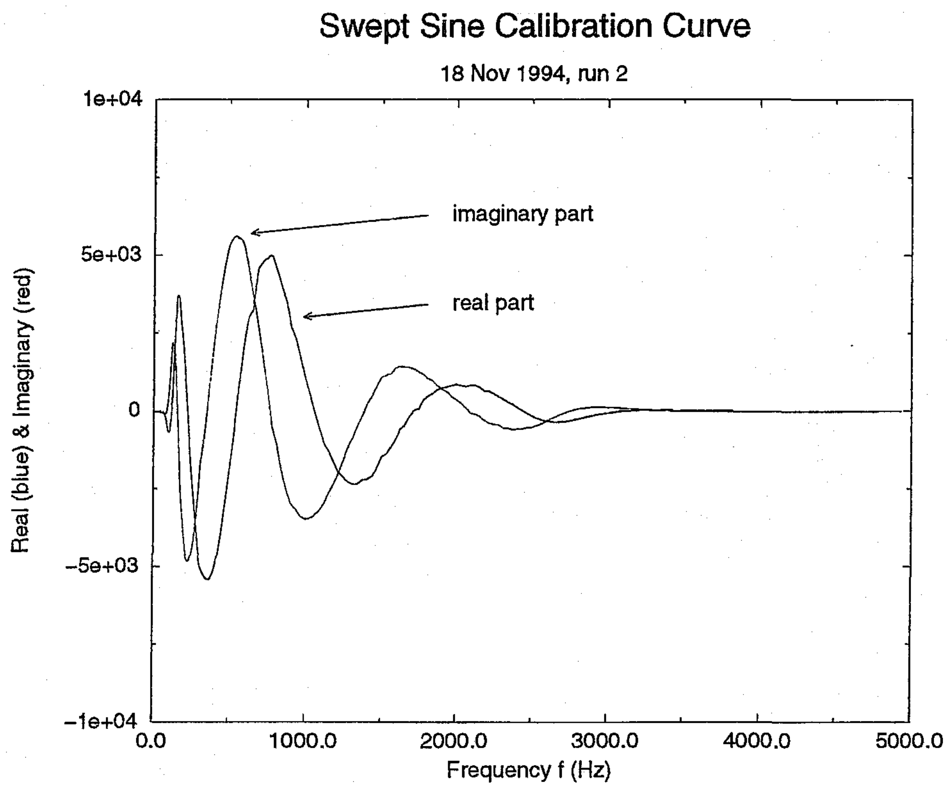


Figure 12: A swept sine calibration curve, showing the real and imaginary parts, produced by the example program `print_ssF`.

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: $\widetilde{\Delta l}(f) = R(f)\widetilde{C_{\rm IFO}}(f)$. 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 `fri[0]`$=f_0$, `fri[1]`$=r_0$, `fri[2]`$=i_0$, `fri[3]`$=f_1$, `fri[4]`$=r_1$, `fri[5]`$=i_1$,... 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!

`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 `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[2`i`]` and `response[2`i`+1]` contain the real and imaginary parts of $R(f)$ at frequency $f = i$`srate`$/N$. 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

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

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

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

We have substituted in equation (4.7.8) which relates $\tilde{V}_{\rm IFO}$ and $\tilde{V}_{\rm coil}$. The IFO voltage is directly proportional to the quantity recorded in the IFO output channel: $V_{\rm IFO} = \text{ADC} \times C_{\rm IFO}$, 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

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

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

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

where the $\tilde{}$ 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 $C_{\rm IFO}$). 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

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

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 -nxy 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 $\widetilde{\Delta l}(f) = R(f)\widetilde{C_{\rm IFO}}(f)$.
2. The fast Fourier transform of Δl, which we denote 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 $\widetilde{\Delta l} = \Delta t$ 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
$$P = \sqrt{\frac{2}{N\Delta t}}\,\Delta t\,|\text{FFT}[\Delta l]| = \sqrt{\frac{2\Delta t}{N}}\,|\text{FFT}[\Delta l]|. \qquad (4.11.1)$$
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 13.

```
/* GRASP: Copyright 1997, 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. Thompson [23], or the textbook by Percival and Walden [24]). 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 implemen them, may be found in Section 10.16.

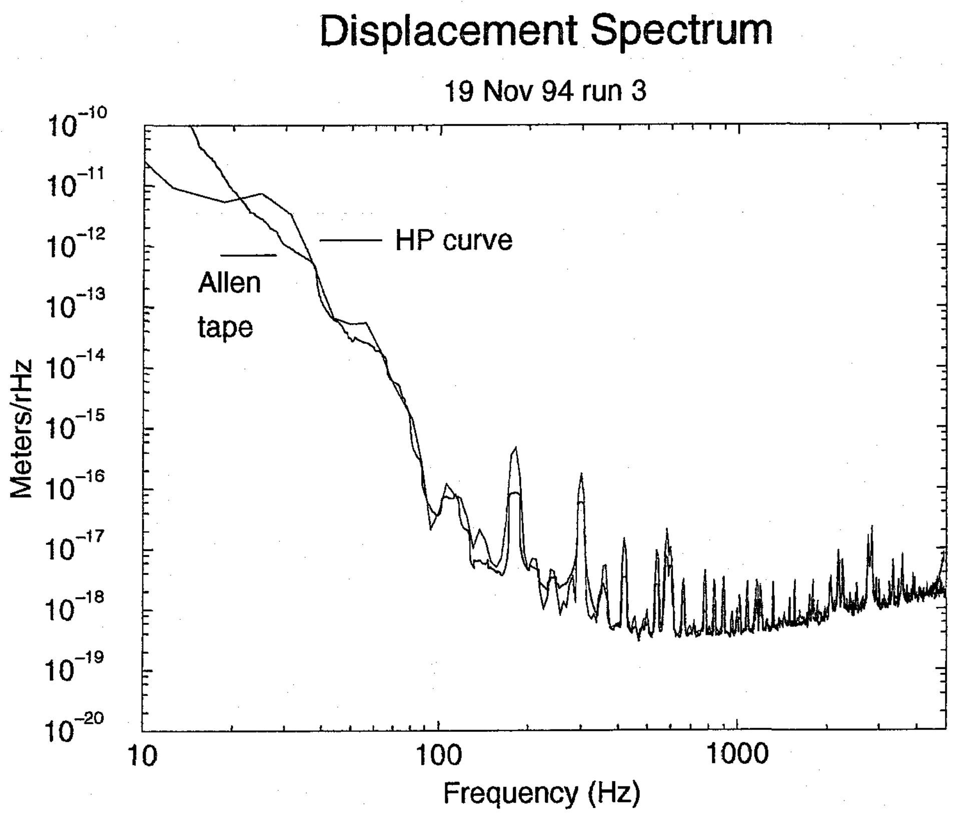


Figure 13: An example of a power spectrum curve produced with `power_spectrumF`. The spectrum produced off a data tape (with 100 point smoothing) is compared to that produced by the HP spectrum analyzer in the lab.

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 14. 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`).

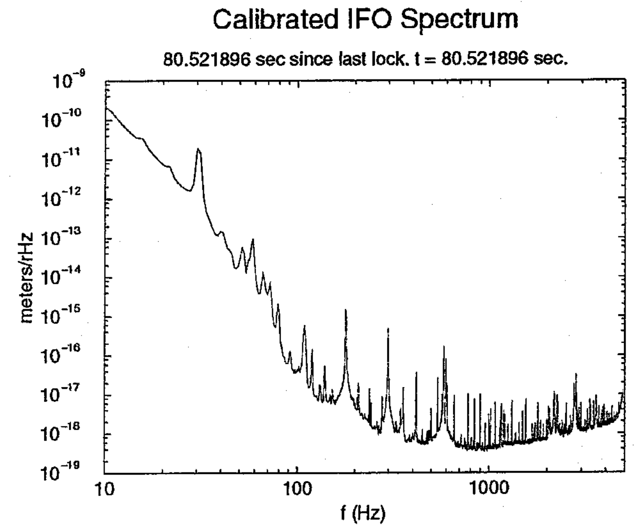


Figure 14: This shows a snapshot of the output from the program `calibrateF` which displays an animated average power spectrum (Welch windowed, 30-second decay time).

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

int main() {
   void graphout(int,float,float);
   float data[NPOINT],average[NPOINT],response[2*NPOINT+4];
   float spec,decaytime;
   float srate,tstart=0,freq,tlock;
   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,2*npoint,srate,response);

    /* Reset if just locked */
    if (code==1) {
        reset=0;
```

```
            tlock=tstart;
        }

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

        /* loop over all frequencies except DC (j=0) & Nyquist (j=npoint/2) */
        for (j=1;j<npoint;j++) {
            /* subscripts of real, imaginary parts */
            ii=(ir=j+j)+1;
            /* frequency of the point */
            freq=0.5*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 \"%.2f sec since last lock. t = %.2f sec.\"\n",time-tlock,time);
        if (!last) printf("@kill s0\n");        /* kill graph; ready to read agai */
    }
    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_spectrumF` example program.

4.13 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 $\Delta S(f) >$ `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 $\Delta S(f) <$ `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 15 and 16. By specifying several different channels on the command line for starting

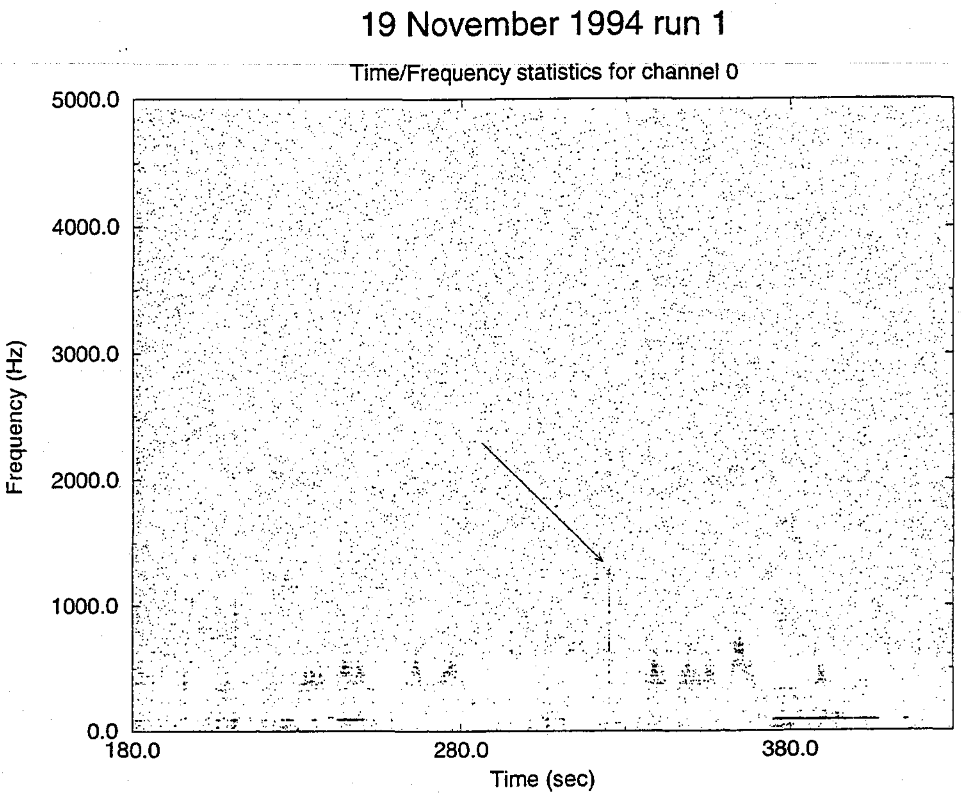


Figure 15: A time-frequency diagnostic graph produced by `diag`. The vertical line pointed to by the arrow is a non-stationary noise event in the IFO output, 325 seconds into the locked section. It sounds like a "drip" and might be due to off-axis modes in the interferometer optical cavities.

`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 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()`).

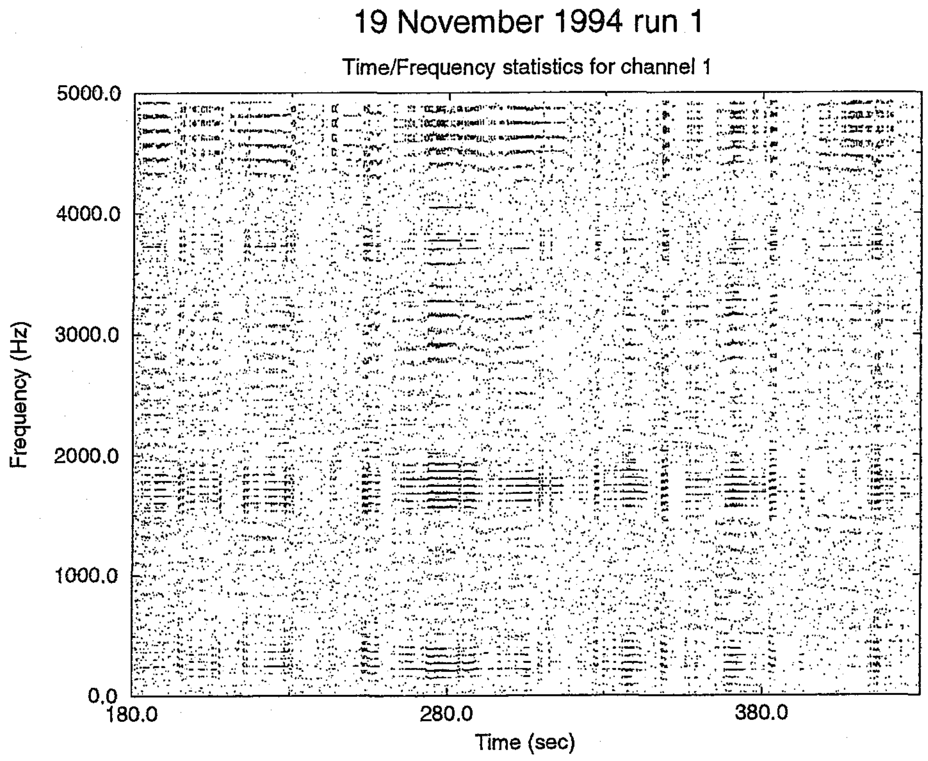


Figure 16: A time-frequency diagnostic graph produced by `diag`. This shows the identical period as the previous graph, but for the magnetometer output. Notice that the spurious event was not caused by magnetic field fluctuations.

5 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

$$M_\odot = 1.989 \times 10^{33}\,\text{grams} \tag{5.0.1}$$

$$T_\odot = \left(\frac{G}{c^3}\right) M_\odot = 4.89128 \times 10^{-6}\,\text{sec.} \tag{5.0.2}$$

GRASP functions typically measure masses in units of $M_\odot$ and times in units of seconds.

5.1 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 [6] and augmented by the spin-orbit and spin-spin corrections presented in [7]. 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.

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 [8, 9] 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 $GM_{\rm tot}/(c^2 r_{orbit})$] used in the derivation are comparable to unity. In post^2-Newtonian calculations higher orders such as post^3-Newtonian terms have been discarded. Because of this truncation, quantities that are 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. The routine `strain()` returns the full second post-Newtonian gravitational wave strain. This can be used for plotting and examining the expected waveform of a given coalescing binary, or to add a "realistic" signal into detector noise. The strain output contains all the (sub)harmonic structure and its amplitude reflects the true astrophysical distance to the source.

5.2 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 inspiralling 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` nullifys 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.(5.4.1) and (5.4.2).

err_cd_sprs: Input. Error code supression. 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 approximantion 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 occured; 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.

5.3 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, Bruce Allen */
#include "grasp.h"
int main() {
   float m1,m2,spin1,spin2,phaseterms[5],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 */

   /* 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 tail correction */
   phaseterms[4] =1.;           /* The 2PN 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,"Termnination 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:phase_frequency():Frequency evolution no longer monotonic.
Terminated at orbital frequency(Hz): 907.465881  and step: 13515
Terminating chirp. Termination code set to:     1201
Returning to calling routine.

m1=1.400000  m2=1.400000  Initial_Freq=60.000000
steps_filld=13515  steps_alloc=16384  Max_Freq_Actual=907.465881
time_in_band=1.369419 clscnc_time=1.369547
Termination code: 1201

0        0.000000         0.000000          60.000000
1        0.000101         0.038086          60.001659
2        0.000203         0.076416          60.003315
3        0.000304         0.114502          60.004967
4        0.000405         0.152710          60.006622
5        0.000507         0.190918          60.008278
6        0.000608         0.229126          60.009930

 ...      ...              ...               ...

13507    1.368709         807.409851        731.514954
13508    1.368811         807.882446        753.420715
13509    1.368912         808.369873        778.076355
13510    1.369013         808.873962        806.010376
13511    1.369115         809.397034        837.697449
13512    1.369216         809.941467        872.924805
13513    1.369317         810.508545        907.445435
13514    1.369419         811.090820        907.465881
```

The first four 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 13515 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)$\times$Sample_Time is the length (in seconds) of the computed chirp. The quantity clscnc_time is the value of t_c that enters Eqs.(5.4.1) 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 1.37 seconds for two 1.4$M_\odot$ objects to spiral in from an orbital frequency of 60Hz to an orbital frequency of 907Hz. The chirp calculation was terminated at 907Hz

– 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`$/(2\pi) \approx$`129.09` 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/`Sample_Time`= 1.10333×10^{-4} seconds starting from when the binary had an orbital frequency of 60Hz until it neared "coalescence" at 907Hz have been calculated.

5.4 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 [6]

$$\begin{aligned} f(t) &= \frac{M_\odot}{16\pi T_\odot m_{\rm tot}}\Big\{\Theta^{-3/8} + \left(\frac{743}{2688}+\frac{11}{32}\eta\right)\Theta^{-5/8} - \frac{3\pi}{10}\Theta^{-3/4} \\ &\qquad + \left(\frac{1855099}{14450688}+\frac{56975}{258048}\eta+\frac{371}{2048}\eta^2\right)\Theta^{-7/8}\Big\}, \end{aligned} \tag{5.4.1}$$

where $m_{\rm tot}$ is the total mass of the binary in grams. The time integral of this equation gives the orbital evolution in cycles. Multiplying by 2π yields the orbital phase in radians

$$\begin{aligned} \phi(t) &= \phi_c - \frac{1}{\eta}\Big\{\Theta^{5/8} + \left(\frac{3715}{8064}+\frac{55}{96}\eta\right)\Theta^{3/8} - \frac{3\pi}{4}\Theta^{1/4} \\ &\qquad + \left(\frac{9275495}{14450688}+\frac{284875}{258048}\eta+\frac{1855}{2048}\eta^2\right)\Theta^{1/8}\Big\}. \end{aligned} \tag{5.4.2}$$

Here Θ is a dimensionless time variable

$$\Theta = \frac{\eta M_\odot}{5 T_\odot m_{\rm tot}}(t_c - t)\,, \tag{5.4.3}$$

$\eta = \mu/m_{\rm tot}$, 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 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 $T_\odot = 4.89128\times10^{-6}$ in Eq.(5.0.2).

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. (5.4.1) and (5.4.2). 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.* [10] 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.(5.4.1). 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 17 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 17. The primary problem in finding this root is that there may actually be no meaningfull start-time for the specified chirp. For example, if you you were to specify a chirp with two $1.4 M_\odot$ 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 viabilty. 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 17. 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 $\Theta = \frac{\eta(t_c - t)}{5T_\odot m_{\rm tot}} = X^8$ 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.(5.4.1) 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 $X \to 0$ (*i.e.* $t \to t_c$). 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.

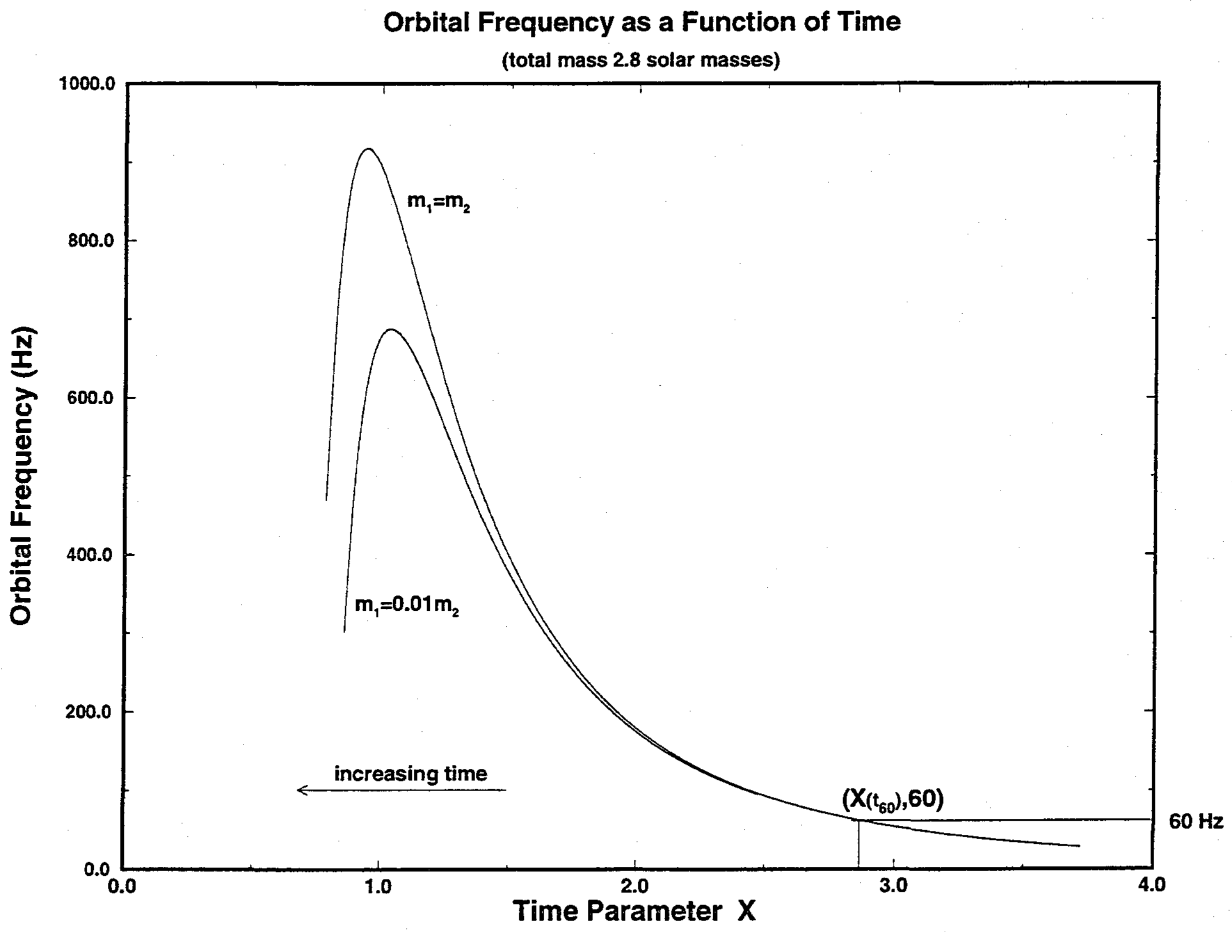


Figure 17: Orbital frequency as a function of the "time" coordinate $X = \left(\frac{\eta(t_c - t)M_\odot}{5T_\odot m_{\text{tot}}}\right)^{1/8}$.

5.5 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 inspiralling binary. The two chirps differ in phase by $\pi/2$ radians. The chirp values are given by Eqs.(5.6.1) and (5.6.2). 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.(5.6.1) 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.(5.6.2) 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 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.(5.4.1) and (5.4.2).

`err_cd_sprs`: Input. Error code supression. 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 approximantion 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 occured; 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.

5.6 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

$$h_c(t) = 2\left(\frac{\mu}{M_\odot}\right)\left[\frac{2\pi T_\odot m_{\rm tot} f(t)}{M_\odot}\right]^{2/3} \cos 2\phi(t) \tag{5.6.1}$$

and the other orbital-phase chirp which is $\pi/2$ out of phase with $h_c(t)$

$$h_s(t) = 2\left(\frac{\mu}{M_\odot}\right)\left[\frac{2\pi T_\odot m_{\rm tot} f(t)}{M_\odot}\right]^{2/3} \sin 2\phi(t)\ , \tag{5.6.2}$$

with all the leading numerical factors we display.

If the so called "restricted" post^2-Newtonian polarizations [leading order in the amplitude, but post^2-Newtonian phase corrections] are desired, they can be easily assembled from h_c and h_s. The "+" (plus) polarization is given by

$$h_+(t) = -\frac{T_\odot c}{D}(1+\cos^2 i)h_c(t)\ , \tag{5.6.3}$$

and the "×" (cross) polarization is given by

$$h_\times(t) = -2\frac{T_\odot c}{D}(\cos i)\ h_s(t)\ . \tag{5.6.4}$$

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 [7] figure 7 for the precise definition of the inclination angle.

The restricted post^2-Newtonian strain amplitude impinging on the detector can also be calculated from the output of `chirp_filters()` by

$$h(t) = F_+ h_+(t) + F_\times h_\times(t)\ , \tag{5.6.5}$$

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.(5.4.2) such that $\phi = 0$ at the beginning of the chirp. The astrophysical convention for defining the orbital phase angle ϕ given in [7] 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.(5.6.1) and (5.6.2). But should we use the logarithmically modulated phase variable

$$\psi(t) = \phi - \frac{4Gm_{\text{tot}}\pi f(t)}{c^3}\ln[f(t)/f_o] \tag{5.6.6}$$

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.(5.6.2) with the logarithmically corrected *sin* 2ψ

$$\begin{aligned}\sin 2\psi &= \sin\Big(2\phi - \frac{8\pi m_{\text{tot}} f G}{c^3}\ln(f(t)/f_o)\Big)\\ &= \sin 2\phi\cos\Big(\frac{8\pi m_{\text{tot}} f G}{c^3}\ln(f(t)/f_o)\Big) - \cos 2\phi \sin\Big(\frac{8\pi m_{\text{tot}} f G}{c^3}\ln[f(t)/f_o)\Big)\\ &\approx \Big(1+O(1/c^6)\Big)\sin 2\phi - \Big(\frac{8\pi m_{\text{tot}} f G}{c^3}\ln(f(t)/f_o)\Big)\cos 2\phi\,.\end{aligned} \tag{5.6.7}$$

The $O[1/c^6]$ is a post^3-Newtonian term and can be neglected in the present post^2-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 post^2-Newtonian analysis. This logarithmic term is included in the waveform calculation in the `strain()` routine. However, the last line of Eq.(5.6.7) 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 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 post^4-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.(5.6.6). 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.(5.4.3). In order to 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.

5.7 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, Bruce Allen */
#include "grasp.h"
int main() {
    float m1,m2,spin1,spin2,phaseterms[5],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 tail correction */
    phaseterms[4] =1.;          /* The 2PN 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;
}
```

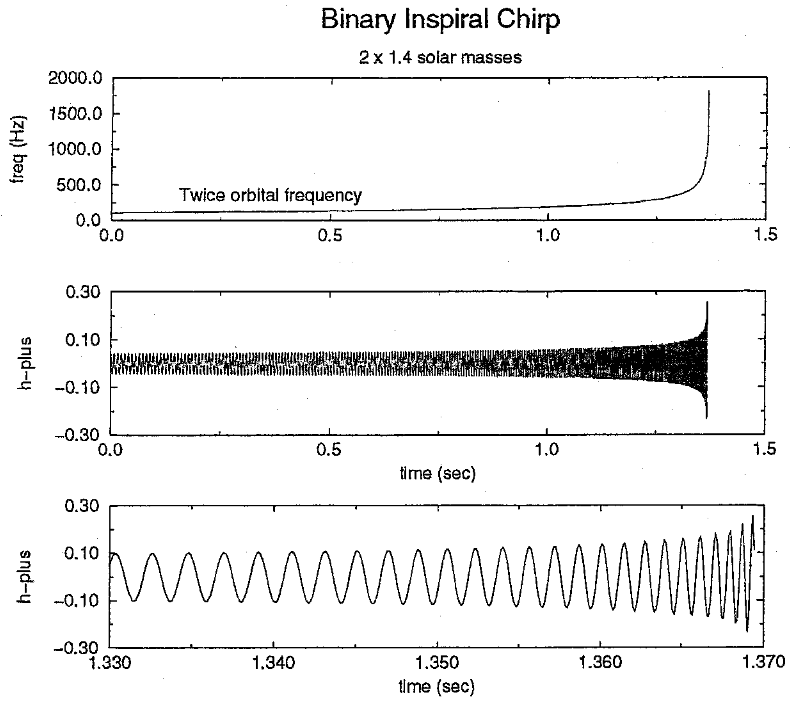


Figure 18: The zero-phase chirp waveform from a $2 \times 1.4 M_{\odot}$ binary system, starting at an orbital frequency of 60 Hz. The top graph shows the frequency of the dominant quadrupole radiation as a function of time, and the middle graph shows the waveform. The bottom graph shows a 40-msec stretch 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 reqested 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]`.

5.8 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 32. 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 5.14 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 32. 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 32. 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 32; you are now ready for wholesale production.

5.9 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)`

This function is an even more stripped down chirp generator, which fills a pair of arrays with waveforms for an inspiralling binary. The two chirps differ in phase by $\pi/2$ radians and are given by Eqs.(5.6.1) and (5.6.2). This routine assumes spinless masses, and computes a chirp with phase corrections up to and including second-order 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 coalescense measured from the first point output, in `ch*[0]`.

`err_cd_sprs`: Input. Error code supression. 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()`.

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.

5.10 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 $\mathrm{C}(t)$ 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 $\mathrm{C}(t) = T(t - t_0)$ where $T(t)$ is the waveform of a chirp which enters the sensitivity band of the interferometer at time $t = 0$ (for the moment, forget about the fact that the chirps come in two different phase "flavors").

We will construct a signal S which is a number, defined by

$$S = \int_{-\infty}^{\infty} dt\, h(t)Q(t), \tag{5.10.1}$$

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

$$\begin{aligned} S &= \int_{-\infty}^{\infty} dt \int_{-\infty}^{\infty} df \int_{-\infty}^{\infty} df' e^{-2\pi i f t + 2\pi i f' t} \tilde{h}(f)\tilde{Q}^*(f') \\ &= \int_{-\infty}^{\infty} df \int_{-\infty}^{\infty} df' \delta(f - f')\tilde{h}(f)\tilde{Q}^*(f') \\ &= \int_{-\infty}^{\infty} df \tilde{h}(f)\tilde{Q}^*(f). \end{aligned} \tag{5.10.2}$$

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:

$$h(t) = C(t) + n(t). \tag{5.10.3}$$

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

$$\langle S \rangle = \int_{-\infty}^{\infty} df \langle \tilde{h}(f) \rangle \tilde{Q}^*(f) = \int_{-\infty}^{\infty} df \tilde{C}(f)\tilde{Q}^*(f). \tag{5.10.4}$$

We now define the *noise* N to be the difference between the signal value and its mean for any given element of the ensemble:

$$N \equiv S - \langle S \rangle = \int_{-\infty}^{\infty} df \tilde{n}(f)\tilde{Q}^*(f). \tag{5.10.5}$$

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

$$\langle \tilde{n}(f)\tilde{n}^*(f')\rangle = \frac{1}{2}S_h(|f|)\delta(f-f'), \tag{5.10.6}$$

and has the property that

$$\langle n^2(t)\rangle = \int_0^\infty S_h(f)\, df. \tag{5.10.7}$$

We can now find the expected value of N^2, by squaring equation (5.10.5), taking the expectation value, and using (5.10.6), obtaining

$$\begin{aligned}\langle N^2\rangle &= \int_{-\infty}^{\infty} df \int_{-\infty}^{\infty} df' \tilde{Q}^*(f)\langle \tilde{n}(f)\tilde{n}^*(f')\rangle \tilde{Q}(f') \\ &= \frac{1}{2}\int_{-\infty}^{\infty} df\; S_h(|f|)|\tilde{Q}(f)|^2 \\ &= \int_0^\infty df\; S_h(f)|\tilde{Q}(f)|^2. \end{aligned} \tag{5.10.8}$$

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

$$(A,B) = \int_{-\infty}^{\infty} df\; A(f)B^*(f)S_h(|f|). \tag{5.10.9}$$

Because S_h is real, 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

$$\langle S\rangle = (\frac{\tilde{C}}{S_h}, \tilde{Q}) \quad \text{and} \quad \langle N^2\rangle = \frac{1}{2}(\tilde{Q},\tilde{Q}). \tag{5.10.10}$$

(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: to maximize the signal-to-noise ratio

$$\left(\frac{S}{N}\right)^2 = \frac{\langle S\rangle^2}{\langle N^2\rangle} = 2\frac{(\frac{\tilde{C}}{S_h},\tilde{Q})^2}{(\tilde{Q},\tilde{Q})} \tag{5.10.11}$$

we choose

$$\tilde{Q}(f) = \frac{\tilde{C}(f)}{S_h(|f|)} = \frac{\tilde{T}(f)}{S_h(|f|)}\, e^{2\pi i f t_0}. \tag{5.10.12}$$

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

$$S \;=\; \int_{-\infty}^{\infty} df\; \tilde{h}(f)\tilde{Q}^*(f)$$

$$= \int_{-\infty}^{\infty} df \, \frac{\tilde{h}(f)\tilde{C}^*(f)}{S_h(|f|)}$$
$$= \int_{-\infty}^{\infty} df \, \frac{\tilde{h}(f)\tilde{T}^*(f)}{S_h(|f|)} \, e^{-2\pi i f t_0}. \tag{5.10.13}$$

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 (5.10.12) into equation (5.10.11), obtaining

$$\left(\frac{S}{N}\right)^2 = 2(\tilde{Q},\tilde{Q}) = 2\int_{-\infty}^{\infty} df \frac{|\tilde{C}(f)|^2}{S_h(|f|)} = 4\int_0^{\infty} df \frac{|\tilde{C}(f)|^2}{S_h(f)}. \tag{5.10.14}$$

You will notice that the signal-to-noise ratio S/N in (5.10.11) is independent of the overall normalization of the optimal 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 will frequently specify the normalization of the filter so that the expected noise-squared from a specified source is unity: $\langle N^2 \rangle = 1$. 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/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. Suppose that normalizing the optimal filter for this system so that $\langle N^2 \rangle = 1$ corresponds to putting the system at a distance of 15 megaparsecs (i.e., choosing $C(t)$ 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 ration $S/N = 30$, this corresponds to detecting an optimally-oriented source at a distance of half a megaparsec.

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 s:utility, 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 (5.6.1) and (5.6.2). 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:

$$h(t) = \alpha T_0(t) + \beta T_{90}(t) + n(t). \tag{5.10.15}$$

Here the subscripts 0 and 90 label the two possible orbital phases; the constants α and β depend upon the distance to the source (or 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 (5.6.1) and $T_{90}(t)$ denotes the (suitably normalized) function $h_s(t)$ given by equation (5.6.2).

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

$$\tilde{Q}(f) = \frac{\tilde{T}_0(f) - i\tilde{T}_{90}(f)}{S_h(|f|)} \, e^{2\pi i f t_0}. \tag{5.10.16}$$

We will assume that the individual filters for each polarization are normalized by the convention just described, and that they are orthogonal:

$$\left(\frac{\tilde{T}_0}{S_h}, \frac{\tilde{T}_0}{S_h}\right) = 2, \text{ and } \left(\frac{\tilde{T}_{90}}{S_h}, \frac{\tilde{T}_{90}}{S_h}\right) = 2, \text{ and } \left(\frac{\tilde{T}_0}{S_h}, \frac{\tilde{T}_{90}}{S_h}\right) = 0. \tag{5.10.17}$$

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

$$S(t_0) = \left(\frac{\tilde{h}}{S_h}, \tilde{Q}\right) \tag{5.10.18}$$

is a complex-valued functions of the lag t_0. We define the noise as before, by $N = S - \langle S \rangle$. It's mean-squared modulus is

$$\begin{aligned} \langle |N|^2 \rangle &= \frac{1}{2}(\tilde{Q}, \tilde{Q}) \\ &= \frac{1}{2}\left(\frac{\tilde{T}_0 - i\tilde{T}_{90}}{S_h}, \frac{\tilde{T}_0 - i\tilde{T}_{90}}{S_h}\right) \\ &= \frac{1}{2}\left[\left(\frac{\tilde{T}_0}{S_h}, \frac{\tilde{T}_0}{S_h}\right) + \left(\frac{\tilde{T}_{90}}{S_h}, \frac{\tilde{T}_{90}}{S_h}\right)\right] = 2, \end{aligned} \tag{5.10.19}$$

where we have made use of the orthornormality relation (5.10.17). Now the expected signal at zero lag $t_0 = 0$ is

$$\langle S \rangle = \left(\frac{\langle \tilde{h} \rangle}{S_h}, \tilde{Q}\right) = \left(\frac{\alpha\tilde{T}_0 + \beta\tilde{T}_{90}}{S_h}, \frac{\tilde{T}_0 - i\tilde{T}_{90}}{S_h}\right) = 2\alpha + 2i\beta. \tag{5.10.20}$$

Hence the signal-to-noise ratio is

$$\frac{\langle S \rangle}{\sqrt{\langle |N|^2 \rangle}} = \alpha + i\beta. \tag{5.10.21}$$

In the absence of a signal $\langle S \rangle = 0$ and the variance of this quantity (from the definition of N) is unity:

$$\frac{\langle |S|^2 \rangle}{\langle |N|^2 \rangle} = 1. \tag{5.10.22}$$

In the presence of a signal, the signal-to-noise ratio is

$$\frac{|\langle S \rangle|^2}{\langle |N|^2 \rangle} = \alpha^2 + \beta^2 = \frac{1}{2}\sqrt{\left(\frac{\tilde{h}}{S_h}, \frac{\tilde{T}_0}{S_h}\right)^2 + \left(\frac{\tilde{h}}{S_h}, \frac{\tilde{T}_{90}}{S_h}\right)^2}. \tag{5.10.23}$$

The attentive reader will notice 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. 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 (5.10.11) 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 $\langle S \rangle / N = \alpha$.

5.11 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 (5.10.13):

$$s(t) = \frac{1}{2}\int_{-\infty}^{\infty} df\, \tilde{h}(f)\tilde{c}^*(f)\tilde{r}(f)\, e^{-2\pi i f t}. \tag{5.11.1}$$

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 (5.11.1) for efficiency reasons; in order to calculate the integral (5.10.13) 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

$$t = 0, \Delta t, 2\Delta t, \cdots, (n-1)\Delta t. \tag{5.11.2}$$

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 $\tilde{h}(f_{\rm Nyquist})$ 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 $\Delta f = 2 f_{\rm Nyquist}/n = (n\Delta t)^{-1}$. Array elements `h[2j]` and `h[2j+1]` contain the real and imaginary parts of $\tilde{h}(j\ \Delta f)$ for $j = 1, \cdots, n/2-1$. 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 $s[0] \geq 0$.

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 (5.10.13), 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 $j = 0, 1, \cdots, n/2$. 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

$$\mathtt{s}_k = \frac{1}{2}\sum_{j=0}^{\mathtt{n}-1} \mathtt{h}_j \mathtt{c}_j^* \mathtt{r}_j e^{-2\pi i jk/\mathtt{n}} \tag{5.11.3}$$

where it is understood that $\tilde{h}_{\mathtt{n}-j} = \tilde{h}_j^*$ and that $\tilde{c}_{\mathtt{n}-j} = \tilde{c}_j^*$, and that $\tilde{r}_{\mathtt{n}-j} = \tilde{r}_j$.

5.13 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 $2/S_h$.

`n`: Input. Defines the length of the arrays: `ch0tilde[0..n-1]`, `ch90tilde[0..n-1]`, and `twice_inv_noise[`

`n0`: Output. The normalization of the 0-phase chirp.

`n90`: Output. The normalization of the 90°-phase chirp.

Using the notation of (5.10.9) one may define an inner product of the chirps. The normalizations are defined as follows:

$$\frac{1}{n_0^2} \equiv \frac{1}{2}(Q_0, Q_0), \tag{5.13.1}$$

where Q_0 is the optimal filter defined for the zero-phase chirp T_0. The chirps are orthogalized 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 $T_{90} \rightarrow T_{90} - \epsilon T_0$. This ensures that the inner product (Q_{90}, Q_0) vanishes. The normalization for this newly-defined chirp is then defined by

$$\frac{1}{n_1^2} \equiv \frac{1}{2}(Q_{90}, Q_{90}). \tag{5.13.2}$$

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.

5.14 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 it's 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.

$$S(t_0) = \int h(t)T(t-t_0)dt \approx S(i_0) = \sum_j h_j T_{j-i_0}, \tag{5.14.1}$$

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

$$h_j = \sum_{k=0}^{N-1} \mathrm{e}^{-2\pi ijk/N}\tilde{h}_k \quad \text{and} \quad T_{j-i_0} = \sum_{k'=0}^{N-1} \mathrm{e}^{-2\pi i(j-i_0)k'/N}\tilde{T}_{k'} \tag{5.14.2}$$

one can easily compute that the signal as a function of lag i_0 is

$$\begin{aligned} S(i_0) &= \sum_{j=0}^{N-1}\sum_{k=0}^{N-1}\sum_{k'=0}^{N-1} \mathrm{e}^{-2\pi ijk/N}\tilde{h}_k \mathrm{e}^{-2\pi i(j-i_0)k'/N}\tilde{T}_{k'} && (5.14.3)\\ &= \sum_{k=0}^{N-1}\sum_{k'=0}^{N-1} N\delta_{k,-k'}\mathrm{e}^{2\pi ii_0k'/N}\tilde{h}_k\tilde{T}_{k'} && (5.14.4)\\ &= \sum_{k=0}^{N-1} N\mathrm{e}^{-2\pi ii_0k/N}\tilde{h}_k\tilde{T}^*_k. && (5.14.5) \end{aligned}$$

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 19 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 20 then the 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 21 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 $i = 0\cdots, N-m$.

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) varities of non-stationary interferometer noise. Suppose that a chirp filter is triggered by some kind of transient event in

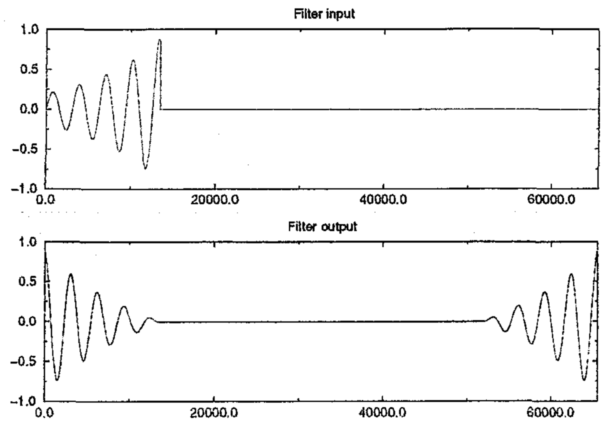


Figure 19: A chirp starting at initial time $i = 0$ and ending at time $i = 13500$ is processed through a chirp filter, whose output peaks at time $i = 0$. Notice that because of wraparound, the (non-causal) filter output begins "earlier" than $i = 0$.

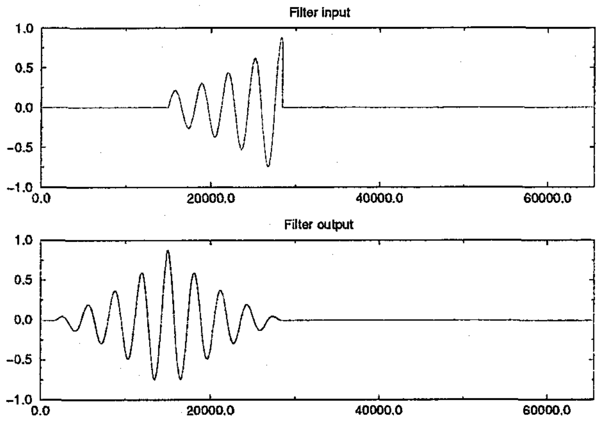


Figure 20: A chirp starting at initial time $i = 15,000$ and ending at time $i = 28,500$ is processed through a chirp filter, whose output peaks at time $i = 15,000$.

the IFO output. At what time did this transient event ocurr? 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 the range $i = 0, \cdots, N-m-1$ we can detect either true chirps in the data stream, starting in the time interval $i = 0, \cdots, N-m-1$ and coalescing (roughly speaking) in the time inverval $i = m, \cdots, N-1$, or we can detect transient impulse-like events in the data stream, which take place in the time interval $i = m, \cdots, N-1$. In the GRASP optimal filtering code, after examining the stretch of N data points, we then shift the data points $i = N-m, \cdots, N-1$ into the range $i = 0, \cdots, m-1$ and aquire a new additional set of $N-m$ data points covering remaining (new) time interval.

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

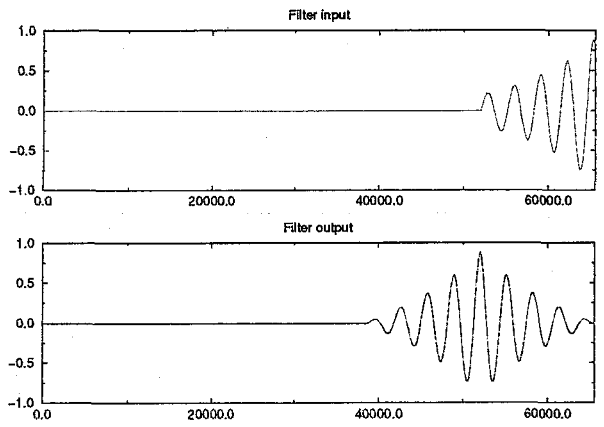


Figure 21: A chirp starting at initial time $i = 52,035$.

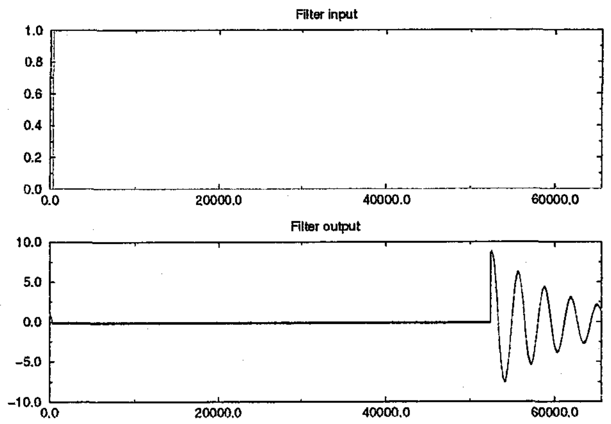


Figure 22: An impulse shortly after $i = 0$.

filtering example. You lose a tiny bit of efficiency but reduce the likelyhood that boundary effects from the data discontinuity at the start/end of the rectangular window will significantly stimulate the optimal filter output for $i = 0, \cdots, N - m - 1$. (See Figs. 22 and 25 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 caseful about wrap-around errors in relating the output of a filter to the time of occurence of a signal or impulse.

There is one remaining ugly question. The optimal filter $\tilde{Q}$ depends upon the noise power spectrum 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

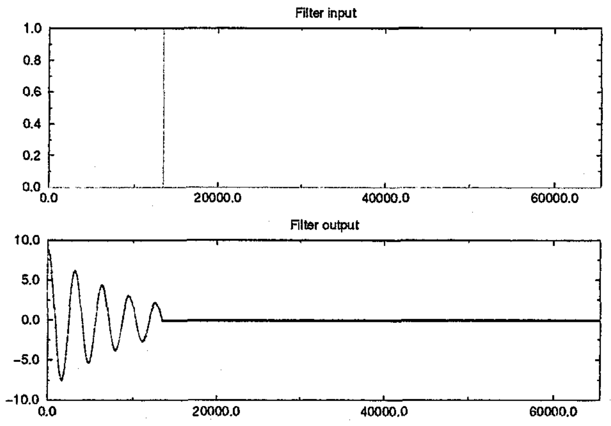


Figure 23: An impulse at $i = 15,000$.

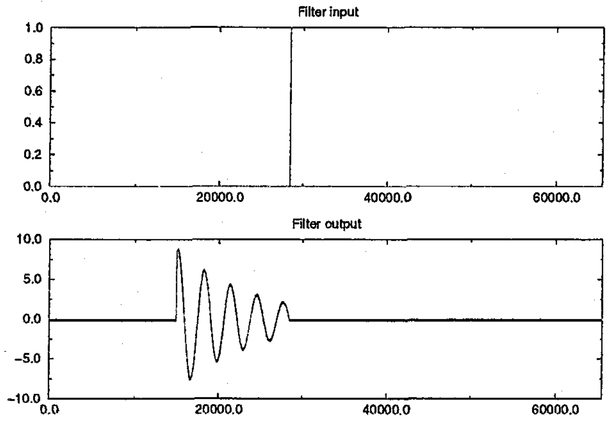


Figure 24: An impulse at $i = 28,500$.

for using only properly-windowed data to find the noise power spectrum to use in determing 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.

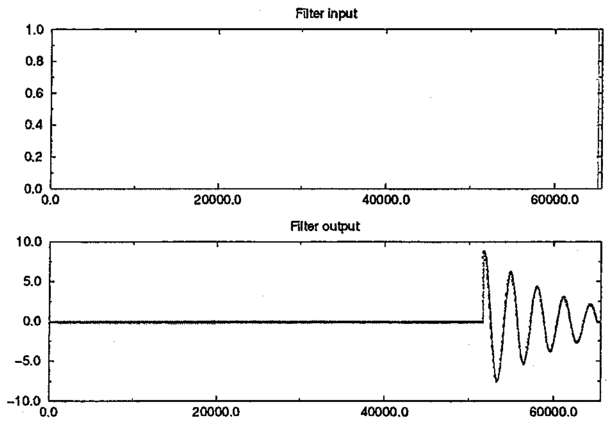


Figure 25: An impulse shortly before $i = 65535$

5.15 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.

`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.

5.16 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 it's 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.

5.17 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 it's 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.

5.18 Vetoing techniques

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, interupted 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 ouput 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.

The second technique is described here, and operates in the frequency domain. 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.

Suppose that one of our optimal chirp filters $\tilde{Q}$ is triggered with a large SNR at time t_0. We will denote the signal value at this time by S:

$$S = \int_{-f_{\rm Ny}}^{f_{\rm Ny}} df \, \frac{\tilde{h}(f)\tilde{T}^*(f)}{S_h(|f|)} \, e^{-2\pi i f t_0}. \tag{5.18.1}$$

(Here, $f_{\rm Ny}$ denotes the Nyqist frequency, one-half of the sampling rate.) The chirp template T is normalized so that the expected value $\langle N^2 \rangle = 1$:

$$\int_0^{f_{\rm Ny}} df \, \frac{|\tilde{T}(f)|^2}{S_h(|f|)} = 1. \tag{5.18.2}$$

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 $\Delta f_1, \cdots, \Delta f_p$ 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:

$$\begin{aligned} \Delta f_1 &= \{f \mid 0 < f < f_1\} \\ \Delta f_2 &= \{f \mid f_1 < f < f_2\} \\ &\cdots \\ \Delta f_p &= \{f \mid f_{p-1} < f < f_{\rm Ny}\}, \end{aligned} \tag{5.18.3}$$

are defined by the condition that the *expected signal contributions in each frequency band from a chirp are equal*:

$$\int_{\Delta f_i} df\, \frac{|\tilde{T}(f)|^2}{S_h(|f|)} = \frac{1}{p}\int_0^{f_{\rm Ny}} df\, \frac{|\tilde{T}(f)|^2}{S_h(|f|)} \tag{5.18.4}$$

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 fairly narrow in regions of frequency space where the interferometer is quiet, and they are fairly wide in regions where the IFO is noisy.

Now, define a set of p signal values, one for each frequency interval:

$$S_i = \int_{-\Delta f_i \cup \Delta f_i} df\, \frac{\tilde{h}(f)\tilde{T}^*(f)}{S_h(|f|)}\, {\rm e}^{-2\pi i f t_0} \quad \text{for } i = 1, \cdots, p. \tag{5.18.5}$$

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, then the expected value of each of these signal values is $\langle S_i \rangle = S/p$. In this case the values of $\Delta S_i \equiv S_i - S/p$ are independent normal random variables with a mean value of zero and a variance σ determined by the expected value of the noise-squared. Because of our choice of template normalization this is:

$$\sigma = \langle \Delta S_i^2 \rangle = \langle N^2 \rangle / p = 1/p. \tag{5.18.6}$$

Hence, in the presence of a true chirp and interferometer noise, the probability distribution of the ΔS_i is given by

$$P(\Delta S_1, \cdots, \Delta S_p) = \prod_{i=1}^{p} (2\pi\sigma)^{-1/2} {\rm e}^{-\Delta S_i^2/2\sigma} = (2\pi\sigma)^{-p/2} {\rm e}^{-(\Delta S_1^2 + \cdots + \Delta S_p^2)/2\sigma}. \tag{5.18.7}$$

Thus, if our optimal chirp filter is triggered by an event, we can check the contributions to the signal in each of p frequency subintervals, to determine if the distribution of frequency and the arrival times in the p distinct subintervals is consistent with "chirp + Gaussian noise".

Because the ΔS_i are independent random variables with zero mean and variance $1/p$, the sum of their squares is described by a χ^2 probability distribution. Define the statistic

$$r^2 = \sum_{i=1}^{p} (\Delta S_i)^2. \tag{5.18.8}$$

Then one can easily compute the probability distribution of r. The probability that $r > R$ in the presence of a true chirp signal is

$$\begin{aligned} P(r > R) &= (2\pi\sigma)^{-p/2}\Omega_{p-1} \int_R^\infty r^{N-1} {\rm e}^{-r^2/2\sigma} dr & (5.18.9) \\ &= \frac{1}{\Gamma(p/2)} \int_{R^2/2\sigma}^\infty x^{p/2-1} {\rm e}^{-x} dx & (5.18.10) \\ &= Q(p/2, R^2/2\sigma), & (5.18.11) \end{aligned}$$

where Ω_p is the $p-$volume of a unit-radius $p-$sphere S^p. The incomplete gamma function Q is the same function that describes the likelyhood function in the traditional χ^2 test.

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 $Q(4, 10.0) = 0.0103\cdots$ so if $r^2 > 2.5$ then one can conclude that the likelyhood that a given trigger is actually due to a chirp is less than 1%; rejecting or vetoing such events will only reduce the “true event” rate by 1%. 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!

Note that this technique is probably a computationally-efficient and simple version of the maximum-likelyhood statistical test. This test is probably obtained in the limit where the number of frequency bins equals p.

5.19 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 $\tilde{T}(f_{\rm Nyquist})$ 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 $\Delta f = 2f_{\rm Nyquist}/n = (n\Delta t)^{-1}$. Array elements `template[2j]` and `template[2j+1]` contain the real and imaginary parts of $\tilde{T}(j\,\Delta f)$ for $j = 1, \cdots, n/2 - 1$.

`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 (5.10.13), 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 $j = 0, 1, \cdots, n/2$. 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 choose 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 $f_{\rm Nyquist}$.

`indices`: Ouput. 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..ind`
Note that `indices[p-1]=n-1`.

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

Comments: None.

5.20 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.

`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 $i = 1, \cdots, p$.

`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.

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

$$s(t) = \frac{1}{2} \int_{-\infty}^{\infty} df \; |\tilde{c}(f)|^2 \tilde{r}(f) \; e^{-2\pi i f t}. \qquad (5.11.4)$$

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).

5.12 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/Ouput. 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.

`twice_inv_noise`: Output. The array `twice_inv_noise[0..n/2]` contains $2/S_h(f)$. It is set to zero for $f < f_{low}$.

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.

5.21 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.

`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 $i = 1, \cdots, p$.

`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.

5.22 Example: `optimal` program

This program reads the 40-meter data stream, and then filters it though 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, $\Delta t = 1/\texttt{srate}$ is the sample time in seconds, and $\Delta f = (n\Delta t)^{-1} = \texttt{srate/npoint}$ is the size of a frequency bin, in Hz. Here $n = \texttt{npoint}$ 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

$$\tilde{T}_0(f) = \frac{\Delta t}{\texttt{HSCALE}} \texttt{chirp0[]} \tag{5.22.1}$$

$$\tilde{T}_{90}(f) = \frac{\Delta t}{\texttt{HSCALE}} \texttt{chirp1[]} \tag{5.22.2}$$

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

$$\texttt{response[]} = \texttt{HSCALE/ARMLENGTH} \times R(f), \tag{5.22.3}$$

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

$$\tilde{h}(f) = \frac{\Delta t}{\texttt{HSCALE}}\texttt{htilde[]}. \tag{5.22.4}$$

The one-sided noise power spectrum $S_h(f)$ is the average of

$$S_h(f) = \frac{2}{n\Delta t}|\tilde{h}(f)|^2 = \frac{2}{n\Delta t}\frac{(\Delta t)^2}{\texttt{HSCALE}^2}|\texttt{htilde[]}|^2 = \frac{2\Delta t}{n\ \texttt{HSCALE}^2}|\texttt{htilde[]}|^2. \tag{5.22.5}$$

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

$$S_h(f) = \frac{2\Delta t}{n\ \texttt{HSCALE}^2}\langle|\texttt{htilde[]}|^2\rangle = \frac{\Delta t}{n\ \texttt{HSCALE}^2}\texttt{mean\_pow\_spec[]} \tag{5.22.6}$$

Twice the inverse of this average is stored in the array `twice_inv_noise[ ]`, so that

$$\frac{2}{S_h(f)} = \frac{n\ \texttt{HSCALE}^2}{\Delta t}\ \texttt{twice\_inv\_noise[]}. \tag{5.22.7}$$

The expected noise-squared for the plus polarization is given by equation (5.10.8):

$$\langle N^2 \rangle = \frac{1}{2}(Q,Q) = \frac{1}{2}\int_{-\infty}^{\infty} df\ \frac{|\tilde{T}_0(f)|^2}{S_h(f)}$$

$$
\begin{aligned}
&= \frac{1}{2}\frac{1}{n\Delta t} FFT_0^{-1}\left[\frac{(\Delta t)^2}{\texttt{HSCALE}^2}|\texttt{chirp0[]}|^2 \frac{n\texttt{HSCALE}^2}{\Delta t}\frac{1}{2}\texttt{twice\_inv\_noise[]}\right] \\
&= \frac{1}{2} FFT_0^{-1}\left[|\texttt{chirp0[]}|^2 \frac{1}{2}\texttt{twice\_inv\_noise[]}\right] \\
&\rightarrow \frac{1}{2}\texttt{correlate}(\cdots, \texttt{chirp0[]}, \texttt{chirp0[]}, \texttt{twice\_inv\_noise[]}, \texttt{npoint}). \qquad (5.22.8)
\end{aligned}
$$

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

$$
\begin{aligned}
S &= (\frac{\tilde{h}}{S_h}, \tilde{Q}) \\
&= \int_{-\infty}^{\infty} df\, \frac{\tilde{h}(f)\tilde{T}_0^*(f)}{S_h(f)}\, e^{-2\pi i f t_0} \\
&= \frac{1}{n\Delta t} FFT_i^{-1}\left[\frac{\Delta t}{\texttt{HSCALE}}\texttt{htilde[]}\frac{\Delta t}{\texttt{HSCALE}}\, (\texttt{chirp0[]})^* \,\frac{n\,\texttt{HSCALE}^2}{\Delta t}\,\frac{1}{2}\texttt{twice\_inv\_noise[]}\right] \\
&= FFT_i^{-1}\left[\texttt{htilde[]}\ \texttt{chirp0[]}\ \frac{1}{2}\texttt{twice\_inv\_noise[]}\right] \\
&\rightarrow \texttt{correlate}(\cdots, \texttt{htilde[]}, \texttt{chirp0[]}, \texttt{twice\_inv\_noise[]}, \texttt{npoint}), \qquad (5.22.9)
\end{aligned}
$$

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, 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 ARMLENGTH 40.0        /* Armlength of the IFO, in meters */
#define HSCALE 1.e21          /* A convenient scaling factor; results independent of it */
#define MIN_INTO_LOCK 3.0     /* Number of minutes to skip into each locked section */
#define SAFETY 1000           /* Padding safety factor to avoid wraparound errors */

int main() {
    void realft(float*,unsigned long,int);
    int i,code,npoint,remain=0,maxi,chirplen,needed,diff,impulseoff,chirppoints,indices[8];
    float distance,snr_max,srate=9868.4208984375,tstart,var,*mean_pow_spec,timeoff,timestart;
    float *data,*htilde,*output90,*output0,*chirp0,*chirp90,*ch0tilde,*ch90tilde,*response;
    float n0,n90,inverse_distance_scale,decaytime,*twice_inv_noise,datastart,tc;
    float lin0,lin90,invMpc_inject,varsplit,stats[8],gammq(float,float);
    double decay,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");
    fplock=grasp_open("GRASP_DATAPATH","channel.10");
    fpss=grasp_open("GRASP_DATAPATH","swept-sine.ascii");

    /* 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 */
output0=(float *)malloc(sizeof(float)*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;

/* manufacture two chirps (dimensionless strain at 1 Mpc distance) */
make_filters(1.4,1.4,chirp0,chirp90,FLO,npoint,srate,&chirppoints,&tc,0);
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;
}

/* 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 aquires data, then filters it */
while (1) {
   /* Seek MIN_INTO_LOCK minutes into a locked stretch of data */
   while (remain<needed) {
      code=get_data(fpifo,fplock,&tstart,MIN_INTO_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);

/* inject a signal in frequency domain, if desired */
invMpc_inject=0.0;    /* To inject a signal at 10 kpc, set this to 100.0 */
freq_inject_chirp(-0.406,0.9135,23456,invMpc_inject,ch0tilde,ch90tilde,htilde,npoint);

/* 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);

/* 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=0.5/n0+0.5/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 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,8,indices,stats,output0,htilde);
   prob=gammq(4.0,4.0*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) coalescence time %.2f\n",
                maxi,timestart,timestart+tc);
   printf("   Normalization: S/N=1 at %.2f kpc\n",1000.0*distance);
   printf("   Linear combination of max SNR: %.4f x phase_0 + %.4f x phase_pi/2\n",lin0,lin9
   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;
     }
  }
```

5.23 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 x phase_0 + 0.3638 x 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 x phase_0 + 0.9185 x 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 x phase_0 + 0.8787 x 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 if 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,npoint)
```

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.

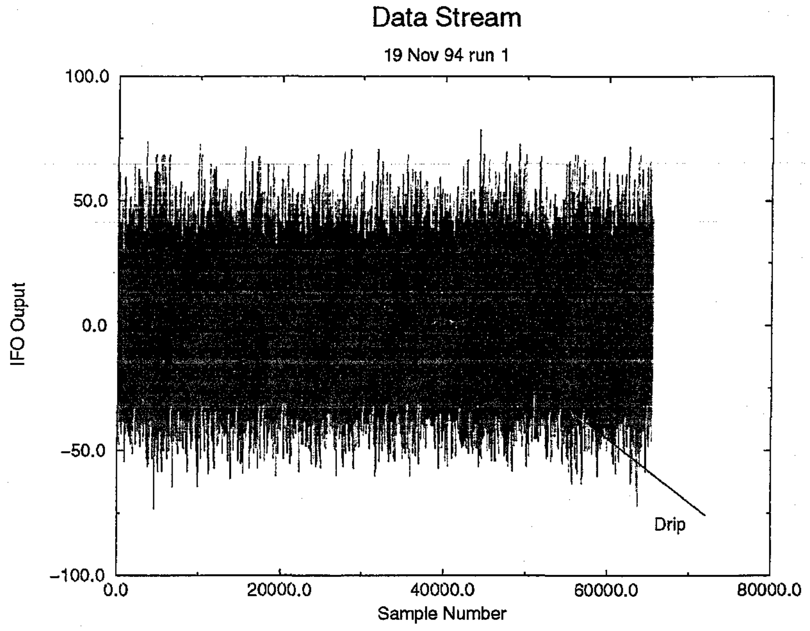


Figure 26: This shows the event that triggered the 2×1.4 solar mass binary inspiral filter with a SNR of 8.71 (see the first set of sample output from the optimal filtering code above, at time 325.23). This same "event" can also be seen in Figure 7. The horizontal axis is sample number, with samples $\approx 10^{-4}$ 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.

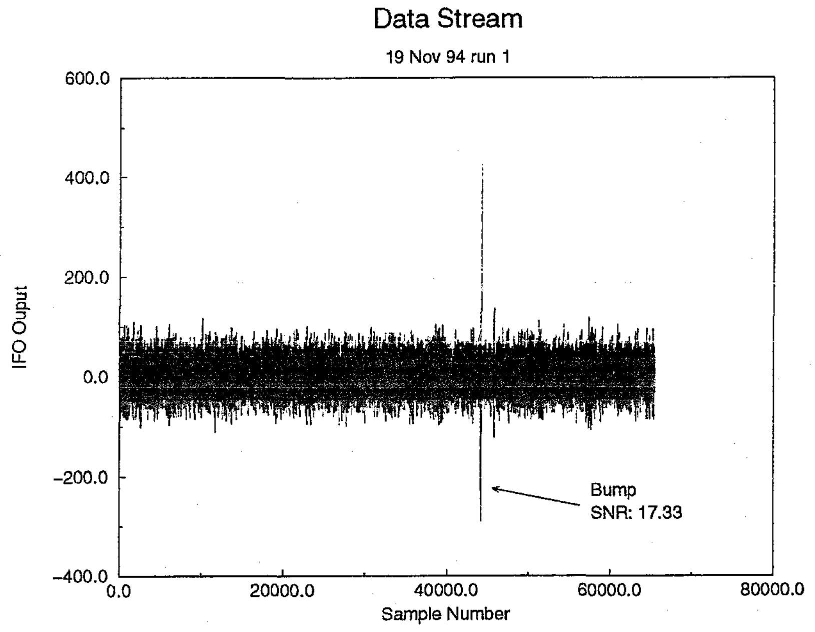


Figure 27: This another event that triggered the 2×1.4 solar mass binary inspiral filter with a SNR of 17.33. This event sounds like a "bump"; it is probably due to a bad cable connection. It can be easily seen (and vetoed) in the time domain. A close-up of this is shown in the next figure.

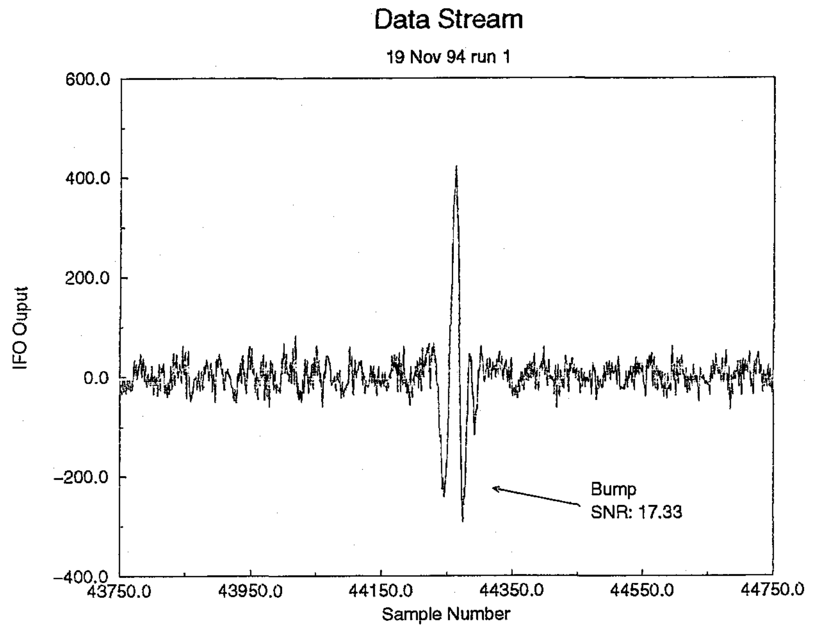


Figure 28: A close-up of the previous graph, showing the structure of the "bump".

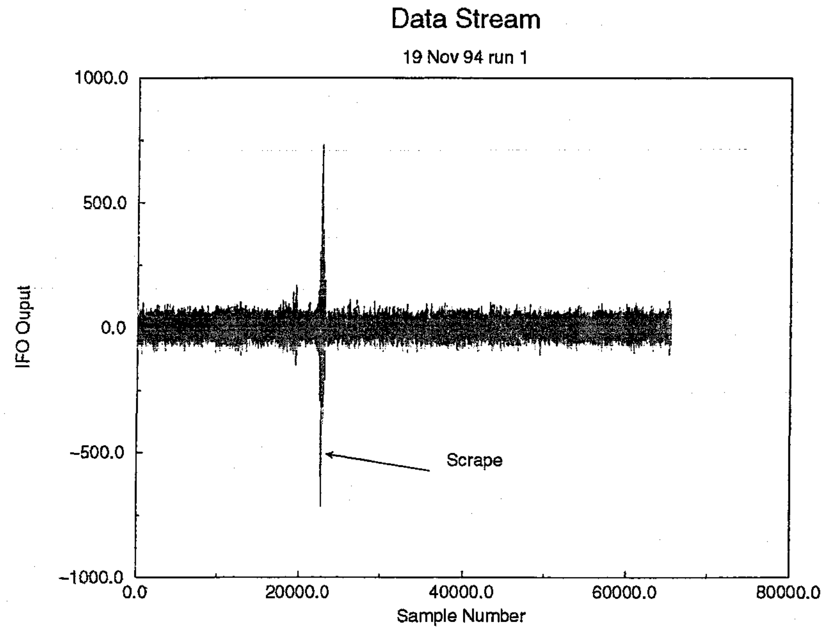


Figure 29: This another event that triggered the 2 × 1.4 solar mass binary inspiral filter with a SNR of 32.77. This event sounds like a shovel scraping on the ground; its origin is unknown. It can be easily seen (and vetoed) in the time domain.

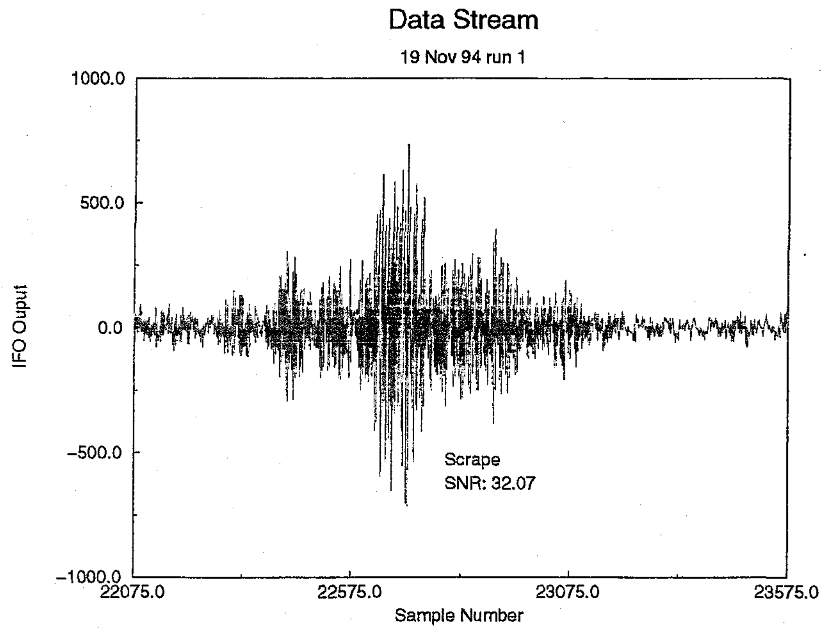


Figure 30: A close-up of the previous graph, showing the structure of the "scrape".

5.24 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

$$\begin{aligned} \tau_0 &= \frac{5}{256}\left(\frac{GM}{c^3}\right)^{-5/3} \eta^{-1}(\pi f_0)^{-8/3} \\ &= \frac{5}{256}\left(\frac{M}{M_\odot}\right)^{-5/3} \eta^{-1}(\pi f_0)^{-8/3} T_\odot^{-5/3} \end{aligned} \quad (5.24.1)$$

and

$$\begin{aligned} \tau_1 &= \frac{5}{192}\left(\frac{c^3}{G\eta M}\right)\left(\frac{743}{336} + \frac{11}{4}\eta\right)(\pi f_0)^{-2} \\ &= \frac{5}{192}\left(\frac{M_\odot}{M}\right)\left(\frac{743}{336}\eta^{-1} + \frac{11}{4}\right)(\pi f_0)^{-2} T_\odot^{-1}. \end{aligned} \quad (5.24.2)$$

The symbol

$$M \equiv m_1 + m_2 \quad (5.24.3)$$

denotes the total mass of the binary system, and

$$\eta \equiv \frac{m_1 m_2}{(m_1 + m_2)^2} \quad (5.24.4)$$

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 31. Since we may take $m_2 \le m_1$ 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 32 In this diagram, the lower curve $\tau_1 \propto \tau_0^{2/3}$ 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).

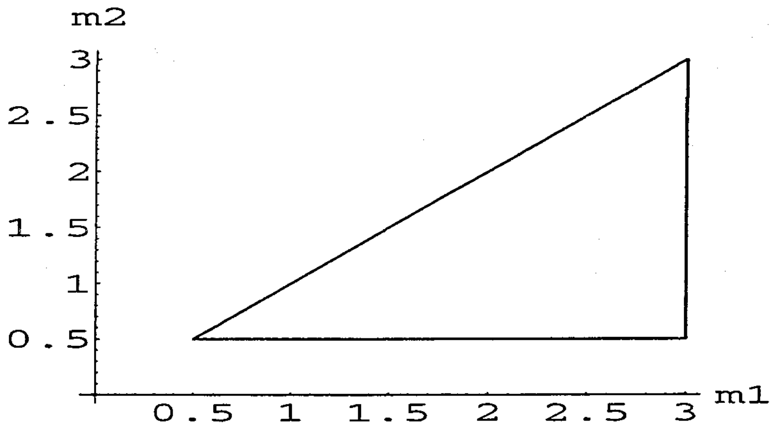


Figure 31: The set of binary stars with masses lying between set minimum and maximum values defines the interior of a triangle in parameter space

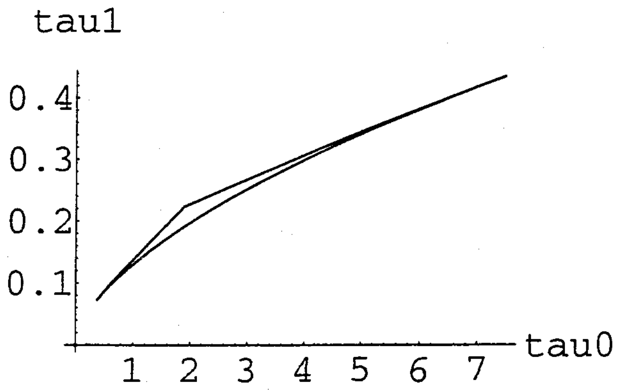


Figure 32: The triangular region of the previous figure is mapped into a distorted triangle in the (τ_0, τ_1) plane. Here f_0 is 120 Hz.

5.25 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 32. 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 of the major axis of the constant ambiguity ellipses, in radians counter-clockwise from the τ_0 axis.

`float dp`: The diameter along the minor axis of the ellipse (in sec). This is twice the radius dx_0 given in Table 7.

`float dq`: The diameter along the major axis of the ellipse (in sec). This is twice the radius dx_1 given in Table 7.

`float f_start`: The frequency f_0 used in the definitions of τ_0 and τ_1 (5.24.1,5.24.2); 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.

`};`

5.26 Function: `tau_of_mass()`

`void tau_of_mass(float m1, float m2, float pf, float *tau0, float *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 (5.24.1) and (5.24.2) 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.

5.27 Function: m_and_eta()

```
int m_and_eta(float tau0, float tau1, float *M, float *eta, float Mmin, float Mmax,
float 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 $M_{\rm min}$ to $M_{\rm max}$ 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 (5.24.1) and τ_1 (5.24.2) to obtain the following relation:

$$c_1 + c_2 \left(\frac{M}{M_\odot}\right)^{5/3} - c_3 \left(\frac{M}{M_\odot}\right) = 0, \tag{5.27.1}$$

with the constants given by:

$$\begin{aligned} c_1 &= 1155\, T_\odot \\ c_2 &= 47552\, (\pi f_0 T_\odot)^{8/3} \tau_0 \\ c_3 &= 16128\, (\pi f_0 T_\odot)^2 \tau_1 . \end{aligned} \tag{5.27.2}$$

Given (τ_0, τ_1) our goal is to find the roots of equation (5.27.1). It is easy to see that the function on the lhs of (5.27.1) 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 $M \to \infty$. Hence the function on the lhs of (5.27.1) 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 $M_{\rm crit}$ which satisfies

$$\frac{M_{\rm crit}}{M_\odot} = \left(\frac{3}{5}\frac{c_3}{c_2}\right)^{3/2} \tag{5.27.3}$$

Hence the lhs of (5.27.1) has no roots if its value is positive at $M = M_{\rm crit}$ or it has two roots if that value is negative. (The "set of measure zero" possibility is a single root at $M_{\rm crit}$.)

If $2M_{\rm min} < M_{\rm crit} < 2M_{\rm max}$ then m_and_eta() searches for roots $2M_{\rm min} < M < M_{\rm crit}$ and $M_{\rm crit} < M < 2M_{\rm max}$ separately, else it looks for a root M in the range $2M_{\rm min} < M < 2M_{\rm max}$. If the lhs of (5.27.1) changes sign at the upper and lower boundaries of the interval, then 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 (5.24.1) to be

$$\eta = \frac{5}{256}\left(\frac{M}{M_\odot}\right)^{-5/3}(\pi f_0 T_\odot)^{-8/3}\frac{T_\odot}{\tau_0} \tag{5.27.4}$$

If $\eta \le 1/4$ then the smaller and larger masses are calculated from

$$m_1 = \frac{M}{2}\left(1-\sqrt{1-4\eta}\right) \quad m_2 = \frac{M}{2}\left(1+\sqrt{1-4\eta}\right). \tag{5.27.5}$$

(If both roots for M correspond to $\eta \le 1/4$ then an error message is generated and the routine aborts.) If both m_1 and m_2 are in the desired range $M_{\rm min} < m_1, m_2 < M_{\rm max}$ 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: None.

5.28 Function: tauspace_area()

`float tauspace_area(struct Scope *Grid)`

This function computes the area of the enclosed region of parameter space shown in Figure 32.

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 (5.24.1,5.24.2) 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

$$\begin{aligned} A_1 &= \int_{m_{\rm max}}^{m_{\rm min}} \tau_1(m_{\rm min}, m) \frac{d\tau_0(m_{\rm min}, m)}{dm} dm \\ &= A_0 \left[\frac{m_{\rm min}}{M_\odot}\right]^{8/3} \left\{ \frac{-[3+2(4+2a)u+(5+9a)u^2]}{2u^2(1+u)^{2/3}} \right. \\ &+ \frac{9a-1}{\sqrt{3}} \arctan\left[\frac{1+2(1+u)^{1/3}}{\sqrt{3}}\right] \\ &+ \left. \frac{9a-1}{6} \log\left[\frac{1+(1+u)^{1/3}+(1+u)^{2/3}}{1-2(1+u)^{1/3}+(1+u)^{2/3}}\right] \right\}_{u=m_{\rm min}/m_{\rm max}}^{u=1} . \end{aligned}$$

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 $m_{\rm min}$ and $m_{\rm max}$. (If you wish to use geomtrized units in which the solar mass is 4.92×10^{-6} 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

$$\begin{aligned} A_3 &= \int_{m_{\rm max}}^{m_{\rm min}} \tau_1(m, m) \frac{d\tau_0(m, m)}{dm} dm \\ &= \frac{60875}{2064384} \frac{M_\odot^2}{(\pi f_0 M_\odot)^{14/3}} \left[\left(\frac{M_\odot}{m_{\rm min}}\right)^{8/3} - \left(\frac{M_\odot}{m_{\rm max}}\right)^{8/3}\right] \left(\frac{c^3}{G}\right)^{8/3} . \end{aligned}$$

These three results can be combined to give the total area enclosed

$$A_{total} = A_1 + A2 - A3 . \qquad (5.28.1)$$

Equation (5.28.1) is the basis of `tauspace_area()`; the next example shows an application of this function.

Author: Alan Wiseman, agw@tapir.caltech.edu

Comments: None.

5.29 Example: area program

This example uses the function tauspace_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, Bruce Allen */
#include "grasp.h"

int main() {
        struct Scope Grid;
        float area;
        float template_area(struct Scope *);

        /* 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\n",area);
        return 0;
}
```

5.30 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 32).

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 32 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 is as follows. We work in coordinates (x_0, x_1) which are rotated versions of (τ_0, τ_1), aligned along the minor and major axis of the template ellipses. The input angle `Grid.theta` is the counterclockwise angle through which the (τ_0, τ_1) axes are rotated in order to produce the (x_0, x_1) axes.

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_o 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.

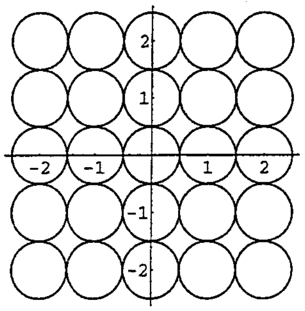


Figure 33: Covering a plane with a square lattice of pennies (or templates) leaves 21% of the area exposed

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 33. It is easy to show that the fraction of the plane (i.e., parameter space!) which is not covered by any pennies is $\epsilon = 1 - \pi/4 = 0.214\cdots$ or about 21%.

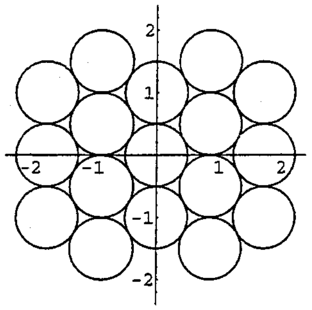


Figure 34: Staggering the pennies (or templates) decreases the uncovered fraction of the plane to 9.3%

Now suppose that we "stagger" the pennies as shown in Figure 34. In this case, the fraction of area not covered is $\epsilon = 1 - \frac{\pi}{2\sqrt{3}} = 0.093\cdots$ 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 35. By increasing the number-density of pennies on the plane by 25% we have successfully covered up the remaining 9.3% of the area.

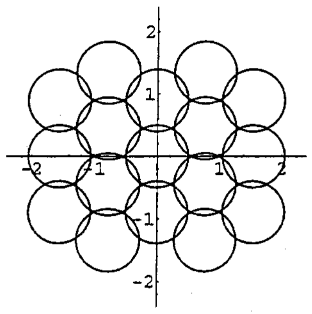


Figure 35: Decreasing the spacings of the pennies (or templates) by a factor of $(5/4)^{1/2} = 1.118\cdots$ then covers the entire plane.

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 was as for Figure 35 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.

Author	Detector	f_0/Hz	dx_0/msec	dx_1/msec	θ/rad
Sathyaprakash	Caltech 40m (Oct 94)	120	2.27	35.2	0.978
Sathyaprakash	Caltech 40m (Nov 94)	120	2.55	33.2	1.025
Sathyaprakash	Caltech 40m (Nov 94)	140	2.13	32.0	0.964
Owen	Initial LIGO	200	0.162	2.109	0.5066
Owen	Advanced LIGO	70	0.352	3.970	0.4524

Table 7: Orientation and dimensions of 0.97 ambiguity templates.

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 35 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 7 gives information about the appropriate template sizes, spacings and orientations as found in the recent literature. Note that the quantities dx_0 and dx_1 are the *radii* or semi-minor and semi-major axes of the constant-ambiguity ellipses, along the x_0 and x_1 axes as defined earlier. Equation (3.16-18) of reference [5] do not appear to agree with Table 7, but that is because the dx_i of [5] are defined by $(dx_i)_{\rm Owen} = dl_i/\sqrt{E_i}$. 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 $(dx_i)_{\rm Owen} = dl_i/\sqrt{E_i}$. In our $N = 2$ dimensional case, this gives $dx_i = (dx_i)_{\rm Owen}/\sqrt{2}$. 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 an 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.

5.31 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.

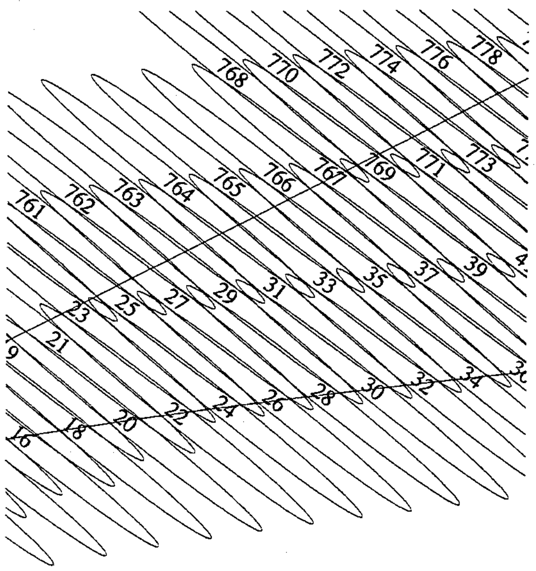


Figure 36: Part of some sample output from `plot_template()`.

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 36. (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.

5.32 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, Bruce Allen */
#include "grasp.h"

int main() {
        struct Scope Grid;

        /* Set parameters for the inspiral search */
        Grid.m_mn=0.8;
        Grid.m_mx=50.0;
        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 */
        template_grid(&Grid);

        /* create a postscript file showing locations of templates */
        plot_template("temp_list.ps",Grid,15,1);
        return 0;
}
```

Part of a typical picture contained in the output file `temp_list.ps` is shown in Figure 36 (though for different parameters than those shown above).

5.33 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 ouput 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.

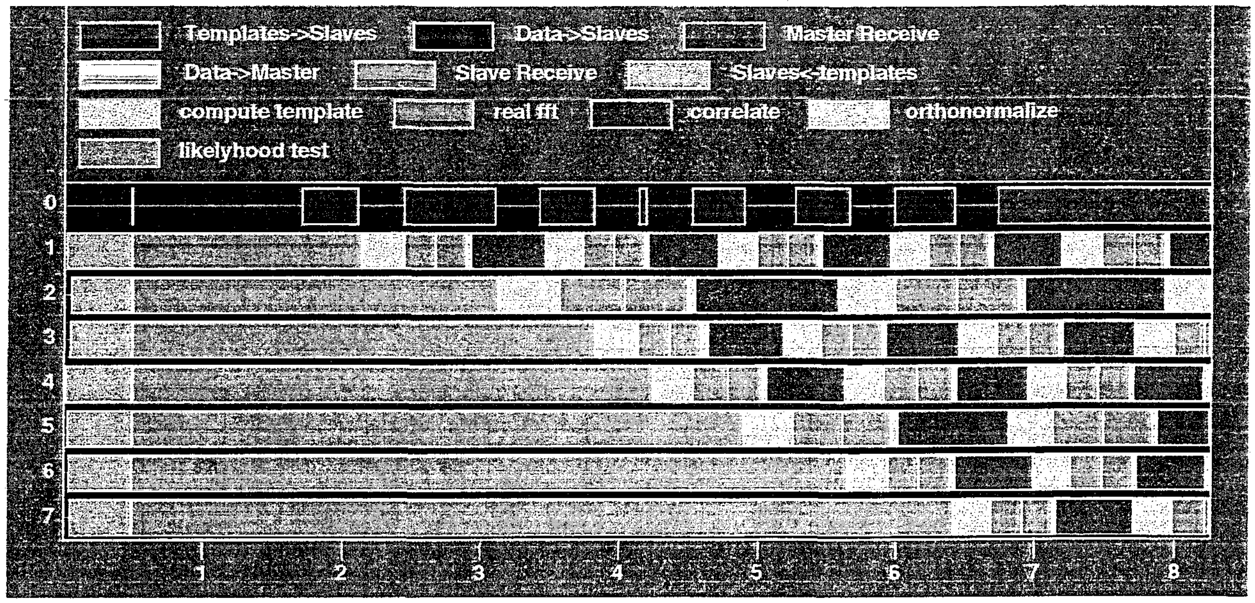


Figure 37: Output of the `nupshot` profiling tool, showing the behavior of the `multifilter` program running on a workstation network of 8 machines (the fastest of these are Sparc-20 class processors). This shows the first 8 seconds of operation (time on the horizontal axis). The gray segments show the 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.

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 8.

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.

Task	Color	Approximate time	Processing done
data $\rightarrow$ slaves	dark blue	350 msec	transfer 384 kbytes
data $\rightarrow$ master	yellow	1 msec	transfer 3 kbytes
correlate	purple	500 msec	2 ffts of 64k floats, and search
splitup (likelyhood)	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 $\approx$ 18k floats
orthonormalize templates	wheat	25 msec	several runs through 64k floats

Table 8: Approximate computation times for different elements of the optimal-filtering process.

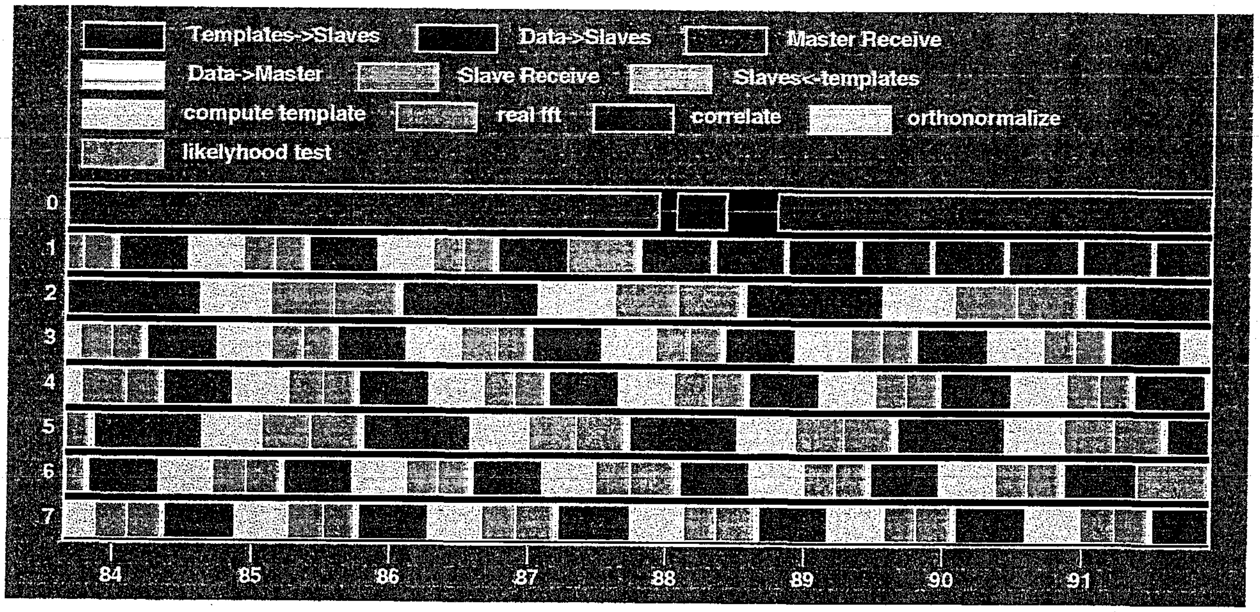


Figure 38: This is a continutation of the previous figure. Slave number 1 has completed its computation of the templates, and during the orange segment, waits to make a connection with the master. This is followed by a (very small) yellow segment, during which the slave transmits data back to the master, 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

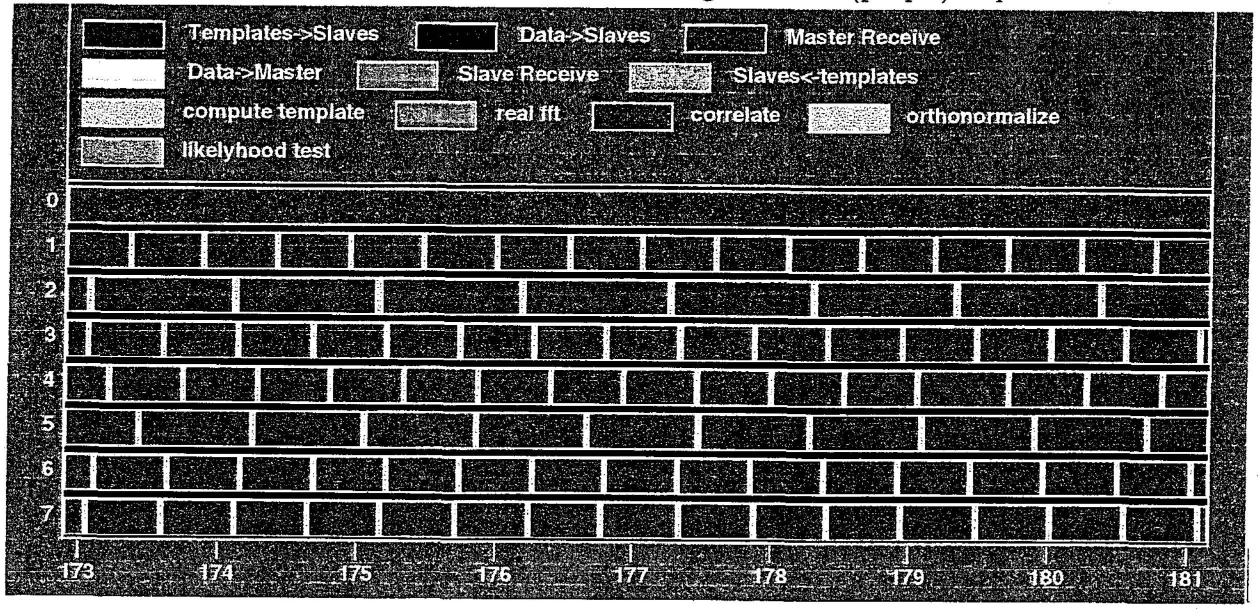


Figure 39: This is a continutation of the previous figure, and represents the "long-term" or "steady-state" behavior of the multiprocessing system. In this state, the different processors are spending all of their time doing correlation measurements of the data, as indicated by the purple segments, and the master is waiting for the results of the analysis (brown segments).

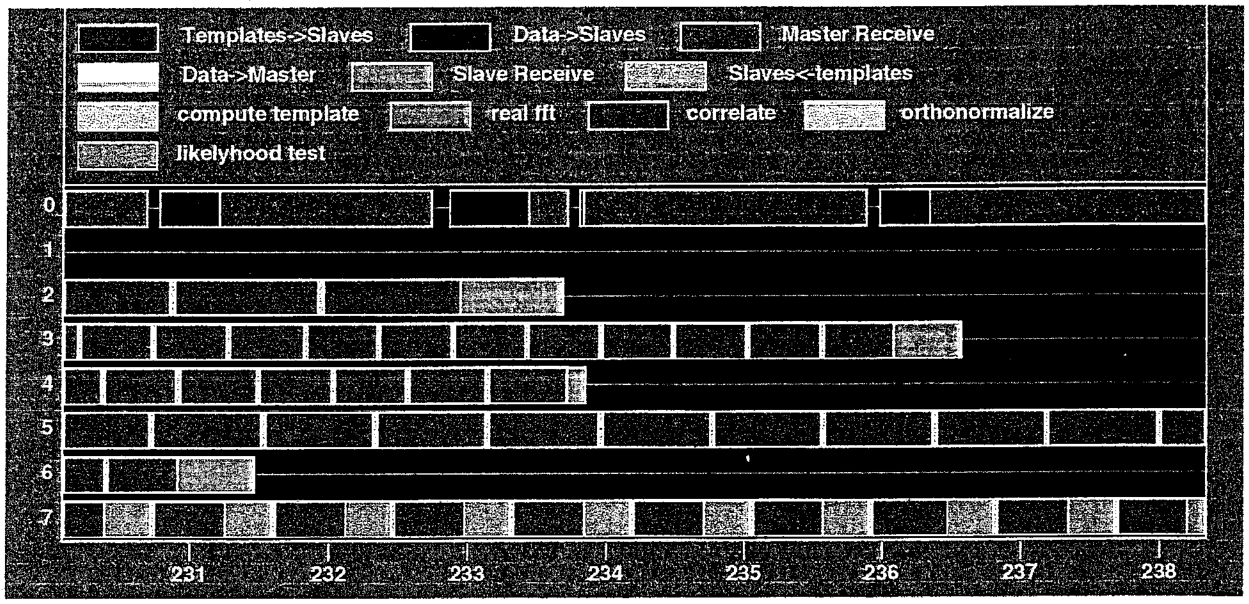


Figure 40: This is a continuation of the previous figure, and shows the termination of some of the slave processes (all the data has been analyzed, and there is no new data remaining). The blue segments (data being sent to slaves) are actually termination messages being sent to the different processes 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 likelyhood test, shown as light blue segments) on the data.

```
/* GRASP: Copyright 1997, Bruce Allen */
/* multifilter.c
This code is intended for machines where computation is cheap,
and communication is expensive. The processsing 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
coalescense 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 (6.5 secs) */
#define FLO 120.0             /* The low frequency cutoff for filtering */
#define ARMLENGTH 40.0        /* Armlength of the IFO, in meters */
#define HSCALE 1.e21          /* A convenient scaling factor; results independent of it */
#define MIN_INTO_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 1     /* 0: slaves recompute templates. 1: slaves save templates. */

void shiftdata();
void realft(float*,unsigned long,int);

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);
MPE_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");
fplock=grasp_open("GRASP_DATAPATH","channel.10");
fpss=grasp_open("GRASP_DATAPATH","swept-sine.ascii");

/* 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;

/* 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_recv) {
       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_recv++;

       /* 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);
```

```
    abort();
}

/* 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);
            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);
                abort();
            }

            /* normalize the chirp template */
            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=0.5/n0+0.5/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 */
   impulseoff=(maxi+*chirppoints)%npoint;
   timeoff=impulseoff/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]=impulseoff;
   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=gammq(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,&srate,0);
    datastart=tstart-diff/srate;

    /* 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,srate,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;
}
```

5.34 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 that the `realft()` routine.

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 41 is a timing diagram illustrating the relative computational costs of these operations.

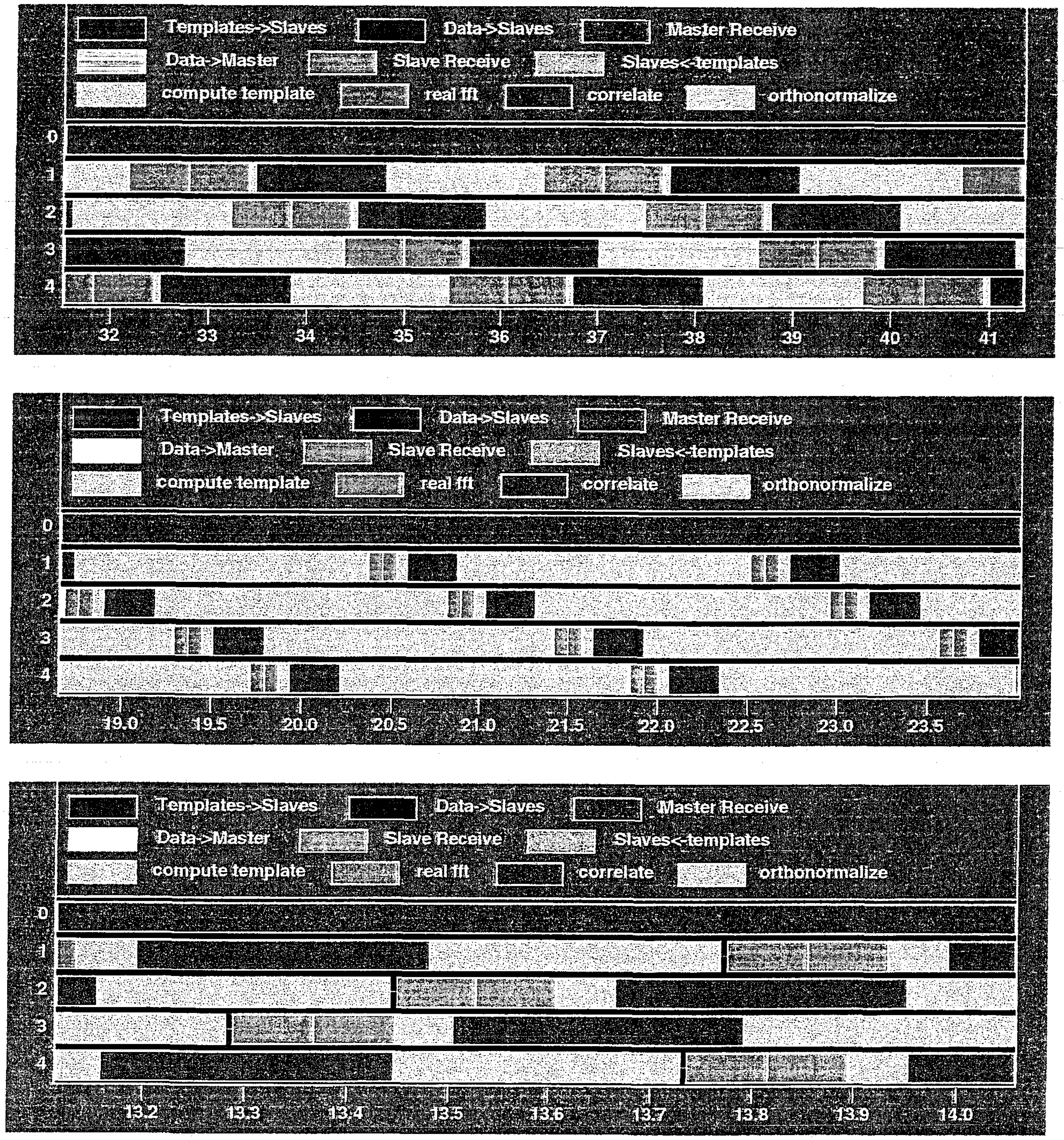


Figure 41: This shows the performance of an "on the fly" template search on the Intel Paragon, with different levels of optimization. The top diagram uses the *Numerical Recipes* FFT routine `realft()`, and takes about 4.2 seconds to process 6 seconds of data. The middle diagram shows identical code 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 5x3600x66x(0.780)/(256x6) seconds = 10.1 minutes.

6 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 5; 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.

6.1 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 [17] who was able to separate the differential equation to obtain solutions of the form

$$(r - i\mu a)^4 \Psi_4 = e^{-i\omega t} {}_{-2}R_{\ell m}(r) {}_{-2}S_{\ell m}(\mu) e^{im\beta} \tag{6.1.1}$$

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 [17], 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*, $\hat{a} = c^2a/GM = c^3J/GM^2$. 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 42). 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.

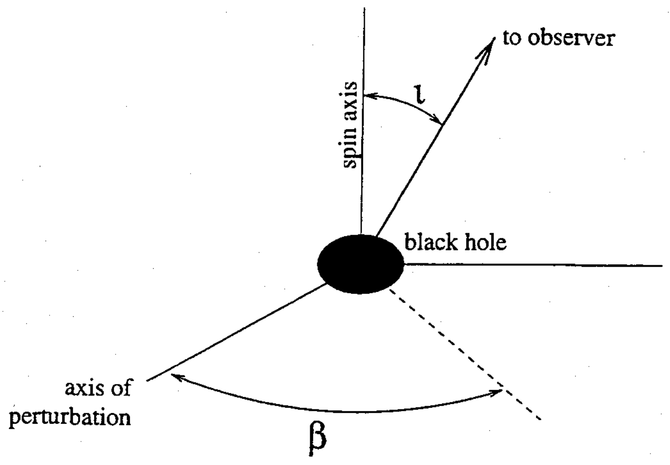


Figure 42: The polar angle, ι, and the azimuthal angle, β, of the observer relative to the spin axis of a black hole and the (somewhat arbitrary) axis of perturbation.

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 "×" polarizations of the strain induced by the gravity waves are found by [17]

$$h_+ - ih_\times = -\frac{2c^2}{|\omega|^2}\Psi_4 \,. \tag{6.1.2}$$

The quantity $h_+ = h_{\hat{\iota}\hat{\iota}}$ is the metric perturbation that represents the linear polarization state along $\mathbf{e}_{\hat{\iota}}$ and $\mathbf{e}_{\hat{\beta}}$, while the quantity $h_\times = h_{\hat{\iota}\hat{\beta}}$ represents the linear polarization state along $\mathbf{e}_{\hat{\iota}} \pm \mathbf{e}_{\hat{\beta}}$. The power radiated towards the observer (per unit solid angle) is

$$\frac{d^2E}{dt\,d\Omega} = \lim_{r\to\infty} \frac{c^7r^2}{4\pi G|\omega|^2}|\Psi_4|^2 \,. \tag{6.1.3}$$

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 quasinormal 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:

$$\Psi_4 \approx \frac{A}{r} e^{-i\omega t_{\rm ret}} {}_{-2}S_{\ell m}(\mu) e^{im\beta}. \tag{6.1.4}$$

Here $t_{\rm ret} = t - r^\star/c$ represents the retarded time, where $r^\star$ 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 6.1.4, we can integrate equation 6.1.3 over the entire sphere and the interval $t_{\rm ret} \in [0, \infty)$ 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: $A^2 = 4Gc^{-7}E|\omega|^2(-\mathrm{Im}\,\omega)$. The gravitational waveform is found to be

$$h_+ - ih_\times \approx -\frac{4c}{r}\left(\frac{-\mathrm{Im}\,\omega}{|\omega|^2}\right)^{1/2}\left(\frac{GE}{c^5}\right)^{1/2} e^{-i\omega t_{\rm ret}} {}_{-2}S_{\ell m}(\mu) e^{im\beta}. \tag{6.1.5}$$

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 $H_+ - iH_\times = (c^2r/GM_\odot)(h_+ - ih_\times)$; the normalization of these waveforms to the correct source distance is left to the calling routine. The distance normalization can be computed as follows:

$$\frac{c^2 r}{GM_\odot} = \frac{r}{T_\odot c} = \left(\frac{r}{1.4766\,\mathrm{km}}\right) = 2.090 \times 10^{19}\left(\frac{r}{\mathrm{Mpc}}\right). \tag{6.1.6}$$

where $T_\odot = 4.89128\,\mu$s is the mass of the sun expressed in seconds (see equation 5.0.2). It will be convenient to write the time dependence of the strain as the complex function $\mathcal{H}(\sqcup_{\rm ret})$ so that $H_+ - iH_\times = \mathcal{H}(\sqcup_{\rm ret})_{-\in}\mathcal{S}_{\ell\Updownarrow}(\mu)]^{\rangle\Updownarrow\beta}$. 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 $\mathcal{H}(\sqcup_{\rm ret})$ is

$$\mathcal{H}(\sqcup_{\rm ret}) \approx -\triangle\epsilon^{\infty/\in}\frac{(-\mathrm{Im}\,\hat\omega)^{1/2}}{|\text{⩋}|}\left(\frac{\mathcal{M}}{\mathcal{M}_\odot}\right)\exp\left[-\rangle\text{⩋}\left(\frac{\sqcup_{\rm ret}}{\mathcal{T}_\odot}\right)\left(\frac{\mathcal{M}}{\mathcal{M}_\odot}\right)^{-\infty}\right] \tag{6.1.7}$$

where ϵ is the fractional mass loss due to the radiation in the excited quasinormal mode.

6.2 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]` $= \mathrm{Re}\,\hat{\omega}$, `eigenvalues[1]` $= \mathrm{Im}\,\hat{\omega}$, `eigenvalues[2]` $= \mathrm{Re}\,\mathrm{A}$, and `eigenvalues[3]` $= \mathrm{Im}\,\mathrm{A}$.

`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 42).

`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 [14]. 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 [17], are not allowed.

6.3 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 6.2). 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 43. The blue curves in figure 43 can be compared to figure 5 of reference [15] keeping in mind the conversion factors between Leaver's convention (which is also used in [15]) and the convention used here (see subsection 6.2).

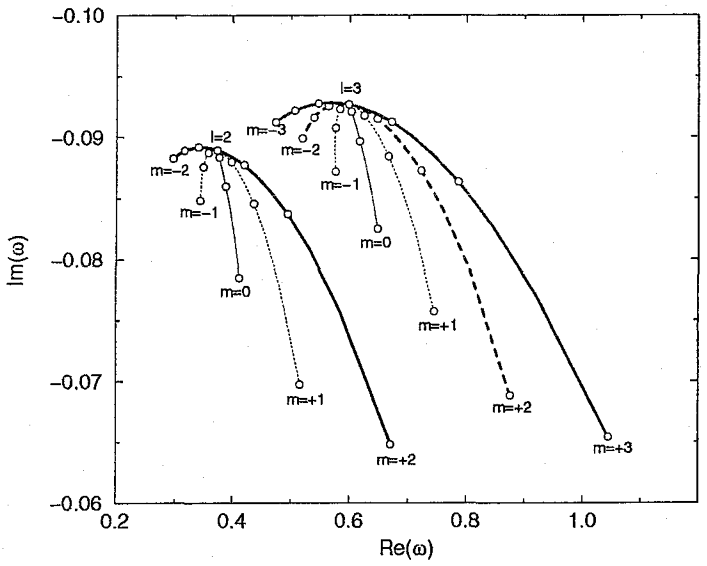


Figure 43: The real and imaginary parts of the eigenfrequencies, $\hat{\omega}$, as computed by the program **`eigenvalues`** with $s = -2$. Each curve corresponds to a range of values of $\hat{a}$ from -0.9 (left) to $+0.9$ (right) for a single mode ℓ and $|m|$. The open circles are placed at the values $\hat{a} = -0.9$, -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, Bruce Allen */
#include "grasp.h"

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\t%f\n",a,eigen[0],eigen[1],eigen[2],eigen[3]);
  }

  return 0;
}
```

Author: Jolien Creighton, jolien@tapir.caltech.edu

6.4 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; $-1 < \mathtt{mu} < 1$.

`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, $-1 < \mathtt{a} < 1$, 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 $\mathtt{l} \geq |\mathtt{s}|$.

`m`: Input. The mode integer $|\mathtt{m}| \leq \mathtt{l}$.

`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]` $= \mathrm{Re}\,\hat{\omega}$, `eigenvalues[1]` $= \mathrm{Im}\,\hat{\omega}$, `eigenvalues[2]` $= \mathrm{Re}\,\mathrm{A}$, and `eigenvalues[3]` $= \mathrm{Im}\,\mathrm{A}$. 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 [14]. 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 $\mu \to -1$ in order to agree with the sign of the spin-weighted spherical harmonics ${}_sY_{\ell m}(\mu, 0)$ (see [13]).

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 [16]:

$$ {}_{-s}S_{\ell m}(\mu, a\omega) = {}_sS_{\ell m}(-\mu, a\omega) \quad \text{with} \quad {}_{-s}\mathrm{E}_{\ell m}(a\omega) = {}_s\mathrm{E}_{\ell m}(a\omega) \tag{6.4.1} $$

and

$$ {}_sS_{\ell m}(\mu, -a\omega) = {}_sS_{\ell,-m}(-\mu, a\omega) \quad \text{with} \quad {}_s\mathrm{E}_{\ell m}(-a\omega) = {}_s\mathrm{E}_{\ell,-m}(a\omega) \tag{6.4.2} $$

where ${}_sE_{\ell m} = {}_sA_{\ell m} + s(s+1)$.

Author: Jolien Creighton, jolien@tapir.caltech.edu

6.5 Example: `spherical` program

The program `spherical` is an example implementation of the routine `sw_spheroid()` to compute the standard spin-weighted spherical harmonics [13]. The program also computes these functions using equation (3.1) of [13] for comparison. According to the normalization convention stated in subsection 6.4, the relationship between the spin-weighted spheroidal harmonics and the spin-weighted spherical harmonics is:

$$_sY_{\ell m}(\theta,\phi) = (2\pi)^{-1/2}{}_sS_{\ell m}(\cos\theta)e^{im\phi} \tag{6.5.1}$$

with $a\omega = 0$ and $A = (\ell - s)(\ell + s + 1)$.

To invoke the program, type:

`spherical` $s\ \ell\ m$

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 [13]. 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 9.

μ	Goldberg	`sw_spheroid()`	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 9: A comparison of the values of the spin-weighted spherical harmonic $_{-2}Y_{22}(\mu,0)$ calculated by equation (3.1) of Goldberg [13], 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, 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;
}


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] = (l - s)*(l + 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

6.6 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` $s\ \ell\ m$

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 44 depicts the output for the spin-weighted spheroidal wave function ${}_{-2}S_{22}(\mu)$.

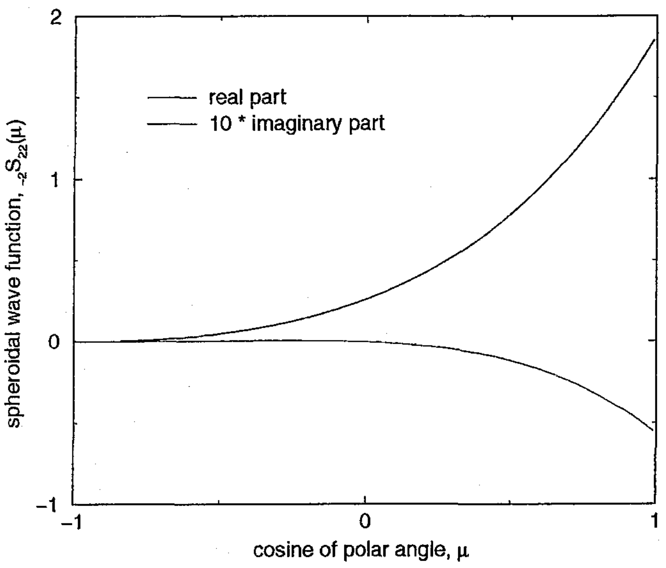


Figure 44: A plot of the real and imaginary parts of the $\ell = m = -s = 2$ spin-weighted spheroidal wave function, ${}_{-2}S_{22}(\mu)$, associated with a black hole with dimensionless angular momentum parameter $\hat{a} = 0.98$. The imaginary part is scaled by a factor of ten.

```
/* GRASP: Copyright 1997, Bruce Allen */

#include "grasp.h"

#define SPIN 0.98 /* the dimensionless angular momentum parameter */

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,1,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

6.7 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 $GM_\odot/c^2 = T_\odot c \simeq 1.4766\,\text{km}$. To obtain the waveforms at a distance r, multiply the result by $GM_\odot/c^2r = T_\odot c/r$. 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, $0 \leq$ `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. ($0 \leq$ `beta` $\leq 2\pi$.)

`eps`: Input. The fraction of the total mass lost in gravitational radiation from the particular mode. ($0 <$ `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 42).

`l`: Input. The mode integer ℓ. (`l` ≥ 2)

`m`: Input. The mode integer m. ($|$`m`$| \leq$ `l`)

`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 $A_{\text{cutoff}} = A_0\,\text{alog}_{10}(-0.1 \times$ `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 $H_+ - iH_\times = \mathcal{H}(\sqcup_{\text{ret}})_{-\in}\mathcal{S}_{\ell\Updownarrow}(\mu)\rceil^{\rangle\Updownarrow\beta}$ with $\mathcal{H}(\sqcup_{\text{ret}})$ given by equation (6.1.7). The spheroidal wave function is obtained from the function `sw_spheroid()`.

Author: Jolien Creighton, jolien@tapir.caltech.edu

6.8 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 45.

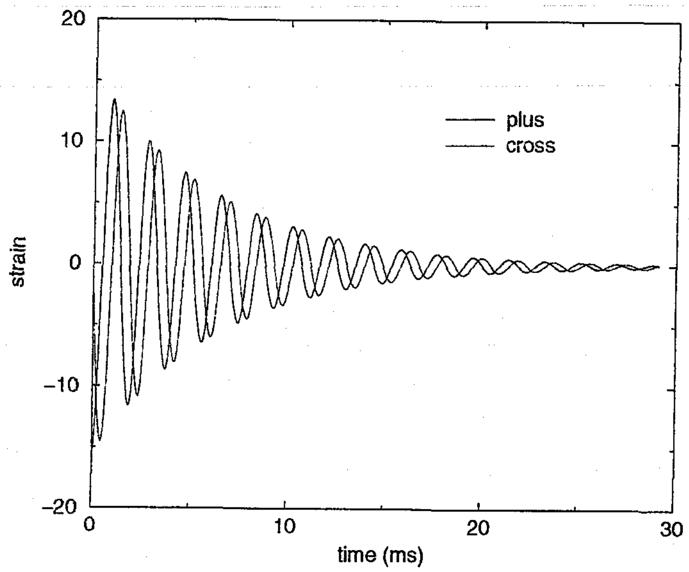


Figure 45: A plot of the plus and cross polarizations of the gravitational wave strain, at a (unphysical!) distance $GM_\odot/c^2 = T_\odot c \simeq 1.4766\,\mathrm{km}$, for the fundamental $\ell = m = 2$ mode of a black hole with mass $M = 50M_\odot$, dimensionless angular momentum parameter 0.98, 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, 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 leven (dB) */
#define MAX 1024         /* max number of points in waveform */

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

6.9 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 $GM_\odot/c^2 = T_\odot c \simeq 1.4766\,\mathrm{km}$. 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. ($0 <$ 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 42).

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_ring() 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*

$$H_{\mathrm{ave}}(t_{\mathrm{ret}}) = \frac{1}{\sqrt{5}}\mathrm{Re}\,[\mathcal{H}(\sqcup_{\mathrm{ret}})]^{\rangle\psi_\prime}], \tag{6.9.1}$$

where $\mathcal{H}(\sqcup_{\mathrm{ret}})$ is given by equation (6.1.7), 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 $GM_\odot/c^2 = T_\odot c \simeq 1.4766\,\mathrm{km}$) that would be observed by an interferometer is given by $H(t_{\mathrm{ret}}) = F_+(\theta,\phi,\psi)H_+(t_{\mathrm{ret}},\iota,\beta) + F_\times(\theta,\phi,\psi)H_\times(t_{\mathrm{ret}},\iota,\beta)$ 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,

$$\begin{aligned}
\langle H^2(t_{\mathrm{ret}})\rangle_{\theta,\phi,\psi,\iota,\beta} &= \tfrac{1}{5}\langle H_+^2(t_{\mathrm{ret}},\iota,\beta) + H_\times^2(t_{\mathrm{ret}},\iota,\beta)\rangle_{\iota,\beta} \\
&= \tfrac{1}{5}\langle|(H_+ - iH_\times)(t_{\mathrm{ret}},\iota,\beta)|^2\rangle_{\iota,\beta} \\
&= \tfrac{1}{10}|\mathcal{H}(\sqcup_{\mathrm{ret}})|^{\in} \\
&\approx \overline{H_{\mathrm{ave}}^2}
\end{aligned} \tag{6.9.2}$$

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 $H_{\rm ave}(t_{\rm ret})$ 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 [12]. These expressions are:

$$\hat{\omega} \simeq f(\hat{a})(1 - \tfrac{1}{4} i g(\hat{a})) \tag{6.9.3}$$

with

$$\begin{aligned} f(\hat{a}) &= 1 - 0.63(1-\hat{a})^{3/10} && (6.9.4)\\ g(\hat{a}) &= (1-\hat{a})^{9/20}. && (6.9.5) \end{aligned}$$

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.

6.10 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

$$q(t) = e^{-\pi f t/Q} \cos(2\pi f t) \tag{6.10.1}$$

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

6.11 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 $1 = \langle N^2 \rangle \to \frac{1}{2}$`correlate(...,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 (5.10.9). Thus, the normalization constant is given by

$$\frac{1}{\alpha^2} = \frac{1}{2}(q, q). \tag{6.11.1}$$

Author: Jolien Creighton, jolien@tapir.caltech.edu

6.12 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 5.14.

`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 5.14.

`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

6.13 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()`.

6.14 Vetoing techniques for ringdown waveforms

Vetoing techniques for binary inspirals have already been described in subsection 5.18; 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.

6.15 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, Bruce Allen */

#include "grasp.h"

#define NPOINT 131072          /* number of data points */
#define HSCALE 1.0e21          /* convenient scaling factor */
#define ARMLENGTH 40.0         /* armlength (meters) */
#define FLO 120.0              /* low frequency cutoff for filtering */
#define MIN_INTO_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[NPOINT+2],srate=9868.4208984375;
short datas[NPOINT];
int needed=NPOINT;

main()
{
  void realft(float *, unsigned long, int);

  double norm;
  float data[NPOINT],htilde[NPOINT],output[NPOINT];
  float mean_pow_spec[NPOINT/2+1],twice_inv_noise[NPOINT/2+1];
  float *ring,ringtilde[NPOINT],template[NPOINT];
  float decaytime,decay,scale,snr,mean,var,tmpl_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;

  /* 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");
    fplock = grasp_open("GRASP_DATAPATH","channel.10");
    fpss = grasp_open("GRASP_DATAPATH","swept-sine.ascii");
    /* 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;
    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_INTO_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;
}
```

6.16 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.

6.17 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 [12].

6.18 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 $d\ell = \surd(2ds^2_{\text{threshold}})$ (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 hilight 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 [11]. 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 $ds^2_{ij} = 1 - (u_i|u_j)$ 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

$$\begin{aligned} ds^2 &= \frac{1}{8}\left\{\frac{3+16Q^4}{Q^2(1+4Q^2)^2}\,dQ^2 - 2\frac{3+4Q^2}{fQ(1+4Q^2)}\,dQ\,df + \frac{3+8Q^2}{f^2}\,df^2\right\} &(6.18.1)\\ &\approx \frac{1}{8}\frac{dQ^2}{Q^2} - \frac{1}{4}\frac{dQ}{Q}\frac{df}{f} + Q^2\frac{df^2}{f^2}. &(6.18.2)\end{aligned}$$

In the approximate metric of equation (6.18.2), 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 $ds^2_{\text{threshold}}$ from the nearest template can be found by integrating the volume element $\surd\det g_{\alpha\beta}$ over the parameter space. Using the approximate metric and the parameter ranges $Q \le Q_{\max}$ and $f \in [f_{\min}, f_{\max}]$, we find that the number of templates required is

$$\begin{aligned} \mathcal{N} &\approx \frac{1}{4\surd 2}(ds^2_{\text{threshold}})^{-1}Q_{\max}\log(f_{\max}/f_{\min}) \\ &\simeq 2700\left(\frac{ds^2_{\text{threshold}}}{0.03}\right)^{-1}\left(\frac{Q_{\max}}{100}\right)\left\{1 + \frac{1}{\log 100}\left[\log\left(\frac{f_{\max}}{10\,\text{kHz}}\right) - \log\left(\frac{f_{\min}}{100\,\text{Hz}}\right)\right]\right\}. &(6.18.3)\end{aligned}$$

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 $Q = Q_{\mathrm{min}}$, 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, $d\ell = \surd(2ds^2_{\mathrm{threshold}})$. 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 $ds^2_{\mathrm{threshold}}$ from the nearest template.

Author: Jolien Creighton, jolien@tapir.caltech.edu

7 GRASP Routines: Stochastic background detection

7.1 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.

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. 7.3 and 7.4.) 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 Hanford-initial noise_init.dat whiten_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 Livingston-initial noise_init.dat whiten_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 XXXXX XXXXX
#
# GEO-600 as of April 1995
# Albrecht Ruediger atr@mpq.mpg.de
4 52.2467 -9.82167 26.0 292.5 6.e4 GEO-600 XXXXX XXXXX
#
# Garching 30 Meter Interferometer
# Albrecht Ruediger atr@mpq.mpg.de
5 48.244 -11.675 329.0 239.0 3.e3 Garching-30 XXXXX XXXXX
#
# Glasgow 10 Meter Interferometer
# Albrecht Ruediger atr@mpq.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 40noise.dat 40whiten.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 XXXXX XXXXX
#
# 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 Hanford-advanced noise_adv.dat whiten_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 Livingston-advanced noise_adv.dat whiten_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.

7.2 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 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. 7.1. 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 value of `site_choice` should be chosen to match the site identification number for one of the detectors contained in this file.

`site_parameters`: Output. `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).

`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. 7.3 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. 7.4 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 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(). 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. [21].) 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.

7.3 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. 7.1, 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. 7.4.)

`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 strain^2/Hz (or seconds). `power[i]` contains the value of $P(f)$ evaluated at the discrete frequency $f_i = i\Delta f$, where $i = 0, 1, \cdots, N-1$.

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. [20]:

$$\langle \tilde{n}^*(f)\tilde{n}(f')\rangle =: \frac{1}{2}\delta(f-f')\, P(f)\ . \tag{7.3.1}$$

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 strain^2/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 $0.0, 0.1, 0.2, \cdots, 0.9$ Hz equal to its 1 Hz value seems to be sufficient. (See Sec. 7.4 for a similar comment regarding whiten().)

7.4 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. 7.1 and 7.3, 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 $\sec^{-1/2}$), 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 $i = 0, 1, \cdots, N-1$.

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 $\sec^{-1/2}$). 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. 7.3), `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 $0.0, 0.1, 0.2, \cdots, 0.9$ Hz equal to their 1 Hz values seems to be sufficient. (See Sec. 7.3 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.

7.5 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 $i = 0, 1, \cdots, N-1$.

The values of $\gamma(f)$ calculated by `overlap()` are defined by equation (3.9) of Ref. [20]:

$$\gamma(f) := \frac{5}{8\pi} \int_{S^2} d\hat{\Omega}\, e^{2\pi i f \hat{\Omega}\cdot\Delta\vec{x}/c} \left(F_1^+ F_2^+ + F_1^\times F_2^\times \right) . \tag{7.5.1}$$

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

$$F_1^{+,\times} = \frac{1}{2} \left(\hat{X}_1^a \hat{X}_1^b - \hat{Y}_1^a \hat{Y}_1^b \right) e_{ab}^{+,\times}(\hat{\Omega}) , \tag{7.5.2}$$

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.

7.6 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 $i = 0, 1, \cdots, N-1$. 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. 46.)

```
/* 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 500                          /* number of frequency points */
#define DELTA_F 1.0                   /* frequency spacing (in Hz) */
#define OUT_FILE "LIGO_overlap.dat"   /* output filename */

main()
{
  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;
}
```

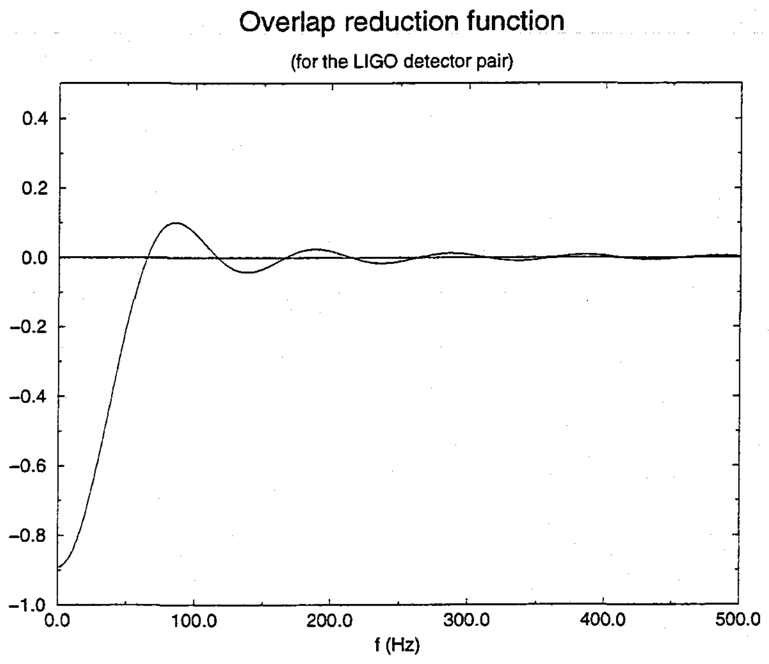


Figure 46: The overlap reduction function $\gamma(f)$ for the Hanford, WA and Livingston, LA LIGO detector pair.

7.7 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 $o_1(t)$ evaluted at the discrete time $t_i = i\Delta t_1$, where $i = 0, 1, \cdots, N-1$ 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.

7.8 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 strain^2/Hz (or seconds). `power[i]` contains the value of $P(f)$ evaluated at the discrete frequency $f_i = i/(N\Delta t)$, where $i = 0, 1, \cdots, N/2 - 1$.

`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 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 $i = 0, 1, \cdots, N/2 - 1$.

`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 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 $i = 0, 1, \cdots, N - 1$.

`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:

$$\langle \tilde{n}^*(f_i)\tilde{n}(f_j)\rangle = \frac{1}{2}\ T\ \delta_{ij}\ P(f_i)\ , \tag{7.8.1}$$

where $\langle\ \rangle$ denotes ensemble average. The above equation is just the discrete frequency version of Eq. (7.3.1). This equation can be realized by setting

$$\tilde{n}(f_i) = \frac{1}{2}\sqrt{T}\ P^{1/2}(f_i)\ (u_i + iv_i)\ , \tag{7.8.2}$$

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:

$$\tilde{o}(f_i) := \tilde{n}(f_i)\ \tilde{W}(f_i)\ . \tag{7.8.3}$$

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 $i = 0, 1, \cdots, N-1$.

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.

7.9 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: $\Omega_{\rm gw}(f) = \Omega_0$ for $f_{\rm low} \le f \le f_{\rm high}$. 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 $\Omega_{\rm gw}(f)$ for the stochastic background:

$$\Omega_{\rm gw}(f) = \begin{cases} \Omega_0 & f_{\rm low} \le f \le f_{\rm high} \\ 0 & \text{otherwise.} \end{cases}$$

Ω_0 should be greater than or equal to zero.

`f_low`: Input. The frequency $f_{\rm low}$ (in Hz) below which the spectrum $\Omega_{\rm gw}(f)$ of the stochastic background is zero. $f_{\rm low}$ should lie in the range $0 \le f_{\rm low} \le f_{\rm Nyquist}$, where $f_{\rm Nyquist}$ is the Nyquist critical frequency. (The Nyquist critical frequency is defined by $f_{\rm Nyquist} := 1/(2\Delta t)$, where Δt is the sampling period of the detectors.) $f_{\rm low}$ should also be less than or equal to $f_{\rm high}$.

`f_high`: Input. The frequency $f_{\rm high}$ (in Hz) above which the spectrum $\Omega_{\rm gw}(f)$ of the stochastic background is zero. $f_{\rm high}$ should lie in the range $0 \le f_{\rm high} \le f_{\rm Nyquist}$. It should also be greater than or equal to $f_{\rm low}$.

`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 $i = 0, 1, \cdots, N/2-1$.

`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 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 $i = 0, 1, \cdots, N/2-1$.

`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 $\sec^{-1/2}$), 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 $i = 0, 1, \cdots, N-1$.

`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 $\Omega_{\rm gw}(f) = \Omega_0$ for $f_{\rm low} \le f \le f_{\rm high}$ 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:

$$\begin{aligned} \langle \tilde{h}_1^*(f_i)\tilde{h}_1(f_j)\rangle &= \frac{1}{2}\,T\,\delta_{ij}\,\frac{3H_0^2}{10\pi^2}\,f_i^{-3}\,\Omega_0 && (7.9.1)\\ \langle \tilde{h}_2^*(f_i)\tilde{h}_2(f_j)\rangle &= \frac{1}{2}\,T\,\delta_{ij}\,\frac{3H_0^2}{10\pi^2}\,f_i^{-3}\,\Omega_0 && (7.9.2)\\ \langle \tilde{h}_1^*(f_i)\tilde{h}_2(f_j)\rangle &= \frac{1}{2}\,T\,\delta_{ij}\,\frac{3H_0^2}{10\pi^2}\,f_i^{-3}\,\Omega_0\,\gamma(f_i)\,, && (7.9.3)\end{aligned}$$

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. [20], with $\Omega_{\rm gw}(f) = \Omega_0$ for $f_{\rm low} \le f \le f_{\rm high}$. They can be realized by setting

$$\begin{aligned} \tilde{h}_1(f_i) &= \frac{1}{2}\sqrt{T}\left(\frac{3H_0^2}{10\pi^2}\right)^{1/2} f_i^{-3/2}\,\Omega_0^{1/2}\,(x_{1i}+iy_{1i}) && (7.9.4)\\ \tilde{h}_2(f_i) &= \tilde{h}_1(f_i)\,\gamma(f_i) + && (7.9.5)\\ & \frac{1}{2}\sqrt{T}\left(\frac{3H_0^2}{10\pi^2}\right)^{1/2} f_i^{-3/2}\,\Omega_0^{1/2}\,\sqrt{1-\gamma^2(f_i)}\,(x_{2i}+iy_{2i})\,, && (7.9.6)\end{aligned}$$

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. 7.8.) 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 (7.9.2). 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:

$$\begin{aligned} \tilde{o}_1(f_i) &:= \tilde{h}_1(f_i)\,\tilde{W}_1(f_i) && (7.9.7)\\ \tilde{o}_2(f_i) &:= \tilde{h}_2(f_i)\,\tilde{W}_2(f_i)\,. && (7.9.8)\end{aligned}$$

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 $i = 0, 1, \cdots, N-1$. 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. 7.8.)

7.10 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 $1 \le \texttt{which} \le 16$.

`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:

$$\begin{aligned}
&\texttt{buf[i]}+=\texttt{sin[i}*\texttt{M\_PI/n]}*\texttt{in1[i]} \quad \text{for} \quad 0 \le \texttt{i} \le \texttt{n/2}-1\\
&\texttt{buf[i]}+=\texttt{sin[i}*\texttt{M\_PI/n]}*\texttt{in1[i]}+\texttt{sin[(i}-\texttt{n/2)}*\texttt{M\_PI/n]}*\texttt{in2[i}-\texttt{n/2]} \quad \text{for} \quad \texttt{n/2} \le \texttt{i} \le \texttt{n}-1\\
&\texttt{buf[i]}+=\texttt{sin[(i}-\texttt{n/2)}*\texttt{M\_PI/n]}*\texttt{in2[i}-\texttt{n/2]} \quad \text{for} \quad \texttt{n} \le \texttt{i} \le 3*\texttt{n/2}-1\ .
\end{aligned}$$

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

$$\texttt{sin}^2\texttt{[i}*\texttt{M\_PI/n]}+\texttt{sin}^2\texttt{[(i}-\texttt{n/2)}*\texttt{M\_PI/n]}=1\ . \tag{7.10.1}$$

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.

7.11 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: $\Omega_{\rm gw}(f) = \Omega_0$ for $f_{\rm low} \leq f \leq f_{\rm high}$. 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 $\Omega_{\rm gw}(f)$ for the stochastic background:

$$\Omega_{\rm gw}(f) = \begin{cases} \Omega_0 & f_{\rm low} \leq f \leq f_{\rm high} \\ 0 & \text{otherwise.} \end{cases}$$

Ω_0 should be greater than or equal to zero.

`f_low`: Input. The frequency $f_{\rm low}$ (in Hz) below which the spectrum $\Omega_{\rm gw}(f)$ of the stochastic background is zero. $f_{\rm low}$ should lie in the range $0 \leq f_{\rm low} \leq f_{\rm Nyquist}$, where $f_{\rm Nyquist}$ is the Nyquist critical frequency. (The Nyquist critical frequency is defined by $f_{\rm Nyquist} := 1/(2\Delta t)$, where Δt is the sampling period of the detector.) $f_{\rm low}$ should also be less than or equal to $f_{\rm high}$.

`f_high`: Input. The frequency $f_{\rm high}$ (in Hz) above which the spectrum $\Omega_{\rm gw}(f)$ of the stochastic background is zero. $f_{\rm high}$ should lie in the range $0 \leq f_{\rm high} \leq f_{\rm Nyquist}$. It should also be greater than or equal to $f_{\rm low}$.

`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 $i = 0, 1, \cdots, N/2-1$.

`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 strain^2/Hz (or seconds). `power1[i]` contains the value of $P_1(f)$ evaluated at the discrete frequency $f_i = i/(N\Delta t)$, where $i = 0, 1, \cdots, N/2-1$.

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 $\sec^{-1/2}$), 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 $i = 0, 1, \cdots, N/2 - 1$.

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 $\sec^{-1/2}$), 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 $i = 0, 1, \cdots, N - 1$.

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.

7.12 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: $\Omega_{\rm gw}(f) = \Omega_0$ for $f_{\rm low} \le f \le f_{\rm high}$. 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 $\Omega_{\rm gw}(f) = 1.0 \times 10^{-3}$ for 5 Hz $\le f \le$ 5000 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. 47.

By modifying the parameters listed at the top of the example program, one can also simulate an unwhitened stochastic background signal (Fig. 48), and whitened and unwhitened data streams corresponding to the noise intrinsic to an initial LIGO detector (Figs. 49 and 50). 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.

```
/* 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 */

main()
{
  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;
```

```
}

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;
}
```

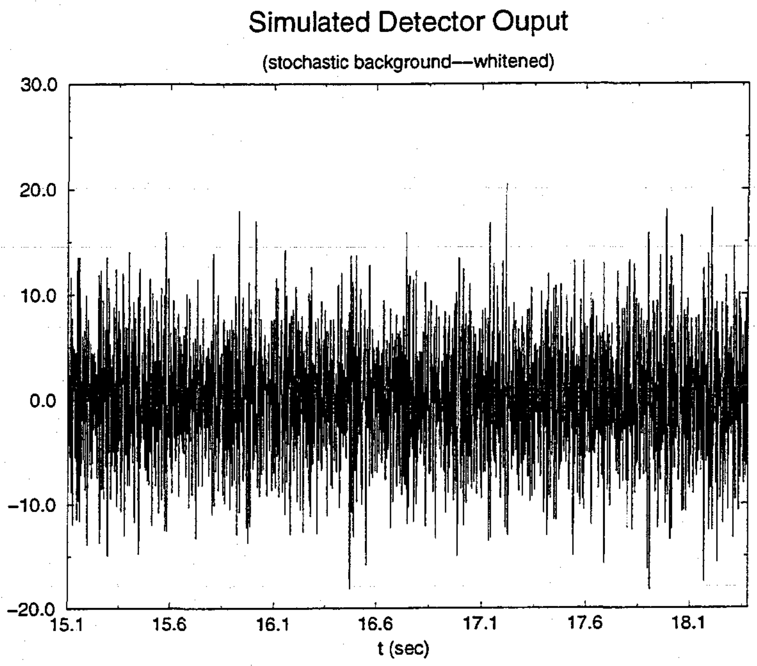


Figure 47: Time-series data (whitened) for a stochastic background having a constant frequency spectrum: $\Omega_{\mathrm{gw}}(f) = 1.0 \times 10^{-3}$ for 5 Hz $\leq f \leq$ 5000 Hz.

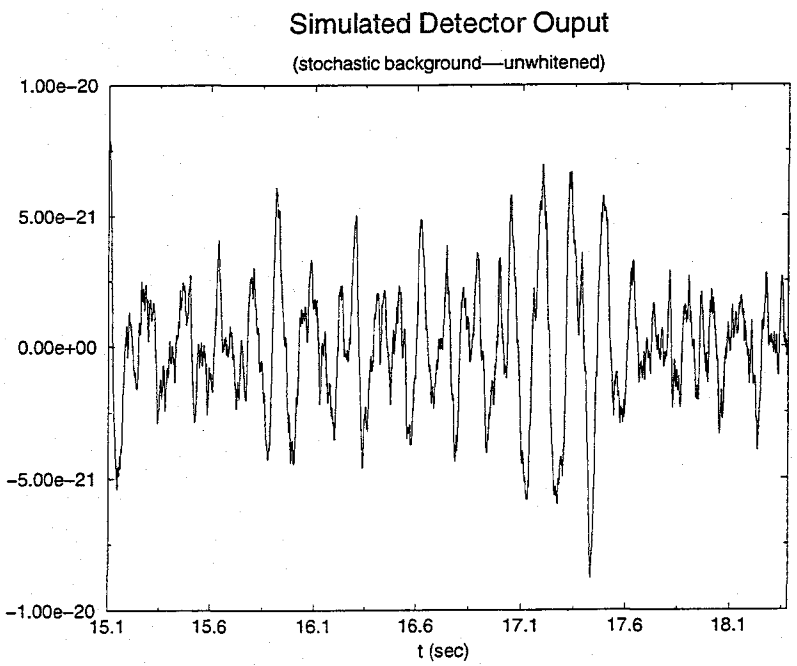


Figure 48: Time-series data (unwhitened) for a stochastic background having a constant frequency spectrum: $\Omega_{\mathrm{gw}}(f) = 1.0 \times 10^{-3}$ for 5 Hz $\leq f \leq$ 5000 Hz.

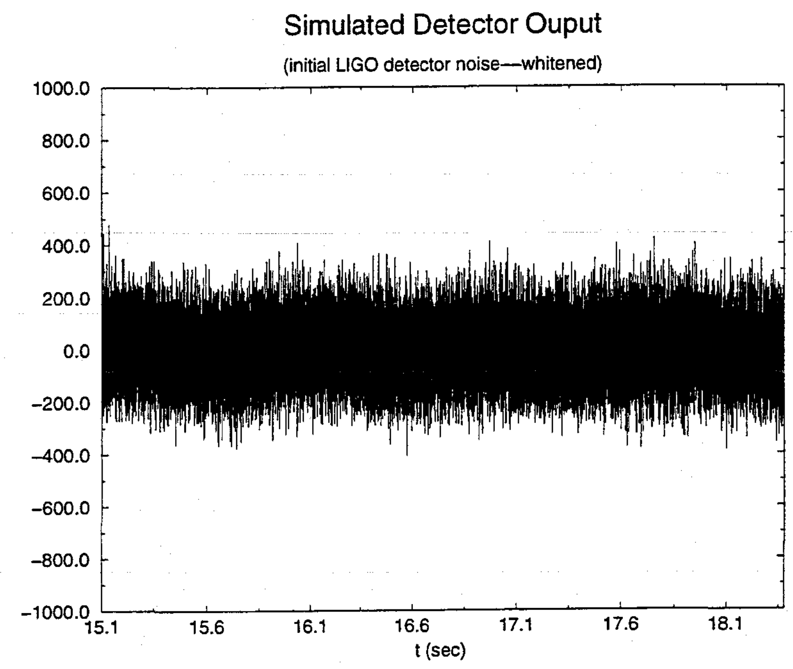


Figure 49: Time-series data (whitened) for the noise intrinsic to an initial LIGO detector.

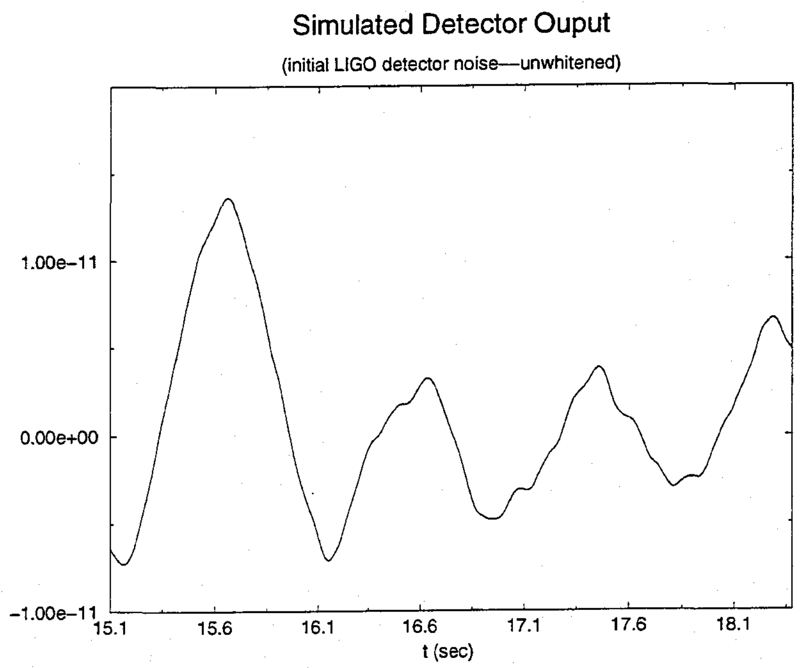


Figure 50: Time-series data (unwhitened) for the noise intrinsic to an initial LIGO detector.

7.13 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. 10.4 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.

7.14 Function: `extract_noise()`

`void extract_noise(int average, int which, float *in, int n, float delta_t, double *whiten_ou`
`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 $1 \leq$ `which` ≤ 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 $W(t)$ 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 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 $i = 0, 1, \cdots, N-1$.

`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 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 $i = 0, 1, \cdots, N/2 - 1$.

`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,

$$P(f) := \frac{2}{T}\ \tilde{s}^*(f)\tilde{s}(f)\ , \tag{7.14.1}$$

where $\tilde{s}(f)$ is the Fourier transform of the unwhitened data stream $s(t)$ produced by the detector. These variables have units of strain^2/Hz (or seconds). `power[i]` contains the value of $P(f)$ evaluated at the discrete frequency $f_i = i/(N\Delta t)$, where $i = 0, 1, \cdots, N/2 - 1$.

`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:

$$w(t) := \sqrt{\frac{8}{3}} \cdot \frac{1}{2} \left[1 - \cos\left(\frac{2\pi t}{T}\right) \right] . \tag{7.14.2}$$

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:

$${}^{(1)}P(f) := \frac{2}{T} \, {}^{(1)}\tilde{s}^*(f) \, {}^{(1)}\tilde{s}(f) . \tag{7.14.3}$$

(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:

$$P(f) := \frac{1}{2} \left[\, {}^{(1)}P(f) + \, {}^{(2)}P(f) \, \right] . \tag{7.14.4}$$

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` $\neq 1$, 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` $\neq$ 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.

7.15 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 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 $i = 0, 1, \cdots, N-1$.

`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 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 $i = 0, 1, \cdots, N/2-1$.

`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

$$\tilde{s}_{12}(f) := (\tilde{s}_1^*(f)\ \tilde{s}_2(f) + \text{c.c.})\ , \tag{7.15.1}$$

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 strain^2·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 $i = 0, 1, \cdots, N/2-1$.

`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:

$$w(t) := \sqrt{\frac{8}{3}} \cdot \frac{1}{2}\left[1 - \cos\left(\frac{2\pi t}{T}\right)\right] . \tag{7.15.2}$$

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) spectra $\tilde{W}_1(f)$ and $\tilde{W}_2(f)$, which represent the whitening filters of the two detectors. The resulting unwhitened frequency components are denoted by ${}^{(1)}\tilde{s}(f)$ and ${}^{(1)}\tilde{s}(f)$; 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:

$${}^{(1)}\tilde{s}_{12}(f) := \left[\ {}^{(1)}\tilde{s}_1^*(f)\ {}^{(1)}\tilde{s}_2(f) + \text{c.c.}\ \right] . \tag{7.15.3}$$

(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:

$$\tilde{s}_{12}(f) := \frac{1}{2}\left[\ {}^{(1)}\tilde{s}_{12}(f) + {}^{(2)}\tilde{s}_{12}(f)\ \right] . \tag{7.15.4}$$

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` $\neq 1$, 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` $\neq 1$.

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. 7.14.)

7.16 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: $\Omega_{\rm gw}(f) = \Omega_0$ for $f_{\rm low} \le f \le f_{\rm high}$.

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 $f_{\rm low}$ (in Hz) below which the spectrum $\Omega_{\rm gw}(f)$ of the stochastic background—and hence the optimal filter $\tilde{Q}(f)$—is zero. $f_{\rm low}$ should lie in the range $0 \le f_{\rm low} \le f_{\rm Nyquist}$, where $f_{\rm Nyquist}$ is the Nyquist critical frequency. (The Nyquist critical frequency is defined by $f_{\rm Nyquist} := 1/(2\Delta t)$, where Δt is the sampling period of the detectors.) $f_{\rm low}$ should also be less than or equal to $f_{\rm high}$.

`f_high`: Input. The frequency $f_{\rm high}$ (in Hz) above which the spectrum $\Omega_{\rm gw}(f)$ of the stochastic background—and hence the optimal filter $\tilde{Q}(f)$—is zero. $f_{\rm high}$ should lie in the range $0 \le f_{\rm high} \le f_{\rm Nyquist}$. It should also be greater than or equal to $f_{\rm low}$.

`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 $i = 0, 1, \cdots, N-1$.

`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 strain^2/Hz (or seconds). `power1[i]` contains the value of $P_1(f)$ evaluated at the discrete frequency $f_i = i\Delta f$, where $i = 0, 1, \cdots, N-1$.

`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 $i = 0, 1, \cdots, N-1$.

The values of $\tilde{Q}(f)$ calculated by `optimal_filter()` are defined by equation (3.32) of Ref. [20]:

$$\tilde{Q}(f) := \lambda \frac{\gamma(f)\Omega_{\rm gw}(f)}{f^3 P_1(f) P_2(f)} \,. \tag{7.16.1}$$

Such a filter maximizes the cross-correlation signal-to-noise ratio SNR $:= \mu/\sigma$, where

$$\mu := \langle S\rangle = T\,\frac{3H_0^2}{20\pi^2}\int_{-\infty}^{\infty} df\ \gamma(|f|)|f|^{-3}\Omega_{\rm gw}(|f|)\tilde{Q}(f) \qquad (7.16.2)$$

$$\sigma^2 := \langle S^2\rangle - \langle S\rangle^2 \approx \frac{T}{4}\int_{-\infty}^{\infty} df\ P_1(|f|)P_2(|f|)|\tilde{Q}(f)|^2\,. \qquad (7.16.3)$$

(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. 7.16.3 for σ^2 needs to be modified, as discussed in Sec. 7.18.

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

$$\Omega_{\rm gw}(f) = \begin{cases} \Omega_0 & f_{\rm low} \le f \le f_{\rm high} \\ 0 & \text{otherwise,} \end{cases}$$

it is convenient to choose λ so that

$$\mu = \Omega_0\,T\,. \qquad (7.16.4)$$

From equations (7.16.1) and (7.16.2), it follows that

$$\lambda = \left[\frac{3H_0^2}{10\pi^2}\,\Omega_0\int_{f_{\rm low}}^{f_{\rm high}} df\ \frac{\gamma^2(f)}{f^6P_1(f)P_2(f)}\right]^{-1} \qquad (7.16.5)$$

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.

7.17 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 explictly 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. 7.1 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 $i = 0, 1, \cdots, N-1$. A plot of this data is shown in Fig. 51.

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. 52 is a plot of the optimal filter function for the advanced LIGO detectors.

```
/* 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 500.0                   /* maximum frequency (in Hz) */
#define OUT_FILE "LIGO_filter.dat"     /* output filename */

main()
{
  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;
}
```

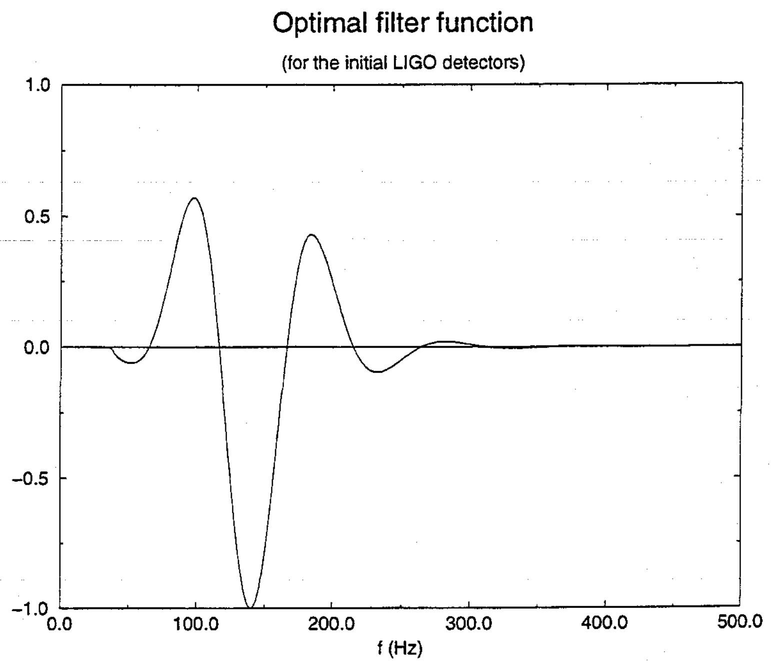


Figure 51: Optimal filter function $\tilde{Q}(f)$ (normalized to 1) for the initial LIGO detectors.

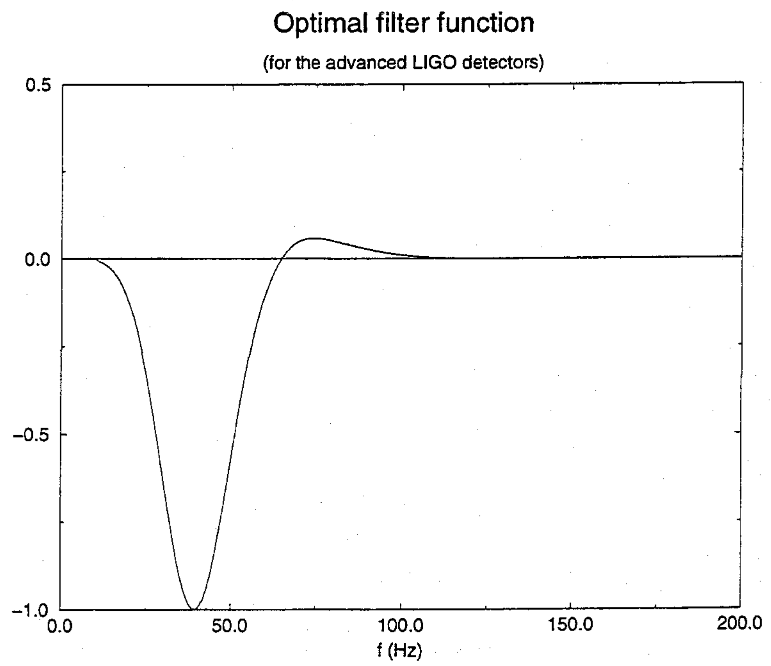


Figure 52: Optimal filter function $\tilde{Q}(f)$ (normalized to 1) for the advanced LIGO detectors.

7.18 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 natually 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 theroretically 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. 7.20 and 7.21.

To answer the above question, we will need to evaluate both the mean value

$$\mu := \langle S \rangle \tag{7.18.1}$$

and the variance

$$\sigma^2 := \langle S^2 \rangle - \langle S \rangle^2 \tag{7.18.2}$$

of the stochastic background cross-correlation signal S. The signal-to-noise ratio SNR is then given by

$$\text{SNR} := \frac{\mu}{\sigma} \,. \tag{7.18.3}$$

As described in Sec. 7.16, 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

$$\mu = T\,\frac{3H_0^2}{20\pi^2}\int_{-\infty}^{\infty} df\ \gamma(|f|)|f|^{-3}\Omega_{\rm gw}(|f|)\tilde{Q}(f) \tag{7.18.4}$$

$$\sigma^2 \approx \frac{T}{4}\int_{-\infty}^{\infty} df\ P_1(|f|)P_2(|f|)|\tilde{Q}(f)|^2 \,, \tag{7.18.5}$$

where $\tilde{Q}(f)$ is an arbitrary filter function. The choice

$$\tilde{Q}(f) := \lambda\frac{\gamma(f)\Omega_{\rm gw}(f)}{f^3 P_1(f)P_2(f)} \tag{7.18.6}$$

maximizes the signal-to-noise ratio (7.18.3). It is the *optimal* filter for stochastic background searches. As also described in Sec. 7.16, if the stochastic background has a constant frequency spectrum

$$\Omega_{\rm gw}(f) = \begin{cases} \Omega_0 & f_{\rm low} \le f \le f_{\rm high} \\ 0 & \text{otherwise,} \end{cases}$$

it is convenient to choose the normalization constant λ so that

$$\mu = \Omega_0\, T \,. \tag{7.18.7}$$

For such a λ,

$$\sigma^2 \approx \frac{T}{2}\left(\frac{10\pi^2}{3H_0^2}\right)^2\left[\int_{f_{\rm low}}^{f_{\rm high}} df\ \frac{\gamma^2(f)}{f^6 P_1(f)P_2(f)}\right]^{-1} \,, \tag{7.18.8}$$

which leads to the *squared* signal-to-noise ratio

$$(\mathrm{SNR})^2 = T\,\Omega_0^2\,\frac{9H_0^4}{50\pi^4}\int_{f_\mathrm{low}}^{f_\mathrm{high}} df\;\frac{\gamma^2(f)}{f^6 P_1(f)P_2(f)}\,. \tag{7.18.9}$$

This is equation (3.33) in Ref. [20].

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. (7.18.5) 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 (7.18.4) for μ is unaffected.) This change in σ^2 implies that Eq. (7.18.6) 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. (7.18.6) 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

$$\mu = \Omega_0\,T\,, \tag{7.18.10}$$

(the same λ as before works), but now

$$\begin{aligned}\sigma^2 = \frac{T}{2}\left[\int_{f_\mathrm{low}}^{f_\mathrm{high}} df\;\frac{\gamma^2(f)}{f^6P_1(f)P_2(f)}\right]^{-2}&\left\{\left(\frac{10\pi^2}{3H_0^2}\right)^2\int_{f_\mathrm{low}}^{f_\mathrm{high}} df\;\frac{\gamma^2(f)}{f^6P_1(f)P_2(f)}\right.\\ &+\Omega_0\left(\frac{10\pi^2}{3H_0^2}\right)\int_{f_\mathrm{low}}^{f_\mathrm{high}} df\;\frac{\gamma^2(f)}{f^9P_1^2(f)P_2(f)}+\Omega_0\left(\frac{10\pi^2}{3H_0^2}\right)\int_{f_\mathrm{low}}^{f_\mathrm{high}} df\;\frac{\gamma^2(f)}{f^9P_1(f)P_2^2(f)}\\ &\left.+\Omega_0^2\int_{f_\mathrm{low}}^{f_\mathrm{high}} df\;\frac{\gamma^2(f)}{f^{12}P_1^2(f)P_2^2(f)}\left(1+\gamma^2(f)\right)\right\}.\end{aligned} \tag{7.18.11}$$

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 (7.18.8). 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 (7.18.11) for σ^2 should be used. The function `calculate_var()`, which is defined in the following section, calculates the variance using this equation.

7.19 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 $\Omega_{\rm gw}(f)$ for the stochastic background:

$$\Omega_{\rm gw}(f) = \begin{cases} \Omega_0 & f_{\rm low} \le f \le f_{\rm high} \\ 0 & \text{otherwise.} \end{cases}$$

Ω_0 should be greater than or equal to zero.

`f_low`: Input. The frequency $f_{\rm low}$ (in Hz) below which the spectrum $\Omega_{\rm gw}(f)$ of the stochastic background is zero. $f_{\rm low}$ should lie in the range $0 \le f_{\rm low} \le f_{\rm Nyquist}$, where $f_{\rm Nyquist}$ is the Nyquist critical frequency. (The Nyquist critical frequency is defined by $f_{\rm Nyquist} := 1/(2\Delta t)$, where Δt is the sampling period of the detector.) $f_{\rm low}$ should also be less than or equal to $f_{\rm high}$.

`f_high`: Input. The frequency $f_{\rm high}$ (in Hz) above which the spectrum $\Omega_{\rm gw}(f)$ of the stochastic background is zero. $f_{\rm high}$ should lie in the range $0 \le f_{\rm high} \le f_{\rm Nyquist}$. It should also be greater than or equal to $f_{\rm low}$.

`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 $i = 0, 1, \cdots, N-1$.

`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 strain^2/Hz (or seconds). `power1[i]` contains the value of $P_1(f)$ evaluated at the discrete frequency $f_i = i\Delta f$, where $i = 0, 1, \cdots, N-1$.

`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. (7.18.11) of Sec. 7.18. As discussed in that section, Eq. (7.18.11) for σ^2 makes no assumption about the relative strengths of the stochastic background and detector noise signal, but it does use Eq. (7.18.6) 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 of the parameter Ω_0 should be set to zero. The variance for this case is given by Eq. (7.18.8).

Authors: Bruce Allen, ballen@dirac.phys.uwm.edu, and Joseph Romano, romano@csd.uwm.edu

Comments: None.

7.20 Example: snr program

As mentioned in Sec. 7.18, an interesting question to ask in regard to stochastic background searches is: "What is the theroretically 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 show 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: $\Omega_{\rm gw}(f) = \Omega_0$ for $f_{\rm low} \le f \le f_{\rm high}$. 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 Hz $\le f \le$ 5000 Hz. (The answer is 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.

```
/* 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 5.0                  /* minimum frequency (in Hz) */
#define F_HIGH (5.0e+3)            /* 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) */

main()
{
  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;
}
```

7.21 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 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 inital 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: $\Omega_{\rm gw}(f) = \Omega_0$ for 5 Hz $\leq f \leq$ 5000 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. 7.18, the squared signal-to-noise ratio can be written in the following form:

$$({\rm SNR})^2 = \frac{T\,\Omega_0^2}{A + B\,\Omega_0 + C\,\Omega_0^2}\,, \qquad (7.21.1)$$

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. (7.21.1) becomes a quadratic for Ω_0:

$$a\,\Omega_0^2 + b\,\Omega_0 + c = 0\,, \qquad (7.21.2)$$

which we can easily solve. It is this procedure that we implement in the following program.

```
/* 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 5.0                /* minimum frequency (in Hz) */
#define F_HIGH (5.0e+3)          /* 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) */

main()
{
  int    i;
  float  f;

  double factor,f3,f6,f9,f12,p1,p2,g2;
  double int1,int2,int3,int4;
  double a,b,c,omega_0;

  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 INTEGRALS FOR VARIANCE */
int1=int2=int3=int4=0.0;

for (i=0;i<N;i++) {
  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];

    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("Minumum Omega_0 = %e\n",omega_0);
  printf("\n");

  return;
}
```

7.22 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 real_time_noise1, int real_time_noise2, double *power1, double *power2, double *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: $\Omega_{\rm gw}(f) = \Omega_0$ for $f_{\rm low} \le f \le f_{\rm high}$. 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 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 $i = 0, 1, \cdots, N-1$.

`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 $f_{\rm low}$ (in Hz) below which the spectrum $\Omega_{\rm gw}(f)$ of the stochastic background is assumed to be zero. $f_{\rm low}$ should lie in the range $0 \le f_{\rm low} \le f_{\rm Nyquist}$, where $f_{\rm Nyquist}$ is the Nyquist critical frequency. (The Nyquist critical frequency is defined by $f_{\rm Nyquist} := 1/(2\Delta t)$, where Δt is the sampling period of the detectors.) $f_{\rm low}$ should also be less than or equal to $f_{\rm high}$.

`f_high`: Input. The frequency $f_{\rm high}$ (in Hz) above which the spectrum $\Omega_{\rm gw}(f)$ of the stochastic background is assumed to be zero. $f_{\rm high}$ should lie in the range $0 \le f_{\rm high} \le f_{\rm Nyquist}$. It should also be greater than or equal to $f_{\rm low}$.

`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 $i = 0, 1, \cdots, N/2-1$.

`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 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 $i = 0, 1, \cdots, N/2-1$.

`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 strain^2/Hz (or seconds). `power1[i]` contains the value of $P_1(f)$ evaluated at the discrete frequency $f_i = i/(N\Delta t)$, where $i = 0, 1, \cdots, N/2-1$. If `real_time_noise1` $= 1$, the values of `power1[0..n/2-1]` are changed to

$$P_1(f) := \frac{2}{T}\, \tilde{s}_1^*(f)\tilde{s}_1(f) \ , \tag{7.22.1}$$

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

$$S := \int_{f_{\rm low}}^{f_{\rm high}} df\ \tilde{s}_{12}(f)\ \tilde{Q}(f) \ , \tag{7.22.2}$$

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. (7.22.2). 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. 7.14 and 7.15 for more details.)

Authors: Bruce Allen, ballen@dirac.phys.uwm.edu, and Joseph Romano, romano@csd.uwm.edu

Comments: None.

7.23 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 $\Omega_{\rm gw}(f)$ for the stochastic background:

$$\Omega_{\rm gw}(f) = \begin{cases} \Omega_0 & f_{\rm low} \le f \le f_{\rm high} \\ 0 & \text{otherwise.} \end{cases}$$

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 $(i = 1, 2, \cdots, n)$ denote a set of n statistically independent random variables, each having the same mean value

$$\mu := \langle S_i \rangle \ , \tag{7.23.1}$$

but different variances

$$\sigma_i^2 := \langle S_i^2 \rangle - \langle S_i \rangle^2 \ , \tag{7.23.2}$$

then one can show that the weighted-average

$$\bar{S} := \frac{\sum_{i=1}^n \lambda_i S_i}{\sum_{j=1}^n \lambda_j} \tag{7.23.3}$$

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

$$T_{\rm tot} := n\, T \ , \tag{7.23.4}$$

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

$$\mu_{\text{theory}} = \Omega_0\, T\ . \tag{7.23.5}$$

This follows from our choice of normalization constant for the optimal filter function. (See Sec. 7.16 for more details.)

(iii) The theoretical variance is given by

$$\sigma^2_{\text{theory}} = \frac{n}{\sum_{i=1}^n \sigma_i^{-2}}\ . \tag{7.23.6}$$

Note that when the detector noise power spectra are constant, $\sigma_i^2 =: \sigma^2$ for $i = 1, 2, \cdots, n$ and $\sigma^2_{\text{theory}} = \sigma^2$. 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

$$\text{SNR}_{\text{theory}} = \sqrt{n}\,\frac{\mu_{\text{theory}}}{\sigma_{\text{theory}}}\ . \tag{7.23.7}$$

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

$$\mu_{\text{expt}} := \frac{\sum_{i=1}^n \sigma_i^{-2} S_i}{\sum_{j=1}^n \sigma_j^{-2}}\ . \tag{7.23.8}$$

(vi) The experimental variance is given by

$$\sigma^2_{\text{expt}} := \frac{\sum_{i=1}^n \sigma_i^{-2} S_i^2}{\sum_{j=1}^n \sigma_j^{-2}} - \mu^2_{\text{expt}}\ . \tag{7.23.9}$$

When the weights σ_i^{-2} are constant, the above formula reduces to the usual expression

$$\sigma^2_{\text{expt}} = \frac{1}{n}\sum_{i=1}^n S_i^2 - \left(\sum_{i=1}^n S_i\right)^2 \tag{7.23.10}$$

for the variance of n measurements S_i.

(vii) The experimental signal-to-noise ratio is given by

$$\text{SNR}_{\text{expt}} = \sqrt{n}\,\frac{\mu_{\text{expt}}}{\sigma_{\text{expt}}}\ . \tag{7.23.11}$$

(viii) The relative error in the signal-to-noise ratios is

$$\text{relative error} := \left|\frac{\text{SNR}_{\text{theory}} - \text{SNR}_{\text{expt}}}{\text{SNR}_{\text{theory}}}\right| \cdot 100\%\ . \tag{7.23.12}$$

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.

7.24 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. 7.22 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()`.

7.25 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: $\Omega_{\rm gw}(f)=\Omega_0$ for $f_{\rm low}\le f\le f_{\rm high}$. 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, $f_{\rm low}$, and $f_{\rm high}$) 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. 7.26.

```
/* 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 (5.0)            /* minimum frequency (in Hz) */
#define F_HIGH (5.0e3)         /* 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 1600          /* number of runs (for simulation) */
#define NUM_BINS 200           /* number of bins (for statistics) */

main()
{
  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;
}
```

7.26 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. 7.25. Also shown, in Fig. 53, 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
```

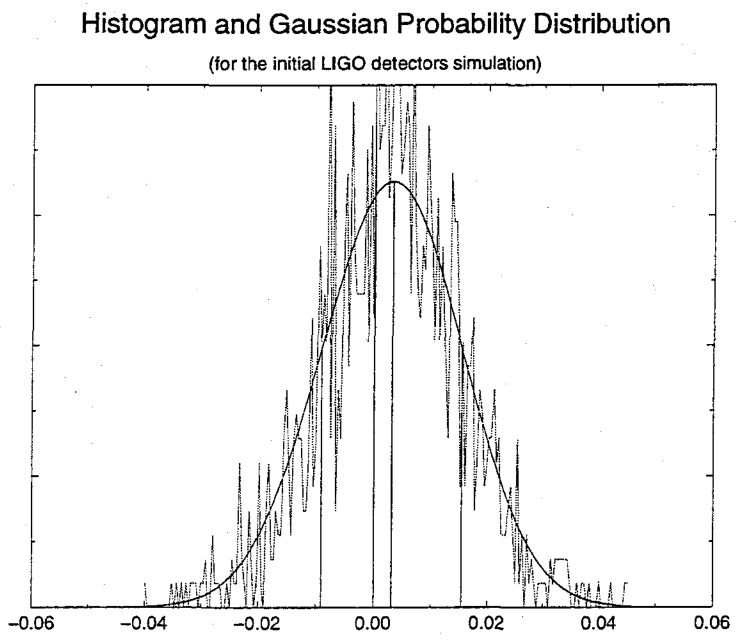


Figure 53: Histogram of the measured cross-correlation signal vaues, and the corresponding best-fit Gaussian probability distribution for the stochastic background simulation.

8 GRASP Routines: Supernovae and other transient sources

9 GRASP Routines: Periodic and quasi-periodic sources

10 GRASP Routines: General purpose utilities

This section includes general purpose utility functions for a variety of purposes. For example, these include functions to calculate time-averaged power spectra, and functions to graph data, listen to data, etc.

10.1 Function: `grasp_open()`

`FILE* grasp_open(const char *environment_variable,char *shortpath)`

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.

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")`, then the routine opens the file `/usr/local/data/14nov94.2/channel.0` and returns a pointer to it.

10.2 Function: avg_spec()

`void avg_spec(float *data,float *average,int npoint,int *reset,float srate,float decaytime,i`
`windowtype)`

This routine calculates the power spectrum of the (time-domain) input stream `data[ ]`, averaged over time with a user-set exponential decay, and several possible choices of windowing.

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-1]`. The value of `average[m]` is the average power spectrum at frequency

$$f = \frac{\mathtt{m} \times \mathtt{srate}}{2 \times \mathtt{N}}. \tag{10.2.1}$$

This is twice the number of distinct frequency values which appear in the FFT of `N` samples; this is because of the overlapping technique described below. 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.

The methods used in this routine are quite similar to those used in the `overlap=1` version of the *Numerical Recipes* [1] routine `spctrm()`, and the reader interested in the details of this routine should first read the corresponding section of [1]. A continuous sample of the input data of twice the length of the array `data[ ]` is maintained by `avg_spec()`. Thus, each element of the array `data[ ]` utilized twice; once with the first point `data[0]` right in the middle of the time-domain window function, and once more with that same point right at the beginning of the window function. Note that to reproduce (exactly) the procedure described in *Numerical Recipes* [1] one must have `npoint`=M where M is the variable used in the procedure `spctrm()`, and the decay time must be very large (so that the two successive spectra are equally weighted). For example, if you are doing analysis with 2048 samples, using that as the number of samples which you FFT and correlate, then you should make two calls to `avg_spec()`. in each of which `npoint`=1024; this will give you one spectral bin per FFT bin.

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

$$\langle A(f)\rangle_t = \frac{\int_{-\infty}^{t} dt'\ A_{t'}(f) e^{-(t-t')/\tau}}{\int_{-\infty}^{t} dt'\ e^{-(t-t')/\tau}}, \tag{10.2.2}$$

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 nth pass through the averaging routine. The discrete analog of the previous equation (10.2.2) is

$$\langle A(f)\rangle_N = \frac{\sum_{n=0}^{N} A_n(f) e^{-\alpha(N-n)}}{\sum_{n=0}^{N} e^{-\alpha(N-n)}}. \tag{10.2.3}$$

Here,

$$\alpha = \frac{\texttt{npoint}}{\texttt{srate} \times \texttt{decaytime}} \tag{10.2.4}$$

is determined by the averaging time desired. The average defined by (10.2.3) can be easily determined by a recursion relation. We denote the the normalization factor by

$$\mathcal{N}_N = \sum_{n=0}^{N} e^{-\alpha(N-n)}. \tag{10.2.5}$$

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

$$\langle A(f)\rangle_N = e^{-\alpha}\frac{\mathcal{N}_{N-1}}{\mathcal{N}_N}\langle A(f)\rangle_{N-1} + \frac{A_N(f)}{\mathcal{N}_N} \quad \text{for } N = 0, 1, 2, \cdots \tag{10.2.6}$$

(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.

$$\text{units}\,(\texttt{average[ ]}) = (\text{units}\,(\texttt{data[ ]}))^2\,/\text{Hz}. \tag{10.2.7}$$

The example program `calibrate` described earlier makes use of the routine `avg_spec()`.

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

Comments: See comments for `calibrate()`.

10.3 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-digitial 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 $-$`offset`, $-$`offset` $+ 1, \cdots,$ `offset` $- 1$ you may end up over-writing another array! You'll then spend unhappy hours trying to locate the source of bizzare 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.

10.4 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:

$$\bar{x} = \frac{1}{n}\sum_{i=0}^{n-1} x_i. \tag{10.4.1}$$

- 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

$$\sum_{i=-s}^{s} N[i+\bar{x}] > n\,\mathrm{erf}(1/\sqrt{2}) = n\frac{1}{\sqrt{2\pi}}\int_{-1}^{1} \mathrm{e}^{-x^2/2}dx. \tag{10.4.2}$$

  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:

$$n(\mathrm{erfc}(3/\sqrt{2}) - \mathrm{erfc}(5/\sqrt{2})). \qquad (10.4.3)$$

- 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.

10.5 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]`, $\cdots$, `array[(n-1)*spacing]`.

`n`: Input. The number of array elements that are set to zero.

`spacing`: Input. The spacing in the array between succesive elements that are set to zero.

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

Comments: None.

10.6 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.

10.9 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.

10.10 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.

10.11 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.

10.12 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.

10.13 Function: `audio()`

`void audio(short *array,int n)`

Makes a Sun workstation play music!

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 equiped 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.

10.14 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.

10.15 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 (in Coordinated Universal Time). An example of such a file (produced by `translate`) is:
`C1-94_10_15_06_18_02`
where `C1` denotes the Caltech 40-meter prototype. Suggested names for the other sites are `H2` and `H4` for the two Hanford LIGO detectors, `L1` for the Livingston LIGO detector `V1` for the Virgo detector, `G1` for the GEO detector, `T1` for the Tama detector, `S1` for the Glasgow detector, `M1` for the Max-Plank detector, and `A1` for the AIGO detector. 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.

```
#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 */

/* 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;
    int i,code=1,num,large=50000,small=5000,n,first=1,firsttime,anyopen=0,nlines;
    long buffSize;
    float fastrate=9868.4208984375,tblock,slowrate=986.84208984375,*real,*imag,*freq,theta;
    double firstmsec,first_estimate,second_estimate,diff,dt,dtslow;
    float starttime=-100.0,guesstime;
    double currenttime=-200;
    int blockcount=0;
    struct FrFile *outputfile;
    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 time,*gtime,gts;
    time_t caltime;
    FILE *fp[16],*fpsweptsine;
    void unhappyexit(int i);
    int get_run_number(int firsttime);

    /* 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 */
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;

/* Correct for oblateness of earth, use reference spheriod 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 */
theta=(M_PI/180.0)*(90.0-frdetect->latitude);
frdetect->altitude=EQUATORIAL*(1.0-FLAT*cos(theta)*cos(theta));

/* 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++) {
   sprintf(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);
   anyopen=(anyopen||fp[i]);
}

/* if there are no open files, then please exit with a warning message */
if (!anyopen) 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 */
/* create structure to store localtime of tape */
time.tm_sec=main_header.tod_second;
time.tm_min=main_header.tod_minute;
time.tm_hour=main_header.tod_hour;
time.tm_mday=main_header.date_day;
time.tm_mon=main_header.date_month;
time.tm_year=main_header.date_year;
time.tm_wday=main_header.date_dow;
time.tm_yday=-1;   /* info not available, but filled in by mktime */
time.tm_isdst=-1;  /* info not available, but filled in by mktime */
caltime=mktime(&time);
gtime=gmtime(&caltime);
gts=*gtime;

if (caltime!=main_header.epoch_time_sec) {
    fprintf(stderr,"I am confused about the correct time: %d or %d\n",
                    (int)caltime,main_header.epoch_time_sec);
    fprintf(stderr,"If not running on a PST time-zone machine, ignore error!\n");
}

/* 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("Local start time: %s\n",ctime(&caltime));
    printf("UTC start time:   %s",asctime(&gts));

    /* 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 (overwrites long blank space above) */
        if (frame->run<2)
            chan_name=prenov15;
        else
            chan_name=postnov15;

        for (i=0;i<16;i++)
            if (fp[i]!=NULL) {
                strcpy(adc[i]->name,chan_name[i]);
                /* put in the physical volts/counts conversion */

                adc[i]->data->unit[0]=(char *)malloc((strlen(units)+1)*sizeof(char));
                strcpy(adc[i]->data->unit[0],units);

                /* 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;
        frame->UTimeS=firsttime+(int)currenttime;
        frame->UTimeN=(int)(1.e9*(currenttime-(int)currenttime));
        frame->dt=76.0/15.0;
    }
    else {
        /* put the time stamp into the frame structure */
        tblock+=firstmsec;
        frame->UTimeS=firsttime+(int)tblock;
        frame->UTimeN=(int)(1.e9*(tblock-(int)tblock));
        frame->dt=num/slowrate;
    }
      /* frame->type[0]=0; */

    /* put in the history information (only once per translation) */
    if (first) {
        first=0;
        histnew=hist;
        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,"     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");
    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 */
    framevec=FrVectNew(FR_VECT_F,1,3*nlines,1.0,"Vifo/Vcoil");
    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 */
    staticdata=FrStatDataNew("sweptsine",
         "swept sine calibration:\npacking: freq[i], real[i], imaginary[i]",
         frame->UTimeS,MAXINT,1,framevec);
    FrStatDataAdd(&frame->detectRec->sData,staticdata);

   /* put in lock range (INCLUSIVE low->high) Rolf: if 0=unlock and 1=lock
      then you need LOCKLO=LOCKHI=1
    */
    framevecS=FrVectNew(FR_VECT_S,1,2,1.0,"adcCounts");
    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 */
    staticdataS=FrStatDataNew("locklo/lockhi",
         "lock range:\npacking: array[0]=locklo array[1]=lockhi",
         frame->UTimeS,MAXINT,1,framevecS);
    FrStatDataAdd(&frame->detectRec->sData,staticdataS);
}

/* is the time stamp for this data block consistent with start time+offset? */
first_estimate=frame->UTimeS+1.e-9*frame->UTimeN;
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! */
    sprintf(filename,"C1-%02d_%02d_%02d_%02d_%02d_%02d",gts.tm_year,gts.tm_mon+1,
              gts.tm_mday,gts.tm_hour,gts.tm_min,gts.tm_sec);
    printf("Filename: %s\n",filename);
    outputfile=FrFileONew(filename, NO, buff, buffSize);
}
```

```
      /* 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 reqire periodic updates. Should the year have four digits (eg, 1994) for easier sorting?

10.16 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 [23]. There is now a standard textbook on the subject, by Percival and Walden [24], 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 [25]. 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 unfamilar with these techniques, we suggest reading Chapter 7 of [24]. 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 339 of [24]) 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 [24]) 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 [24], 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.

10.17 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. It uses the method described in Percival and Walden [24] 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 10.16).

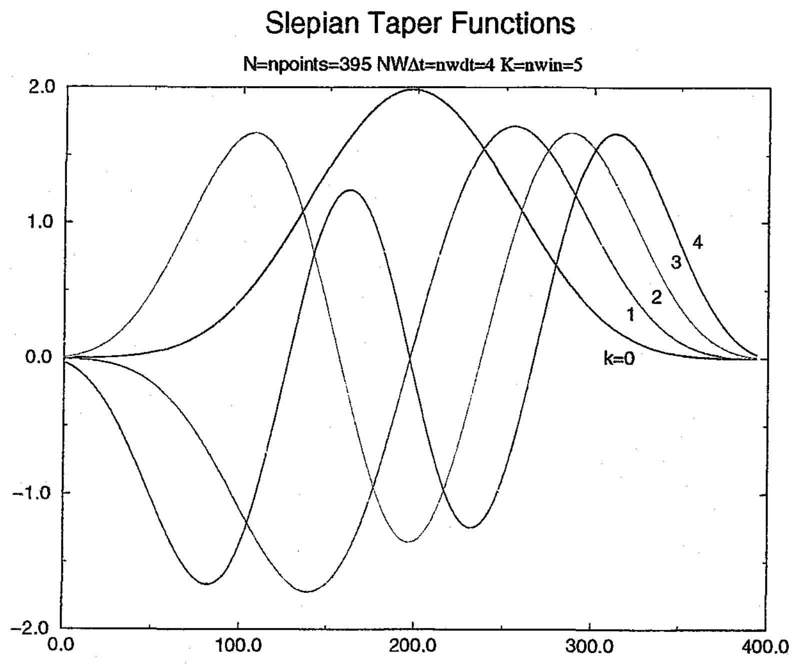


Figure 54: Here are five Slepian tapers computed with `slepian_tapers()`. The parameters are `npoints=395`, `nwdt=4.0` and `nwin=5`.

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.

10.18 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 klen, float *cest, int dospec)`

This function computes the multi-taper spectrum, as defined for example by Percival and Walden [24] equation (333). For the sake of efficiency, it computes 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.

The arguments are:

`data`: Input. Pointer to the time-domain data array, `data[0..npoints-1]`.

`npoints`: Input. Number of data points in the `data` array.

`kind`: Input. If set to 1, compute the normal multi-taper spectrum. If set to 2, compute the "adaptive" spectrum defined by 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..klen/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 `nwin-1` for `kind=1` above, and only useful in the adaptive case where `kind=2`. The array range is `dof[0..klen/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..klen/2]`. Warning -this is an *odd* number of entries. The user must provide a pointer to sufficient storage space.

`klen`: Input. An integer power of 2, greater than (or equal to) `npoints`. The (tapered) data is zero-padded out to this length. You generally want `klen` 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 $N_f = 1 + \texttt{klen}/2$.

`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..klen+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.

10.19 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.

10.20 Function: `fvalue_cmp()`

`int fvalue_cmp(const void *f1, const void *f2)`

This is a function which may be used to compare the fvalues 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 stuctures. 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.

10.21 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 stuctures. 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.

10.22 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 spectrum, of the original data, and a "final" multi-taper 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 of zero-padded data that will be analyzed. Note that this 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`: Ouput. 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 power spectrum of the *initial* `data[]` array, including both DC and Nyquist frequency bins. The array range is `mtap_spec_init[0..padded_length/2]`. 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 power 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]`. 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 power 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 these power 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..`klen/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 `fimin` is 0..`klen/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 $1 - 1/$`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.

10.23 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 envionment variable giving the path to the river data, for example:

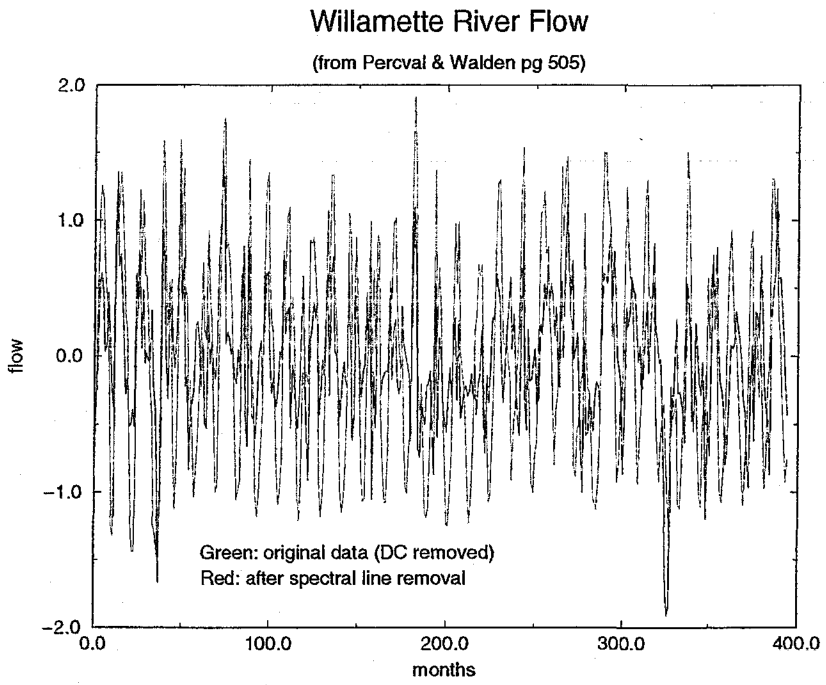
```
setenv GRASP_PARAMETERS /usr/local/GRASP/parameters
```

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)
```

```
#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");
   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;
}
```



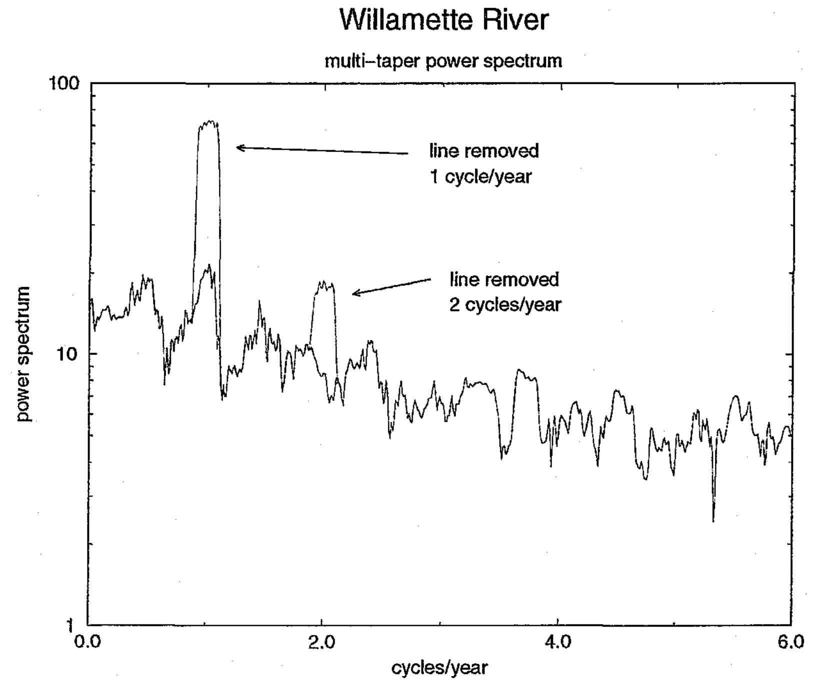


Figure 55: Output of the example program `river`, making use of `remove_spectral_lines()` to automatically find and remove two "spectral line" features from a data set. This is the same example treated by Percival and Walden in Section 10.13 of their textbook.

10.24 Example: ifo_clean

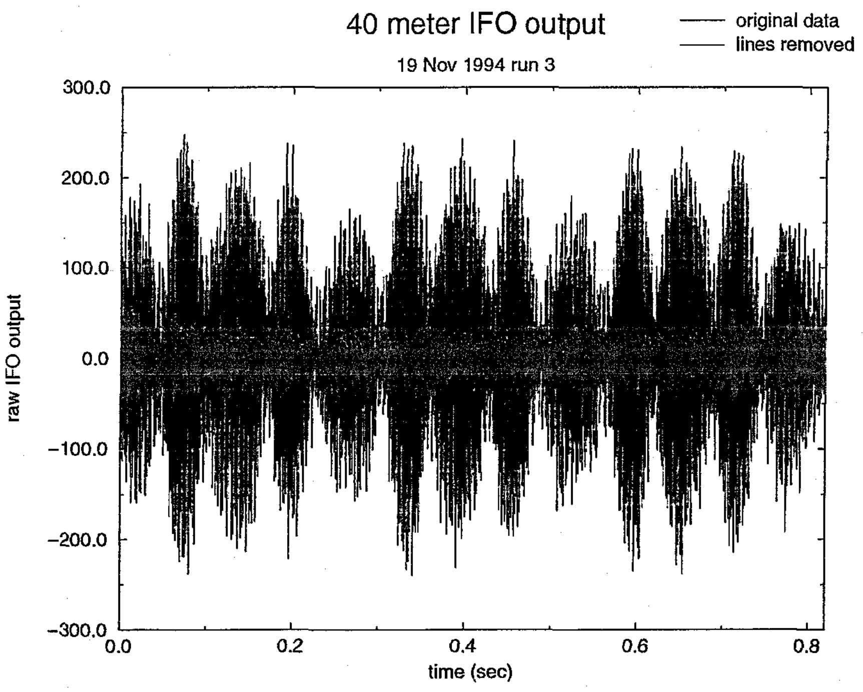
This example program uses `remove_spectral_lines()` to automatically identify and remove "spectral lines" from the output of the 40-meter IFO. To run this program, be sure to set the data path environment variable, for example:
`setenv GRASP_DATAPATH /usr/local/GRASP/data/19nov94.3`
The program outputs 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 56. 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 $A(t) = 21.91 \sin(360\pi t/\mathrm{sec})$. 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 56 is the result of these five harmonics.

```
#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");
  fplock=grasp_open("GRASP_DATAPATH","channel.10");

  /* 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;
}
```



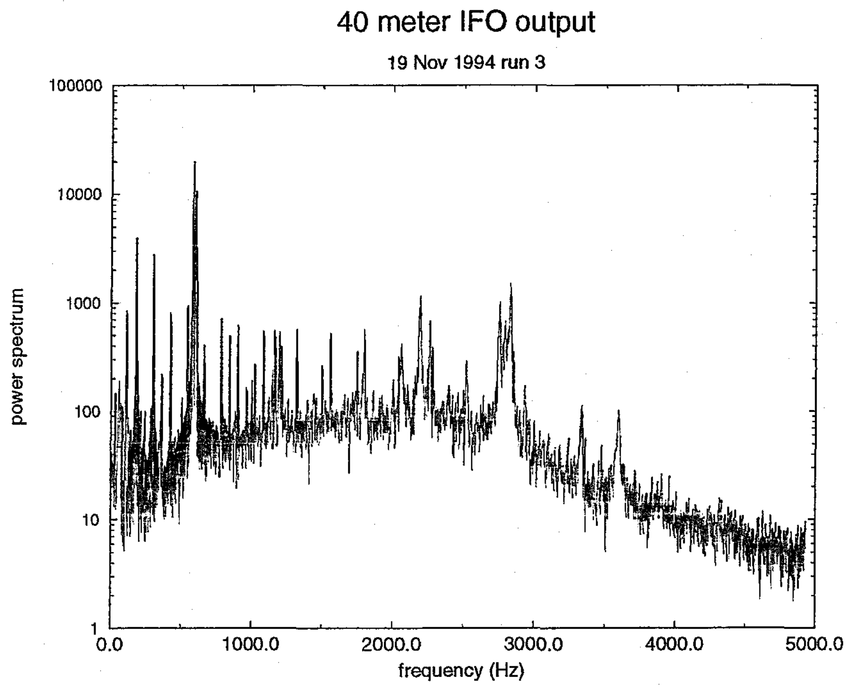


Figure 56: Output of the example program `ifo_clean`, making use of `remove_spectral_lines()` to automatically identify and remove "spectral line" features from the (whitened) output of the Caltech 40-meter interferometer. The black curve and the red curve show the before/after time series and spectra. We have deliberately choosen a stretch of data immediately after the IFO locks, so that the suspension and pendulum modes are excited.

10.25 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

$$\Delta f = \pm \frac{\texttt{nbins}}{\Delta t(\texttt{num\_points} \times \texttt{padding\_factor} + 2)} \qquad (10.25.1)$$

`num_display`: the number of points displayed by `tracker`. The total amount of time covered by the output is `num_display` $\times$ `num_points` $\times \Delta t$ 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.

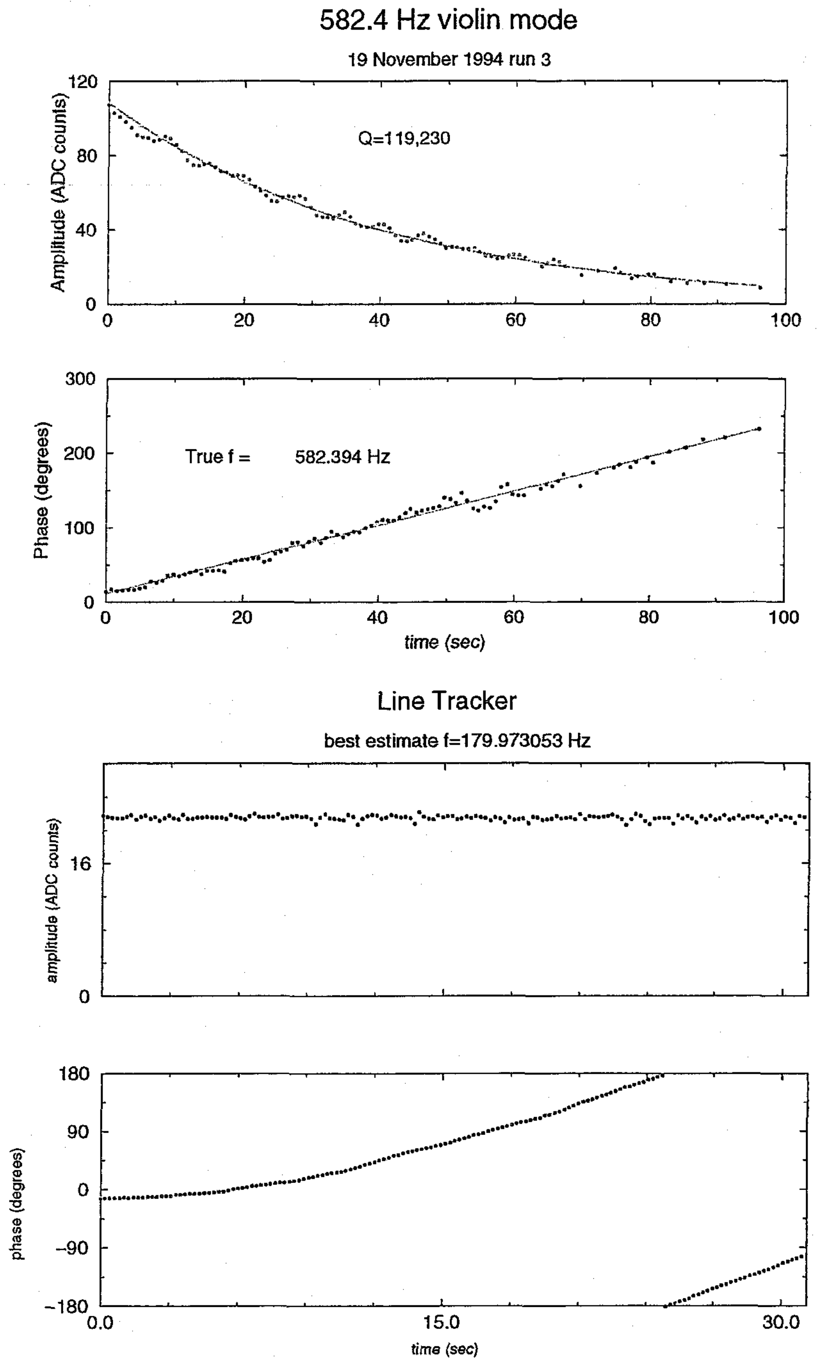


Figure 57: Output of the example program `tracker`, making use of `remove_spectral_lines()` to track the amplitude and phase of a selected "spectral line" features from the (whitened) output of the Caltech 40-meter interferometer. The upper two graphs show the approximately exponential decay of the 582.396 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).

10.26 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.

```
#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)>=0.0?(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);


main() {
   short *datas;
   int npass=1,num_points,num_win,num_freq,padded_length,max_lines,num_removed,remain,code,firstpass=
   int i,top,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,sdoub,tsdoub,tinitial,tstart,initime=0.0;
   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].dcdbr+
           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);
    linfitx[0]=times[index];
    linfity[0]=phase[index];
    for (i=1;i<nprint;i++) {
        index=MOD(top+i-nprint+1);
        linfitx[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,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.

11 References

[1] W.H. Press, B.P. Flannery, S.A. Teukolsky and W.T. Vettering, *Numerical recipes in C: the art of scientific computing*, 2nd edition, Cambridge University Press.

[2] Message Passing Interface Forum, *MPI: a message passing interface standard*, International Journal of Supercomputer Applications **8** 3/4 1994.

[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.

[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.

[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.

[6] 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).

[7] 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.

[8] C. Cutler *et al.*, *The Last Three Minutes: Issues in Gravitational-Wave Measurements of Coalescing Binaries*, Phys. Rev. Lett. **70** (1993) 2984-2987.

[9] 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.

[10] 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.

[11] 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).

[12] F. Echeverria *Gravitational-wave measurements of the mass and angular momentum of a black hole,* Phys. Rev. **D40** 3194–3203 (1989).

[13] J.N. Goldberg, A.J. Macfarlane, E.T. Newman, F. Rohrlich, and E.C.G. Sudarshan *Spin-s spherical harmonics and $\eth$,* J. Math. Phys. **8** 2155–2161 (1967).

[14] E.W. Leaver *An analytic representation for the quasi-normal modes of Kerr black holes,* Proc. Roy. Soc. Lond. **A402** (1985).

[15] H. Onozawa *Detailed study of quasinormal frequencies of the Kerr black hole,* Phys. Rev. **D55** 3593–3602 (1997).

[16] 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).

[17] 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).

[18] A.D. Gillespie, *Thermal Noise in the Initial LIGO Interferometers*, Caltech PhD thesis, 1995.

[19] T.T. Lyons, *An optically recombined laser interferometer for gravitational wave detection*, Caltech PhD thesis, 1997.

[20] B. Allen, *The stochastic gravity-wave background: sources and detection*, in 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, to be published)

[21] See equations (4.13) and (4.14) in *Spacecraft attitude determination and control*, Ed. James R. Wortz, (D. Reidel Publishing Co., Boston, 1985).

[22] A. Abramovici et al., *Science* **256**, 325 (1992).

[23] D.J. Thomson, *Spectrum estimation and harmonic analysis*, Proceedings of the IEEE, **70**, 1055-96, (1982).

[24] D.B. Percival and A.T. Walden, *Spectral analysis for physical applications*, first edition, Cambridge University Press, (1993).

[25] J.M. Lees and J. Park, *Multiple-taper spectral analysis: A stand-alone C subroutine*, Computers and Geology **21**, 199-236.