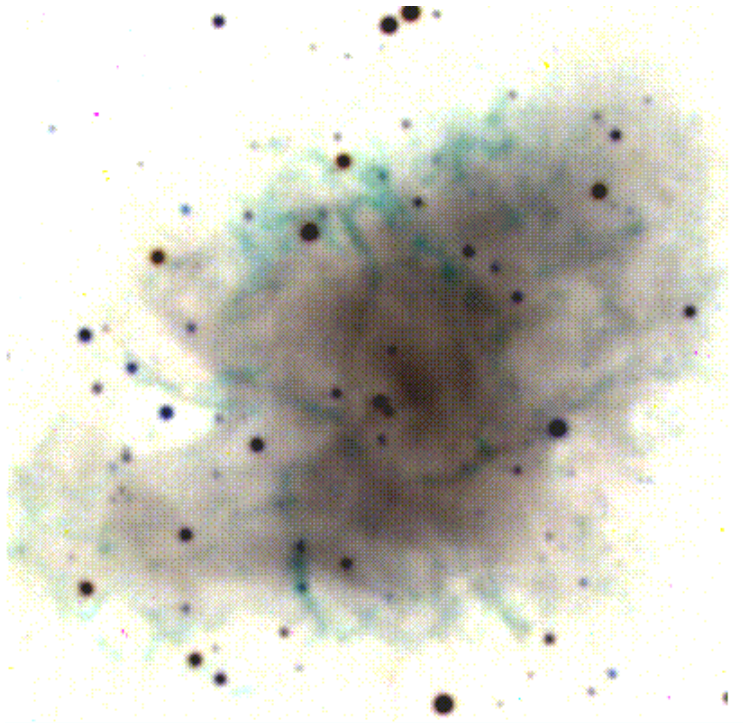




Software for the Periodic Source Search

User Guide



Version 02-11-2005

Updated version in http://grwavsf.roma1.infn.it/PSS/OtherDoc/PSS_UG.pdf

Contents

Introduction	6
Programming environments	8
Matlab.....	8
The gw project.....	10
C	11
SFC formats	12
Basics	12
Compressed data formats.....	13
LogX format	13
Sparse vector formats	14
PSS SFC files.....	16
Data preparation	18
Format change	18
Data selection.....	21
Basic sds operations.....	21
Choice of periods.....	23
Search for events	25
Filtering in a ds framework.....	25
The ev-ew structures	25
Coincidences.....	26
Event periodicities	27
SFDB.....	29
Theory	29
Procedure.....	30
Software	31
pss_sfdb.....	31
Software	32
Time-frequency data quality	33
Peak map	35
Peak map creation	35
Other peak map creation procedures.....	37
Hough transform	42
Theory	42
Implementation.....	43
Use of the library	47
Function prototypes	49
Program flow from the user point of view.....	49
User assigned parameters	50
Performance issue	50
Results of gprof.....	50
Comments	55
pss_explorer	57
pss_hough.....	57

Supervisor.....	58
Basics	58
Outline of the supervisor	60
Implementation of the Supervisor	61
Candidate database and coincidences	62
The database.....	62
Browsing the PSC database.....	64
Searching for coincidences in the PSC database.....	64
Coherent follow-up	65
Theory and simulation.....	66
Snag pss gw project.....	66
PSS detection theory	66
Sampled data simulation	66
Time-frequency map simulation.....	68
Peak map simulation	69
Low resolution simulation	69
High resolution simulation	70
Candidate simulation.....	71
Time and astronomical functions.....	72
Time	72
Astronomical coordinates	73
Source and Antenna structures	74
Doppler effect	75
Sidereal response.....	79
Tests and benchmarks.....	80
The PSS_bench program.....	80
The interactive program.....	80
The reports.....	82
SFDB.....	85
Hough transform.....	85
Service routines	86
Matlab service routines	86
pss_lib.....	87
pss_rog.....	87
General parameter structure	88
Main pss_ structure	88
const_ structure	90
source_ structure	91
antenna_ structure.....	92
data_ structure	93
fft_ structure	94
band_ structure.....	95
sfdb_ structure.....	96
tfmap_ structure	97
tfpmap_ structure.....	98
hmap_ structure.....	99
cohe_ structure	100
ss_ structure	101

candidate_ structure	102
event_ structure	103
computing_ structure	104
The PSS databases	105
General structure of PSS databases	105
The h-reconstructed database	107
The sfdb database	107
The normalized spectra database	107
The peak-map database	107
The candidate database	107
Database Metadata	107
Server docs	107
Analysis docs	107
Antenna docs	108
File System utilities	108
Appendix	109
Doppler effect computation	109
pss_astro	109
Programming tips	113
Windows FrameLib	113

Introduction

The PSS software is intended to process data of gravitational antennas to search for periodic sources.

It is based on two programming environments: MatLab and C. The first is basically oriented to interactive work, the second to batch or production work (in particular on the Beowulf farms).

The input gravitational antenna data on which the PSS software operates can be in various formats, as the **frame** (Ligo-Virgo) format, the **R87** (ROG) format, or the **sds** format (that is one of the Snag-SFC formats). The data produced at the various stages of the processing are stored in one of the Snag-SFC formats. The candidate database has a particular format.

There are some procedure to prepare data for processing. There is a basic check for timing and basic quality control. A report is created. Then the Short FFT Database (SFDB) is created. This is done in different way depending on the antenna type. For interferometric antennas it is done for 4 bands, obtaining 4 SFDBs. For bar antennas it is done for a single band.

The SFDB contains also a collection of “very short” periodograms. It has many uses, in particular it is used for the time-frequency data quality. From the SFDB the peak map is obtained; it is the starting point for the Hough transform.

The Hough transform (the “incoherent step” of a hierarchical procedure), that is the main part of our procedure, is normally run on a Beowulf computer farm. A Supervisor program creates and manages the set of tasks.

The Hough transform produces a huge number (billions) of candidate sources, each defined by a starting frequency, a position in the sky and a value of the spin-down parameter. These are stored in a database and when there are independent data analysis (for different periods or for different antennas), a coincidence search is performed on them. The resultant candidates are then followed-up to verify their compliance with the hypothesis of being a periodic gravitational source, to refine their parameters and to compute other (like polarization).

An important part of the package is the simulation modules.

This guides ends with a report of various tests done of some parts of this package.

More information can be found on the programming guides and other documents:

- Snag2_PG
- PSS_PG
- PSS_astro_PG

- PSS_astro_UG
- PSS_Hough_PG
- Supervisor_PG

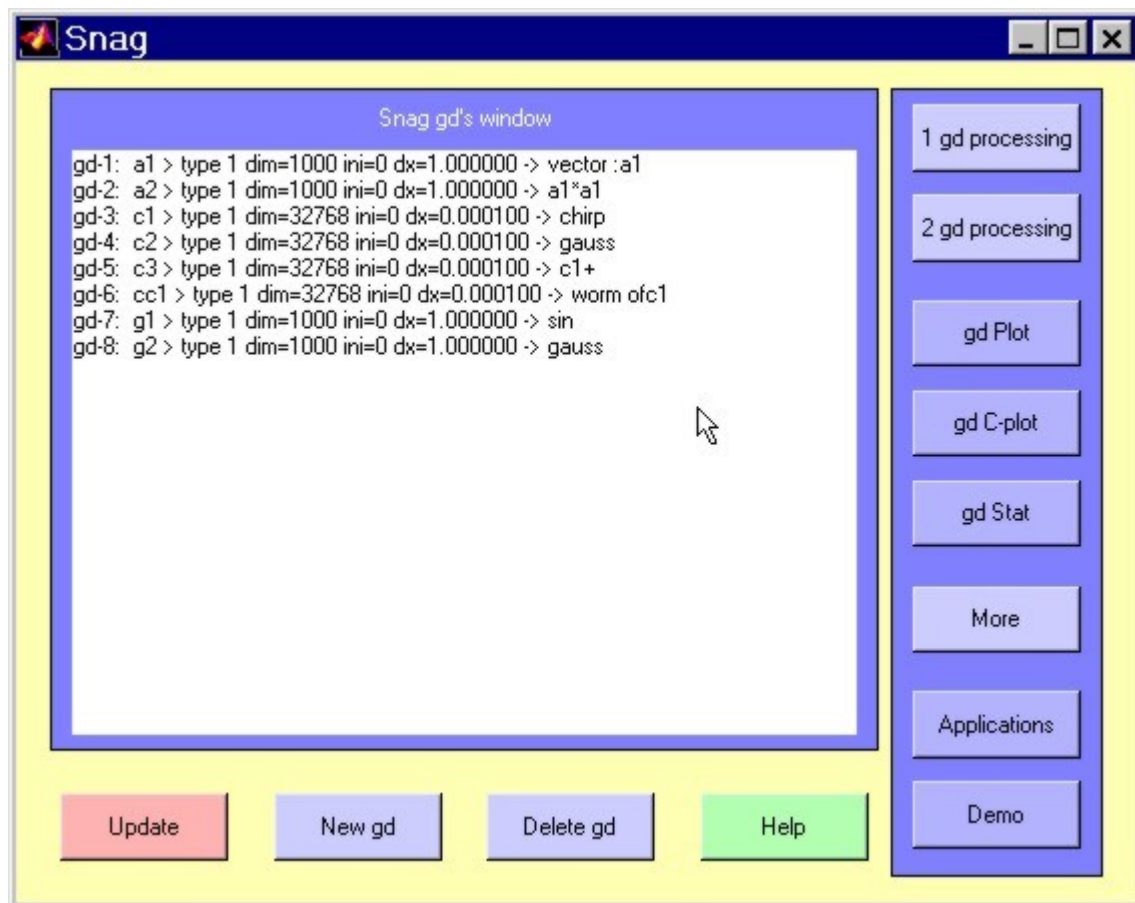
Programming environments

Matlab

For the MatLab environment the Snag toolbox is used. It contains more than 800 m functions (February 2004) and has PSS as one of the projects regarding the gravitational waves (in **snag\projects\gw\pss**). It is almost completely independent from other toolboxes.

There are two useful interactive gui programs in Snag:

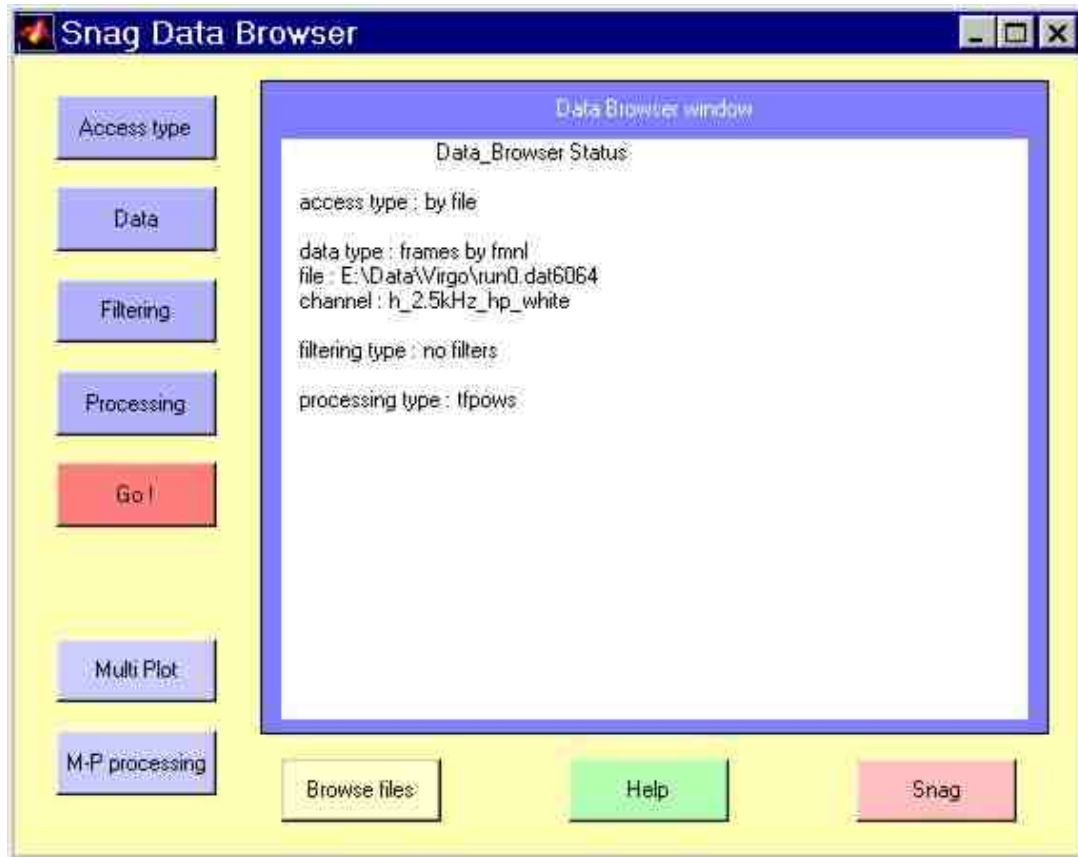
- ➡ **snag** , provides a GUI access to the **Snag** functionalities. It can be used “stand-alone”, or in conjunction with the normal Matlab prompt use of **Snag**. At the Matlab command prompt, type **snag** . A window appears:



It has a text window, where are listed the **gds** that have been created and some buttons.

For a more extensive description of this, see the Snag2_UG.

- ➡ **data_browser** , that is a **Snag** application (part of the **gw** project) to access and simulate gravitational antennas data. It is started by typing **data_browser** (or just **db**, if the alias is activated). This opens a window



with a text window and some buttons. The text window shows the “status” of the **DataBrowser** , as is due to the default and user’s settings.

The parts that are developed in this environment are labeled **[MatLab environment]**.

The gw project

Inside Snag, a gravitational wave project has been developed. An important of this project is the DataBrowser (showed in the preceding sub-section).

Other parts of this project are:

- **astro** , on astronomical computations (coordinate conversion, doppler computations)
- **time** , with a set of functions dealing with the time. Among the others:
 - conversions between mjd (modified julian date), gps and tai times
 - sidereal time
 - conversions between vectorial and string time formats
- **sources** , about gravitational sources (pulses, chirps and periodic signals)
- **pss** , specifically for the PSS software (see the sub-section devoted to it in Simulation and theory)
- **gw_sim**, with data simulation
- **radpat** , for the radiation pattern and response of antennas (sidereal response of an antenna, sky coverage,...)

C

The C environment contains a library and some module. The library contains:

- **pss_snag** : routines to operate with the snag objects (GD, DM, DS, RING, MCH)
- **pss_math** : basic mathematical routines
- **pss_serv** : service routine (among the others, vector utilities, string utilities, bit utilities, interactive utilities, “simple file” management)
- **pss_gw** : physical parameters management
- **pss_astro** : astronomical routines
- **pss_frame** : routines for frame format access
- **pss_r87** : routines for r87 format access
- **pss_sfc** : routines for the sfc file formats management
- **pss_snf** : routines for snf format management (partially obsolete)

The other modules are:

- **pss_bench** : for computer benchmarks
- **pss_math** : basic mathematical routines
- **pss_sfdb** : for short FFT data base and peak maps creation and management
- **pss_hough** : for hough transform
- **pss_cohe** : for the coherent step of the hierarchical search
- **pss_ss** : Hough tasks management and supervision

The parts that are developed in this environment are labeled **[C environment]**.

SFC formats

Basics

The basic feature of the file formats here collected is the ease of access to the data.

The "ease of access" means:

- the software to access the data consists in a few lines of basic code
- the data can be accessed easily by any environment and language
- the byte level structure is immediately intelligible
- no unneeded information is present
- the number of pointers and structures is minimized
- the structure fits the needs
- the access is fast and, possibly, direct
- the need for generality is tempered by the need for easiness.

The collection is composed by:

- **sds**, *simple data stream* format, for finite or "infinite" number of equispaced samples, in one or more channels, all with the same sampling time
- **sbl**, *simple block* data format, in a more general case; a block can contain one or more data types: any block have the same structure (i.e. the sequence and the format of the channels is the same) and the same length (i.e. the number of data in a block for a certain channel, is always the same).
- **vbl**, *varying length block* data format, where the structure of all the blocks is the same, but the length can be different.
- **gbl**, *general block* data format: it is not a format, but practically a sequence of superblocks, each following one of the preceding formats; it is a repository of data, not necessary well structured for an effective analysis, but good for storage, exchange, etc..

A set of files can be:

- **internally collected**, i.e. ordered serially or in parallel using the internal file pointers (for example subsequent data files, or to put together different sampling time channels)
- **externally collected**, i.e. logically linked by a collection script file, as it happens for internal collecting
- **embedded** in a single file, with a toc at the beginning or at the end. This is the case of the gbl files.

A file can be **wrapped** by adding one or more external headers (for example describing the computer which wrote the file).

The SFC data formats are presented in the Snag2 Programming Guide (Snag2_PG.pdf).

Compressed data formats

LogX format

This is a format that can describe a real number (float) with little more than 16, 8, 4, 2 or 1 bits. X indicates this number of bits.

It uses normally a logarithmic coding, but can use also linear coding and, in particular cases, the normal floating 32-bit format. In the case that all the data to be coded are equal, only one data is archived (plus the stat variable).

It best applies to sets of homogeneous numbers.

Let us divide the data in sets that are enough homogeneous, as a continuous stretch of sampled data. The conversion procedure computes the minimum and the maximum of the set and the minimum and the maximum of the absolute values of the set, checks if the numbers are all positive or negative, or if are all equal, then computes the better way to describe them as a power of a certain base multiplied by a constant (plus a sign). So, any **non-zero** number of the set is represented by

$$x_i = S_i * m * b^E_i$$

or, if all the number of the set have the same sign,

$$x_i = S * m * b^E_i$$

where

- S_i is the sign (one bit)
- m is the minimum absolute value of the numbers in the set
- b is the base, computed from the minimum and the maximum absolute value of the numbers of the set
- E_i is the (positive) exponent (15 or 16 bits for Log16, 7 or 8 bits for Log8, and so on).

The coded datum 0 always codes the uncoded value 0 (also if such a value doesn't exist).

m , b , and a control variable that says if all the number are positive, negative or mixed are stored in a header. The data bits contain S and E or only E .

The minimum and maximum values can be imposed externally, as saturation values.

In case of mixed sign data, in order to have automatic computation of m and b , an **epsval** (a minimum non-zero absolute value) should be defined. If this is put to 0, this value is substituted with the minimum non-zero absolute datum.

The zero, in the case of mixed sign data, is coded as "111...11", while "000...00" is the code for the number m ("1000...00" is $-m$, "0111...11" is the maximum value and "111...110" the minimum).

The mean percentage error in the case of a gaussian white sequence is, in the case of Log16, better then 10^{-4} .

Also a linear coding is possible:

$$x_i = m + b * E_i$$

Also in this case, the coded datum 0 always codes the uncoded value 0 (also if such a value doesn't exist).

In case of linear coding, if the data are "mixed sign" (really or imposed) and X is 8 or 16, E is a signed integer, otherwise it is an unsigned integer: normally, in the first case, m is 0.

In case of data dimension X less than 8 (4, 2 or 1: the sub-byte coding), the logarithmic format is substituted by a look-up table format. In such case, a look-up table of $(2^X - 1)$ fixed thresholds t_k ($0 < k < 2^X - 2$), in ascending order, must be supplied. Data $< t_0$ are coded as 0, data between t_{k-1} and t_k are coded as k and data greater than the last threshold are coded as 11..1. In the case of linear sub-byte coding, the coded data are unsigned.

Here is a summary of the LogX format:

Number of bits	32	16	8	4	2	1	0
Coding	float	2 linear 2 logarithmic	2 linear 2 logarithmic	linear look-up	linear look-up	linear look-up	constant

Logarithmic coding can be done using X or X-1 bits for the exponent, depending if the last bit is used for the sign. Linear coding can be (for X = 8 or 16) signed or unsigned integer coded.

Linear and logarithmic coding can be adaptive.

So, totally, we have 16 different LogX formats (7 linear, 4 logarithmic, 3 look-up table, 1 float and 1 constant float), 11 of which can be adaptive.

Sparse vector formats

Sparse vector is a vector where most of the elements are 0. We call "density" the percentage of non-zero elements. Sparse matrixes are formed by sparse vectors.

Sometimes (binary matrixes) the non-zero elements are all ones and sometimes they are also aggregated. In this last case the binary derivative (0 if no variation, 1 if a variation is present) is often a sparse vector with lower density value.

We represent sparse vectors with the "run-of-0 coding". It consists in giving just the number of subsequent zeros, followed by the value of the non-zero element. In the case of binary vectors, the value of the non-zero element is not reported.

Examples:

{1.2 0 0 0 0 0 3.2 0 0 0 0 0 0 2.3 0 0 0 0 0 0 0 0 3.0 0 0 0 2.}

coded as {0 1.2 5 3.2 6 2.3 8 3.0 3 2.}

binary case:

{0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 1 0 0 0}

coded as {3 6 8 0 3}

aggregate binary case:

{1 1 1 1 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1}

binary derived as

{1 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0}

coded as

{0 2 7 3 5 4 9 2}.

In practice the number of subsequent zeroes is expressed by an unsigned integer variable with $b = 4, 8, 16$ or 32 bits; one is added to the coded values, in such a way that the value 0 is an escape character used if more than $2^b - 2$ zeroes should be represented; in such case the datum is put in a side array of uint32.

In practice, there are 5 different cases:

sparse, non-binary	\Rightarrow	the 0-runs and the non-zero elements
sparse, binary	\Rightarrow	only the 0-runs of the sequence
sparse, derived binary	\Rightarrow	only the 0-runs of the derived sequence
non-sparse, non-binary	\Rightarrow	normal vector (a float per element)
non-sparse, binary	\Rightarrow	one bit per element

PSS SFC files

The PSS (Periodic Source Search) project uses many different types of data to be stored. Namely:

- **h-reconstructed sampled data, raw and purged**
- **Short FFT data bases**
- **Peak maps**
- **Hough maps**
- **PS candidates**
- **Events**

Each of these has a peculiar type of SFC.

- **h-reconstructed sampled data, raw and purged**

This type of data are normally stored with simple SDS.

- **Short FFT data bases**

The data are stored in a SBL file.

In the user field there are other information like:

- [I] **FFT length** (number of samples of the time series)
- [I] **Interlacing size** (number of interlaced samples)
- [D] **sampling time** of the time series
- [S] **window** (used on the time series)

The blocks contain:

- one half of the FFT of purged sampled data
- one short power spectrum
- one one-minute mean vector
- a set of parameters as:
 - [I] **number of added zeros** (for errors, holes or over-resolution)
 - [D] **time stamp** of the first time datum (**mjd**)
 - [D] **time stamp** of the first time datum (**gps time**)
 - [D] **fraction** of the FFT time that was padded with zeros
 - [D] **velocity of the detector** at time of the first datum (**vix,viy,viz**: coordinates in Ecliptic reference frame, fraction of c)
 - [D] **velocity of the detector** at time of the middle datum (**vmx,vmy,vmz**: coordinates in Ecliptic reference frame, fraction of c)
 - [D] **velocity of the detector** at time of the last datum (**vfx,vfy,vfz**: coordinates in Ecliptic reference frame, fraction of c)

- [D] **mean velocity of the detector** during the FFT time (vx,vy,vz: coordinates in Ecliptic reference frame, fraction of c)
- [D] initial **sidereal time**

- **Peak maps**

The data are stored in a VBL file. The structure is similar to that of the SFDB, but a peak vector takes the place of the FFT. the format of the peak vector is a sparse binary vector, so the real length of each block is not constant.

- **Hough maps**

The data are stored in SBL or VBL files. The parameter to be stored in each block (containing a single Hough map) are:

- the **length** of the record
- the **parameters** of the hough map (**amin, da, na, dmin, da, nd**)
- the **spin down parameters** (**nspin, spin1,spin2,...**)
- the **number of used periodograms** and the type (**interlaced, windowed,...**)
- the **initial times and length** of each periodogram
- the **type and the parameters** of the **threshold**

- **PS candidates and Events**

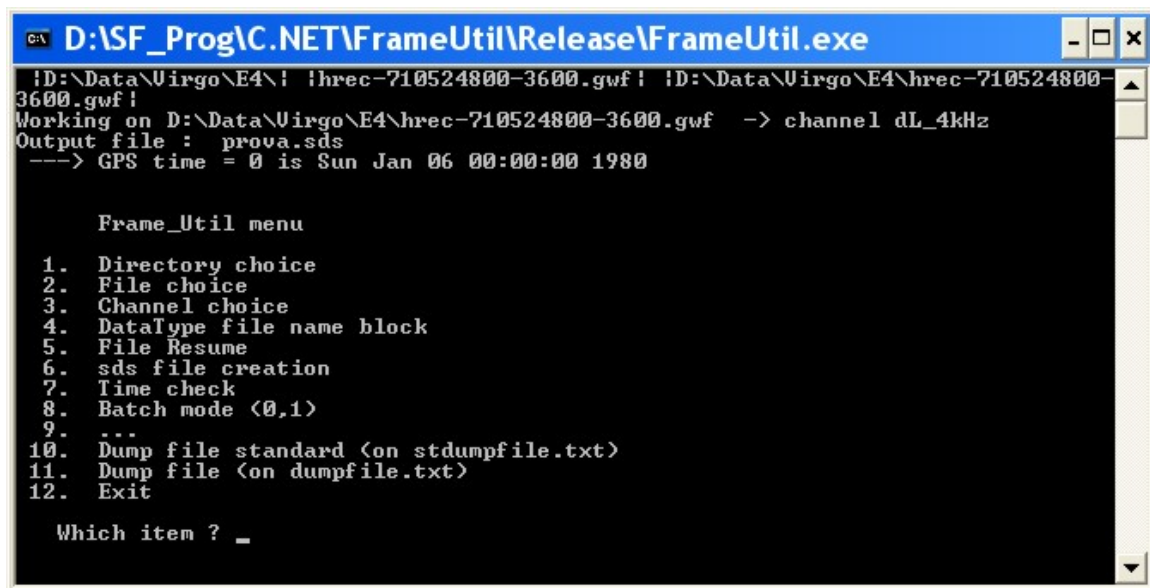
- These data could be stored in an SDS file, with many channels, but, for the necessity of easy random access needed for such data bases, a peculiar format will be used.

Data preparation

Format change

[C environment]

In order to use some Snag features in a more proficient way, the frame format data must be must be converted in the sds format. This can be accomplished by the interactive program FrameUtil.exe:



```
C:\ D:\SF_Prog\C.NET\FrameUtil\Release\FrameUtil.exe
!D:\Data\Virgo\E4\! hrec-710524800-3600.gwf! !D:\Data\Virgo\E4\hrec-710524800-
3600.gwf!
Working on D:\Data\Virgo\E4\hrec-710524800-3600.gwf -> channel dL_4kHz
Output file : prova.sds
---> GPS time = 0 is Sun Jan 06 00:00:00 1980

Frame_Util menu

1. Directory choice
2. File choice
3. Channel choice
4. DataType file name block
5. File Resume
6. sds file creation
7. Time check
8. Batch mode <0,1>
9. ...
10. Dump file standard <on stdumpfile.txt>
11. Dump file <on dumpfile.txt>
12. Exit

Which item ? _
```

Very useful are the two dump file facilities, that give a resume of all the frames.

The program can be used also in batch, creating a batch file as this:

8	! ask batch mode
1	! sets batch mode
1	! ask item choice of the directory
X:\x\E4\h_recon\	! the directory
3	! ask channel choice
dL_20kHz	! channel
4	! ask item DataType file name block
hout	! the block
2	! ask item File choice
hrec-710517600-3600.gwf	! the file
6	! creates the sds file
2	! ask item File choice
hrec-710521200-3600.gwf	!
6	! creates the sds file
2	! ask item File choice
hrec-710524800-3600.gwf	!
6	! creates the sds file

To create easily the batch file, create a list of the files to be converted and then edit it. In Windows the command is

```
dir /b > list.txt
```

and can be issued with the command file **to_list.bat** ; it creates a file list.txt, to be edited to create the batch command file.

To start the program a batch command can be created containing something like

```
D:\SF_Prog\C.NET\FramUtil\Release\frameutil < batchwork.txt > out.txt
```

If we have a set of sds files, we can "concatenate" them, i.e. put in each of them the correct values for filspre and filspost, so the data can be seen as a continuous stream, and one can access at any of them pointing to any file of the chain (also not containing the given datum). This concatenation is performed, for example, by the function

- ➡ **sds_concatenate(folder,filelist)** , where **folder** is the folder containing the files and **filelist** a file containing the file list (similar to the above list.txt) in the correct order. **Be sure that the files in the list are in the correct order !**

A more complex operation on a set of sds files is performed by

- ➡ **sds_reshape(listin,maxndat,folderin,folderout)** , that constructs a new set of sds files with different maximum length and concatenates them. In this way a more efficient data set is built.
 - **listin** is a file containing the file list (similar to the above list.txt) **in the correct order**
 - **maxndat** is the maximum number of data for a channel
 - **folderin** and **folderout** are the folders containing the input and output data

When all the files of a run are produced and concatenated (possibly with sds_reshape), they should be checked by

- ➡ **check_sds_conc(outfile)** , that analyzes the set of files, producing a report in outfile (or on the screen, if outfile is absent). Here is an example of one of these reports (c5-data.check):

```
VIR_hrec_20041203_004502_.sds 03-Dec-2004 00:45:02.000000 duration: 3790.000000 s chs:
h_4kHz - err = 0.000000
1598.000000 s --> HOLE at 03-Dec-2004 01:48:12.000000
VIR_hrec_20041203_021450_.sds 03-Dec-2004 02:14:50.000000 duration: 12500.000000 s chs:
h_4kHz - err = 0.000000
VIR_hrec_20041203_054310_.sds 03-Dec-2004 05:43:10.000000 duration: 1949.000000 s chs:
h_4kHz - err = 0.000000
50.000000 s --> HOLE at 03-Dec-2004 06:15:39.000000
VIR_hrec_20041203_061629_.sds 03-Dec-2004 06:16:29.000000 duration: 4490.000000 s chs:
h_4kHz - err = 0.000000
13172.000000 s --> HOLE at 03-Dec-2004 07:31:19.000000
```

```

VIR_hrec_20041203_111051_.sds 03-Dec-2004 11:10:51.000000 duration: 881.000000 s chs:
h_4kHz - err = 0.000000
57.000000 s --> HOLE at 03-Dec-2004 11:25:32.000000
VIR_hrec_20041203_112629_.sds 03-Dec-2004 11:26:29.000000 duration: 803.000000 s chs:
h_4kHz - err = 0.000000
23392.000000 s --> HOLE at 03-Dec-2004 11:39:52.000000
VIR_hrec_20041203_180944_.sds 03-Dec-2004 18:09:44.000000 duration: 776.000000 s chs:
h_4kHz - err = 0.000000
106.000000 s --> HOLE at 03-Dec-2004 18:22:40.000000
VIR_hrec_20041203_182426_.sds 03-Dec-2004 18:24:26.000000 duration: 9454.000000 s chs:
h_4kHz - err = 0.000000
51.000000 s --> HOLE at 03-Dec-2004 21:02:00.000000
VIR_hrec_20041203_210251_.sds 03-Dec-2004 21:02:51.000000 duration: 15.000000 s chs:
h_4kHz - err = 0.000000
42.000000 s --> HOLE at 03-Dec-2004 21:03:06.000000

```

.....

Summary

```

133 files start: 03-Dec-2004 00:45:02.000000 end: 06-Dec-2004 14:28:21.000000 Tobs =
3.571748 days

```

```

129 holes of total duration 141025.000000 s percentage = 0.456985

```

Data selection

[MatLab environment]

Basic sds operations

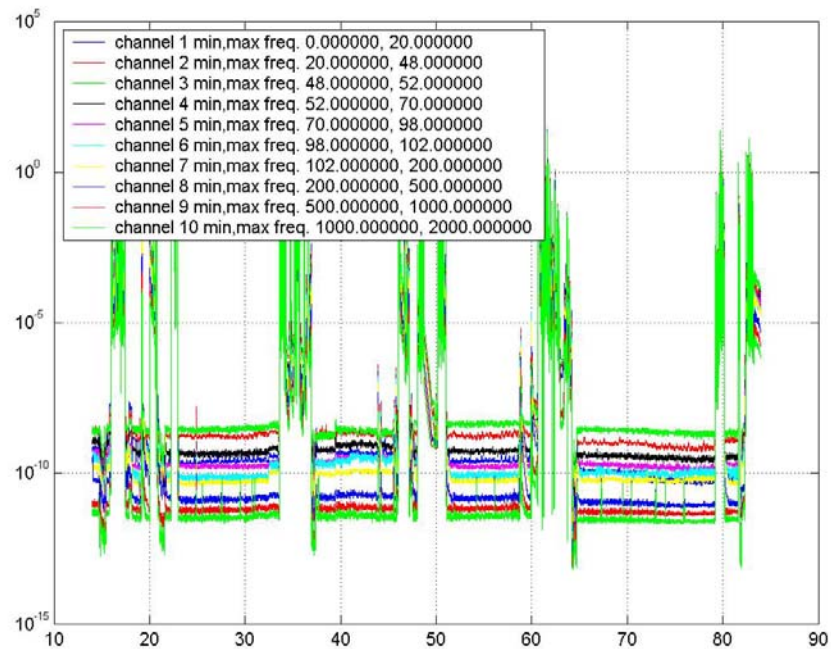
If the sampled data are in the sds format, it is easy to perform a variety of tasks. Here we will speak of higher level tasks (lower levels are discussed in the programming guides). Among the others, of particular interest for the PSS :

- ➡ **sfc_=sfc_open(file)** , that outputs the **sfc** structure of the file
- ➡ **[chss,ndatatot,fsamp,t0,t0gps]=sds_getchinfo(file)** , that shows the UTC time and outputs channels (in a cell array), the total number of data, the sampling frequency and the initial time both in MJD (modified julian date) and gps.
- ➡ **g=sds2gd_selt(file,chn,t)** , creates a gd from file, channel number chn and t = [initial time, duration]; if the parameters are not present, asks interactively.
- ➡ **sds_spmean(frbands,file,chn,ffilen,nmax)** , creates an sds file, named **psmean.sds**, containing the spectral means for many different bands. **frbands** is an Nx2 matrix containing the bands; if it is not present, it can be input as a text file like, for example,

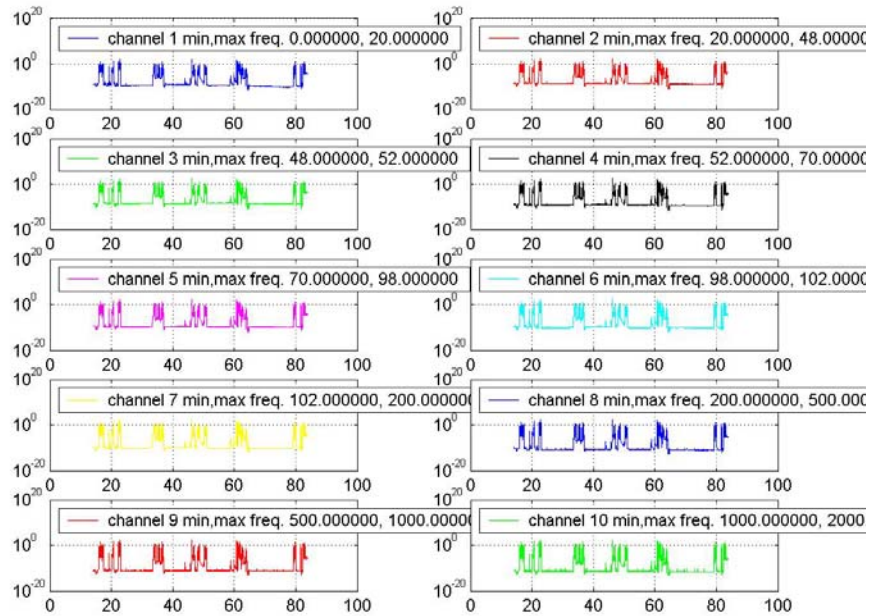
```
0 20
20 48
48 52
52 70
70 98
98 102
102 200
200 500
500 1000
1000 2000
```

file and **chn** are the file and the channel number, **ffilen** is the length of the FFT and **nmax** the maximum number of output data (put a big number and all the concatenated files will be analyzed).

- ➡ **m=sds2mp(file,t)** , creates an **mp** (multi-plot structure) from an sds file (the command can be issued without parameter and asks interactively). For example, it can be applied to the spectral mean sds file created by **sds_spmean**. The mp structure can be showed by **mp_plot(m,3)** (m is the mp and 3 means log y), obtaining (on E4 data of the CIF)



or else (among other choices)



The abscissa is in hours from the 0 hours of the first day.

➡ **crea_ps(sdsfile,chn,lfft,red)** , creates an **sbl** file containing power spectra of data (similar to that produced for the **sfdb**), from channel number **chn**, a "big FFT length" **lfft** and a length of the power spectra **lfft/red**.

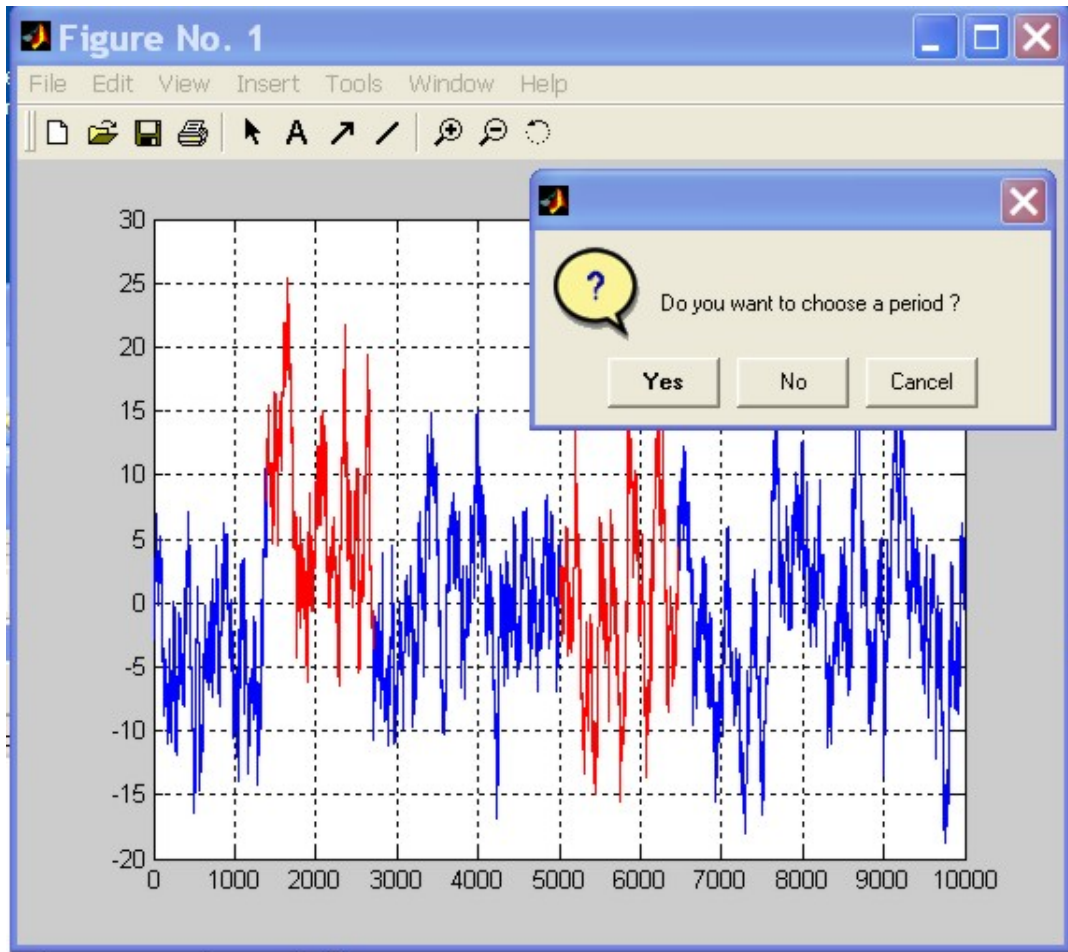
Choice of periods

[MatLab environment]

The choice of the periods on which the SFDB should be created (and then are to be analyzed) can be done by the use of the Virgo data quality information and of the basic instruments like those shown in the preceding section.

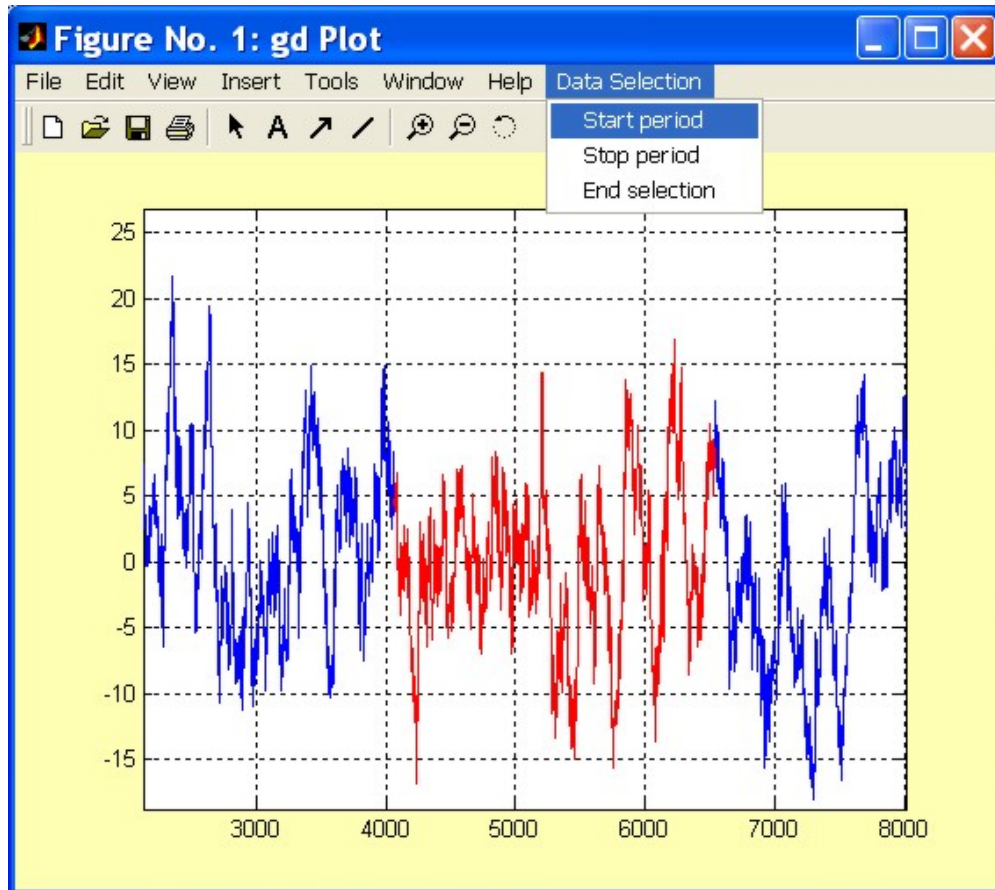
In Snag there are some useful interactive functions that helps in choosing prtiolds:

- ➡ **xx=sel_absc(typ,y,x,file)** , the easiest one, where **typ** (0,1,2,3) indicates the type of plot (simple, logx, logy, logxy), **y** is a gd or an array, **x** is simply 0 (if y is a gd) or the abscissa array, **file** (if present) is the file to put the output, i.e. the starting and ending points of the chosen periods; **xx** is an (n,2) array with the bounds of the chosen periods. This program is very simple to use: you can directly choose the start and stop abscissa of as many periods you want; when you chose the stop point, the chosen period is colored in red and you are prompted if you choose another period

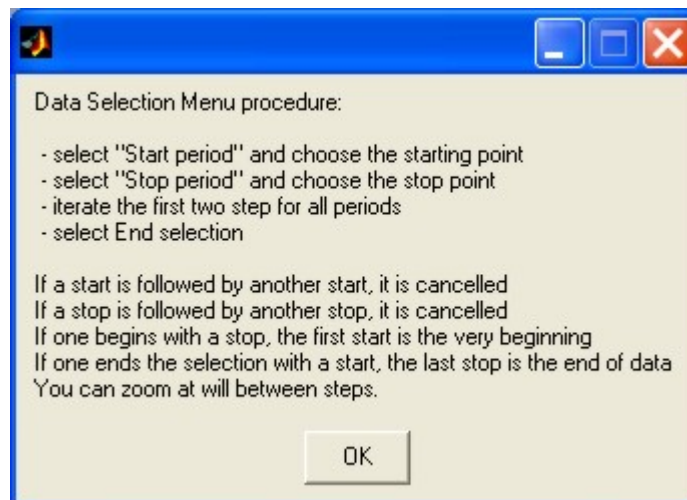


The problem with this function is that it is not possible to zoom the plot for more precise choice.

- ➡ **sel_absc_hp(typ,y,x)** , permits the use of the zoom and so an high precision choice. It uses global variables (**ini9399**, **end9399**, **n9399** as the beginning times, ending times and number of periods); The data of the periods are put in a file named **fil9399.txt**. The input variables **typ**, **y** and **x** have the same use of the function **sel_absc**.



There are some easy rules that appear at beginning:



Search for events

[MatLab environment]

Filtering in a ds framework

The ev-ew structures

The event management is done by the use of the **ev-ew** Snag structure (see the programming guide Snag2_PG.pdf). An event is defined by a set of parameters, like

- the (starting) time of the event (in days, normally mjd)
- the time of the maximum (in days, normally mjd)
- the channel
- the amplitude
- the critical ratio
- the length (in seconds)
- ...

The difference between the **ev** and **ew** structures is that the first describes a single event (so a set of events is an array of structures), the second describes a set of events. The **ew** structure is normally more efficient, but the **ev** structure is more rich (it can contain also the shape of the event). The two function **ew2ev** and **ev2ew** transform one type in the other (losing the shape, if present).

A set of events is associated to a channel structure that describes the channels that produced the events, constituting a new event/channel structure **evch**.

There is a number of auxiliary functions to manage events:

- ➡ **chstr=crea_chstr(nch)** , creates a channel structure for events; nch is the number of channels
- ➡ **evch=crea_ev(n,chstr,tobs)** , creates an event-channel structure, simulating n events in the time span tobs, and with the channel structure chstr
- ➡ **evch=crea_evch(chstr,evstr)** , creates an event-channel structure, from a channel structure and an event structure
- ➡ **[fil,dir,fid]=save_ech(ch,direv,fil,mode,capt)** , save a channel structure in an ascii file that can be edited; ch is the channel structure, direv and fil are the default folder and file, mode is 0 for standard, 1 for the full evch, capt is the caption
- ➡ **[fil,dir]=save_ev(ev,direv,fil,mode,fid,capt)** , save an event structure in an ascii file that can be edited; ev is the event structure, direv and fil are the default

folder and file, mode is 0 for standard, 1 for the full evch, fid is the file identifier (or 0), capt is the caption

- ➡ **save_evch(evch)** , saves an evch structure in Matlab format
- ➡ **load_evch** , interactively loads an evch structure in Matlab format
- ➡ **eo=sort_ev(ei)** , time sorts an event structure
- ➡ **out=ev_sel(in,ch,t,a,l)** , selects events on the basis of the channel, time occurrence, amplitude and length.
 - **in** and **out** are the input and output **evch** structure,
 - **ch** , if it is an array, it is the probability selection of different channels (if < 0, the channel disappear), otherwise is not used;
 - **t, a, l** , if it is an array of length 2, defines the interval of acceptance; if the first element is greater than the second, they defines the interval of rejection
- ➡ **chstr=stat_ev(evch)** , statistics for events
- ➡ **dd=ev_dens(evch,selch,dt,n)** , event densities;
 - **evch** event/channel structure
 - **selch** selection array for the channels (0 exclude, 1 include)
 - **dt** time interval
 - **n** number of time intervals
- ➡ **ev_plot(evch,type)** , plots events. type is:
 - 0. simple
 - 1. amplitude colored
 - 2. length colored
 - 3. both
 - 4. stem3 amplitude
 - 5. stem3 length
 - 6. stem3 both

Coincidences

To study coincidences between events a set of functions is provided:

- ➡ **[dcp,ini3,len3,dens] = ev_coin(evch,selch,dt,n,type,coifun)** , creates a delay coincidence plot (**dcp**) and finds coincident events (**ini3, len3** are the initial times and lengths, **dens** is the event density, if used). In input:
 - **evch** is an event/channel structure

- **selch** is a selection array, with the dimension of the number of channels, that defines which channels are to be put in coincidence: every channels can be
 - 0 excluded
 - 1 put in the first group
 - 2 put in the second group
 - 3 put in both
- **dt** is the time resolution (s)
- **n** is the number of delays for each side
- **type** an array indicating the coincidence type:
 - type(1)** : 1 only event maxima
2 whole length coincidences
 - type(2)** : 1 normal
2 density normalized
 - type(3)** : density time scale (s) for density normalization
- **coincfun** if exists, external coincidence function is enabled. The coincidence function is > 0 if the events are "compatible"; the inputs are (len1,len2,amp1,amp2), that are the lengths and amplitudes for the two coincident event.

It produces the plot of the delay coincidences and its histogram.

- ➡ **[dcp,in3,len3]=vec_coin(in1,len1,in2,len2,dt,n,coincfun,a1,a2)** , is one of the coincidence engines used inside **ev_coin**. It considers the length of the events
- ➡ **[dcp,in3]=vec_coin_nolen(in1,in2,dt,n,coincfun,a1,a2)** , is one of the coincidence engines used inside **ev_coin**. It considers the length of the events as **dt**.
- ➡ **ch=ev_corr(evch,dt,mode)** , computes and visualize the correlation matrix between all the channels. mode = 1 is for symmetric operation, mode = 0 is for "time arrow" coincidence (causality). It produces also the map of the matrix.
- ➡ **evcho=cl_search(evchi,dt)** , identifies cluster of events and labels the events with the cluster index

Event periodicities

An important point in the event analysis is the study of periodicities. This is performed by the following functions:

- ➡ **sp=ev_spec(evch,selev,minfr,maxfr,res)** , that performs the event spectrum; in input we need:

- **evch** event/channel structure
- **selch** channel selection array (0 excluded channel)
- **minfr** minimum frequency
- **maxfr** maximum frequency
- **res** resolution (minimum 1, typical 6)

➡ **pd=ev_period(evch,selch,dt,n,mode,long,narm)**, event periodicity study (phase diagram); in input we need:

- **evch** event/channel structure
- **selch** channel selection array (0 excluded channel)
- **dt** period
- **n** number of bins of the phase diagram (**pd**)
- **mode** = 0 simple events
= 1 density normalization; mode(1) = 1, mode(2) = bin width (s)
= 2 amplitude; mode(1) = 2, mode(2) = 0,1,2 (normal, abs, square)
- **long** longitude (for local periodicities (local solar and sidereal))
- **narm** number of harmonics

SFDB

Theory

The maximum time length of an FFT such that a Doppler shifted sinusoidal signal remains in a single bin is

$$T_{\max} = T_E \cdot \sqrt{\frac{c}{4\pi^2 R_E \nu_G}} \approx \frac{1.1 \cdot 10^5}{\sqrt{\nu_G}} s$$

where T_E and R_E are the period and the radius of the “rotation epicycle” and ν_G is the maximum frequency of interest of the FFT.

Because we want to explore a large frequency band (from ~10 Hz up to ~2000 Hz), the choice of a single FFT time duration is not good because, as we saw,

$$T_{\max} \propto \nu_G^{-\frac{1}{2}}$$

so we propose to use 4 different SFDB bands:

Short FFT data base

	Band 1	Band 2	Band 3	Band 4
Max frequency per band	2000.0000	500.0000	125.0000	31.2500
Observed bands	1500.0000	375.0000	93.7500	23.4375
Max duration for an FFT	2445.6679	4891.3359	9782.6717	19565.3435
Max len for an FFT (max freq)	9.7827E+06			
Length of an FFT (max freq)	8.3886E+06			
Length of the FFTs	8388608	4194304	2097152	1048576
FFT duration	2097.15	4194.30	8388.61	16777.22
Number of FFTs	9.5367E+03	4.7684E+03	2.3842E+03	1.1921E+03
SFDB storage (GB)	160.00	40.00	10.00	2.50
Storage for sampled data (GB)	80.00			
Total disk storage (GB)	292.50			

Procedure

- apply a frequency domain anti-aliasing filter and sub-sample (if 20 kHz data)
- **high events identification and removal**
 - create a stream with low and high frequency attenuation
 - identify the events on this stream (starting time and length) with the adaptive procedure. After this, this stream is no more used.
 - smooth removal of the events in the original stream (purged stream)
 - estimate the power of the purged stream every minute (or so), creating the **PTS** (power time series)
- **for each of the 4 sub-bands of the SFDB :**
 - (not for the first band) apply anti-aliasing and subsample
 - create the (windowed, interlaced) FFTs, both simple and double resolution: the simple resolution are archived for the following steps, the double is used for the peak map
 - create a low resolution spectrum estimate (**VSPS** - Very Short Power Spectrum, e.g. length 16384)

Software

[C environment]

Routines

pss_sfdb

Software

[MatLab environment]

There is also a software to create a SFDB using Matlab. This can be used for checking and particular purposes.

- ➡ **crea_sfdb(sdsfile,chn,lfft,red)** , can be used also interactively without arguments; **sdsfile** is the first sds file to be processed, **chn** is the channel number, **lfft** is the length of the (non-reduced) ffts, **red** is the reduction factor for the requested band (normally 1, 4, 16, 64). A file sfdb.sbl is created, with the fft collection.

Time-frequency data quality

[MatLab environment]

The time-frequency data quality analysis is done using

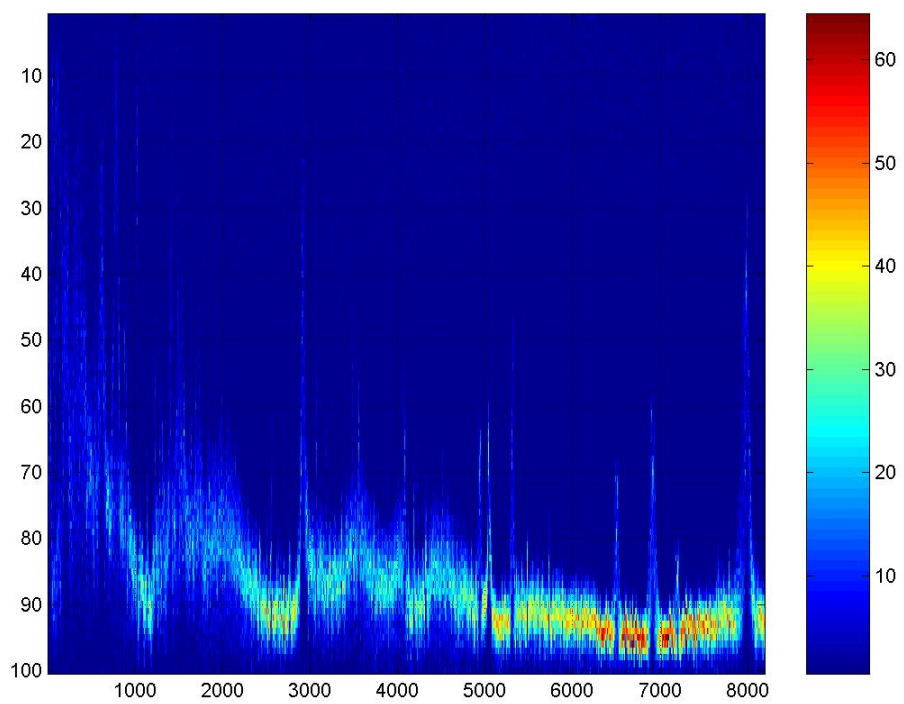
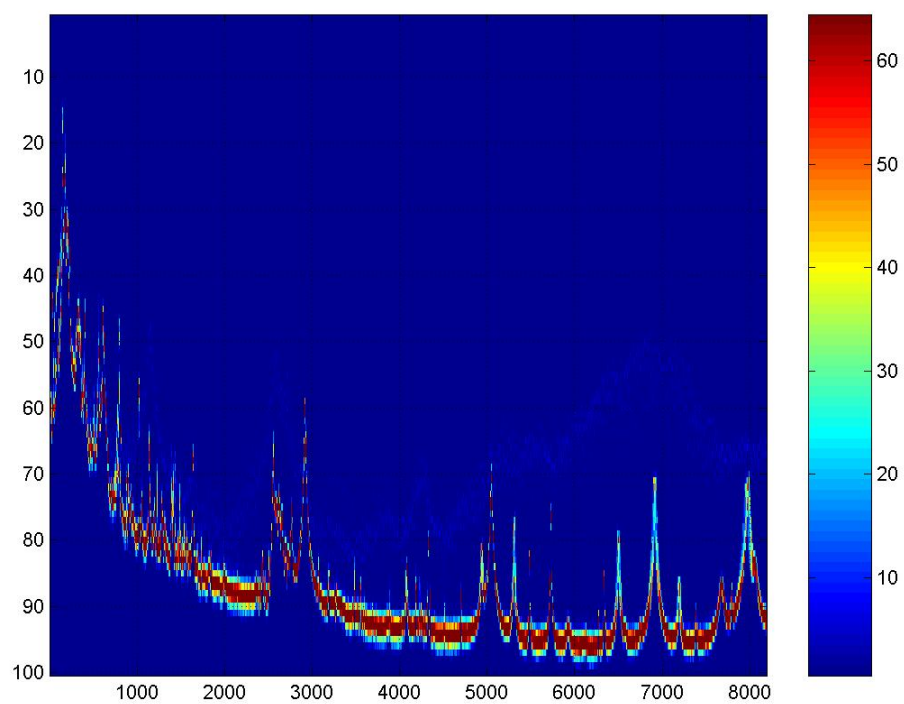
- a) the set of power spectra created together with the **SFDB**
- b) the set of power spectra created by the function **crea_ps**
- c) the high resolution periodograms obtained directly from the **SFDB** FFTs

In the cases a) and b), the time-frequency map can be imported in a **gd2** with the function

- ➡ **g2=sbl2gd2_sel(file,chn,x,y)** , where **file** is the **sbl** file containing the power spectra (for example, an **sfdb** file), **chn** the channel number in the **sbl** file, **x** a 2 value array containing the min and the max block, **y** a 2 value array containing the min and the max index of the spectrum frequencies

From this **gd2** an array can be extracted and on it the **map2hist** and **imap_hist** can be applied:

- ➡ **[h,xout]=map2hist(m,n,par,linlog)** , creates a set of histograms, one for each frequency bin, of the various spectral amplitudes of that bin at all the times. **m** is the time-frequency spectral map, **n** is the number of bins for the histograms, **linlog** (= 0,1) determines if the histogram is done on the value spectral values of the spectra or on their logarithms, **par** is the set of parameters to do the histograms:
 - if **m** is an **mx2** array, it contains the min and the max of the histograms
 - if **m** = 0, the bounds of the histograms are computed automatically by taking the minimum and the maximum of all the data
 - if **m** = 1, the bounds of the histograms are computed automatically by taking the minimum and the maximum of every bin.
- ➡ **imap_hist(h,x,y)** , is used to plot the histogram map; **h** is the histogram map, **x** is the frequency value, **y** the spectral values (if the scale is unique). Here are the two maps for the two cases of **par**=0 and **par**=1.



Peak map

From the SFDB we can obtain the "peak map", i.e. a time-frequency map containing the relative maxima of the periodograms built taking the square modulus of the short FFTs.

To obtain the peak map, the procedure is the following:

- read a short FFT of the data from the database
- using this, construct an enhanced resolution periodogram
- equalize the periodogram, using, for example, the `ps_autoequalize` procedure
- find the maxima of the equalized spectrum above a given threshold (for example 2)

The stored data can be just 1 and 0 (binary sparse matrix) or also the values of the maxima of the not equalized spectra (this in order to evaluate the "physical" amplitude (instead of the "statistical" amplitude).

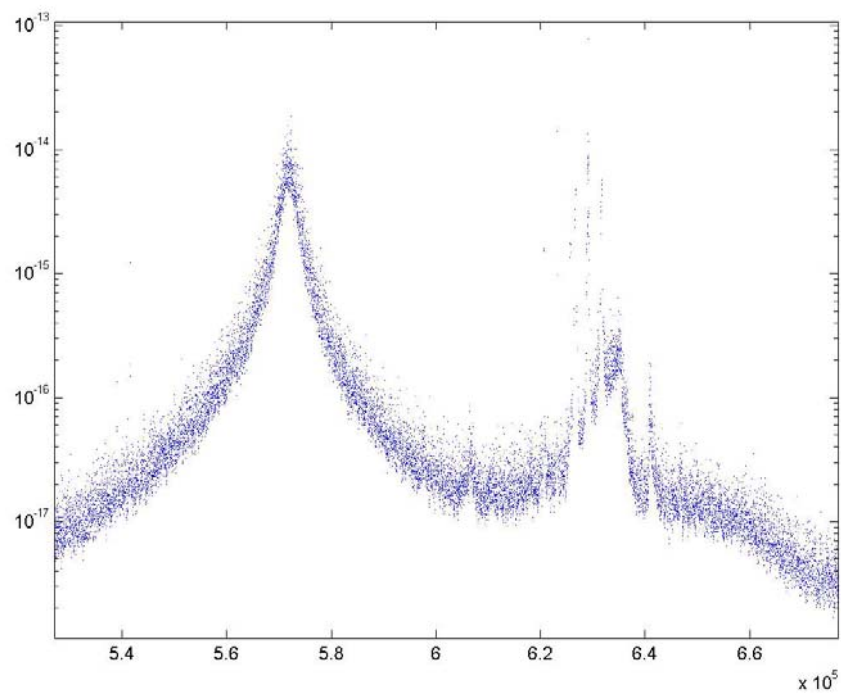
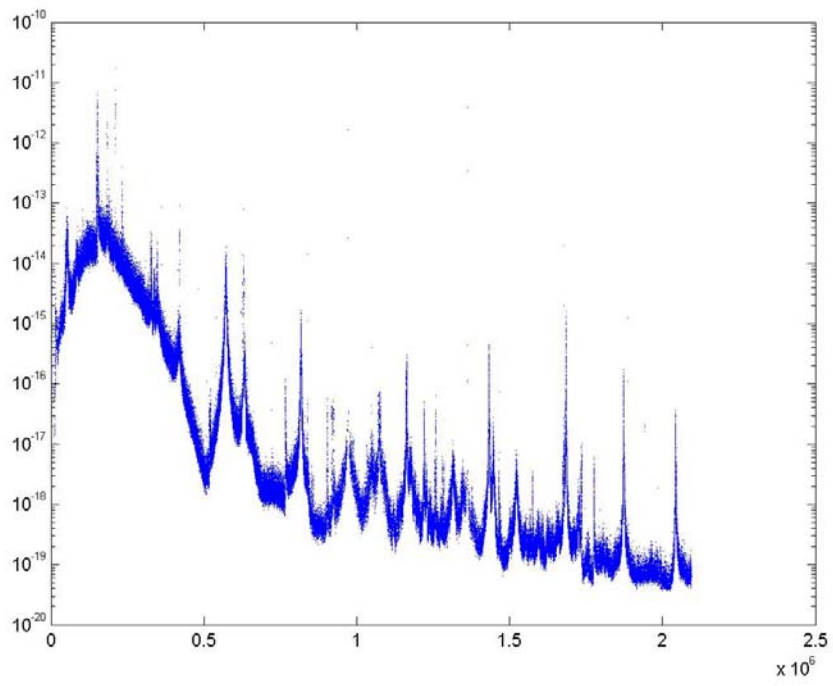
The format of the file can be `sbl` or `vbl`, depending if a normal or compressed form is chosen. The peak map file contains all the side information of the SFDB "parent" file.

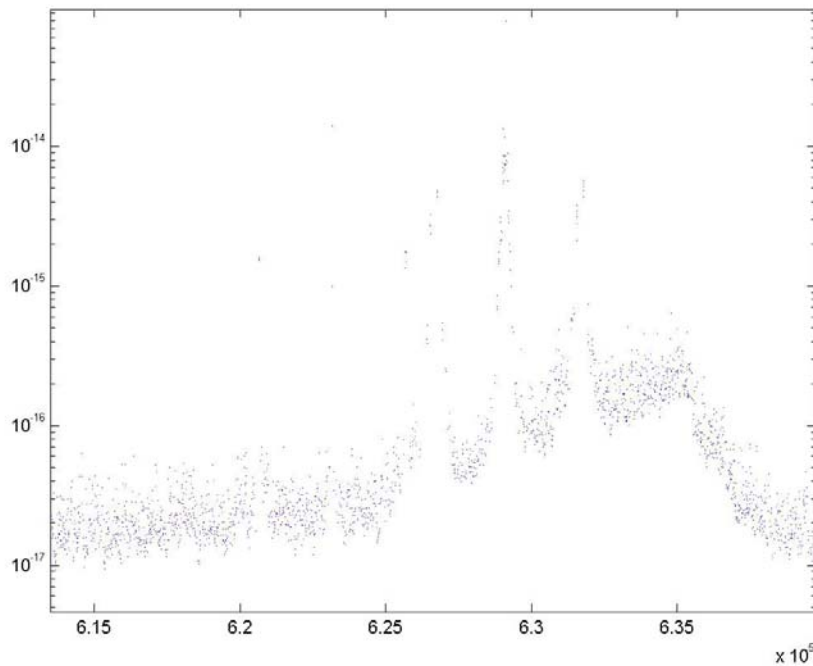
Peak map creation

[MatLab environment]

The peak map can be created by

- ➡ **`crea_peakmap(sblfil,thresh,typ)`** , where **`sblfil`** is the SFDB file, **`thresh`** is the threshold and **`typ`** is the type (0 normal with amplitudes, 1 compressed with amplitudes, 2 normal only binary, 3 compressed only binary). Here there is an image of the peak data (with zooms):





Other peak map creation procedures

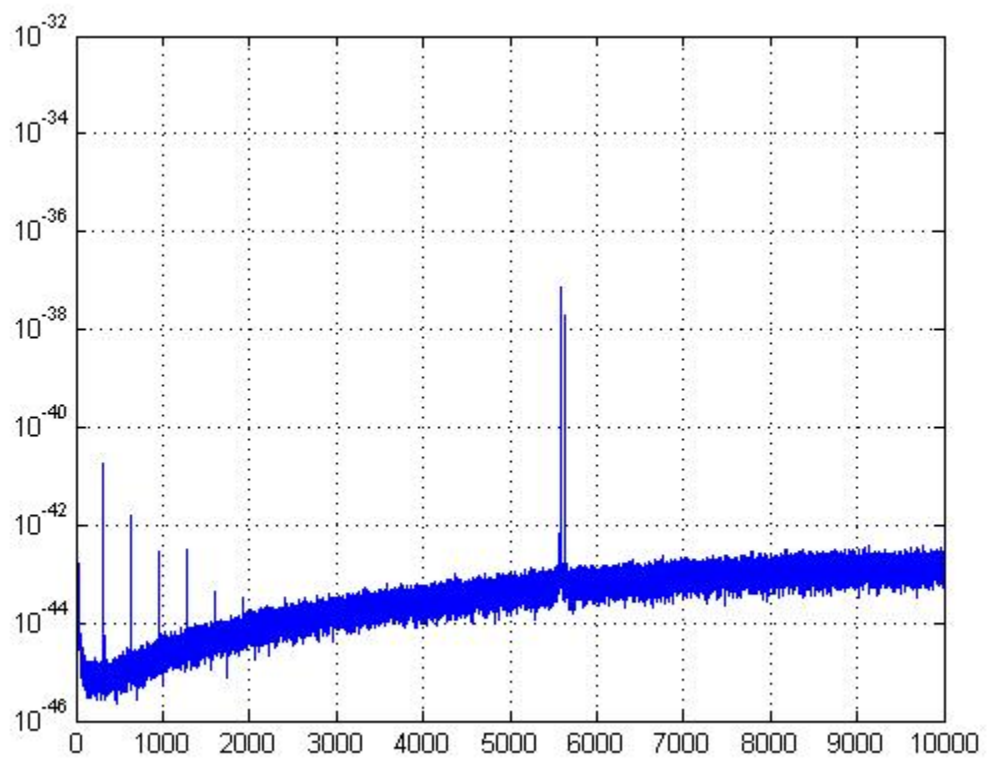
Various techniques have been studied to construct effective peak maps. The main problem is that of the presence of big peaks, that “obscure” the area around.

The basic procedure is the following:

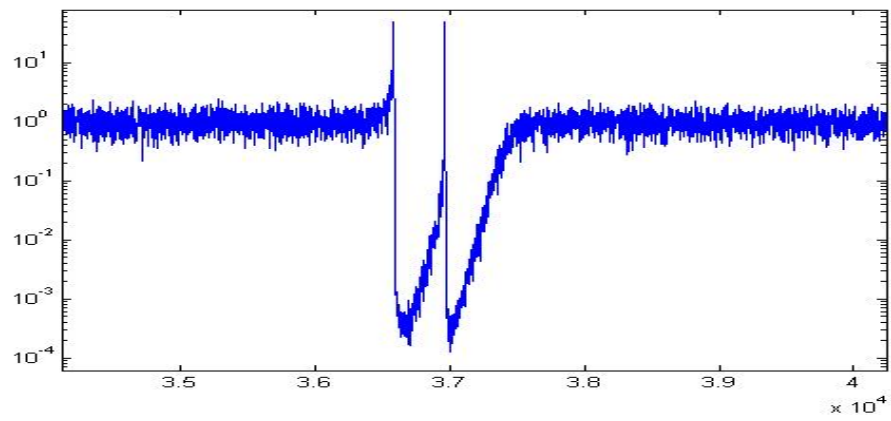
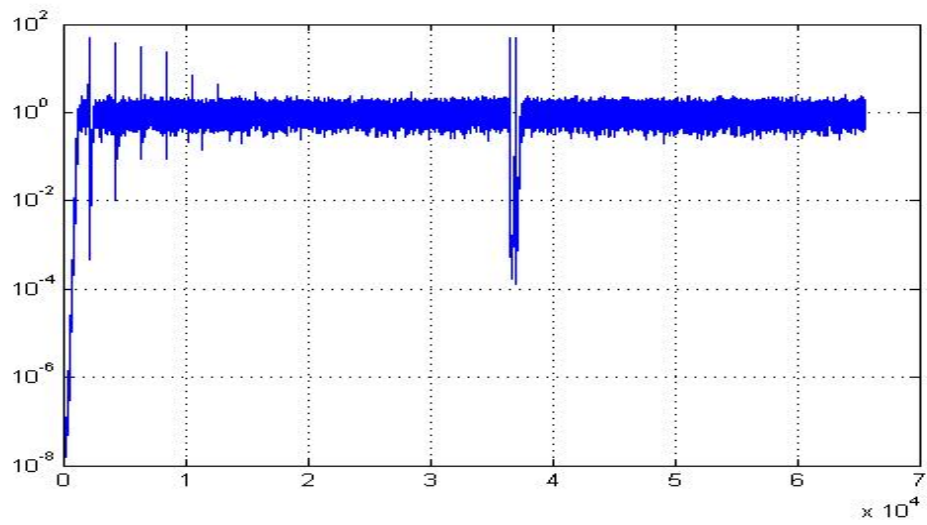
➡ **[ind,fr,snr1,amp,peaks,snr]=sp_find_fpeaks(in,tau,thr,inv)**, based on the gd_findev function idea, that is adaptive mean computation, with

- **in** input gd or array
- **tau** AR(1) tau (in samples)
- **thr** threshold
- **inv** 1 -> starting from the end, 2 -> min of both
- **ind** index of the peak (of the snr array)
- **fr** peak frequencies
- **snr1** peak snr
- **amp** peak amplitudes
- **peaks** sparse array
- **snr** the total snr array

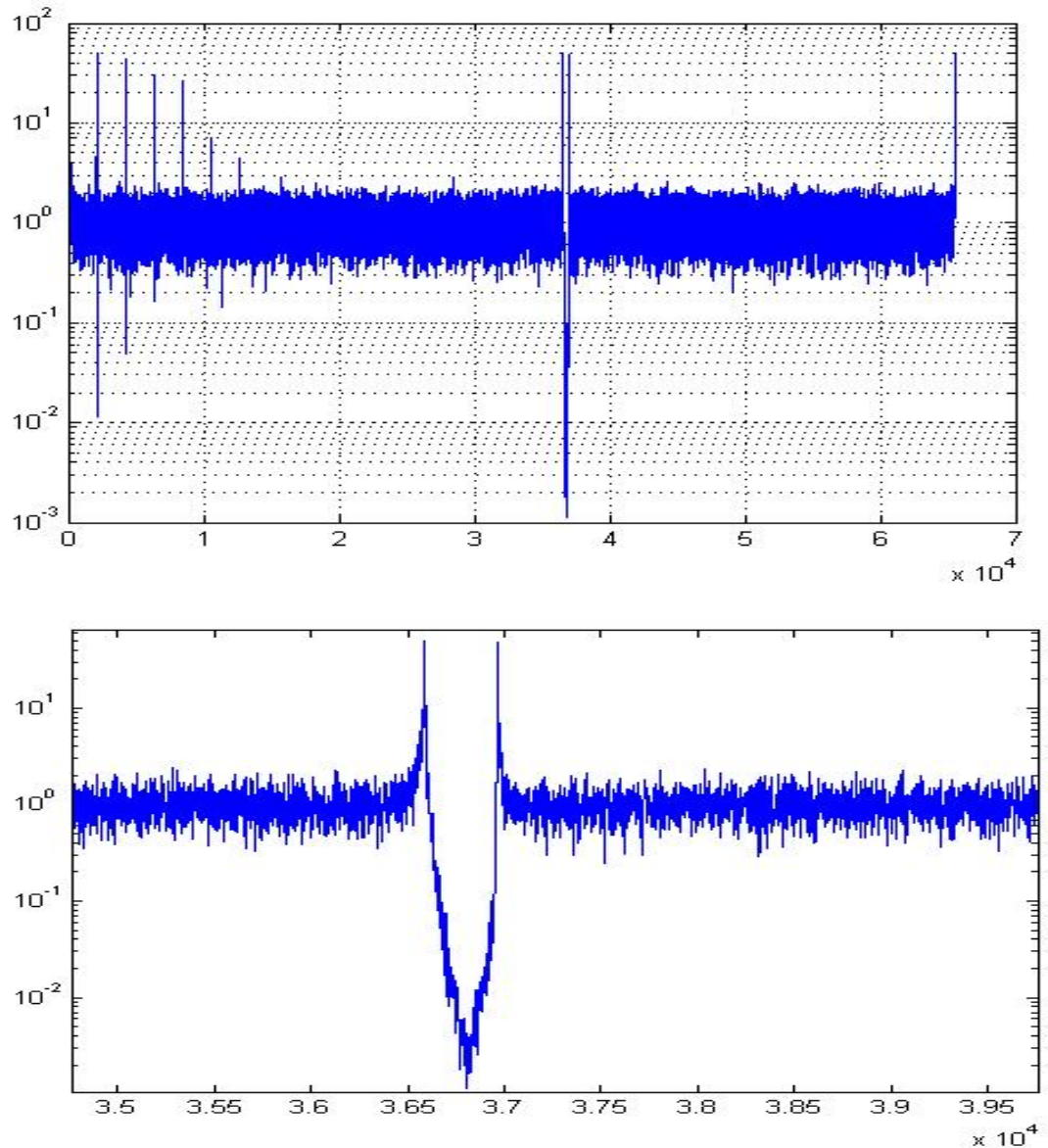
Applying this procedure to a plot like



We have, with $\text{inv}=0$,



And with $\text{inv} = 2$,

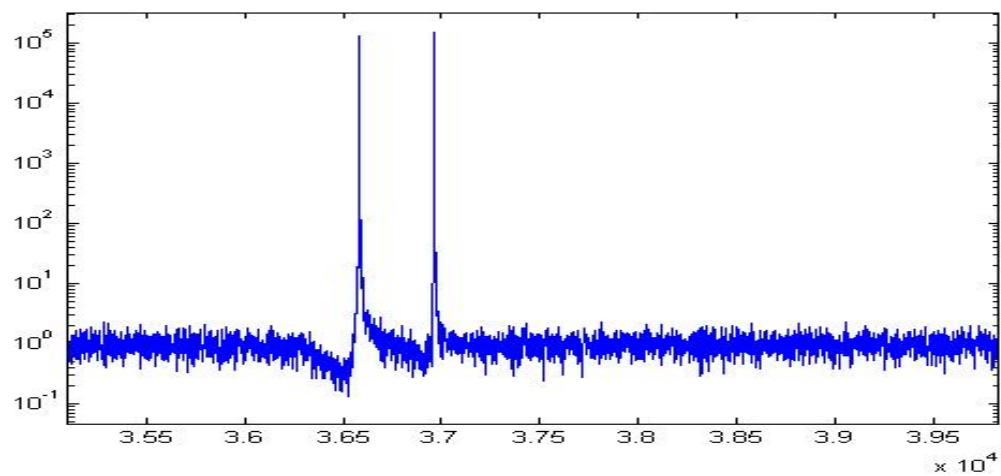
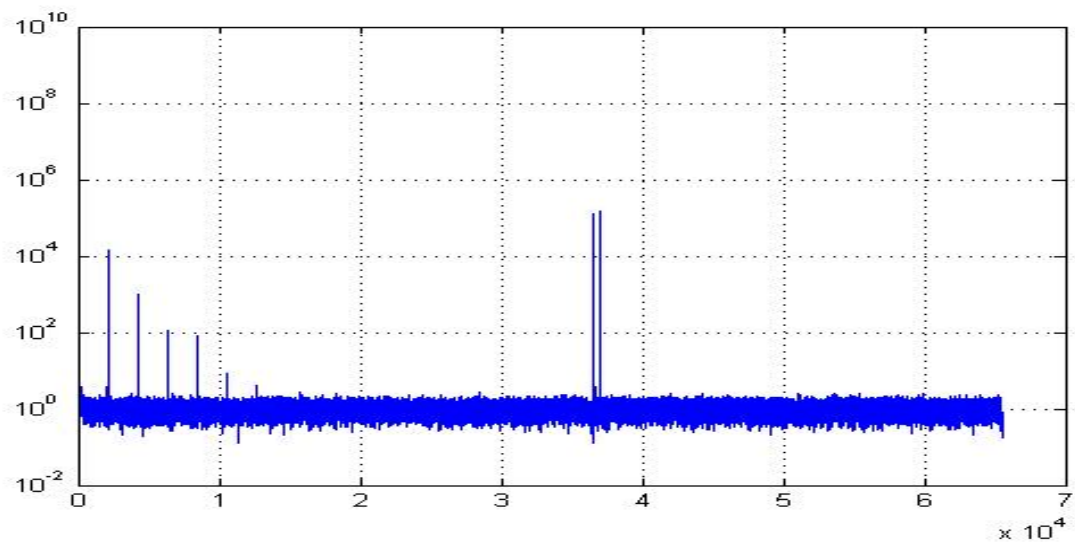


➡ **[i,fr,s,peaks,snr]=sp_find_fpeaks_nl(in,tau,maxdin,maxage,thr,inv)**, based on the **gd_findev_nl** function idea, that is adaptive mean computation non-linearly corrected, with

- **in** input gd or array
- **tau** AR(1) tau (in samples)
- **maxdin** maximum input variation to evaluate statistics
- **maxage** max age to not update the statistics (same size of tau)
- **thr** threshold
- **inv** 1 -> starting from the end, 2 -> min of both
-

- **ind** index of the peak (of the snr array)
- **fr** peak frequencies
- **snr1** peak snr
- **amp** peak amplitudes
- **peaks** sparse array
- **snr** the total snr array

Applied to the same spectrum obtains:



Hough transform

[C environment]

PSS_hough is a C code which performs the all-sky Hough transform of a given time-frequency peak map.

Theory

The Hough transform is a robust parameter estimator for patterns in digital images. It can be used to estimate the parameters of a curve which best fits a given set of points. The basic idea is that of mapping the data into the parameter space and identifying the parameter values as clusters of points.

For instance, suppose our data are distributed as a straight line: $y=m*x+q$. In this case the parameters to be determined are the slope m and the intercept q . To each point (x,y) a new straight line in the parameter space with equation $q=y-m*x$ corresponds. That is, we can draw a line in the parameter space for each pair (x,y) and all these will intersect in a point (m,q) identifying the parameters of the original line. If noise is present several “clusters” of points will appear in the parameter space.

In our case, the original data are the points in the time-frequency peak map and the transformed data are points in the source parameter space: $(\alpha, \delta, f_0, \dot{f}_0, \ddot{f}_0, \dots)$, i.e. the source position (in ecliptical coordinates), the source intrinsic frequency and the value of the spin-down parameters.

Assuming that there is no spin-down, the relation among each point in the time-frequency plane and the points in the parameter space is given by the Doppler effect equation

$$f_0 - f_k = f_0 \frac{\vec{v} \cdot \vec{n}}{c}$$

where f_k is the frequency of a peak, \vec{v} is the detector velocity vector and \vec{n} is the versor identifying the direction of the source. From this equation we find that the locus of points in the sky to which a source emitting a signal at frequency at f_0 could belong, if a peak at frequency f_k is found, is a circle with radius ϕ given by:

$$\cos \phi = \frac{f_0 - f_k}{f_0 \frac{v}{c}}$$

and centered in the direction of the detector velocity vector.

Due to the frequency discretization, in fact we have an “annulus”, delimited by two circles rather than a single circle for each peak.

Peaks belonging to the same spectrum produce concentric annuli, while moving from one spectrum to the following the center of the annuli moves on the celestial sphere around the ecliptic. Moreover, also the circles radius changes in time, because of the variation of the modulus of the detector velocity vector.

For a given value of the source reference frequency f_0 and spin-down, calculating the annuli corresponding to all the peaks in the time-frequency map and properly summing them, in order to take into account the source spin-down, we compute the Hough map.

For each value of the source reference frequency we have several Hough maps, depending on the number of possible different values of the spin-down parameters.

The number of spin-down parameters that must be taken into account depends on the

minimum source decay time, $\tau = \frac{f_0}{\dot{f}_0}$ which we search for. In order to limit the overall

needed computing power to a reasonable value, we choose $\tau \approx 10^4 \text{ yr}$ which implies that only the first spin-down parameter, \dot{f}_0 , must be considered.

The total number of Hough maps that are built is given by $N_{f_0} \cdot N_{\dot{f}_0}$, i.e. the product of the number of different frequency values and of the number of different values of the first spin-down parameter.

As explained in the introduction, due to the noisy data, billions of candidates will be selected in this ensemble of Hough maps.

Implementation

As said in the previous section, at each peak in the time-frequency peak map an annulus on the celestial sphere corresponds. The computation of the Hough transform consists, basically, in the computation of the annuli associated to all the peaks present in the time-frequency peak map and in summing them in order to take into account the spin-down. For performance reasons, in our implementation we use a set of look-up tables (LUT), each of which is, basically, a standard C array where the coordinates of the points of all the possible left semi-circles (which are annuli borders) corresponding to a given source reference frequency are stored.

In order to build a LUT we need:

- the source reference frequency f_0 ;
- the detector velocity vector \vec{v} at a given time;
- the frequency bin width df (this is fixed in each frequency band).

Then,

- make a loop on the latitude of the circle centers δ_0 (the longitude is chosen zero);
- make a loop on the frequency bins in the Doppler band around f_0 ; for each frequency bin calculate the circle radius $\cos(\varphi)$;
- calculate the minimum and maximum pixel of each circle (i.e $i(\delta_{\min}), i(\delta_{\max})$);
- make a loop on the “ordinate” (int) of each circle (i.e from $i(\delta_{\min})$ to $i(\delta_{\max})$);
- calculate the “abscissa” α (float) of the circle points (in the left semi-plane) using the equation $\cos(\alpha) = (\cos(\varphi) - \sin(\delta)\sin(\delta_0)) / \cos(\delta)\cos(\delta_0)$.

In principle, a new LUT should be built for each frequency f_0 ; however it can be shown that the same LUT holds, generally, for several frequencies (i.e. the circle radius varies much less than a pixel moving from a frequency to a near one).

For the construction of the Hough map, for a fixed source reference frequency, we pick from the corresponding LUT the semi-circles corresponding to the selected peaks in the time frequency map.

For each peak we need to take from the LUT a pair of consecutive semi-circles.

Each semi-circle is properly shifted in α in order to take into account the actual sidereal time; the right part of the circle is obtained by reflecting the left part around the line $\alpha = \alpha_0$.

The coordinates of all the points are discretized. At the end, we have 4 borders for each peak.

We sum +1 to the pixels of the external left border and of the inner right border and, on the contrary, we sum -1 to the external right border and to the inner left border. This means that annuli are represented through their ‘derivative’. The detector beam pattern and the noise non stationarity can be taken into account at this stage.

Then, for each time slice we build a Partial Hough Map Derivative (PHMD) and we sum the PHMDs in different ways corresponding to different time slices depending on the spin-down value. In this way we obtain a total Hough Map Derivative (HMD). Finally, each HMD is integrated obtaining the total Hough Map (HM). Practically, each HM is a standard C array where the number count of all the sky pixels is registered.

In the present implementation the same LUT is used for all the time slices, for a given f_0 .

As a matter of fact, when moving from a time slice to another, the modulus of the detector velocity vector changes and then changes also the radius of the circle corresponding to a given peak. As a consequence, an error is introduced. This could be taken into account for instance by interpolating the circles stored in the LUT.

These are the main steps of the program:

1. **Read the file containing the peaks and related information (da aggiornare).**

The input file contains the following general information:

- detector name;
- source reference frequency f_0 ;
- total number of peaks in the file;
- FFT duration (T_{FFT});
- number of time slices considered (nSpec);
- spin-down decay time (tau);

Then, for each time slice, the following quantities are given:

- sidereal time (tsid);
- snr (associated to tsid);
- number of peaks at that time;
- detector velocity ($|\vec{v}|, v_\alpha, v_\delta$);
- list of peaks.

2. **Inizialization of the LUT and of the Hough map**

This is done using some of the information read in the input file.

3. **Construction of the LUT**

The LUT is built for a given source reference frequency f_0 and a given time and then it is used for all the times and for a range of frequencies. If needed, i.e. if the

frequency goes out of the its range of validity, it is re-calculated during the computation of the Hough transform.

4. Construction and sum of the PHMDs

This is done for each initial reference frequency f_0 . The sums of the PHMDs are done according to the different possible values of the spin-down parameter and each corresponds to a given “slope” in the time-frequency plane. Each sum produces a total Hough map derivative (HMD).

5. Integration of the HMD

This operation is done once for each reference frequency and produces the total Hough map (HM).

6. Selection of candidates

Once an HM has been built, candidates are selected and used in the next steps of the analysis.

The PSS_hough code is made of the following files:

pss_hough.c: main file

readParameters.c: contains the function *long readParameters(int, char *)*

readPeaks.c: contains the function *int * readPeaks(char *, float *, float *, int *, float *, float *, float *, char *)*

flush_cache.c: contains the function *void flush_cache()*; only for test purpose – not used in production

lutInitialize.c: contains the function *void lutInitialize()*

lutBuild.c: contains the function *void lutBuild(float, float *)*

drawCircles.c: contains the function *void DrawLeftCircle(float, float, float, float, float, int, int, float *)*

houghRad.c: contains the function *void houghRad(char *)*

radpat.c: contains the functions *float * radpat_interf(struct Antenna *)*
*float * radpat_interf_eclip(struct Antenna *)*
*float * radpat_bar(struct Antenna *)*
*float * radpat_bar_eclip(struct Antenna *)*

houghBuild.c: contains the function *void houghBuild(FILE *, float *, float *, float *, float *, int *, int *, float *)*

houghInitialize.c: contains the function *void houghInitialize()*

phmdDrawCircle.c: contains the functions

*void DrawCircleNext(float, float, float, int, int, int, int, float *, float *)*
*void DrawCircleNextOpp(float, float, float, int, int, int, int, float *, float *)*
*void DrawCircleNextInit(float, float, float, int, int, float *)*
*void DrawCircleNextFin(float, float, float, int, int, float *)*
*void DrawCircleNextNoBeam(float, int, int, int, int, float *, float *)*
*void DrawCircleNextOppNoBeam(float, int, int, int, int, float *, float *)*
*void DrawCircleNextInitNoBeam(float, int, int, float *)*
*void DrawCircleNextFinNoBeam(float, int, int, float *)*

phmdtophm.c: contains the function *void phmdtophm()*

candidates.c: contains the function *void candidates(FILE *, float, float)*

lutData.c: contains external variables definition

cycle_counter.c: contains function *unsigned long long realcc(void)*; only for test purpose – not used in production

lutParameters.h: contains parameters definition

lut.h: contains global variables declaration

The program is compiled through the following Makefile:

```
CC2 = /usr/pgi/linux86/bin/pgcc
CC = gcc

#regular compile
CFLAGS = -O3 -ffast-math -funroll-loops -fexpensive-optimizations
CFLAGS2 = -O2 -fastsse -Mcache_align -Minfo

# -----

OBJS = lutData.o readParameters.o lutInitialize.o\
      houghInitialize.o lutBuild.o readPeaks.o houghBuild.o \
      drawCircles.o phmdDrawCircle.o \
      cycle_counter.o flush_cache.o phmdtophm.o \
      houghRad.o radpat.c candidates.o

LUTH = lut.h lutParameters.h
RADH = radpat.h
# -----
# default target

all: pss_hough

# -----
# produce object files

cycle_counter.o: cycle_counter.c cycle_counter.h
    $(CC) -c $(CFLAGS) cycle_counter.c

lutData.o: lutData.c $(LUTH)
    $(CC) -c $(CFLAGS) lutData.c

readParameters.o: readParameters.c $(LUTH)
    $(CC) -c $(CFLAGS) readParameters.c

lutInitialize.o: lutInitialize.c $(LUTH)
    $(CC) -c $(CFLAGS) lutInitialize.c

houghInitialize.o: houghInitialize.c $(LUTH)
    $(CC) -c $(CFLAGS) houghInitialize.c

lutBuild.o: lutBuild.c $(LUTH)
    $(CC) -c $(CFLAGS) lutBuild.c

readPeaks.o: readPeaks.c $(LUTH)
    $(CC) -c $(CFLAGS) readPeaks.c

houghBuild.o: houghBuild.c $(LUTH)
    $(CC) -c $(CFLAGS) houghBuild.c
```

```

drawCircles.o: drawCircles.c $(LUTH)
    $(CC) -c $(CFLAGS) drawCircles.c

#
phmdDrawCircle.o: phmdDrawCircle.c $(LUTH)
    $(CC2) -c $(CFLAGS2) phmdDrawCircle.c

phmdtophm.o: phmdtophm.c $(LUTH)
    $(CC) -c $(CFLAGS) phmdtophm.c

houghRad.o: houghRad.c $(RADH)
    $(CC) -c $(CFLAGS) houghRad.c

radpat.o: radpat.c $(RADH)
    $(CC) -c $(CFLAGS) radpat.c

candidates.o: candidates.c $(LUTH)
    $(CC) -c $(CFLAGS) candidates.c

# -----

# link test code

pss_hough: pss_hough.c cycle_counter.h $(LUTH) $(OBJJS)
    $(CC) $(CFLAGS) -DANYCC=realcc pss_hough.c $(OBJJS) -o $@ -lm

# -----

# cleaning...

clean:
    rm -f pss_hough

cleanobjs:
    rm -f $(OBJJS)

cleanall: cleanobjs clean

#-----end makefile-----

```

Note that the pgcc compiler is used for the file phmdDrawCircle, because this gives a much more performing code.

The program is launched with `./pss_hough <inputfile>`.
 If no <inputfile> is specified, the default one (peakmap.in) is used.

Use of the library

The library can be divided, from a logical point of view, into two parts.

- 1) Building of the Look Up Table (LUT) for a given reference frequency.
Given a source search frequency, the circles corresponding to all the possible peaks in the Doppler band of the chosen frequency are computed. The ecliptic coordinate system is used. As a consequence, the circles have centers in a narrow belt around the ecliptic. This belt is discretized so that circle center ordinates are taken in discrete set. Also the sky is discretized in a number of pixels, then two circles with the same center are distinguishable only if their radii differ by at least one pixel.

For each circle center and for each radius, there is a loop on the ordinate values and, correspondingly, the values of the abscissa (as real values) for the left semi-circle are computed. This is done for performance reasons, since it is then very simple to determine the corresponding right semi-circle using symmetry arguments.

The computed abscissae are stored into an array which is the look-up table. Another array is created, containing the index (along the vertical direction) of the pixels corresponding to the minimum and the maximum of each circle and cumulative difference between them.

- 2) Calculation of the Hough Map (HM).
The HM is an histogram in the sky coordinates. A list of peaks is read from time-frequency peak map. For each peak, two semi-circles are read from the LUT: one corresponding to the current peak and one corresponding the peak immediately before (in the LUT). This is done because each peak, in a discretized space, produces an annulus of pixels, delimited by a pair of circles. The center of each semicircle is properly shifted, depending on the time index, and the corresponding right half is computed; then for each peak we have four semicircles. Semicircles with radius less than 90 degrees and circles with radius larger than 90 degrees are treated separately. To the pixels of each semicircle are assigned values +1 and -1 respectively, or a real value, properly determined, if the adaptive Hough map is computed.

These procedure is applied for each peak of the peak map and for each time the reference frequency is properly shifted in order to take into account the source spin-down.

At the end we have a Hough map derivative (HMD) which is then integrated to produce the final Hough map. For each assumed value of the source spin-down and for each source reference frequency, a HM is obtained. In each HM candidates, i.e. pixels where the number count is above a given threshold, are then selected.

Function prototypes

```
void lutInitialize(float);
void houghInitialize(houghD_t *, hough_t *, hough_t *);
long readParameters(int argc, char *argv[], char *, char *, float
*);
void readInfoPeakmap(char *, int *, int *, float *, float *, int
*, float *, int *);
void lutBuild(float, float *);
int * readPeaks(float, float, float, float, float *, float *, int
*, float *, float *, float *, char *, int *, int *);
void houghBuild(FILE *, float, float, int, int, float *, char *,
houghD_t *, hough_t *,
hough_t *);
void DrawLeftCircle(float, float, float, float, float, int, int,
float *);
void DrawCircleNextAdap(float, float, float, int, int, int, int,
float *, float *, houghD_t *);
void DrawCircleNextOppAdap(float, float, float, int, int, int,
int, float *, float *, houghD_t *);
void DrawCircleInitAdap(float, float, float, int, int, float *,
houghD_t *);
void DrawCircleFinAdap(float, float, float, int, int, float *,
houghD_t *);
void DrawCircleNext(float, int, int, int, int, float *, float *,
houghD_t *);
void DrawCircleNextOpp(float, int, int, int, int, float *, float
*, houghD_t *);
void DrawCircleInit(float, int, int, float *, houghD_t *);
void DrawCircleFin(float, int, int, float *, houghD_t *);
void houghRad(char *);
float *radpat_interf(struct Antenna *);
float *radpat_interf_eclip(struct Antenna *);
float *radpat_bar(struct Antenna *);
float *radpat_bar_eclip(struct Antenna *);
void hmd2hm(houghD_t *, hough_t *, hough_t *);
void writeMaps(houghD_t *, hough_t *, hough_t *);
void candidates(FILE *, float, float, float, hough_t *);
```

In green is the function through which the user can interact with the library.

Program flow from the user point of view

From the user point of view, the library takes a time-frequency peak map as input (binary file) and produces a set of candidates at the output, each defined by a reference frequency, a position in the sky and a spin-down value. The number of frequencies to be explored is automatically determined on the base of the input peak map and the number of spin-down values is computed on the basis of the minimum spin-down decay time (which can be assigned by the

user). If the adaptive Hough transform must be computed, the user can choose a detector (default: Virgo); this is needed to calculate the detector response which is a function of the detector geometry and its position and orientation on the Earth. At the moment, only the Virgo and Nautilus detectors are supported. The library is thought as part of the PSS data analysis program; this means that a user typically will run the entire data analysis procedure and only occasionally will run the PSS_hough library as a standalone program. As a consequence, the possibilities of interaction of the user with the library are rather limited. In the next section, the parameters that a user can assign are listed.

User assigned parameters

The user can assign three parameters:

1. the detector name (option: d; default value: virgo)
2. the name of the input file (eventually full path) containing the peak map (option: f; default value: pm.dat)
3. the minimum spin-down decay time, in seconds (option: s; default value: 10^4 years)

To run the library (as a standalone program):

```
./pss_hough d <detector name> f <input file name> s <minimum decay time>
```

Performance issue

Performance is an important issue for the PSS_hough library. The computation of the Hough transform is the heaviest part of the data analysis procedure and it is then important to have a software as much efficient as possible. The present implementation of the Hough transform computation is the fastest we have developed up to now. Possibly, faster versions will be released in the future.

In the next subsections results of the timing and profiling of the library are given and discussed.

Results of gprof

Machine: grwavcp.roma1.infn.it
Processor: Intel Pentium 4
L2 cache: 1MB
RAM: 1GB

The code has been compiled with options:

```
-pg -O3 -ffast-math -funroll-loops -fexpensive-optimizations
```

Program parameters:
Search frequency: 330Hz
Number of spectra: 4911 (corresponding to 6 months of data)
RAD=0 (non-adaptive Hough map)

The following profile has been obtained with the unix/linux tool *gprof*.

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
48.02	5.33	5.33	103601	0.00	0.00	DrawCircleNext
45.95	10.43	5.10	106354	0.00	0.00	DrawCircleNextOpp
3.42	10.81	0.38	6270	0.00	0.00	DrawLeftCircle
1.71	11.00	0.19	1	0.19	0.19	writeMaps
0.36	11.04	0.04	1	0.04	10.53	houghBuild
0.27	11.07	0.03	1	0.03	0.03	candidates
0.09	11.08	0.01	799	0.00	0.00	DrawCircleInit
0.09	11.09	0.01	1	0.01	0.01	hmd2hm
0.09	11.10	0.01	1	0.01	0.01	readPeaks
0.00	11.10	0.00	169	0.00	0.00	DrawCircleFin
0.00	11.10	0.00	1	0.00	0.00	gridInitialize
0.00	11.10	0.00	1	0.00	0.00	houghInitialize
0.00	11.10	0.00	1	0.00	0.38	lutBuild
0.00	11.10	0.00	1	0.00	0.00	readInfoPeakmap
0.00	11.10	0.00	1	0.00	0.00	readParameters

%
time the percentage of the total running time of the
program used by this function.

cumulative a running sum of the number of seconds accounted
seconds for by this function and those listed above it.

self
seconds the number of seconds accounted for by this
function alone. This is the major sort for this
listing.

calls the number of times this function was invoked, if
this function is profiled, else blank.

self
ms/call the average number of milliseconds spent in this
function per call, if this function is profiled,
else blank.

total
ms/call the average number of milliseconds spent in this
function and its descendents per call, if this
function is profiled, else blank.

name the name of the function. This is the minor sort
for this listing. The index shows the location of
the function in the gprof listing. If the index is
in parenthesis it shows where it would appear in
the gprof listing if it were to be printed.

Call graph (explanation follows)

granularity: each sample hit covers 4 byte(s) for 0.09% of 11.10 seconds

index	% time	self	children	called	name
					<spontaneous>
[1]	100.0	0.00	11.10		main [1]
		0.04	10.49	1/1	houghBuild [2]
		0.00	0.38	1/1	lutBuild [6]
		0.19	0.00	1/1	writeMaps [7]
		0.00	0.00	1/1	readParameters [16]
		0.00	0.00	1/1	readInfoPeakmap [15]
		0.00	0.00	1/1	gridInitialize [13]

		0.04	10.49	1/1	main [1]
[2]	94.9	0.04	10.49	1	houghBuild [2]
		5.33	0.00	103601/103601	DrawCircleNext [3]
		5.10	0.00	106354/106354	DrawCircleNextOpp [4]
		0.03	0.00	1/1	candidates [8]
		0.01	0.00	799/799	DrawCircleInit [9]
		0.01	0.00	1/1	readPeaks [11]
		0.01	0.00	1/1	hmd2hm [10]
		0.00	0.00	169/169	DrawCircleFin [12]
		0.00	0.00	1/1	houghInitialize [14]

		5.33	0.00	103601/103601	houghBuild [2]
[3]	48.0	5.33	0.00	103601	DrawCircleNext [3]

		5.10	0.00	106354/106354	houghBuild [2]
[4]	45.9	5.10	0.00	106354	DrawCircleNextOpp [4]

		0.38	0.00	6270/6270	lutBuild [6]
[5]	3.4	0.38	0.00	6270	DrawLeftCircle [5]

		0.00	0.38	1/1	main [1]
[6]	3.4	0.00	0.38	1	lutBuild [6]
		0.38	0.00	6270/6270	DrawLeftCircle [5]

		0.19	0.00	1/1	main [1]
[7]	1.7	0.19	0.00	1	writeMaps [7]

		0.03	0.00	1/1	houghBuild [2]
[8]	0.3	0.03	0.00	1	candidates [8]

		0.01	0.00	799/799	houghBuild [2]
[9]	0.1	0.01	0.00	799	DrawCircleInit [9]

		0.01	0.00	1/1	houghBuild [2]
[10]	0.1	0.01	0.00	1	hmd2hm [10]

		0.01	0.00	1/1	houghBuild [2]
[11]	0.1	0.01	0.00	1	readPeaks [11]

		0.00	0.00	169/169	houghBuild [2]
[12]	0.0	0.00	0.00	169	DrawCircleFin [12]

		0.00	0.00	1/1	main [1]
[13]	0.0	0.00	0.00	1	gridInitialize [13]

		0.00	0.00	1/1	houghBuild [2]
[14]	0.0	0.00	0.00	1	houghInitialize [14]

		0.00	0.00	1/1	main [1]
[15]	0.0	0.00	0.00	1	readInfoPeakmap [15]

		0.00	0.00	1/1	main [1]
[16]	0.0	0.00	0.00	1	readParameters [16]

This table describes the call tree of the program, and was sorted by the total amount of time spent in each function and its children.

Each entry in this table consists of several lines. The line with the index number at the left hand margin lists the current function. The lines above it list the functions that called this function, and the lines below it list the functions this one called.

This line lists:

- index A unique number given to each element of the table.
Index numbers are sorted numerically.
The index number is printed next to every function name so it is easier to look up where the function in the table.
- % time This is the percentage of the 'total' time that was spent in this function and its children. Note that due to different viewpoints, functions excluded by options, etc, these numbers will NOT add up to 100%.
- self This is the total amount of time spent in this function.
- children This is the total amount of time propagated into this function by its children.
- called This is the number of times the function was called.
If the function called itself recursively, the number only includes non-recursive calls, and is followed by a '+' and the number of recursive calls.
- name The name of the current function. The index number is printed after it. If the function is a member of a cycle, the cycle number is printed between the function's name and the index number.

For the function's parents, the fields have the following meanings:

- self This is the amount of time that was propagated directly from the function into this parent.
- children This is the amount of time that was propagated from the function's children into this parent.
- called This is the number of times this parent called the function '/' the total number of times the function

was called. Recursive calls to the function are not included in the number after the `/'.

name This is the name of the parent. The parent's index number is printed after it. If the parent is a member of a cycle, the cycle number is printed between the name and the index number.

If the parents of the function cannot be determined, the word `<spontaneous>' is printed in the `name' field, and all the other fields are blank.

For the function's children, the fields have the following meanings:

self This is the amount of time that was propagated directly from the child into the function.

children This is the amount of time that was propagated from the child's children to the function.

called This is the number of times the function called this child `/' the total number of times the child was called. Recursive calls by the child are not listed in the number after the `/'.

name This is the name of the child. The child's index number is printed after it. If the child is a member of a cycle, the cycle number is printed between the name and the index number.

If there are any cycles (circles) in the call graph, there is an entry for the cycle-as-a-whole. This entry shows who called the cycle (as parents) and the members of the cycle (as children.) The `+' recursive calls entry shows the number of function calls that were internal to the cycle, and the calls entry for each member shows, for that member, how many times it was called from other members of the cycle.

Index by function name

[12] DrawCircleFin	[8] candidates	[6] lutBuild
[9] DrawCircleInit	[13] gridInitialize	[15]
readInfoPeakmap		
[3] DrawCircleNext	[10] hmd2hm	[16]
readParameters		
[4] DrawCircleNextOpp	[2] houghBuild	[11]
readPeaks		
[5] DrawLeftCircle	[14] houghInitialize	[7]
writeMaps		

Comments

It is clear from the profiling that most of the time is spent in the functions DrawCircleNext and DrawCircleNextOpp. This is due to two facts:

1. these functions are called a number of times equal to the number of circles which must be 'drawn'. The time spent in each call is about 50 microseconds.
2. In these functions a large array, the Hough map derivative, is accessed in a random way and this results in a large number of cache misses. If we could access it in a more regular way, a gain in performances of at least a factor of 2 could be obtained. We will try to reduce this bottleneck in the next releases of the library.

Compiling the library with the *pgcc* compiler, performances improve by about 20%.

Similar results are obtained taking a search frequency in the highest frequency band (500Hz-2kHz).

For lower bands (<31.25Hz; 31.25Hz-125Hz), the weight of the functions DrawCircleNext and DrawCircleNextOpp gradually decreases because the dimension of the Hough map derivatives becomes smaller and, consequently, decreases the number of cache misses.

A useful way to compare the code performances on different platforms is to compute the "equivalent number of clock cycles" needed to increase by 1 the number count in a given pixel of a Hough map.

In the following table some results are reported for different values of the search frequency *f*. The other parameters are the same as those given at the beginning of the previous section.

	Pentium 4	Pentium 4	Xeon	Opteron 64 bit
CPU (GHz)	2.8	2.8	2.4	2.2
L2 cache (MB)	1	1	.512	1
compiler	gcc 3.2	pgcc 6.1	gcc 3.2	gcc 3.2
f=20Hz	40	34	41	29
f=80Hz	43	34	64	32

f=320Hz	69	57	79	70
---------	----	----	----	----

[MatLab environment]

pss_explorer

pss_hough

Supervisor

Basics

We call **Supervisor** (SV) the ensemble of services and programs we are developing for farm/job management. It is built on top of a batch system which controls job scheduling and load balancing (PBS, see later for a more detailed description). The batch system can be customized according to the user's needs; the SV in some cases uses the PBS API functions. Let us see how the SV enters in the general scheme for the hierarchical procedure developed for the periodic sources search.

Schematically, these are the main steps in the data analysis procedure:

1. **build SFDB**: construction of the SFDB from the h-reconstructed data;
2. **peak selection**: selection of peaks in the periodograms obtained from the SFDB;
3. **incoherent step**: the peaks are the input of the HT;
4. **candidate selection**: candidates in the output of the HT are selected;
5. **coherent step**: initial data are corrected, longer FFTs are calculated and a new peak-map is produced.

Steps 2-5 are repeated until the length of FFTs is equal to the total observation time.

The SV enters in:

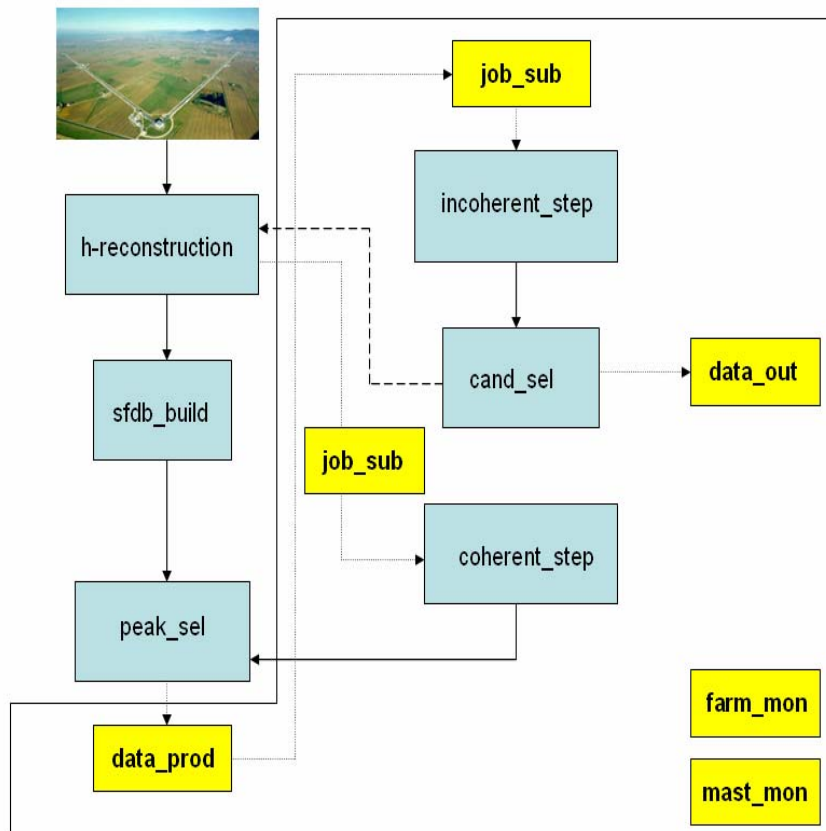
peak selection: after peak selection, a service running on the master node, **data_prod**, will produce input files for the HT; each input file will consist of lists of peaks corresponding to a given frequency band (plus the components of the detector velocity vector).

incoherent step, coherent step: in these phases a service, **job_sub**, is dedicated to job submission and workload management on the farm nodes. Note, concerning the coherent part of the analysis, that at least the first coherent step will be performed on several processors because the computing power needed, though small with respect to what we need for the first incoherent step, is not negligible.

candidate selection: after candidate selection, a service running on the master node, **data_out**, copies output files (two kinds of files, one containing candidates and the other containing information on the jobs) on the storage.

steps (2,5): a service, running on the master is dedicated to the monitoring of farm nodes and of PSS-jobs; a service, running on “secondary masters”, is dedicated to the monitoring of the master node.

In the following picture the relation between the main steps in the data analysis procedure and the Supervisor services is outlined. In particular, monitoring services, i.e. **farm_mon** and **mast_mon**, are not tied to a specific point of the analysis chain but act on the whole ‘space’ delimited by the ‘box’.



Outline of the supervisor

We call **master** the node where the active SV is running; it is the machine from where most of the management is done; we call **slave**, or **Computing Node (CN)**, a machine where calculations are done; the master will be also a CN. We call **Storage Element (SE)** the machine where data (both input and output) are stored.

We assume the cluster is on a private network, with the masters (both “primary” and “secondary”) as the only interface to the LAN. Secondary masters are CN which can replace the master if this fails. Moreover, some of the CN can be also part of a grid farm.

We distinguish two types of jobs: *PSS_jobs*, which are directly managed by the SV, and *N_jobs* which run on the CN. In general, we can think a PSS-job as made of several N-job, plus some information.

Now, we list the main steps in the SV activity:

1. **supervisor start:** SV starts on the master;
2. **master monitoring start:** SV starts the master monitoring service on “secondary” masters; “secondary” masters are ordered according to the rank (active master has rank 0, the second one has rank 1 and so on)
3. **node status monitoring:** SV periodically checks the status of all CN and produce a report;
4. **master status monitoring:** on each “secondary” master a service periodically checks the status of all the masters of greater rank and, if no active master exists, the node where it is running becomes the new primary master and start farm management;
5. **input data production:** The active SV produces input data on the master;
6. **job submission:** The active SV starts the *PSS-job*: creates a PSS-job folder on the SE and manages the workload of the CNs;
7. **job status monitoring:** The status of a *PSS-job* is periodically monitored (querying the node(s) where it is running), saved on a file and distributed among all the secondary masters. This is needed if the active master crushes and must be replaced by another one. The status of a *PSS-job* contains the information listed in **ref A**.
8. **output file copy:** The final results of each *N-job* are stored in the PSS-job folder on the SE; a copy is kept also on the node where the job run.

ref A: the status of a PSS-job contain the following information:

- the name of the submitting machine (the master);
- the submission time;
- on which CNs (one or more) and which of these, if any, have the (sleeping) SV;
- the workload distribution (i.e. the *N_jobs*: name, submission time, input data,...);
- all the important events (end time of a *N-job*, and the exit status, crashes, problems,...).

All these information are collected in a file and also in a C structure.

Implementation of the Supervisor

The steps outlined in the last paragraph can be translated in a number of services which “form” the SV program and which we have already introduced in the previous section:

- a. `data_prod`: Production of input data files starting from the SFDB;
- b. `job_sub`: Job submission, workload management and retrieval of output files;
- c. `data_out`: Management of output files;
- d. `farm_mon`: Monitoring of PSS-jobs and slave nodes;
- e. `mast_mon`: Monitoring of master node.

The mapping between the SV steps and services is the following:

SV activity	Service
node status monitoring	<code>farm_mon</code>
master status monitoring	<code>mast_mon</code>
input data production	<code>data_prod</code>
job submission	<code>job_sub</code>
job status monitoring	<code>farm_mon</code>
output file copy	<code>data_out</code>

In the following figure a schematic view of the SV services and their relation with the cluster environment is given.

In the implementation of the services we sometimes rely on the PBS batch system. This has been conceived for job distribution and workload management and gives to the user a set of commands also for job/node/queues monitoring. In particular, we use the *Batch Interface Library* (IFL) which provides a set of user callable functions with, approximately, a one to one correlation with client commands.

Candidate database and coincidences

The database

Periodic Source Candidates are stored in particular huge data bases, named **PSC_DB**. In this case only one spin-down parameter is considered.

A PSC_DB is a collection of files and folders, contained in a folder with name PSC_DBxxxxxxx .

This folder contains

- a **readme.txt** file,
- a data-base creation log file **psc.log**,
- a doc file **psc.doc** with the documentation,
- a **psc.dat** file that is a script with peculiarities of that DB, like the starting time, the sampling time, the length of the ffts, the number of spin-down parameters, the antenna coordinates,...
- 20 folders named **0000, 0100, 0200, ..., 1900**

each of these folders contain 10 folders named **00, 10, 20, ..., 90** and each of these last contain 10 files, one for each hertz of starting frequency. The name of the files refer to the name of the PSC_DB and to the covered frequency range, for example, **pss_cand_1394.cand** or **pss_cand_0101.cand** .

Each file has the following structure:

Header		
protocol (e.g. 1 now)	int4	1-4
caption	char128	5-132
initial time (mjd)	double	133-140
sampling time	double	141-148
FFT length	int8	149-156
initial frequency of the file (fft bins – first bin 0)	int8	157-164
delta lambda	float	165-168
delta beta	float	169-172
delta sd1	float	173-176
delta CR	float	177-180

delta mean Hough map (mh)	float	181-184
delta h	float	185-188
Any candidate		
frequency bin (from initial basic group)	int2	1-2
lambda index	int2	3-4
beta index	int2	5-6
sd1 index	int2	7-8
CR index	int2	9-10
mh index	int2	11-12
h index	int2	13-14

If the data-base contain 10^9 candidates, each file should contain about 500000 candidates.
So the mean value for the dimension of the file is

$$500000 \times 12 + 44 \sim \mathbf{6\ MB}$$

and the total dimension of the data base should be about **12 GB**.

To perform the coincidence analysis, the files are supposed to be

Browsing the PSC database

Searching for coincidences in the PSC database

Coherent follow-up

Theory and simulation

Snag pss gw project

PSS detection theory

Sampled data simulation

The basic periodic simulation function is

➡ **sim_sds(sim_str,fdb_str,doptab)** , that creates one or more sds data files. It is based on two structures:

- **sim_str** , a simulation structure with elements

- type = 0 stationary, 1 non-stationary
- nss non-stationarity structure (if type=1)
- ant antenna structure (only for for signal simulation)
- sour source structure (only for for signal simulation)
- lfft fft length for simulation
- t0 initial time (mjd)
- dt sampling time (s)

- **fdb_str** , a file database structure, with elements

- folder database folder
- head filename header (p.es. 'VIR_hrec_')
- tail filename tail (p.es. '_crab')
- ndat total number of data
- fndat number of data per file

- **doptab** , the Doppler table.

➡ **fileout=realsim_sds(sds_in,chn,sim_str,fdb_str,doptab)** , that adds simulated periodic source signal to a real data file. The parameters are:

- **sds_in** , the input sds file to be used
- **chn** , the channel number
- **sim_str** , a simulation structure with elements

- `ant` antenna structure (only for for signal simulation)
 - `sour` source structure (only for for signal simulation)
 - `lchunk` chunk length for simulation
- **`fdb_str`** , a file database structure, with elements
 - `folder` database folder
 - **`doptab`** , the Doppler table.

These functions, to simulate the periodic source signals, use

- ➡ **`[d,source,data]=ps_chunk(source,antenna,data,doptab)`** , where `source`, `antenna` and `data` are the pss structures and `doptab` is the correct Doppler table. The data are simulated in chunks during which the frequency and amplitude is standard (but the chunks can also last a single sample; it is reasonable that last at least one or more seconds). The produced chunks can be added to real data or data simulated with `sim_sds` (or other).

Time-frequency map simulation

The functions to simulate time-frequency spectra are obtained by the same functions that simulate the peak maps, with the input keyword **tfspec** set to 1.

Peak map simulation

There are two simulation functions, one, easier, that can be used for the first incoherent step, when the data chunks have the length of no more than some hours, and another for the subsequent analysis, when the data chunks can be of more than one day.

There are some function to create files to store peak map data:

- ➡ **pss_w_pm_0(pm,file)** , that stores the peak map structure **pm** in the file **file**, in a very easy format.

Low resolution simulation

- ➡ **pm=pss_sim_pm_lr(ant,sour,pmstr,doptab,nois,level,tfspec)** , where:

- **ant** is an antenna structure, with elements:
 - **lat** latitude (in degrees)
 - **long** longitude (in degrees)
 - **azim** azimuth
 - **type** antenna type (1 -> bar, 2 -> interferometer)
- **sour** source structure array; elements used:
 - **a** right ascension (degrees)
 - **d** declination (degrees)
 - **eps** fraction of linear polarization power
 - **psi** angle of linear polarization
 - **f0** un-Dopplered frequency at start time
 - **df0** coefficient of first power spin-down (Hz/day)
 - **ddf0** coefficient of second power spin-down (Hz/day²)
 - **snr** signal-to-noise ratio (linear; supposing white noise)
- **pmstr** peak map property structure
 - **dt** sampling time (s)
 - **lfft** fft length
 - **frin** initial frequency
 - **nfr** number of frequency bins
 - **res** spectral resolution (in normalized units)
 - **t0** initial time (mjd)
 - **np** number of periodograms
 - **thresh** threshold (typically 2)
 - **win** window type (0 -> no, 1 -> pss);

- **doptab** Doppler table (created by ...)
 - table containing the Doppler data (depends on antenna and year) (a matrix (n*4) or (n*5) with:
 - **first column** containing the times (normally every 10 min)
 - **second col** x (in c units, in equatorial cartesian frame)
 - **third col** y
 - **fourth col** z
- **nois** a pmstr.np array with noise levels, in standard deviations; if the dimension is not exact, the value 1 is given for all the periodograms
- **level** simulation level:
 - 1** no sid modulation, direct frequency computation, res=1
 - 2** sid modulation, direct frequency computation, res=1
 - 3** no sid modulation, fft frequency computation
 - 4** sid modulation, fft frequency computation
- **tfspec** if = 1, time-frequency spectrum, otherwise only peak map; default 0

Output peak map structure:

- **pm** output peak map structure; elements:
 - **np** number of periodograms
 - **frin** initial frequency
 - **nfr** number of frequency bins
 - **dt** sampling time (s)
 - **lfft** fft length
 - **dfr** frequency bin width
 - **res** spectral resolution (in normalized units)
 - **t0** initial time (mjd)
 - **thresh** threshold (typically 2)
 - **win** window type (0 -> no, 1 -> pss);
 - **t(:)** periodograms time
 - **v(:,3)** periodograms detector velocity
 - **PM** peak map (sparse matrix) or t-f spectrum (matrix)

High resolution simulation

Candidate simulation

[MatLab environment]

The basic candidate simulation function is

- ➡ **crea_pssfakecand**, that creates a candidate database. The folder structure should exist. It can be copied from the template that exists in the metadata sub-directory. The input data are:
 - **dircand**, the candidate root directory, containing the folder structure (containing up to 2000 files in 200 folders, grouped in 20 parent folders)
 - **N**, the number of candidates to create
 - **band**, wich band (1,2,3 or 4)
 - **pss_cand_head**, pss candidate file header structure and simulation parameters

This function (no return values) can be launched by the m file **batch_fakecreation** (to be edited), that creates two pss candidate databases.

The creation of a pss database of 10^8 candidates takes about 7 minutes on a 3GHz two-cpu computer.

Time and astronomical functions

[MatLab environment]

Time

In Snag there is a number of time conversion functions.

The basic time used in Snag is MJD (Modified Julian Date). Other time used are TAI and GPS time; the form of time (normally UTC time) is also string or vector.

- ➡ **t=s2mjd(str)** and **str=mjd2s(mjd)** (**str=mjds2s(mjd)** for multiple operations), converts string time to mjd (modified julian date) and viceversa; example: **mjd=s2mjd('25-Apr-2004 18:44:11')** produces **mjd = 53120.7806828704**, and **str=mjd2s(mjd)** produces **25-Apr-2004 18:44:11.000004**.
- ➡ **str=mjd2s(mjd)**, converts a modified julian date to string time; example:
- ➡ **v=mjd2v(mjd)** and **t=v2mjd(v)**, converts a modified julian date to vectorial time ([year month day hour minute second]) and viceversa.
- ➡ **tgps=mjd2gps(mjd)** and **mjd=gps2mjd(tgps)**, converts mjd to gps time and viceversa.
- ➡ **tai=mjd2tai(mjd)** and **mjd=tai2mjd(tai)**, converts mjd to tai and viceversa.
- ➡ **tdt=tai2tdt(tai)**, conversion from TAI to Terrestrial Dynamical Time
- ➡ **tsid=sid_tim(mjd,long)**, sidereal time (in hours);
 - **mjd** modified julian date (days)
 - **long** longitude (positive if west of Greenwich; degrees)

Astronomical coordinates

➡ **[ao,do]=astro_coord(cin,cout,ai,di)** , astronomical coordinate conversion, from cin to cout. Angles and tsid are in radians. Local tsid and latitude is needed for conversions to and from the horizon coordinates.

○ **cin** and **cout** can be

'horizontal'	azimuth, altitude
'equatorial'	celestial equatorial: right ascension, declination
'ecliptical'	ecliptical: longitude, latitude
'galactic'	galactic longitude, latitude

epsilon = 23.4392911 deg is the aberration to right asc. and declination if they are referred to the standard equinox of year 2000.

(epsilon = 23.4457889 deg, if they are referred to the standard equinox of year 1950.)

Source and Antenna structures

All the functions that need antenna or source information, use the following structures:

➡ **source** structure (interactive function **sour=i_source**)

- **a** right ascension (degrees)
- **d** declination (degrees)
- **eps** percentage of linear polarization
- **psi** angle of linear polarization (respect to the source meridian)
- **f0** frequency (epoch 0)
- **df0** frequency first derivative (epoch 0) (frequency variation per day)
- **ddf0** frequency second derivative (epoch 0) (df0 variation per day)
- **snr** signal-to-noise ratio
- **fi** difference of phase between circular and linear polarization

➡ **antenna** structure (interactive function **sour=i_antenna**)

- **type** bar (1), interferometer (2),...
- **lat** latitude (degrees)
- **long** longitude (degrees)
- **heig** height (m)
- **azim** azimuth (degrees)
- **incl** inclination (degrees)

Doppler effect

The computation of the motion of the Earth respect to the Solar System Barycenter is computed by pss_astro, an adaptation and enhancement of a code from JPL and NOVAS software. Among the other possible uses of pss_astro, there is the production of a table, depending on the location of the antenna and the year, that contains the velocity vector of the detector at certain times. The program, in C, is [crea_table](#). For a given detector and a given time interval, expressed in mjd, it gives an output file with the following information (1 minute step):

- ➡ detector velocity vector \mathbf{v} (normalized to c) in rectangular Equatorial J2000 coordinates referred to SSB
- ➡ frequency variation, deinstein, due to the relativistic Einstein effect

For a given detector and source, emitting at f_s , the observed frequency f_{obs} is evaluated, using the following formula:

$$f_{obs} = f_s * (1 - \mathbf{p} \cdot \mathbf{v}) - f_s * deinstein$$

(where \mathbf{p} is the unitary position vector of the source, in rectangular equatorial J2000 coordinates).

The code asks for the name of the detector. Presently only the detectors Virgo, Explorer and Nautilus are considered, but we plan to insert soon all the others (it is only a question of inserting them in the "detector" structure)

- ➡ Detector ? (virgo, explorer, nautilus)
- ➡ Initial mjd, final mjd ? (from 1 Jan 1991 (48257)) e.g.: 48300 48500

The output is a file with the name table.dat. This is a typical output:

An example of the tables is the following (tableVirgo_2000-2010.dat, beginning):

```
virgo: lat=43.631389, long=10.504444, azim=10.00000 (deg)
velx/C, vely/C, velz/C in rect. Equatorial coord., deinstein
Observed frequency at the detector = source_freq*(1-(source_pos x vel/C) - deinstein)

51544.00000000 -0.000100557372253 -0.000016371087845 -0.000006927597163 0.000000000335206537
51544.00694444 -0.000100537101901 -0.000016427927612 -0.000006932417453 0.000000000335208710
51544.01388889 -0.000100514848156 -0.000016483926814 -0.000006937237591 0.000000000335210877
51544.02083333 -0.000100490649671 -0.000016538999573 -0.000006942057575 0.000000000335213040
51544.02777778 -0.000100464548820 -0.000016593061781 -0.000006946877404 0.000000000335215197
51544.03472222 -0.000100436591616 -0.000016646031264 -0.000006951697077 0.000000000335217350
```

Notes and observation:

1. this code is based on the PSS_astro library, whose documentation is given in the PSS_astro_UG.doc and PSS_astro_PG.doc

2. it uses the JPL ephemeris file DE405 and files from IERS for fine corrections of the time and nutation. These fine corrections (use of UT1 and not UTC, application of corrections to the nutation angles DP_{ψ} and DE_{ϵ}) are provided by IERS only up-to-date, with the file eopc04.62-now. In case these files are not present for the time lag (or part of it) provided by the user, then the code does not use these fine corrections and a message appears on the screen and also in the file which is created.
3. the deinstein corrections is at a level less than one part on one million.
4. As shown with details in the documentation, the Einstein effect is only a small correction to the final Doppler effect, but, given the fact that it can be evaluated using only information on the detector, that is it does not depend on the source position, we have decided to store also this information. The Einstein effect gives a contribution which is -roughly speaking- 5 orders of magnitude lower compared to the revolution and 3 orders of magnitude lower compared to the rotation. On the contrary, the Shapiro effect (which is even smaller) depends also on the source and thus cannot be evaluated at this stage. There is a function in the PSS_astro code to add the Shapiro effect, if a very high precision is needed. The Shapiro effect gives a contribution which is -roughly speaking- 3 orders of magnitude lower compared to the Einstein effect.

The time is **MJD**. Interpolation from a table at 10 minutes (one year about 5 MB, in matlab format 2 MB), gives rise to an error on each component less than 10^{-13} , so obtaining a maximum relative error of the order of 10^{-9} . Attention ! The computation of the MJD for future dates can be wrong by the unknown future leap seconds.

The Doppler table used by the software is a [5 (or 4) x n] matrix, with the first column containing the times (in TAI days) and the following 3 columns contain the 3 components of the detector velocity divided by c. The fifth column, if exists, contains the Shapiro effect. It is produced by the function

➡ **doptab=read_doppler_table(file,subsamp,fileout)** , with

- file file created by crea_table
- subsamp subsampling (one every...)
- fileout output file (can be not present)

The following function puts the Doppler table (the x, y, and z component of the velocity of the detector in single precision) in an sds file

➡ **doptab2sds(doptab,sdsfile,capt)**, with

- doptab the Doppler table
- sdsfile the sds file
- capt the caption of the sds file

These sds files (that are tiny) can be read to produce the matrix doptab; the procedure to do it is:

➡ **doptab=doptab_from_sds(subsamp,file)** , with

- **subsamp** the subsampling factor
- **file** the sds file (like **tableVirgo_2000-2010.sds** of 6670 kb)

Normally the Doppler tables are computed and stored for long periods (e.g. 2000-2010), but for practical purposes (typically for spline interpolation) it is more practical and fast to use shorter tables. This is accomplished by extracting a sub-table from a big table by

➡ **subdoptab=reduce_doptab(doptab,tmin,tmax)** , where doptab is the original table, tmin and tmax the time interval of the sub-table.

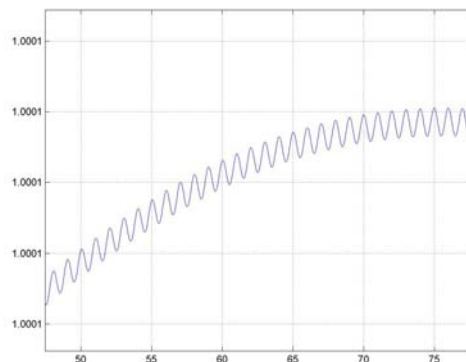
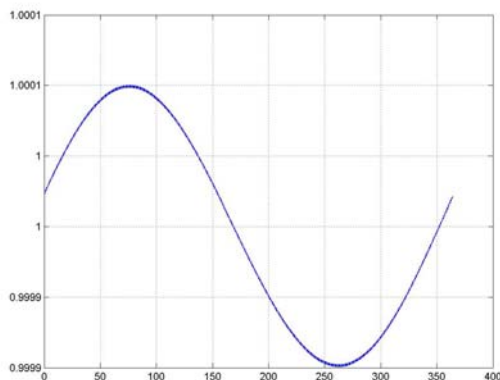
The following function is a simple use of **doptab**:

➡ **dop=gw_doppler(doptab,source,t)** , computes the percentage Doppler shift for the Earth motion. It works with a single time or with a single source

- **doptab** table containing the Doppler data (depends on antenna and year)
 (a matrix (n*4) or (n*5) with:
 - first column containing the times (normally every 10 min)
 - second col x (in c units, in equatorial cartesian frame)
 - third col y
 - fourth col z
 - fifth col de-einstein (if present)
- **source** pss structures or a double array (n*2) with
 - first col the sources alpha (in deg)
 - second col the source delta (in deg)
- **t** time array TAI (in mjd days)

This function uses the tables created by pss_astro, interpolating with **spline()**.

Here is the results for 2003:



There is also another function that computes the Doppler effect,

- ➡ **dop=gw_doppler1(v,source)** , that uses the velocity vector of the detector and the source.

Another example is

- ➡ **[v_x,v_y,v_z]=v_detector(doptab,t)** , that creates the three arrays containing the three components of the detector velocity at the times of the array t.

Sidereal response

Because of the radiation pattern of the antenna, the response to a gravitational source varies depending on the sidereal hour. A number of functions dealing with these problem are developed:

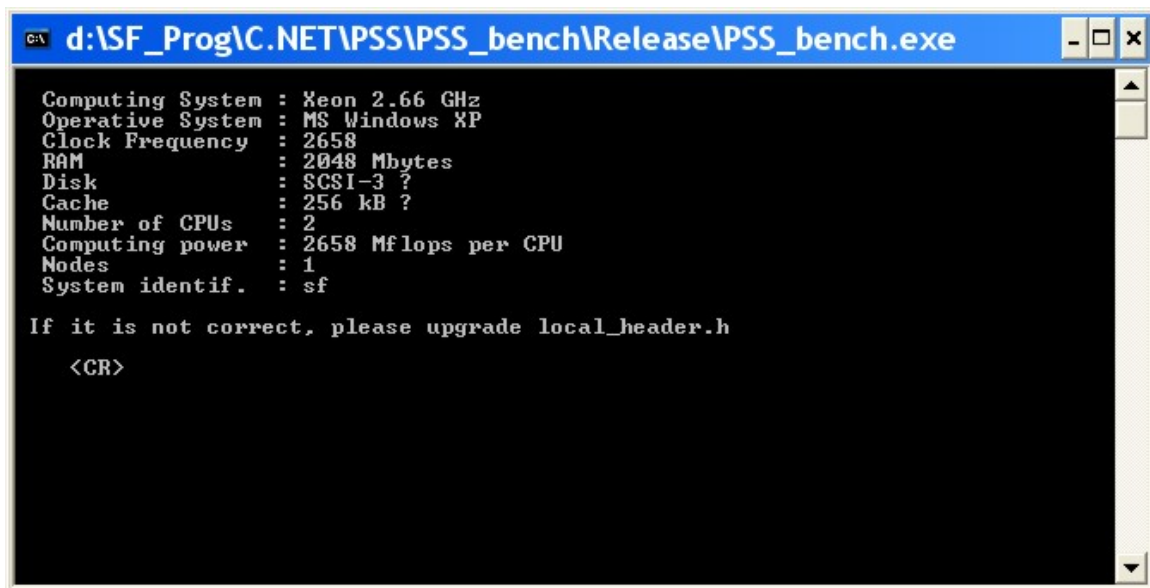
- ➡ **sr=sid_resp(antenna,source,n)** , gives the sidereal response in energy; the parameters are:
 - **antenna**, an antenna structure
 - **source**, a source structure
 - **n** , the number of points in a sidereal day

Tests and benchmarks

The PSS_bench program

The interactive program

The program starts from the command console. At the beginning the computer information appears



```
C:\ d:\SF_Prog\C.NET\PSS\PSS_bench\Release\PSS_bench.exe

Computing System : Xeon 2.66 GHz
Operative System : MS Windows XP
Clock Frequency  : 2658
RAM               : 2048 Mbytes
Disk              : SCSI-3 ?
Cache            : 256 kB ?
Number of CPUs   : 2
Computing power   : 2658 Mflops per CPU
Nodes            : 1
System identif.  : sf

If it is not correct, please upgrade local_header.h
<CR>
```

The computer information resides in a header named **local_header.h** and should be changed anytime that the benchmark is installed on a new computer. It should be re-compiled and linked for different configuration (important is the **no_optimization**).

Then the main menu appears. It shows the different classes of benchmarks that can be run.


```
D:\SF_Prog\C.NET\PSS\PSS_bench\Release\PSS_bench.exe
Computing power : 1680 Mflops per CPU
Nodes          : 1
System identif. : sf

If it is not correct, please upgrade local_header.h

<CR>

C benchmarks

1. Basic benchmarks
2. FFT benchmarks
3. Hough benchmarks
4. Coherent procedure benchmarks

6. Basic report
7. FFT report
8. Vector report
9. Hough report
10. Disc report

12. Exit

Which item ? _
```

Then the menu of the particular class of benchmark appears. For example, for the basic benchmark:

```
D:\SF_Prog\C.NET\PSS\PSS_bench\Release\PSS_bench.exe
12. Exit

Which item ? 1

Basic benchmarks chosen

Basic benchmarks

1. Integer
2. Float
3. Float new
4. Double

6. Sine function

8. Vectors
9. Crazy Vectors

11. Disk access
12. Existing file access
13. Create junk file

15. Main menu

Which item ?
```

The benchmarks started from the main menu can be also reports.

The reports

Basic report

PSS_bench report on Tue Feb 17 15:07:17 2004

```
Computing System : Xeon 2.66 GHz
Operative System : MS Windows XP
Clock Frequency  : 2658
RAM              : 2048 Mbytes
Disk             : SCSI-3 ?
Cache           : 256 kB ?
Number of CPUs   : 2
Computing power  : 2658 Mflops per CPU
Nodes           : 1
System identif.  : sf
```

Comment: ...
Lower values are better results

Basic tests

```
Integer: rough1,rough2,sum,product
11.615460,16.187220,4.598340,27.005280
Float   : rough1,rough2,sum,product
13.290000,6.219720,1.674540,1.249260
Double  : rough1,rough2,sum,product 19.934999,7.894260,-
0.850560,-0.398700
```

Sines : 203.337006

```
Vectors: length 100000 -> rough,sum = -7.043700,19.084440
Crazy   : length 100000 -> no crazy,crazy =
4.173060,40.667400
```

FFT tests

Four1 : 54.664391 (length 1048576)

Hough tests

LUT : 12.260115 (nalpha, ndelta = 720, 360)

FFT report

PSS_bench FFT (fftw) report on Tue Feb 17 15:07:42 2004

Computing System : Xeon 2.66 GHz
Operative System : MS Windows XP
Clock Frequency : 2658
RAM : 2048 Mbytes
Disk : SCSI-3 ?
Cache : 256 kB ?
Number of CPUs : 2
Computing power : 2658 Mflops per CPU
Nodes : 1
System identif. : sf

Comment: ...

Lower values are better results

length	efficiency	loss
1024	4.28	(8.2474e-005 s/fft)
2048	3.93	(1.6667e-004 s/fft)
4096	3.38	(3.1250e-004 s/fft)
8192	3.12	(6.2500e-004 s/fft)
16384	5.99	(2.5833e-003 s/fft)
32768	5.59	(5.1667e-003 s/fft)
65536	11.91	(2.3500e-002 s/fft)
131072	11.21	(4.7000e-002 s/fft)
262144	13.18	(1.1700e-001 s/fft)
524288	10.41	(1.9500e-001 s/fft)
1048576	12.48	(4.9250e-001 s/fft)
2097152	10.09	(8.3600e-001 s/fft)
4194304	10.04	(1.7425e+000 s/fft)
8388608	9.86	(3.5780e+000 s/fft)

PSS_bench FFT (fourl) report on Tue Feb 17 17:38:49 2004

Computing System : Xeon 2.66 GHz
Operative System : MS Windows XP
Clock Frequency : 2658
RAM : 2048 Mbytes
Disk : SCSI-3 ?
Cache : 256 kB ?
Number of CPUs : 2
Computing power : 2658 Mflops per CPU
Nodes : 1
System identif. : sf

Comment: ...

Lower values are better results

length	efficiency	loss
1024	840.26	(1.6186e-002 s/fft)
2048	729.80	(3.0927e-002 s/fft)
4096	535.14	(4.9479e-002 s/fft)
8192	169.10	(3.3875e-002 s/fft)
16384	5.99	(2.5833e-003 s/fft)
32768	5.59	(5.1667e-003 s/fft)
65536	43.60	(8.6000e-002 s/fft)
131072	50.34	(2.1100e-001 s/fft)
262144	54.58	(4.8450e-001 s/fft)
524288	59.61	(1.1170e+000 s/fft)
1048576	60.20	(2.3750e+000 s/fft)
2097152	64.78	(5.3670e+000 s/fft)
4194304	69.76	(1.2110e+001 s/fft)
8388608	77.66	(2.8188e+001 s/fft)
16777216	79.28	(6.0047e+001 s/fft)

SFDB

Hough transform

Service routines

Matlab service routines

- ➡ i_antenna
- ➡ i_source
- ➡ i_data
- ➡ pss_par
- ➡ ipss_par
- ➡ source_tab

pss_lib

pss_rog

General parameter structure

All the parameters used for this software are organized in nested structures. Here all these structures are described. Any parent structure can lack some children.

Main pss_ structure

This is the container for the other high level structures. The parameters are classified in 4 classes:

- 0 fixed
- 1 primary or input
- 2 derived
- 3 analysis results
- 4 other

Structures	Type	Use
const_		Mathematics, Physics and Astronomy constants
source_	may contain arrays	periodic source parameters
antenna_	may contain arrays	detector physical parameters
data_		data parameters
fft_		fft parameters
band_		band description
sfdb_	may contain arrays	short fft data-base parameters
tfmap_	may contain arrays	time-frequency map
tfpmap_	may contain arrays	time-frequency peak map
hmap_	may contain arrays	hough map parameters
cohe_		coherent follow-up parameters
ss_		supervisor parameters
candidate_	may contain arrays	periodic source candidates
event_	may contain arrays	event parameters
computing_		computing parameters

const_ structure

It contains:

constant	class	type	what
pi	0		3.1415926535897932384626433832795
e	0		2.7182818284590452353602874713527
c	0		light velocity - 299792458
G	0		gravitational constant - 6.67259E-11
deg2rad	0		degree to radiants conversion
Eorbv	0		Earth orbital velocity
Erotrv	0		Earth rotational velocity (at the equator)
SY_s	0		sidereal year seconds
SD_s	0		sidereal day seconds

source_ structure

All the angles are in degrees.

parameter	class	type	what
name	1		
a	1		right ascension or ecliptical longitude
d	1		declination or ecliptical latitude
eps	1		percentage of linear polarization power
psi	1		polarization angle
t00	1		f0 epoch (typically 1-1-2000)
f0	1		initial (original) frequency
df0	1		initial first derivative of the frequency (frequency variation per day)
ddf0	1		second derivative of the frequency (df0 variation per day)
h	1		h amplitude
snr	1		signal-to-noise ratio
coord	1	?	0 -> equatorial, 1 -> ecliptical
n	1		number of sources
chphase	4		phase of the last chunk (used by pss_chunk)

antenna_ structure

parameter	class	type	what
name	1		
long	1		
lat	1		
azim	1		azimuth
alt	1		altitude over sea level
incl	1		inclination
type	1		
bar_		structure	
itf_		structure	
n	1		number of antennas

data_ structure

parameter	class	type	what
dt	1		sampling time (s)
sf	2		sampling frequency
tobs	1		total observation time days)
t0	1		initial time (mjd)
iniwin	1	double array	starting time of windows
finwin	1	double array	ending time of windows
nwin	1		number of windows
t	4		time (mjd)

fft_ structure

This is a sub-structure used in (or in conjunction with) sfdb_, tfmap_ and tfpmap_ structures. It describes the working band. It can be also used alone.

parameter	class	type	what
len	1		fft length (number of samples)
tlen	2		fft time length
N	1		number of ffts
onev	1		take one fft every...
df	2		frequency bin
res	1		resolution
interl	1		interlacing
wind	1		window type
frin	1		initial frequency
tin	1		initial time
unit	1		unity

band_ structure

This is a sub-structure used in `sfdb_`, `tfmap_` and `tfpmap_` structures. It describes the working band. It can be also used alone.

parameter	class	type	what
Bf1	1		initial frequency of the full band
Bf2	1		final frequency of the full band
f0	1		supposed initial unshifted frequency
df0	1		f0 first derivative
ddf0	1		f0 second derivative
errf0	1		$(\text{band.f0-source.f0})/\text{fft.df}$
natb	3		"natural" sub-band (working band)
kmatb	1		widening factor of natural sub-band
bf1	3		sub-band initial frequency
bf2	3		sub-band final frequency

sfdb_ structure

parameter	class	type	what

tfmap_ structure

parameter	class	type	what

tfpmap_ structure

parameter	class	type	what

hmap_ structure

parameter	class	type	what

cohe_ structure

parameter	class	type	what

ss_ structure

parameter	class	type	what

candidate_ structure

parameter	class	type	what

event_ structure

parameter	class	type	what

computing_ structure

parameter	class	type	what

The PSS databases

General structure of PSS databases

The general organization of the PSS databases, i.e. the folder tree, is the following:

- First level: antennas and general metadata (general, server and analysis)
- Second level: data categories and antenna metadata. Data categories are:
 - **sd** – (sampled data) h-reconstructed or equivalent
 - **sfdb** - sfdb data
 - **nsp** - normalized spectra
 - **pm** - peak maps
 - **cand** - candidate
 - **metadata** - antenna metadata
 - **analysis** – analysis reports
- Third level (optional): different data types
- Other level: optional internal organization of sub-databases

The naming of files is such that they are alphabetically ordered for the basic key (normally time). A possible implementation is the following:

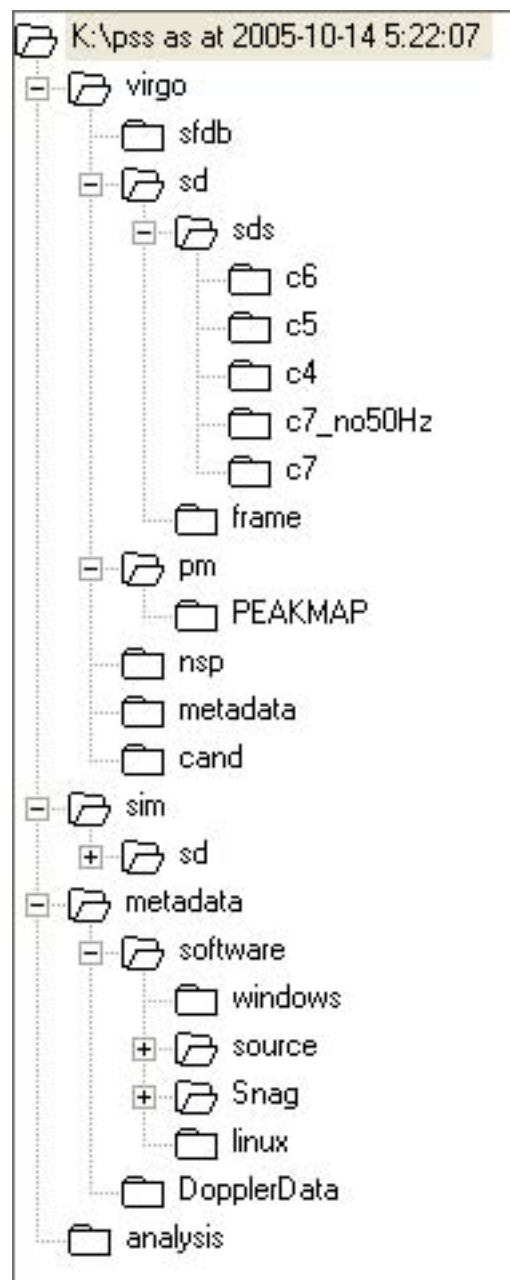
AntennaName_DataType_StartTime_Characteristics.FormatExt

where

- **AntennaName**, for example:
 - Vir
 - Nau
 - Exp
- **DataType**, for example:
 - raw
 - hrec
 - sfdb
 - nsp

- pm
- **StartTime**, in the format **YYMMDD_hhmmss**
- **Characteristics** depends on the category
- **FormatExt** depends on the data format (frame, r87, sds,...)

The pss directory structure



The h-reconstructed database

Third level is: frame, r87, sds. Fourth level can be runs or years.

The naming is derived as easily as possible, by the original antenna name.

The sfdb database

The normalized spectra database

The peak-map database

The candidate database

See “Candidate database and coincidences”, page 62.

Database Metadata

Server docs

It should contain information on the available servers, how to reach them, what is supposed to contain.

Analysis docs

This should contain the analysis batch log and where the results are stored.

Antenna docs

It should contain at least:

- Antenna basic information (e.g. the position)
- Channel names
- Basic information on the runs

File System utilities

Appendix

Doppler effect computation

pss_astro

PSS_astro, is a C code for the computation of various astrometric quantities, coordinate transformations, evaluation of Doppler effect. It uses a JPL C code to read and manage the ephemerides file **DE405**.

It also uses **NOVAS** (Naval Observatory Vector Astrometry Subroutines), a C code, for the computation of a wide variety of common astrometric quantities.

The JPL Planetary and Lunar Ephemerides can be downloaded from the site

<ftp://navigator.jpl.nasa.gov/pub/ephem/export/>

by choosing the appropriate way, depending on the operating system (**/unix** is the directory for unix users and **/ascii** for non-unix users).

The instructions are written in the description file, linked at

http://ssd.jpl.nasa.gov/eph_info.html

or directly at

<ftp://navigator.jpl.nasa.gov/pub/ephem/export/usrguide>

Each ascii file contains 20 years of data, while each unix binary file contains 50 years of data. Anyway, we have experienced some problems with the binary UNIX files, while the ASCII files work properly even on an UNIX machine.

Then, the procedure we have used to download the ephemerides **DE405** for the years **1980-2020** is:

- download, from the directory **/ASCII**, the files: **ascp1980.405**, **ascp2000.405**
- download, from the directory **/FORTRAN**, the code **asc2eph.f**, which must be used to convert the ASCII files into binary files and to merge them to form a single ephemeris file.

Then :

- in **asc2eph.f**, set the NRECL parameter to 4.
- compile and link
`(g77 -c asc2eph.f g77 -o asc2eph.out binmerge.o)`
- run the code, with the following syntax:
on Unix:
`cat header.405 ascp1980.405 ascp2000.405 | ./asc2eph.out`
on windows (using command console):
`copy header.405 ascp1980.405 ascp2000.405 infile.405`
- and then run `asc2eph.out infile.405`

This procedure produces the binary file **jpleph**.
We have renamed it to **Jpleph.405**.

Theory: Astronomical Times

A good introduction to the definitions of the Astronomical times is given here:
<http://www.gb.nrao.edu/~rfisher/Ephemerides/times.html>

TAI is the International Atomic Time

UTC is the Coordinated Universal Time

UTC = TAI - (number of leap seconds)

TDT is the terrestrial dynamic time. It is not yet used for planetary motions calculations, but it is only used to calculate TDB, which takes into account relativistic effects.

$TDT = TAI + 32.184 = UTC + (\text{number of leap seconds}) + 32.184$

it is tied to the atomic time by a constant offset of 32.184 seconds.

TDB is the Barycentric Dynamic Time

approximately:

$TDB = TT + 0.001658 \sin(g) + 0.000014 \sin(2g)$ seconds

where

$g = 357.53 + 0.9856003 (JD - 2451545.0)$ degrees

and JD is the Julian Date.

UT1 is the Universal Time, and it is a measure of the actual rotation of the Earth. Hence it is not uniform.

It is the rotation of the Earth with respect to the mean position of the Sun.

UTC is incremented by integer seconds (leap seconds) to stay within 0.7 seconds of UT1.

Then the difference between UT1 and UTC is never greater than this.

Planetary motions are computed using TDB.

UT1 should be used to evaluate

GMST, Greenwich Mean Sidereal Time

but UTC can be used as a good approximation of UT1.

In the code, we can use either UTC or UT1, by setting a flag.

Sidereal time is the measure of the Earth's rotation with respect to distant celestial objects.

Theory: Contributions to the Doppler effect

The Doppler shift, that is the frequency ν_{doppler} , observed at the detector, for a given source whose intrinsic frequency, supposed to be constant, is $\text{source} \rightarrow \text{frequency}$ is mainly the result of :

- 1) Earth revolution around the Sun
 - 2) Rotation
- and,
- 1) relativistic delay (Einstein effect)
 - 2) light deflection in the Sun's field (Shapiro effect).

To get an idea of the relative weight of these effects, we have run the code and we have written the separate contributions.

We have used: detector Parkes,
PSR 437 (supposed to be at rest, with ra and dec given at Epoch J2000). Their coordinates are both defined in the header file, *daspostare.h*.
Let us consider the quantity

$$z = (\text{source} \rightarrow \text{frequency} - \nu_{\text{doppler}}) / \text{source} \rightarrow \text{frequency}$$

At the MJD 49353.0833 (January 1th, 1994) we get:

- 1) Revolution: $2.9166917\textcolor{red}{23} * 10^{(-5)}$
- 2) Rotation: $-4.3508\textcolor{red}{614} * 10^{(-7)}$
- 3) Deinstein: $-3.35\textcolor{red}{40} * 10^{(-10)}$
- 4) Dshapiro: $-4.\textcolor{red}{65} * 10^{(-13)}$

Thus, the total value of the considered variable z is

$$2.8731\textcolor{red}{4952} * 10^{(-5)}, \text{ including Einstein and Shapiro.}$$

We have done a (not formal) comparison of these results with C. Cutler (Potsdam AEI). The results of the comparison have been:

- 1) Cutler Revolution: $2.9166917\textcolor{red}{3} * 10^{(-5)}$;
 - 2) Cutler Rotation: $-4.3508\textcolor{red}{340} * 10^{(-7)}$;
 - 3) Cutler Deinstein: $-3.3\textcolor{red}{439} * 10^{(-10)}$;
 - 4) Cutler Shapiro $-4.73 * 10^{(-13)}$;
- Cutler total value is $2.8731\textcolor{red}{5000} * 10^{(-5)}$.

We have indicated in red those numbers that are different from Cutler's.

Hence the difference between our result and Cutler result is, for the considered variable z, $4.8 * 10^{(-12)}$.

We recall that this comparison has been done not formally, on August 2000. In particular, we don't know if Cutler now is

still getting these numbers.

We have done this, and other comparisons using TEMPO.

Programming tips

Windows FrameLib

[FrameLib 6r18] To construct the FrameLib.lib, we have to comment the line

```
#include <unistd.h>
```

in **FrIO.c** . The functions **dup, open, close, write, read, lseek** are not defined. They are used if it is not defined **FRIOCFILE**.