Snag v2

Programming Guide

vers 14 February 2013

http://grwavsf.roma1.infn.it/snag/

Contents

New features of Snag2

The MatLab data analysis tool Snag started in the spring of 1998. In the spring of 2003 it consisted of about 500 m-files (in February 2005 they are 700, excluding obsolete and user functions), but there were difficulties to manage and develop such a number of functions. The goals of the project were partially changed and also the basic concept of the MatLab Snag was evolved. So it was decided to re-think some parts of the tool, and to mark the change, it was decided to name the new Snag, Snag v.2 or Snag2. Also if now (end of June 2003) about 300 m-files of the new Snag (that contains at that date more than 700 m-files, about 60 of which obsolete) were written before the year 2000.
The most evident feature of the second version of Snag is the organization of the function folders, to manage more easily the big amount of files. In particular only a very little number of m-files are allowed in the principal folder snag; there is a subfolder, also named snag, containing all the m-files related to the Snag GUI. A subfolder named projects contains, in an organized way, all the non-general use functions: in particular the sub-subfolder gw contains many different subfolders grouping different arguments.
Another big change is that the snf format is become obsolete, substituted by the "sfc" formats, more easy and and with new ideas behind. The full compatibility with the frame format is no more present (and may be it will be never again). This because an efficient use of the frame format should be possible with a MatLab frame library (written, with a certain effort, for the frame version 3 and updated for version 4, but now we are at version 6…). So to use efficiently Snag, some conversion tools are developed, to translate data from frame to sfc.
No more classes were added (the old ones remain with little changes), but two structures (almost classes ;-)) are added; ev (and the twin structure ew), that deals with events, and mp (that was created in 1999 for the data_browser) that deals with sets of sampled channels. Why structures and not classes ? In MatLab a structure is more agile than a class, so, if there are no other advantage, in some cases a structure can be preferable.

Snag Folders

The snag m functions (about 800 in January 2004), documentation and data files are grouped in folder:

@ds	**ds** (data stream) class members
@gd	**gd** (data group) class members
@gd2	**gd2** (2-D data group) class members
@rg	**rg** (data ring) class members
@rs	**rs** (resonances) class members
am	**am** (ARMA filters) structure functions
ev	**ev/ew** (events) structure functions
mp	**mp** (multi-plot) structure function
arp	double array processing functions
gdproc	**gd** processing functions
gds	**gd**s creation
serv	service functions
sfc	**sfc** (simple file format) functions
snag	**snag** gui functions
special	special functions
project	different user projects functions
extern	m functions and dll to access non-matlab programs
data	data (like filters or spectra)
doc	documentation
local	localization files (not in the path)
obsolete	obsolete files (not in the path)

gd Class

A **gd** is a “group of data”, defined by an abscissa and a single value; about these data, it is known the number. An example of a gd is a set of sampled data: if the sampling is uniform, one can overlook the abscissa (it can be computed by the beginning value and the sampling time), in which case we say it is a “virtual abscissa” gd (otherwise it is a “real abscissa” gd).

The data members are

x	abscissa (absent if type = 1, otherwise a column vector
y	ordinate (column vector)
n	length (number of data)
type	= 1 for virtual abscissa gd, = 2 for real abscissa gd
ini	beginning of (virtual) abscissa
dx	sampling interval (e.g. “time”) - for virtual abscissa gds
capt	caption
unc	uncertainty on y (optional)
uncx	uncertainty on x (optional)
cont	control variable - normally absent (0); may be an array or cell array, for particular uses)

gd2 (or DM) Class

A **gd2** (called also **dm**, Data Map) is a gd with a two dimension abscissa. It can be used for bi-dimensional data, as, for example, a time frequency spectrum. It can be used also for many uni-dimensional data groups with the same abscissae. A gd2 can be "virtual" or "real" abscissa, but only for the "primary" dimension; the secondary dimension is always "virtual".

Remember that MatLab stores matrices by columns, so it should be important to use the column index (the second) for the thing that changes more slowly.

The data members are

x	abscissa (absent if type = 1, otherwise a column vector)
y	ordinate (a (n/m)*m matrix)
n	total number of data
type	= 1 for virtual abscissa gd, = 2 for real abscissa gd
ini	beginning of (virtual) primary abscissa
dx	sampling "time" (for virtual abscissa gds)
m	secondary dimension
ini2	beginning of seconadary abscissa
dx2	secondary abscissa sampling
mcapt	multiple captions (a cell array, case of m uni-dimensional data)
capt	caption
cont	control variable - normally absent (0); may be an array or cell array, for particular uses)

ds Class

As a **gd** is a “group of data” of determined (known and not too big) length, a **ds** (Data Stream) is used to handle sampled data (in time domain) with unknown (or very big) length, of which one has at a given moment just a chunk.
The data members are:

len	length of the chunks
dt	sampling time
capt	caption
type	type (1: not interlaced, y2 contains last but one; 2: interlaced by the half, alternate y1 and y2; 0 not interlaced, y2 not used)
treq	time requested (to start)
y1	odd chunk (last chunk if not interlaced) **OCCUPIES len*5/4**
y2	even chunk (last chunk but one if not interlaced)
tini1	time of the first sample of y1
tini2	time of the first sample of y2
ind1	index for y1 (particular uses)
ind2	index for y2 (particular uses)
nc1	serial number of y1 chunks
nc2	serial number of y2 chunks
lcw	last chunk written (“produced”)
lcr	last chunk read (“served” – for multiple clients use)
cont	control variable - normally absent (0); may be an array or cell array, for particular uses)

ds Operation

The basic idea is the client-server metaphor: the client asks for chunks of data, defining at the beginning the modalities and then calling iteratively the server.
There are three fundamental operations:

• Initial Setting

At this stage, a ds is created and the modalities of the data service are defined. This is done mainly by the methods **ds** (the constructor) and **edit_ds** (a modifier). At this stage the fundamental constants of the ds are set:

- **len**
- **dt**
- **capt**
- **type**
- **treq**

Then some variables are initialized:

- **nc1 = 0**
- **nc2 = 0**
- **lcw = 0**
- **lcr = 0**
- **ind1 = 1**
- **ind2 = 2**
- **cont = 0**
- **tini1 = -d.len*d.d**t (sometimes necessary)

A particular method is **reset_ds**, that resets the variables to the initial values.

• ds Servicing

This is done by particular methods that carry out the ds server operation.
Examples are:

- **data simulation servers**, as
  - **signal_ds**, that creates continuous signals (sinusoid,ramp,…)
  - **noise_ds**, that simulates a noise of given power spectrum

- **data access servers,** as
  - **r872ds**, that accesses data in **r87** format
  - **fr2ds**, that accesses data in **frame** format
  - **snf2ds,** that accesses data in **snf** format
- **other,** as
  - **gd2ds**, that creates a **ds** from a “long” **gd**

A **ds** server has the duty of setting the **ds** variables (except **lcr** and **cont**) .

The interlaced operations has the following scheme:

```
               Interlaced operation:

          chunk 1 y1    0 1 1 1 | 1
          chunk 2 y2    1 1 1 2 |
          chunk 3 y1    1 2 2 2 | 2   No new data
          chunk 4 y2    2 2 2 3 |
          chunk 5 y1    2 3 3 3 | 3   No new data
          chunk 6 y2    3 3 3 4 |
                     .   .   .   .
```

Attention: for this reason the real dimension of y1 is 5*len/4.

Moreover, in the case of interlaced operations, the data are shifted of len/4: the first len/4 data are set to 0; this because to not overlook the beginning data.

• Client Processing

This is done by functions that operates on the chunks served by a ds server.

Examples are

- **pows_ds, ipows_ds, ipows_ds_ng**, that compute running power spestra, with different characteristics
- **running_ds**, that does running plot of the data
- **stat_ds**, that performs running statistics.
- **write_snf_ds**, to store the data on a file (see the SNF section).

Tipically a client ds processor works independently of the type of the ds server.

Besides these three basic operations, there is another important operation:

• ds transformation

This is performed by a ds transformer, that has both the characteristics of a ds server and of a client processor. Examples are

- **to_interlace_ds**, that transforms a type 1 ds to a type 2 ds
- **de_interlace_ds**, that transforms a type 2 ds to a type 1 ds
- **ffilt_go_ds**, that creates a frequency domain filtered ds, from another ds.

A ds transformer has the same “duties” of a ds server, for the generated ds.

am structure

The **am** structure describes an ARMA filter of the type

```
y(n) = b0 * x(n) + b(1) * x(n-1) +  ... + b(nb) * x(n-nb)
                 + a(1) * y(n-1) -  ... - a(na) * y(n-na)
```

The elements of the structure are :

am.a	AR coefficients
am.na	number of AR coefficients
am.b	MA coefficients
am.nb	number of MA coefficients
am.b0	MA 0-th coefficient
am.bilat	= 1 bilateral filter
am.capt	caption
am.cont	control variable

Basic functions are:

- **crea_am** that creates different types of am filters (low-pass, high-pass, resonances,…)
- **am_multi** that multiplies two filters
- **am_divi** that divides two filters
- **am_filter** that applies an am filter to a data vector
- **am_trfun** that computes the transfer function of a filter
- **am_pulres** that computes the pulse response of a filter

mp structure

The **mp** structure is introduced to contain multi-channel sampled data.

The elements of the structure are:

mp.nch	number of channels
mp.ch(i).name	channel name
mp.ch(i).n	dimension of x,y
mp.ch(i).x	... ; if dim x is 1, x(1) is the beginning and mp.ch(i).dx
mp.ch(i).dx	…
mp.ch(i).y	…
mp.ch(i).unitx	…
mp.ch(i).unity	…
mp.ch(i).ch	ch number (primary key for chstr - optional)
mp.x	abscissa (equal for all channels) if ch(i).x is absent; if dim x is 1, x(1) is the beginning and mp.dx is the sampling period
mp.dx	…
mp.n	…
mp.unitx	…

ev / ew structure

The **ev** structure is used for describing events or sets of events (**ew**).
The **ew** array structure is more compact and simpler to use, but doesn't contain the event shape.
The two functions ev2ew and ew2ev convert ev in ew and vice versa (ev2ew obviously loses the shape).
A set of events is associated with a channel structure that describes all the available channels (for one or more antennas), so each event is associated to a channel of the set.

The channel structure

All cases	
ch.na	number of antennas
ch.nch()	number of channels for each antenna
ch.an(nchtot)	antenna
ch.st(nchtot)	sampling time
ch.ty(nchtot)	type
ch.cf(nchtot)	central frequencies
ch.bw(nchtot)	bandwidth
ch.win(:,nchtot)	windows (the win vector contains alternate, for any channel, ini,fin of each window); may be absent.
Simulation	
ch. lcn(nchtot)	lambdas for channel noise (norm to tobs)
ch. lds(nchtot)	local disturbance sensitivity
ch.lml()	lambdas for loc. dist. (for any antenna - norm to tobs)
ch. gws(nchtot)	gravitational wave sensitivity
ch.lmg	lambda for gw (norm to tobs)
Statistics	
ch. nev(nchtot)	number of events found
Analysis	
ch. dt	dead time for cluster analisys (s)

The ev structure

ev(i).t	(starting) time (days, normally mjd)
ev(i).tm	time of the maximum
ev(i).ch	channel
ev(i).a	amplitude
ev(i).cr	critical ratio
ev(i).a2	secondary amplitude (e.g. bandwidth)

ev(i).l	length (s)
ev(i).fl	flag
ev(i).ci	cluster index (0 default)
ev(i).shy()	shape
ev(i).sht	shape initial time
ev(i).shdt	shape sampling time

The ew structure

ew.nev	number of event
ew.t()	(starting) time (days, normally mjd)
ew.tm()	time of the maximum
ew.ch()	channel
ew.a()	amplitude
ew.cr()	critical ratio
ew.a2()	secondary amplitude (e.g. bandwidth)
ew.l()	length (s)
ew.fl()	flag
ew.ci()	cluster index (0 default)

PSC – Periodic Source Candidate

The periodic source candidates are structures containing the following elements:

psc.n	number of candidates
psc.tim	starting time (mjd)
psc.dt	number of spin-down parameters
psc.lfft	length of the fft
psc. nsd	number of spin-down parameters
psc.fr(i)	start frequency
psc.lam(i)	ecliptical longitude (lambda)
psc.bet(i)	ecliptical latitude (beta)
psc.sd1(i)	first spin-down parameter
psc.sd2(i)	second spin-down parameter (possibly absent)
psc.sd3(i)	third spin-down parameter (possibly absent)
…	
psc.cr(i)	statistical significance
psc.h(i)	h amplitude

It is intended that a psc structure has the same tim, dt, lfft and nsd for all the candidates that contains.

PSC Data Base

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_DBxxxxxx .

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, **xxxxxx1394** or **xxxxxx0101**.

Each file has the following structure:

Header		
initial time (mjd)	**double**	**1-8**
sampling time	**double**	**9-16**
log2 FFT length	**int2**	**17-18**
initial frequency basic group (multiples of 32768)	**int2**	**19-20**
delta lambda	**float**	**21-24**
delta beta	**float**	**25-28**
delta sd1	**float**	**29-32**
delta CR	**float**	**33-36**
ini h	**float**	**37-40**
delta h	**float**	**41-44**
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**
h index	**int2**	**11-12**

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*12+44 ~ **6 MB**

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

FDF – Frequency Domain Filters

Here is the description, from the programming point of view, of the Snag Multi-Filter methods.
A multi-filter is a set of filters that operate mainly in the frequency domain and have a common part (normally adaptive). It is described by an **ff_struct** structure

0	**.n**	number of filters (multiplicity)
1	**.lfft**	length of the principal fft (must be set equal to the input ds chunk length
0	**.pfilt**	principal filter type (‘nothing’, ‘whitening’, wiener’,…)
210	**.pfy**	double array containing the principal filter (may be adaptively variable)
0	**.tau**	adaptivity time (in number of periodograms)
1	**.stau**	adaptivity time (in seconds)
1	**.w**	AR coefficient
2	**.wnorm**	normalization variable
0	**.capt**	general caption
0	**.sfilt(k)**	sub-filters structures
0	**.sfilt.rlfft**	sub-filter ratio lfft primary/secondary
0	**.sfilt.fshift**	sub-band shift (in frequency)
1	**.sfilt.nshift**	sub-band shift (in frequency bins)
0	**.sfilt.mode**	sub-filter type (‘nothing’, ‘gauss’, ‘lorentz’,…)
0	**.sfilt.par(k)**	sub-filter parameters
0	**.sfilt.capt**	sub-filter caption
10	**.sfilt.sfy**	array containing the sub-filter

The first column codes are:

0	set by the user
1	set by **ffilt_open_ds**
2	set by **ffilt_go_ds**

A **ff_struct** (the **0** members, set by the user), can be “stored” in an m-file (typically in the snag/analysis/filters folder) and “retrieved” by the **run_m_file** snag function (interactively called by **irun_m_file**).

FDF Operation

A Snag multi-filter operates by means of two functions:

- **ffilt_open_ds**, that performs the initializations
- **ffilt_go_ds**, that applies the filters.

SFC – Simple File Format Collection

The SFC substitutes the SNF format for storing any Snag object, or, in general, many types of simple or complex data sets.

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
- **dds**, similar to sds but in double precision
- **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..
- **sev**, *simple event* data format, used to store events, all with the same byte length.

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).

Basic SFC file format

(SDS and SBL are some of the file formats of the SFC collection)

The format of the General SFC Header is the following :

label	8 bytes string (1-8)	type label (e.g. #SFC#SDS)
prot	4 bytes int32 (9-12)	protocol (integer; now 1)
coding	4 bytes int32 (13-16)	machine and data coding (normally 0)
nch	4 bytes int32 (17-20)	number of channels (**N**; integer)
point0 ex inidat	4 bytes int32 (21-24)	pointer to the beginning of data (>= 944+N*128)
len	8 bytes int64 (25-32)	number of blocks (sbl) or data per channel (sds) (0 if it is not known)
t0	8 bytes double (33-40)	beginning time (double); may be not meaningful or have user meaning
dt	8 bytes double (41-48)	sampling time (double); may be not meaningful or have user meaning
capt	128 bytes string	caption
filme	128 bytes string	original name of the file
filmaster	128 bytes string	master file or #NOFILE or original directory
filspre	128 bytes string	serial preceding file or #NOFILE
filspost	128 bytes string	serial subsequent file or #NOFILE
filppre	128 bytes string	parallel preceding file or #NOFILE or filspre directory (without / or \)
filppost	128 bytes string	parallel subsequent file or #NOFILE or filspost directory (without / or \)

Here ends the general part, then starts the peculiar part (for sds, sbl, vbl, gbl):

ch( )	N*128 bytes	depends on the sfc type the channels can be sampled data, single parameters, sets of parameters, matrices, strings, etc.
user	free	user file header The user header length is (inidat - 944+N*128) bytes
data	…	the data (depends on the sfc type) the data are in N parallel streams (sds) or divided in blocks (sbl)

To access the user header, fseek(fid,944+N*128,'bof')

sfc_ structure

sfc_.file	file name (with path)
sfc_.pnam	file path
sfc_.fid	fid
sfc_.label	label (e.g. #SFC#SDS)
sfc_.prot	protocol
sfc_.coding	machine and data coding
sfc_.t0	beginning time
sfc_.dt	sampling time
sfc_.capt	caption
sfc_.nch	number of channels
sfc_.ch(nch)	channel structures (depends on type)
sfc_.hlen	non-user header length (bytes)
sfc_.len	length of a stream (the total number of data is len*nch) or number of blocks
sfc_.userlen	length of the user header (bytes)
sfc_.point0	pointer to beginning of data
sfc_.eof	end of file (1; for chained files, 2 -> end of chain, 3 -> error, -1 -> end chosen period)
sfc_.acc	access number; 0 at beginning, incremented by user if not accessed in standard ways
sfc_.filme	original name of the file
sfc_.filmaster	master file
sfc_.filspre	serial preceding file
sfc_.filspost	serial subsequent file
sfc_.filppre	parallel preceding file
sfc_.filppost	parallel subsequent file

SDS file format

SDS is one of the file formats of the SFC collection

The SDS (Simple DS) format is intended for storing one or more data streams, with the same time beginning and the same sampling time.

The label field must contain #SFC#SDS. The first 944 bytes are the general SFC header.

The peculiar part is:

ch( )	N*128 bytes	The channels can be sampled data, single parameters, sets of parameters, matrices, strings, etc.
user	free	user file header The user header length is (inidat - 944+N*128) bytes
data	N*4*len bytes float	the data are in N parallel streams (len for each channel; float)

The sds_ structure derives from the sfc_ structure. The added or modified members are:

sds_.ch(nch)	channel captions
sds_.len	length of a stream (the total number of data is len*nch)
sds_.point	pointer for next data

A particular type of sds file is the t-type in which the first channel is a time abscissa (normally in days).It is recognized by the channel caption that begins with "timing channel". Typically the time abscissa is relative to the integer part of sds_.t0 (to reduce the effect of the low precision of float32 data.

SDS file operation

The main function available for SDS files and data management are:

- **Service functions**

  ➧ **sds_show**

- ➧ check_sds
- ➧ check_sds_conc
- ➧ sds_concatenate
- ➧ sds_reshape
- ➧ sds_selch
- ➧ sds_getchinfo
- ➧ sds_check_time
- ➧ sds_resume

- **Read/Write functions**
  - ➧ sds_openw
  - ➧ sds_open
  - ➧ i_open_sds
  - ➧ basic_sds, a template for sds_openw
  - ➧ vec_from_sds, fills vec with data from sds files

    [vec,sds_,tim0,holes]=vec_from_sds(sds_,chn,len,alpers)

    Rules:

    - If the file ends, opens the following one
    - In case of holes, fills with zeros
    - Vectors doesn't starts in holes, but with the beginning of next file
    - If "alpers" (Allowed Periods) is operative (present), non-allowed periods are zeroed. All-zeroes vectors are jumped

sds_	sds structure
chn	number of the channel
len	length
alpers	(n,2) array containing the start and stop time of the n allowed periods It is omitted or 0 if there are no selection periods.

vec	the data
tim0	first sample time (in mjd)
holes	structure describing holes
.nztot	total number of inserted zeros

.nzeros	number of inserted zeros (for each hole - possibly an array)
.kzeros	vec index that starts the zeros (possibly an array)

- **ss_vec_from_sds**, similar to vec_from_sds, but sub-samples the data

- **GD, DS and MP functions**
  - **sds2gd**
  - **sds2gd_selind**
  - **sds2gd_selt**
  - **sdst2gd** creates a type-2 gd from a t-type sds
  - **sds2gd2** creates a gd2 with (row,col) = (nch,len/nch)
  - **sds2mp**
  - **sds_writegd**
  - **sds_writegd2**
  - **read_sfc_ds**

- **Application functions**
  - **sds_simVirgo**
  - **sds_3chan**
  - **sds_ns**
  - **sds_spmean**

DDS file format

DDS is one of the file formats of the SFC collection

The DDS (Double Precision DS) format is analogous to the SDS format, but the data are stored in double precision.

The label field must contain #SFC#DDS. For all the rest the format is similar to the SDS.

SBL file format

SBL is one of the file formats of the SFC collection

The SBL (Simple block data format) format is intended for storing one or more data sets by means of "blocks" composed of sub-blocks, in a variety of different cases.

The label field must contain #SFC#SBL. The first 944 bytes are the general SFC header.

The peculiar part is:

ch( )	N*128 bytes	(total)
ch( ).dx	double (1-8)	sampling first dimension (if any, otherwise 0)
ch( ).dy	double (9-16)	sampling second dimension (if any, otherwise 0)
ch( ).lenx	int32 (17-20)	length first dimension (number of rows)
ch( ).leny	int32 (21-24)	length second dimension (for single dim arrays, 1)
ch( ).inix	double (25-32)	beginning of first dimension
ch( ).iniy	double (33-40)	beginning of second dimension
ch( ).type	int32 (41-44)	type of data: 1 byte (typically unsigned) 2 int16 3 int32 4 float 5 float complex 6 double 7 double complex .. compressed formats
ch( ).name	84 bytes string (45-128)	channel name (the first run of non-blank chars) and caption
user	free	user file header
data blocks	…	…

The sbl_ structure derives from the sfc_ structure. The added or modified members are:

sbl_.ch(nch)	channel captions
.dx	
.dy	
.lenx	number of rows
.leny	number of columns
.type	data type of the channel
.name	channel name

.capt	caption
.len	length of the sub-block (in bytes)
.inix	initial value of the first abscissa
.iniy	initial value of the second abscissa
….	other (depending by the data types)
.bias	position of the first data of the block (in bytes)
.k	number of read data (pointer to the next data)
sbl_.len	number of blocks
sbl_.blen	block length (in bytes)
sbl_.point	pointer for next data
sbl_.bltime	block time
sbl_.eob	end of block

Structure of the sbl blocks

There are ordinary and extra-ordinary blocks. All blocks have the same length. All ordinary blocks have the same structure. Extra-ordinary blocks are user managed.
Each block is composed by "channels" (or sub-blocks), containing a short header and data, in the following way:

block number	a 16 byte string as "[BLNxxxxxxxxxxx]" for ordinary blocks or "[BLExxxxxxxxxxx]" for extra-ordinary blocks	**Block header**
block time	double; may be not meaningful or have user meaning	
ch(1).inix	double; always present; may be not meaningful or have user meaning	**sub-block 1**
ch(1).iniy	double; always present; may be not meaningful or have user meaning	
ch(1).par1	special data type specific; absent for standard data	
ch(1).par2	special data type specific; absent for standard data	
…	…	
A(lenx,leny)	data of channel 1	
ch(2).inix	double; always present; may be not meaningful or have user meaning	**sub-block 2**
ch(2).iniy	double; always present; may be not meaningful or have user meaning	
ch(2).par1	special data type specific; absent for standard data	
ch(2).par2	special data type specific; absent for standard data	
…	…	
A(lenx,leny)	data of channel 2	
…		**other sub-blocks**

The standard data types are:

Type	**i.e.**	**Number of bytes**
1	**char**	**1**
2	**int16**	**2**
3	**int32**	**4**
4	**float**	**4**

5	**float complex**	**8**
6	**double**	**8**
7	**double complex**	**16**

The length of a block (in bytes) can be computed (for standard data), as

$$24 + \sum_{i=1}^{N_{ch}} \left(16 + L_i \cdot n_i\right)$$

where the sum is on all the channels, L is the number of data for each channel and n the number of bytes for one datum of each

VBL file format

VBL is one of the file formats of the SFC collection

The vbl (Variable block data format) format is intended for storing one or more data sets by means of “blocks” composed of sub-blocks, in a variety of different cases.
The base structure is similar to the SBL format, the unique difference is that every sub-block array is preceded by a label as [CHxxxx] and two int32 with the two dimensions of the array (plus the inix and iniy as in SBL). At the end an “index” with the pointers to the blocks may be present.

The label field must contain #SFC#VBL. The first 944 bytes are the general SFC header.

The peculiar part is the same of SBL format (also if some ch parameters may be not meaningful (or have user meaning), because are substitutes by the sub-block values)

ch( )	N*128 bytes	
ch( ).dx	double (1-8)	sampling first dimension (if any, otherwise 0)
ch( ).dy	double (9-16)	sampling second dimension (if any, otherwise 0)
ch( ).lenx	int32 (17-20)	length first dimension (number of rows)
ch( ).leny	int32 (21-24)	length second dimension (for single dim arrays, 1)
ch( ).inix	double (25-32)	length first dimension (number of rows)
ch( ).iniy	double (33-40)	length second dimension (for single dim arrays, 1)
ch( ).type	int32 (41-44)	type of data: 1 byte (typically unsigned) 2 int16 3 int32 4 float 5 float complex 6 double 7 double complex .. compressed formats
ch( ).name	84 bytes string (45-128)	channel name (the first run of non-blank chars) and caption
user	free	user file header
data blocks	…	…

The vbl_ structure derives from the sfc_ structure. They are equal to the case of SBL, but some parameters change at every block and there is the pointer “**nextblock**”:

vbl_.ch(nch)	channel captions

.dx	
.dy	
.lenx	number of rows
.leny	number of columns
.type	data type of the channel
.name	channel name
.capt	caption
.len	length of the sub-block (in bytes)
.inix	initial value of the first abscissa
.iniy	initial value of the second abscissa
….	other (depending by the data types)
.bias	position of the first data of the block (in bytes)
.k	number of read data (pointer to the next data)
vbl_.len	number of blocks
vbl_.point(nch)	pointer for next data
vbl_.block	block number
vbl_.nextblock	pointer to next block
vbl_.bltime	block time
vbl_.eob	end of block
vbl_.ch0	
.chnum	parameters of present channel
.dx	“
.dy	“
.lenx	“
.leny	“
.type	“
.inix	“
.iniy	“
.next	“

Structure of the vbl blocks

There are ordinary and extra-ordinary blocks. All ordinary blocks have the same structure. Extra-ordinary blocks are user managed.
Each ordinary block is composed by "channels" (or sub-blocks), containing a short header and data, in the following way:

block number	a 16 byte string as "[BLNxxxxxxxxxxx]" for ordinary blocks or "[BLExxxxxxxxxxx]" for extra-ordinary blocks	**Block header**
block time	double; may be not meaningful or have user meaning	
pointer to next block	int64 (may be 0)	
ch number	a 8 byte string as "[CHxxxx]"	**sub-block 1**
pointer	int64 pointer to next channel (0 → next block)	
ch(k1).dx	double; always present; may be not meaningful or have user meaning	
ch(k1).dy	double; always present; may be not meaningful or have user meaning	
ch(k1).lenx	int32	
ch(k1).leny	int32	
ch(k1).inix	double; always present; may be not meaningful or have user meaning	
ch(k1).iniy	double; always present; may be not meaningful or have user meaning	
ch(k1).type	int32	
ch(k1).par1	special data type specific; absent for standard data (user managed)	
ch(k1).par2	special data type specific; absent for standard data (user managed)	
…	…	
A(lenx,leny)	data of channel 1	
ch number	a 8 byte string as "[CHxxxx]"	**sub-block 2**
pointer	pointer to next channel (0 → next block)	
ch(k2).dx	double; always present; may be not meaningful or have user meaning	
ch(k2).dy	double; always present; may be not meaningful or have user meaning	
…	…	
…	…	
A(lenx,leny)	data of channel 2	
…		**other sub-blocks**

Typical length of a channel (if no special data are present) is

ch_header (60 bytes) + data length

Typical length of a block (if no special data are present) is

block_header(32 bytes)+channels

At the end, after all blocks, may be present a block index, with the following structure:

pointer to bl 1	8 byte integer
pointer to bl 2	8 byte integer
…	
total number of blocks	8 byte integer
[-INDEX]	8 byte integer

The length of the index is (Nbl+2)*8 bytes.

SEV file format

SEV is one of the file formats of the SFC collection

The sev (Simple event data format) format is intended for storing a number of events, each of them has a predetermined number of parameters (and so a predetermined byte occupancy).
The data for each event are divided in 4 groups:

type	number (as channels)	length	notes
double precision	nd	nd*8	the first typically is the time as mjd
integer	ni	ni*4	
float	nf	nf*4	
array	na (1 or 0)	lar*4	all float (may contain the event shape)

So there are NC=nd+ni+nf+na parameters (or channels in the language of SFC) and NC appears as the **nch** of the basic header.

In the basic header we have **len** as the total number of events. Typically the **t** and **dt** are the beginning and the duration of the observation time (but it is not mandatory).

After the basic header there are the standard “channels” (or parameter) descriptions (128 bytes each), then possibly the user data (also a simple text description, whose length is put in the sev_.userlen), and then, at sev_.point0, a short supplementary header with the 4 32-bits integers with the values of nd, ni, nf and lar.

The events start at (sev_.point0+16).

The length of each event in bytes is

sev_.evlen = nd*8+(ni+nf+lar)*4

Compressed data formats

The goal of these formats is to achieve high compression with little loss (if any) of information.

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

$$\mathbf{x_i = S_i * m * b^{E}{}_i}$$

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

$$\mathbf{x_i = S * m * b^{E}{}_i}$$

where

- $\mathbf{S_i}$ 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
- $\mathbf{E_i}$ 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.

When archived, LogX data are stored in this way:

Data	**Type**	**Position**
stat , a variable that describes the coding	short	1-16
N , the number of the coded data the number of bytes to be read is computed from this and stat	long	17-48
m and **b** (if the coding is linear or logarithmic)	2 doubles	49-112
- the coded data - or just the original float data in case of X=32 - or the constant float if X=0	…	…

.
The look-up table, if needed, must be defined elsewhere (e.g. in a file header).

The stat variable has the following bit fields:

stat[0]	= 0 all negative,	= 1 all positive
stat[1]	= 0 all with the same sign,	= 1 mixed

	(overcomes stat[0])
stat[2]	= 1 all equal data
	(overcomes stat[6])
stat[3-5]	= expX : X = 2^expX (max 32 == not coded)
stat[6]	= 0 logarithmic, = 1 linear

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 matrices) 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 0 1 1 0 0 0}

coded as {3 6 8 0 3}

aggregate binary case:

{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}

binary derived as

{1 0 0 1 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 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^n-2 zeroes should be represented; in such case [obsolete: a "0" means 2^n-1 zeroes without a non-zero. For example, with n=8, {0 0 7} means a run of 255+255+6 = 516 zeroes] the datum is put in a side array of uint32.

The practical coding of a sparse vector produces one or two vectors (zero runs and, in the non-binary case, the non-zero element arrays) preceded by a long defining the length of the vector/s and an uint variable stat the bits of which give the information on the actual format (index starts from 0, the first 8 are reserved for LogX):

stat(8)	= 0 non-binary,	= 1 binary
stat(9)	= 0 non-sparse,	= 1 sparse
stat(10)	= 0 normal,	= 1 derived
stat(11-12)	= dimen :	b = 4*2^dimen
stat(14)	= 1 logx format for non-zero elements	

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

When archived, SpVec data are stored in this way:

- the short stat

if a sparse non-binary format is used
- a long with the number of the original array elements
- a long with the number of the sparse elements
- the char array with the 0-run array; the number of char to be read is computed from stat and the number of the sparse elements
- a LogX array (see the preceding subsection) with the non-zero values or a long with the number of non-zero elements and a float array with the non-zero elements

if a sparse binary format is used (derived or not)
- a long with the number of the original array elements
- a long with the number of the sparse elements
- the char array with the 0-run array; the number of char to be read is computed from stat and the number of the sparse elements
- a float with the value of the non-zero element

if a non-sparse non-binary is used
- a LogX array (see the preceding subsection) with all the values or a long with the number of the elements and a float array with the vector

if a non-sparse binary is used
- a long with the number of original array elements
- the char array with the bit array

Use of SFC formats for standard Snag objects

The standard Snag data objects are:

- **GD**
- **DM or GD2**
- **DS**
- **MP**
- **EV**
- **PSC**

Each of these has a peculiar use of the SFC.

There are some function to do this:

- **sds_writegd(folder,file,gd)** that writes a gd in an sds file.
- **sds2gd(file,k),** that puts in a gd the content of an sds file
- **sds2gd_selind(file,k,minind,maxind)**, that puts in a gd the content of a selection (based on the channel number and the index of the samples) of an sds file
- **sds2gd_selt(file,chn,t)**, that puts in a gd the content of a selection (based on the channel number and the abscissa of the samples) of an sds file
- **sds2mp(file,t)**, creates an mp with the data contained in an sds file
- **[d,sds_]=sds2ds(d,sds_,chn),** generates a data-stream from an sds file; this is a ds server

Use of SFC formats for PSS

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 SBL file or in an VBL file, depending if a standard or compressed format is chosen. The structure is similar to that of the SFDB, but a peak vector takes the place of the FFT. If the standard format is chosen, the real vector contains many zeros and the values of the peaks above a statistical threshold. If the compressed format is chosen, the peak vectors are stored as sparse binary vector or sparse vectors (with also the amplitude information), so the real length of each block is not constant (and the VBL file format is chosen).

- **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_Browser

The data browser is a GUI application to easy access or simulate data.

Almost all the operations are based on the use of a structure, the **D_B** structure.

string	**D_B.access**	"by file" or "by time"	
double	**.data.type**	1: snf, 2: frames, 3: R87, 4: A.V. format, 100: simulation	
string	**.data.file**	selected data file; in the simulation case, the file containing the spectrum	
double	**.data.initim**	initial time (MJD)	
double	**.data.duration**	duration (D)	
string	**.data.chname**	channel name	
double	**.data.chnumber**	channel number	
	.data.dt	sampling time	
	.data.dlen	data chunks length	
	.data.sp	spectrum vector	
	.data.frame4par	a structure containing channel parameters for Frame Format version 4	
	.machtype	machine type	
	.uleaps	leap seconds between GPS/TAI and UTC	
	.nframe	number of frames in 'file'	
	.loctime	local seasonal time – UTC in seconds	
	.t0	frame start time	
	.dt	sampling time	
	.framedurat	frame durations in sec	
	.distch(nframe)	positions of selected channel in bytes from beginning of file	
	.compress	compression type	
	.type	data type	
	.ndata	Length of data vector (same as dlen)	
	.filter	an ff_struct containing information for the filter	
	.proc.type	**rplot**	running plot

		rpows	running power spectrum
		tfpows	time-frequency power spectrum
		rhist	running histogram
		evenf	event finder
		summary	summary of the data
		extrgd	extraction of data into a gd
		dtfpows	differential time-frequency power spectrum
	.proc.iter	number of iterations	

D_B Operation

The sequence of operation is :

Where	Call	What
D_B	**[ntype,file,pnam]=db_fildatsel**	file selection
D_B	**[chn,chs,dt,dlen,...]=db_selch(ntype,file)**	channel selection
D_B	**[out_db,D_B]=go_db(D_B)**	processing distribution
go_db	**[out_sp,D_B]=d_b_rpows(D_B)** **or [out_sp,D_B]=d_b_rplot(D_B)** **or [out_sp,D_B]=d_b_tfpows(D_B)** **or [out_sp,D_B]=d_b_rhist(D_B)** **or …**	processing effective procedure
d_b_xxx	**[d,r,fid,reclen,g,r_struct]=db_open(...)**	opens the files and initialize the ds and r_struct
d_b_xxx	**[d,r,r_struct]=db_gods(...)**	opens the files and initialize the ds and r_struct
db_gods	**ds server** e.g. **[d,r_struct]=sds2ds(d,r_struct,chn)**	takes data
d_b_xxx	**[powsout,answ]=ipows_ds(...)**	data operation

r_struct is an snf read structure, defined in **read_snf_gd** (obsolete), or (for sds files) an sds open structure, defined in **sds_open**.

GW_Sim

GW_Sim is a Snag application to simulate signals and noises for gravitational wave antennas.
The simulation parameters are stored in a GWS structure, that is the following (work in progress):

	GWS		
		.antenna	antenna substructure
		.antenna.type	‘virgo’, ‘ligo’, ‘explorer’,…
		.antenna.par(k)	antenna parameter substructure
		.antenna.par.name	
		.antenna.par.val1	
		.antenna.par.val2	
		.antenna.par.p12	
		.antenna.par.p21	
		.antenna.par.tau	
		.antenna.par.w	
		.burstnoise	burst noise structure
		.chirp	chirp signal structure
		.pulse	pulse signal structure
		.perw	periodic wave structure
		.stoch	stochastic wave structure
		.ds.len	ds chunk length
		.ds.type	ds type