**SnagPy User Guide**

**Rome Virgo Group**



[Introduction 4](#_Toc128066184)

[**Basic classes** 5](#_Toc128066185)

[**GD** 5](#_Toc128066186)

[**GD2** 6](#_Toc128066187)

[**DS** 7](#_Toc128066188)

[**DS operation** 8](#_Toc128066189)

[**MGD** 11](#_Toc128066190)

[**Themes** 12](#_Toc128066191)

[**Symbols** 12](#_Toc128066192)

[**Plotting** 13](#_Toc128066193)

[**Mapping** 16](#_Toc128066194)

[**Data files** 17](#_Toc128066195)

[**Particular structures** 20](#_Toc128066196)

[**Support to Matlab-type structures** 22](#_Toc128066197)

[**From a gd to a “decorated” bsd: the role of the cont structure** 23](#_Toc128066198)

[**Astro&Time** 25](#_Toc128066199)

[**Time** 25](#_Toc128066200)

[**Position on the Earth** 28](#_Toc128066201)

[**Astronomical coordinates** 29](#_Toc128066202)

[**Other** 30](#_Toc128066203)

[**Antennas and sources** 32](#_Toc128066204)

[**Array table** 33](#_Toc128066205)

[**Some procedures** 34](#_Toc128066206)

[**CW\_histogram** 34](#_Toc128066207)

[**gd\_pows & gd\_welch** 36](#_Toc128066208)

[**gd\_spectrogram** 37](#_Toc128066209)

[**gd\_period** 39](#_Toc128066210)

[**gd\_tperiod** 40](#_Toc128066211)

[**SnagPy modules** 41](#_Toc128066212)

[**External packages** 43](#_Toc128066213)

[**Reference** 45](#_Toc128066214)

[**SnagPy modules (synthetic)** 45](#_Toc128066215)

# Introduction

SnagPy is a Python package for data analysis mainly in the gravitational signal field. It is a quasi-porting of the Snag Matlab toolbox developed by the Virgo Rome Data Analysis group.

To start:

* Put the files contained in the zip where you want, maintaining the folder structure.
* Set the PYTHONPATH in the environment variables (how to do this depends on the operative system. See for example [How to set python path (net-informations.com)](http://net-informations.com/python/intro/path.htm) or [Using PYTHONPATH — Functional MRI methods (bic-berkeley.github.io)](https://bic-berkeley.github.io/psych-214-fall-2016/using_pythonpath.html) , or anything similar googling ‘’set path python”).
* Use another folder for your python work.
* Set the environment variables SNAGPY\_PATH, ANT\_TAB, SOUR\_TAB and HINJ\_TAB to the relative files.

External packages to install:

* **NumPy**
* **SciPy**
* **Matplotlib**
* **Astropy, jplephem, Skyfield**
* **Mat73** for reading mat v7.3 files
* **h5py** for supporting HDF5 format
* **silx**

# **SnagPy modules**

SnagPy is composed by various modules, divided in main modules and secondary modules.

The main modules are in the folder SnagPy, the secondary modules are in the folders Exper, containing experimental modules, Pers, containing the personal or personally modified modules, and Examples, containing analysis examples using SnagPy.

In this guide only the main modules are described, and some examples are reported.

The main modules are:

* GD the management of the basic container class for 1-D data
* GD2 for the 2-D data GD
* DS data stream management
* MGD multiple GD management
* BASIC general management routines
* SERV service programming routines
* ML\_PY for using Matlab objects
* STAT statistics
* SIGNAL signal processing routines
* IMAGE image processing
* ASTROTIME time and astronomy routines
* GWDATA gravitational wave data (sources, antennas, periods…)
* PSS periodic source search
* BSD BSD analysis
* GWOTH other gw analysis
* GUISNAG guis for SnagPy
* EXT\_PACK module using critical external packages
* MAN\_SUPER management and supervision
* PARGPU parallel and GPU computing
* WEB\_SNAG routines for the web and phone and tablets apps
* DEEPSNAG deep learning for SnagPy
* FANCY\_FIG fancy figures and other

The secondary modules are in the folders:

* Projects containing projects related modules
* Exper containing experimental and test modules
* Pers containing personal modules
* Examples containing example scripts

Other folders:

* GWdata basic GW parameters (antennas, sources, Doppler,…)
* doc docs and guides
* docs Sphinx documentation

The file starting.py contains some helpful initial settings.

## **External packages**

One of the basic principle of SnagPy is the use of the minimal necessary external packages.

Here are the package that are considered necessary:

* **NumPy**
* **SciPy**
* **Matplotlib**
* **Astropy,jplephem,Skyfield**
* **Mat73** for reading mat v7.3 files
* **h5py** for supporting HDF5 format
* **silx**

No more used:

* hdfdict for supporting dictionaries in HDF5 format
* openpyxl for supporting Excel and tables

To be tested:

* PyQt5
* Seaborn
* Pillow
* Sympy
* Skyfield
* Cartopy
* ligo.skymap
* Numba
* CUDA Python
* Multiprocessing
* Flask
* Django

# **Basic classes**

## **GD**

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) |
| **typ** | = 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 any complex, for particular uses) |

## **GD2**

**A gd2 is a gd with a two dimension abscissa. It can be used for bi-dimensional data, as, for example, a time frequency spectrum. A gd2 can be “virtual” or “real” abscissa, but only for the “primary” dimension; the secondary dimension is always “virtual”.**

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 |
| **typ** | = 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 |
| **capt** | caption |
| **cont** | control variable - normally absent (0); may be an array or cell array, for particular uses) |

## **DS**

**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
* **fr2ds**, that accesses data in **frame** format
* **h52ds,** that accesses data in **HDF5** format
* **bsd2ds, that access data from bsds**
* **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 necessity of interlaced operations is due to how frequency domain filtering is performed.

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).

Typically 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.

## **MGD**

The MGD, multiple GD class, is introduced to contain multi-channel sampled data.

The attributes 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** | **…** |

## **Intervals**

Intervals is used to manage part of GD, GD2 or 1-D or 2-D arrays that should be selected or neglected. It operates with little differences with 1-D and 2-D values. This class is defined in the module SERV.

These are the attributes:

|  |  |
| --- | --- |
| **lar** | array length (number of columns) |
| **nar** | second dimension of the array (def=1) (n. of rows) |
| **ini** | intervals beginning |
| **fin** | intervals end (excluded value) |
| **typ** | 1 or 2 as GDs |
| **xini** | initial “physical”abscissa value (def=0) |
| **dx** | “physical” abscissa step (def=1) |
| **x** | “physical” abscissa (in the case of typ=2) |
| **label** | e.g. 'hole', 'data', 'good',...; default '' |
| **cover** | total coverage (number of 1s, per row) |

and these are the methods:

|  |  |
| --- | --- |
| **set\_data** | sets the label to “data” |
| **set\_hole** | sets the label to “hole” |
| **invert** | invert the interval |
| **check** | checks the correctness of the interval |
| **x\_** | extracts the abscissa |
| **show** | shows the interval object |
| **mask** | creates the mask from the interval |

There are many basic functions that manage intervals:

* **cover\_calc** that computes the coverage value or values
* **mask\_interv** that computes the intervals from a mask
* **interv\_mask** that creates a mask starting from the intervals
* **interv2dict** transforms an intervals object to a dictionary
* **dict2interv** transforms a dictionary to an interval
* **x\_interv** gets the abscissa
* **show\_interv** shows the intervals attributes
* **check\_interv** checks the syntactic correctness of an interval
* **invert\_interv** inverts an intervals object
* **interv\_and**  does the AND of a number of intervals
* **interv\_or** does the OR of a number of intervals
* **interv\_coverage** given more the one intervals objects, for each point of the mask

gives the number of different intervals

* **sel\_interv** select the data inside the intervals
* **win\_interv** computes windowing for a given mask
* **findnodata** f inds holes in data
* **FND2interv** from findnodata to hole interv
* **show\_nodata** used with findnodata

# **Themes**

## **Symbols**

To simplify the SnagPy work, is fundamental to define some symbols to reach folders or files.

The basic symbol is the path to the SnagPy main folder. It is defined by the user as an environment variable and should be defined as SNAGPY\_PATH, for example as D:\OneDrive\SF\\_Prog\Python\SnagPy . To verify you can give the command BASIC.envir\_var('SNAGPY\_PATH') .

Many symbols are created, starting from SNAGPY\_PATH, by the function GWDATA.set\_symbols(). These and other useful settings can be applied by copying and pasting, possibly at the beginning of work, the content of the file starting.py:

# copy and paste helper snippets

# initial importing ------------------

import numpy as np

import GD,BASIC,ML\_PY,SERV,STAT,SIGNAL,ASTROTIME,GD2,GWDATA,BSD,GUISNAG

import importlib

# importlib.reload(Module)

# useful services -------------

from BASIC import tic,toc

tic()

# symbols -----------------

snagpy\_p,gwdata\_p,examples\_p,exper\_p,pers\_p,projects\_p,\

antennas\_p,cwinj\_o2\_p,cwinj\_o3\_p,cwsour\_p,\

table\_ligoh\_p,table\_ligol\_p,table\_virgo\_p,table\_kagra\_p,de440s\_p\

    =GWDATA.set\_symbols()

table\_par=[1261872018.0, 1577490618.0, 600.0]   # start, stop in gps, step in s

# Rectangular equatorial: posx,posy,posz in light seconds , velx/C vely/C velz/C deinstein

# Time is: TDT given as mjd and gpstime [s]. Leap seconds: 37 last added January 2017

virgo,ligol,ligoh,kagra=GWDATA.antennas(antennas\_p)

## **Plotting**

The main procedure to plot gds and mono-dimensional arrays is based on the use of, normally, four functions:

1) newfig that can define the dimension of the window. If the

parameter siz is a two element list, it defines the relative

enhancement of the horizontal and vertical dimension of the

window, if it is just a number it applies to both h and v.

Example: newfig([1.33,1])

2) plot\_helper that defines various aspects of the graph, e.g.:

mode : normal plot, steps, scatter plot

scale : linear or logatithmic scale for each axis.

grid : grid lines

absc    'norm','min','hour','days','week','mjd','date'

fmt : format (color and texture)

linewid: line width

It creates a dictionary called P\_H.

3) plot\_gd general plotting function; the imput are:

ingd the input gd or array

P\_H the output of plot\_helper; if absent, the default

it is possible to give a new abscissa

4) post\_plot defines title and labels.

The plot can be modified by the functions xlog, ylog, xlin, ylin,

xlim, ylim and others.

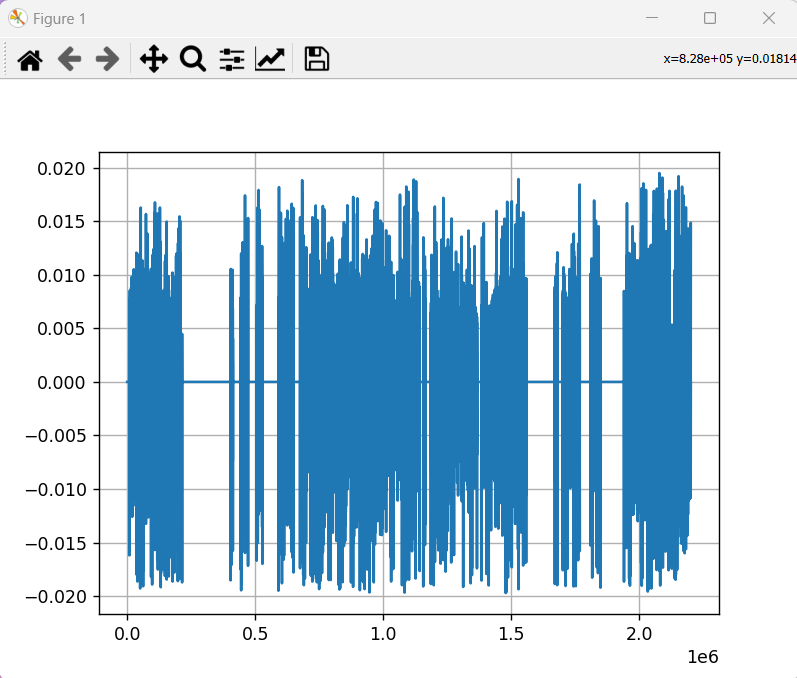
Example:

>>> l\_bsd=ML\_PY.gd\_lm7('L\_C02\_20170104\_0060\_0070\_O2\_tfstr.mat')

>>> GD.newfig()

>>> GD.plot\_gd(l\_bsd)

and we have:



Then

>>> s=STAT.gd\_pows(l\_bsd,npiece=10,res=2)

>>> GD.newfig([1.33,1])

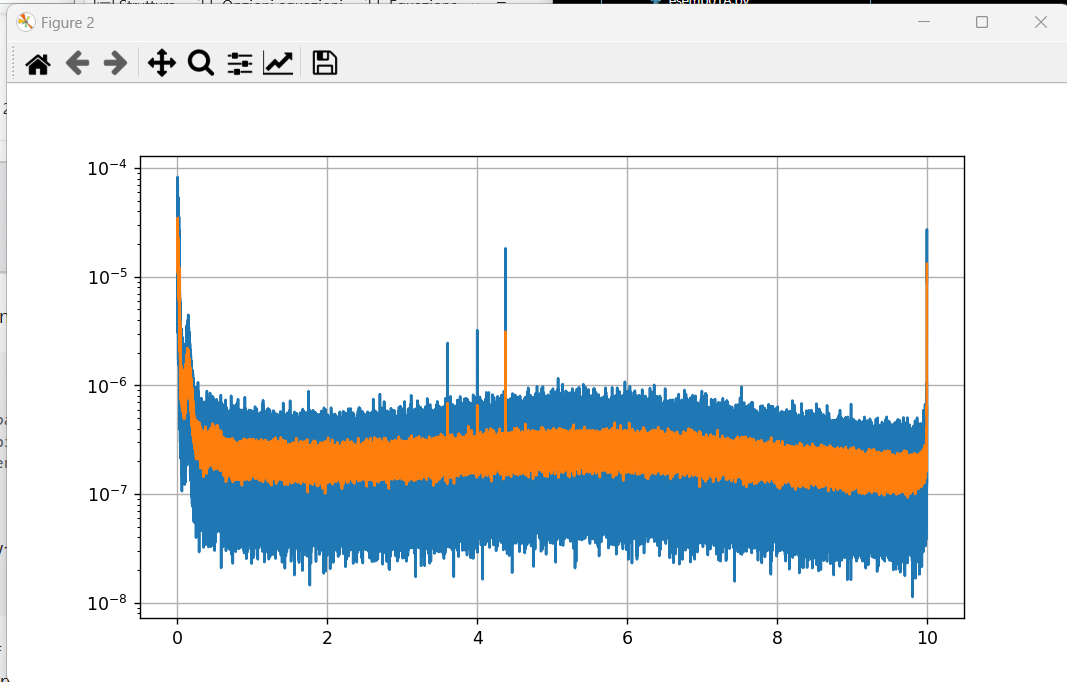
>>> P\_H=GD.plot\_helper(scale='lilo')

>>> GD.plot\_gd(s,P\_H)

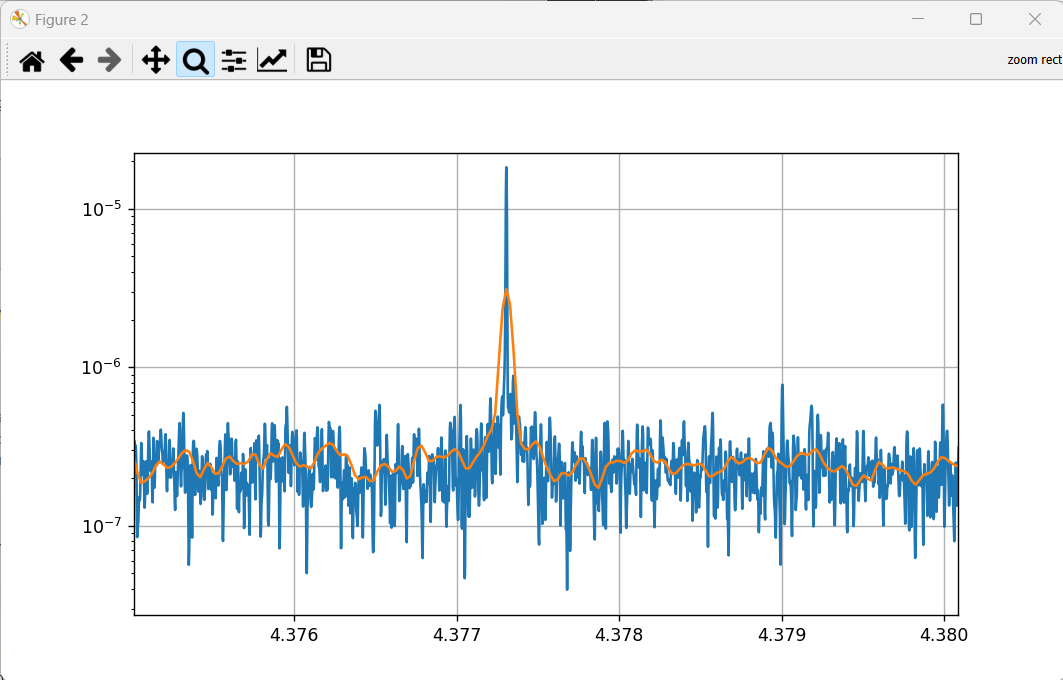
>>> s=STAT.gd\_pows(l\_bsd,npiece=100,res=2)

>>> GD.plot\_gd(s,P\_H)

obtaining:



and zooming:



## **Mapping**

This is the main procedure to map a gd2 (or a 2-D array).

The simplest way is just give the command

>>> GD2.grey\_map(ingd2)

A more rich procedure is:

>>> MH= GD2.map\_helper(cmap='cool',norm='linear',alpha=1,gridlin=0.5,gridstyl='--',gridcol='y')

>>> GD2.grey\_map(ingd2,MH, fun='none')

>>> GD2. post\_map(tit,xlab,ylab)

where all the parameters can be changed.

## **Data files**

A central problem in the SnagPy project is the exchange of data with Matlab and with other groups. We have chosen three formats: the two Matlab v7 and v7.3 mat file format and the HDF5.

Typical data produces by Snag (and SnagPy) are gds, bsds and (sometime complex) structures.

* **Matlab v7 format** (functions in ML\_PY)

This format is fundamental for reading data created by Matlab. This can be accomplished by loadmat7(fil) that is based on SciPy. It creates a dictionary that contains the variables. When a variable is extracted, we obtain a numpy.ndarray, that can be not easy to manage if the variable is a complex object.

In the case of a gd in the file we can use gd\_lm7(fil) . If, as is in the case pf the bsds, the cont attributes is a complex structure, that appears in the form of a very complex numpy.ndarray, we must use other functions (in BASIC). They are expl\_array(arr,Names=[],Cval=[],StName=[]), that is a recursive function, to explore all the substructures of the structrure, show\_array\_struct(Names,StName), that shows all the structures and variables, and array\_extract(kstr,var,Names,Cval,outdic=0), that extracts a given variable.

As an example, let us see the cont structure of a bsd (the gds and the gd2s are considered as structures)

>>> g=ML\_PY.gd\_lm7('L\_C02\_20170104\_0060\_0070\_O2\_tfstr.mat')

>>> cont=g.cont

>>> Names,Cval,StName= BASIC.expl\_array(cont)

>>> BASIC.show\_array\_struct(Names,StName)

obtaining

Immagine che contiene testo, nero, vicino, argento

Descrizione generata automaticamente

where in the first column is the order numbers of the 10 structures found, in the second column are the names of the structure (the first structure, cont, is reported as “start”) than the tuples contains the names of all the variables. Then we can read any variables of any structure giving the order number of the structure and name of the variable to the function array\_extract(kstr,var,Names,Cval,outdic=0). Choosing 8 and ‘persist’ we have

>>> aa=BASIC.array\_extract(9,'persist',Names,Cval)

>>> aa

array([[0.03911935, 0.05279117, 0.06647439, ..., 0.20507735, 0.16796811, 0.00053578]])

that can be squeezed by

>>> aa.squeeze()

array([0.03911935, 0.05279117, 0.06647439, ..., 0.20507735, 0.16796811, 0.00053578])

* **Matlab v7.3 format** (functions in ML\_PY)

The Matlab format v7.3 is, in practice, an implementation of the HDF5. This is accomplished by loadmat73(fil), based on the package mat73 (that doesn’t implement the save). This procedure creates a dictionary, with all the variable names and values; if there are sub-structures, these become sub-dictionaries, items of the main dictionary.

The function gd\_lm73 reads a gd data and creates a gd with them. If the cont attribute of the gd is a structure (as is in the case of the bsds), this becomes a sub-dictionary of the main dictionary.

Some functions can be used to manage these (or any other) dictionaries:

* + expl\_dict(dic,Keys=[],Dics=[]) that can explore the structure of the dictionary. This function is similar to expl\_array.
  + show\_dict\_struct(Keys,Dics) that shows the structure. This function is similar to show\_array\_struct.
  + dict\_extract\_2(dic,listkeys) that can extract the value associated to a key
* **HDF5** (functions in BASIC)

This format is very similar to the Matlab v7.3 (just a little simpler) and can be used to load and save data.

As an example, here is how we converted the jpl data table in HDF5, and how we can use them.

Here we convert the data matrix to array\_table and then to dictionary

def conv\_jpl\_data(datin,noline,nr,items):

# converts text data (typically table\_xxxx.dat) to hdf5 format

#   datin    complete path

#   noline   number of lenes to jump (typ. 4)

#   nr       number of output rows   (typ. 526032)

#   items    column to output (e.g. [0,1,3,7], typ. [1,2,3,4,5,6,7,8])

    sep=os.sep

    path,fil,ext=BASIC.path\_fil\_ext(datin)

    arr=BASIC.text2array(path+sep+fil+ext,noline,nr,items)

    at=BASIC.array\_table(['gpst','x','y','z','vx','vy','vz','einst'],arr)

    dic=BASIC.array\_table\_to\_dict(at,fil+'.hdf5')

    return dic

and then the dictionary is written in an HDF5 file:

def write\_hdf5\_s(dic,fil):

# writes in a hdf5 file a simple dictionary

#  dats   simple dictionary containing the data

or by:

def write\_dic2hdf5(dic,fil):

# writes a dictionary in a hdf5 file

Now let us see how to read the JPL tables in HDF5 format:

def ant\_pos\_vel(table\_p,table\_par,tin,tfi,au=0):

# position and velocity of an antenna + Einstein effect

# table\_p    symbol (e.g.: table\_virgo\_p)

# table\_par  table parameters symbol (e.g.: table\_par)

# tin,tfi    vectorial time ini,fin (e.g. [2021,1,20,0])

# au = 0   pos is in light seconds, vel in fraction of c

# au = 1   pos is converted to au

# au = 2   pos is converted to km

## **Particular structures**

* **Simple dictionaries**

The simp\_dict structure is a small table that can be viewed as a dictionary that hasn’t a dictionary as element and the list or tuples in its elements are not too long, so it can be easily viewed. The simp\_dicts are introduced to contain a small table to describe e.g. the main parameters of a function or of a measurement.

It can be created as a file like

Antenna     =  "Virgo"

Long        =  12.5

Lat         =  42

Azimut      =  30

Altro       =  [1, 23, 4.5]

CC          =  (11+4.2j)

and can be transformed in a dictionary by the function simp2dict(file) and showed and possibly saved in a file by show\_simp(sdic,spac=10,file=0). The simp\_dict functions are in the BASIC module.

* **Snag\_Table**

A snag\_table is similar to a Matlab table and it is implemented using a NumPy array structure.

It is implemented as a list (primary) of lists (secondary). The secondary lists are the rows, all equally composed, except for the first one that contains the names of the columns.

The snag\_table is a class (defined in the BASIC module) and has the following fields:

* + data (all the data, with also the titles)
  + nr the number of rows
  + nc the number of columns
  + titles the titles of the columns
  + capt a caption
  + cont a control variable (for future uses)

There are a few service function:

* + extr\_st\_col(st,which) to extract a column from the table
  + extr\_st\_rows(st,which) to extract a set of rows
  + csv2list(fil) to read a table in csv format
* **Array\_Table**

Similar to the snag\_table, but for much larger, essentially numeric tables.

* **Intervals**

To manage time or other intervals.

## **Support to Matlab-type structures**

In Python there are not the structures like those in Matlab or in C. There are three alternatives:

* dictionaries
* NumPy structured arrays
* classes possibly without methods (except those of the type class)

The last alternative is less interesting, because it is not general, but it is necessary to build a very peculiar class for any use case.

In the BASIC module there are functions to support the first and second alternatives.

These are used to deal with the files coming from Matlab.

## **From a gd to a “decorated” bsd: the role of the cont structure**

A gd has the following principal attributes:

> y the ordinate (basic data)

> n the length

> ini initial abscissa (used in type 1 gds)

> dx sampling step (used in type 1 gds)

> x abscissas (used in type 2 gds)

> typ determine type 1 (virtual abscissa) or type 2 (real abscissa)

> capt caption (a string)

> cont a control variable (in the case of a bsd, a special structure)

If the gd contains sample data, typically the gd is type 1 and dx is in seconds. If we want to assign a date (or epoch) to the data, we can put ini=0 and assign a date as a value t0 or T0 (typically t0 if the date is in mjd and T0 if the date is given in TAI or gps time) as a field of the cont structure. Such a gd is called a “dated gd”.

In the case of a bsd (that is a dated gd), we have a more complex cont structure. Here is an example

t0: 57757.15592592592

inifr: 60

bandw: 10

v\_eq: [4301×4 double]

p\_eq: [4301×4 double]

Tfft: 1024

ant: 'ligol'

run: 'O2'

cal: 'C02\_2048'

clean\_fraction: 5.3483e-04

tcreation: '05-Apr-2019 12:57:09'

durcreation: 64.7034

mi\_abs\_lev: 0.0020

To this basic structure we add a field containing another more complex structure, tfstr, that we call a “decoration”, containing a basic analysis of the bsd. The tfstr contains a certain number of nested sub-structures.

**cont**

**tfstr**

**subsp pt bsd\_par clean**

**typ gd2 sky** .

>> BASIC.show\_array\_struct(Names,StName)

0 start ('t0', 'inifr', 'bandw', 'v\_eq', 'p\_eq', 'Tfft', 'ant', 'run', 'cal', 'clean\_fraction', 'tcreation', 'durcreation', 'mi\_abs\_lev', 'tfstr')

1 tfstr ('peaktype', 'ant', 't0', 'dt', 'inifr', 'bandw', 'lfft', 'Nfft', 'dfr', 'DT', 'gdlen', 'subsp', 'pt', 'bsd\_par', 'clean', 'zeros')

2 subsp ('Nsstim', 'len\_ss', 'ssred1', 'init', 'typ', 'hdens', 'wn')

3 typ ('sstype', 'ssred', 'minno0', 'maxwn')

4 hdens ('x', 'y', 'n', 'type', 'ini', 'dx', 'capt', 'cont', 'm', 'ini2', 'dx2', 'mcapt', 'va', 'vd', 've', 'gd')

5 gd ('x', 'y', 'n', 'type', 'ini', 'dx', 'capt', 'cont', 'unc', 'uncx')

6 pt ('thr', 'tpeaks', 'npeaks', 'peaks', 'ntotpeaks', 'ok', 'index')

7 bsd\_par ('enh', 'frmax', 'tfft0', 'lfft0', 'tfft', 'lfft', 'dfr0', 'Ndop', 'Nfr', 'Ndb', 'dsd0', 'Nsky', 'sky')

8 sky ('x', 'b', 'lam', 'bet', 'ulam', 'ubet', 'index', 'nlon', 'nskypoint', 'nbeta', 'x2', 'maxbeta', 'minbeta')

9 clean ('nt', 'nfr', 'NF', 'NT', 'DF', 'DT', 'PHT', 'mpers', 'persist', 'tfhist', 'htfhist', 'xhtfhist', 'perscutfr', 'tfcut', 'tfhist0', 'htfhist0', 'xhtfhist0', 'persist0')

- - - - - - - - - - - - - - - - - - - -

start ('t0', 'inifr', 'bandw', 'v\_eq', 'p\_eq', 'Tfft', 'ant', 'run', 'cal', 'clean\_fraction', 'tcreation', 'durcreation', 'mi\_abs\_lev', 'tfstr')

['tfstr'] ('peaktype', 'ant', 't0', 'dt', 'inifr', 'bandw', 'lfft', 'Nfft', 'dfr', 'DT', 'gdlen', 'subsp', 'pt', 'bsd\_par', 'clean', 'zeros')

['tfstr', 'subsp'] ('Nsstim', 'len\_ss', 'ssred1', 'init', 'typ', 'hdens', 'wn')

['tfstr', 'subsp', 'typ'] ('sstype', 'ssred', 'minno0', 'maxwn')

['tfstr', 'subsp', 'hdens'] ('x', 'y', 'n', 'type', 'ini', 'dx', 'capt', 'cont', 'm', 'ini2', 'dx2', 'mcapt', 'va', 'vd', 've', 'gd')

['tfstr', 'clean'] ('nt', 'nfr', 'NF', 'NT', 'DF', 'DT', 'PHT', 'mpers', 'persist', 'tfhist', 'htfhist', 'xhtfhist', 'perscutfr', 'tfcut', 'tfhist0', 'htfhist0', 'xhtfhist0', 'persist0')

[['tfstr'], ['tfstr', 'subsp'], ['tfstr', 'subsp', 'typ'], ['tfstr', 'subsp', 'hdens'], ['tfstr', 'subsp', 'hdens', 'gd'], ['tfstr', 'pt'], ['tfstr', 'bsd\_par'], ['tfstr', 'bsd\_par', 'sky'], ['tfstr', 'clean']])

## **Astro&Time**

Some of these functions uses Astropy and Skyfield.

### **Time**

In SnagPy some different times are used. They can be divided in continuous and leap times. Normally we use:

* Continuous time:
  + TAI
  + GPS time
  + TT
  + TDB
* Leap time:
  + UTC
  + local time
  + JD
  + MJD

Some of these times can be expressed in various ways (vectorial, strings, floating numbers).

The so called “unix time” is not used for scientifical purposes and normally is an UTC expressed as the number of (conventional) seconds since 1-Jan=1970 00:00:00. Sometimes SI seconds are considered.

A basic time function is:

def now(vt=0,typ='UTC',form='string'):

# Time now

#   vt      vect time, if absent, present time

#           vt=(year, month [1-12], day [1-31], hour, min, s, 0,0,0)

#   typ    'UTC' or 'local'

#   form   'string', 'struct', 'mjd', 'gps', 'unix'

It uses the Python class time.struct\_time.

Another function, using Astropy, is:

def set\_time(times,form=0):

# set time values as Time objects

#  times   array or list

#  form    see https://docs.astropy.org/en/stable/time/index.html#time-format

#          if absent it can be interpreted

#

# The output t are time object, that can be easily converted

# example:

#  >>> t = Time('2023-01-01')

#  >>> t

#  <Time object: scale='utc' format='iso' value=2023-01-01 00:00:00.000>

#  >>> t.mjd

#  59945.0

#  >>> t.gps

#  1356566418.0

It creates Astropy Time objects, that ca be transformed easily to other time forms.

Time conversion can be obtained also with

def t\_conv(tin,fmtin,fmtout):

# time conversion

# tin            input time

# fmtin, fmtout  'str', 'vec', 'mjd', 'gps'

#  only fmtout   'obj' time object

#                'ut1', 'tt', 'tai',tcb,tcg,tdb

#      vec as (2000, 1, 1, 0, 0)

#      str as '2000-01-20 00:00:00.000'

Sidereal time can be obtained by:

def sid\_time(tim,loc=0):

# local or Greenwich (default) sidereal time

#

#  tim   time object or as set\_time

#  loc   Earth locality object or [lon,lat] (in decimal deg)

or by

def gmst(t,long=0):

# Greenwich mean sidereal time or local ST

#  t     time object or jd or mjd

#  long  local longitude in deg or EL object (for local ST)

Using Skyfield, we can use:

def sf\_arrtime(tini,tfin,step):

# Skyfield time array

#  tini    initial time tuple (a,m,d,h,m,s)

#  fion    final time tuple

#  step    in s

There are two useful functions, defined in BASIC, similar to the homonymous Matlab service, to measure the system time: tic() and toc(). tic() start the timer, toc() shows the time elapsed. various tocs can be used. A tic is authomatically called inside starting.py .

### **Position on the Earth**

def EarLoc(lon,lat,h=0):

# Earth location object

#  lon,lat  angles

#  h        height in m

#

# Angles can be expressed as:

#

# Angle('10.2345d')               String with 'd' abbreviation for degrees

# Angle('1:2:30.43 degrees')      Sexagesimal degrees

# Angle('1°2′3″')                 Unicode degree, arcmin and arcsec symbols

# Angle('1°2′3″N')                Unicode degree, arcmin, arcsec symbols and direction

# Angle('1d2m3.4s')               Degree, arcmin, arcsec

# if it is expressed as a decimal number xxxx.xx, is equivalent to 'xxxx.xxd'

creates an object of the class astropy.coordinates.earth.EarthLocation .

def sites():

# sites for gravitational antennas

gives the Earth location objects for the gravitational antennas.

### **Astronomical coordinates**

def astro\_coord(cin, cout, ai, di):

# ASTRO\_COORD   astronomical coordinate conversion, from cin to cout

#

#  cin and cout can be

#

#    'equ'      celestial equatorial: right ascension, declination

#    'ecl'      ecliptical: longitude, latitude

#    'gal'      galactic longitude, latitude

#

#    ai,di      input coordinates  (deg)

#    ao,do      output coordinates (deg)

Using Astropy, we can use:

def astropy\_coord(cin, cout, ai, di, lon=[], lat=[], time=[]):

# Astronomical coordinate system change

#  Angles are in degrees. Local tsid (in hours) and latitude is needed

#  for conversions to and from the horizon coordinates.

#

#  cin and out can be

#

#    'hor'         Horizontal coordinate: azimuth, altitude (az,alt)

#                      YOU SHOULD ADD lat and lon!

#    'equ'         celestial equatorial: right ascension, declination (ra,dec)

#    'ecl'         ecliptical: longitude, latitude (lon,lat)

#    'gal'         galactic longitude, latitude (l,b)

#

#    ai,di         input coordinates

#    lon,lat,time  local sidereal time and latitude (only for horizontal coordinates)

#                  various form for time see https://docs.astropy.org/en/stable/time/index.html

#

# Angles can be expressed as:

#

# Angle('10.2345d')               String with 'd' abbreviation for degrees

# Angle('1:2:30.43 degrees')      Sexagesimal degrees

# Angle('1°2′3″')                 Unicode degree, arcmin and arcsec symbols

# Angle('1°2′3″N')                Unicode degree, arcmin, arcsec symbols and direction

# Angle('1d2m3.4s')               Degree, arcmin, arcsec

# if it is expressed as a decimal number xxxx.xx, is equivalent to 'xxxx.xxd'

### **Other**

Astropy constants:

def const\_table(file=0):

# file, if it is present, save the constants table in a file

    lis0=['G', 'GM\_earth', 'GM\_jup', 'GM\_sun', 'L\_bol0', 'L\_sun', 'M\_earth', 'M\_jup', 'M\_sun',

     'N\_A', 'R', 'R\_earth', 'R\_jup','R\_sun', 'Ryd', 'a0', 'alpha', 'atm', 'au', 'b\_wien', 'c',

     'e', 'eps0', 'g0', 'h', 'hbar','k\_B', 'kpc', 'm\_e', 'm\_n', 'm\_p', 'mu0', 'muB',

     'pc','sigma\_T', 'sigma\_sb']

|  |  |  |  |
| --- | --- | --- | --- |
| **Symbol** | **What** | **Value** | **RelErr** |
| G | Gravitational constant | 6.674300E-11 | 2.25E-05 |
| GM\_earth | Nominal Earth mass parameter | 3.986004E+14 | 0 |
| GM\_jup | Nominal Jupiter mass parameter | 1.266865E+17 | 0 |
| GM\_sun | Nominal solar mass parameter | 1.327124E+20 | 0 |
| L\_bol0 | Luminosity for absolute bolometric magnitude 0 | 3.012800E+28 | 0 |
| L\_sun | Nominal solar luminosity | 3.828000E+26 | 0 |
| M\_earth | Earth mass | 5.972168E+24 | 2.25E-05 |
| M\_jup | Jupiter mass | 1.898125E+27 | 2.25E-05 |
| M\_sun | Solar mass | 1.988410E+30 | 2.25E-05 |
| N\_A | Avogadro's number | 6.022141E+23 | 0 |
| R | Gas constant | 8.314463E+00 | 0 |
| R\_earth | Nominal Earth equatorial radius | 6.378100E+06 | 0 |
| R\_jup | Nominal Jupiter equatorial radius | 7.149200E+07 | 0 |
| R\_sun | Nominal solar radius | 6.957000E+08 | 0 |
| Ryd | Rydberg constant | 1.097373E+07 | 1.91E-12 |
| a0 | Bohr radius | 5.291772E-11 | 1.51E-10 |
| alpha | Fine-structure constant | 7.297353E-03 | 1.51E-10 |
| atm | Standard atmosphere | 1.013250E+05 | 0 |
| au | Astronomical Unit | 1.495979E+11 | 0 |
| b\_wien | Wien wavelength displacement law constant | 2.897772E-03 | 0 |
| c | Speed of light in vacuum | 299792458.000000 | 0 |
| e | Electron charge | 1.602177E-19 | 0 |
| eps0 | Vacuum electric permittivity | 8.854188E-12 | 1.47E-10 |
| g0 | Standard acceleration of gravity | 9.806650E+00 | 0 |
| h | Planck constant | 6.626070E-34 | 0 |
| hbar | Reduced Planck constant | 1.054572E-34 | 0 |
| k\_B | Boltzmann constant | 1.380649E-23 | 0 |
| kpc | Kiloparsec | 3.085678E+19 | 0 |
| m\_e | Electron mass | 9.109384E-31 | 3.07E-10 |
| m\_n | Neutron mass | 1.674927E-27 | 5.67E-10 |
| m\_p | Proton mass | 1.672622E-27 | 3.05E-10 |
| mu0 | Vacuum magnetic permeability | 1.256637E-06 | 1.51E-10 |
| muB | Bohr magneton | 9.274010E-24 | 3.02E-10 |
| pc | Parsec | 3.085678E+16 | 0 |
| sigma\_T | Thomson scattering cross-section | 6.652459E-29 | 9.02E-10 |
| sigma\_sb | Stefan-Boltzmann constant | 5.670374E-08 | 0 |

## **Antennas and sources**

Antennas, sources and other data are managed by snag\_tables. For examples, for the antennas, we have a csv file that, open with Excel appears as

Immagine che contiene tavolo

Descrizione generata automaticamente

By the function

def anten(ant,anten\_tab=0):

# extract data for an antenna (a dictionary)

#  ant         antenna name ('virgo','ligol',...)

#  anten\_tab   antenna table (def possible if the environment variable is set)

we can extract the data for an antenna as a dictionary. anten\_tab is the name of the table, that can be omitted is it set an environment variable.

At the beginning of a work, we can call the function

def antennas(anten\_tab=0):

# creates dictionarys for each antenna

#  typical call: virgo,ligol,ligoh,kagra=GWDATA.antennas()

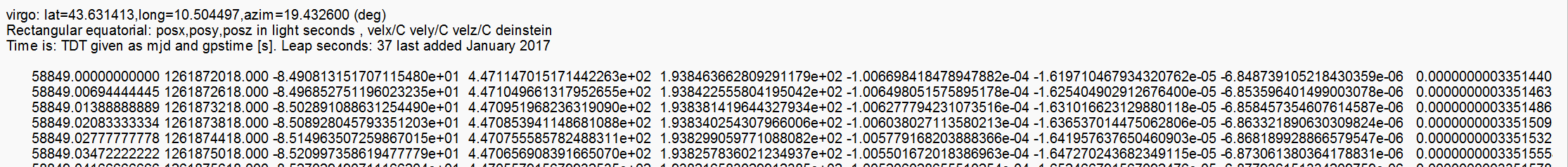
that creates all the dictionaries for all the antennas, as e.g.

>>> virgo

{'Antenna': 'virgo', 'Lat': 43.6314133, 'Long ': 10.5044968, 'Azim': 199.4326, 'Height': 4, 'Incl': 0, 'whour': 1, 'shour': 2, 'type': 2}

## **Array table**

If the data are huge and all numeric of the same kind, the snag array is not ideal, so we use a similar structure, but based on the NumPy arrays. Such data can be stored on disc using pickle or hfpy (and in this case it can be compressed). For example the Doppler table



with more than half a million of rows (with 9 columns) occupies about 112000 kB in text, that become 33000 in pickle and HDF5 (not compr.) and 26000, in HDF5 compressed. All the operations are very fast (~1 s).

# **Some procedures**

## **CW\_histogram**

This function uses linear filter procedure to produce histograms. We can use different moving average filters (namely triangular, gaussian, rectangular and any other user choice). The values can be “weighted”. The result is normally a better estimation of the probability density of the data especially in case of low numerosity.

Examples:

>>> y=np.random.randn(100)\*2

>>> out,dic,fun=STAT.CW\_histogram(y,typ='tri')

>>> out1,dic1,fun1=STAT.CW\_histogram(y,typ='gau')

>>> out2,dic2,fun2=STAT.CW\_histogram(y,typ='rec')

>>> stat,Hist=GD.stat\_gd(y,12)

>>> GD.newfig()

>>> GD.plot\_gd(out)

>>> GD.plot\_gd(out1)

>>> GD.plot\_gd(out2)

>>> GD.plot\_gd(Hist)

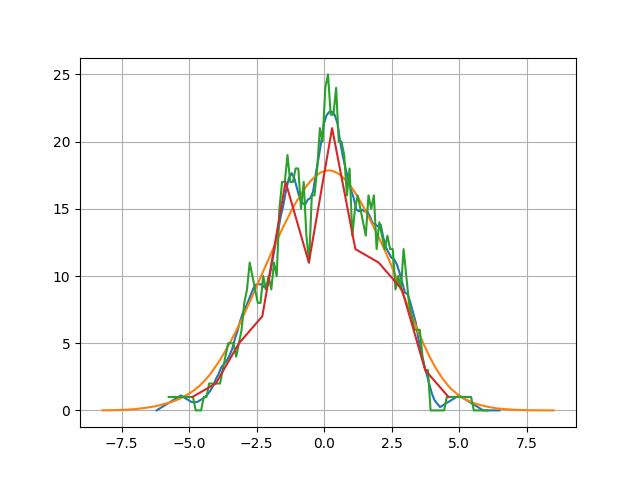


Table:

N = 100

xmean = 0.07272396010818402

xmedian = 0.13018008281292792

xstd = 1.9357823522046353

xskew = -0.13044953095050532

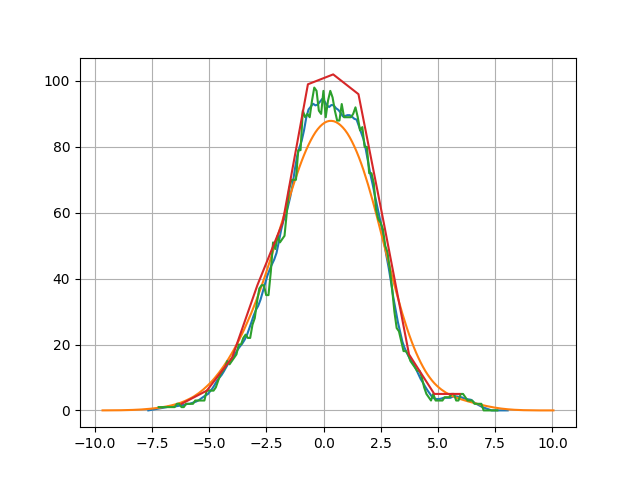
xkurt = -0.2646605027619664

type = Gaussian

mu\_hist = 0.06938821144710516

sig\_hist = 2.1710251925929915

With other numbers and parameters:



## **gd\_pows & gd\_welch**

>>> gg=GD.rand\_gd(10000,'norm')

>>> gg1=GD.set\_gd(10000,'sin',par2=0.1)

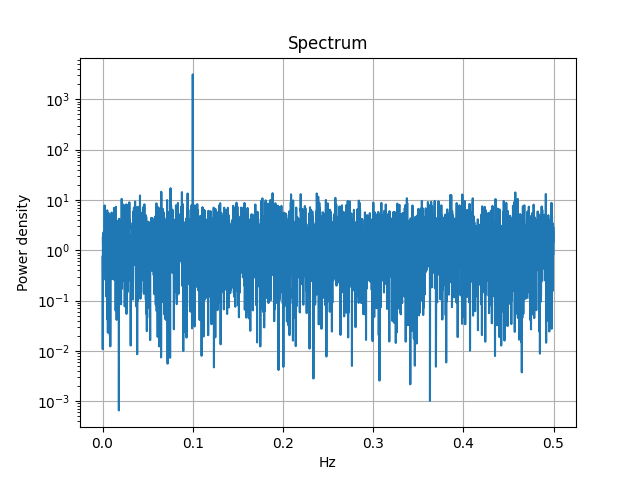
>>> gg2=gg+gg1

>>> S=STAT.gd\_pows(gg2)

>>> GD.newfig()

>>> PH=GD.plot\_helper(scale='lilo')

>>> GD.plot\_gd(S,P\_H=PH)



## **gd\_spectrogram**

>>> g=ML\_PY.gd\_lm7('L\_C02\_20170104\_0060\_0070\_O2\_tfstr.mat')

>>> spec,tt,ff=STAT.gd\_spectrogram(g,1000)

>>> GD2.show\_gd2(spec)

type 1

n 88060000

m 20000

n/m 4403.0

ini 500.0

dx 500.0

ini2 0

dx2 0.0005

y [[0.000000…

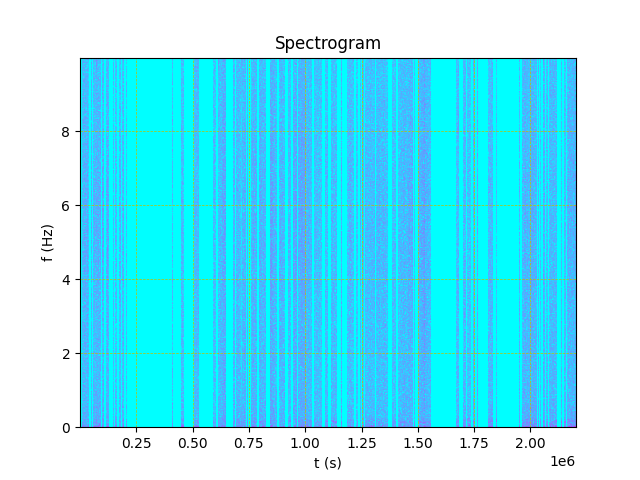
5.98135467e-05 4.28094276e-05]]

capt

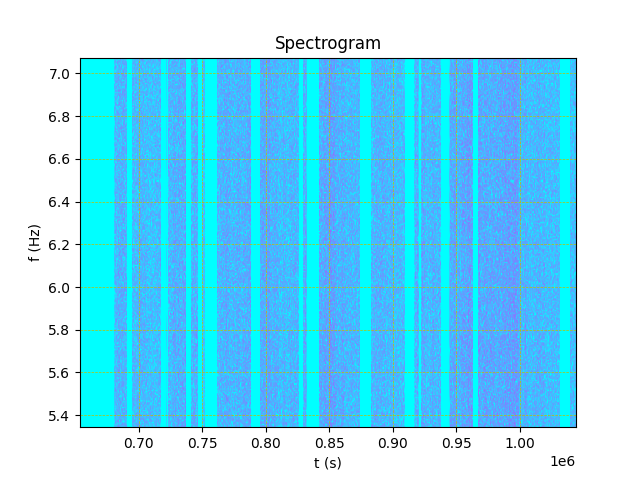
cont 0

SnagPy gd2 created on Thu Feb 23 17:23:12 2023

>>> GD2.grey\_map(spec,fun='log')



zoomando:



**gd\_period**

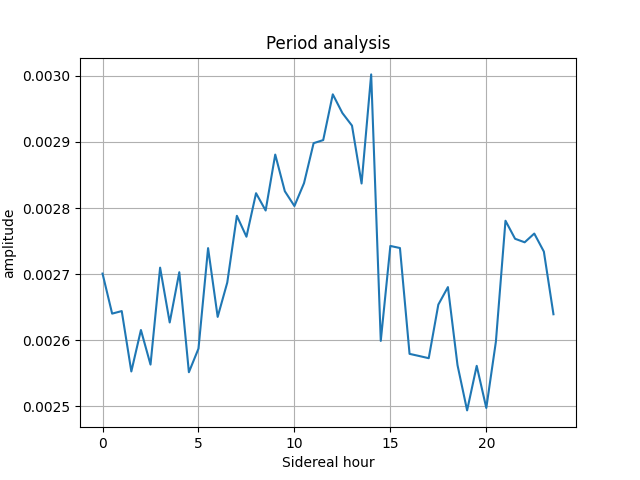
>>> g=ML\_PY.gd\_lm7('L\_C02\_20170104\_0060\_0070\_O2\_tfstr.mat')

>>> period,meanp,harm,perclean,win=STAT.gd\_period(g,'sid')

>>> GD.newfig()

>>> GD.plot\_gd(period)

>>> GD.post\_plot('Period analysis','Sidereal hour','amplitude')

****

**gd\_tperiod**

# **Reference**

## **SnagPy modules (synthetic)**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Module GD

Section class gd -----------------------------------------------

class gd

Functions: edit\_gd, x\_gd, div\_gd, zero\_nan\_gd,

Section gd - dictionary management ------------------------

Functions: dict2gd, gd2dict,

Section gd display -----------------------------

Functions: show\_gd,

Section set functions -----------------------------------------------

Functions: set\_gd, rand\_gd,

Section modification function -----------------------------------------------

Functions: modif\_gd, rota\_gd, resamp\_gd, parallel\_gd, stat\_gd, fft\_gd,

Section plot functions -----------------------------------------------

Functions: newfig, plot\_gd, plot\_helper, post\_plot, c\_plot\_gd, close\_fig, gridon, xlog, ylog, xlin, ylin, xlim, ylim, fig\_limits, ioff, ion, holdoff,

Section calc functions -------------------------------

Functions: minmax\_gd,

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Module GD2

Section class gd2 -----------------------------------------------

class gd2

Functions: edit\_gd2, x\_gd2, x2\_gd2, zero\_nan\_gd2,

Section gd - dictionary management ------------------------

Functions: dict2gd2, gd22dict,

Section gd2 display -----------------------------

Functions: show\_gd2,

Section modification function -----------------------------------------------

Functions: modif\_gd2, rota\_gd2, fft\_gd2, im\_cut, im\_reduce, gd2\_stat\_nz,

Section map functions -----------------------------------------------

Functions: newfig2, grey\_map, map\_helper, post\_map, scat\_gd2, post\_fig2,

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Module MGD

Section class gd -----------------------------------------------

class mgd

Functions:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Module BASIC

Section Small apps ---------------------------------------

Functions: envir\_var, var, Exec, isa, byte2str, tic, toc, num\_var\_odd, path\_fil\_ext, ind\_from\_inidx,

Section load & save dictionary: text, csv, json and pickle ---------------

Functions: dict2text, dict2csv, dict2json, dict2pkl, pkl2dict,

Section HDF5 -------------------------

Functions: list\_baskeys\_hdf5, write\_hdf5\_s, read\_hdf5\_s, write\_dic2hdf5, read\_hdf52dic, read\_hdf5\_part,

Section List ----------------------------

Functions: list2string, show\_list,

Section Dictionary --------------------------

Functions: show\_dict, expl\_dict, show\_dict\_struct, show\_dict\_struct\_2, dict\_extract\_2,

Section simple dict ----------------------

Functions: eq\_interpr, simp2dict, show\_simp,

Section numpy structures -------------------------------------

Functions: expl\_array, expl\_array\_2, show\_array\_struct\_2, show\_array\_struct, array\_extract, arrstruct2dict, val\_from\_key,

Section SnagTable --------------------------

class snag\_table

Functions: snag\_table\_show, extr\_st\_col, extr\_st\_rows, extr\_st\_row, extr\_st\_dict, csv2list, csv2st, st2csv, decode\_simp\_list,

Section ArrayTable --------------------------

class array\_table

Functions: text2array, array\_table\_to\_dict,

Section Intervals -------------------------

class intervals

Functions:

Section system -------------------------

Functions: func\_in\_module, list\_of\_func, deshape, ana\_module, all\_modules,

Section Graphic ------------------------

Functions: fig\_dim, inter\_grid, plot\_colortable, base\_colors, tab\_palette, css\_colors,

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Module SERV

Section ervice computational routines ---------------------------------

Functions: rota, thresh, atan3, mask, ifr, range\_shift, vec\_ccdot,

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Module ML\_PY

Section read - write v.7 format mat file -------------

Functions: loadmat7, gd\_lm7, gdsavedicmat7, gdloaddicmat7,

Section read - write v.7.3 format mat file ---------------------

Functions: loadmat73, gd\_lm73,

Section read - write csv --------------------------

Functions: csv\_read,

Section BSD --------------------------------

Functions:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Module STAT

Section Zero management ----------------------------

Functions: ana\_zero, stat\_nozero,

Section Histograms and Parameters ----------------------------

Functions: CW\_histogram, param\_from\_hist,

Section Power Spectra ----------------------------

Functions: gd\_pows, gd\_welch, cross\_pw, coher, lombscargle, pulse\_spectrum, stft,

Section Spectrograms ----------------------------

Functions: gd\_spectrogram,

Section Period analysis ------------------------

Functions: gd\_period, gd\_tperiod, gd\_worm,