sigpyproc Documentation

Release 1.0

Ewan Barr

November 20, 2012

CONTENTS

Contents:

CHAPTER

ONE

INTRODUCTION

1.1 What is sigpyproc?

sigpyproc is a pulsar data analysis library for Python. It provides an OOP approach to pulsar data handling through the use of objects representing pulsar data types (e.g. filterbank, time series, fourier series, etc.). As pulsar data processing is often time critical, speed is maintained through the use of compiled C libraries that are accessed via the python standard library ctypes module. Additional performance increases are obtained via the use of multi-threading with OpenMP, a threading library standard to most linux and mac systems.

1.2 Why sigpyproc?

sigpyproc was initially intended to be an Python wrapper for the SigProc pulsar signal processing toolbox, but over time it has developed and become an independent project in its own right. Unlike Sigproc and Presto, sigpyproc does not currently have full capabilities as a piece of pulsar searching software. Instead, sigpyproc provides data manipulation routines which are well suited to preprocessing and micromanagement of pulsar data. The structure of the package also makes it an ideal development environment, with a simple plug-and-play system with new modules and extensions.

Benefits of sigpyproc:

- Straight forward and easy to follow syntax
- C files are not swig wrapped and can be altered quickly and efficently
- Economy of code
- Structure of the code makes it easy to add support for new data formats
- Ideal for development, with step-by-step guides and reference documentation
- Opens up the vast power of Python's libraries (Numpy, Scipy, Matplotlib etc.)
- Similarily opens up the vast power of C's libraries (OpenMP, FFTW etc.)
- Allows for easy combination of the best bits of C and of Python (e.g. FFTW, Numpy, Scipy, OpenMP, Matplotlib)
- Python's garbage collection removes the need to explicity free buffers (eliminating memory leaks)
- Object oriented structure allows for a simple plug and play system with new modules, functions, methods or classes
- Can be used interactively in the Python or IPython shells
- C code segements are generally kept small and clean, only being used for base level data processing

- Very few dependencies

CHAPTER

TWO

INSTALLATION

2.1 Requirements

- numpy
- ctypes
- FFTW3
- OpenMP

2.2 Step-by-step guide

As both setuptools and distutils do not have any clear method of support for distributing C libraries for ctypes, the onus is on the user to ditribute the c libraries once built

1. Clone or download the git repository from https://github.com/ewanbarr/sigpyproc
2. Unzip and untar if needed and move to source directory
3. Make sure FFTW3 has been compiled with the `--enable-float` and `--enable-shared` options
4. run `sudo python setup.py install`
5. a lib/c and bin/ directories will be created in the source directory
6. Distribute the contents of these directories if required
7. If you are developing, then append the lib/c directory to the LD_LIBRARY_PATH environment variable

CHAPTER

THREE

TUTORIAL

This tutorial covers some of the basic functionality of the sigpyproc package. For a guide on how to extend the package, see the

3.1 Getting started

The first thing we will want to do is to open up the interactive python environment. Here I am using IPython 0.13, but any IPython release should be suitable. For test puproses a small 2-bit filterbank file is included in the `/examples/` directory of the sigpyproc package. The rest of this tutorial will be conducted from within the `/examples/` directory.

3.1.1 Loading data into sigpyproc

Lets start by loading our filterbank file into sigpyproc. To do this, we require the `FilReader` class from the `sigpyproc.Readers` module.

```
In [1]: from sigpyproc.Readers import FilReader

In [2]: myFil = FilReader("tutorial.fil")

In [3]: myFil
Out[3]: <sigpyproc.Readers.FilReader at 0x10a70f1d0>
```

`myFil` now contains an instance of the `sigpyproc.Readers.FilReader` class. We can access obervational meta-data through the `myFil.header` attribute:

```
In [4]: myFil.header

Out[4]:
{'bandwidth': 69.76,
'basename': 'tutorial',
'data_type': 1,
'dec_deg': 0.0,
'dec_rad': 0.0,
'dtype': '<u1',
'extension': '.fil',
'fbottom': 1440.785,
'fcenter': 1475.665,
'fch1': 1510.0,
'filelen': 3000564,
'filename': 'tutorial.fil',
'foff': -1.09,
```

```
'ftop': 1510.545,
'hdrlen': 244,
'machine_id': 0,
'nbits': 2,
'nbytes': 3000320,
'nchans': 64,
'nifs': 1,
'nsamples': 187520,
'obs_date': '09/10/1995',
'obs_time': '00:00:00.00000',
'ra_deg': 0.0,
'ra_rad': 0.0,
'source_name': 'P: 250.000000000000 ms, DM: 30.000',
'src_dej': 0,
'src_raj': 0,
'telescope_id': 0,
'tobs': 60.006400000000006,
'tsamp': 0.00032,
'tstart': 50000.0}
```

where

```
In [5]: type(myFil.header)
Out[5]: sigpyproc.Header.Header
```

All values stored in the `myFil.header` attribute may be accessed both as dictionary items and/or as attributes, i.e.:

```
In [6]: myFil.header["nchans"]
Out[6]: 64

In [7]: myFil.header.nchans
Out[7]: 64
```

Now that we know how to load a file into sigpyproc, let's look at at doing something with the loaded data.

3.1.2 Dedispersing the data

One of the most used techniques in pulsar processing is dedispersion, wherein we add or remove frequency dependent time delays to the data.

To dedisperse our `myFil` instance, we simply call the dedisperse method:

```
In [9]: myTim = myFil.dedisperse(30)

Filterbank reading plan:
------------------------
Called on file:       tutorial.fil
Called by:            dedisperse
Number of samps:      187520
Number of reads:      18
Nsamps per read:      10000
Nsamps of final read: 7826
Nsamps to skip back:  17

Execution time: 0.028745 seconds

In [10]: myTim
Out[10]: TimeSeries([ 108.,  100.,  102., ...,  105.,  111.,  107.], dtype=float32)
```

```
In [11]: type(myTim)
Out[11]: sigpyproc.TimeSeries.TimeSeries
```

Here we have dedispersed to a DM of 30 pc cm^-3 with the result being an instance of the `sigpyproc.TimeSeries.TimeSeries` class, which we have called `myTim`.

The `sigpyproc.TimeSeries.TimeSeries` class in a subclass of `numpy.ndarray`, and is capable of using all standard numpy functions. For example:

```
In [12]: myTim.sum()
Out[12]: TimeSeries(19636856.0, dtype=float32)

In [13]: myTim.max()
Out[13]: TimeSeries(121.0, dtype=float32)

In [14]: myTim.min()
Out[14]: TimeSeries(88.0, dtype=float32)

In [15]: np.median(myTim)
Out[15]: TimeSeries(105.0)
```

The use of `numpy.ndarray` subclasses is important in allowing sigpyproc to easily interface with many 3rd party python libraries.

3.1.3 Performing a Fourier transform

To perform a discrete fourier transform of the data contained in the myTim instance we may invoke the `myTim.rFFT` method.

```
In [22]: myFS = myTim.rFFT()

In [23]: type(myFS)
Out[23]: sigpyproc.FourierSeries.FourierSeries

In [24]: myFS
Out[24]:
FourierSeries([  1.96368840e+07,   0.00000000e+00,  -2.94284393e+02, ...,
                -1.77933350e+03,  -2.76700000e+03,   0.00000000e+00], dtype=float32)
```

The `sigpyproc.FourierSeries.FourierSeries` is also a subclass of `numpy.ndarray`, where array elements are [real,imaginary,real,imaginary,real,imaginary,real,...].

Using the `rednoise` method of `myFS`, we can de-redden the Fourier series:

```
In [31]: myFS_red = myFS.rednoise()

In [33]: myFS_red
Out[33]:
FourierSeries([ 1.        ,  0.        , -0.24660645, ..., -1.6417625 ,
               -2.55306649,  0.        ], dtype=float32)
```

with the dereddened fourier series, we can now form the power spectrum of the observation:

```
In [34]: mySpec = myFS_red.formSpec(interpolated=True)

In [36]: mySpec
Out[36]:
```

```
PowerSpectrum([ 1.        ,  0.91754919,  0.78776252, ...,  2.00740767,
                2.25372458,  2.55306649], dtype=float32)
```

Here we have set the `interpolated` flag to True, causing the `formSpec` function to perform nearest bin interpolation.

`mySpec` contains several convenience methods to help with navigating the power spectrum. For instance:

```
In [37]: mySpec.period2bin(0.25)
Out[37]: 240

In [38]: mySpec.freq2bin(5.0)
Out[38]: 300
```

We can also perofrm Lyne-Ashworth harmonic folding to an arbitrary number of harmonics:

```
In [72]: folds = mySpec.harmonicFold(5)

In [73]: folds
Out[73]:
[PowerSpectrum([ 1.        ,  1.83509839,  1.70531178, ...,  3.46097231,
                3.70728922,  2.55306649], dtype=float32),
 PowerSpectrum([ 1.        ,  3.7526474 ,  3.62286091, ...,  3.46097231,
                3.70728922,  2.55306649], dtype=float32),
 PowerSpectrum([ 1.        ,  7.58774567,  7.45795918, ...,  3.46097231,
                3.70728922,  2.55306649], dtype=float32),
 PowerSpectrum([ 1.        ,  15.2579422 ,  15.12815571, ...,   3.46097231,
                3.70728922,   2.55306649], dtype=float32),
 PowerSpectrum([ 1.        ,  30.59833527,  30.46854782, ...,   3.46097231,
                3.70728922,   2.55306649], dtype=float32)]
```

Where the variable `folds` is a python list containing each of the requested harmonic folds.

3.1.4 Folding data

Both the `sigpyproc.TimeSeries.TimeSeries` and the `sigpyproc.FourierSeries.FourierSeries` have methods to phase fold their data. Using our earlier `myFil` instance, we will fold our filterbank file with a period of 250 ms and a DM of pc cm $^{-3}$ and acceleration of 0 ms $^{-2}$.

```
In [116]: myFold = myFil.fold(0.25,30.,accel=0,nbins=128,nints=32,nbands=16)

Filterbank reading plan:
------------------------
Called on file:       tutorial.fil
Called by:            fold
Number of samps:      187520
Number of reads:      18
Nsamps per read:      10000
Nsamps of final read: 7826
Nsamps to skip back:  17

Execution time: 0.234850 seconds

In [119]: type(myFold)
Out[119]: sigpyproc.FoldedData.FoldedData

In [120]: myFold
Out[120]:
```

```
FoldedData([[[ 1.64893615,  1.60416663,  1.57000005, ...,  1.73809528,
        1.64772725,  1.59523809],
      [ 1.61702132,  1.58333337,  1.65999997, ...,  1.66666663,
        1.5795455 ,  1.73809528],
      [ 1.60638297,  1.55208337,  1.63999999, ...,  1.59523809,
        1.65909088,  1.66666663],
      ...,
      [ 1.59183669,  1.59523809,  1.5       , ...,  1.66666663,
        1.66999996,  1.64583337],
      [ 1.66326535,  1.57142854,  1.65909088, ...,  1.53125   ,
        1.61000001,  1.58333337],
      [ 1.70408165,  1.63095236,  1.59090912, ...,  1.71875   ,
        1.63      ,  1.6875    ]]], dtype=float32)

In [123]: myFold.shape
Out[123]: (32, 16, 128)
```

The the `sigpyproc.FoldedData.FoldedData` has several functions to enable simple slicing and summing of the folded data cube. These include:

- `getSubband`: select all data in a single frequency band
- `getSubint`: select all data in a single subintegration
- `getFreqPhase`: sum the data in the time axis
- `getTimePhase`: sum the data in the frequency axis
- `getProfile`: get the pulse profile of the fold

We can also tweak the DM and period of the fold using the `updateParams` method:

```
In [129]: myFold.updateParams(period=0.2502,dm=100)
```

3.1.5 Tips and tricks

There are several tips and tricks to help speed up sigpyproc and also make it more user friendly. For people who are familiar with Python and IPython these will be old news, but for newbies these may be of use.

Tab completion : One of the many nice things about IPython is that it allows for tab completion:

```
In [131]: myFil.  #then press tab
myFil.bandpass      myFil.getChan       myFil.readPlan
myFil.bitfact       myFil.getStats      myFil.sampsize
myFil.collapse      myFil.header        myFil.setNthreads
myFil.dedisperse    myFil.invertFreq    myFil.splitToChans
myFil.downsample    myFil.itemsize      myFil.upTo8bit
myFil.filename      myFil.lib
myFil.fold          myFil.readBlock
```

Docstrings : by using question marks or double question marks we can access both information about a function and its raw source:

```
In [132]: myFil.downsample?
Type:       instancemethod
String Form:<bound method FilReader.downsample of <sigpyproc.Readers.FilReader object at 0x10a70f1d0>
File:       /lib/python2.7/site-packages/sigpyproc/Filterbank.py
Definition: myFil.downsample(self, tfactor=1, ffactor=1, gulp=512, filename=None, back_compatible=Tru
Docstring:
Downsample data in time and/or frequency.
```

```
:param tfactor: factor by which to downsample in time
:type tfactor: int

:param ffactor: factor by which to downsample in frequency
:type ffactor: int

:param gulp: number of samples in each read
:type gulp: int

:param filename: name of file to write to (def = 'basename_tfactor_ffactor.fil' )
:type filename: string

:param back_compatible: sigproc compatibility flag
:type back_compatible: bool

:return: string -- name of new filterbank file
```

Note that all docstrings are written in *reStructuredText*. This is to facilitate automatic documentation creation with the Sphinx package.

Threading : The use of OpenMP means that several of c library calls in sigpyproc can be sped up:

```
In [149]: myFil.setNthreads(2) # enable OpenMP to use 2 threads
```

It should be noted that threading will not always speed up a process, and that the performance does not scale linearly with number of threads used.

Chaining : The ability to chain together methods, combined with history recall in IPython means that it is simple to condense a sigpyproc request into a single line:

```
In [157]: spectrum = FilReader("tutorial.fil").collapse().rFFT().rednoise().formSpec(True)
```

Here we create a `FilReader` instance, which is then collapsed in frequency, FFT'd, cleaned of rednoise and interpolated to form a power spectrum. In the intrests of readability, this is not always a good idea, however for testing code quickly, it is invaluable.

CHAPTER

FOUR

DEVELOPERS GUIDE

Here we will cover all the steps required to add new functionality to the sigpyproc package. To do this, we will first consider adding a new function to the `sigpyproc.Filterbank.Filterbank` class, before going on to extend the function to deal with an arbitrary data format.

4.1 Adding a new function: `bandpass()`

Aim: Add a new function called bandpass, which will return the total power as a function of frequency for our observation.

Files to be modified: `sigpyproc/Filterbank.py`, `c_src/sigpyproc8.c` and `c_src/sigpyproc32.c` (if we wish our function to work with 32-bit data).

4.1.1 Step 1: Write the Python part

The first step is to write the Python side of the function. As this function will run on data with both time and frequency resolution, it belongs in the `sigpyproc.Filterbank.Filterbank` class.

```
def bandpass(self,gulp=512):
    bpass_ar = np.zeros(self.header.nchans,dtype="float32")
    bpass_ar_c = as_c(bpass_ar)

    for nsamps,ii,data in self.readPlan(gulp):
        self.lib.getBpass(as_c(data),
                          bpass_ar_c,
                          C.c_int(self.header.nchans),
                          C.c_int(nsamps))

    new_header = self.header.newHeader({"nchans":1})
    return TimeSeries(bpass_ar, new_header)
```

Looking at the important lines, we have:

```
def bandpass(self,gulp=512):
```

1. A normal Python function definition. Here we define `gulp` and give in the default value 512. I'll come back to the gulp keyword later on.

```
bpass_ar = np.zeros(self.header.nchans,dtype="float32")
```

2. Create an empty array to store the output of our function. Here we must explicitly state the `dtype` for the array, such that it can be passed to a C library function.

```
bpass_ar_c = as_c(bpass_ar)
```

3. Create a C version of our empty array. This does not create a copy of the array, but rather a new view of the array, enabling it to be passed as an argument to a C function.

```
for nsamps,ii,data in self.readPlan(gulp):
```

5. Call the `readPlan` method with `gulp` as an argument. `readPlan` returns an itterable generator object that provides a simple interface for reading through a given file type. For each iteration of the generator, it yeilds three parameters; `data`, which is a `numpy.ndarray` instance containing a block of frequency-major order data (i.e. standard column-major order filterbank data); `nsamps`, which is the number of time samples in `data` (i.e. there are `nsamps*nchans` points in the `data` array); and `ii`, the current index number of the read (i.e. the first read has `ii=0`, while the last has index `ii=nreads`).

```
self.lib.getBpass(as_c(data),
                  bpass_ar_c,
                  C.c_int(self.header.nchans),
                  C.c_int(gulp))
```

6. Call the C library function. This will be covered in more detail further on.

```
new_header = self.header.newHeader({"nchans":1})
```

11. Update the observational metadata accordingly, such that it can be passed on to the output of our function.

```
return TimeSeries(bpass_ar, new_header)
```

12. Return an instance of the `sigpyproc.TimeSeries.TimeSeries` class. The `sigpyproc.TimeSeries.TimeSeries` class takes two arguments, an instance of `numpy.ndarray` and an instance of `sigpyproc.Header.Header`.

Now we have something similar to a normal `numpy.ndarray`, which exports several other methods for convenience.

4.1.2 Step 2: Write the C part

In line 6 of the Python code, we called a c library function named `getBpass`. This function belongs in the `c_src/sigpyproc8.c` and `c_src/sigpyproc32.c` files (or anywhere you please, as long as it is compiled into the `libsigpyproc8.so` and `libsigpyproc32.so` shared object libraries). In `sigpyproc8.c`, our function looks like:

```c
void getBpass(unsigned char* inbuffer,
              float* outbuffer,
              int nchans,
              int nsamps){
  int ii,jj;
#pragma omp parallel for default(shared) private(jj,ii)
  for (jj=0;jj<nchans;jj++){
    for (ii=0;ii<nsamps;ii++){
      outbuffer[jj]+=inbuffer[(nchans*ii)+jj];
    }
  }
}
```

with the only change for `sigpyproc32.c` being a type change in the first line:

```c
void getBpass(float* inbuffer,
```

This function receives a block of data and sums that block along the time axis. The line:

```
#pragma omp parallel for default(shared) private(jj,ii)
```

is an optional C prepreprocessor directive to enable OpenMP threading. Realistically this call is not required here, but it is left here of a clear example of how to quicly multi-thread a system of loops.

4.1.3 Step 3: Putting it together

The last step is to rebuild the package:

```
python setup.py install
```

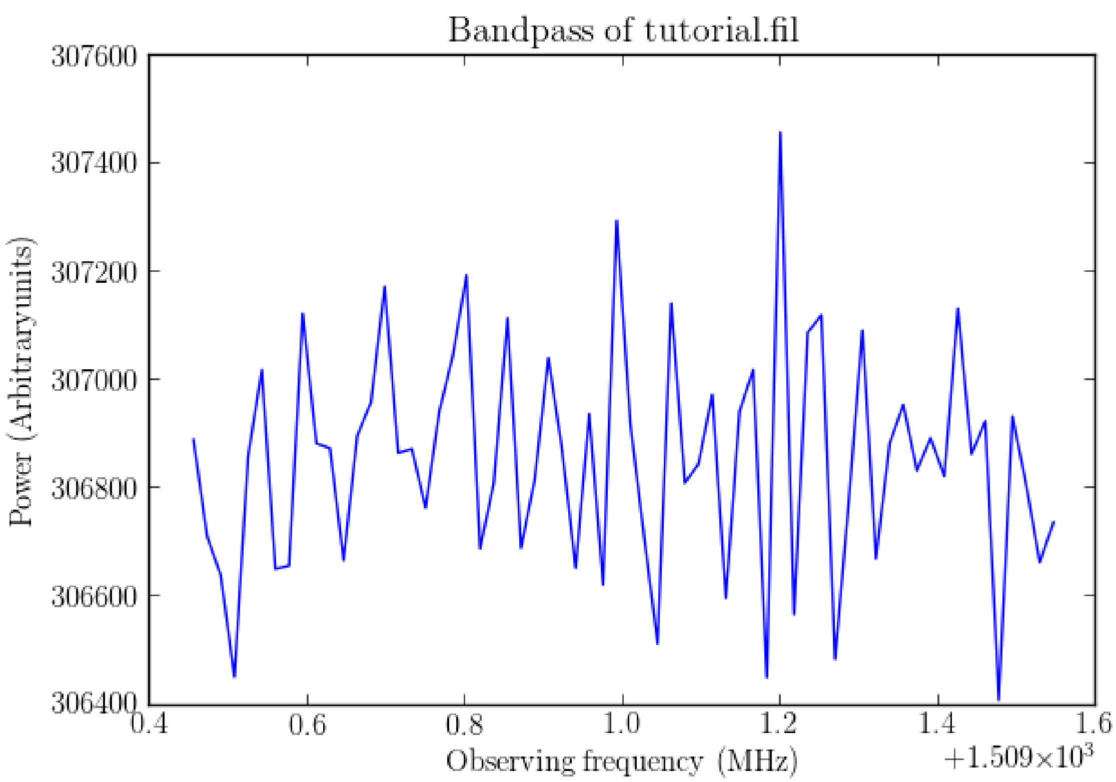
We have now added new functionality to the package, with an example usage being:

```python
from sigpyproc.Readers import FilReader
import matplotlib.pyplot as plt

#compute the bandpass and frequency of each channel
my_bpass = FilReader("tutorial.fil").bandpass()
freqs = np.linspace(my_bpass.header.ftop, my_bpass.header.fbottom, my_bpass.size)

#plot the result
plt.plot(freqs,my_bpass)
plt.title("Bandpass of tutorial.fil")
plt.xlabel("Observing frequency (MHz)")
plt.ylabel("Power (Arbitrary units)")
plt.show()
```

Which will produce:



4.2 Understanding sigpyproc IO

To understand how sigpyproc works, we must take a closer look at the `sigpyproc.Readers` module.

In the tutorial, we looked at loading a filterbank format file into sigpyproc. Below, we can see what happens during this action:

1. `sigpyproc.Readers.FilReader.__init__` is called with the argument `"tutorial.fil"`.
2. In the `sigpyproc.Readers.FilReader.__init__` method, `"tutorial.fil"` is opened and its header is read with the `parseSigprocHeader` function. This returns an instance of `Header`, which is stored in the `header` attribute.
3. `FilReader` then inherits from `Filterbank` via Python's built-in `super()` method.

We now have an instance of the `FilReader` class that allows us to read through the filterbank file in chunks via the `readPlan()` method, while manipulating each chunk with the methods of the `Filterbank` class.

As the `sigpyproc.Readers.FilReader.readPlan()` method will be different from `sigpyproc.Readers.SomeOtherReader.readPlan()`, we must always call specific data types with their correct reading class.

4.2.1 Adding support for hypothetical `.neu` format

As we saw above, the `FilReader` class is simply an interface that allows us to get data from files in filterbank format and manipulate that data so that it can be used by the methods of the `Filterbank` class. It is clear, therfore, that to extend sigpyproc to support a new data format we must simply write a new class to read the data.

Lets imagine a hypothetical data format that is denoted by the file extension `.neu`. To enable sigpyproc to read this file, we would do the following:

1. Write a class called `sigpyproc.Readers.NeuReader` which inherits from `Filterbank`.
2. Give `NeuReader` an attribute called `header` which contains an instance of `Header`. This instance should contain all the relevant observational metadata in sigpyproc format (i.e. the same format as seen in dictionaries contianed in the `sigpyproc.HeaderParams` module)
3. Give `NeuReader` a method called `readPlan()` which takes at least the following four arguments:
   - **Arg:** gulp – the number of time samples in each read (gulp*nchans*nbits/8 bytes will be read)
   - **Kwarg:** skipback=0 – the number of samples to skip back after each read (default should be zero)
   - **Kwarg:** start=0 – the first sample to start reading from (should default to start of data)
   - **Kwarg:** nsamps=None – the total number of samples to read (defaults to `total number of samples - start`)

   The `readPlan()` method should return a generator object which yields the following for each itteration (in this order):

   - The number of samples in the current read
   - The index of the current read (where 0 is the first read)
   - A 1-D frequency-major order `numpy.ndarray` instance containing the data from the current read
4. Give `NeuReader` a method called `readBlock()` which takes at least the following two arguments:
   - **Arg:** start – the first sample to start reading from
   - **Arg:** nsamps – the total number of samples to read

The `readBlock()` method should read a block of data and transform it into time-major order, before returning an instance of the `FilterbankBlock` class.

5. For complicated file formats, it may be necessary to either subclass or write a new version of the `sigpyproc.Utils.File` class (this class handles reading and writing of arbitrary bit size data to file).

CHAPTER

FIVE

SIGPYPROC PACKAGE

5.1 `Filterbank` Module

class `sigpyproc.Filterbank.`**`Filterbank`**

Bases: `object`

Class exporting methods for the manipulation of frequency-major order pulsar data.

Note: The Filterbank class should never be instantiated directly. Instead it should be inherited by data reading classes.

`bandpass`(*gulp=512*)

Sum across each time sample for all frequencies.

Parameters **gulp** (*int*) – number of samples in each read

Returns the bandpass of the data

Return type `TimeSeries`

`collapse`(*gulp=512*, *start=0*, *nsamps=None*)

Sum across all frequencies for each time sample.

Parameters **gulp** (*int*) – number of samples in each read

Returns A zero-DM time series

Return type `TimeSeries`

`dedisperse`(*dm*, *gulp=10000*)

Dedisperse the data to a time series.

Parameters

- **dm** (*float*) – dispersion measure to dedisperse to
- **gulp** (*int*) – number of samples in each read

Returns a dedispersed time series

Return type `TimeSeries`

Note: If gulp < maximum dispersion delay, gulp is taken to be twice the maximum dispersion delay.

`downsample`(*tfactor=1*, *ffactor=1*, *gulp=512*, *filename=None*, *back_compatible=True*)

Downsample data in time and/or frequency and write to file.

Parameters

- **tfactor** (*int*) – factor by which to downsample in time
- **ffactor** (*int*) – factor by which to downsample in frequency
- **gulp** (*int*) – number of samples in each read
- **filename** (*str*) – name of file to write to (defaults to `basename_tfactor_ffactor.fil`)
- **back_compatible** (*bool*) – sigproc compatibility flag (legacy code)

Returns output file name

Return type `str()`

`fold`(*period*, *dm*, *accel=0*, *nbins=50*, *nints=32*, *nbands=32*, *gulp=10000*)

Fold data into discrete phase, subintegration and subband bins.

Parameters

- **period** (*float*) – period in seconds to fold with
- **dm** (*float*) – dispersion measure to dedisperse to
- **accel** (*float*) – acceleration in m/s/s to fold with
- **nbins** (*int*) – number of phase bins in output
- **nints** (*int*) – number of subintegrations in output
- **nbands** (*int*) – number of subbands in output
- **gulp** (*int*) – number of samples in each read

Returns 3 dimensional data cube

Return type `FoldedData`

Note: If gulp < maximum dispersion delay, gulp is taken to be twice the maximum dispersion delay.

`getChan`(*chan*, *gulp=512*)

Retrieve a single frequency channel from the data.

Parameters

- **chan** (*int*) – channel to retrieve (0 is the highest frequency channel)
- **gulp** (*int*) – number of samples in each read

Returns selected channel as a time series

Return type `TimeSeries`

`getStats`(*gulp=512*)

Retrieve channelwise statistics of data.

Parameters **gulp** (*int*) – number of samples in each read

Function creates four instance attributes:

- `chan_means`: the mean value of each channel
- `chan_stdevs`: the standard deviation of each channel
- `chan_max`: the maximum value of each channel

- `chan_min`: the minimum value of each channel

invertFreq(*gulp=512*, *start=0*, *nsamps=None*, *filename=None*, *back_compatible=True*)
Invert the frequency ordering of the data and write new data to a new file.

Parameters

- **gulp** (*int*) – number of samples in each read
- **start** (*int*) – start sample
- **nsamps** (*int*) – number of samples in split
- **filename** (*string*) – name of output file (defaults to `basename_inverted.fil`)
- **back_compatible** (*bool*) – sigproc compatibility flag (legacy code)

Returns name of output file

Return type `str()`

setNthreads(*nthreads=None*)
Set the number of threads available to OpenMP.

Parameters nthreads (*int*) – number of threads to use (def = 4)

splitToChans(*gulp=1024*, *back_compatible=True*)
Split the data into component channels and write each to file.

Parameters

- **gulp** (*int*) – number of samples in each read
- **back_compatible** (*bool*) – sigproc compatibility flag (legacy code)

Returns names of all files written to disk

Return type `list()` of `str()`

Note: Time series are written to disk with names based on channel number.

upTo8bit(*filename=None*, *gulp=512*, *back_compatible=True*)
Convert 1-,2- or 4-bit data to 8-bit data and write to file.

Parameters

- **filename** (*str*) – name of file to write to (defaults to `basename_8bit.fil`)
- **gulp** (*int*) – number of samples in each read
- **back_compatible** (*bool*) – sigproc compatibility flag

Returns name of output file

Return type `str()`

class `sigpyproc.Filterbank.`**FilterbankBlock**
Bases: `numpy.ndarray`

Class to handle a discrete block of data in time-major order.

Parameters

- **input_array** (`numpy.ndarray`) – 2 dimensional array of shape (nchans,nsamples)
- **header** (`Header`) – observational metadata

Note: Data is converted to 32 bits regardless of original type.

dedisperse(*dm*)
Dedisperse the block.

Parameters **dm** (*float*) – dm to dedisperse to

Returns a dedispersed version of the block

Return type `FilterbankBlock`

Note: Frequency dependent delays are applied as rotations to each channel in the block.

downsample(*tfactor=1*, *ffactor=1*)
Downsample data block in frequency and/or time.

Parameters

- **tfactor** (*int*) – factor by which to downsample in time
- **ffactor** (*int*) – factor by which to downsample in frequency

Returns 2 dimensional array of downsampled data

Return type `FilterbankBlock`

Note: ffactor must be a factor of nchans.

normalise()
Divide each frequency channel by its average.

Returns normalised version of the data

Return type `FilterbankBlock`

toFile(*filename=None*, *back_compatible=True*)
Write the data to file.

Parameters

- **filename** (*str*) – name of the output file (defaults to `basename_split_start_to_end.fil`)
- **back_compatible** (*bool*) – sigproc compatibility flag (legacy code)

Returns name of output file

Return type `str()`

5.2 FoldedData Module

class `sigpyproc.FoldedData.`**FoldSlice**
Bases: `numpy.ndarray`

Class to handle a 2-D slice of a `FoldedData` instance.

Parameters **input_array** (`numpy.ndarray`) – a 2-D array with phase in x axis.

getProfile()

Return the pulse profile from the slice.

Returns a pulse profile

Return type `Profile`

normalise()

Normalise the slice by dividing each row by its mean.

Returns normalised version of slice

Return type `FoldSlice`

class `sigpyproc.FoldedData.`**FoldedData**

Bases: `numpy.ndarray`

Class to handle a data cube produced by any of the sigpyproc folding methods.

Parameters

- **input_array** (`numpy.ndarray`) – 3-D array of folded data
- **header** (`Header`) – observational metadata
- **period** (*float*) – period that data was folded with
- **dm** (*float*) – DM that data was folded with
- **accel** (*float*) – accleration that data was folded with (def=0)

Note: Data cube should have the shape: (number of subintegrations, number of subbands, number of profile bins)

---

centre()

Try and roll the data cube to center the pulse.

getFreqPhase()

Return the data cube collapsed in time.

Returns a 2-D array containing the frequency vs. phase plane

Return type `FoldSlice`

getProfile()

Return a the data cube summed in time and frequency.

Returns a 1-D array containing the power as a function of phase (pulse profile)

Return type `Profile`

getSubband(*n*)

Return a single subband from the data cube.

Parameters **n** (*int*) – subband number (n=0 is first subband)

Returns a 2-D array containing the subband

Return type `FoldSlice`

getSubint(*n*)

Return a single subintegration from the data cube.

Parameters **n** (*int*) – subintegration number (n=0 is first subintegration)

Returns a 2-D array containing the subintegration

Return type `FoldSlice`

getTimePhase()
Return the data cube collapsed in frequency.

Returns a 2-D array containing the time vs. phase plane

Return type `FoldSlice`

updateParams(*dm=None*, *period=None*)
Install a new folding period and/or DM in the data cube.

Parameters

- **dm** (*float*) – the new DM to dedisperse to
- **period** (*float*) – the new period to fold with

class `sigpyproc.FoldedData.`**Profile**
Bases: `numpy.ndarray`

Class to handle a 1-D pulse profile.

Parameters **input_array** (`numpy.ndarray`) – a pulse profile in array form

SN()
Return a rudimentary signal-to-noise measure for the profile.

Note: This is a bare-bones, quick-n'-dirty algorithm that should not be used for high quality signal-to-noise measurements.

---

retroProf(*height=0.7*, *width=0.7*)
Display the profile in ASCII formay in the terminal window.

Parameters

- **height** (*float*) – fraction of terminal rows to use
- **width** – fraction of terminal columns to use
- **width** –

Note: This function requires a system call to the Linux/Unix `stty` command.

5.3 `FourierSeries` Module

class `sigpyproc.FourierSeries.`**FourierSeries**
Bases: `numpy.ndarray`

Class to handle output of FFT'd time series.

Parameters

- **input_array** (`numpy.ndarray`) – 1 dimensional array of shape (nsamples)
- **header** (`Header`) – observational metadata

formSpec(*interpolated=True*)
Form power spectrum.

Parameters **interpolated** (*bool*) – flag to set nearest bin interpolation (def=True)

Returns a power spectrum

Return type `PowerSpectrum`

iFFT()

Perform 1-D complex to real inverse FFT using FFTW3.

Returns a time series

Return type `TimeSeries`

reconProf(*freq*, *nharms=32*)

Reconstruct the time domain pulse profile from a signal and its harmonics.

Parameters

- **freq** (*float*) – frequency of signal to reconstruct
- **nharms** (*int*) – number of harmonics to use in reconstruction (def=32)

Returns a pulse profile

Return type `sigpyproc.FoldedData.Profile`

rednoise(*startwidth=6*, *endwidth=100*, *endfreq=1.0*)

Perform rednoise removal via Presto style method.

Parameters

- **startwidth** (*int*) – size of initial array for median calculation
- **endwidth** (*int*) – size of largest array for median calculation
- **endfreq** (*float*) – remove rednoise up to this frequency

Returns whitened fourier series

Return type `FourierSeries`

toFFTFile(*basename=None*)

Write spectrum to file in sigpyproc format.

Parameters **basename** – basename of .fft and .inf file to be written

Returns name of files written to

Return type `tuple()` of `str()`

toFile(*filename=None*)

Write spectrum to file in sigpyproc format.

Parameters **filename** (*str*) – name of file to write to (def=‘‘basename.spec‘‘)

Returns name of file written to

Return type `str()`

class `sigpyproc.FourierSeries.`**PowerSpectrum**

Bases: `numpy.ndarray`

Class to handle power spectra.

Parameters

- **input_array** (`numpy.ndarray`) – 1 dimensional array of shape (nsamples)
- **header** (`Header`) – observational metadata

bin2freq(*bin_*)
Return centre frequency of a given bin.

Parameters **bin** (*int*) – bin number

Returns frequency of bin

Return type float

bin2period(*bin_*)
Return centre period of a given bin.

Parameters **bin** (*int*) – bin number

Returns period of bin

Return type float

freq2bin(*freq*)
Return nearest bin to a given frequency.

Parameters **freq** (*float*) – frequency

Returns nearest bin to frequency

Return type float

harmonicFold(*nfolds=1*)
Perform Lyne-Ashworth harmonic folding of the power spectrum.

Parameters **nfolds** (*int*) – number of harmonic folds to perform (def=1)

Returns A list of folded spectra where the i th element is the spectrum folded i times.

Return type `list()` of `PowerSpectrum`

period2bin(*period*)
Return nearest bin to a given periodicity.

Parameters **period** (*float*) – periodicity

Returns nearest bin to period

Return type float

5.4 `Header` Module

class `sigpyproc.Header.`**Header**(*info*)
Bases: `dict`

Container object to handle observation metadata.

Note: Attributes are mirrored as items and vice versa to facilitate cleaner code.

SPPHeader(*back_compatible=True*)
Get Sigproc/sigpyproc format binary header.

Parameters **back_compatible** (*bool*) – Flag for returning Sigproc compatible header (legacy code)

Returns header in binary format

Return type `str()`

dedispersedHeader(*dm*)
Get a dedispersed version of the current header.

Parameters **dm** (*float*) – dispersion measure we are dedispersing to

Returns A dedispersed version of the header

Return type `Header`

getDMdelays(*dm*, *in_channels=True*)
For a given dispersion measure get the dispersive ISM delay for each frequency channel.

Parameters **dm** (*float*) – dispersion measure to calculate delays for

Returns delays for each channel (highest frequency first)

Return type `numpy.ndarray`

makeInf(*outfile=None*)
Make a presto format .inf file.

Parameters **outfile** (*string*) – a filename to write to.

Returns if outfile is unspecified .inf data is returned as string

Return type `str()`

mjdAfterNsamps(*nsamps*)
Find the Modified Julian Date after nsamps have elapsed.

Parameters **nsamps** (*int*) – number of samples elapsed since start of observation.

Returns Modified Julian Date

Return type

newHeader(*update_dict=None*)
Create a new instance of `Header` from the current instance.

Parameters **update_dict** (`dict()`) – values to overide existing header values

Returns new header information

Return type `Header`

prepOutfile(*filename*, *updates=None*, *nbits=None*, *back_compatible=True*)
Prepare a file to have sigproc format data written to it.

Parameters

- **filename** (*string*) – filename of new file
- **updates** (*dict*) – values to overide existing header values
- **nbits** (*int*) – the bitsize of data points that will written to this file (1,2,4,8,32)
- **back_compatible** (*bool*) – flag for making file Sigproc compatible

Returns a prepared file

Return type `File`

updateHeader()
Check for changes in header and recalculate all derived quantaties.

`sigpyproc.Header.`**MJD_to_Gregorian**(*mjd*)
Convert Modified Julian Date to the Gregorian calender.

Parameters **mjd** – Modified Julian Date

Returns date and time

Return type `tuple()` of `str()`

`sigpyproc.Header.`**`dec_to_rad`**(*dec_string*)
Convert declination string to radians.

`sigpyproc.Header.`**`dms_to_deg`**(*deg*, *min_*, *sec*)
Convert (degrees, arcminutes, arcseconds) to degrees.

`sigpyproc.Header.`**`dms_to_rad`**(*deg*, *min_*, *sec*)
Convert (degrees, arcminutes, arcseconds) to radians.

`sigpyproc.Header.`**`hms_to_hrs`**(*hour*, *min*, *sec*)
Convert (hours, minutes, seconds) to hours.

`sigpyproc.Header.`**`hms_to_rad`**(*hour*, *min*, *sec*)
Convert (hours, minutes, seconds) to radians.

`sigpyproc.Header.`**`ra_to_rad`**(*ra_string*)
Convert right ascension string to radians.

`sigpyproc.Header.`**`rad_to_dms`**(*rad*)
Convert radians to (degrees, arcminutes, arcseconds).

`sigpyproc.Header.`**`rad_to_hms`**(*rad*)
Convert radians to (hours, minutes, seconds).

`sigpyproc.Header.`**`radec_to_str`**(*val*)
Convert Sigproc format RADEC float to a string.

Parameters **val** (*float*) – Sigproc style RADEC float (eg. 124532.123)

Returns 'xx:yy:zz.zzz' format string

Return type `str()`

5.5 `Readers` Module

class `sigpyproc.Readers.`**`FilReader`**(*filename*)
Bases: `sigpyproc.Filterbank.Filterbank`

Class to handle the reading of sigproc format filterbank files

Parameters **filename** (`str()`) – name of filterbank file

Note: To be considered as a Sigproc format filterbank file the header must only contain keywords found in the `HeaderParams.header_keys` dictionary.

`readBlock`(*start*, *nsamps*)
Read a block of filterbank data.

Parameters

- **start** (*int*) – first time sample of the block to be read
- **nsamps** (*int*) – number of samples in the block (i.e. block will be nsamps*nchans in size)

Returns 2-D array of filterbank data

Return type `FilterbankBlock`

readPlan (*gulp*, *skipback=0*, *start=0*, *nsamps=None*, *verbose=True*)
A generator used to perform filterbank reading.

Parameters

- **gulp** (*int*) – number of samples in each read
- **skipback** (*int*) – number of samples to skip back after each read (def=0)
- **start** (*int*) – first sample to read from filterbank (def=start of file)
- **nsamps** (*int*) – total number samples to read (def=end of file)
- **verbose** (*bool*) – flag for display of reading plan information (def=True)

Returns An generator that can read through the file.

Return type generator object

Note: For each read, the generator yields a tuple `x`, where:

- `x[0]` is the number of samples read
- `x[1]` is the index of the read (i.e. `x[1]=0` is the first read)
- `x[2]` is a 1-D numpy array containing the data that was read

The normal calling syntax for this is function is:

```
for nsamps, ii, data in self.readPlan(*args,**kwargs):
    # do something
```

where data always has contains `nchans*nsamps` points.

`sigpyproc.Readers.`**parseInfHeader** (*filename*)
Parse the metadata from a presto `.inf` file.

Parameters **filename** (`str()`) – file containing the header

Returns observational metadata

Return type `Header`

`sigpyproc.Readers.`**parseSigprocHeader** (*filename*)
Parse the metadata from a Sigproc-style file header.

Parameters **filename** (`str()`) – file containing the header

Returns observational metadata

Return type `Header`

`sigpyproc.Readers.`**readDat** (*filename*, *inf=None*)
Read a presto format .dat file.

Parameters **filename** (`str()`) – the name of the file to read

Params inf the name of the corresponding .inf file (def=None)

Returns an array containing the whole dat file contents

Return type `TimeSeries`

Note: If inf=None, the function will look for a corresponding file with the same basename which has the .inf file extension.

`sigpyproc.Readers.readFFT(filename, inf=None)`
Read a presto .fft format file.

Parameters **filename** (`str()`) – the name of the file to read

Params inf the name of the corresponding .inf file (def=None)

Returns an array containing the whole file contents

Return type `FourierSeries`

Note: If inf=None, the function will look for a corresponding file with the same basename which has the .inf file extension.

`sigpyproc.Readers.readSpec(filename)`
Read a sigpyproc format spec file.

Parameters **filename** (`str()`) – the name of the file to read

Returns an array containing the whole file contents

Return type `FourierSeries`

Note: This is not setup to handle `.spec` files such as are created by Sigprocs seek module. To do this would require a new header parser for that file format.

`sigpyproc.Readers.readTim(filename)`
Read a sigproc format time series from file.

Parameters **filename** (`str()`) – the name of the file to read

Returns an array containing the whole file contents

Return type `TimeSeries`

5.6 `TimeSeries` Module

class `sigpyproc.TimeSeries.TimeSeries`
Bases: `numpy.ndarray`

Class for handling pulsar data in time series.

Parameters

- **input_array** (`numpy.ndarray`) – 1 dimensional array of shape (nsamples)
- **header** (`Header`) – observational metadata

`applyBoxcar(width)`
Apply a boxcar filter to the time series.

Parameters **width** (*int*) – width in bins of filter

Returns filtered time series

Return type `TimeSeries`

Note: Time series returned is of size nsamples-width with width/2 removed removed from either end.

---

downsample(*factor*)

Downsample the time series.

Parameters **factor** (*int*) – factor by which time series will be downsampled

Returns downsampled time series

Return type `TimeSeries`

Note: Returned time series is of size nsamples//factor

---

fold(*period*, *accel=0*, *nbins=50*, *nints=32*)

Fold time series into discrete phase and subintegration bins.

Parameters

- **period** (*float*) – period in seconds to fold with
- **nbins** (*int*) – number of phase bins in output
- **nints** (*int*) – number of subintegrations in output

Returns data cube containing the folded data

Return type `FoldedData`

pad(*npad*)

Pad a time series with mean valued data.

Parameters **npad** – number of padding points

Returns padded time series

Return type `sigpyproc.TimeSeries.TimeSeries`

rFFT()

Perform 1-D real to complex forward FFT.

Returns output of FFTW3

Return type `FourierSeries`

runningMean(*window=10001*)

Filter time series with a running mean.

Parameters **window** (*int*) – width in bins of running mean filter

Returns filtered time series

Return type `TimeSeries`

Note: Window edges will be dealt with only at the start of the time series.

---

runningMedian(*window=10001*)

Filter time series with a running median.

Parameters **window** (*int*) – width in bins of running median filter

Returns filtered time series

Return type `TimeSeries`

Note: Window edges will be dealt with only at the start of the time series.

---

toDat (*basename*)
Write time series in presto `.dat` format.

Parameters **basename** (*string*) – file basename for output `.dat` and `.inf` files

Returns `.dat` file name and `.inf` file name

Return type `tuple()` of `str()`

Note: Method also writes a corresponding .inf file from the header data

---

toFile (*filename*)
Write time series in sigproc format.

Parameters **filename** (*str*) – output file name

Returns output file name

Return type `str()`

5.7 `Utils` Module

class `sigpyproc.Utils.`**File** (*filename*, *mode*, *nbits*)
Bases: `file`

A class to handle writing of arbitrary bit size data to file.

Parameters

- **filename** (`str()`) – name of file to open
- **mode** (`str()`) – file access mode, can be either "r", "r+", "w" or "a".
- **nbits** – the bit size of units to be read from or written to file

---

Note: The File class handles all packing and unpacking of sub-byte size data under the hood, so all calls can be made requesting numbers of units rather than numbers of bits or bytes.

---

cread (*nunits*)
Read nunits of data from the file.

Parameters **nunits** (*int*) – number of units to be read from file

Returns an array containing the read data

Return type `numpy.ndarray`

cwrite (*ar*)
Write an array to file.

Parameters **ar** (`numpy.ndarray`) – a numpy array

Note: Regardless of the dtype of the array argument, the data will be packed with a bitsize determined by the nbits attribute of the File instance. To change this attribute, use the _setNbits methods.

`sigpyproc.Utils.`**editInplace** (*inst*, *key*, *value*)
Edit a sigproc style header in place

Parameters

- **inst** (`Header`) – a header instance with a `.filename` attribute
- **key** (`str()`) – name of parameter to change (must be a valid sigproc key)
- **value** – new value to enter into header

Note: It is up to the user to be responsible with this function, as it will directly change the file on which it is being operated. The only fail contition of editInplace comes when the new header to be written to file is longer or shorter than the header that was previously in the file.

`sigpyproc.Utils.`**`nearestFactor`**(*n*, *val*)

Find nearest factor.

Parameters

- **n** (*int*) – number that we wish to factor
- **val** (*int*) – number that we wish to find nearest factor to

Returns nearest factor

Return type int

`sigpyproc.Utils.`**`rollArray`**(*y*, *shift*, *axis*)

Roll the elements in the array by ‘shift’ positions along the given axis.

Args: y – array to roll shift – number of bins to shift by axis – axis to roll along

Returns: shifted Ndarray

`sigpyproc.Utils.`**`stackRecarrays`**(*arrays*)

Wrapper for stacking `numpy.recarrays`

CHAPTER

SIX

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

S

INDEX

N

P

R

S

T

U