PsrSimSig Documentation

Release 1.0.0

Jeffrey S. Hazboun

Oct 15, 2020

Contents

CHAPTER 1

PsrSigSim

1.1 The NANOGrav Pulsar Signal Simulator

- Free software: MIT license
- Documentation: https://psrsigsim.readthedocs.io.

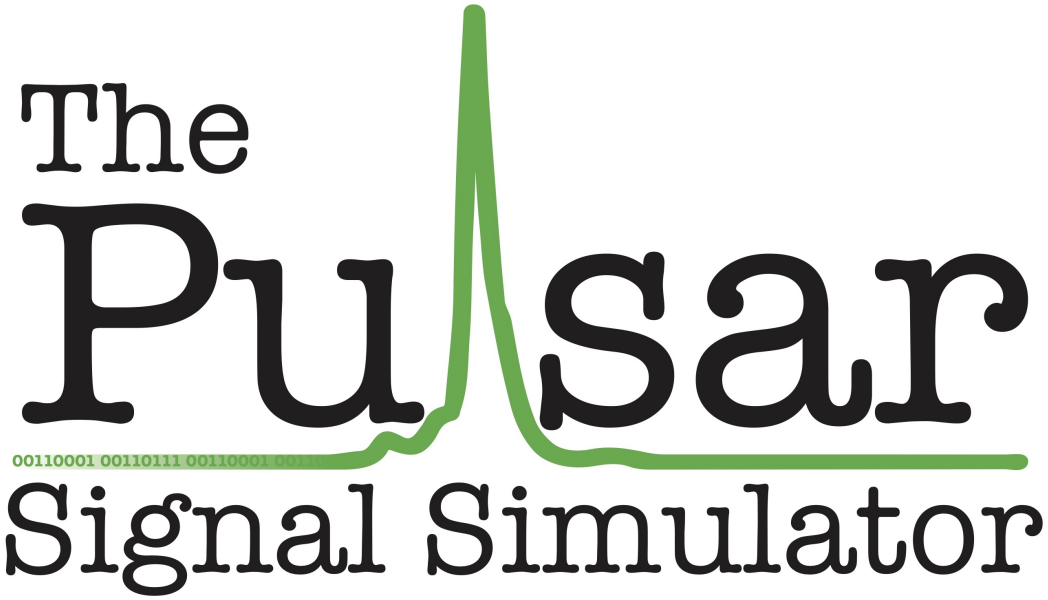


Image Credit: Caitlin Witt

The Pulsar Signal Simulator (*PsrSigSim*) is a Python package developed by the North American Nanohertz Observatory for Gravitational Waves (NANOGrav). This software is free to use and is designed as a tool for simulating realistic pulsar signals, as well as educating students and researchers about those signals. Various models from the copious pulsar literature are used for simulating emission, propagation effects and data processing artifacts.

1.2 Goals

- **Investigate sources of time-of-arrival errors:** Simulate various sources of time of arrival errors, including interstellar medium effects, intrinsic pulsar noise, various pulsar emission mechanisms and gravitational waves. Simulate instrumental noise sources, including radio frequency interference, radiometer noise and backend sampling effects.
- **Education and Outreach:** Allow users to build pulsar signals piece-by-piece to demonstrate to students how pulsar signals change as they propagate and how they are changed by the signal processing done at the telescope. Make materials for talks and outreach including signal diagrams and sound files of sonified pulsars.
- **Search algorithms and search training:** Test new search algorithms on signals with known parameters. Use simulated signals for search training. Assess the sensitivity of current search algorithms with simulated signals.

1.3 Code of Conduct

The *PsrSigSim* community expects contributors to follow a Code of Conduct outlined with our documentation.

1.4 Credits

Development Team: Jeffrey S. Hazboun, Brent Shapiro-Albert, Paul T. Baker

This package was created with Cookiecutter and the audreyr/cookiecutter-pypackage project template.

1.5 Contents

1.5.1 PsrSigSim

The NANOGrav Pulsar Signal Simulator

- Free software: MIT license
- Documentation: https://psrsigsim.readthedocs.io.

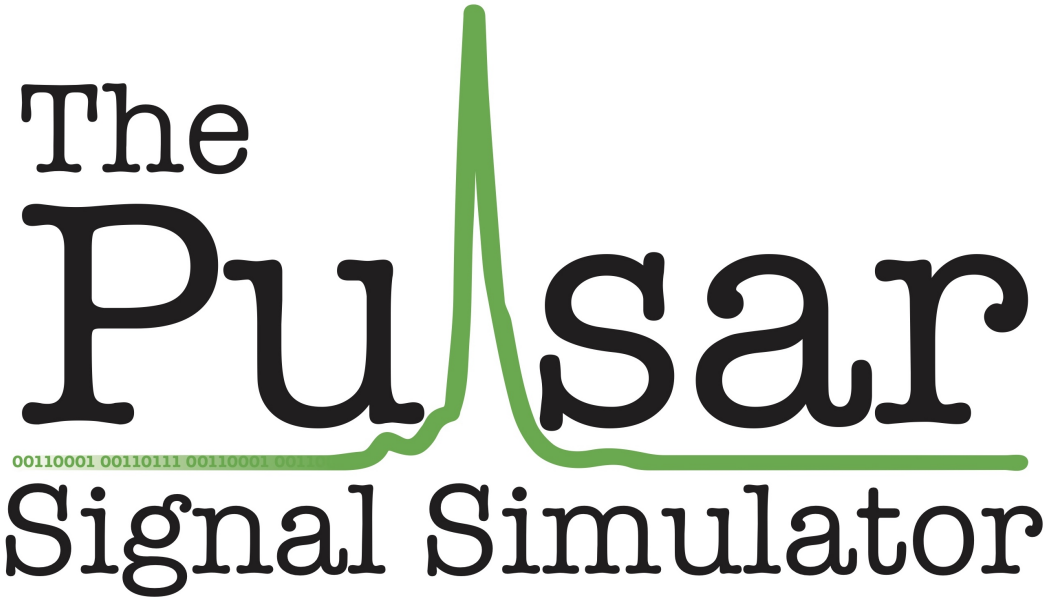


Image Credit: Caitlin Witt

The Pulsar Signal Simulator (*PsrSigSim*) is a Python package developed by the North American Nanohertz Observatory for Gravitational Waves (NANOGrav). This software is free to use and is designed as a tool for simulating realistic pulsar signals, as well as educating students and researchers about those signals. Various models from the copious pulsar literature are used for simulating emission, propagation effects and data processing artifacts.

Goals

- **Investigate sources of time-of-arrival errors:** Simulate various sources of time of arrival errors, including interstellar medium effects, intrinsic pulsar noise, various pulsar emission mechanisms and gravitational waves. Simulate instrumental noise sources, including radio frequency interference, radiometer noise and backend sampling effects.

- **Education and Outreach:** Allow users to build pulsar signals piece-by-piece to demonstrate to students how pulsar signals change as they propagate and how they are changed by the signal processing done at the telescope. Make materials for talks and outreach including signal diagrams and sound files of sonified pulsars.
- **Search algorithms and search training:** Test new search algorithms on signals with known parameters. Use simulated signals for search training. Assess the sensitivity of current search algorithms with simulated signals.

Code of Conduct

The *PsrSigSim* community expects contributors to follow a Code of Conduct outlined with our documentation.

Credits

Development Team: Jeffrey S. Hazboun, Brent Shapiro-Albert, Paul T. Baker

This package was created with Cookiecutter and the audreyr/cookiecutter-pypackage project template.

1.5.2 Installation

Stable release

To install PsrSimSig, run this command in your terminal:

```
$ pip install psrsigsim
```

This is the preferred method to install PsrSimSig, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

If you have issues with the *fitsio* installation and you already have *cfitsio* installed you may need to reinstall or reconfigure with different flags.

From sources

The sources for PsrSimSig can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/psrsigsim/psrsigsim
```

Or download the tarball:

```
$ curl  -OL https://github.com/psrsigsim/psrsigsim/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

1.5.3 Usage

To use PsrSimSig in a project:

```
import psrsigsim
```

An advanced example, based on [link to Brent's paper] can be found on MyBinder:

Note: This tutorial was generated from a Jupyter notebook that can be downloaded here.

1.5.4 Getting Started: Introductory Tutorial 1

This notebook introduces the basic features of the pulsar signal simulator, and leading the user through the steps of how to simulate a pulsar signal from start to finish.

The `PsrSigSim` can be run in a jupyter notebook or python script.

```
# import some useful packages
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

# import the pulsar signal simulator
import psrsigsim as pss
```

The Signal

The first thing we need to do in order to simulate a pulsar is to initialize our signal. This will be done for a filterbank-style signal class. This type of signal needs parameters first though. One needs to enter the number of frequency channels the signal should be recorded with, what the bandwidth of the signal is, what the center frequency of the signal is, and how quickly it should record the data, or the sampling rate. To make single pulses, we also need to set the 'fold' flag to False (the default is True).

For this example, we will simulate single pulses from a 350 MHz observation from the Green Bank Telescope.

```
# Define our signal variables.
f0 = 820 # center observing frequecy in MHz
bw = 200.0 # observation MHz
Nf = 128 # number of frequency channels
f_samp = 0.001526 # sample rate of data in MHz (here 0.6554 ms for size purposes)
# Now we define our signal
signal_1 = pss.signal.FilterBankSignal(fcent = f0, bandwidth = bw, Nsubband=Nf, fold␣
↪= False)
```

The Pulsar

Next we define a pulsar object. The pulsar needs a pulse shape though. There are a number of ways to define this in the pulsar signal simulator, but here we will make a simple, predefined Gaussian profile. The Guassian needs three parameters, an amplitude, a width (or sigma), and a peak, the center of the Gaussian in phase space (e.g. 0-1).

```
# We define the Guassian profile
gauss_prof = pss.pulsar.GaussProfile(peak = 0.5, width = 0.05, amp = 1.0)
```

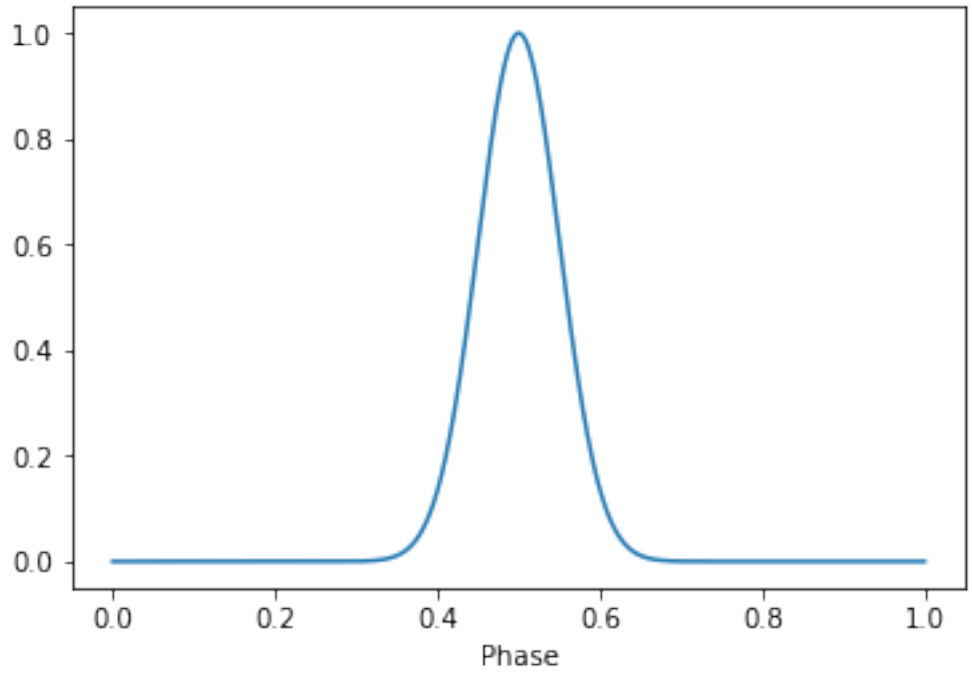
Defining the profile just tells the simulator how to make the pulses. If we want to see what they look like, we need to initialize the profile, and then we can give it a number of phase bins and plot it.

```
# We want to use 2048 phase bins and just one frequency channel for this test.
gauss_prof.init_profiles(2048, Nchan = 1)
```

```
# We can look at the shape of the profile array to make sure it matches with what we␣
↪expect
print(np.shape(gauss_prof.profiles))
```

```
(1, 2048)
```

```
# And then we can plot the array to see what the profile looks like
plt.plot(np.linspace(0,1,2048), gauss_prof.profiles[0])
plt.xlabel("Phase")
plt.show()
plt.close()
```



Now we can define the pulsar object itself. Out pulsar needs a period (s), a mean flux (Jy), a profile, which we've defined above, and a name (e.g. JXXXX+XXXX).

```
# Define the values needed for the puslar
period = 1.0 # pulse period of our simulated pulsar, here one second
Smean = 10.0 # The mean flux of the pulsar, here 10.0 Jy (note that this is very␣
↪bright for a pulsar)
psr_name = "J0000+0000" # The name of our simulated pulsar
# Now we define the pulsar
pulsar_1 = pss.pulsar.Pulsar(period, Smean, profiles=gauss_prof, name = psr_name)
```

The ISM

Now we define the interstellar medium (ISM) properties that will affect our pulsar signal as it 'travels' from the pulsar to our telescope. The main property here is the dispersion measure, DM, which is the number of electrons along the

line of sight from us to the pulsar. These electrons will delay the pulsed emission from the pulsar, causing lower radio frequencies to arrive at the telescope later than higher radio frequencies. Here we will just define the ISM object and the DM we would like the pulsar to have.

```
# Define the dispersion measure
dm = 40.0 # pc cm^-3
# And define the ISM object, note that this class takes no initial arguements
ism_1 = pss.ism.ISM()
```

The Telescope

The last thing we need to define is the telescope object. While you can define a telescope with any properties that you like with the pulsar signal simulator, it also comes with two pre-defined telescopes: The Arecibo Telescope and the Green Bank Telescope 9GBT). We will set up the GBT as our telescope. The telescope class when set up from a predefined telescope needs no additional input.

```
tscope = pss.telescope.telescope.GBT()
```

Simulating the Signal

Now we have everything set up to actually simulate our signal, though there is one extra value we need to define: the simulated observation length (s). For size and time purposes, we will only simulate 2 seconds of observing, which amounts to just two pulse periods.

```
# define the observation length
obslen = 2.0 # seconds
```

Now we can make the pulses! This is done using the make_pulses() function of the `pulsar` object we made before. It takes just the signal object, and the observation length.

```
pulsar_1.make_pulses(signal_1, tobs = obslen)
```

Next we disperse our pulses, or propagate them through the interstellar medium. We can do that easily using the disperse() function of the ISM object. This again takes the signal object, as well as the DM value defined above.

```
ism_1.disperse(signal_1, dm)
```

```
98% dispersed in 1.399 seconds.
```

Now we need to observe the signal with our telescope. This will add radiometer noise from the telescope receiver and backend to the signal. This is done using the observe() function of the telescope object, which takes the signal, the pulsar, the system name (for the GBT telescope this is either '820_GUPPI' or 'Lband_GUPPI'), and make sure that the noise variable is set to 'True'.

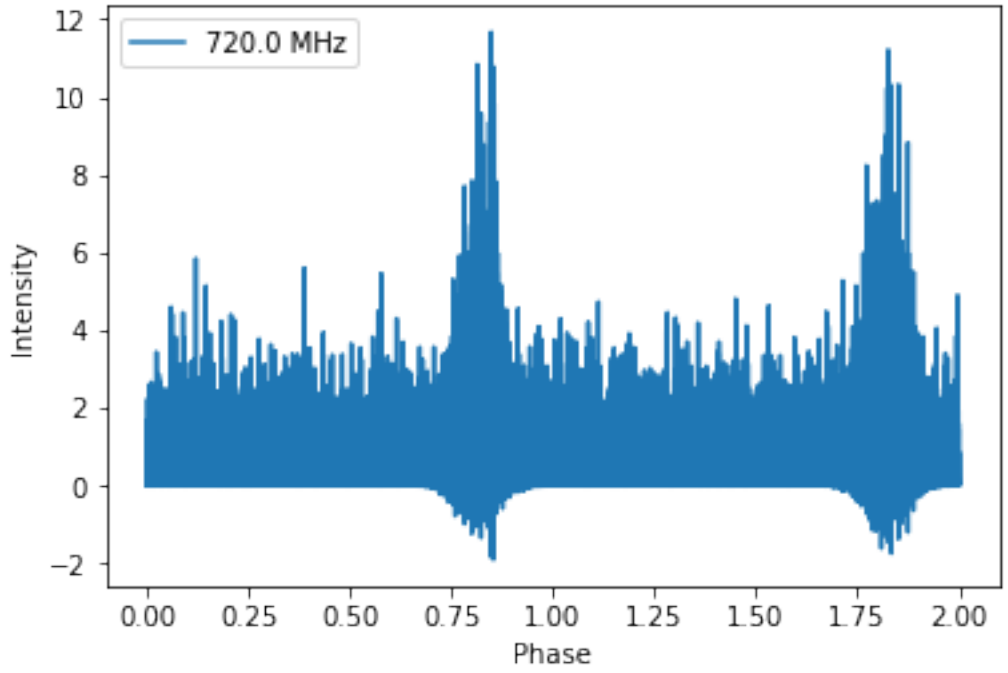
```
tscope.observe(signal_1, pulsar_1, system="820_GUPPI", noise=True)
```

```
WARNING: AstropyDeprecationWarning: The truth value of a Quantity is ambiguous. In␣
↪the future this will raise a ValueError. [astropy.units.quantity]
```

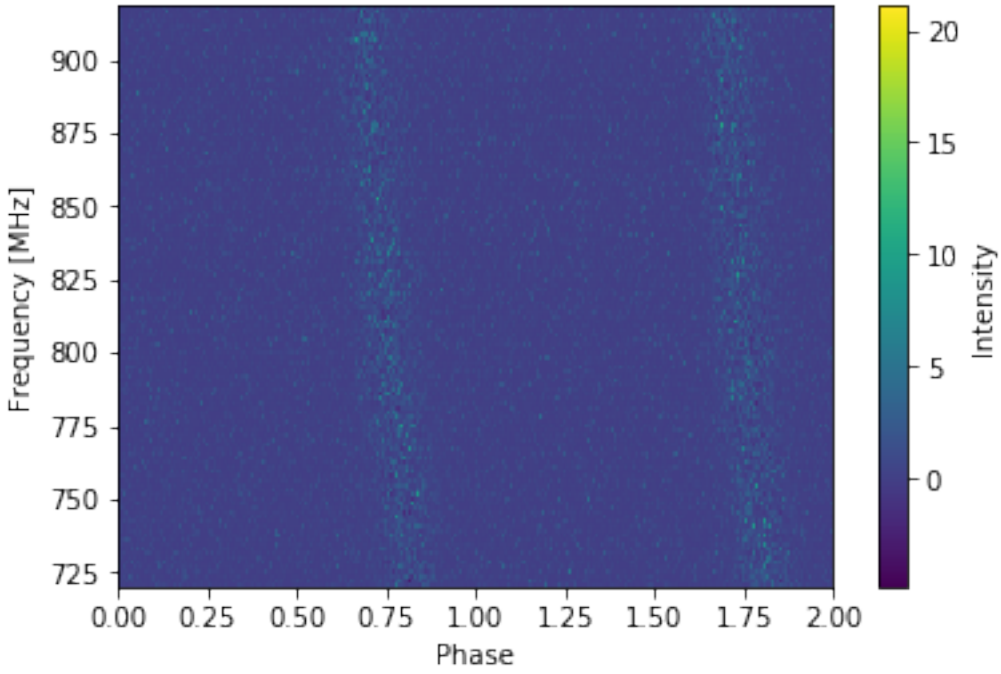
Looking at the Results

And that's all that needs to be done to simulate a signal! If you want to view the simulated signal, you can access the full data array through `signal_1.data`. Two ways to look at the data are to just plot an individual frequency channel (a phase plot), or make a 2-D of the power as a function of the pulse phase and frequency channel (a filterbank plot), both of which are demonstrated below.

```
# Get the phases of the pulse
phases = np.linspace(0, obslen/period, len(signal_1.data[0,:]))
# Plot just the pulses in the first frequency channels
plt.plot(phases, signal_1.data[0,:], label = signal_1.dat_freq[0])
plt.ylabel("Intensity")
plt.xlabel("Phase")
plt.legend(loc = 'best')
plt.show()
plt.close()
```




```
# Make the 2-D plot of intensity v. frequency and pulse phase. You can see the slight␣
↪dispersive sweep here.
plt.imshow(signal_1.data, aspect = 'auto', interpolation='nearest', origin = 'lower',␣
↪\
           extent = [min(phases), max(phases), signal_1.dat_freq[0].value, signal_1.
↪dat_freq[-1].value])
plt.ylabel("Frequency [MHz]")
plt.xlabel("Phase")
plt.colorbar(label = "Intensity")
plt.show()
plt.close()
```



Note: This tutorial was generated from a Jupyter notebook that can be downloaded here.

1.5.5 Fold Mode: Introductory Tutorial 2

This notebook will build on the first tutorial, showing more features of the PsrSigSim. Details will be given for new features, while other features have been discussed in the previous tutorial notebook. This notebook shows the details of using the fold-mode, or simulation of subintegrated data. This may be useful for simulating pulsar timing style data, as a way to save disk space and computation time. For Pulsar Timing Arrays, and most long term monitoring of millisecond pulsars, this is the preferred mode.

The example we use here is for simulating precision pulsar timing data. Instead of simulating the single pulses from a pulsar and then folding them to obtain a high signal-to-noise pulse profiles to use for precision pulsar timing, we can simulate a pre-folded observation.

```
# import some useful packages
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

# import the pulsar signal simulator
#import psrsigsim as pss
import psrsigsim as pss
```

Setting up the Folded Signal

Here we will again set up the signal class, but this time we will add some additional flags, namely the `fold`, `sample_rate`, and `sublen` flags. Setting `fold=True` tells the simulator that we want to simulate folded data,

with subinegration lengths of `sublen=X` where `X` is some number of seconds. We will set the sample rate such that we will simulate 2048 samples across the pulse period. A pulse period of 10 ms is used for this simulation.

We will simulate a 20 minute long observation total, with subintegrations of 1 minute. The other simulation parameters will be 64 frequency channels each 12.5 MHz wide (for 800 MHz bandwidth) observed with the Green Bank Telescope at L-band (1500 MHz center frequency).

```
# Define our signal variables.
f0 = 1500 # center observing frequecy in MHz
bw = 800.0 # observation MHz
Nf = 64 # number of frequency channels
# We define the pulse period early here so we can similarly define the frequency
period = 0.010 # pulsar period in seconds
f_samp = (1.0/period)*2048*10**-6 # sample rate of data in MHz (here 2048 samples␣
↪across the pulse period
sublen = 60.0 # subintegration length in seconds, or rate to dump data at
# Now we define our signal
signal_fold = pss.signal.FilterBankSignal(fcent = f0, bandwidth = bw, Nsubband=Nf,␣
↪sample_rate = f_samp,
                                          sublen = sublen, fold = True) # fold is set to␣
↪`True`
```

```
Warning: specified sample rate 0.20479999999999998 MHz < Nyquist frequency 1600.0 MHz
```

Pulsar, ISM and Telescope

Here we set up `Pulsar`, `ISM` and `telescope` objects in the same way as in the previous tutorial. We will again use a basic Gaussian profile, but will have a more realisitic mean flux of 5 mJy (or 0.005 Jy).

```
# We define the Guassian profile
gauss_prof = pss.pulsar.GaussProfile(peak = 0.5, width = 0.05, amp = 1.0)

# Define the values needed for the puslar
Smean = 0.005 # The mean flux of the pulsar, here 0.005 Jy
psr_name = "J0000+0000" # The name of our simulated pulsar
# Now we define the pulsar
pulsar_fold = pss.pulsar.Pulsar(period, Smean, profiles=gauss_prof, name = psr_name)

# Define the dispersion measure
dm = 40.0 # pc cm^-3
# And define the ISM object, note that this class takes no initial arguements
ism_fold = pss.ism.ISM()

tscope = pss.telescope.telescope.GBT()
```

Simulating the Signal

Now we will simulate the signal. Here the commands are the same as before, we just need to define an observation length (20 minutes), make the pulses with the pulsar, disperse the data, And then observe the pulsar with our telescope. For the telescope, we will use the `Lband_GUPPI` system predefined by in the GBT telescope class.

```
# define the observation length
obslen = 60.0*20 # seconds, 20 minutes in total
```

```
# Make the pulses
pulsar_fold.make_pulses(signal_fold, tobs = obslen)
```

```
# Disperse the data
ism_fold.disperse(signal_fold, dm)
```

```
98% dispersed in 0.154 seconds.
```

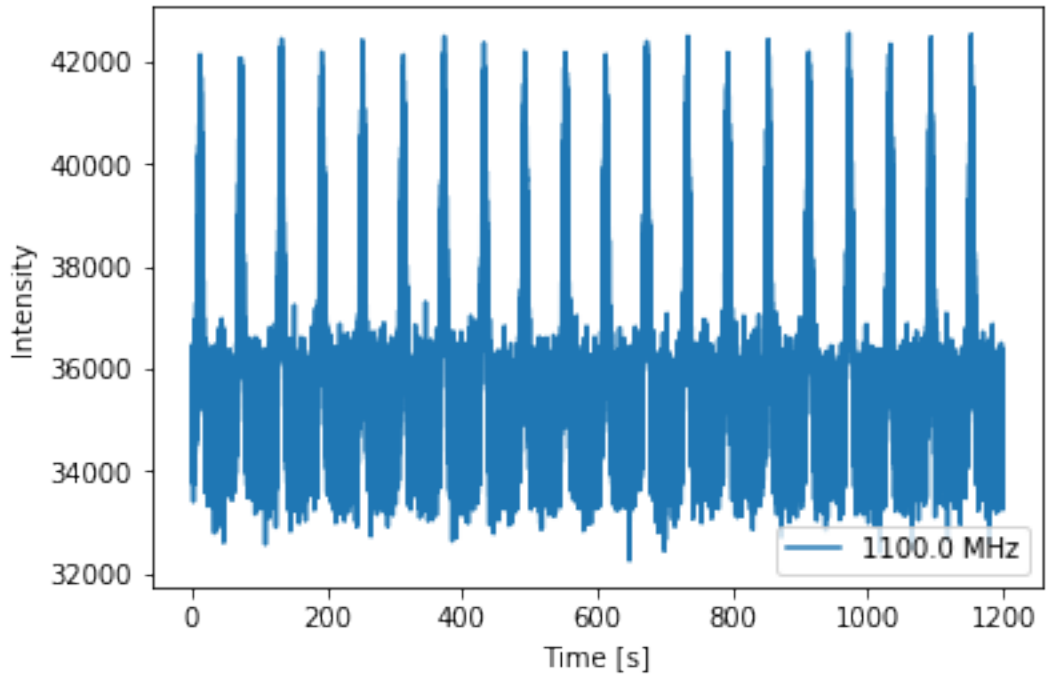
```
# Observe with the telescope
tscope.observe(signal_fold, pulsar_fold, system="Lband_GUPPI", noise=True)
```

```
WARNING: AstropyDeprecationWarning: The truth value of a Quantity is ambiguous. In␣
↪the future this will raise a ValueError. [astropy.units.quantity]
```

Visualizing the Data

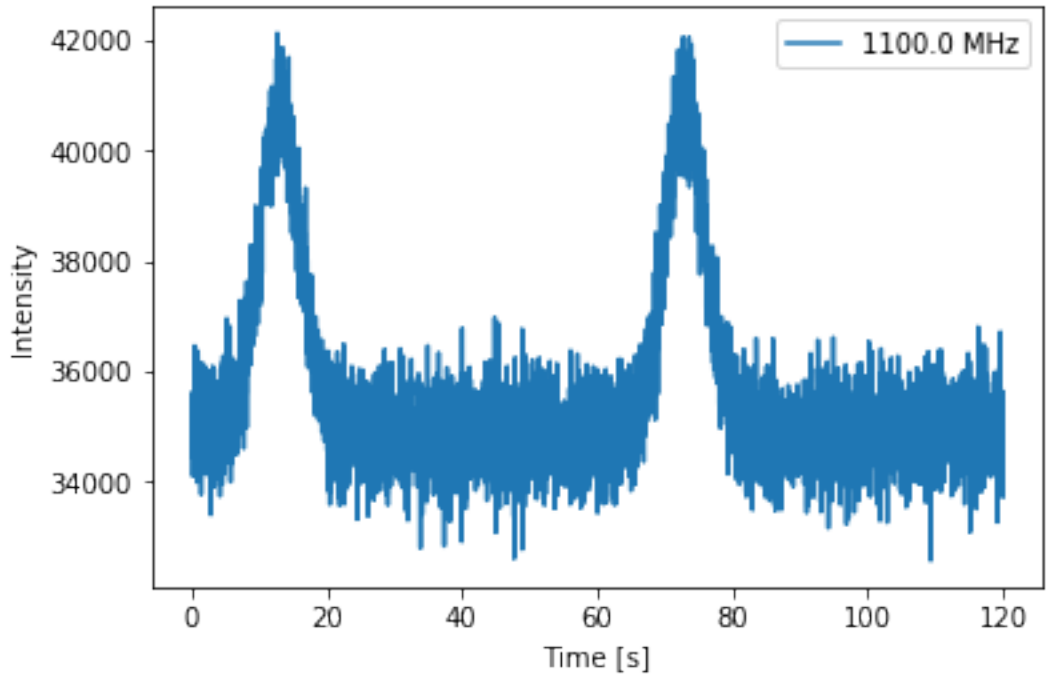
Now that we've simuluated the signal, we can take a look at the subintegrated data that we have produced. We can access it the same way as described in the previous tutorial.

```
# Get the phases of the pulse
time = np.linspace(0, obslen, len(signal_fold.data[0,:]))
# Plot just the pulses in the first frequency channels
plt.plot(time, signal_fold.data[0,:], label = signal_fold.dat_freq[0])
plt.ylabel("Intensity")
plt.xlabel("Time [s]")
plt.legend(loc = 'best')
plt.show()
plt.close()
```



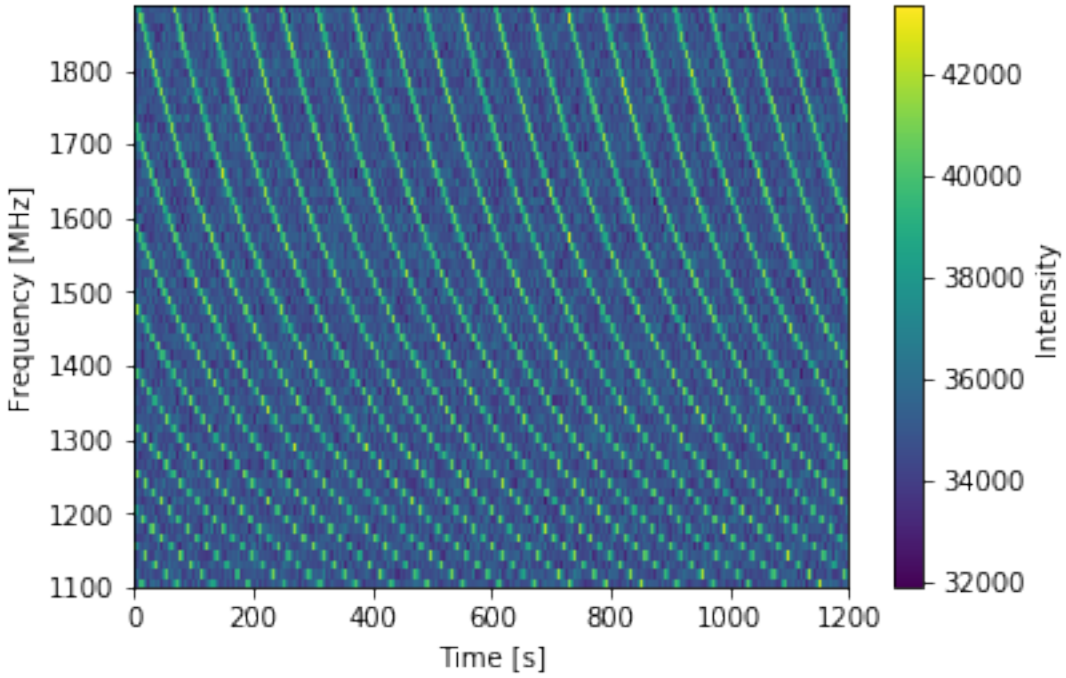
If we zoom in on just the first two pulse periods. . .

```
# Since we know there are 2048 bins per pulse period, we can index the appropriate
↪amount
plt.plot(time[:4096], signal_fold.data[0,:4096], label = signal_fold.dat_freq[0])
plt.ylabel("Intensity")
plt.xlabel("Time [s]")
plt.legend(loc = 'best')
plt.show()
plt.close()
```



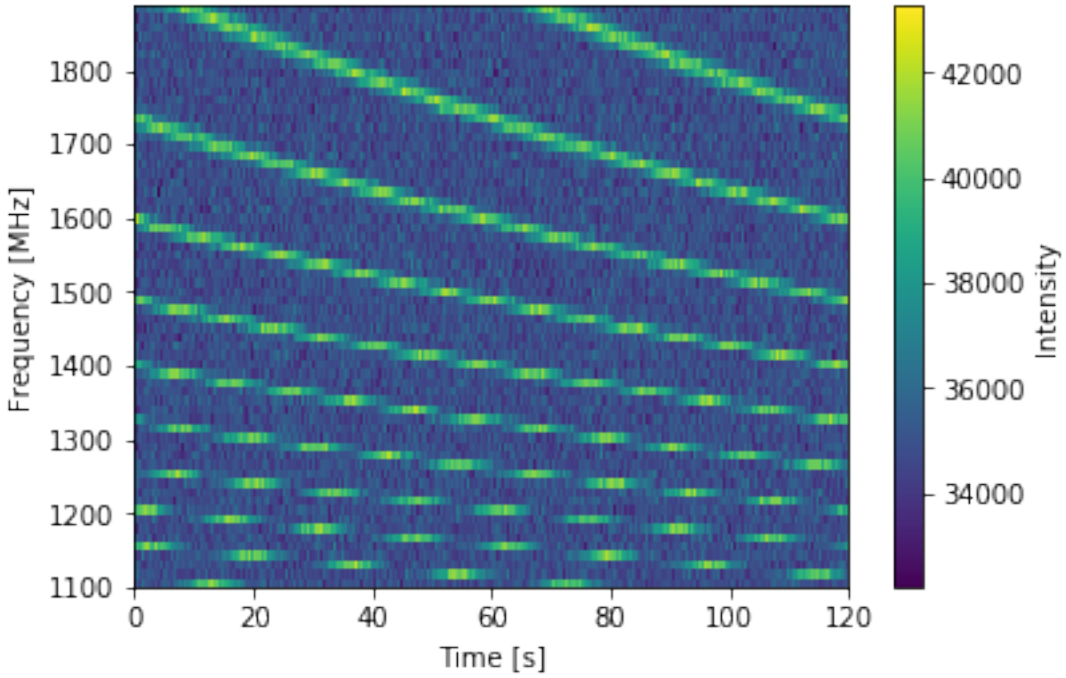
We can clearly see the pulse profile above the noise level now. By making subintegrated data, we build up the signal of the simulated pulses to be easily visible. With a 10 ms period and 1 minute subintegrations, each of these pulses acts as if we have folded (1 minutes / 10 ms) = 6000 pulses together. We can look at the 2D spectrogram of these pulses as well.

```
# Make the 2-D plot of intensity v. frequency and pulse phase. You can see the slight
↪dispersive sweep here.
plt.imshow(signal_fold.data, aspect = 'auto', interpolation='nearest', origin = 'lower
↪', \
           extent = [min(time), max(time), signal_fold.dat_freq[0].value, signal_fold.
↪dat_freq[-1].value])
plt.ylabel("Frequency [MHz]")
plt.xlabel("Time [s]")
plt.colorbar(label = "Intensity")
plt.show()
plt.close()
```



The pulse and dispersive sweep is clearly visible with the high signal-to-noise ratio. Again, zooming in on the first two subintegrations...

```
plt.imshow(signal_fold.data[:,:4096], aspect = 'auto', interpolation='nearest',␣
↪origin = 'lower', \
           extent = [min(time[:4096]), max(time[:4096]), signal_fold.dat_freq[0].
↪value, signal_fold.dat_freq[-1].value])
plt.ylabel("Frequency [MHz]")
plt.xlabel("Time [s]")
plt.colorbar(label = "Intensity")
plt.show()
plt.close()
```



Here the dispersion clearly shows that the pulses are dispersed for over two minutes across the observing bandwidth.

Note: This tutorial was generated from a Jupyter notebook that can be downloaded here.

1.5.6 Pulse Profiles: Introductory Tutorial 3

This notebook will build on the previous tutorials, showing more features of the `PsrSigSim`. Details will be given for new features, while other features have been discussed in the previous tutorial notebook. This notebook shows the details of different methods of defining pulse profiles for simulated pulsars. This is useful for simulating realisitic pulse profiles and pulse profile evolution (with observing frequency.)

We again simulate precision pulsar timing data with high signal-to-noise pulse profiles in order to clearly show the input pulse profile in the final simulated data product.

```
# import some useful packages
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

# import the pulsar signal simulator
import psrsigsim as pss
```

Setting up the Folded Signal

Here we will again set up the folded signal class as in the second introductory tutorial. We will again simulate a 20 minute long observation total, with subintegrations of 1 minute. The other simulation parameters will be 64 frequency channels each 12.5 MHz wide (for 800 MHz bandwidth) observed with the Green Bank Telescope at L-band (1500 MHz center frequency).

However, as part of this tutorial, we will simulate a real pulsar, J1713+0747, as we have a premade profile for this pulsar. The period, dm, and other relavent pulsar parameters come from the NANOGrav 11-yr data release.

```
# Define our signal variables.
f0 = 1500 # center observing frequecy in MHz
bw = 800.0 # observation MHz
Nf = 64 # number of frequency channels
# We define the pulse period early here so we can similarly define the frequency
period = 0.00457 # pulsar period in seconds for J1713+0747
f_samp = (1.0/period)*2048*10**-6 # sample rate of data in MHz (here 2048 samples␣
↪across the pulse period
sublen = 60.0 # subintegration length in seconds, or rate to dump data at
# Now we define our signal
signal_1713 = pss.signal.FilterBankSignal(fcent = f0, bandwidth = bw, Nsubband=Nf,␣
↪sample_rate = f_samp,
                                          sublen = sublen, fold = True) # fold is set to␣
↪`True`
```

```
Warning: specified sample rate 0.4481400437636761 MHz < Nyquist frequency 1600.0 MHz
```

The ISM and Telescope

Here we set up `ISM` and `telescope` objects in the same way as in the previous tutorial. Since we can set these up in any order, we will do these first to better show the different pulse profiles later.

```
# Define the dispersion measure
dm =  15.921200 # pc cm^-3
# And define the ISM object, note that this class takes no initial arguements
ism_fold = pss.ism.ISM()

# We intialize the telescope object as the Green Bank Telescope
tscope = pss.telescope.telescope.GBT()
```

Pulse Profiles

In previous tutorials, we have defined a very simple Gaussian profile as the pulse profile. However, the `PsrSigSim` allows users to define profiles in a few different ways, including multiple Gaussians, a user input profile in the form of a Python array, and two dimensional versions of the pulse profiles called pulse portaits.

We will go through a few different ways to set up the pulse profiles, and then will simulate different initial pulsars and the subsequent data, through the full pipeline.

Gaussian Profiles

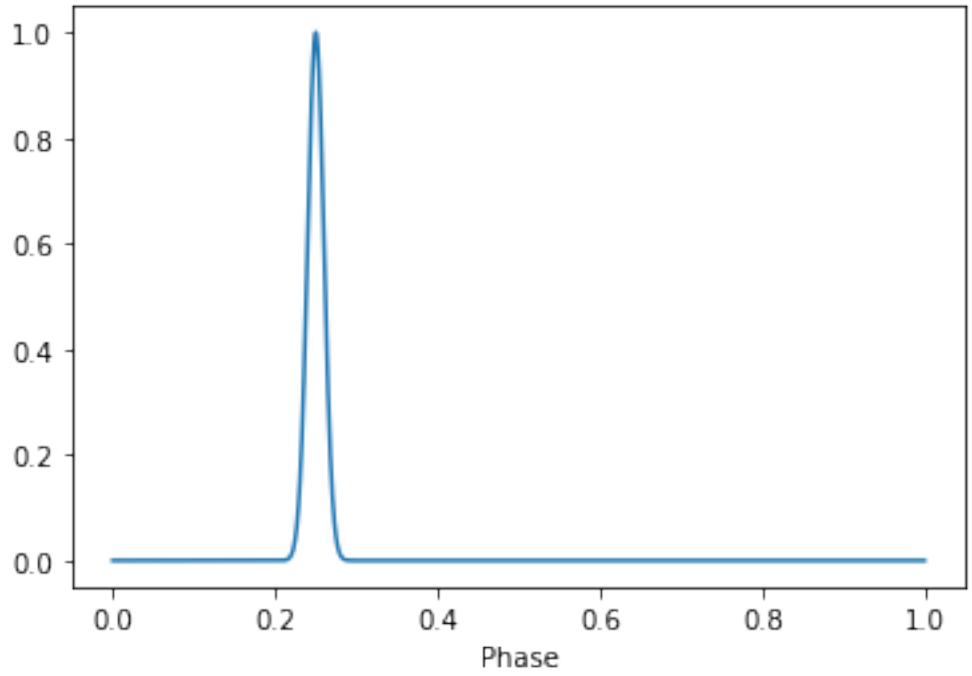
The first method is the Gaussian profile, which has been demonstrated in previous tutorials. The Guassian needs three parameters, an amplitude, a width (or sigma), and a peak, the center of the Gaussian in phase space (e.g. 0-1). The simplest profile that can be defined is a single Gauassian.

```
gauss_prof = pss.pulsar.GaussProfile(peak = 0.25, width = 0.01, amp = 1.0)
```

Defining the profile just tells the simulator how to make the pulses. If we want to see what they look like, we need to initialize the profile, and then we can give it a number of phase bins and plot it.

```
# We want to use 2048 phase bins and just one frequency channel for this test.
gauss_prof.init_profiles(2048, Nchan = 1)
```

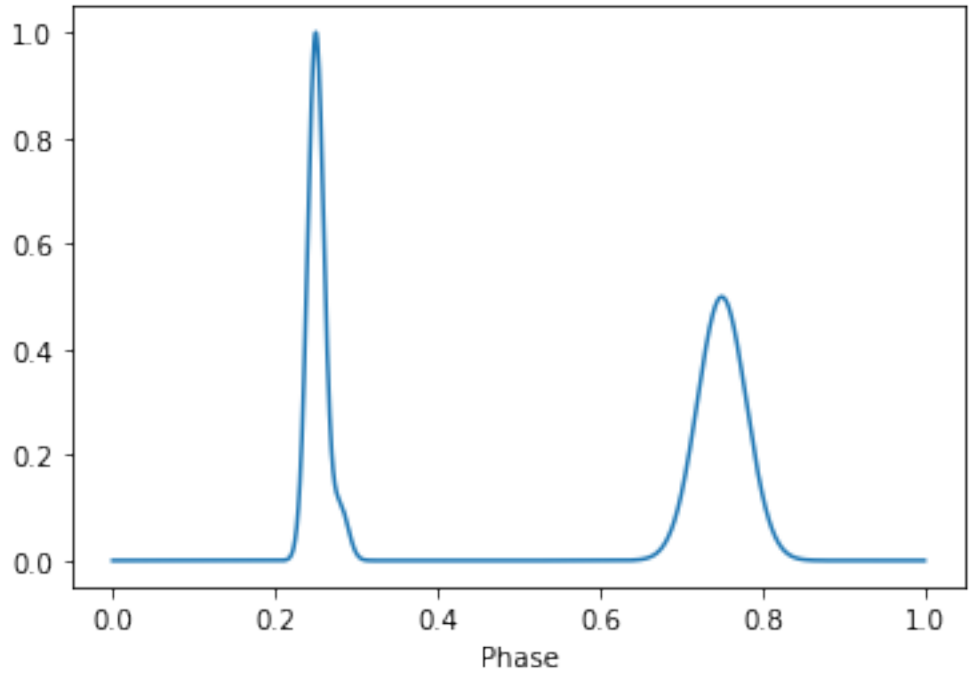
```
# And then we can plot the array to see what the profile looks like
plt.plot(np.linspace(0,1,2048), gauss_prof.profiles[0])
plt.xlabel("Phase")
plt.show()
plt.close()
```



However the Gaussian profile can also be used to make a pulse profile with multiple Gaussian components. Instead of inputting a single value into each of the three fields (peak, width, amp), we input an array of the corresponding values, e.g. the second value in each array are the components of the second Gaussian component. Below we build on the previous single Gaussian profile by adding a small "shoulder" to the main pulse profile, as well as a broad interpulse to the profile.

Note - curently the input for multiple Gaussian components must be an array, it cannot be a list.

```
# Define the number and value of each Gaussain component
peaks = np.array([0.25, 0.28, 0.75])
widths = np.array([0.01, 0.01, 0.03])
amps = np.array([1.0, 0.1, 0.5])

# Define the profile using multiple Gaussians
mulit_gauss_prof = pss.pulsar.GaussProfile(peak = peaks, width = widths, amp = amps)
# We want to use 2048 phase bins and just one frequency channel for this test.
mulit_gauss_prof.init_profiles(2048, Nchan = 1)
# And then we can plot the array to see what the profile looks like
plt.plot(np.linspace(0,1,2048), mulit_gauss_prof.profiles[0])
plt.xlabel("Phase")
plt.show()
plt.close()
```

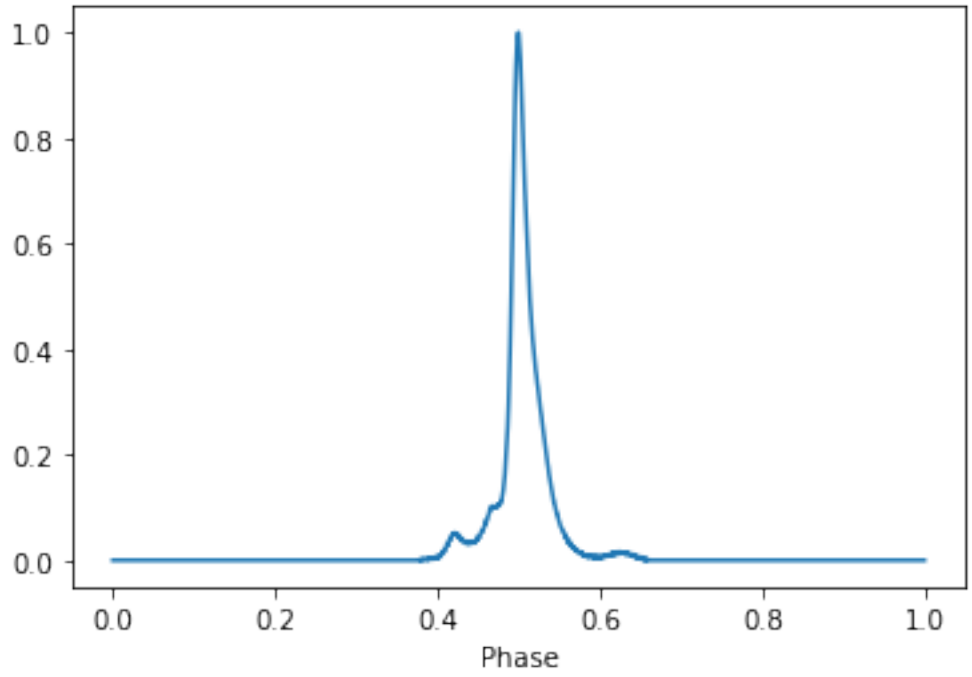


Data Profiles

The PsrSigSim can also take arrays of data points as the pulse profile in what is called a `DataProfile`. This array represents pulse profile and may be used to define the pulse profile shape. The number of bins in the input data profile does not need to be the equivalent to the input sampling rate. This option may be useful when simulating real pulsars or realistic pulsar data.

Here we will use a premade profile of the pulsar J1713+0747 as the pulse profile.

```
# First we load the data array
path = 'psrsigsim/data/J1713+0747_profile.npy'
J1713_dataprof = np.load(path)

# Now we define the data profile
J1713_prof = pss.pulsar.DataProfile(J1713_dataprof)
# Now we can initialize and plot the profile the same way as the Gaussian profile
J1713_prof.init_profiles(2048, Nchan = 1)
# And then we can plot the array to see what the profile looks like
plt.plot(np.linspace(0,1,2048), J1713_prof.profiles[0])
plt.xlabel("Phase")
plt.show()
plt.close()
```



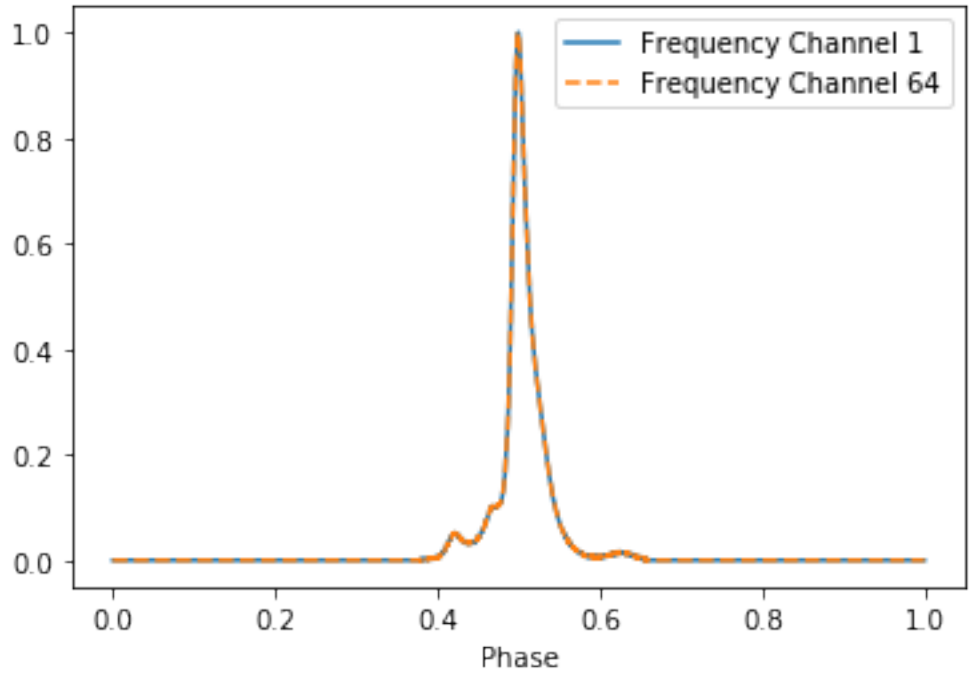
Data Portraits

While the `Profile` objects initialize a 1-D pulse profile, there are also `Portrait` objects that have the ability to initialize a 2-D pulse profile. A `Profile` object will use the same pulse profile for every simulated frequency channel, while a `Portrait` can use different versions of the profile at different frequencies.

To illustrate this, we will initialize a pulse `Portrat` for J1713+0747 where they are scaled in power. We start by showing how a pulse `Profile` uses the same profile at every frequency, then how a `Portrait` is initialized, and finally, how different profiles may be input at each frequency using a pulse `Portrait`.

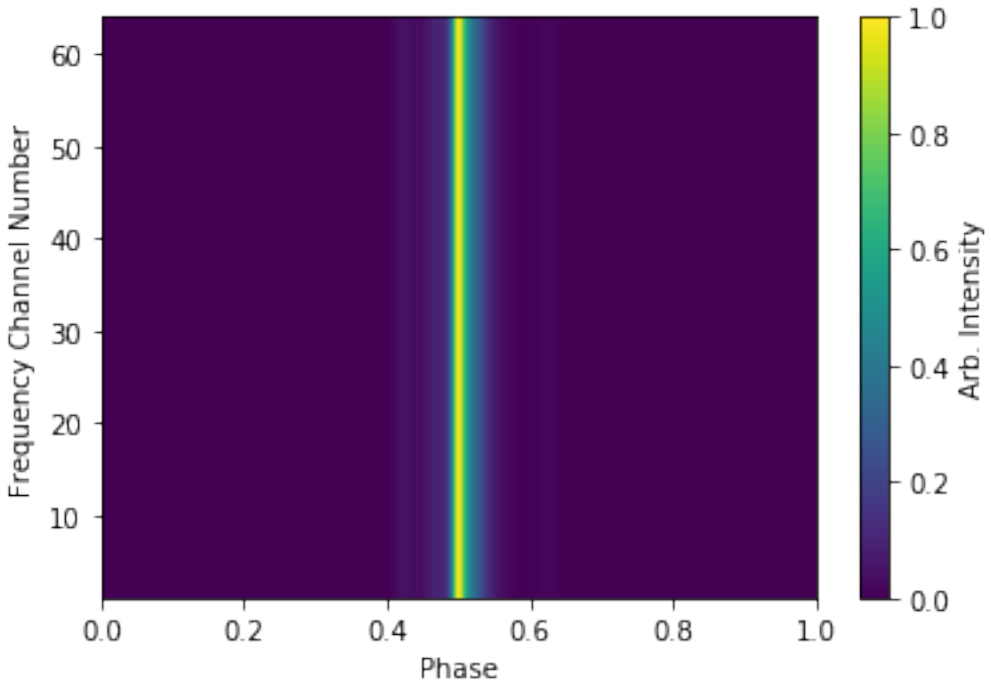
Using the same profile as above, we will initialize a multi-frequency profile, and show that it has the same shape and power at different frequencies.

```
# Initialize a multi-channel profile
J1713_prof.init_profiles(2048, Nchan = 64)
# And then we can plot the array to see what the profile looks like
plt.plot(np.linspace(0,1,2048), J1713_prof.profiles[0], label = "Frequency Channel 1")
plt.plot(np.linspace(0,1,2048), J1713_prof.profiles[-1], ls = '--', label =
↪"Frequency Channel 64")
plt.xlabel("Phase")
plt.legend(loc='best')
plt.show()
plt.close()
```



It is easy to see that the two profiles are identical. If we plot a 2-D image of the profile phase as a function of frequency channel, we can see that they are all identical.

```
plt.imshow(J1713_prof.profiles, aspect = 'auto', interpolation='nearest', origin =
↪'lower', \
           extent = [0.0, 1.0, 1, 64])
plt.ylabel("Frequency Channel Number")
plt.xlabel("Phase")
plt.colorbar(label = "Arb. Intensity")
plt.show()
plt.close()
```



We can similarly initialize a pulse `Portait`. Here we will first create a mulitdimensional array of pulse profile, as well as an array to scale them by. We will then initialize a pulse `Portrait` object and show that the profiles generated retain the scaling.

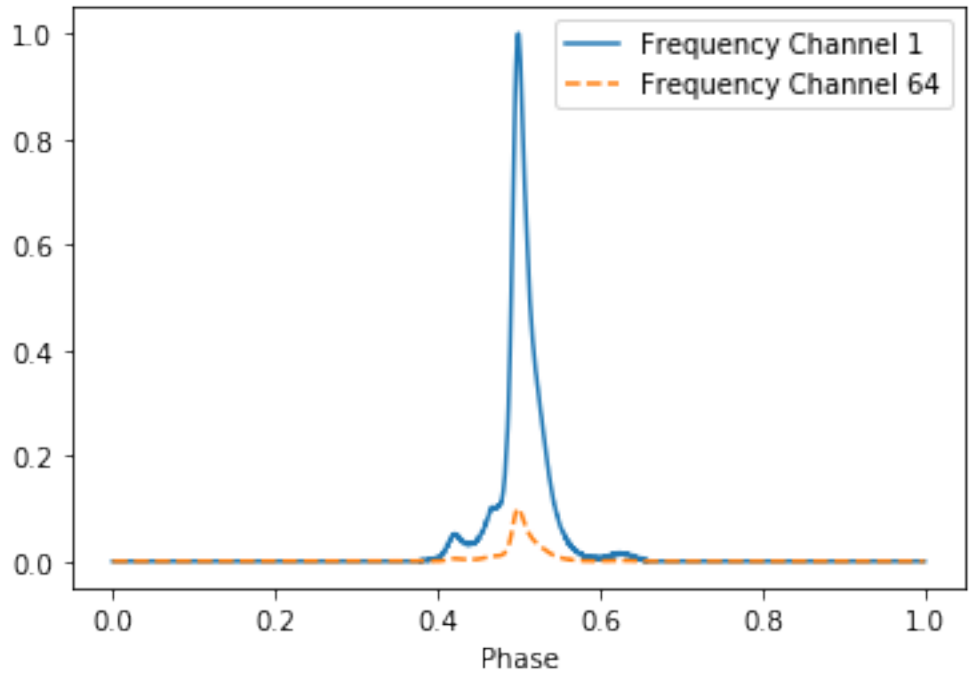
```
# Make a 2-D array of the profiles
J1713_dataprof_2D = np.tile(J1713_dataprof, (64,1))
# Now we scale them linearly so that lower frequency channels are "brighter"
scaling = np.reshape(np.linspace(1.0, 0.1, 64), (64,1))
J1713_dataprof_2D *= scaling
# Now we make a `Portrait`
J1713_prof_2D = pss.pulsar.DataPortrait(J1713_dataprof_2D)
# Now we initialize the profiles
J1713_prof_2D.init_profiles(2048, 64)
```

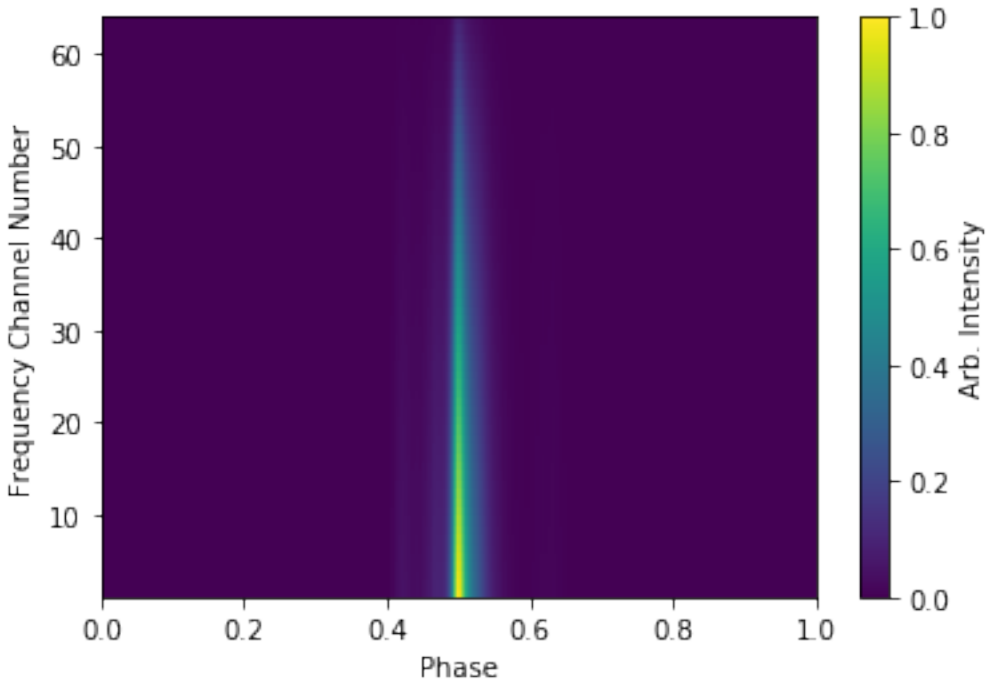
```
# Now we can plot the first and last profile, as well as the 2-D power of the input␣
↪profiles at each frequency
# And then we can plot the array to see what the profile looks like
plt.plot(np.linspace(0,1,2048), J1713_prof_2D.profiles[0], label = "Frequency Channel␣
↪1")
plt.plot(np.linspace(0,1,2048), J1713_prof_2D.profiles[-1], ls = '--', label =
↪"Frequency Channel 64")
plt.xlabel("Phase")
plt.legend(loc='best')
plt.show()
plt.close()
# And the 2-D image
plt.imshow(J1713_prof_2D.profiles, aspect = 'auto', interpolation='nearest', origin =
↪'lower', \
           extent = [0.0, 1.0, 1, 64])
plt.ylabel("Frequency Channel Number")
plt.xlabel("Phase")
plt.colorbar(label = "Arb. Intensity")
```

(continues on next page)

(continued from previous page)

```
plt.show()
plt.close()
```





We can see that the generated profiles then retain the scaling they have been given. This is just a simplistic version of what can be done, using the `Portrait` class.

Pulsars

Now we will set up a few different `Pulsar` classes and simulate the full dataset, showing how the input profiles are retained through the process of dispersion and adding noise to the simulated data.

```
# Define the values needed for the puslar
Smean = 0.009 # The mean flux of the pulsar, J1713+0747 at 1400 MHz from the ATNF␣
↪pulsar catatlog, here 0.009 Jy
psr_name_1 = "J0000+0000" # The name of our simulated pulsar with a mulit-gaussian␣
↪profile
psr_name_2 = "J1713+0747" # The name of our simulated pulsar with a scaled, 2-D␣
↪profile

# Now we define the pulsar with multiple Gaussian defineing is profile
pulsar_mg = pss.pulsar.Pulsar(period, Smean, profiles=mulit_gauss_prof, name = psr_
↪name_1)

# Now we define the pulsar with the scaled J1713+0747 profiles
pulsar_J1713 = pss.pulsar.Pulsar(period, Smean, profiles=J1713_prof_2D, name = psr_
↪name_2)
```

Simulations

We run the rest of the simulation, including dispersion and "observing" with our telescope. The same parameters are used for both `Pulsars` and simulated data sets with the only difference being the input profiles. We then show the resutls of each simulation and how they retain the intial input profile shapes.

We first run the simultion for our fake mulit-gaussian profile pulsar.

```
# define the observation length
obslen = 60.0*20 # seconds, 20 minutes in total
# Make the pulses
pulsar_mg.make_pulses(signal_1713, tobs = obslen)
# Disperse the data
ism_fold.disperse(signal_1713, dm)
# Observe with the telescope
tscope.observe(signal_1713, pulsar_mg, system="Lband_GUPPI", noise=True)
```

```
98% dispersed in 0.122 seconds.
```

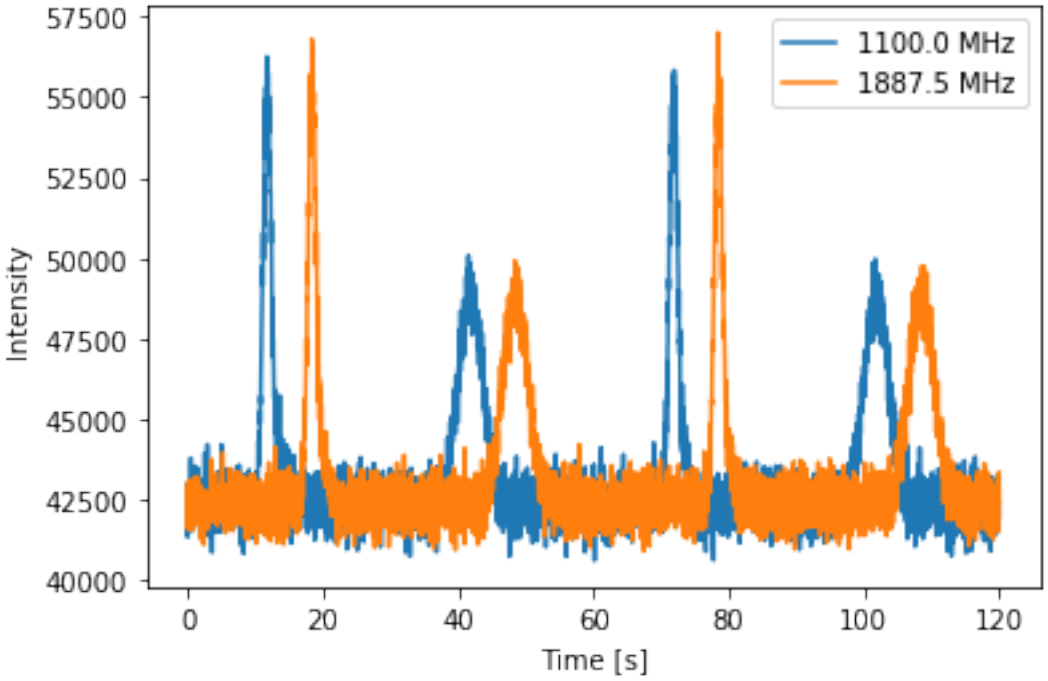
```
WARNING: AstropyDeprecationWarning: The truth value of a Quantity is ambiguous. In␣
↪the future this will raise a ValueError. [astropy.units.quantity]
```

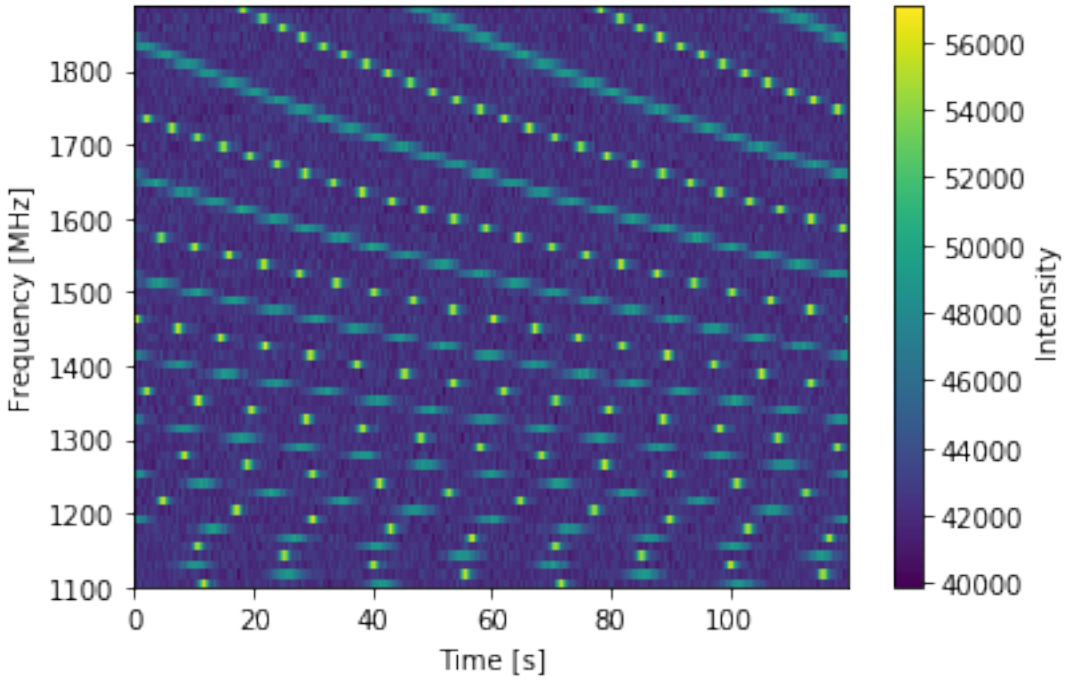
```
# Now we plot these profiles
# Get the phases of the pulse
time = np.linspace(0, obslen, len(signal_1713.data[0,:]))
# Since we know there are 2048 bins per pulse period, we can index the appropriate␣
↪amount
plt.plot(time[:4096], signal_1713.data[0,:4096], label = signal_1713.dat_freq[0])
plt.plot(time[:4096], signal_1713.data[-1,:4096], label = signal_1713.dat_freq[-1])
plt.ylabel("Intensity")
plt.xlabel("Time [s]")
plt.legend(loc = 'best')
plt.show()
plt.close()
```

(continues on next page)

(continued from previous page)

```
# And the 2-D plot
plt.imshow(signal_1713.data[:,:4096], aspect = 'auto', interpolation='nearest',␣
↪origin = 'lower', \
           extent = [min(time[:4096]), max(time[:4096]), signal_1713.dat_freq[0].
↪value, signal_1713.dat_freq[-1].value])
plt.ylabel("Frequency [MHz]")
plt.xlabel("Time [s]")
plt.colorbar(label = "Intensity")
plt.show()
plt.close()
```





It is clear that we have maintianed the initial shape of this profile. Now we will do the same thing but with the scaled 2-D pulse `Portrait` pulsar.

```
# We first must redefine the input signal
signal_1713 = pss.signal.FilterBankSignal(fcent = f0, bandwidth = bw, Nsubband=Nf,␣
↪sample_rate = f_samp,
                                          sublen = sublen, fold = True) # fold is set to␣
↪`True`

# define the observation length
obslen = 60.0*20 # seconds, 20 minutes in total
# Make the pulses
pulsar_J1713.make_pulses(signal_1713, tobs = obslen)
# Disperse the data
ism_fold.disperse(signal_1713, dm)
# Observe with the telescope
tscope.observe(signal_1713, pulsar_J1713, system="Lband_GUPPI", noise=True)
```

```
Warning: specified sample rate 0.4481400437636761 MHz < Nyquist frequency 1600.0 MHz
98% dispersed in 0.115 seconds.
```

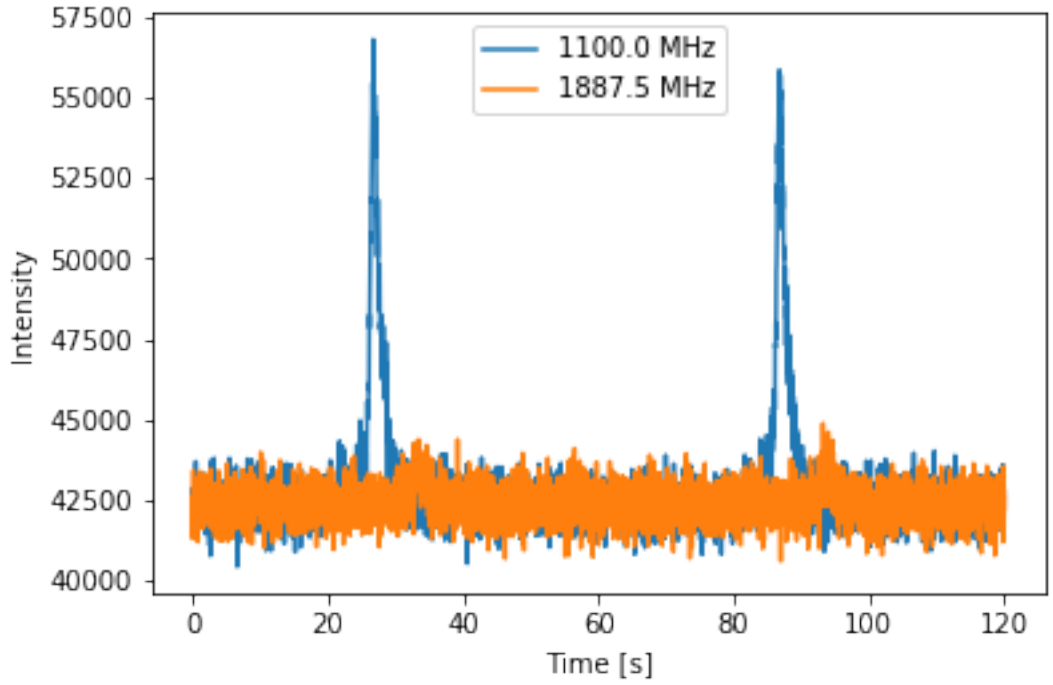
```
WARNING: AstropyDeprecationWarning: The truth value of a Quantity is ambiguous. In␣
↪the future this will raise a ValueError. [astropy.units.quantity]
```

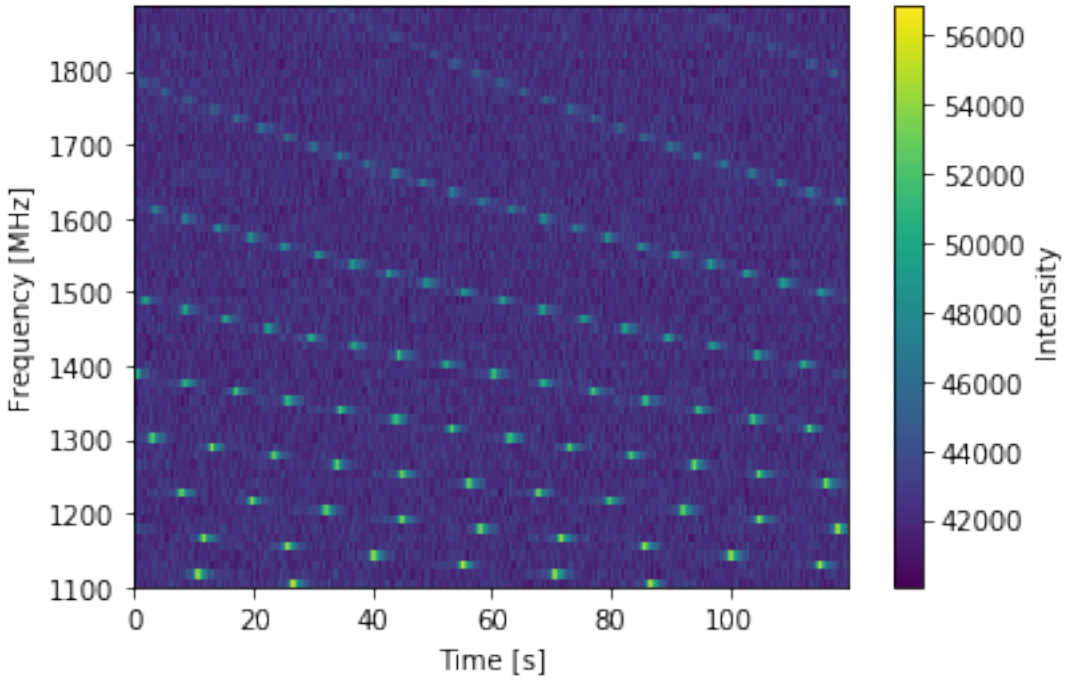
```
# Now we plot these profiles
# Get the phases of the pulse
time = np.linspace(0, obslen, len(signal_1713.data[0,:]))
# Since we know there are 2048 bins per pulse period, we can index the appropriate␣
↪amount
plt.plot(time[:4096], signal_1713.data[0,:4096], label = signal_1713.dat_freq[0])
plt.plot(time[:4096], signal_1713.data[-1,:4096], label = signal_1713.dat_freq[-1])
plt.ylabel("Intensity")
```

(continues on next page)

(continued from previous page)

```
plt.xlabel("Time [s]")
plt.legend(loc = 'best')
plt.show()
plt.close()

# And the 2-D plot
plt.imshow(signal_1713.data[:,:4096], aspect = 'auto', interpolation='nearest',␣
↪origin = 'lower', \
           extent = [min(time[:4096]), max(time[:4096]), signal_1713.dat_freq[0].
↪value, signal_1713.dat_freq[-1].value])
plt.ylabel("Frequency [MHz]")
plt.xlabel("Time [s]")
plt.colorbar(label = "Intensity")
plt.show()
plt.close()
```





Here it is clear the the scaling has also been maintained, with lower frequency pulses being brighter than high frequency pulses.

Note: This tutorial was generated from a Jupyter notebook that can be downloaded here.

1.5.7 ISM Delays: Tutorial 4

This notebook will build on the previous tutorials, showing more features of the `PsrSigSim`. Details will be given for new features, while other features have been discussed in the previous tutorial notebook. This notebook shows the details of different delays and effects due to the interstellar medium (ISM) that can be added to the simulated data.

We again simulate precision pulsar timing data with high signal-to-noise pulse profiles in order to clearly show the input pulse profile in the final simulated data product.

```
# import some useful packages
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

# import the pulsar signal simulator
import psrsigsim as pss
```

Setting up the Folded Signal

Here we will again set up the folded signal class as in previous introductory tutorials. We will again simulate a 20 minute long observation total, with subintegrations of 1 minute. The other simulation parameters will be 64 frequency channels each 12.5 MHz wide (for 800 MHz bandwidth) observed with the Green Bank Telescope at L-band (1500 MHz center frequency).

We will simulate a real pulsar, J1713+0747, as we have a premade profile for this pulsar. The period, dm, and other relavent pulsar parameters come from the NANOGrav 11-yr data release.

```
# Define our signal variables.
f0 = 1500 # center observing frequecy in MHz
bw = 800.0 # observation MHz
Nf = 64 # number of frequency channels
# We define the pulse period early here so we can similarly define the frequency
period = 0.00457 # pulsar period in seconds for J1713+0747
f_samp = (1.0/period)*2048*10**-6 # sample rate of data in MHz (here 2048 samples␣
↪across the pulse period
sublen = 60.0 # subintegration length in seconds, or rate to dump data at
# Now we define our signal
signal_1713 = pss.signal.FilterBankSignal(fcent = f0, bandwidth = bw, Nsubband=Nf,␣
↪sample_rate = f_samp,
                                          sublen = sublen, fold = True) # fold is set to␣
↪`True`
```

```
Warning: specified sample rate 0.4481400437636761 MHz < Nyquist frequency 1600.0 MHz
```

The Pulsar and Profiles

Now we will load the pulse profile as in Tutorial 3 and intilialize a single `Pulsar` object. We will also make the pulses now so that we can add different ISM effects to them later.

```
# First we load the data array
path = 'psrsigsim/data/J1713+0747_profile.npy'
J1713_dataprof = np.load(path)

# Now we define the data profile
J1713_prof = pss.pulsar.DataProfile(J1713_dataprof)
```

```
# Define the values needed for the puslar
Smean = 0.009 # The mean flux of the pulsar, J1713+0747 at 1400 MHz from the ATNF␣
↪pulsar catatlog, here 0.009 Jy
psr_name = "J1713+0747" # The name of our simulated pulsar

# Now we define the pulsar with the scaled J1713+0747 profiles
pulsar_J1713 = pss.pulsar.Pulsar(period, Smean, profiles=J1713_prof, name = psr_name)
```

```
# define the observation length
obslen = 60.0*20 # seconds, 20 minutes in total
# Make the pulses
pulsar_J1713.make_pulses(signal_1713, tobs = obslen)
```

The Telescope

We will set up the `telescope` object in the same way as in the previous tutorials. Since we can set these up in any order, we will do these first to better show the different `ISM` properties later.

```
# We intialize the telescope object as the Green Bank Telescope
tscope = pss.telescope.telescope.GBT()
```

The ISM

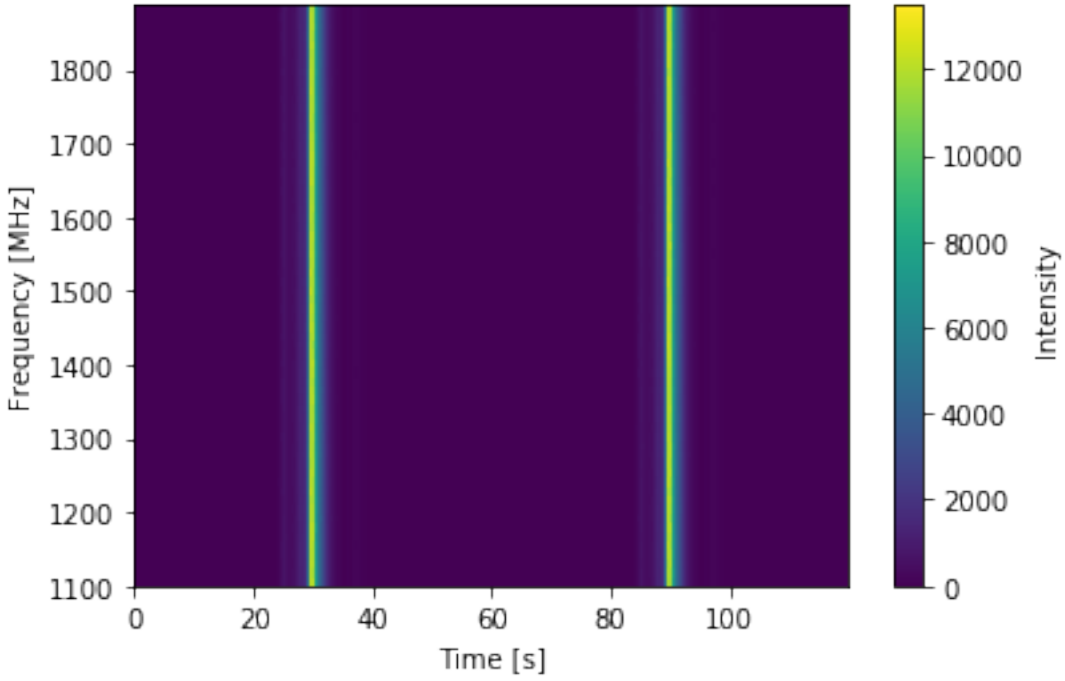
Here we will initialize the ISM class and show the various different delays that may be added to the simulated data that are due to the ISM or are specifically frequency dependent delay. In particular these include dispersion due to the ISM, delays due to pulse scatter broadening, and other frequency dependen, or "FD", parameters as defined Zhu et al. 2015 and Arzoumanian et al. 2016.

```
# Define the ISM object, note that this class takes no initial arguements
ism_sim = pss.ism.ISM()
```

Pulse Dispersion

We first show how to add dispersion of pulsars due to the ISM. This has been shown in previous tutorials as well. To do this, we first define the dispersion measure, or DM, the number of free electrons along the line of sight. This follows a frequeny^-2 relation that can be found in the Handbook of Pulsar Astronomy, by Lorimer and Kramer, 2005. The DM we use here is the same as in the NANOGrav 11-yr par file for PSR J1713+0747. We show the pulses both before and after dispersion to show the effects.

```
# We first plot the first two pulses in frequency-time space to show the undispersed␣
↪pulses
time = np.linspace(0, obslen, len(signal_1713.data[0,:]))
# And the 2-D plot
plt.imshow(signal_1713.data[:,:4096], aspect = 'auto', interpolation='nearest',␣
↪origin = 'lower', \
           extent = [min(time[:4096]), max(time[:4096]), signal_1713.dat_freq[0].
↪value, signal_1713.dat_freq[-1].value])
plt.ylabel("Frequency [MHz]")
plt.xlabel("Time [s]")
plt.colorbar(label = "Intensity")
plt.show()
plt.close()
```

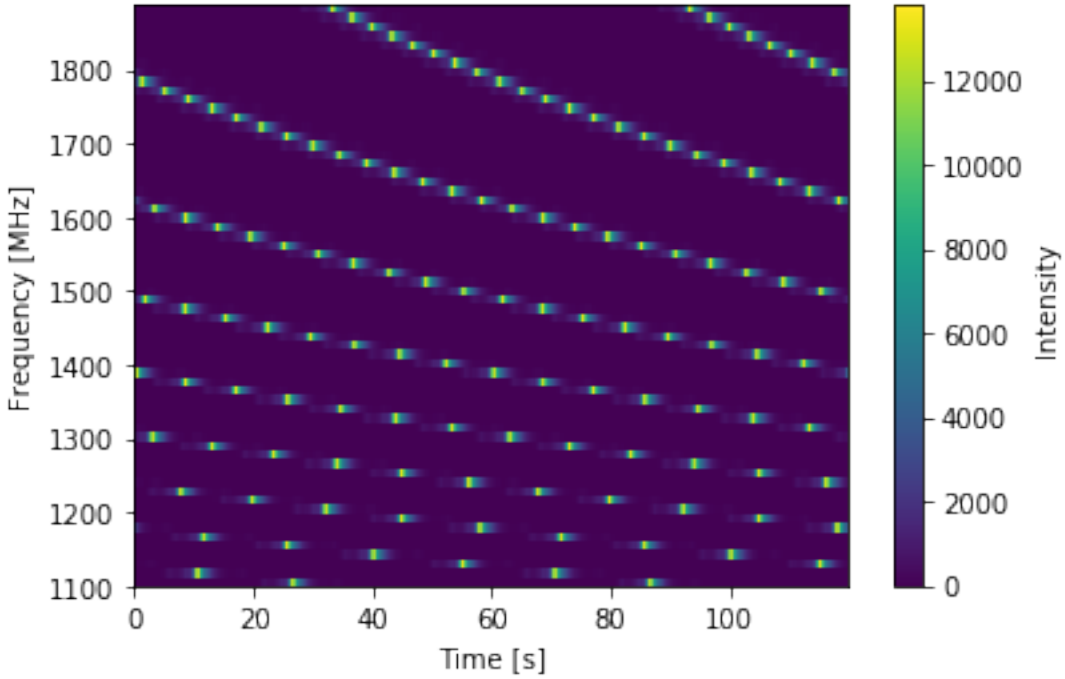


```
# Define the dispersion measure
dm =  15.921200 # pc cm^-3
```

```
# Now disperse the pulses. Once this is done, the psrsigsim remember that the␣
↪simulated data have been dispersed.
ism_sim.disperse(signal_1713, dm)
```

```
98% dispersed in 0.143 seconds.
```

```
# Now we can plot the dispersed pulses
plt.imshow(signal_1713.data[:,:4096], aspect = 'auto', interpolation='nearest',␣
↪origin = 'lower', \
           extent = [min(time[:4096]), max(time[:4096]), signal_1713.dat_freq[0].
↪value, signal_1713.dat_freq[-1].value])
plt.ylabel("Frequency [MHz]")
plt.xlabel("Time [s]")
plt.colorbar(label = "Intensity")
plt.show()
plt.close()
```



One can clearly see the time delay as a function of observing frequency that has been added to the signal. However, addition effects can be added either separtely or in combination with dispersion.

Frequency Dependent Delays

We can also add frequency dependent, or FD, delays to the simulated data. The formula for these FD parameters can be found in Zhu et al. 2015 and Arzoumanian et al. 2016. These delays are usually attributed to pulse profile evolution in frequency, but with the psrsigsim can also be directly injected into the simulated data without a frequency dependent pulse `Portait`.

The input for these delays are a list of coefficients (in units of seconds) that are used to determine the FD delay as

computed in log-frequency space. FD delays are referenced such that the delay due to FD parameters is 0 at observing frequencies of 1 GHz.

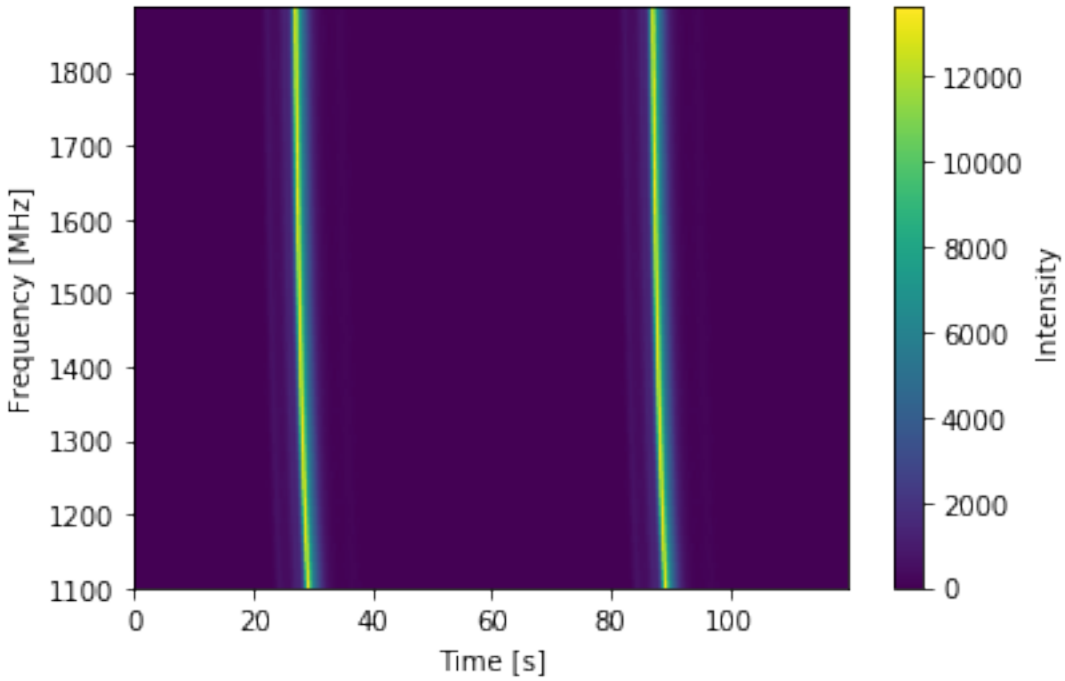
```
# We can input any number of FD parameters as a list, but we will use the FD␣
↪parameters in the NANOGrav 11-yr parfile
FD_J1713 = [-5.68565522e-04, 5.41762131e-04, -3.34764893e-04, 1.35695342e-04, -2.
↪87410591e-05] # seconds
```

As delays due to FD parameters are usually much smaller than those from dispersion, we will re-instantiate the pulse signal to better show the delays added from FD parameters.

```
# Re-make the pulses
pulsar_J1713.make_pulses(signal_1713, tobs = obslen)

# Now add the FD parameter delay, this takes two arguements, the signal and the list␣
↪of FD parameters
ism_sim.FD_shift(signal_1713, FD_J1713)
```

```
98% shifted in 0.113 seconds.
```

```
# Show the 2-D plot with the frequency dependent effects
plt.imshow(signal_1713.data[:,:4096], aspect = 'auto', interpolation='nearest',␣
↪origin = 'lower', \
           extent = [min(time[:4096]), max(time[:4096]), signal_1713.dat_freq[0].
↪value, signal_1713.dat_freq[-1].value])
plt.ylabel("Frequency [MHz]")
plt.xlabel("Time [s]")
plt.colorbar(label = "Intensity")
plt.show()
plt.close()
```



The shfit here is clearly visible at lower frequencies, though it is easy to see that the significance of this shift is much smaller than that from DM.

Scattering Broadening Delays

We can also add delays due to pulse scatter broadening to the simulated data. We can do this two different ways, both of which will be demonstrated here. The first is by directly shifting the simulated profile in time by the appropriate scattering delay. The second is by convolving an exponential scattering tail, based on the input parameters, and then convolving it with the pulse profile. Both of these effects are frequency dependent, so the direct shifts are frequency dependent, and the exponential tails are similarly frequency dependent.

Note that delays from scattering due to the ISM tend to be very small for low DM pulsars at low radio frequencies, so the scattering delay we will use here will be largely inflated so the effects are visible by-eye.

```
# We will first define the scattering timescale and reference frequency for the␣
↪timescale
tau_d = 1e-4 # seconds; note this is an unphysical number
ref_freq = 1500.0 # MHz, reference frequency of the scatter timescale input
```

Direct Shifting in Time

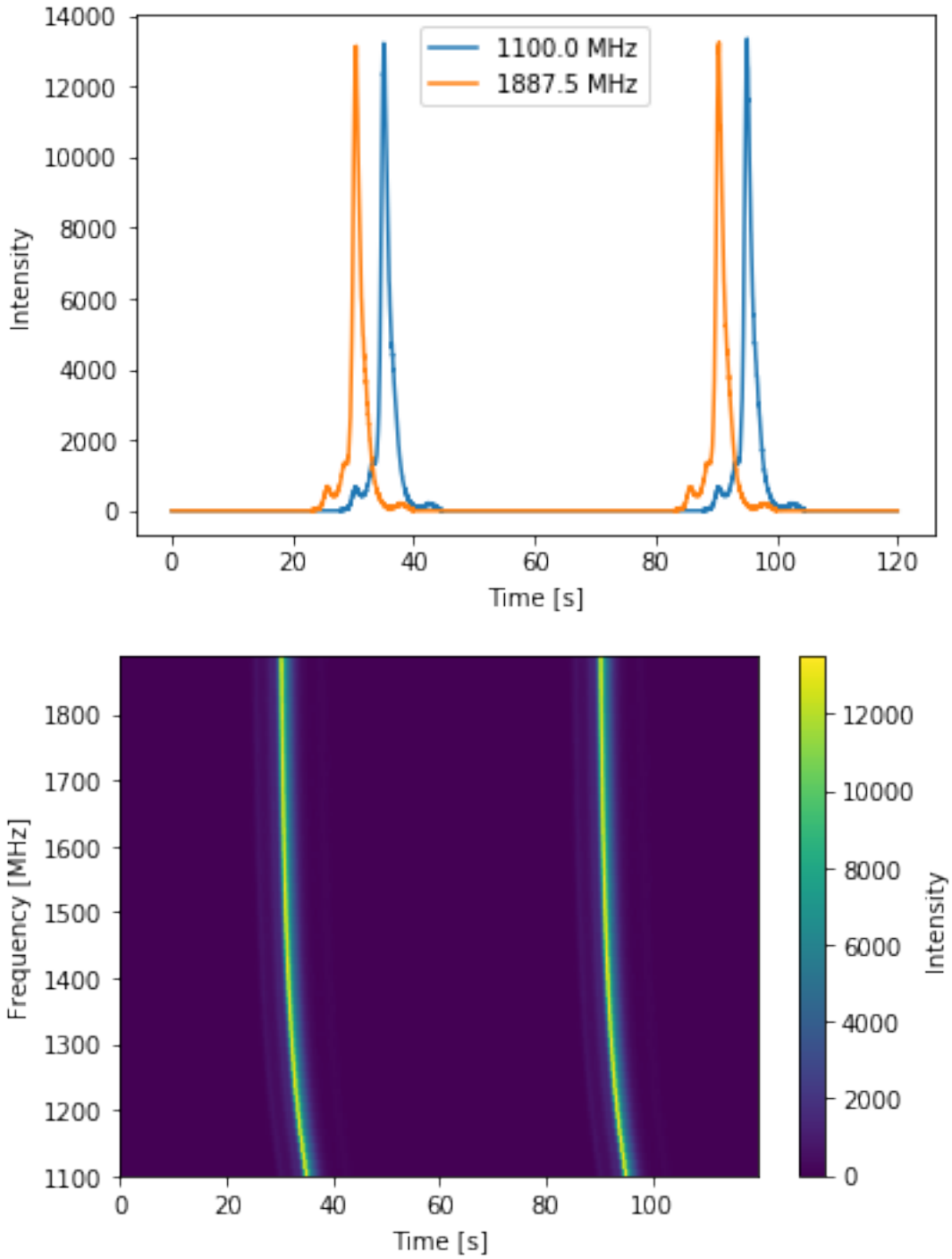
We start by showing how to directly shift the pulse profiles in time by the scattering timescale. We note that this does not add any pulse broadening, it simply shifts the peak of the pulse very slightly. Again, we remake the signal to better show the scatter broadening separately from the other ISM effects.

We also note here that `convolve=False` and `pulsar=None` are default inputs, and are not necessary for a direct shift.

```
# Re-make the pulses
pulsar_J1713.make_pulses(signal_1713, tobs = obslen)

# Now add the FD parameter delay, this takes two arguements, the signal and the list␣
↪of FD parameters
ism_sim.scatter_broaden(signal_1713, tau_d, ref_freq, convolve = False, pulsar = None)
```

```
98% scatter shifted in 0.115 seconds.
```

```
# Now we plot these profiles
# Since we know there are 2048 bins per pulse period, we can index the appropriate␣
↪amount
plt.plot(time[:4096], signal_1713.data[0,:4096], label = signal_1713.dat_freq[0])
plt.plot(time[:4096], signal_1713.data[-1,:4096], label = signal_1713.dat_freq[-1])
plt.ylabel("Intensity")
plt.xlabel("Time [s]")
plt.legend(loc = 'best')
plt.show()
plt.close()

# Show the 2-D plot with the frequency dependent effects
plt.imshow(signal_1713.data[:,:4096], aspect = 'auto', interpolation='nearest',␣
↪origin = 'lower', \
           extent = [min(time[:4096]), max(time[:4096]), signal_1713.dat_freq[0].
↪value, signal_1713.dat_freq[-1].value])
plt.ylabel("Frequency [MHz]")
plt.xlabel("Time [s]")
plt.colorbar(label = "Intensity")
plt.show()
plt.close()
```



We can see the signal has been shifted in time as a function of frequency in both the profiles and in the 2-D power spectrum. But the input profiles themselves remain unchanged from the input profile, e.g. no eponential scattering convolution as been done.

Scattering Tail Convolution

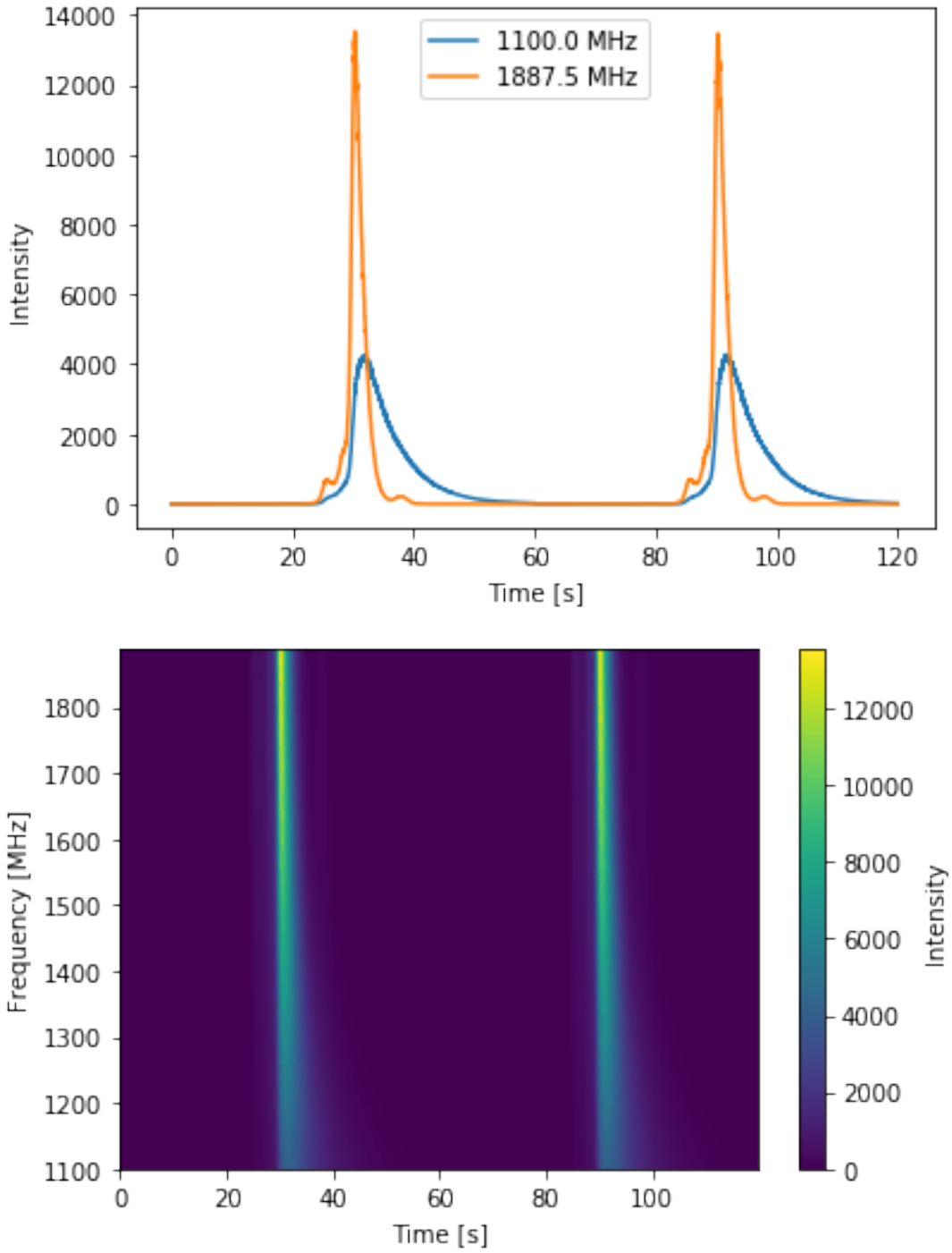
Here we show how to scatter broaden the profiles themselves in order to add pulse scatter broadening delays. The inputs necessary to do this are very similar to dierectly shifting it, with the addition of changing `convolve=True`

and adding the `Pulsar` object as input. Because this acts directly on the profiles, this must be done before `make_pulses()` is run, and cannot be undone.

We also note that the number of input profile channels must match the number of channels specified in the `Signal`. Here this is 64 channels, so we can reinstantiate the profile including the `Nchan=64` flag, and then make the pulsar again.

```
# Now we define the data profile
J1713_prof = pss.pulsar.DataProfile(J1713_dataprof, Nchan=64)

# Now we define the pulsar with the scaled J1713+0747 profiles
pulsar_J1713 = pss.pulsar.Pulsar(period, Smean, profiles=J1713_prof, name = psr_name)

# Now add the FD parameter delay, this takes two arguements, the signal and the list␣
↪of FD parameters
ism_sim.scatter_broaden(signal_1713, tau_d, ref_freq, convolve = True, pulsar =␣
↪pulsar_J1713)

# Re-make the pulses
pulsar_J1713.make_pulses(signal_1713, tobs = obslen)
```

```
# Now we plot these profiles
# Since we know there are 2048 bins per pulse period, we can index the appropriate␣
↪amount
plt.plot(time[:4096], signal_1713.data[0,:4096], label = signal_1713.dat_freq[0])
plt.plot(time[:4096], signal_1713.data[-1,:4096], label = signal_1713.dat_freq[-1])
plt.ylabel("Intensity")
plt.xlabel("Time [s]")
plt.legend(loc = 'best')
plt.show()
plt.close()

# Show the 2-D plot with the frequency dependent effects
plt.imshow(signal_1713.data[:,:4096], aspect = 'auto', interpolation='nearest',␣
↪origin = 'lower', \
           extent = [min(time[:4096]), max(time[:4096]), signal_1713.dat_freq[0].
↪value, signal_1713.dat_freq[-1].value])
plt.ylabel("Frequency [MHz]")
plt.xlabel("Time [s]")
plt.colorbar(label = "Intensity")
plt.show()
plt.close()
```



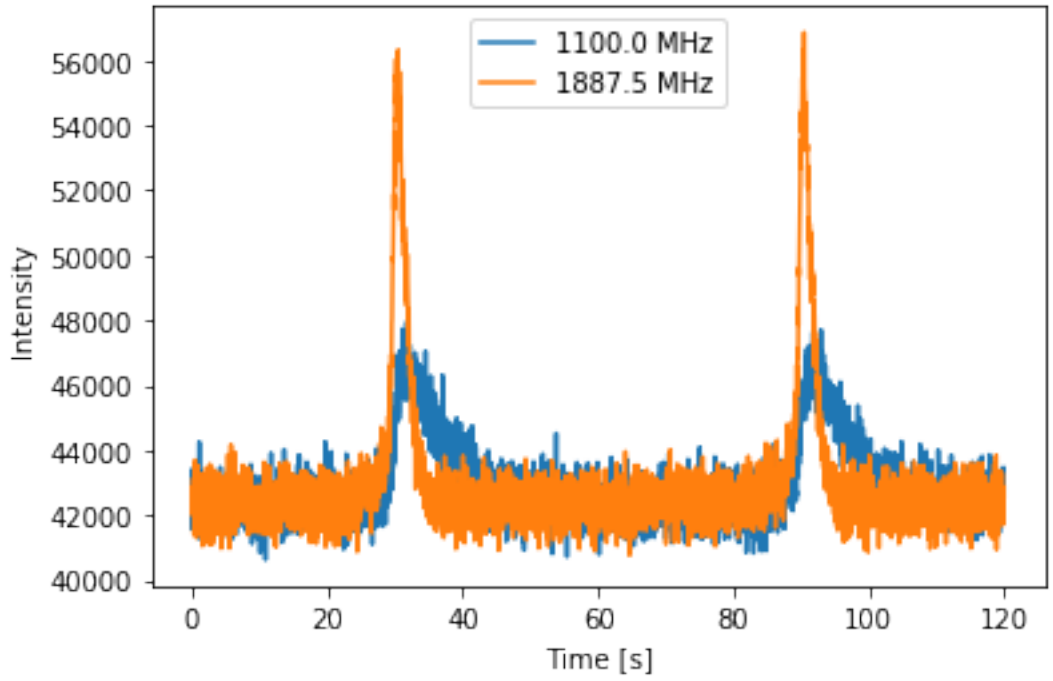
We can see now that the scattering tails have been convolved with the actual profiles, and this shows up in both the individual pulse profiles, which clearly show a shift in the peak of the pulse, as well as in the power spectrum, where the profile is clearly getting scattered out.

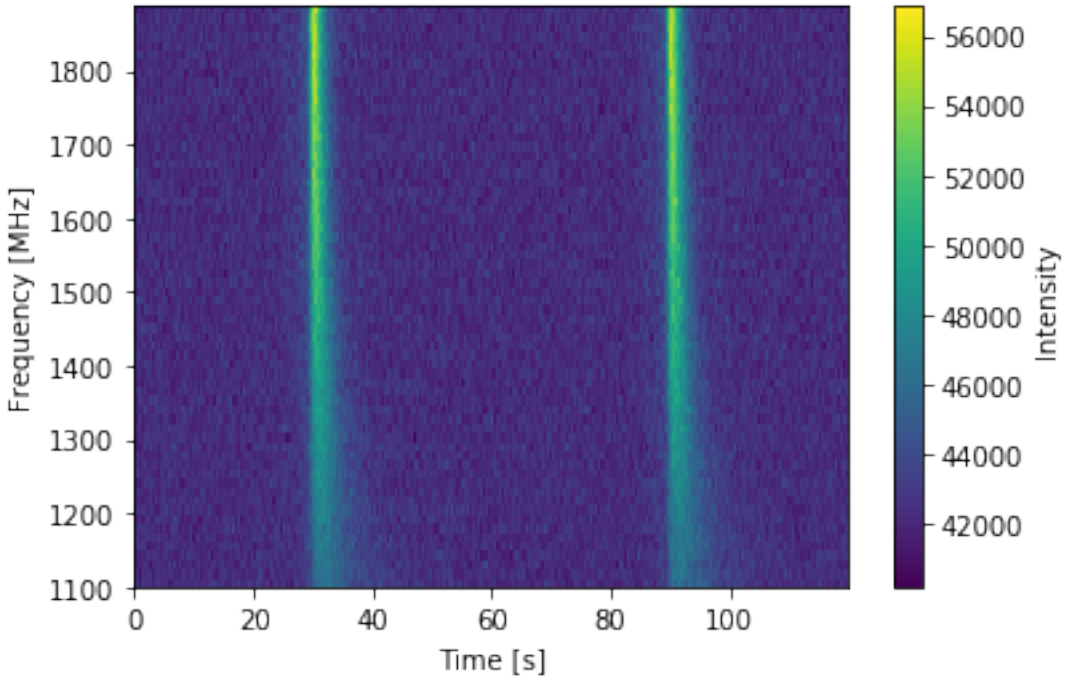
Simulating the Scatter Broadened Pulsar

Now we will finish by `observe`ing the pulsar and looking at the data with the added noise.

```
# Observe with the telescope
tscope.observe(signal_1713, pulsar_J1713, system="Lband_GUPPI", noise=True)
```

```
# Since we know there are 2048 bins per pulse period, we can index the appropriate␣
↪amount
plt.plot(time[:4096], signal_1713.data[0,:4096], label = signal_1713.dat_freq[0])
plt.plot(time[:4096], signal_1713.data[-1,:4096], label = signal_1713.dat_freq[-1])
plt.ylabel("Intensity")
plt.xlabel("Time [s]")
plt.legend(loc = 'best')
plt.show()
plt.close()

# And the 2-D plot
plt.imshow(signal_1713.data[:,:4096], aspect = 'auto', interpolation='nearest',␣
↪origin = 'lower', \
           extent = [min(time[:4096]), max(time[:4096]), signal_1713.dat_freq[0].
↪value, signal_1713.dat_freq[-1].value])
plt.ylabel("Frequency [MHz]")
plt.xlabel("Time [s]")
plt.colorbar(label = "Intensity")
plt.show()
plt.close()
```





Note: This tutorial was generated from a Jupyter notebook that can be downloaded here.

1.5.8 Telescopes: Tutorial 5

This notebook will build on the previous tutorials, showing more features of the `PsrSigSim`. Details will be given for new features, while other features have been discussed in the previous tutorial notebook. This notebook shows the details of different telescopes currently included in the `PsrSigSim`, how to call them, and how to define a user `telescope` for a simulated observation.

We again simulate precision pulsar timing data with high signal-to-noise pulse profiles in order to clearly show the input pulse profile in the final simulated data product. We note that the use of different telescopes will result in different signal strengths, as would be expected.

This example will follow previous notebook in defining all necessary classes except for `telescope`.

```
# import some useful packages
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

# import the pulsar signal simulator
import psrsigsim as pss
```

The Folded Signal

Here we will use the same `Signal` definitions that have been used in the previous tutorials. We will again simulate a 20 minute long observation total, with subintegrations of 1 minute. The other simulation parameters will be 64 frequency channels each 12.5 MHz wide (for 800 MHz bandwidth).

We will simulate a real pulsar, J1713+0747, as we have a premade profile for this pulsar. The period, dm, and other relavent pulsar parameters come from the NANOGrav 11-yr data release.

```
# Define our signal variables.
f0 = 1500 # center observing frequecy in MHz
bw = 800.0 # observation MHz
Nf = 64 # number of frequency channels
# We define the pulse period early here so we can similarly define the frequency
period = 0.00457 # pulsar period in seconds for J1713+0747
f_samp = (1.0/period)*2048*10**-6 # sample rate of data in MHz (here 2048 samples␣
↪across the pulse period
sublen = 60.0 # subintegration length in seconds, or rate to dump data at
# Now we define our signal
signal_1713_GBT = pss.signal.FilterBankSignal(fcent = f0, bandwidth = bw, Nsubband=Nf,
↪ sample_rate = f_samp,
                                              sublen = sublen, fold = True) # fold is set to␣
↪`True`
```

```
Warning: specified sample rate 0.4481400437636761 MHz < Nyquist frequency 1600.0 MHz
```

The Pulsar and Profiles

Now we will load the pulse profile as in Tutorial 3 and intilialize a single `Pulsar` object.

```
# First we load the data array
path = 'psrsigsim/data/J1713+0747_profile.npy'
J1713_dataprof = np.load(path)

# Now we define the data profile
J1713_prof = pss.pulsar.DataProfile(J1713_dataprof)
```

```
# Define the values needed for the puslar
Smean = 0.009 # The mean flux of the pulsar, J1713+0747 at 1400 MHz from the ATNF␣
↪pulsar catatlog, here 0.009 Jy
psr_name = "J1713+0747" # The name of our simulated pulsar

# Now we define the pulsar with the scaled J1713+0747 profiles
pulsar_J1713 = pss.pulsar.Pulsar(period, Smean, profiles=J1713_prof, name = psr_name)
```

```
# define the observation length
obslen = 60.0*20 # seconds, 20 minutes in total
```

The ISM

Here we define the `ISM` class used to disperse the simulated pulses.

```
# Define the dispersion measure
dm =  15.921200 # pc cm^-3
# And define the ISM object, note that this class takes no initial arguements
ism_sim = pss.ism.ISM()
```

Defining Telescopes

Here we will show to use the two predefined telescopes, Green Bank and Arecibo, and the systems accociated with them. We will also show how to define a `telescope` from scratch, so that any current or future telescopes and systems can be simulated.

Predefined Telescopes

We start off by showing the two predefined telescopes.

```
# Define the Green Bank Telescope
tscope_GBT = pss.telescope.telescope.GBT()

# Define the Arecibo Telescope
tscope_AO = pss.telescope.telescope.Arecibo()
```

Each telescope is made up of one or more `systems` consisting of a `Reciever` and a `Backend`. For the predefined telescopes, the systems for the `GBT` are the L-band-GUPPI system or the 800 MHz-GUPPI system. For `Arecibo` these are the 430 MHz-PUPPI system or the L-band-PUPPI system. One can check to see what these systems and their parameters are as we show below.

```
# Information about the GBT systems
print(tscope_GBT.systems)
# We can also find out information about a receiver that has been defined here
rcvr_LGUP = tscope_GBT.systems['Lband_GUPPI'][0]
print(rcvr_LGUP.bandwidth, rcvr_LGUP.fcent, rcvr_LGUP.name)
```

```
{'820_GUPPI': (Receiver(820), Backend(GUPPI)), 'Lband_GUPPI': (Receiver(Lband),␣
↪Backend(GUPPI)), '800_GASP': (Receiver(800), Backend(GASP)), 'Lband_GASP':␣
↪(Receiver(Lband), Backend(GASP))}
800.0 MHz 1400.0 MHz Lband
```

Defining a new system

One can also add a new system to one of these existing telescopes, similarly to what will be done when define a new telescope from scratch. Here we will add the 350 MHz receiver with the GUPPI backend to the Green Bank Telescope.

First we define a new `Receiver` and `Backend` object. The `Receiver` object needs a center frequency of the receiver in MHz, a bandwidth in MHz to be centered on that center frequency, and a name. The `Backend` object needs only a name and a sampling rate in MHz. This sampling rate should be the maximum sampling rate of the backend, as it will allow lower sampling rates, but not higher sampling rates.

```
# First we define a new receiver
rcvr_350 = pss.telescope.receiver.Receiver(fcent=350, bandwidth=100, name="350")
# And then we want to use the GUPPI backend
guppi = pss.telescope.backend.Backend(samprate=3.125, name="GUPPI")
```

```
# Now we add the new system. This needs just the receiver, backend, and a name
tscope_GBT.add_system(name="350_GUPPI", receiver=rcvr_350, backend=guppi)
# And now we check that it has been added
print(tscope_GBT.systems["350_GUPPI"])
```

```
(Receiver(350), Backend(GUPPI))
```

Defining a new telescope

We can also define a new telescope from scratch. In addition to needing the `Receiver` and `Backend` objects to define at least one system, the `telescope` also needs the aperature size in meters, the total area in meters^2, the system temperature in kelvin, and a name. Here we will define a small 3-meter aperature circular radio telescope that you might find at a University or somebodies backyard.

```
# We first need to define the telescope parameters
aperature = 3.0 # meters
area = (0.5*aperature)**2*np.pi # meters^2
Tsys = 250.0 # kelvin, note this is not a realistic system temperature for a backyard␣
↪telescope
name = "Backyard_Telescope"
```

```
# Now we can define the telescope
tscope_bkyd = pss.telescope.Telescope(aperature, area=area, Tsys=Tsys, name=name)
```

Now similarly to defining a new system before, we must add a system to our new telescope by defining a receiver and a backend. Since this just represents a little telescope, the system won't be comparable to the previously defined telescope.

```
rcvr_bkyd = pss.telescope.receiver.Receiver(fcent=1400, bandwidth=20, name="Lband")

backend_bkyd = pss.telescope.backend.Backend(samprate=0.25, name="Laptop") # Note␣
↪this is not a realistic sampling rate
```

```
# Add the system to our telecope
tscope_bkyd.add_system(name="bkyd", receiver=rcvr_bkyd, backend=backend_bkyd)
# And now we check that it has been added
print(tscope_bkyd.systems)
```

```
{'bkyd': (Receiver(Lband), Backend(Laptop))}
```

Observing with different telescopes

Now that we have three different telescopes, we can observe our simulated pulsar with all three and compare the sensitivity of each telescope for the same initial `Signal` and `Pulsar`. Since the radiometer noise from the telescope is added directly to the signal though, we will need to define two additional `Signals` and create pulses for them before we can observe them with different telescopes.

```
# We define three new, similar, signals, one for each telescope
signal_1713_AO = pss.signal.FilterBankSignal(fcent = f0, bandwidth = bw, Nsubband=Nf,␣
↪sample_rate = f_samp,
                                             sublen = sublen, fold = True)
# Our backyard telescope will need slightly different parameters to be comparable to␣
↪the other signals
f0_bkyd = 1400.0 # center frequency of our backyard telescope
bw_bkyd = 20.0 # Bandwidth of our backyard telescope
Nf_bkyd = 1 # only process one frequency channel 20 MHz wide for our backyard␣
↪telescope
```

(continues on next page)

(continued from previous page)

```
signal_1713_bkyd = pss.signal.FilterBankSignal(fcent = f0_bkyd, bandwidth = bw_bkyd,␣
↪Nsubband=Nf_bkyd, \
                                               sample_rate = f_samp, sublen = sublen,␣
↪fold = True)
```

```
Warning: specified sample rate 0.4481400437636761 MHz < Nyquist frequency 1600.0 MHz
Warning: specified sample rate 0.4481400437636761 MHz < Nyquist frequency 40.0 MHz
```

```
# Now we make pulses for all three signals
pulsar_J1713.make_pulses(signal_1713_GBT, tobs = obslen)
pulsar_J1713.make_pulses(signal_1713_AO, tobs = obslen)
pulsar_J1713.make_pulses(signal_1713_bkyd, tobs = obslen)
# And disperse them
ism_sim.disperse(signal_1713_GBT, dm)
ism_sim.disperse(signal_1713_AO, dm)
ism_sim.disperse(signal_1713_bkyd, dm)
```

```
100% dispersed in 0.001 seconds.
```

```
# And now we observe with each telescope, note the only change is the system name.␣
↪First the GBT
tscope_GBT.observe(signal_1713_GBT, pulsar_J1713, system="Lband_GUPPI", noise=True)
# Then Arecibo
tscope_AO.observe(signal_1713_AO, pulsar_J1713, system="Lband_PUPPI", noise=True)
# And finally our little backyard telescope
tscope_bkyd.observe(signal_1713_bkyd, pulsar_J1713, system="bkyd", noise=True)
```

```
WARNING: AstropyDeprecationWarning: The truth value of a Quantity is ambiguous. In␣
↪the future this will raise a ValueError. [astropy.units.quantity]
```

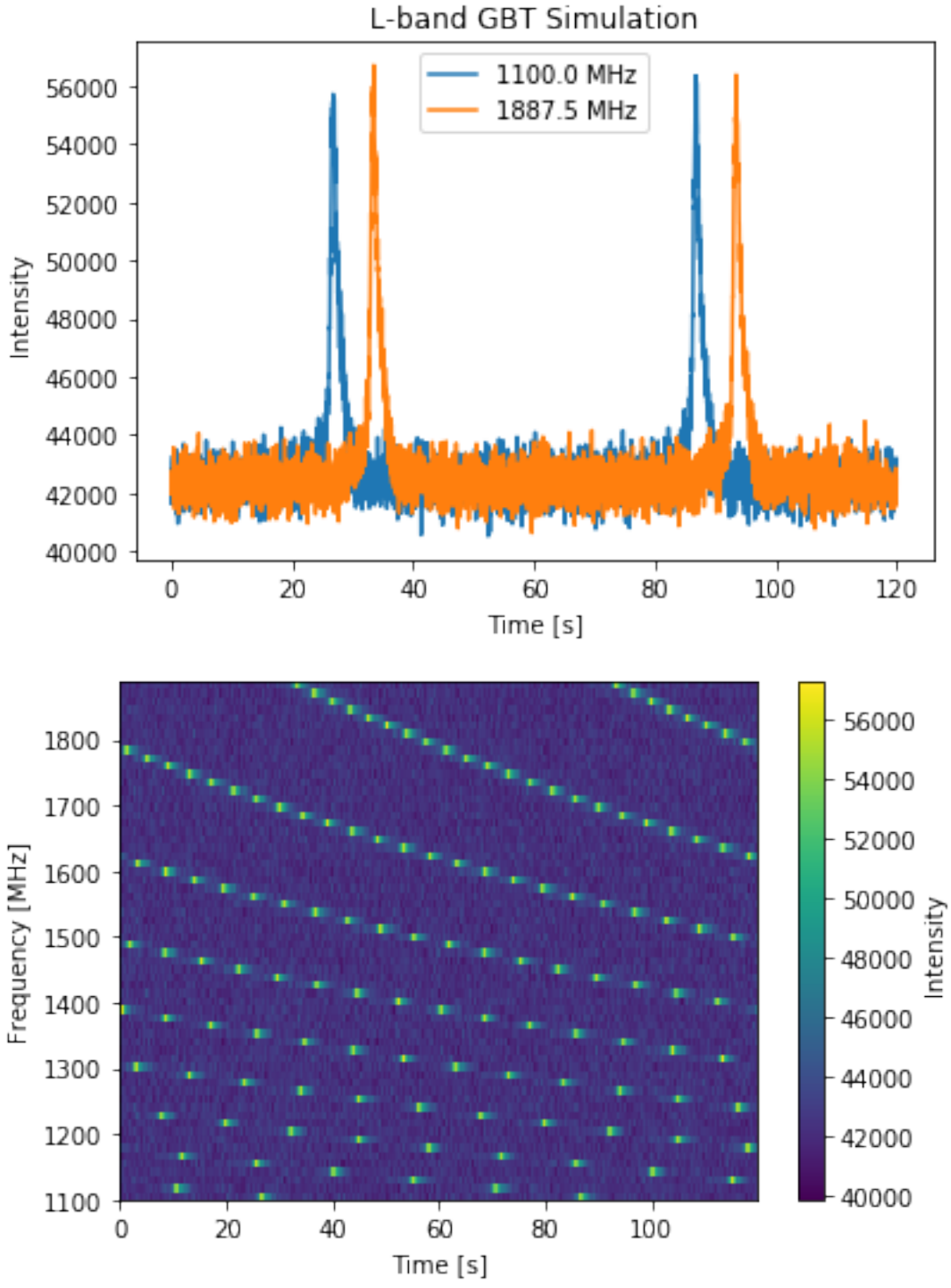
Now we can look at the simulated data and compare the sensitivity of the different telescopes. We first plot the observation from the GBT, then Arecibo, and then our newly defined backyard telescope.

```
# We first plot the first two pulses in frequency-time space to show the undispersed␣
↪pulses
time = np.linspace(0, obslen, len(signal_1713_GBT.data[0,:]))

# Since we know there are 2048 bins per pulse period, we can index the appropriate␣
↪amount
plt.plot(time[:4096], signal_1713_GBT.data[0,:4096], label = signal_1713_GBT.dat_
↪freq[0])
plt.plot(time[:4096], signal_1713_GBT.data[-1,:4096], label = signal_1713_GBT.dat_
↪freq[-1])
plt.ylabel("Intensity")
plt.xlabel("Time [s]")
plt.legend(loc = 'best')
plt.title("L-band GBT Simulation")
plt.show()
plt.close()

# And the 2-D plot
plt.imshow(signal_1713_GBT.data[:,:4096], aspect = 'auto', interpolation='nearest',␣
↪origin = 'lower', \
           extent = [min(time[:4096]), max(time[:4096]), signal_1713_GBT.dat_freq[0].
↪value, signal_1713_GBT.dat_freq[-1].value])
```

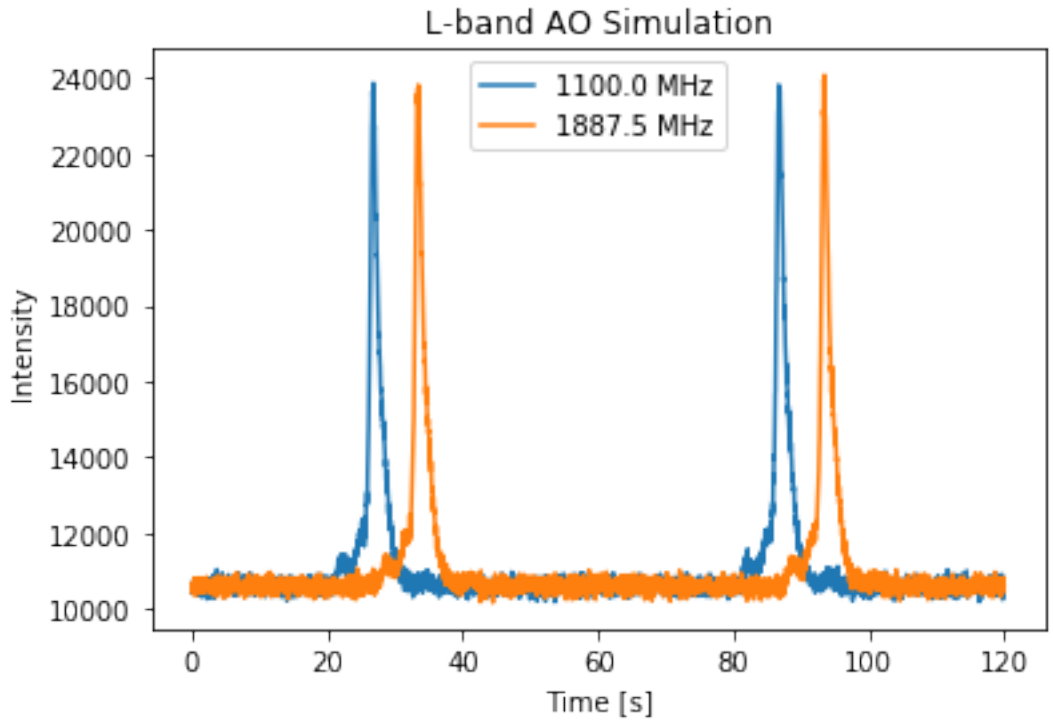
(continues on next page)

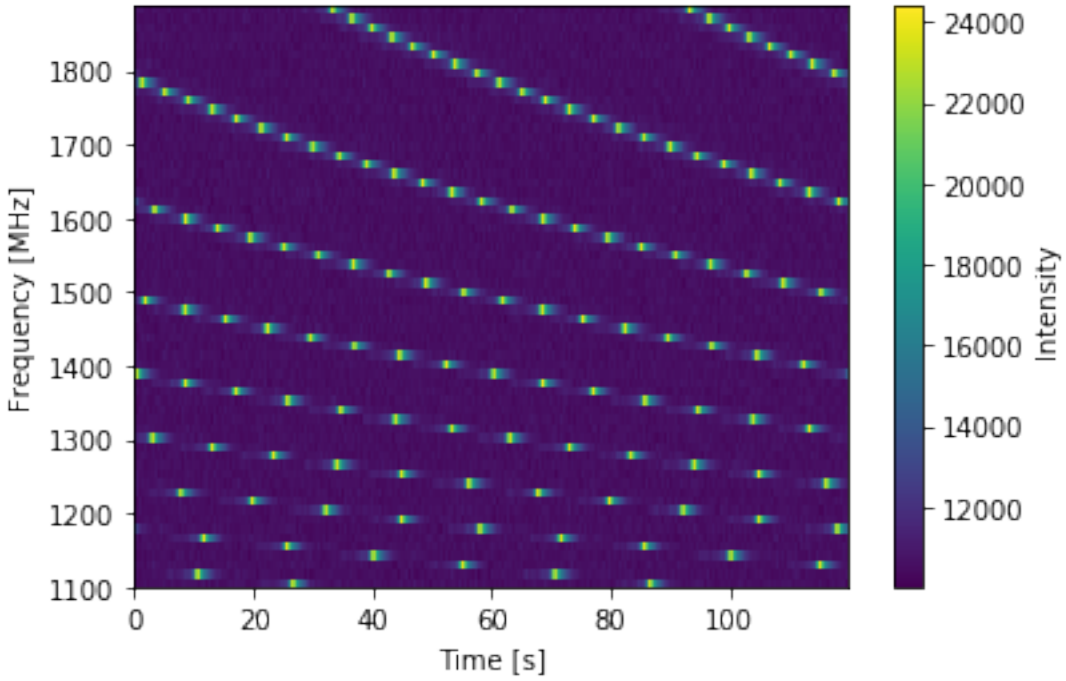
(continued from previous page)

```
plt.ylabel("Frequency [MHz]")
plt.xlabel("Time [s]")
plt.colorbar(label = "Intensity")
plt.show()
plt.close()
```

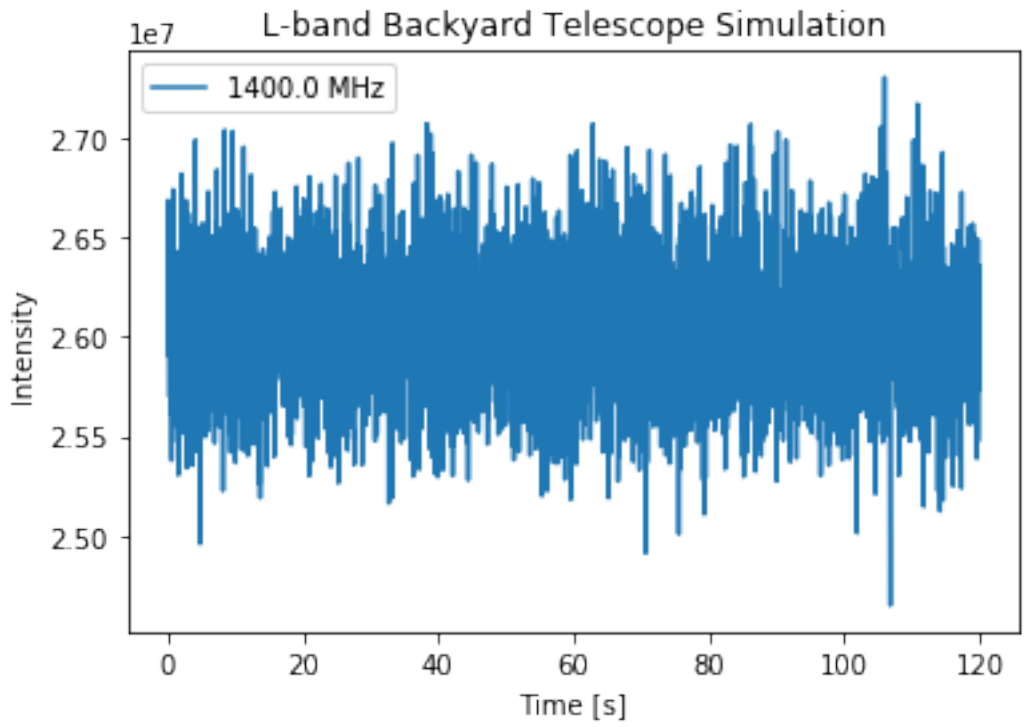


```
# Since we know there are 2048 bins per pulse period, we can index the appropriate␣
↪amount
plt.plot(time[:4096], signal_1713_AO.data[0,:4096], label = signal_1713_AO.dat_
↪freq[0])
plt.plot(time[:4096], signal_1713_AO.data[-1,:4096], label = signal_1713_AO.dat_freq[-
↪1])
plt.ylabel("Intensity")
plt.xlabel("Time [s]")
plt.legend(loc = 'best')
plt.title("L-band AO Simulation")
plt.show()
plt.close()

# And the 2-D plot
plt.imshow(signal_1713_AO.data[:,:4096], aspect = 'auto', interpolation='nearest',␣
↪origin = 'lower', \
           extent = [min(time[:4096]), max(time[:4096]), signal_1713_AO.dat_freq[0].
↪value, signal_1713_AO.dat_freq[-1].value])
plt.ylabel("Frequency [MHz]")
plt.xlabel("Time [s]")
plt.colorbar(label = "Intensity")
plt.show()
plt.close()
```






```
# Since we know there are 2048 bins per pulse period, we can index the appropriate␣
↪amount
plt.plot(time[:4096], signal_1713_bkyd.data[0,:4096], label = "1400.0 MHz")
plt.ylabel("Intensity")
plt.xlabel("Time [s]")
plt.legend(loc = 'best')
plt.title("L-band Backyard Telescope Simulation")
plt.show()
plt.close()
```



We can see that, as expected, the Arecibo telescope is more sensitive than the GBT when observing over the same timescale. We can also see that even though the simulated pulsar here is easily visible with these large telescopes, our backyard telescope is not able to see the pulsar over the same amount of time, since the output is pure noise. The `PsrSigSim` can be used to determine the approximate sensitivity of an observation of a simulated pulsar with any given telescope that can be defined.

Note: This tutorial was generated from a Jupyter notebook that can be downloaded here.

1.5.9 Simulation Class: Tutorial 6

This notebook will demonstrate how to use the `Simulation` class of the pulsar signal simulator for more automated simulation of data. The `Simulation` class is designed as a convenience class within the `PsrSigSim`. Instead of instantiating each step of the simulation, the `Simulation` class allows the input of all desired variables for the simulation at once, and then will run all parts of the simulation. The `Simulation` class also allows for individual running of each step (e.g. `Signal`, `Pulsar`, etc.) if desired. Not all options available within the `Simulation` will be demonstrated in this notebook.

```
# import some useful packages
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

# import the pulsar signal simulator
import psrsigsim as pss
```

Instead of defining each variable individually, the simulation class gets instantiated all at once. This can be done either by defining each variable individually, or by passing a dictionary with all parameters defined to the simulator. The dictionary keys should be the same as the input flags for the `Simulation` class.

```
sim = pss.simulate.Simulation(
                fcent = 430, # center frequency of observation, MHz
                bandwidth = 100, # Bandwidth of observation, MHz
                sample_rate = 1.0*2048*10**-6, # Sampling rate of the data, MHz
                dtype = np.float32, # data type to write out the signal in
                Npols = 1, # number of polarizations to simulate, only one available
                Nchan = 64, # number of subbands for the observation
                sublen = 2.0, # length of subintegration of signal
                fold = True, # flag to produce fold-mode, subintegrated data
                period = 1.0, # pulsar period in seconds
                Smean = 1.0, # mean flux of the pulsar in Jy
                profiles = [0.5, 0.05, 1.0], # Profile - may be a data array, list␣
↪of Gaussian components, or profile class object
                tobs = 4.0, # length of observation in seconds
                name = 'J0000+0000', # name of the simulated pulsar
                dm = 10.0, # dispersion measure in pc cm^-3
                tau_d = None, # scattering timescale in seconds
                tau_d_ref_f = None, # reference frequency of scattering timescale in␣
↪seconds
                aperture = 100.0, # telescope aperture in meters
                area = 5500.0, # telescope area in meters square
                Tsys = 35.0, # telescope system temperature
                tscope_name = "TestScope", # telescope name (default GBT and Arecibo␣
↪available)
                system_name = "TestSys", # observing system name
                rcvr_fcent = 430, # center frequency of the receiver in MHz
                rcvr_bw = 100, # receiver bandwidth in MHz
                rcvr_name ="TestRCVR", # name of receiver
                backend_samprate = 1.5625, # bandend maximum sampling rate in MHz
                backend_name = "TestBack", # bandend name
                tempfile = None, # optional name of template fits file to simulate
                psrdict = None, # optional dictionary to give for input parameters
                )
```

To give the `Simulation` class a dictionary of these parameters, the input may look something like below (Note - all parameters have the same units and names as above).

```
pdict = {'fcent' : 430,
         'bandwidth' : 100,
         'sample_rate' : 1.0*2048*10**-6,
         'dtype' : np.float32,
         'Npols' : 1,
         'Nchan' : 64,
         'sublen' : 2.0,
         'fold' : True,
         'period' : 1.0,
         'Smean' : 1.0,
         'profiles' : [0.5, 0.05, 1.0],
         'tobs' : 4.0,
         'name' : 'J0000+0000',
         'dm' : 10.0,
         'tau_d' : None,
         'tau_d_ref_f' : None,
         'aperture' : 100.0,
         'area' : 5500.0,
         'Tsys' : 35.0,
         'tscope_name' : "TestScope",
```

(continues on next page)

(continued from previous page)

```
          'system_name' : "TestSys",
          'rcvr_fcent' : 430,
          'rcvr_bw' : 100,
          'rcvr_name' : "TestRCVR",
          'backend_samprate' : 1.5625,
          'backend_name' : "TestBack",
          'tempfile' : None,
         }

sim = pss.simulate.Simulation(psrdict = pdict)
```

Simulating the Data

Once the `Simulation` class is initialized with all of the necessary parameters, there are two ways to run the simulation. The first is simply by running the `simulate()` function, which will fully simulate the the data from start to finish.

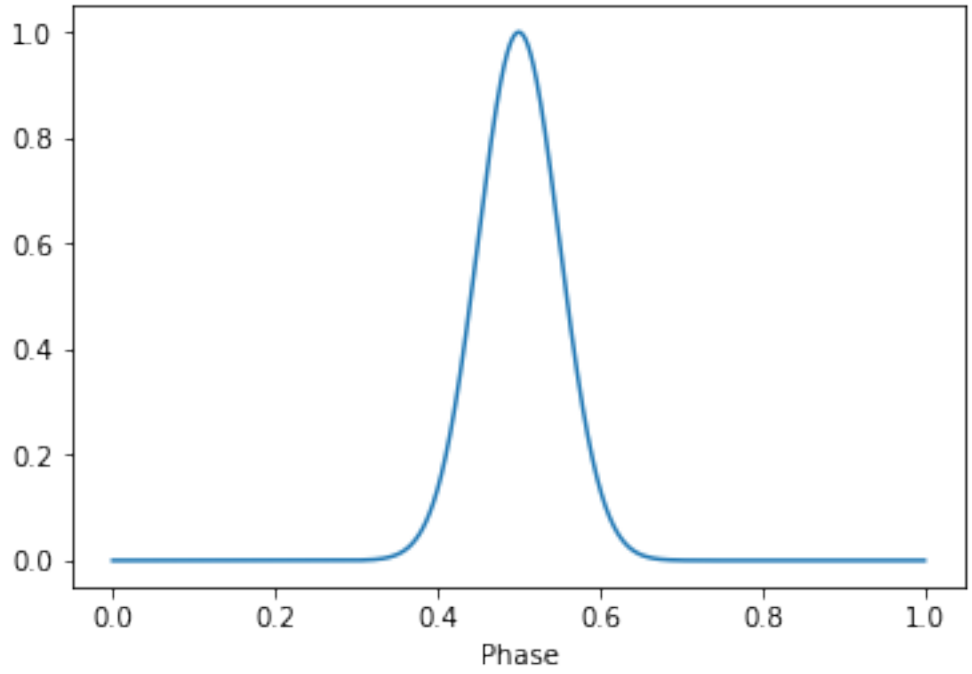
```
sim.simulate()
```

```
Warning: specified sample rate 0.002048 MHz < Nyquist frequency 200.0 MHz
98% dispersed in 0.050 seconds.
```

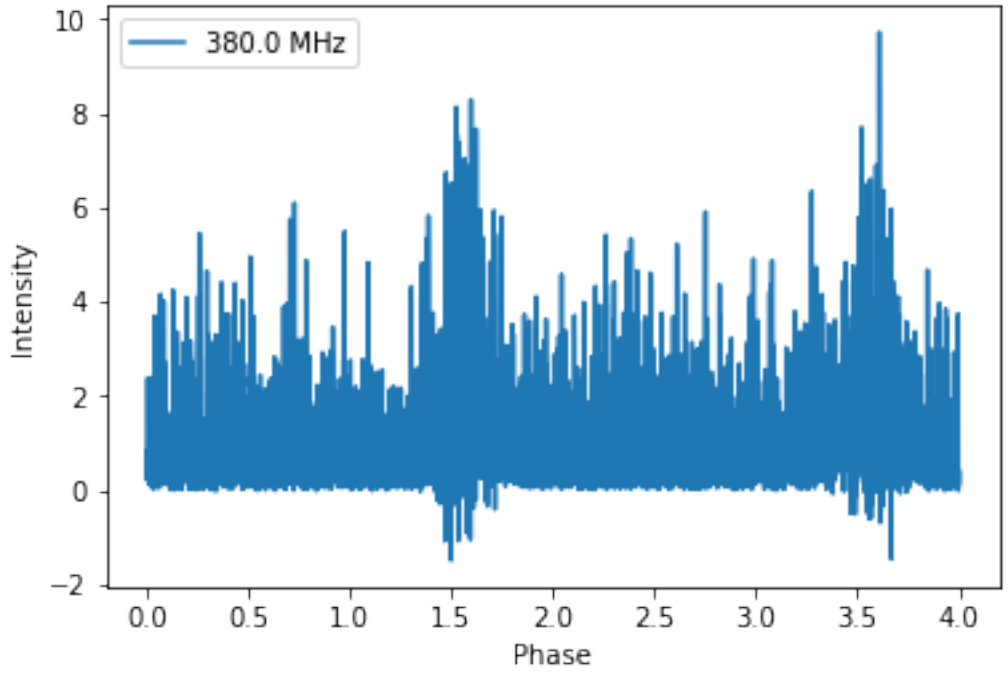
```
WARNING: AstropyDeprecationWarning: The truth value of a Quantity is ambiguous. In␣
↪the future this will raise a ValueError. [astropy.units.quantity]
```

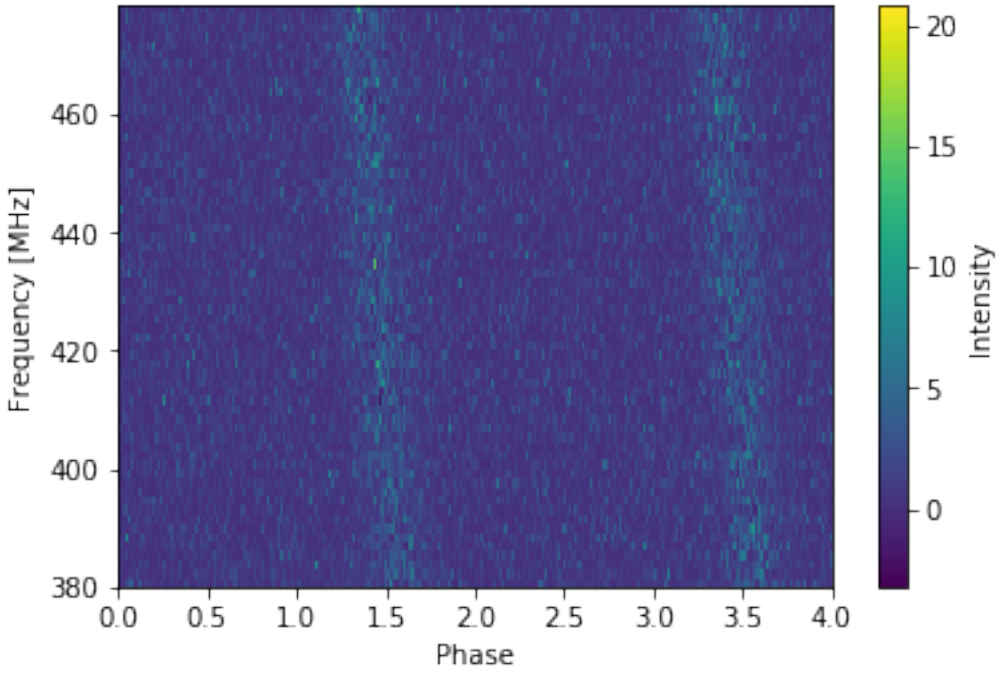
If we want to look at the data that has been simulated, it can be accessed via `sim.signal.data`. The simulate class has attributes for each of the objects simulated (e.g. `signal`, `pulsar`, etc.) if the user would like to access those parameters. We will look at the simulated data and plot it below.

```
# We can look at the simulated profiles
plt.plot(np.linspace(0,1,2048), sim.profiles.profiles[0])
plt.xlabel("Phase")
plt.show()
plt.close()
```




```
# Get the simulated data
sim_data = sim.signal.data

# Get the phases of the pulse
phases = np.linspace(0, sim.tobs/sim.period, len(sim_data[0,:]))
# Plot just the pulses in the first frequency channels
plt.plot(phases, sim_data[0,:], label = sim.signal.dat_freq[0])
plt.ylabel("Intensity")
plt.xlabel("Phase")
plt.legend(loc = 'best')
plt.show()
plt.close()

# Make the 2-D plot of intensity v. frequency and pulse phase. You can see the slight␣
↪dispersive sweep here.
plt.imshow(sim_data, aspect = 'auto', interpolation='nearest', origin = 'lower', \
           extent = [min(phases), max(phases), sim.signal.dat_freq[0].value, sim.
↪signal.dat_freq[-1].value])
plt.ylabel("Frequency [MHz]")
plt.xlabel("Phase")
plt.colorbar(label = "Intensity")
plt.show()
plt.close()
```





A second way to simulate

The second way to run these simulations is by initializing all of the different objects separately, instead of through the simulation class. This allows slightly more freedom, as well as modifications to the initially input simulated parameters.

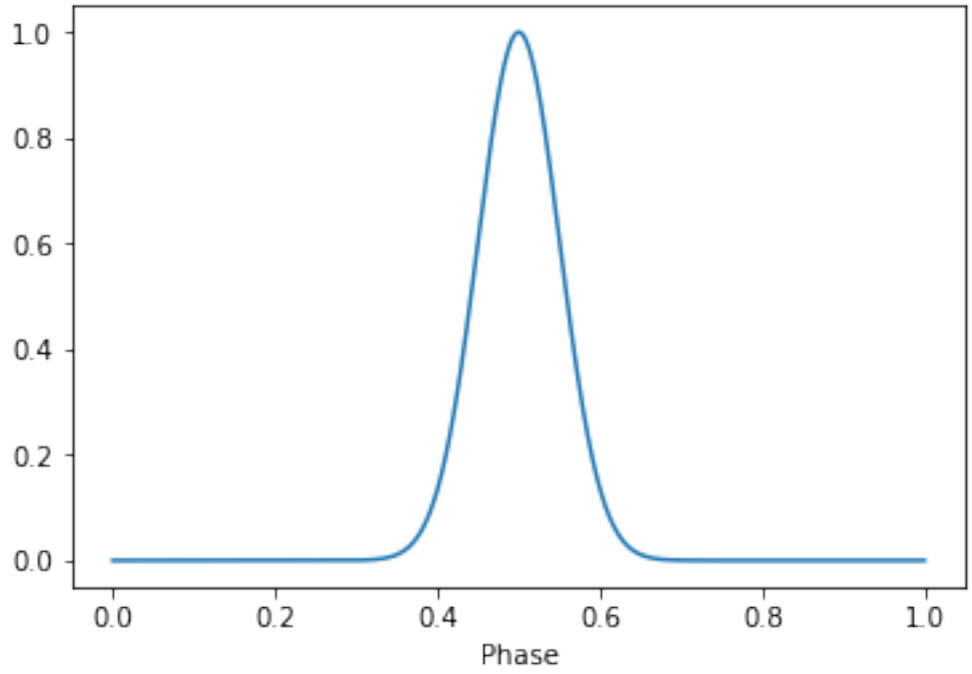
```
# We start by initializing the signal
sim.init_signal()
# Initialize the profile
sim.init_profile()
# Now the pulsar
sim.init_pulsar()
# Now the ISM
sim.init_ism()
# Now make the pulses
sim.pulsar.make_pulses(sim.signal, tobs = sim.tobs)
# disperse the simulated pulses
sim.ism.disperse(sim.signal, sim.dm)
# Now add the telescope and radiometer noise
sim.init_telescope()
# add radiometer noise
out_array = sim.tscope.observe(sim.signal, sim.pulsar, system=sim.system_name,␣
↪noise=True)
```

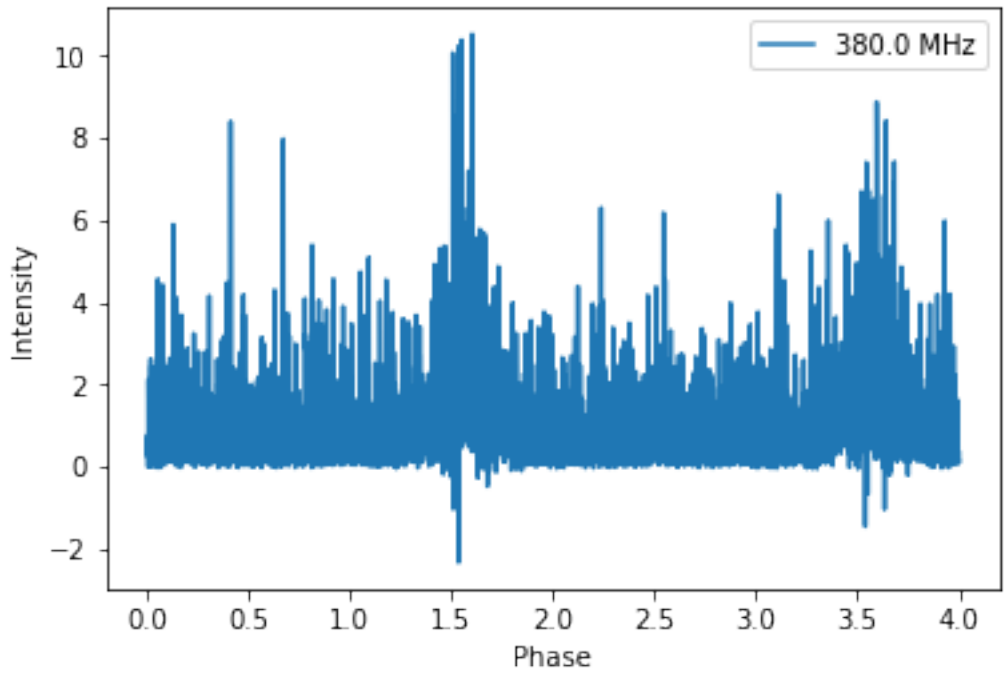
```
Warning: specified sample rate 0.002048 MHz < Nyquist frequency 200.0 MHz
98% dispersed in 0.055 seconds.
```

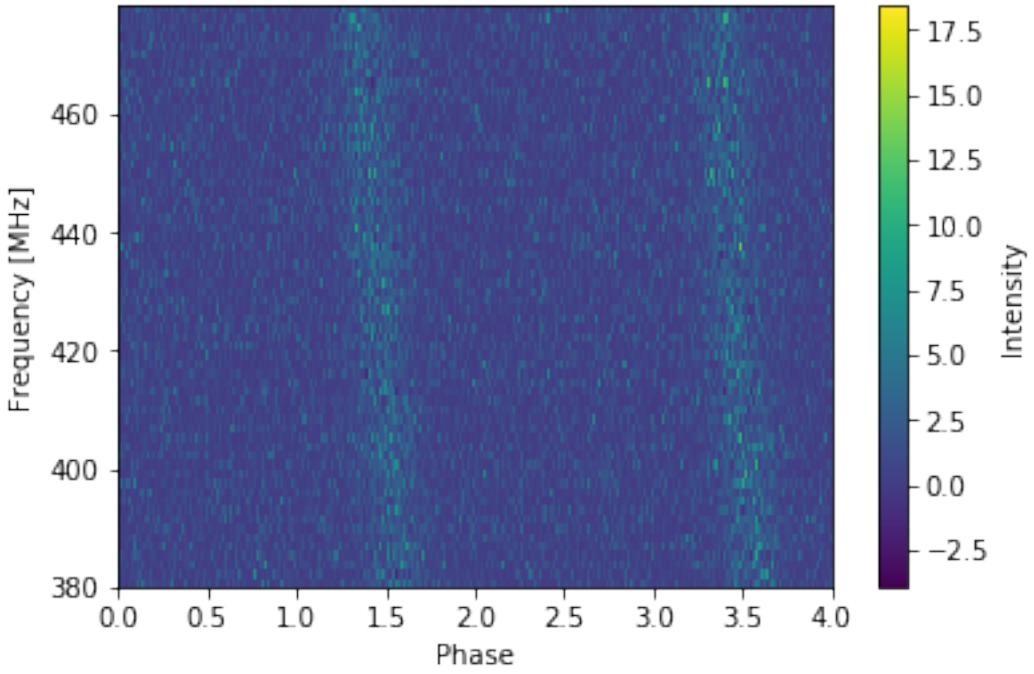
```
WARNING: AstropyDeprecationWarning: The truth value of a Quantity is ambiguous. In␣
↪the future this will raise a ValueError. [astropy.units.quantity]
```

If we plot the results here we find that they are identical within the error of the simulated noise to what we have above.

```
# We can look at the simulated profiles
plt.plot(np.linspace(0,1,2048), sim.profiles.profiles[0])
plt.xlabel("Phase")
plt.show()
plt.close()

# Get the simulated data
sim_data = sim.signal.data

# Get the phases of the pulse
phases = np.linspace(0, sim.tobs/sim.period, len(sim_data[0,:]))
# Plot just the pulses in the first frequency channels
plt.plot(phases, sim_data[0,:], label = sim.signal.dat_freq[0])
plt.ylabel("Intensity")
plt.xlabel("Phase")
plt.legend(loc = 'best')
plt.show()
plt.close()

# Make the 2-D plot of intensity v. frequency and pulse phase. You can see the slight␣
↪dispersive sweep here.
plt.imshow(sim_data, aspect = 'auto', interpolation='nearest', origin = 'lower', \
           extent = [min(phases), max(phases), sim.signal.dat_freq[0].value, sim.
↪signal.dat_freq[-1].value])
plt.ylabel("Frequency [MHz]")
plt.xlabel("Phase")
plt.colorbar(label = "Intensity")
plt.show()
plt.close()
```


1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
Phase


380.0 MHz
10
8
6
4
2
0
−2
Intensity
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
Phase



Note: This tutorial was generated from a Jupyter notebook that can be downloaded here.

1.5.10 Pulse Nulling: Example Notebook

This notebook will serve as an exampe of how to use the pulse nulling feature of the `Pulse Signal Simulator`.

```
# Start by importing the packages we will need for the simulation.
import psrsigsim as pss

# Additional necessary packages
import numpy as np
import matplotlib.pyplot as plt
# helpful magic lines
%matplotlib inline
```

We define a plotting convenience function for later.

```
# Define a function for easier plotting later on/throughout the testing
def plotsignal(signals, nbins=2048):
    # signals can be a list of multiple signals to overplot
    for ii in range(len(signals)):
        # Define the x axis
        phases = np.linspace(0.0, len(signals[ii]), len(signals[ii]))/nbins
        # now plot it
        plt.plot(phases, signals[ii], label="signal %s" % (ii))
    plt.xlim([0.0, np.max(phases)])
    plt.xlabel("Pulse Phase")
    plt.ylabel("Arb. Flux")
    plt.show()
    plt.close()
```

Now we will define some example simulation parameters. The warning generated below may be ignored.

```
# define the required filterbank signal parameters
f0 = 1380 # center observing frequecy in MHz
bw = 800.0 # observation MHz
Nf = 2 # number of frequency channels
F0 = np.double(1.0) # pulsar frequency in Hz
f_samp = F0*2048*10**-6 # sample rate of data in MHz, here 2048 bins across the pulse
subintlen = 1.0 # desired length of fold-mode subintegration in seconds
# Now we define our signal
null_signal = pss.signal.FilterBankSignal(fcent = f0, bandwidth = bw, Nsubband=Nf,\
                                          sample_rate=f_samp, fold=True,␣
↪sublen=subintlen)
```

```
Warning: specified sample rate 0.002048 MHz < Nyquist frequency 1600.0 MHz
```

Now we define an example Gaussian pulse shape. Details on defining a pulse shape from a data array may be found in the exmample notebook in the docs.

```
prof = pss.pulsar.GaussProfile(peak=0.5, width=0.05, amp=1.0)
```

Now we define an example pulsar

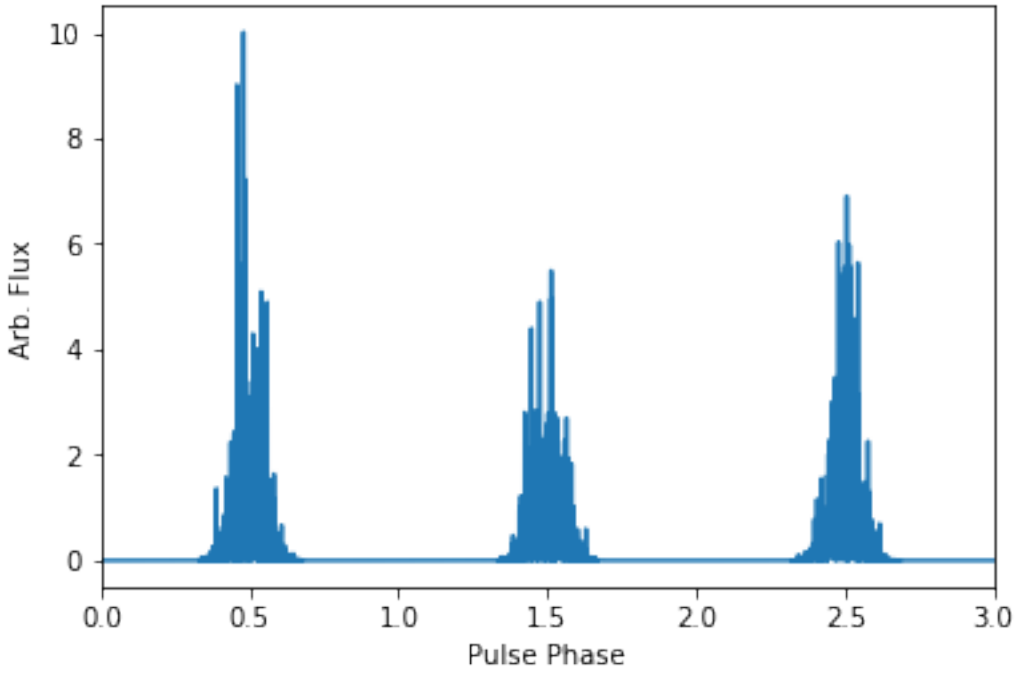
```
# Define the necessary parameters
period = np.double(1.0)/F0 # seconds
flux = 0.1 # Jy
psr_name = "J0000+0000"
# Define the pulsar object
pulsar = pss.pulsar.Pulsar(period=period, Smean=flux, profiles=prof, name=psr_name)
```

Now we actually make the pulsar signal. Note that if the observation length is very long all the data will be saved in memory which may crash the computer or slow it down significantly.

```
# Define the observation time, in seconds
ObsTime = 3.0 # seconds
# make the pulses
pulsar.make_pulses(null_signal, tobs = ObsTime)
```

Now lets take a look at what the signals look like.

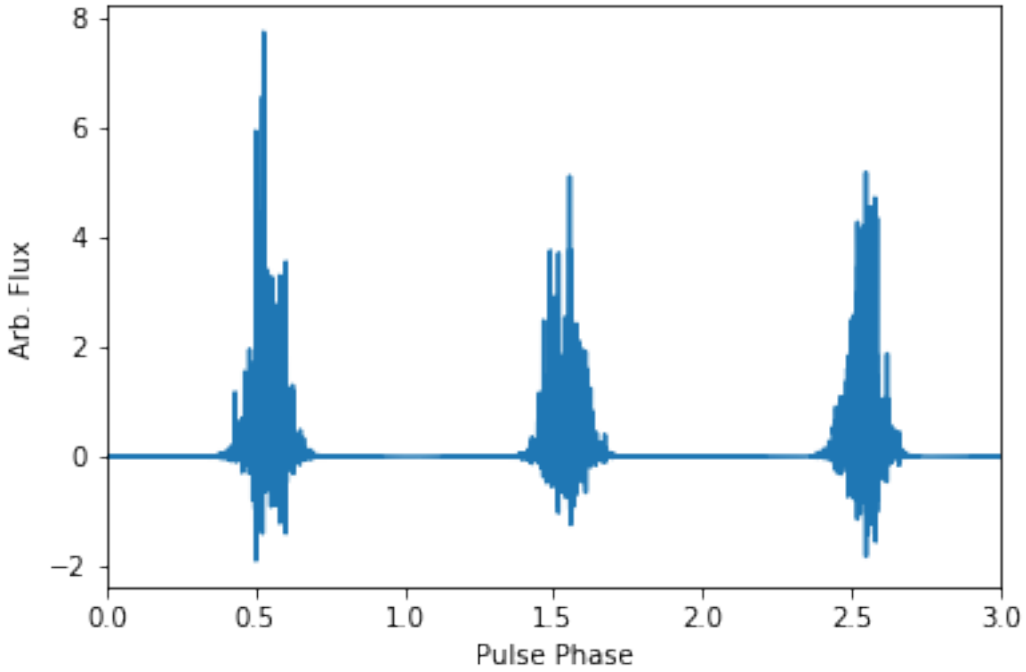
```
# We plot just the first frequency channel, but all pulses simulated
plotsignal([null_signal.data[0,:]])
```



Now we can disperse the simuated data if desired. Note that this is not required, and if you only want to simulate a single frequency channel or simulate coherently dedispersed data, the data does not have to be dispersed.

```
# First define the dispersion measure
dm = 10.0 # pc cm^-3
# Now define the ISM class
ism_ob = pss.ism.ISM()
# Now we give the ISM class the signal and disperse the data
ism_ob.disperse(null_signal, dm)
# If we plot the same pulses as above, you can see that the phase of the pulse has
# been shfited due to the dispersion
plotsignal([null_signal.data[0,:]])
```

```
100% dispersed in 0.001 seconds.
```

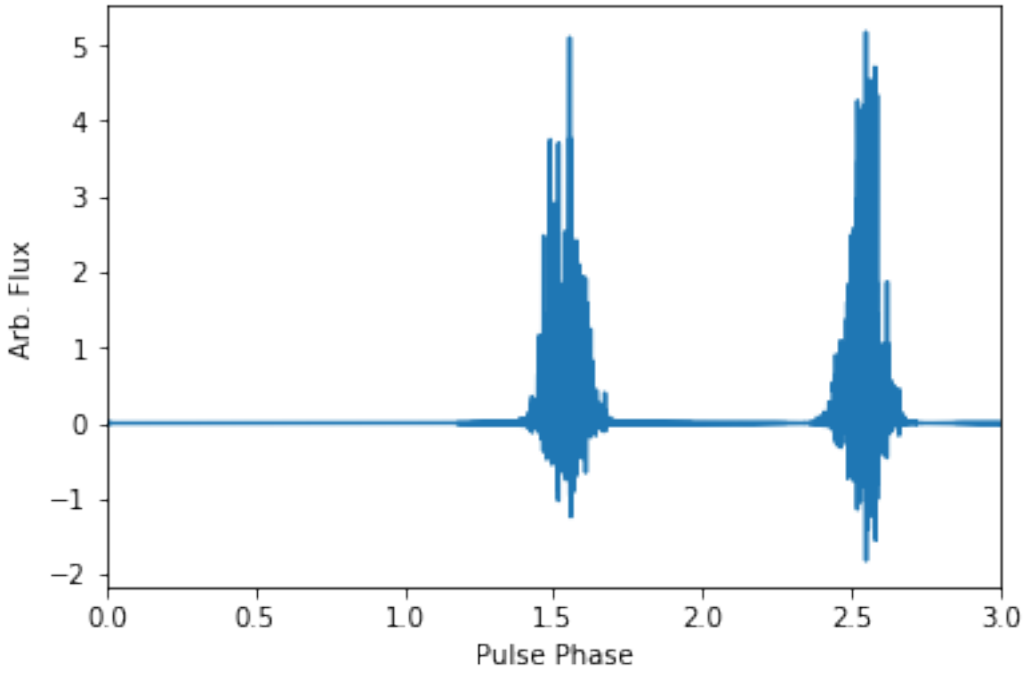


This is where the pulses should be nulled if desired. This can be run easily by giving the pulsar object only the signal class and the null fraction as a value between 0 and 1. The simulator will null as close to the null fraction as desired, and will round to the closest integer number of pulses to null based on the input nulling fraction, e.g. if 5 pulses are simulated and the nulling fraction is 0.5, it will round to null 3 pulses. Additionally, currently only the ability to null the pulses randomly is implemented.

Here we will put in a nulling fraction of 33%

```
pulsar.null(null_signal, 0.34)
```

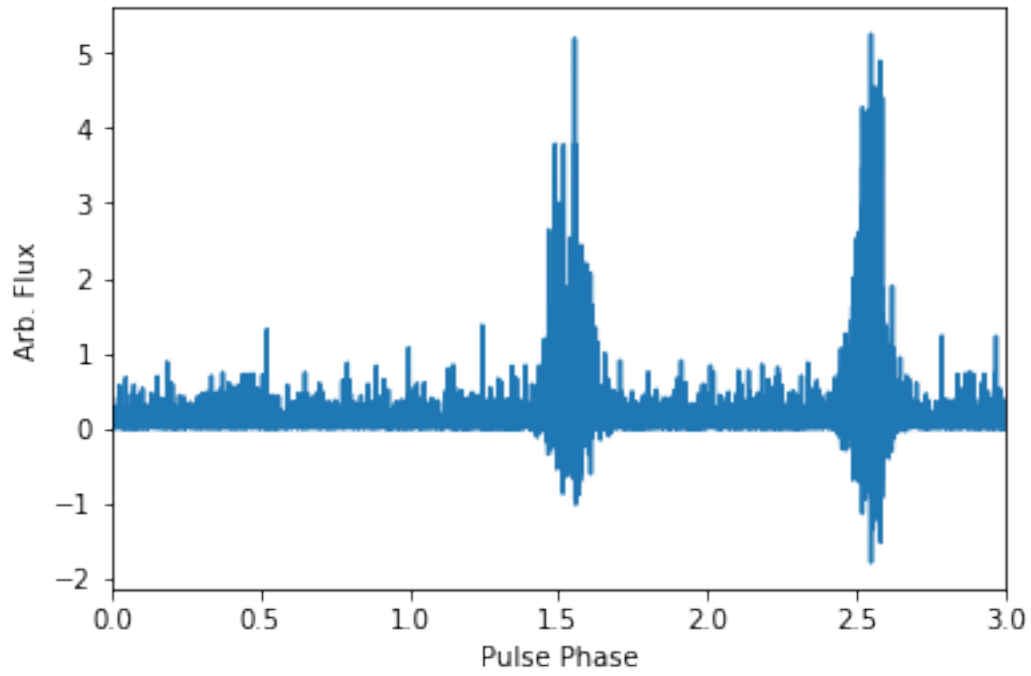
```
# and plot the signal to show the null
plotsignal([null_signal.data[0,:]])
```



We can also add radiometer noise from some observing telescope. This should only be run AFTER the pulsar nulling, but is not required. For our example, we will use the L-band feed for the Arecibo telescope. Note that here since we have set the pulsar flux very high we can easily see the single pulses above the noise.

```
# We define the telescope object
tscope = pss.telescope.telescope.Arecibo()
# Now add radiometer noise; ignore the output here, the noise is added directly to␣
↪the signal
output = tscope.observe(null_signal, pulsar, system="Lband_PUPPI", noise=True)
# and plot the signal to show the added noise
plotsignal([null_signal.data[0,:]])
```

```
WARNING: AstropyDeprecationWarning: The truth value of a Quantity is ambiguous. In␣
↪the future this will raise a ValueError. [astropy.units.quantity]
```



Now we can save the data in a `PSRCHIVE pdv` format. This is done with the `txtfile` class. The save function will dump a new file for every 100 pulses that it writes to the text file. We start by initializing the `txtfile` object. The only input needed here is the `path` variable, which will tell the simulator where to save the data. All files saved will have "_#.txt" added to the end of the `path` variable.

```
txtfile = pss.io.TxtFile(path="PsrSigSim_Simulated_Pulsar.ar")
# Now we call the saving function. Note that depending on the length of the simulated␣
↪data this may take awhile
# the two inputs are the signal and the pulsar objects used to simulate the data.
txtfile.save_psrchive_pdv(null_signal, pulsar)
```

And that's all that there should be to it. Let us know if you have any questions moving forward, or if something is not working as it should be.

1.5.11 Detailed User Interface Information

Signal

Various signal classes in *PsrSigSim* contain the actual data which is passed to various other classes to make realistic pulsar signals.

class psrsigsim.signal.**FilterBankSignal**(*fcent*, *bandwidth*, *Nsubband=512*, *sample_rate=None*, *sublen=None*, *dtype=<class 'numpy.float32'>*, *fold=True*)

A filter bank signal, breaking the time domain signal into RF bins.

Unlike purely time domain signals, *FilterBankSignal* 's are 2-D arrays. Filter banks record the intensity of the signal and can be much more sparsely sampled. The time binning must accurately capture the pulse profile, not the RF oscillations. In practice a filter bank is generated from the observed time domain signal by the telescope backend. We allow for direct filter bank signals to save memory.

Required Args: fcent [float]: central radio frequency (MHz)

bandwidth [float]: radio bandwidth of signal (MHz)

Optional Args:

Nsubband [int]: number of sub-bands, default `512` XUPPI backends use 2048 frequency channels divided between the four Stokes parameters, so 512 per Stokes parameter.

sample_rate [float]: sample rate of data (MHz), default: `None` If no `sample_rate` is given the observation will default to the 20.48 us per sample (about 50 kHz). This is the sample rate for coherently dedispersed filter banks using XUPPI backends.

subint is now depricated for 'FOLD' #subint [bool]: is this a folded subintegration, default `False`

sublen [float]: desired length of data subintegration (sec) if subint is `True`, default: `tobs`. If left as none but subint is `True`, then when pulses are made, the sublen will default to the input observation length, `tobs`

dtype [type]: data type of array, default: `np.float32` supported types are: `np.float32` and `np.int8`

fold [bool]: If *True*, the initialized signal will be folded to some number of subintegrations based on sublen (else will just make a single subintegration). If *False*, the data produced will be single pulse filterbank data. Default is *True*. NOTE - using *False* will generate a large amount of data.

`init_data`(*Nsamp*)
Initialize a data array to store the signal.

Required Args: Nsamp (int): number of data samples

`to_Baseband`()
convert signal to BasebandSignal

`to_FilterBank`(*Nsubband=512*)
convert signal to FilterBankSignal

`to_RF`()
convert signal to RFSignal

`psrsigsim.signal.`**`Signal`**()
helper function to instantiate signals

class `psrsigsim.signal.`**`BasebandSignal`**(*fcent*, *bandwidth*, *sample_rate=None*, *dtype=<class 'numpy.float32'>*, *Nchan=2*)
A time domain base-banded signal.

A *BasebandSignal* covers frequencies from 0 Hz to its bandwidth, e.g. ~1 GHz for L-band GUPPI. In reality the telescope backend basebands the RF signal, but we allow pre-basebanded signals to save memory.

Required Args: fcent [float]: central radio frequency (MHz)

bandwidth [float]: radio bandwidth of signal (MHz)

Optional Args:

sample_rate [float]: sample rate of data (MHz), default: `None` If no `sample_rate` is given the observation will default to the Nyquist frequency. Sub-Nyquist sampling is allowed, but a warning will be generated.

dtype [type]: data type of array, default: `np.float32` any numpy compatible floating point type will work

Nchan [int]: number of frequency channels to simulate, default is 2.

`init_data`(*Nsamp*)
Initialize a data array to store the signal.

Required Args: Nsamp (int): number of data samples

to_Baseband()
convert signal to BasebandSignal

to_FilterBank(*Nsubband=512*)
convert signal to FilterBankSignal

to_RF()
convert signal to RFSignal

class psrsigsim.signal.**RFSignal**(*fcent*, *bandwidth*, *sample_rate=None*, *dtype=<class 'numpy.float32'>*)
a time domain signal at true radio frequency sampling

RFSignals must be sampled at twice the maximum resolved frequency, i.e. a few GHz. As such, RFSignals take up a TON of memory. Consider using BasebandSignal if this is a concern.

Required Args: fcent [float]: central radio frequency (MHz)

bandwidth [float]: radio bandwidth of signal (MHz)

Optional Args:

sample_rate [float]: sample rate of data (MHz), default: None If no sample_rate is given the observation will default to the Nyquist frequency. Sub-Nyquist sampling is allowed, but a warning will be generated.

dtype [type]: data type of array, default: np.float32 any numpy compatible floating point type will work

init_data(*Nsamp*)
Initialize a data array to store the signal.

Required Args: Nsamp (int): number of data samples

to_Baseband()
Convert signal to BasebandSignal

to_FilterBank(*Nsubband=512*)
Convert signal to FilterBankSignal

Parameters

Nsubband [int] Number of frequency subbands.

to_RF()
Convert signal to RFSignal

Returns

psrsigsim.signal.RFSignal

Pulsar

class psrsigsim.pulsar.**Pulsar**(*period*, *Smean*, *profiles=None*, *name=None*)
class for pulsars

The minimal data to instantiate a pulsar is the period, Smean, and pulse profile. The Profile is supplied via a *PulseProfile*-like object.

Parameters

period [float] Pulse period (sec)

Smean [float] Mean pulse flux density (Jy)

profile [*PulseProfile*] Pulse profile or 2-D pulse portrait

name [str] Name of pulsar

make_pulses(*signal*, *tobs*)
generate pulses from Profiles, *PulsePortrait* object

Required Args: signal (Signal-like): signal object to store pulses tobs (float): observation time (sec)

null(*signal*, *null_frac*, *length=None*, *frequency=None*)
Function to simulate pulsar pulse nulling. Given some nulling fraction, will replace simulated pulses with noise until nulling fraction is met. This function should only be run after running any ism or other delays have been added, e.g. disperison, FD, profile evolution, etc., but should be run before adding the radiometer noise ('telescope.observe()'), if nulling is desired.

Parameters

signal [class] [signal class containing the simulated pulses]

null_frac [float] [desired pulsar nulling fraction, given as a] decimal; range of 0.0 to 1.0.

length [float] [desired length of each null in seconds. If not given,] will randomly null pulses. Default is None.

frequency [float] [frequency of pulse nulling, e.g. how often the pulsar] nulls per hour. E.g. if frequency is 2, then the pulsar will null twice per hour for some length of time. If not given, will randomly null pulses. Default is None.

class psrsigsim.pulsar.**PulsePortrait**
Base class for pulse profiles.

A pulse portrait is a set of profiles across the frequency range. They are INTENSITY series, even when using amplitude style signals (like BasebandSignal).

calc_profiles(*phases*, *Nchan=None*)
Calculate the profiles at specified phase(s)

Args: phases (array-like): phases to calc profile

Note: The normalization can be wrong, if you have not run init_profiles AND you are generating less than one rotation.

This is implemented by the subclasses!

init_profiles(*Nphase*, *Nchan=None*)
Generate the profile, evenly sampled.

Parameters

Nphase (int): number of phase bins

class psrsigsim.pulsar.**GaussPortrait**(*peak=0.5*, *width=0.05*, *amp=1*)
Sum of gaussian components.

The shape of the inputs determine the number of gaussian components in the pulse. single float : Single pulse profile made of a single gaussian

1-d array : Single pulse profile made up of multiple gaussians where *n* is the number of Gaussian components in the profile.

Parameters

peak [float)] Center of gaussian in pulse phase.

width [float] Stdev of pulse in pulse phase, default: `0.1`

amp [float] Amplitude of pulse relative to other pulses, *default: ''1'*

Profile is renormalized so that maximum is 1.

See draw_voltage_pulse, draw_intensity_pulse and make_pulses() methods for

more details.

calc_profiles(*phases*, *Nchan=None*)
Calculate the profiles at specified phase(s).

Args: phases (array-like): phases to calc profile

Note: The normalization can be wrong, if you have not run `init_profile` AND you are generating less than one rotation.

init_profiles(*Nphase*, *Nchan=None*)
Generate the profile.

Args: Nphase (int): number of phase bins

class psrsigsim.pulsar.**UserPortrait**
User specified 2-D pulse portrait.

calc_profiles(*phases*, *Nchan=None*)
Calculate the profiles at specified phase(s)

Args: phases (array-like): phases to calc profile

Note: The normalization can be wrong, if you have not run `init_profiles` AND you are generating less than one rotation.

This is implemented by the subclasses!

init_profiles(*Nphase*, *Nchan=None*)
Generate the profile, evenly sampled.

Parameters

Nphase (int): number of phase bins

class psrsigsim.pulsar.**DataPortrait**(*profiles*, *phases=None*)
A pulse portrait generated from data.

The data are samples of the profiles at specified phases. If you have a functional form for the _profiles use *UserProfile* instead.

Parameters

profiles [array, list of lists] Profile data in 2-d array.

phases [array, list of lists (optional)] List of sampled phases. If phases are omitted profile is assumed to be evenly sampled and cover one whole rotation period.

Profile is renormalized so that maximum is 1.

See draw_voltage_pulse, draw_intensity_pulse and make_pulses() methods for

more details.

calc_profiles(*phases*, *Nchan=None*)
Calculate the profile at specified phase(s).

Args: phases (array-like): phases to calc profile

Note: The normalization can be wrong, if you have not run `init_profile` AND you are generating less than one rotation.

init_profiles(*Nphase*, *Nchan=None*)
Generate the profile, evenly sampled.

Parameters

Nphase (int): number of phase bins

class psrsigsim.pulsar.**PulseProfile**
Base class for pulse profiles

Pulse profiles are INTENSITY series, even when using amplitude style signals (like `BasebandSignal`).

calc_profile(*phases*)
Calculate the profile at specified phase(s). This is implemented by the subclasses!

Args: phases (array-like): phases to calc profile

Note: The normalization can be wrong, if you have not run `init_profile` AND you are generating less than one rotation.

calc_profiles(*phases*, *Nchan=None*)
Calculate the profiles at specified phase(s)

Args: phases (array-like): phases to calc profile

Note: The normalization can be wrong, if you have not run `init_profiles` AND you are generating less than one rotation.

This is implemented by the subclasses!

init_profile(*Nphase*)
Generate the profile, evenly sampled.

Args: Nphase (int): number of phase bins

init_profiles(*Nphase*, *Nchan=None*)
Generate the profile, evenly sampled.

Parameters

Nphase (int): number of phase bins

class psrsigsim.pulsar.**GaussProfile**(*peak=0.5*, *width=0.05*, *amp=1*)
Sum of guassian components.

The shape of the inputs determine the number of gaussian components in the pulse.

single float : Single pulse profile made of a single gaussian

1-d array : Single pulse profile made up of multiple gaussians

where *n* is the number of Gaussian components in the profile.

Required Args: N/A

Optional Args: peak (float): center of gaussian in pulse phase, default: `0.5` width (float): stdev of pulse in pulse phase, default: `0.1` amp (float): amplitude of pulse relative to other pulses, default: `1`

Pulses are normalized so that maximum is 1. See draw_voltage_pulse, draw_intensity_pulse and make_pulses() methods for more details.

calc_profiles(*phases*, *Nchan=None*)
Calculate the profiles at specified phase(s).

Args: phases (array-like): phases to calc profile

Note: The normalization can be wrong, if you have not run `init_profile` AND you are generating less than one rotation.

init_profiles(*Nphase*, *Nchan=None*)
Generate the profile.

Args: Nphase (int): number of phase bins

set_Nchan(*Nchan*)
Method to reintialize the portraits with the correct number of frequency channels. Once must run *init_profiles* or *calc_profiles* to remake the *profiles* property.

Note - No phases attribute, function must be updated.

Parameters

Nchan [int] Number of frequency channels.

class psrsigsim.pulsar.**UserProfile**(*profile_func*)
User specified pulse profile

UserProfile's are specified by a function used to compute the profile at arbitrary pulse phase. If you want to generate a profile from empirical data, i.e. a Numpy array, use *DataProfile*.

Required Args:

profile_func (callable): a callable function to generate the profile as a function of pulse phase. This function takes a single, array-like input, a phase or list of phases.

Profile is renormalized so that maximum is 1. See draw_voltage_pulse, draw_intensity_pulse and make_pulses() methods for more details.

calc_profile(*phases*)
Calculate the profile at specified phase(s)

Args: phases (array-like): phases to calc profile

Note: The normalization can be wrong, if you have not run `init_profile` AND you are generating less than one rotation.

calc_profiles(*phases*, *Nchan=None*)
Calculate the profiles at specified phase(s)

Args: phases (array-like): phases to calc profile

Note: The normalization can be wrong, if you have not run `init_profiles` AND you are generating less than one rotation.

This is implemented by the subclasses!

init_profile(*Nphase*)
Generate the profile, evenly sampled.

Args: Nphase (int): number of phase bins

init_profiles(*Nphase*, *Nchan=None*)
Generate the profile, evenly sampled.

Parameters

Nphase (int): number of phase bins

class psrsigsim.pulsar.**DataProfile**(*profiles*, *phases=None*, *Nchan=None*)
A set of pulse profiles generated from data.

The data are samples of the profile at specified phases. If you have a functional form for the profile use *UserProfile* instead.

Required Args: profile (array-like): profile data

Optional Args:

phases (array-like): list of sampled phases. If phases are omitted profile is assumed to be evenly sampled and cover one whole rotation period.

Profile is renormalized so that maximum is 1. See draw_voltage_pulse, draw_intensity_pulse and make_pulses() methods for more details.

calc_profiles(*phases*, *Nchan=None*)
Calculate the profile at specified phase(s).

Args: phases (array-like): phases to calc profile

Note: The normalization can be wrong, if you have not run `init_profile` AND you are generating less than one rotation.

init_profiles(*Nphase*, *Nchan=None*)
Generate the profile, evenly sampled.

Parameters

Nphase (int): number of phase bins

set_Nchan(*Nchan*)
Method to reintialize the portraits with the correct number of frequency channels. Once must run *init_profiles* or *calc_profiles* to remake the *profiles* property.

Note - Has same issue as set_Nchan before.

Parameters

Nchan [int] Number of frequency channels.

Telescope

class psrsigsim.telescope.**Telescope**(*aperture*, *area=None*, *Tsys=None*, *name=None*)
contains: observe(), noise(), rfi() methods

add_system(*name=None*, *receiver=None*, *backend=None*)
append new system to dict systems

init_signal(*system*)
instantiate a signal object with same Nt, Nf, bandwidth, etc as the system to be used for observation

observe(*signal*, *pulsar*, *system=None*, *noise=False*, *ret_resampsig=False*)

Parameters

signal – Signal() instance

pulsar – Pulsar() object, necessary for radiometer noise scaling

system – dict key for system to use

noise [bool] If True will add radiometer noise to the signal data.

ret_resampsig [bool] If True will return the resampled signal as a numpy array. Otherwise will not return anything.

class psrsigsim.telescope.**Receiver**(*response=None*, *fcent=None*, *bandwidth=None*, *Trec=35*, *name=None*)

Telescope receiver. A *Receiver* must be instantiated with either a callable response function or `fcent` and `bandwidth` to use a flat response.

Required Args: N/A

Optional Args: response (callable): frequency response function ("bandpass") of receiver.

fcent (float): center frequency of receiver with flat response (MHz)

bandwidth (float): bandwidth of receiver with flat response (MHz)

Trec (float): receiver temperature (K) for radiometer noise level, default: `35`

radiometer_noise(*signal*, *pulsar*, *gain=1*, *Tsys=None*, *Tenv=None*)

Add radiometer noise to a signal.

Tsys = Tenv + Trec, unless Tsys is given (just Trec if no Tenv)

flux density fluctuations: sigS from Lorimer & Kramer eq 7.12

psrsigsim.telescope.**response_from_data**(*fs*, *values*)

generate a callable response function from discrete data Data are interpolated.

ISM

class psrsigsim.ism.**ISM**

Class for modeling interstellar medium effects on pulsar signals.

FD_shift(*signal*, *FD_params*)

This calculates the delay that will be added due to an arbitrary number of input FD parameters following the NANOGrav standard as defined in Arzoumanian et al. 2016. It will then shift the pulse profiles by the appropriate amount based on these parameters.

FD values should be input in units of seconds, frequency array in MHz FD values can be a list or an array

$$\Delta t_{\mathrm{FD}} = \sum_{i=1}^{n} c_i \log\left(\frac{\nu}{1\ \mathrm{GHz}}\right)^i .$$

convolve_profile(*profiles*, *convolve_array*, *width=2048*)

Function to convolve some array generated by a function with the previously assigned pulsar pulse profiles. Main use case is in convolving exponential scattering tails with the input pulse profiles, however any input array can be convolved.

NOTE: This function only returns the array of convolved profiles, it does NOT reassign the pulsar objects profiles.

Parameters

profiles [array] [data array of pulse profiles generated with the] 'calc_profiles' function.

convolve_array [array] [data array representing the function that] will be convolved with the pulse profiles. Should be the same shape as 'profiles'.

width [int] [number of bins desired from the resulting convolved] profiles. Default is 2048 bins across the profile. Should be the same number as the original input profiles.

disperse(*signal*, *dm*)
Function to calculate the dispersion per frequency bin for $1/\nu^2$ dispersion.

$$\Delta t_{\rm DM} = 4.15 \times 10^6 \text{ ms} \times \text{DM} \times \frac{1}{\nu^2}$$

scale_dnu_d(*dnu_d*, *nu_i*, *nu_f*, *beta=3.6666666666666665*)
Scaling law for scintillation bandwidth as a function of frequency.

Parameters

dnu_d [float] [scintillation bandwidth [MHz]]

nu_i [float] [reference frequency at which du_d was measured [MHz]]

nu_f [float] [frequency (or frequency array) to scale dnu_d to [MHz]]

beta [float] [preferred scaling law for dnu_d, default is for a] Kolmogorov medium (11/3)

scale_dt_d(*dt_d*, *nu_i*, *nu_f*, *beta=3.6666666666666665*)
Scaling law for scintillation timescale as a function of frequency.

Parameters

dt_d [float] [scintillation timescale [seconds]]

nu_i [float] [reference frequency at which du_d was measured [MHz]]

nu_f [float] [frequency (or frequency array) to scale dnu_d to [MHz]]

beta [float] [preferred scaling law for dt_d, default is for a] Kolmoogorov medium (11/3)

scale_tau_d(*tau_d*, *nu_i*, *nu_f*, *beta=3.6666666666666665*)
Scaling law for the scattering timescale as a function of frequency.

Parameters

tau_d [float] [scattering timescale [seconds?]]

nu_i [float] [reference frequency at which du_d was measured [MHz]]

nu_f [float] [frequency (or frequency array) to scale dnu_d to [MHz]]

beta [float] [preferred scaling law for tau_d, default is for a] Kolmoogorov medium (11/3)

scatter_broaden(*signal*, *tau_d*, *ref_freq*, *beta=3.6666666666666665*, *convolve=False*, *pulsar=None*)
Function to add scatter broadening delays to simulated data. We offer two methods to do this, one where the delay is calcuated and the pulse signals is directly shifted by the calculated delay (as done in the disperse function), or the scattering delay exponentials are directy convolved with the pulse profiles. If this option is chosen, the scatter broadening must be done BEFORE pulsar.make_pulses() is run.

Parameters

signal [object] [signal class object which has been previously defined]

tau_d [float] [scattering delay [seconds]]

ref_freq [float] [reference frequency [MHz] at which tau_d was measured]

beta [float] [preferred scaling law for tau_d, default is for a] Kolmoogorov medium (11/3)

convolve [bool] [If False, signal will be directly shifted in time by] scattering delay; if True, scattering delay tails will be directly convolved with the pulse profiles.

pulsar [object] [previously defined pulsar class object with profile] already assigned

IO

class psrsigsim.io.**BaseFile**(*path=None*)
Base class for making files for the PsrSigSim Signal object.

append()
Method for append data to an already existing PSS signal file. Must be implemented in subclass!

load()
Method for loading saved PSS signal files. Must be implemented in subclass!

save(*signal*)
Save PSS signal file to disk. Must be implemented in subclass!

to_psrfits()
Convert file to PSRFITS file. Must be implemented in subclass!

to_txt()
Convert file to txt file. Must be implemented in subclass!

class psrsigsim.io.**PSRFITS**(*path=None*, *obs_mode=None*, *template=None*, *copy_template=False*, *fits_mode='copy'*)
A class for saving PsrSigSim signals as PSRFITS standard files.

Parameters

path: str name and path of new psrfits file that will be saved

obs_mode: str what type of observation is the data, SEARCH, PSR, etc.

template: str the path and name of the template fits file that will be loaded

copy_template: bool Does nothing?

fits_mode: str How we want to save the data, right now just 'copy' is valid

append(*signal*)
Method for appending data to an already existing PSS signal file.

copy_psrfit_BinTables(*ext_names='all'*)
Method to copy BinTables from the PSRFITS file given as the template.

Parameters

ext_names [list, 'all'] List of BinTable Extensions to copy. Defaults to all, but does not copy DATA array in SUBINT BinTable.

load()
Method for loading saved PSS signal files. These files will have an additional BinTable extension, 'PSRSIGSIM', that only contains a header with the various PsrSigSim parameters written for references.

make_signal_from_psrfits()
Method to make a signal from the PSRFITS file given as the template. For subintegrated data will assume the initial period is the pulsar period given in the PSRPARAM header.

TODO: Currently does not support generating 'SEARCH' mode data from a psrfits file

Returns

psrsigsim.Signal

save(*signal*, *pulsar*, *phaseconnect=False*, *parfile=None*, *MJD_start=56000.0*, *segLength=60.0*, *inc_len=0.0*, *ref_MJD=56000.0*, *usePint=True*, *eq_wts=True*)
Save PSS signal file to disk. Currently only one mode of doing this is supported. Saved data can be phase

connected but PSRFITS file metadata must be edited appropriately as well and requires the following input:

Parameters

signal [class] [signal type class (currently only filterbank is supported)] used to get the data array to save and other metadata.

pulsar [class] [pulsar type class used to generate the signal, used for] metadata access.

phaseconnect [bool] [If *False*, will not attempt to phase connect data] rewrite polycos, etc. If *True*, will attempt to phase connect data and all other inputs must be provided.

parfile [string] [path to par file used to generate the polycos. The observing frequency, and observatory will] come from the par file.

MJD_start [float] [Start MJD of the polyco. Should start no later than the beginning of the observation.]

segLength [float] [Length in minutes of the range covered by the polycos generated. Default is 60 minutes.]

ref_MJD [float] [initial time to reference the observations to (MJD). This value] should be the start MJD (fraction if necessary) of the first file, default is 56000.0.

inc_len [float] [time difference (days) between reference MJD and new phase connected] MJD, default is 0 (e.g. no time difference).

usePINT [bool] [Method used to generate polycos. Currently only PINT is supported.]

eq_wts [bool] [If *True* (default), replaces the data weights so that each subintegration and] frequency channel have an equal weight in the file. If *False*, just copies the weights from the template file.

set_sky_info()
Enter various astronomical and observing data into PSRFITS draft.

to_psrfits()
Convert file to PSRFITS file.

to_txt()
Convert file to txt file.

class psrsigsim.io.**TxtFile**(*path=None*)
A class for saving PsrSigSim signals as text files. Multiple different text file data types may be supported, but currently only the PSRCHIVE pdv function output is supported.

Parameters

path: name and path of new text file that will be saved

append()
Method for append data to an already existing PSS signal file. Must be implemented in subclass!

load()
Method for loading saved PSS signal files. Must be implemented in subclass!

save(*signal*)
Save PSS signal file to disk. Must be implemented in subclass!

save_psrchive_pdv(*signal*, *pulsar*)
Function to save simulated data in the same format as the PSRCHIVE pdf function. To avoid large file sizes, every hundred pulses the data will be saved as a text file. Currently one a single polarization (total intensity) is supported. Inputs are:

Parameters

signal [class] [signal class object, currently only filterbank is supported]

pulsar [class] [pulsar class object]

filename [string] [desired name of source/output file. Output files will be saved as] 'filename'_#.txt, where # is the chronological number of the files being saved.

to_psrfits()
Convert file to PSRFITS file. Must be implemented in subclass!

to_txt()
Convert file to txt file. Must be implemented in subclass!

Simulate

class psrsigsim.simulate.**Simulation**(*fcent=None*, *bandwidth=None*, *sample_rate=None*, *dtype=<class 'numpy.float32'>*, *Npols=1*, *Nchan=512*, *sublen=None*, *fold=True*, *period=None*, *Smean=None*, *profiles=None*, *tobs=None*, *name=None*, *dm=None*, *tau_d=None*, *tau_d_ref_f=None*, *aperture=None*, *area=None*, *Tsys=None*, *tscope_name=None*, *system_name=None*, *rcvr_fcent=None*, *rcvr_bw=None*, *rcvr_name=None*, *backend_samprate=None*, *backend_name=None*, *tempfile=None*, *parfile=None*, *psrdict=None*)

convenience class for full simulations.

Necessary information includes all minimal parameters for instances of each other class, Signal, Pulsar, ISM, Telescope.

Input may be specified manually, from a pre-made parfile with additional input, e.g. for the Signal, or from a premade dictionary with appropriate keys.

Parameters

fcent [float]] Central radio frequency (MHz)

bandwidth [float] Radio bandwidth of signal (MHz)

Nsubband [int] Number of sub-bands, default `512` XUPPI backends use 2048 frequency channels divided between the four Stokes parameters, so 512 per Stokes parameter.

sample_rate [float] Sample rate of data (MHz), default: `None` If no `sample_rate` is given the observation will default to the 20.48 us per sample (about 50 kHz). This is the sample rate for coherently dedispersed filter banks using XUPPI backends.

sublen [float] Desired length of data subintegration (sec) if subint is `True`, default: `tobs`. If left as none but subint is `True`, then when pulses are made, the sublen will default to the input observation length, `tobs`

dtype [type] Data type of array, default: `np.float32` supported types are: `np.float32` and `np.int8`

fold [bool] If *True*, the initialized signal will be folded to some number of subintegrations based on sublen (else will just make a single subintegration). If *False*, the data produced will be single pulse filterbank data. Default is *True*. NOTE - using *False* will generate a large amount of data.

period [float] Pulse period (sec)

Smean [float] Mean pulse flux density (Jy)

profile [array or function or Pulse Profile/Portrait Class]

Pulse profile or 2-D pulse portrait, this can take four forms:

array - either an array of Gaussian components in the order [peak phase, width, amplitude] OR A data array representative of the pulse profile, or samples of the profile from phases between 0 and 1. Data array must have more than 3 points.

function - function defining the shape of the profile given a of input phases. CURRENTLY NOT IMPLEMENTED.

class - predefined PsrSigSim Pulse Profile or Pulse Portrait class object.

tobs [float] Total simulated observing time in seconds

name [str] Name of pulsar

dm [float] Dispersion measure of the pulsar (pc cm^-3)

tau_d [float] Scattering timescale to use (s)

tau_d_ref_f [float] reference frequency for the input scattering timescale (MHz)

aperture [float] Telescop aperture (m)

area [float] Collecting area (m^2) (if omitted, assume circular single dish)

Tsys [float] System temperature (K) of the telescope (if omitted use Trec)

tscope_name [string] Name of the telescope. If GBT or Arecibo, will use predefined parameters.

system_name [string] Name of telescope system, backend-recviever combination. May be a list.

rcvr_fcent: float Center frequency of the telescope reciever. May be a list.

rcvr_bw [float] Bandwidth of the telescope reciever. May be a list.

rcvr_name [string] Name of the telescope reciever. May be a list.

backend_samprate [float] Sampling rate (in MHz) of the telescope backend. May be a list.

backend_name [string] Name of the telescope backend. May be a list.

tempfile [string] Path to template psrfits file to use for saving simulated data.

parfile [string] Path to pulsar par file to read in to use for pulsar parameters

psrdict [dictionary] Dictionary of input parameters to generate simualted data from. Keys should be the same as possible input values listed above.

init_ism()
Function to initialize the ISM from the input parameters.

init_profile()
Function to initialize a profile object from input.

init_pulsar()
Function to initialize a pulsar from the input parameters. NOTE - Must have initialized the profile before running this.

init_signal(*from_template=False*)
Function to initialize a signal from the input parameters.

Parameters

from_template [bool] If True, will use the input template file to initialize the signal. If False will use other input values to initialize the signal.

init_telescope()
Function to initialize the telescope from input parameters.

params_from_dict(*psrdict*)
Function to take the input dictionary and assign values from that.

params_from_par(*parfile*)
Function to take input par file and assign values from that.

save_simulation(*outfile='simfits'*, *out_format='psrfits'*, *phaseconnect=False*, *parfile=None*, *ref_MJD=56000.0*, *MJD_start=55999.9861*)
Function to save the simulated data in a default format. Currently only PSRFITS is supported.

Parameters

outfile [string] Path and name of output save file. If not provided, output file is "simfits".

out_format [string] Format of output file (not case sensitive). Options are: 'psrfits' - PSRFITS format. Requires template file. 'pdv' - PSRCHIVE pdv format. Output is a text file.

phaseconnect [bool] Make sure to phase connect output data if output format is PSRFITS.

parfile [string] Parfile to use to make phase connection polycos. If none supplied will attempt to create one.

ref_MJD [float] Reference MJD for phase connection.

MJD_start [float] Desired start time of the simulated observation. Needed for phase connection.

simulate(*from_template=False*)
Function to run the full simulation.

Parameters

from_template [bool] If True, will use the input template file to initialize the signal. If False will use other input values to initialize the signal.

twoD [bool] If True, will generate a 2-D profile array, else will do a 1-D and will tile the profile in frequency.

Utilities

psrsigsim.utils.**make_quant**(*param*, *default_unit*)
Convenience function to intialize a parameter as an astropy quantity.

Parameters

param [attribute] Parameter to initialize.

default_unit [string] Name of an astropy unit, set as default for this parameter.

Returns

An astropy quantity

Examples

self.f0 = make_quant(f0,'MHz')

psrsigsim.utils.**make_par**(*signal*, *pulsar*, *outpar='simpar.par'*)
Function to create a par file for simulated pulsar.

Parameters

signal [class] PsrSigSim Signal class object

pulsar [class] PsrSigSim Pulsar class object

outpar [string] Name of output par file.

psrsigsim.utils.**shift_t**(*y*, *shift*, *dt=1*)
Shift timeseries data in time. Shift array, y, in time by amount, shift. For dt=1 units of samples (including fractional samples) are used. Otherwise, shift and dt are assumed to have the same physical units (i.e. seconds).

Parameters

y [array like, shape (N,), real] Time series data.

shift [int or float] Amount to shift

dt [float] Time spacing of samples in y (aka cadence).

Returns

out [ndarray] Time shifted data.

Examples

>>>shift_t(y, 20) # shift data by 20 samples

>>>shift_t(y, 0.35, dt=0.125) # shift data sampled at 8 Hz by 0.35 sec

Uses np.roll() for integer shifts and the Fourier shift theorem with real FFT in general. Defined so positive shift yields a "delay".

1.5.12 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

Types of Contributions

Report Bugs

Report bugs at https://github.com/PsrSimSig/PsrSigSim/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

Write Documentation

PsrSimSig could always use more documentation, whether as part of the official PsrSimSig docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/PsrSigSim/PsrSigSim/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

Get Started!

Ready to contribute? Here's how to set up *psrsigsim* for local development.

1. Fork the *psrsigsim* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/psrsigsim.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv psrsigsim
$ cd psrsigsim/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 psrsigsim tests
$ python setup.py test or py.test
$ tox
```

To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7, 3.5, and 3.6. Check https://travis-ci.org/PsrSigSim/PsrSigSim/pull_requests and make sure that the tests pass for all supported Python versions.

Tips

To run a subset of tests:

```
$ py.test tests.test_psrsigsim
```

1.5.13 Contributor Covenant Code of Conduct

Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at this email. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 1.4, available here.

For answers to common questions about this code of conduct, see https://www.contributor-covenant.org/faq

1.5.14 Credits

Development Leads

- Jeffrey S. Hazboun <jeffrey.hazboun@nanograv.org>
- Paul T. Baker <paul.baker@nanograv.org>
- Brent Shapiro-Albert <bjs0024@mix.wvu.edu>

Contributors

- Joseph D. Romano
- Cassidy M. Wagner
- Amelia M. Henkel
- Paul Brook
- Michael T. Lam
- Jacob Hesse

1.5.15 History

1.0.0 (2020-10-15)

*First release, coinciding with chromatic timing parameter paper.

0.2.0 (2020-08-17)

- Pre Release version for testing.

0.1.0 (2018-01-16)*

- First release on PyPI.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

U