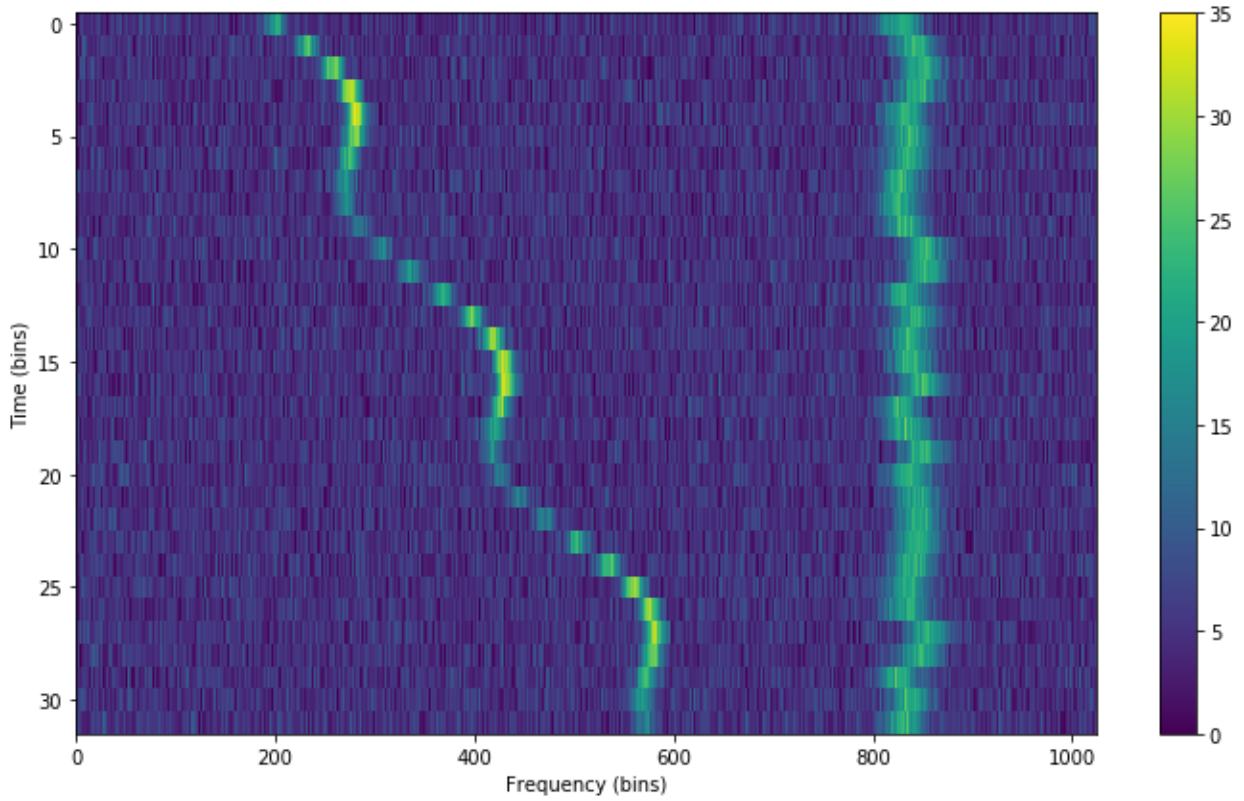
setigen Documentation

Release 2.1.0

Bryan Brzycki

Jul 16, 2021

Contents



`setigen` is a Python library for generating and injecting artificial narrow-band signals into radio requency data. `setigen` interfaces primarily with two types of data: spectrograms or dynamic spectra, saved in two-dimensional NumPy arrays or filterbank files (`.fil` extension), and raw voltages (GUPPI RAW files). Both data formats are instrumental to Breakthrough Listen's data collection and analysis pipelines.

`setigen` allows the user to generate synthetic signals quickly in the time-frequency domain in the form of data Frames. Furthermore, the user may inject these synthetic signals into real observational data loaded from filterbank files. `setigen` plays well with the `blimpy` package.

The `setigen.voltage` module enables the synthesis of GUPPI RAW files via synthetic real voltage "observations" and a software signal processing pipeline that implements a polyphase filterbank, mirroring actual BL hardware. The voltage module supports single and multi-antenna RAW files, and can be GPU accelerated via CuPy.

Breakthrough Listen @ Berkeley: https://seti.berkeley.edu/listen/

CHAPTER 1

Table of Contents

1.1 Installation

You can use `pip` to install the package automatically:

```
pip install setigen
```

Alternately, you can clone the repository and install it directly. At the command line, execute:

```
git clone git@github.com:bbrzycki/setigen.git
python setup.py install
```

One of the dependencies for `setigen` is `blimpy`, which is used for working with BL filterbank data products. Note that you can still generate synthetic data frames even without observational data!

Because of how the `bitshuffle` package was written, if you are working with HDF5 data products (e.g. ending with ".hdf5" or ".h5"), you may also need to do the following, especially if you'd like to save `setigen` frame data as HDF5 files:

```
pip install -U git+https://github.com/h5py/h5py
pip install git+https://github.com/kiyo-masui/bitshuffle
```

Note: this can lead to `h5py` compatibility issues with older versions of Tensorflow. Some work-arounds: if possible, work primarily with filterbank files, or use multiple Python environments to separate data handling and Tensorflow work.

1.1.1 To use GPU with setigen.voltage

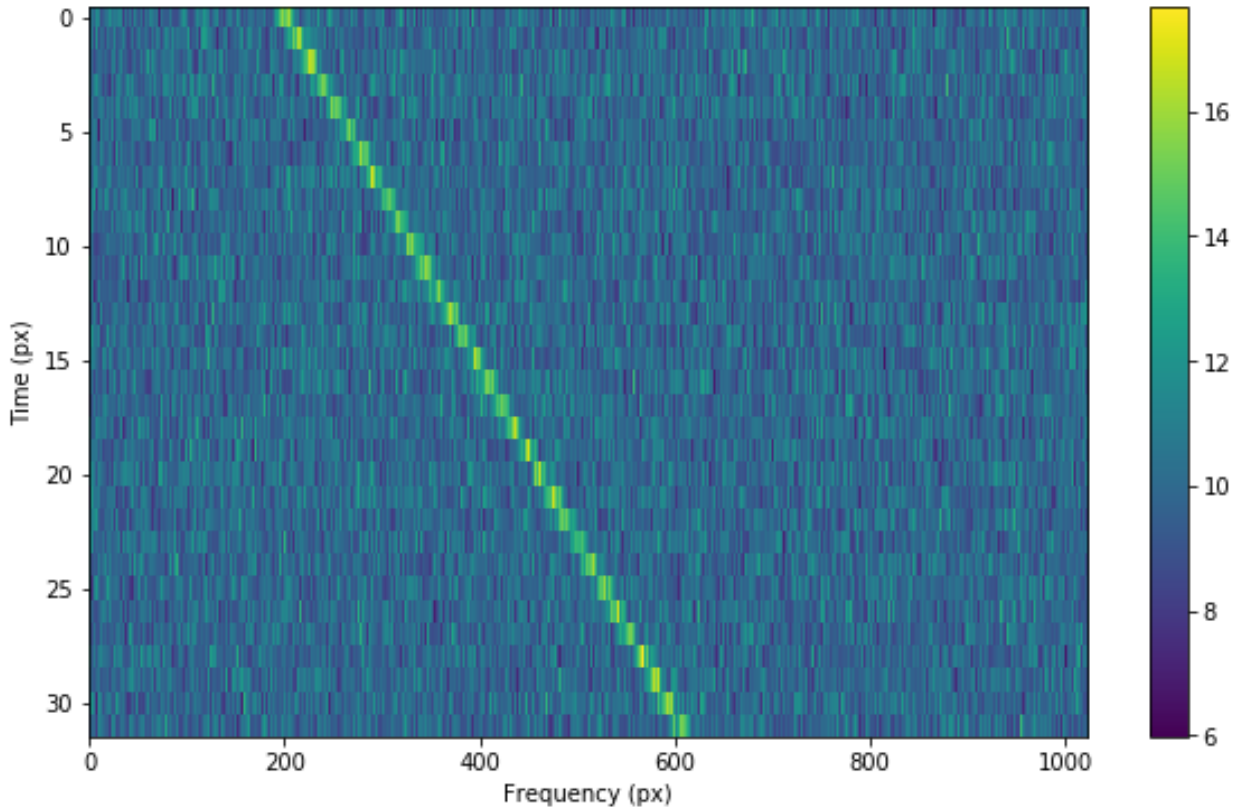
`setigen.voltage`'s GPU acceleration is powered by CuPy (https://docs.cupy.dev/en/stable/install.html). Installation is not required to use vanilla `setigen` or the voltage module, but it is highly recommended to accelerate voltage computations. While it isn't used directly by `setigen`, you may also find it helpful to install `cusignal` (https://github.com/rapidsai/cusignal) for access to CUDA-enabled versions of `scipy` functions when writing custom voltage signal source functions.

1.2 Getting started

The heart of `setigen` is the Frame object. For signal injection and manipulation, we call each snippet of time-frequency data a "frame." There are two main ways to initialize frames, starting from either resolution/size parameters or existing observational data.

Here's a minimal working example for a purely synthetic frame, injecting a constant intensity signal into a background of chi-squared noise. Parameters in `setigen` are specified either in terms of SI units (Hz, s) or `astropy.units`, as in the example:

```
from astropy import units as u
import setigen as stg
import matplotlib.pyplot as plt

frame = stg.Frame(fchans=1024*u.pixel,
                  tchans=32*u.pixel,
                  df=2.7939677238464355*u.Hz,
                  dt=18.253611008*u.s,
                  fch1=6095.214842353016*u.MHz)
noise = frame.add_noise(x_mean=10, noise_type='chi2')
signal = frame.add_signal(stg.constant_path(f_start=frame.get_frequency(index=200),
                                            drift_rate=2*u.Hz/u.s),
                          stg.constant_t_profile(level=frame.get_intensity(snr=30)),
                          stg.gaussian_f_profile(width=40*u.Hz),
                          stg.constant_bp_profile(level=1))

fig = plt.figure(figsize=(10, 6))
frame.render()
plt.savefig('frame.png', bbox_inches='tight')
plt.show()
```



This simple signal can also be generated using the method `frame.add_constant_signal`, which is optimized for created signals of constant intensity and drift rate in large frames:

```
frame.add_constant_signal(f_start=frame.get_frequency(200),
                          drift_rate=2*u.Hz/u.s,
                          level=frame.get_intensity(snr=30),
                          width=40*u.Hz,
                          f_profile_type='gaussian')
```

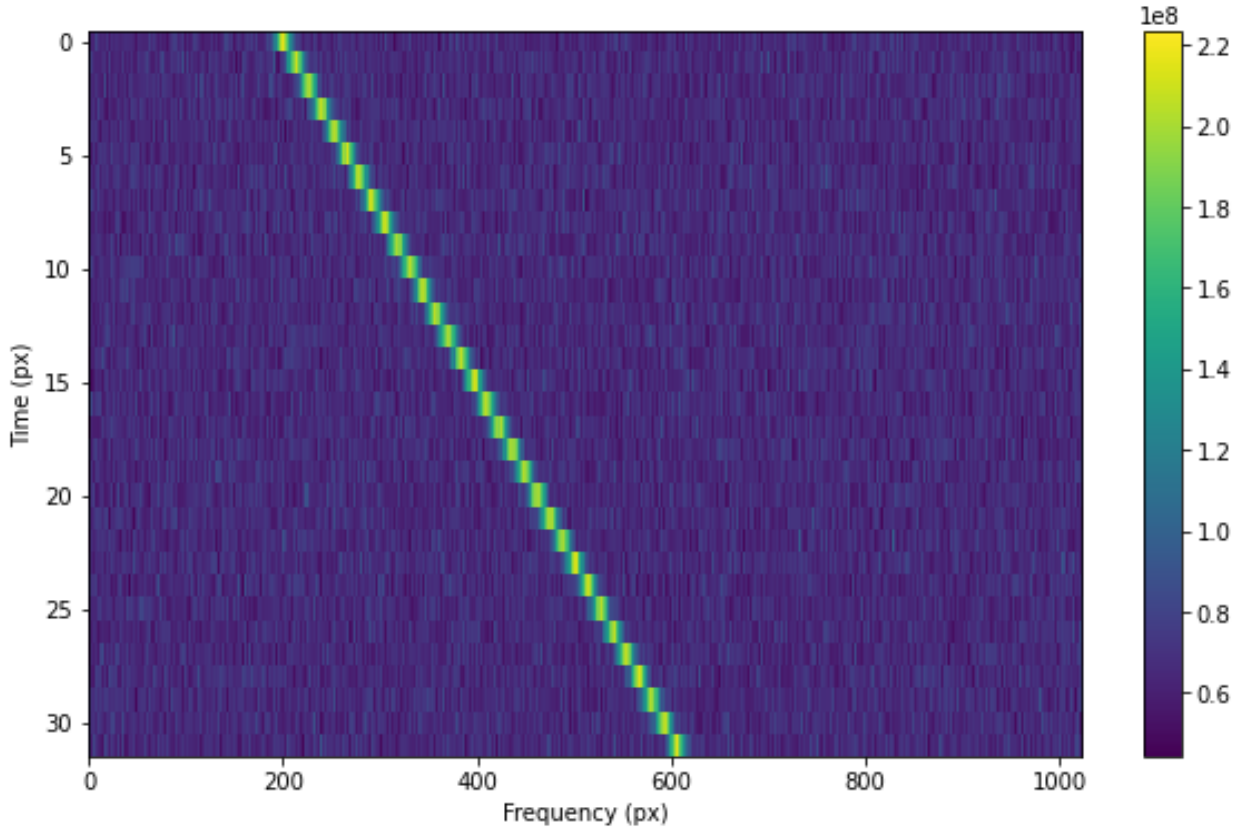
Similarly, here's a minimal working example for injecting a signal into a frame of observational data (from a blimpy Waterfall object). Note that in this example, the observational data also has dimensions 32x1024 to make it easy to visualize here.

```
from astropy import units as u
import setigen as stg
import blimpy as bl
import matplotlib.pyplot as plt

data_path = 'path/to/data.fil'
waterfall = bl.Waterfall(data_path)
frame = stg.Frame(waterfall=waterfall)
frame.add_signal(stg.constant_path(f_start=frame.get_frequency(200),
                                   drift_rate=2*u.Hz/u.s),
                 stg.constant_t_profile(level=frame.get_intensity(snr=30)),
                 stg.gaussian_f_profile(width=40*u.Hz),
                 stg.constant_bp_profile(level=1))

fig = plt.figure(figsize=(10, 6))
frame.render()
```

(continues on next page)

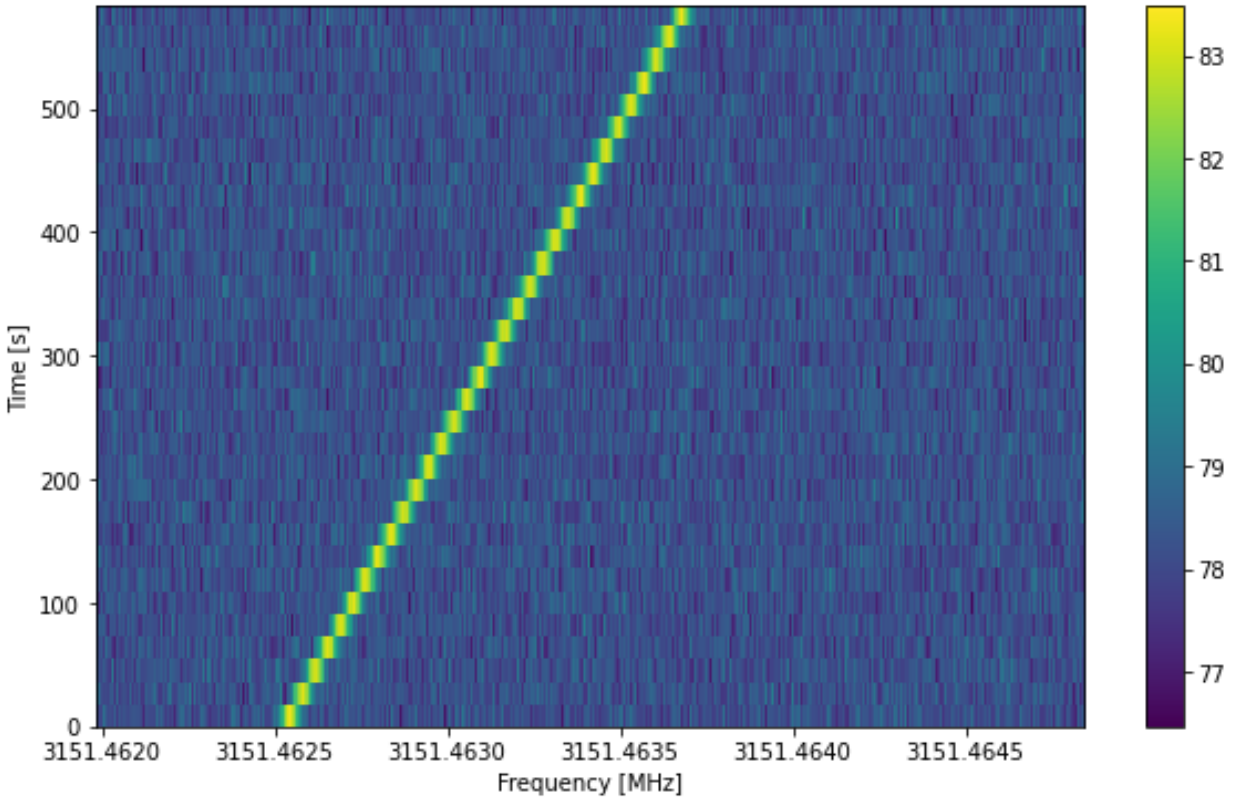
(continued from previous page)

```
plt.show()
```



We can also view this using blimpy's plotting style:

```
fig = plt.figure(figsize=(10, 6))
frame.bl_render()
plt.show()
```



Usually, filterbank data is saved with frequencies in descending order, with the first frequency bin centered at `fch1`. `setigen` works with data in increasing frequency order, and will reverse the data order when appropriate if the frame is initialized with such an observation. However, if you are working with data or would like to synthesize data for which `fch1` should be the minimum frequency, set `ascending=True` when initializing the Frame object. Note that if you initialize Frame using a filterbank file with frequencies in increasing order, you do not need to set `ascending` manually.

```
frame = stg.Frame(fchans=1024*u.pixel,
                  tchans=32*u.pixel,
                  df=2.7939677238464355*u.Hz,
                  dt=18.253611008*u.s,
                  fch1=6095.214842353016*u.MHz,
                  ascending=True)
```

Assuming you have access to a data array, with corresponding resolution information, you can can also initialize a frame as follows. Just make sure that your data is already arranged in the desired frequency order; setting the `ascending` parameter will only affect the frequency values that are mapped to the provided data array.

```
my_data = # your 2D array
frame = stg.Frame.from_data(df=2.7939677238464355*u.Hz,
                            dt=18.253611008*u.s,
                            fch1=6095.214842353016*u.MHz,
                            ascending=True,
                            data=my_data)

frame.render()
```

1.3 Basic usage

1.3.1 Adding a basic signal

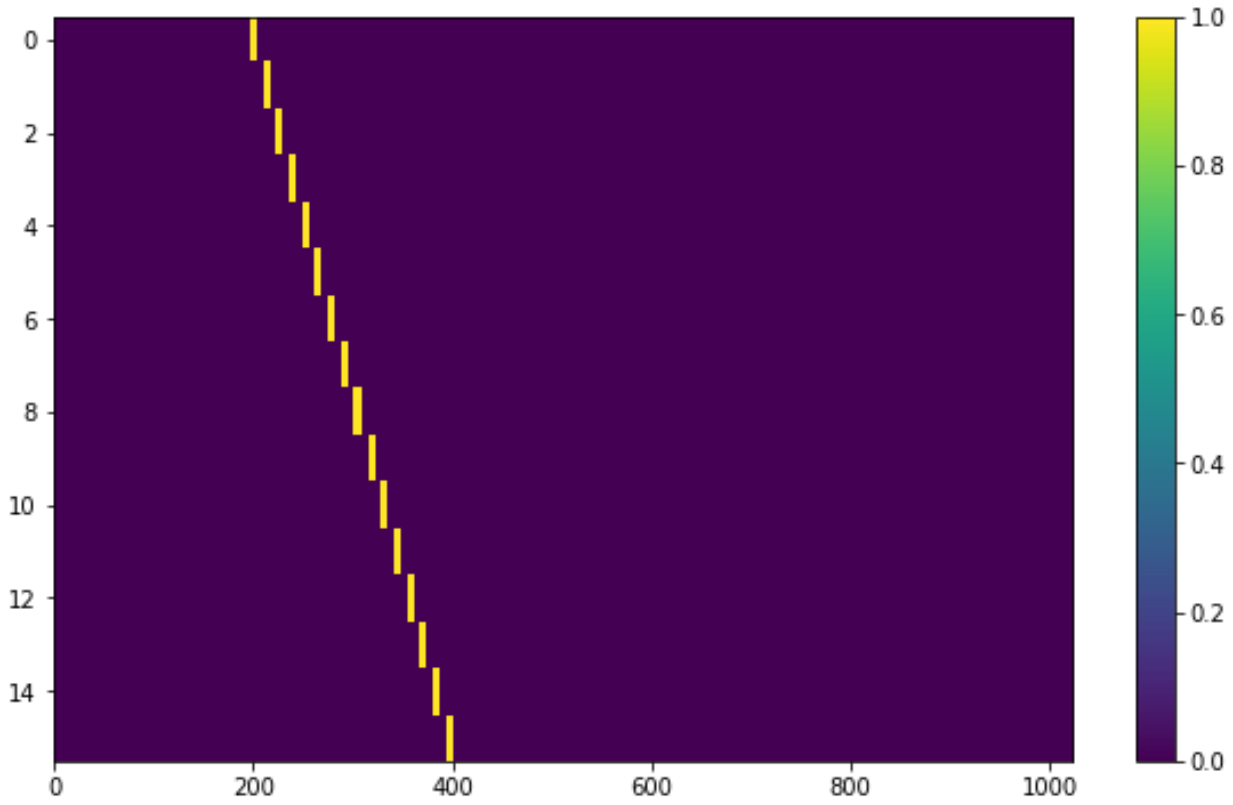
The main method that generates signals is *add_signal()*. This allows us to pass in an functions or arrays that describe the shape of the signal over time, over frequency within individual time samples, and over a bandpass of frequencies. setigen comes prepackaged with common functions (setigen.funcs), but you can write your own!

The most basic signal that you can generate is a constant intensity, constant drift-rate signal. Note that as in the Getting started example, you can also use frame.add_constant_signal, which is simpler and more efficient for signal injection into large data frames.

```
from astropy import units as u
import numpy as np

import setigen as stg

# Define time and frequency arrays, essentially labels for the 2D data array
fchans = 1024
tchans = 16
df = 2.7939677238464355*u.Hz
dt = 18.253611008*u.s
fch1 = 6095.214842353016*u.MHz

# Generate the signal
frame = stg.Frame(fchans=fchans,
                  tchans=tchans,
                  df=df,
                  dt=dt,
                  fch1=fch1)
signal = frame.add_signal(stg.constant_path(f_start=frame.get_frequency(200),
                                            drift_rate=2*u.Hz/u.s),
                          stg.constant_t_profile(level=1),
                          stg.box_f_profile(width=20*u.Hz),
                          stg.constant_bp_profile(level=1))
```

setigen.Frame.add_signal() returns a 2D numpy array containing only the synthetic signal. To visualize the resulting frame, we can use matplotlib.pyplot.imshow():

```
import matplotlib.pyplot as plt
fig = plt.figure(figsize=(10, 6))
plt.imshow(frame.get_data(), aspect='auto')
plt.colorbar()
fig.savefig("basic_signal.png", bbox_inches='tight')
```



In `setigen`, we use `astropy.units` to exactly specify where signals should be in time-frequency space. Astropy automatically handles unit conversions (MHz -> Hz, etc.), which is a nice convenience. Nevertheless, you can also use normal SI units (Hz, s) without additional modifiers, in which case the above code would become:

```
from astropy import units as u
import numpy as np

import setigen as stg

# Define time and frequency arrays, essentially labels for the 2D data array
fchans = 1024
tchans = 16
df = 2.7939677238464355
dt = 18.253611008
fch1 = 6095.214842353016 * 10**6

# Generate the signal
frame = stg.Frame(fchans=fchans,
                  tchans=tchans,
                  df=df,
                  dt=dt,
                  fch1=fch1)
signal = frame.add_signal(stg.constant_path(f_start=frame.get_frequency(200),
                                            drift_rate=2),
                          stg.constant_t_profile(level=1),
                          stg.box_f_profile(width=20),
                          stg.constant_bp_profile(level=1))
```

So, it isn't quite necessary to use `astropy.units`, but it's an option to avoid manual unit conversion and calculation.

1.3.2 Using prepackaged signal functions

With `setigen`'s pre-written signal functions, you can generate a variety of signals right off the bat. The main signal parameters that customize the synthetic signal are `path`, `t_profile`, `f_profile`, and `bp_profile`.

`path` describes the path of the signal in time-frequency space. The `path` function takes in a time and outputs 'central' frequency corresponding to that time.

`t_profile` (time profile) describes the intensity of the signal over time. The `t_profile` function takes in a time and outputs an intensity.

`f_profile` (frequency profile) describes the intensity of the signal within a time sample as a function of relative frequency. The `f_profile` function takes in a frequency and a central frequency and computes an intensity. This function is used to control the spectral shape of the signal (with respect to a central frequency), which may be a square wave, a Gaussian, or any custom shape!

`bp_profile` describes the intensity of the signal over the bandpass of frequencies. Whereas `f_profile` computes intensity with respect to a relative frequency, `bp_profile` computes intensity with respect to the absolute frequency value. The `bp_profile` function takes in a frequency and outputs an intensity as well.

All these functions combine to form the final synthetic signal, which means you can create a host of signals by switching up these parameters!

Here are just a few examples of pre-written signal functions. To see all of the included functions, check out setigen.funcs. To avoid needless repetition, each example script will assume the same basic setup:

```
from astropy import units as u
import numpy as np

import setigen as stg

# Define time and frequency arrays, essentially labels for the 2D data array
fchans = 1024
tchans = 16
df = 2.7939677238464355*u.Hz
dt = 18.253611008*u.s
fch1 = 6095.214842353016*u.MHz

# Generate the signal
frame = stg.Frame(fchans=fchans,
                  tchans=tchans,
                  df=df,
                  dt=dt,
                  fch1=fch1)
```

`paths` - trajectories in time-frequency space

Constant path

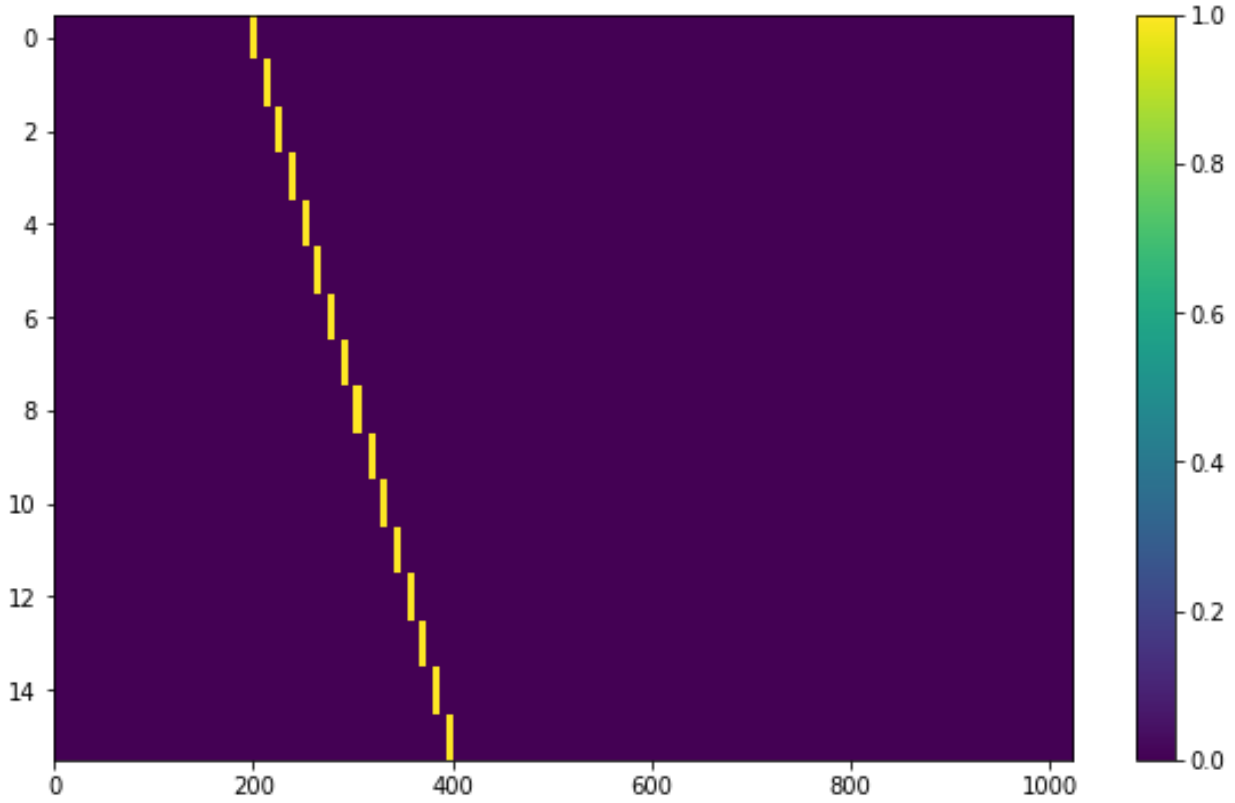
A constant path is a linear Doppler-drifted signal. To generate this path, use *`constant_path()`* and specify the starting frequency of the signal and the drift rate (in units of frequency over time, consistent with the units of your time and frequency arrays):

```
signal = frame.add_signal(stg.constant_path(f_start=frame.get_frequency(200),
                                            drift_rate=2*u.Hz/u.s),
                          stg.constant_t_profile(level=1),
```

(continues on next page)

(continued from previous page)

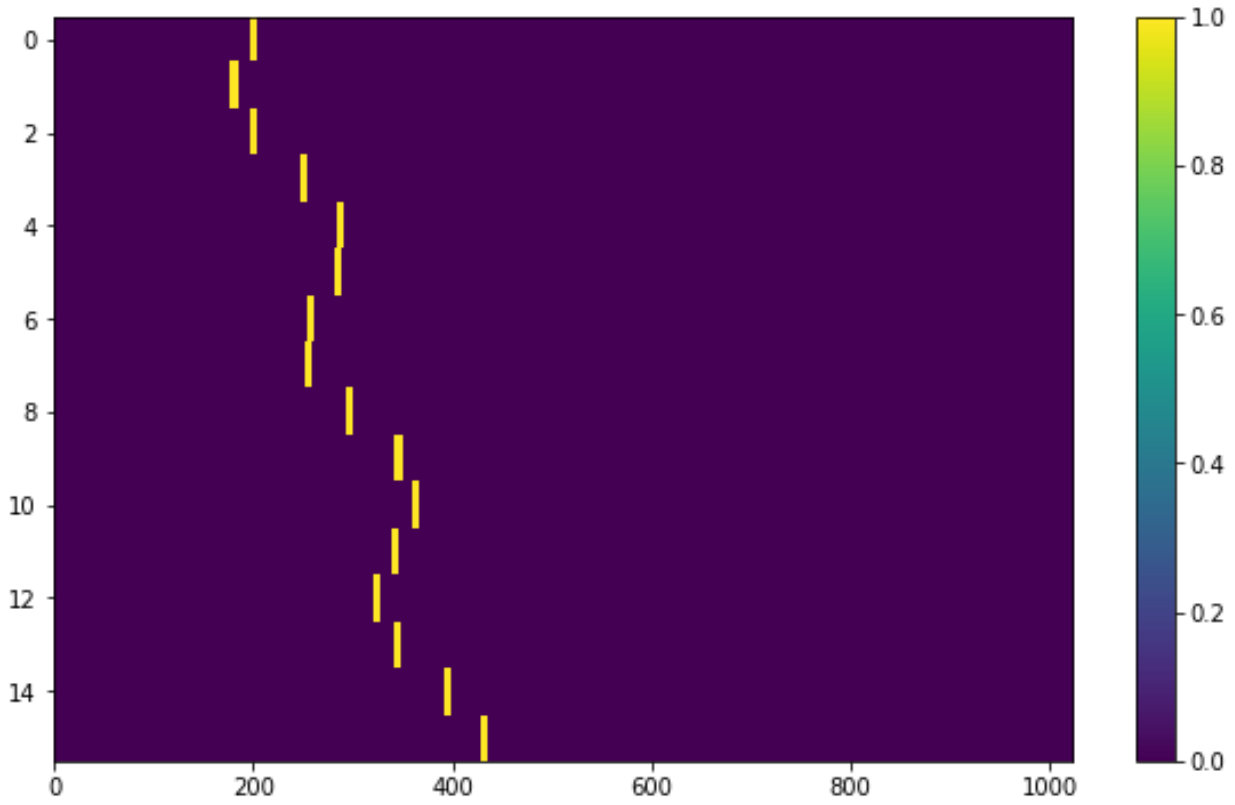
```
                                stg.box_f_profile(width=20*u.Hz),
                                stg.constant_bp_profile(level=1))
```



Sine path

This path is a sine wave, controlled by a starting frequency, drift rate, period, and amplitude, using *sine_path()*.

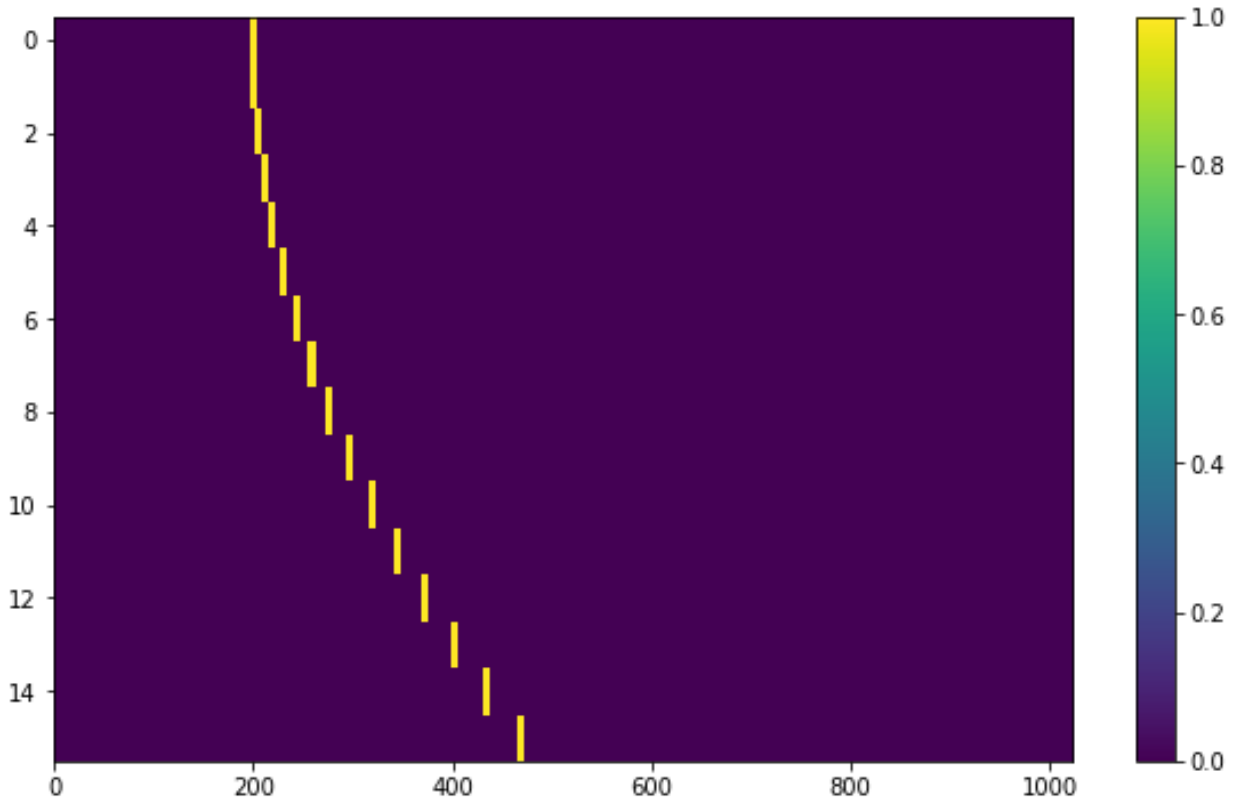
```
signal = frame.add_signal(stg.sine_path(f_start=frame.get_frequency(200),
                                        drift_rate=2*u.Hz/u.s,
                                        period=100*u.s,
                                        amplitude=100*u.Hz),
                          stg.constant_t_profile(level=1),
                          stg.box_f_profile(width=20*u.Hz),
                          stg.constant_bp_profile(level=1))
```



Squared path

This path is a very simple quadratic with respect to time, using *squared_path()*.

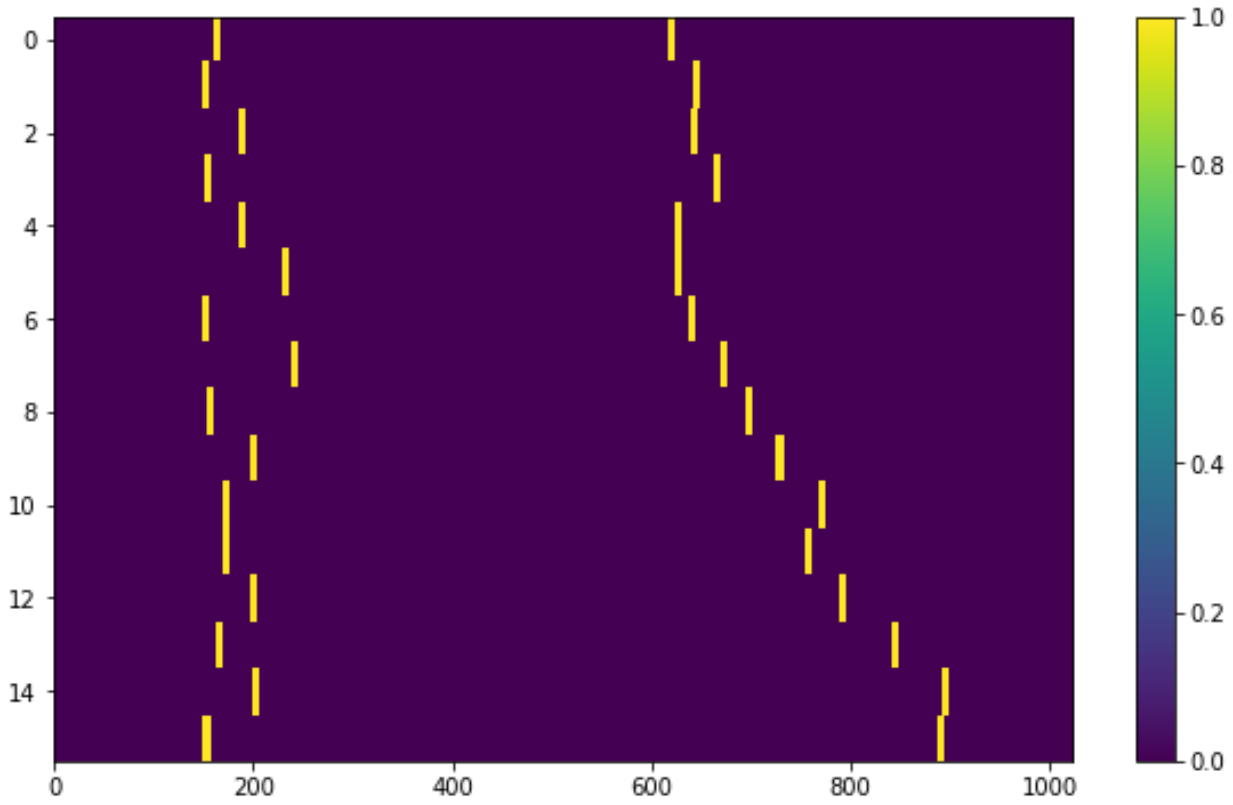
```
signal = frame.add_signal(stg.squared_path(f_start=frame.get_frequency(200),
                                           drift_rate=0.01*u.Hz/u.s),
                          stg.constant_t_profile(level=1),
                          stg.box_f_profile(width=20*u.Hz),
                          stg.constant_bp_profile(level=1))
```



RFI-like path

This path randomly varies in frequency, as in some RFI signals, using *simple_rfi_path()*. The following example shows two such signals, with `rfi_type` set to 'stationary' and 'random_walk'. You can define `drift_rate` to set these signals in relation to a straight line path.

```
frame.add_signal(stg.simple_rfi_path(f_start=frame.fs[200],
                                     drift_rate=0*u.Hz/u.s,
                                     spread=300*u.Hz,
                                     spread_type='uniform',
                                     rfi_type='stationary'),
                 stg.constant_t_profile(level=1),
                 stg.box_f_profile(width=20*u.Hz),
                 stg.constant_bp_profile(level=1))

frame.add_signal(stg.simple_rfi_path(f_start=frame.fs[600],
                                     drift_rate=0*u.Hz/u.s,
                                     spread=300*u.Hz,
                                     spread_type='uniform',
                                     rfi_type='random_walk'),
                 stg.constant_t_profile(level=1),
                 stg.box_f_profile(width=20*u.Hz),
                 stg.constant_bp_profile(level=1))
```

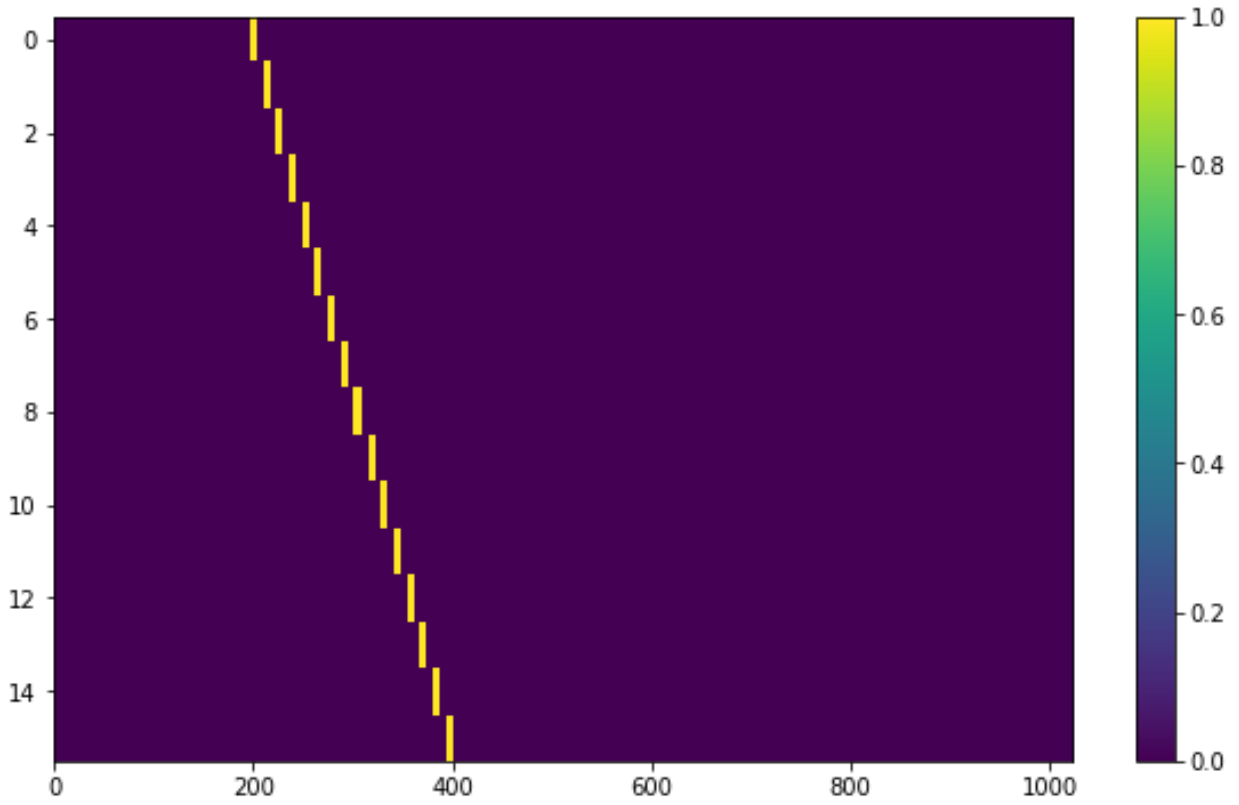


`t_profiles` - intensity variation with time

Constant intensity

To generate a signal with the same intensity over time, use *constant_t_profile()*, specifying only the intensity level:

```
signal = frame.add_signal(stg.constant_path(f_start=frame.get_frequency(200),
                                            drift_rate=2*u.Hz/u.s),
                          stg.constant_t_profile(level=1),
                          stg.box_f_profile(width=20*u.Hz),
                          stg.constant_bp_profile(level=1))
```

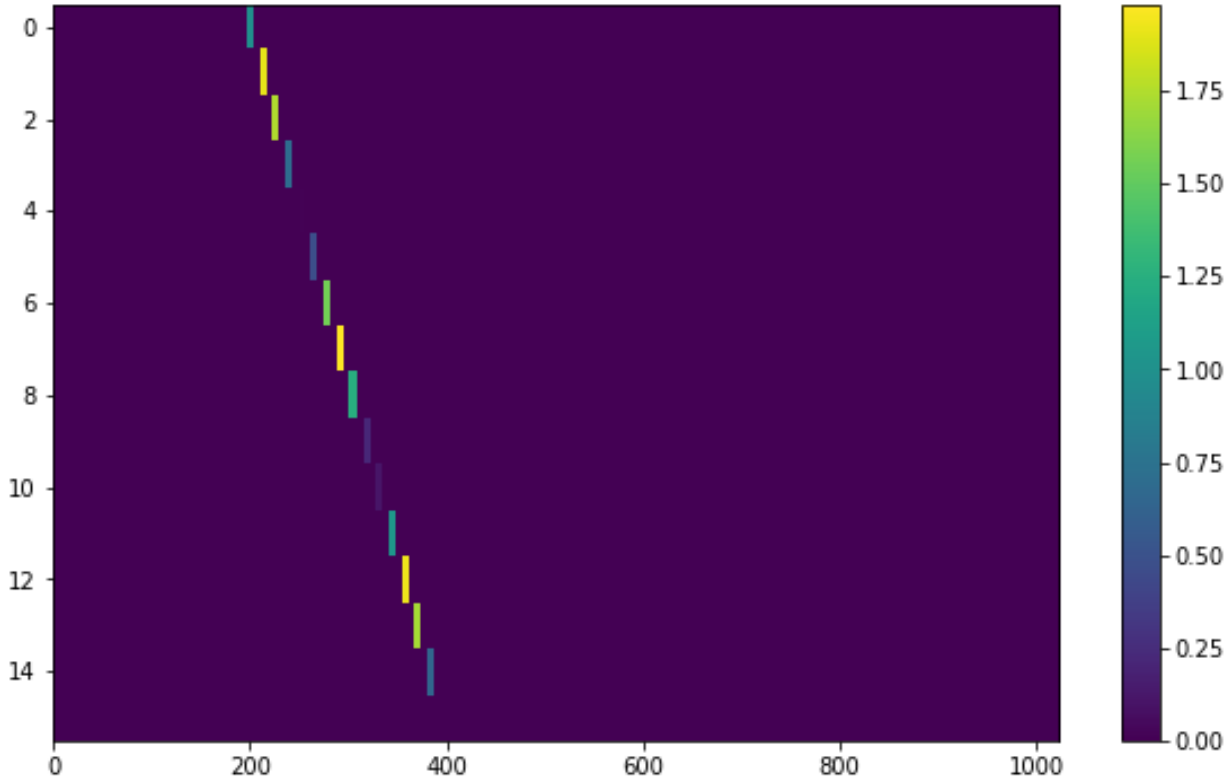


Sine intensity

To generate a signal with sinusoidal intensity over time, use *sine_t_profile()*, specifying the period, amplitude, and average intensity level. The intensity level is essentially an offset added to a sine function, so it should be equal or greater than the amplitude so that the signal doesn't have any negative values.

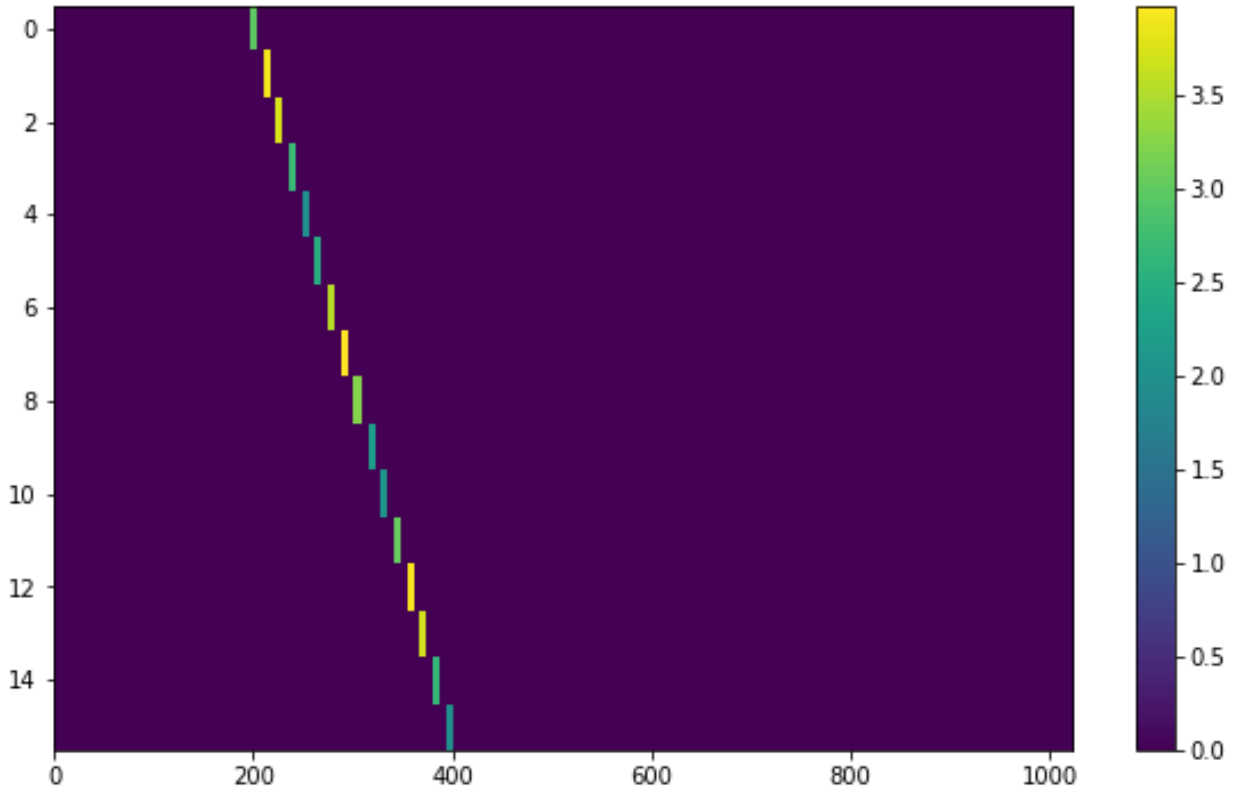
Here's an example with equal level and amplitude:

```
signal = frame.add_signal(stg.constant_path(f_start=frame.get_frequency(200),
                                            drift_rate=2*u.Hz/u.s),
                          stg.sine_t_profile(period=100*u.s,
                                             amplitude=1,
                                             level=1),
                          stg.box_f_profile(width=20*u.Hz),
                          stg.constant_bp_profile(level=1))
```



And here's an example with the level a bit higher than the amplitude:

```
signal = frame.add_signal(stg.constant_path(f_start=frame.get_frequency(200),
                                            drift_rate=2*u.Hz/u.s),
                          stg.sine_t_profile(period=100*u.s,
                                             amplitude=1,
                                             level=3),
                          stg.box_f_profile(width=20*u.Hz),
                          stg.constant_bp_profile(level=1))
```

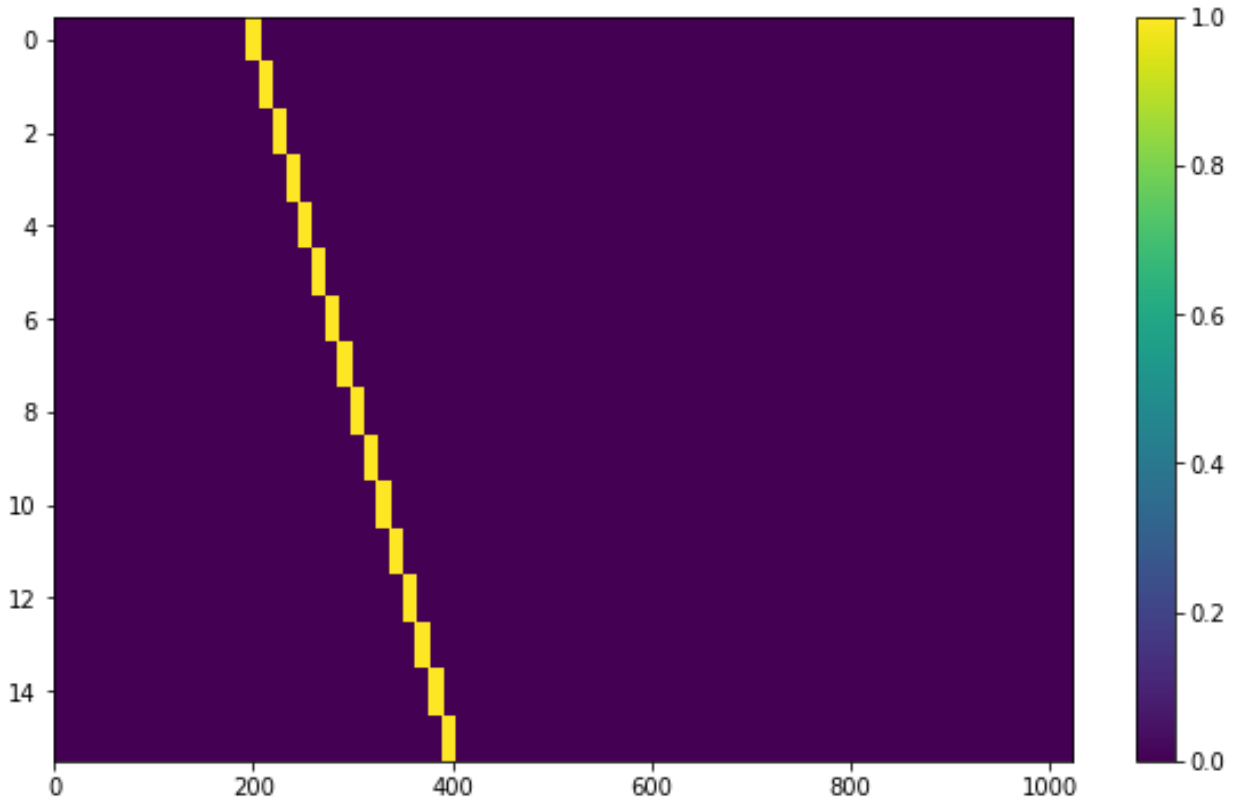


`f_profiles` - intensity variation with time

Box / square intensity profile

To generate a signal with the same intensity over frequency, use *box_f_profile()*, specifying the width of the signal:

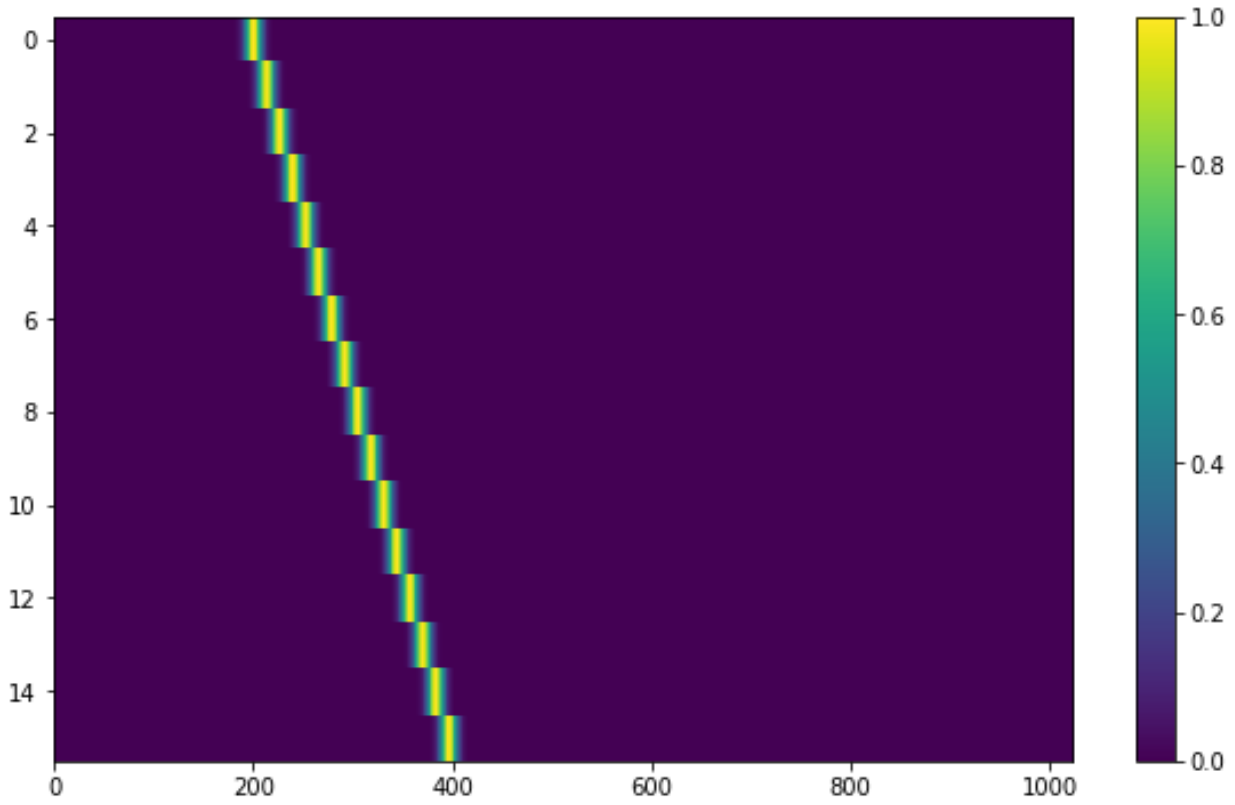
```
signal = frame.add_signal(stg.constant_path(f_start=frame.get_frequency(200),
                                            drift_rate=2*u.Hz/u.s),
                          stg.constant_t_profile(level=1),
                          stg.box_f_profile(width=40*u.Hz),
                          stg.constant_bp_profile(level=1))
```



Gaussian intensity profile

To generate a signal with a Gaussian intensity profile in the frequency direction, use *gaussian_f_profile()*, specifying the width of the signal:

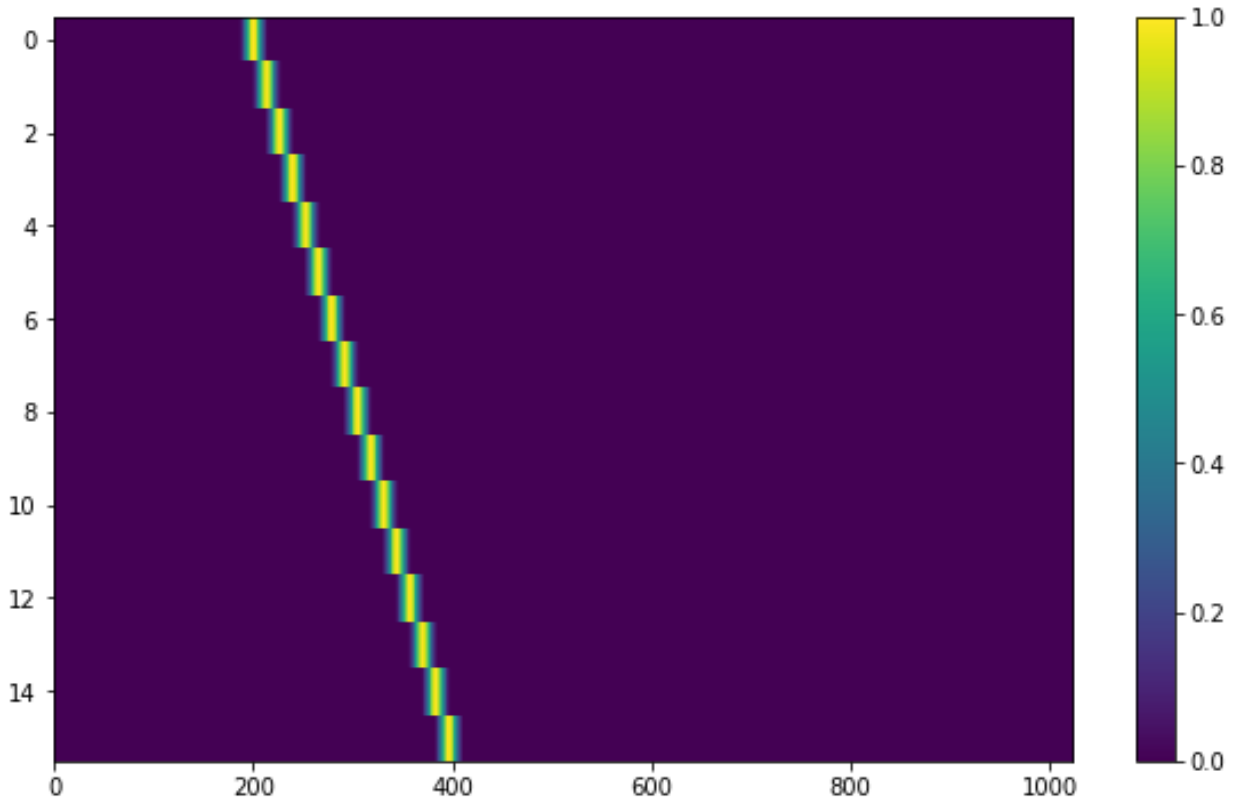
```
signal = frame.add_signal(stg.constant_path(f_start=frame.get_frequency(200),
                                            drift_rate=2*u.Hz/u.s),
                          stg.constant_t_profile(level=1),
                          stg.gaussian_f_profile(width=40*u.Hz),
                          stg.constant_bp_profile(level=1))
```



Sinc squared intensity profile

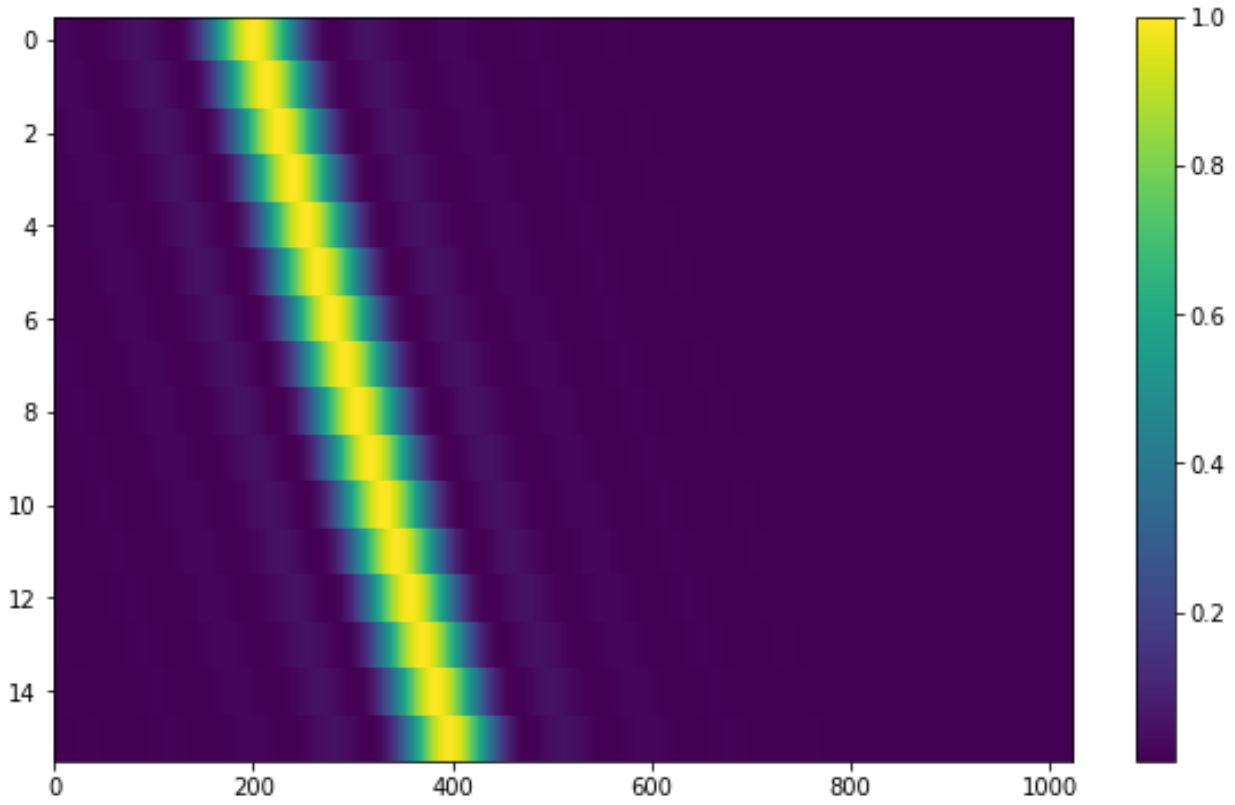
To generate a signal with a sinc squared intensity profile in the frequency direction, use *sinc2_f_profile()*, specifying the width of the signal:

```
signal = frame.add_signal(stg.constant_path(f_start=frame.get_frequency(200),
                                            drift_rate=2*u.Hz/u.s),
                          stg.constant_t_profile(level=1),
                          stg.sinc2_f_profile(width=40*u.Hz),
                          stg.constant_bp_profile(level=1))
```



By default, the function has the parameter `trunc=True` to truncate the sinc squared function at the first zero-crossing. With `trunc=False` and using a larger width to show the effect:

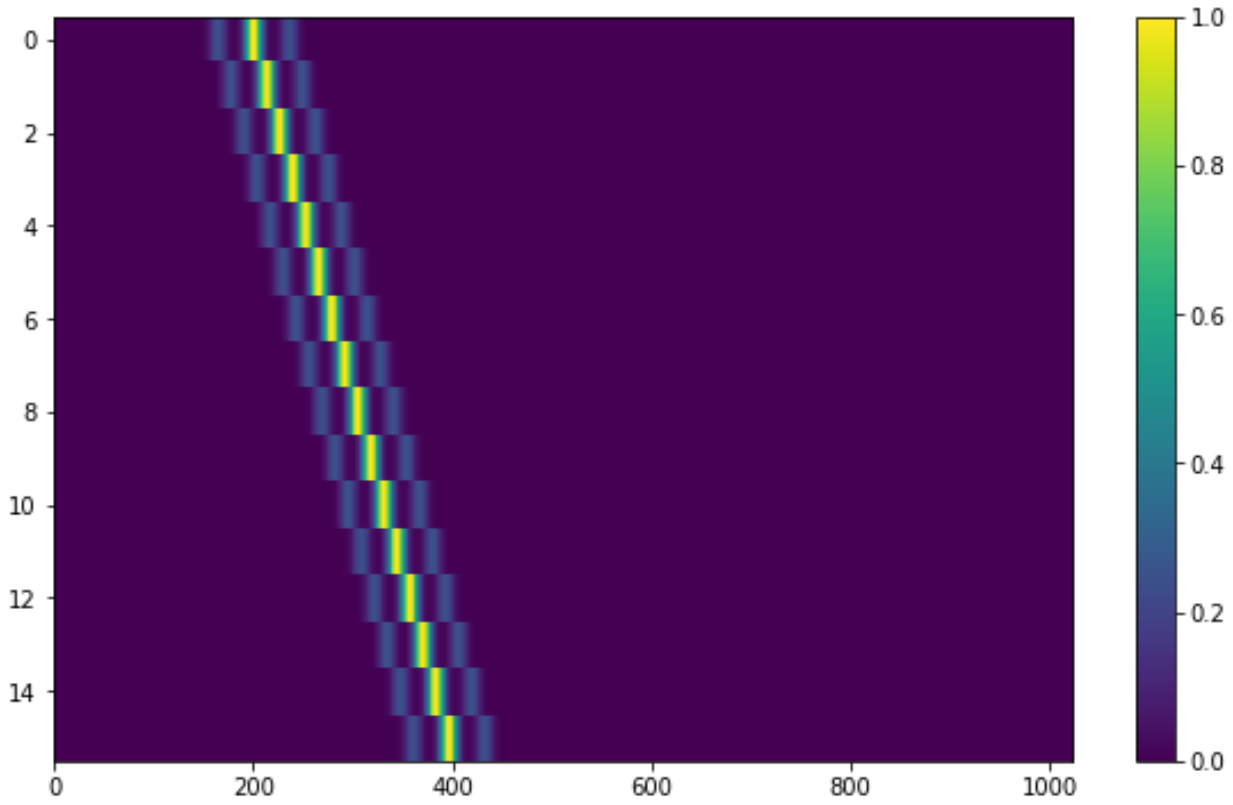
```
signal = frame.add_signal(stg.constant_path(f_start=frame.get_frequency(200),
                                            drift_rate=2*u.Hz/u.s),
                          stg.constant_t_profile(level=1),
                          stg.sinc2_f_profile(width=200*u.Hz, trunc=False),
                          stg.constant_bp_profile(level=1))
```



Multiple Gaussian intensity profile

The profile *multiple_gaussian_f_profile()*, generates a symmetric signal with three Gaussians; one main signal and two smaller signals on either side. This is mostly a demonstration that `f_profile` functions can be composite, and you can create custom functions like this (advanced.rst).

```
signal = frame.add_signal(stg.constant_path(f_start=frame.get_frequency(200),
                                            drift_rate=2*u.Hz/u.s),
                          stg.constant_t_profile(level=1),
                          stg.multiple_gaussian_f_profile(width=40*u.Hz),
                          stg.constant_bp_profile(level=1))
```



1.3.3 Adding synthetic noise

The background noise in high resolution BL data inherently follows a chi-squared distribution. Depending on the data's spectral and temporal resolutions, with enough integration blocks, the noise approaches a Gaussian distribution. `setigen` supports both distributions for noise generation, but uses chi-squared by default.

Every time synthetic noise is added to an image, `setigen` will estimate the noise properties of the frame, and you can get these via `get_total_stats()` and `get_noise_stats()`.

Important note: over a range of many frequency channels, real radio data has complex systematic structure, such as coarse channels and bandpass shapes. Adding synthetic noise according to a pure statistical distribution as the background for your frames is therefore most appropriate when your frame size is somewhat limited in frequency, in which case you can mostly ignore these systematic artifacts. As usual, whether this is something you should care about just depends on your use cases.

Adding pure chi-squared noise

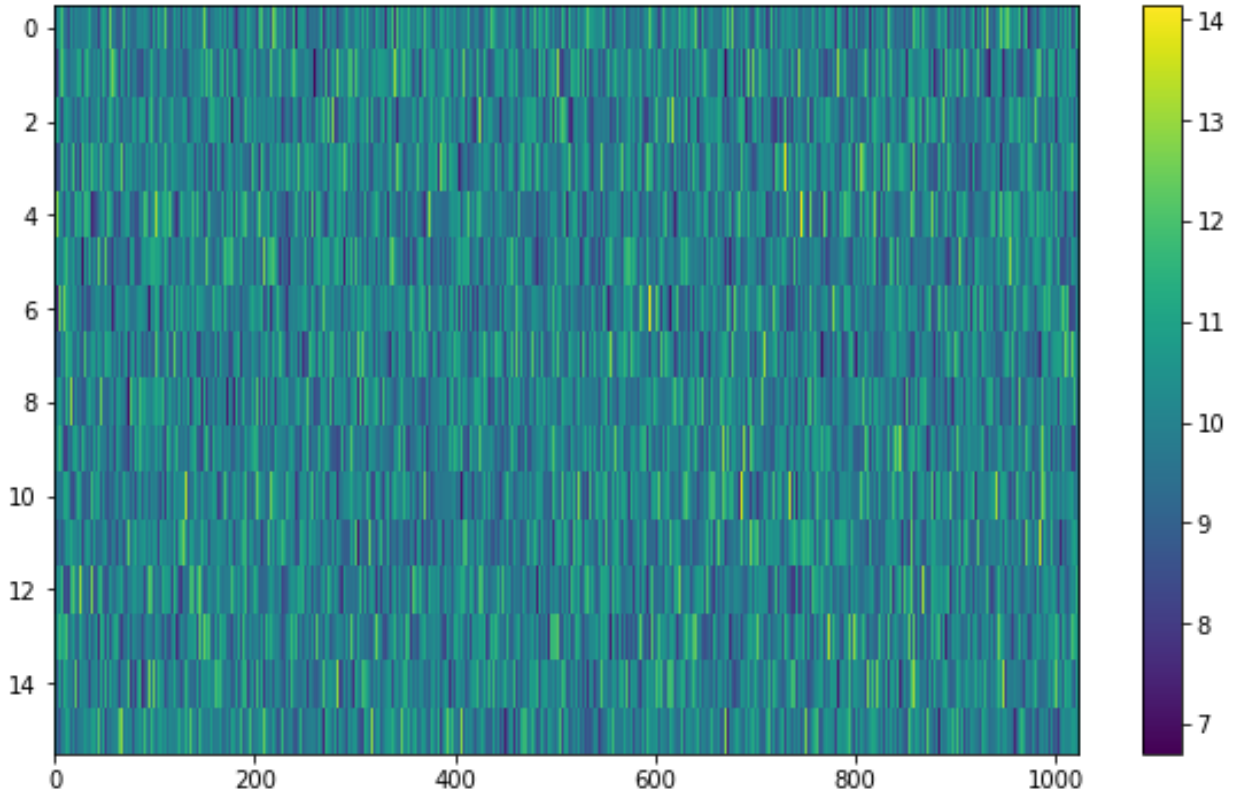
A minimal working example for adding noise is:

```
import matplotlib.pyplot as plt
import numpy as np
from astropy import units as u
import setigen as stg

# Define time and frequency arrays, essentially labels for the 2D data array
fchans = 1024
```

(continues on next page)

(continued from previous page)

```
tchans = 16
df = 2.7939677238464355*u.Hz
dt = 18.253611008*u.s
fch1 = 6095.214842353016*u.MHz

# Generate the signal
frame = stg.Frame(fchans=fchans,
                  tchans=tchans,
                  df=df,
                  dt=dt,
                  fch1=fch1)
noise = frame.add_noise(x_mean=10)

fig = plt.figure(figsize=(10, 6))
plt.imshow(frame.get_data(), aspect='auto')
plt.colorbar()
plt.show()
```

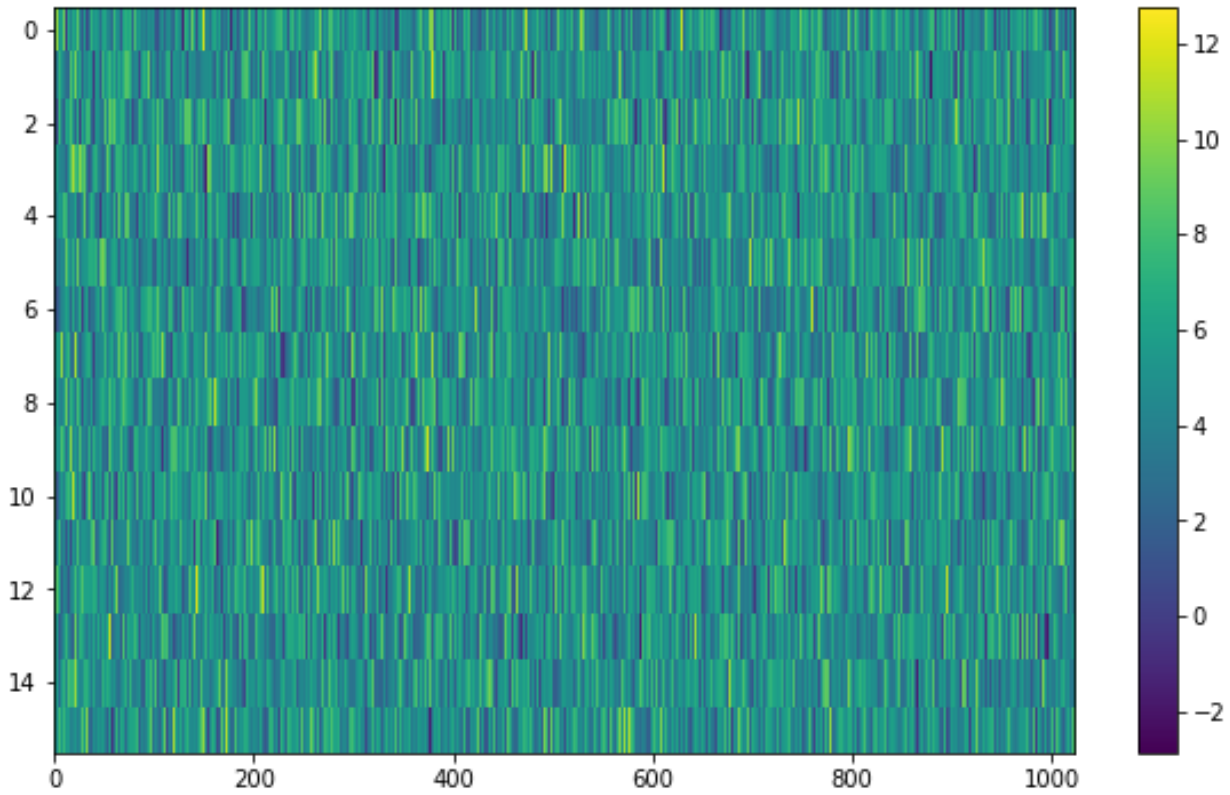


This adds chi-squared noise scaled to a mean of 10. *add_noise()* returns a 2D numpy array containing only the synthetic noise, and uses a default argument of `noise_type=chi2`. Behind the scenes, the degrees of freedom used in the chi-squared distribution are calculated using the frame resolution and can be accessed via the `frame.chi2_df` attribute.

Adding pure Gaussian noise

An example for adding Gaussian noise is:

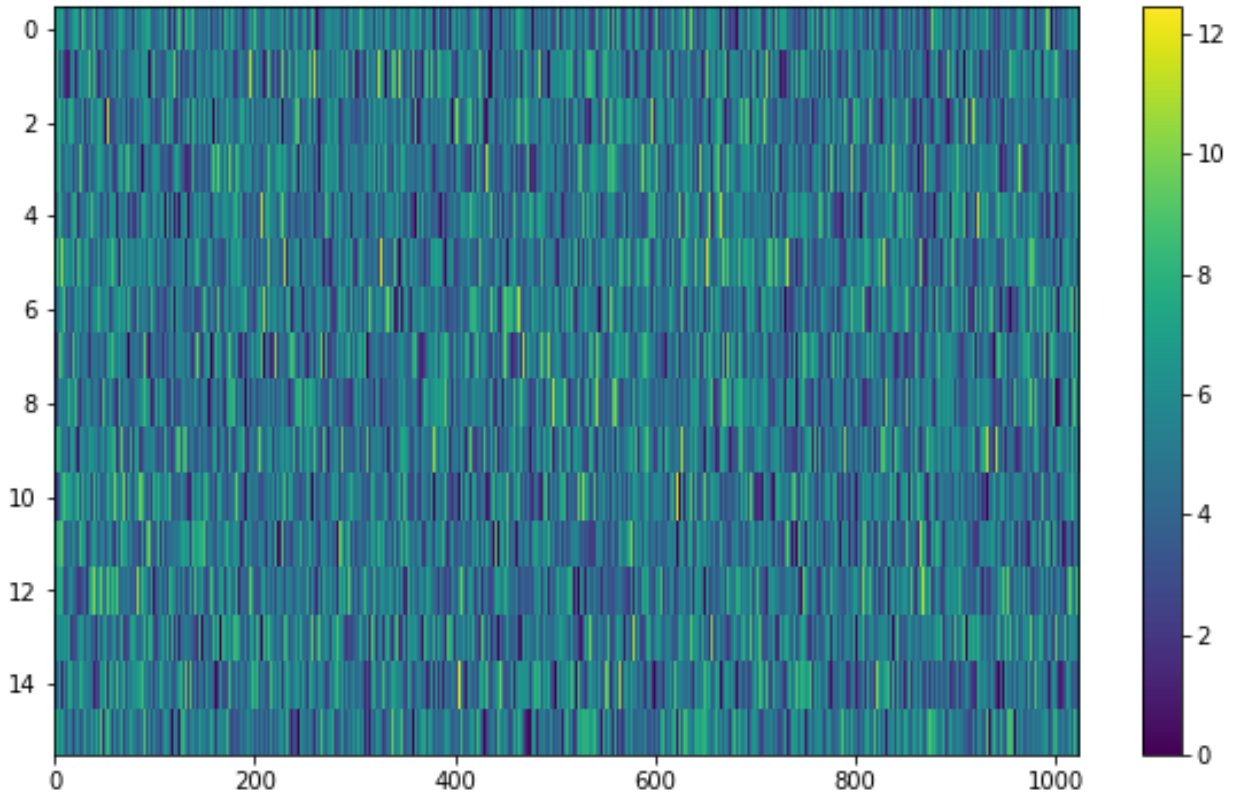
```
import matplotlib.pyplot as plt
import numpy as np
from astropy import units as u
import setigen as stg

# Define time and frequency arrays, essentially labels for the 2D data array
fchans = 1024
tchans = 16
df = 2.7939677238464355*u.Hz
dt = 18.253611008*u.s
fch1 = 6095.214842353016*u.MHz

# Generate the signal
frame = stg.Frame(fchans=fchans,
                  tchans=tchans,
                  df=df,
                  dt=dt,
                  fch1=fch1)
noise = frame.add_noise(x_mean=5, x_std=2, noise_type='gaussian')

fig = plt.figure(figsize=(10, 6))
plt.imshow(frame.get_data(), aspect='auto')
plt.colorbar()
plt.show()
```



This adds Gaussian noise with mean 5 and standard deviation 2 to an empty frame.

In addition, we can truncate the noise at a lower bound specified by parameter *x_min*:

```
noise = frame.add_noise(x_mean=5, x_std=2, x_min=0, noise_type='gaussian')
```



This may be useful depending on the use case; you might not want negative intensities, or simply any intensity below a reasonable threshold, to occur in your synthetic data.

Adding synthetic noise based on real observations

We can also generate synthetic noise whose parameters are sampled from real observations. Specifically, we can select the mean for chi-squared noise, or additionally the standard deviation and minimum for Gaussian noise, from distributions of parameters estimated from observations.

If no distributions are provided by the user, noise parameters are sampled by default from pre-loaded distributions in `setigen`. These were estimated from GBT C-Band observations on frames with (dt, df) = (1.4 s, 1.4 Hz) and (tchans, fchans) = (32, 1024). Behind the scenes, the mean, standard deviation, and minimum intensity over each sub-frame in the observation were saved into three respective numpy arrays.

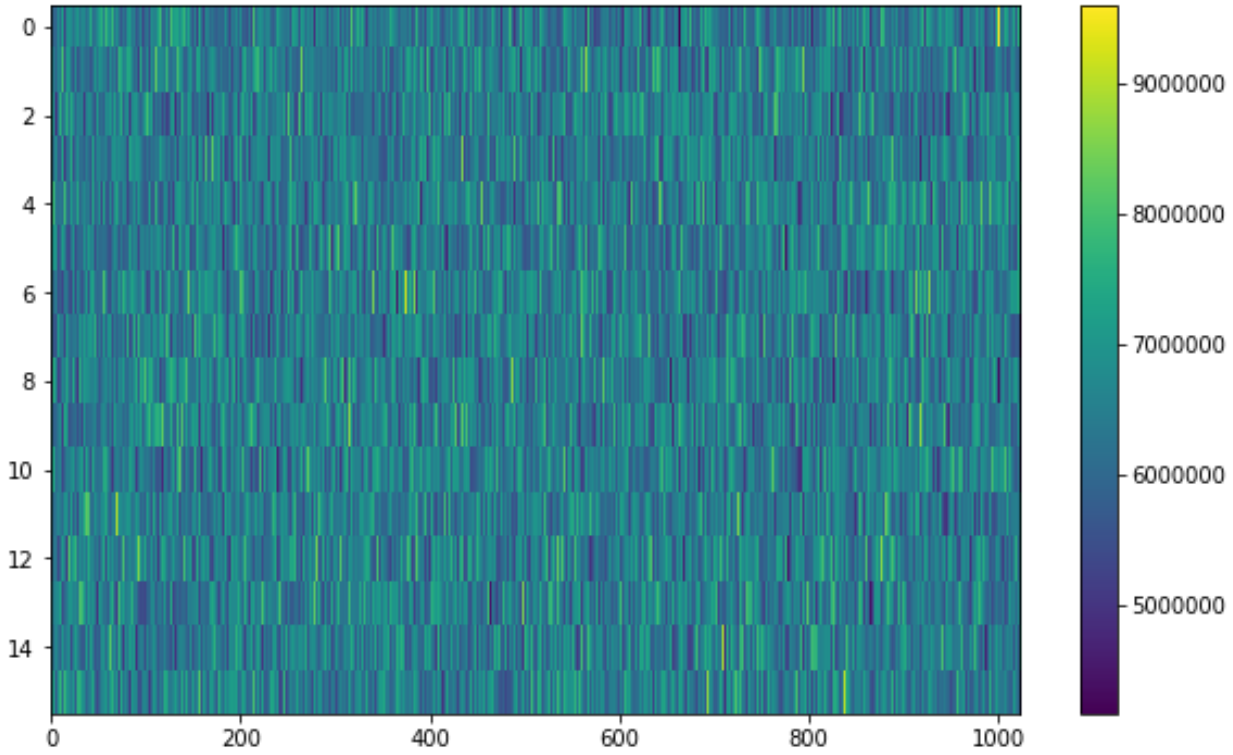
The `frame.add_noise_from_obs` function also uses chi-squared noise by default, selecting a mean intensity from the sampled observational distribution of means, and populating the frame with chi-squared noise accordingly.

Alternately, by setting `noise_type=gaussian` or `noise_type=normal` the function will select a mean, standard deviation, and minimum from these arrays (not necessarily all corresponding to the same original observational sub-frame), and populates your frame with Gaussian noise. You can also set the `share_index` parameter to True, to force these random noise parameter selections to all correspond to the same original observational sub-frame.

Note that these pre-loaded observations only serve as approximations and real observations vary depending on the noise temperature and frequency band. To be safe, you can generate your own parameters distributions from observational data.

For chi-squared noise:

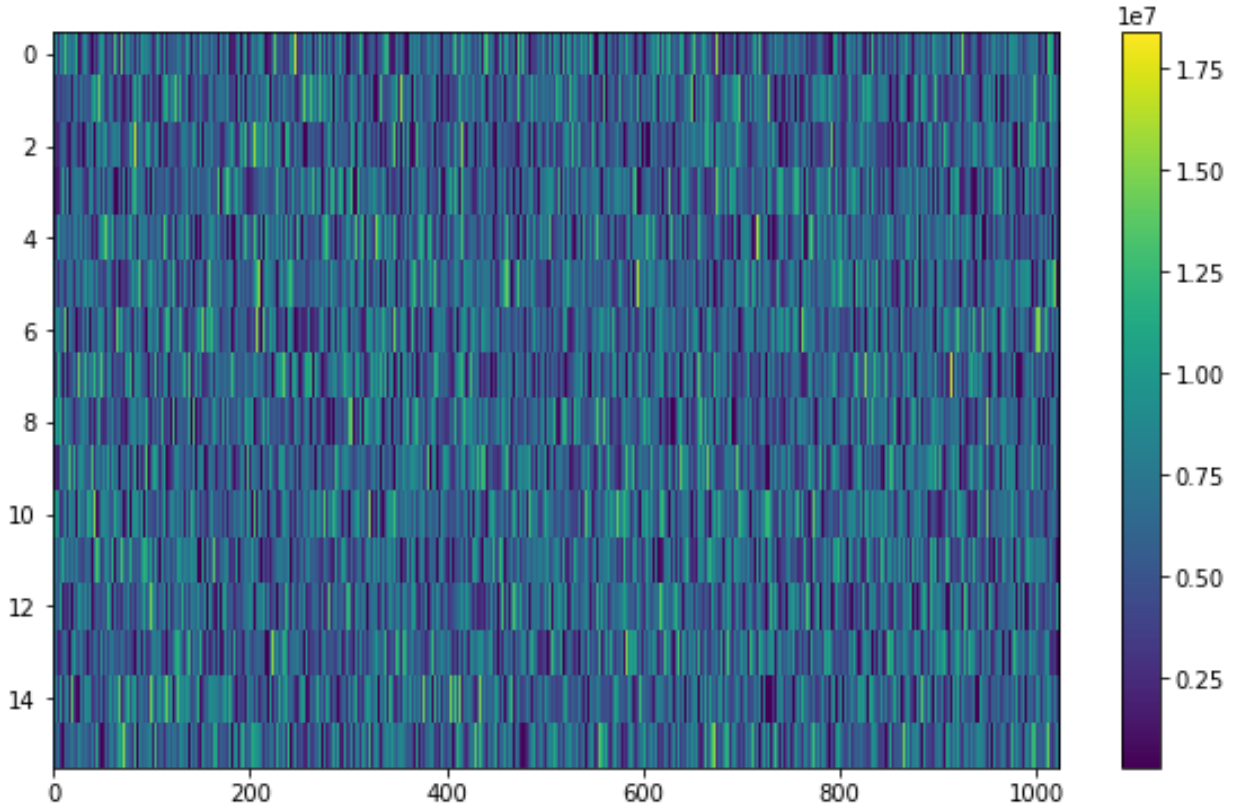
```
noise = frame.add_noise_from_obs()
```



We can readily see that the intensities are similar to a real GBT observation's.

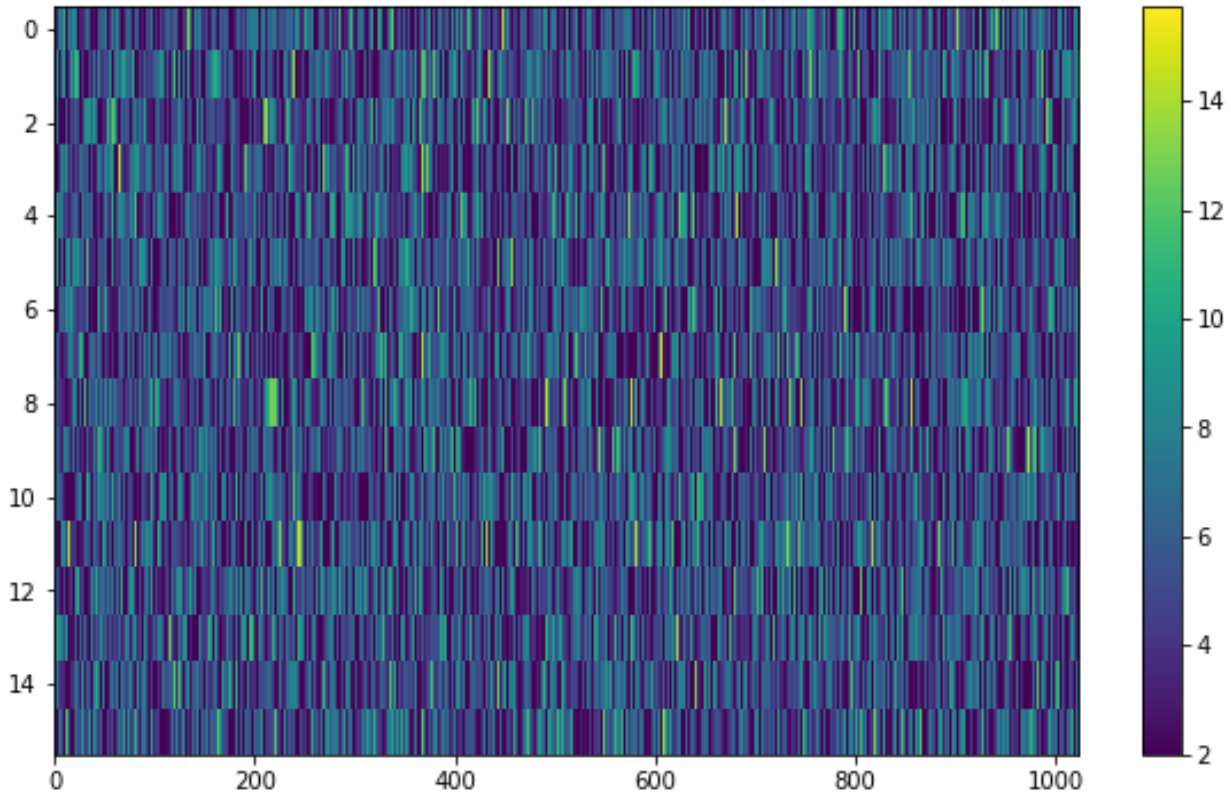
For Gaussian noise:

```
noise = frame.add_noise_from_obs(noise_type='gaussian')
```



We can also specify the distributions from which to sample parameters, one each for the mean, standard deviation, and minimum, as below. Note: just as in the pure noise generation above, you don't need to specify an x_min_array from which to sample if there's no need to truncate the noise at a lower bound.

```
noise = frame.add_noise_from_obs(x_mean_array=[3,4,5],
                                 x_std_array=[1,2,3],
                                 x_min_array=[1,2],
                                 share_index=False,
                                 noise_type='gaussian')
```



For chi-squared noise, only `x_mean_array` is used. For Gaussian noise, by default, random noise parameter selections are forced to use the same indices (as opposed to randomly choosing a parameter from each array) via `share_index=True`.

1.3.4 Convenience functions for signal generation

There are a few functions included in `Frame` that can help in constructing synthetic signals.

Getting frame data

To just grab the underlying intensity data, you can do

```
data = frame.get_data(use_db=False)
```

As it implies, if you switch the `use_db` flag to true, it will express the intensities in terms of decibels. This can help visualize data a little better, depending on the application.

Plotting frames

Examples of the built-in plotting utilities are on the Getting started page:

```
frame.render()
frame.bl_render()
```

Note that both of these methods use `matplotlib.pyplot.imshow` behind the scenes, which means you can still control plot parameters before and after these function calls, e.g.

```
fig = plt.figure(figsize=(10, 6))
frame.render()
plt.title('My awesome title')
plt.savefig('frame.png')
plt.show()
```

SNR <-> Intensity

If a frame has background noise, we can calculate intensities corresponding to different signal-to-noise (SNR) values. Here, the SNR of a signal is obtained from integrating over the entire time axis, e.g. so that it reduces noise by `sqrt(tchans)`.

For example, the included signal parameter functions in `setigen` all calculate signals based on absolute intensities, so if you'd like to include a signal with an SNR of 10, you would do:

```
intensity = frame.get_intensity(snr=10)
```

Alternately, you can get the SNR of a given intensity by doing:

```
snr = frame.get_snr(intensity=100)
```

Frequency <-> Index

Another useful conversion is between frequencies and frame indices:

```
index = frame.get_index(frequency)
frequency = frame.get_frequency(index)
```

Drift rate

For some injection tasks, you might want to define signals based on where they start and end on the frequency axis. Furthermore, this might not depend on frequency per se. In these cases, you can calculate a drift frequency using the `get_drift_rate` method:

```
start_index = np.random.randint(0, 1024)
end_index = np.random.randint(0, 1024)
drift_rate = frame.get_drift_rate(start_index, end_index)
```

Custom metadata

The Frame object includes a custom metadata property that allows you to manually track injected signal parameters. Accordingly, `frame.metadata` is a simple dictionary, making no assumptions about the type or number of signals you inject, or even what information to store. This property is mainly included as an easy way to save the data with the information you care about if you save and load frames with pickle.

```
new_metadata = {
    'snr': 10,
    'drift_rate': 2,
    'f_profile': 'lorentzian'
}
```

(continues on next page)

(continued from previous page)

```
# Sets custom metadata to an input dictionary
frame.set_metadata(new_metadata)

# Appends input dictionary to custom metadata
frame.add_metadata(new_metadata)

# Gets custom metadata dict
metadata = frame.get_metadata()
```

1.3.5 Saving and loading frames

There are a few different ways to save information from frames.

Using pickle

Pickle lets us save and load entire Frame objects, which is helpful for keeping both data and metadata together in storage:

```
# Saving to file
frame.save_pickle(filename='frame.pickle')

# Loading a Frame object from file
loaded_frame = stg.Frame.load_pickle(filename='frame.pickle')
```

Note that `load_pickle` is a class method, not an instance method.

Using numpy

If you would only like to save the frame data as a numpy array, you can do:

```
frame.save_npy(filename='frame.npy')
```

This just uses the `numpy.save` and `numpy.load` functions to save to `.npy`. If needed, you can also load in the data using

```
frame.load_npy(filename='frame.npy')
```

Using filterbank / HDF5

If you are interfacing with other Breakthrough Listen or astronomy codebases, outputting `setigen` frames in filterbank or HDF5 format can be very useful. Note that saving to HDF5 can have some difficulties based on your `bitshuffle` installation and other dependencies, but saving as a filterbank file is stable.

We provide the following methods:

```
frame.save_fil(filename='frame.fil')
frame.save_hdf5(filename='frame.hdf5')
frame.save_h5(filename='frame.h5')
```

To get an equivalent `blimpy` Waterfall object in the same Python session, use

```
waterfall = frame.get_waterfall()
```

1.4 Advanced topics

1.4.1 Advanced signal creation

Behind the scenes, *add_signal()* uses signal parameter functions to compute intensity for each time, frequency pair in the data. This is kept quite general to allow for the creation of complex signals. In this section, we explore some of the flexibility behind *add_signal()*.

Writing custom signal functions

You can go beyond setigen's pre-written signal functions by writing your own. For each *add_signal()* input parameter (path, t_profile, f_profile, and bp_profile), you can pass in your own custom functions. Note that these inputs are themselves functions.

It's important that the functions you pass into each parameter have the correct inputs and outputs. Specifically:

path Function that takes in time [array] t and outputs a frequency [array]

t_profile Function that takes in time [array] t and outputs an intensity [array]

f_profile Function that takes in frequency [array] f and a reference central frequency [array] f_center, and outputs an intensity [array]

bp_profile Function that takes in frequency [array] f and outputs an intensity [array]

For example, here's the code behind the sine path shape:

```
def my_sine_path(f_start, drift_rate, period, amplitude):
    def path(t):
        return f_start + amplitude * np.sin(2 * np.pi * t / period) + drift_rate * t
    return path
```

Alternately, you can use the lambda operator:

```
def my_sine_path(f_start, drift_rate, period, amplitude):
    return lambda t: return f_start + amplitude * np.sin(2 * np.pi * t / period) +␣
↪drift_rate * t
```

These can then be incorporated as:

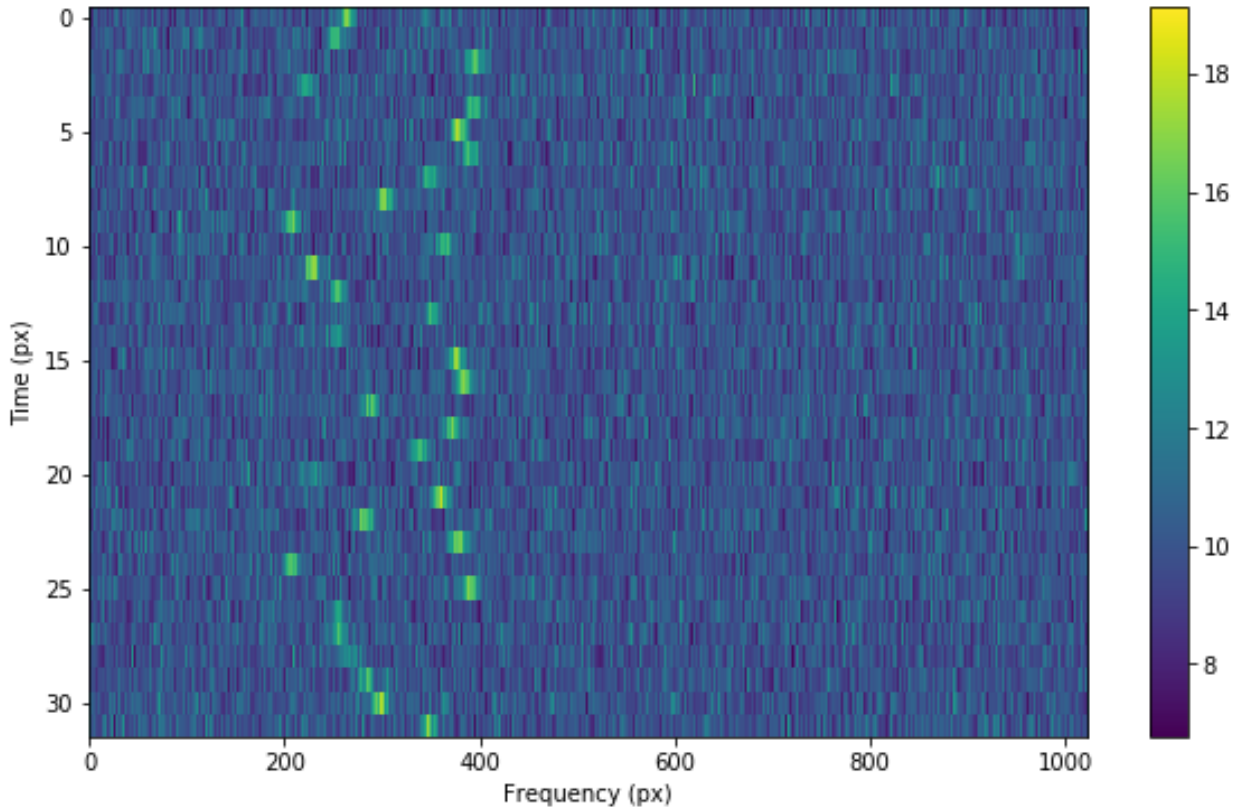
```
signal = frame.add_signal(my_sine_path(f_start=frame.get_frequency(200),
                                       drift_rate=2*u.Hz/u.s,
                                       period=100*u.s,
                                       amplitude=100*u.Hz),
                          stg.constant_t_profile(level=1),
                          stg.box_f_profile(width=20*u.Hz),
                          stg.constant_bp_profile(level=1))
```

To see more examples on how you might write your own parameter functions, check out the source code behind the pre-written functions (setigen.funcs).

Using arrays as signal parameters

Sometimes it can be difficult to wrap up a desired signal property into a separate function, or perhaps there is external existing code that calculates desired properties. In these cases, we can also use arrays to describe these signals, instead of functions.

```
from astropy import units as u
import numpy as np
import setigen as stg
import matplotlib.pyplot as plt

frame = stg.Frame(fchans=1024*u.pixel,
                  tchans=32*u.pixel,
                  df=2.7939677238464355*u.Hz,
                  dt=18.253611008*u.s,
                  fch1=6095.214842353016*u.MHz)
frame.add_noise(x_mean=10)

path_array = np.random.uniform(frame.get_frequency(200),
                               frame.get_frequency(400),
                               32)
t_profile_array = np.random.uniform(frame.get_intensity(snr=20),
                                    frame.get_intensity(snr=40),
                                    32)

frame.add_signal(path_array,
                 t_profile_array,
                 stg.gaussian_f_profile(width=40*u.Hz),
                 stg.constant_bp_profile(level=1))

fig = plt.figure(figsize=(10, 6))
frame.render()
plt.savefig('frame.png', bbox_inches='tight')
plt.show()
```



Optimization and accuracy

By default, `add_signal` calculates an intensity value for every time, frequency pairing. Depending on the situation, this might not be the best behavior.

For example, if you are injecting synthetic narrowband signals into a very large frame of data, it can be inefficient and unnecessary to calculate intensity values for every pixel in the frame. In other cases, perhaps calculating intensity values at only every (dt, df) offset would be too inaccurate.

Optimization

To limit the range of signal computation, you can use the `bounding_f_range` parameter of `add_signal`. This takes in a tuple of frequencies `(bounding_min, bounding_max)`, between which the signal will be computed.

```
signal = frame.add_signal(stg.constant_path(f_start=frame.get_frequency(200),
                                            drift_rate=2*u.Hz/u.s),
                          stg.constant_t_profile(level=1),
                          stg.box_f_profile(width=20*u.Hz),
                          stg.constant_bp_profile(level=1),
                          bounding_f_range=(frame.get_frequency(100),
                                            frame.get_frequency(700)))
```

As an example of how this can reduce needless computation, we can time different frame manipulations for a large frame:

```
import time

times = []
times.append(time.time())

frame = stg.Frame(fchans=2**20,
                  tchans=32,
                  df=2.7939677238464355*u.Hz,
                  dt=18.253611008*u.s,
                  fch1=6095.214842353016*u.MHz)
times.append(time.time())

frame.add_noise(x_mean=10)
times.append(time.time())

# Normal add_signal
frame.add_signal(stg.constant_path(f_start=frame.get_frequency(200),
                                   drift_rate=2*u.Hz/u.s),
                 stg.constant_t_profile(level=frame.get_intensity(snr=30)),
                 stg.gaussian_f_profile(width=40*u.Hz),
                 stg.constant_bp_profile(level=1))
times.append(time.time())

# Limiting computation with bounding_f_range
frame.add_signal(stg.constant_path(f_start=frame.get_frequency(200),
                                   drift_rate=2*u.Hz/u.s),
                 stg.constant_t_profile(level=frame.get_intensity(snr=30)),
                 stg.gaussian_f_profile(width=40*u.Hz),
                 stg.constant_bp_profile(level=1),
                 bounding_f_range=(frame.get_frequency(100),
                                   frame.get_frequency(700)))
times.append(time.time())

x = np.array(times)
print(x[1:] - x[:-1])

>>> [1.14681625 1.4038794  1.6308465  0.02862048]
```

Depending on the type of signal, you should be cautious when defining a bounding frequency range. For signals with constant drift rate and small spectral width, it isn't too hard to define a range. For example, `frame.add_constant_signal` uses bounding ranges automatically to optimize signal creation.

However, for signals with large or stochastic frequency variation, or with long spectral tails, it can be difficult to define a bounding range without cutting off parts of these signals.

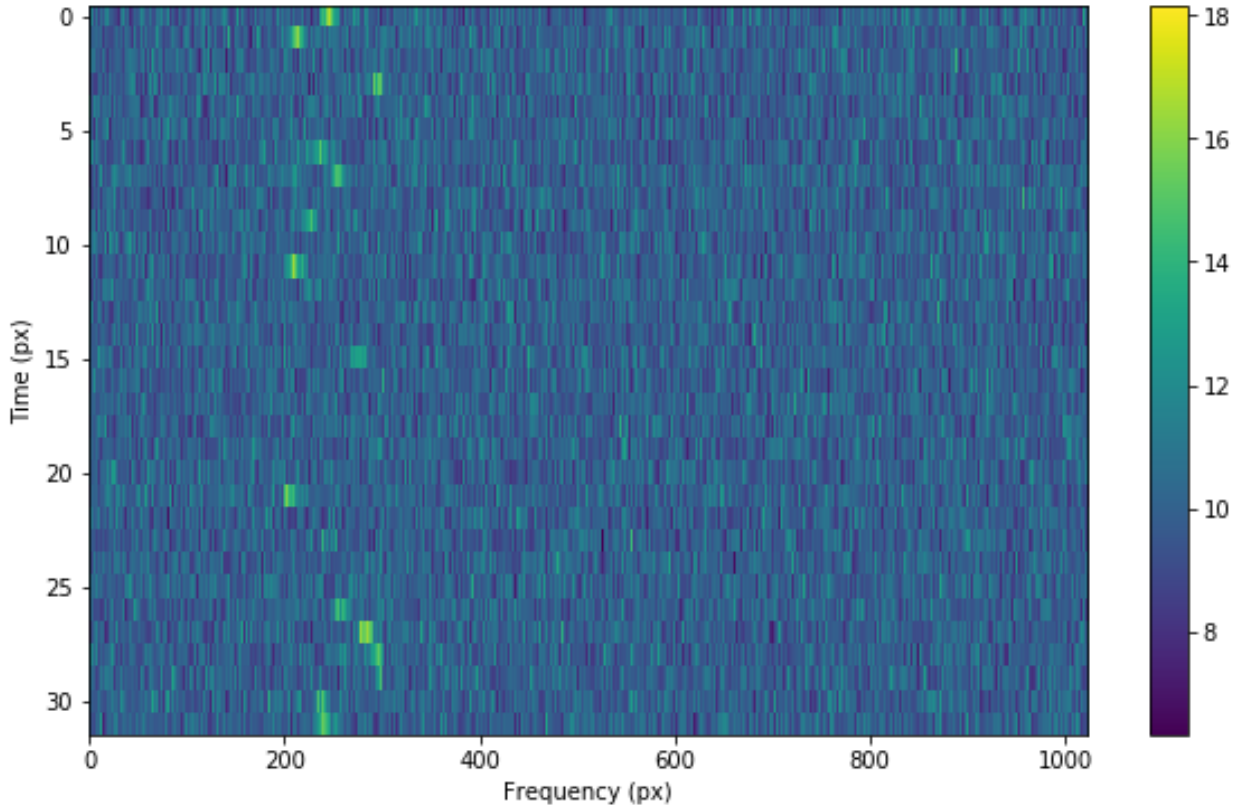
To illustrate this, using the above example that takes arrays as signal parameters, setting too small of a bounding frequency range can look like:

```
frame = stg.Frame(fchans=1024*u.pixel,
                  tchans=32*u.pixel,
                  df=2.7939677238464355*u.Hz,
                  dt=18.253611008*u.s,
                  fch1=6095.214842353016*u.MHz)
frame.add_noise(x_mean=10)

path_array = np.random.uniform(frame.get_frequency(200),
                               frame.get_frequency(400),
                               32)
```

(continues on next page)

(continued from previous page)

```
t_profile_array = np.random.uniform(frame.get_intensity(snr=20),
                                    frame.get_intensity(snr=40),
                                    32)

frame.add_signal(path_array,
                 t_profile_array,
                 stg.gaussian_f_profile(width=40*u.Hz),
                 stg.constant_bp_profile(level=1),
                 bounding_f_range=(frame.get_frequency(200),
                                   frame.get_frequency(300)))
```



Accuracy

To improve accuracy a bit, we can integrate signal computations over subsamples in time and frequency. The function `add_signal` has three boolean parameters: `integrate_path`, `integrate_t_profile`, and `integrate_f_profile`, which control whether various integrations are turned on (by default, they are False). The former two depend on the `t_subsamples` parameter, which is the number of bins per time sample (e.g. per dt) over which to integrate; likewise, `integrate_f_profile` depends on the `f_subsamples` parameter.

`integrate_path` controls integration of the signal's center frequency with respect to time, `path`. If your `path` varies on timescales shorter than the time resolution dt, then it could make sense to integrate to get more appropriate frequency positions.

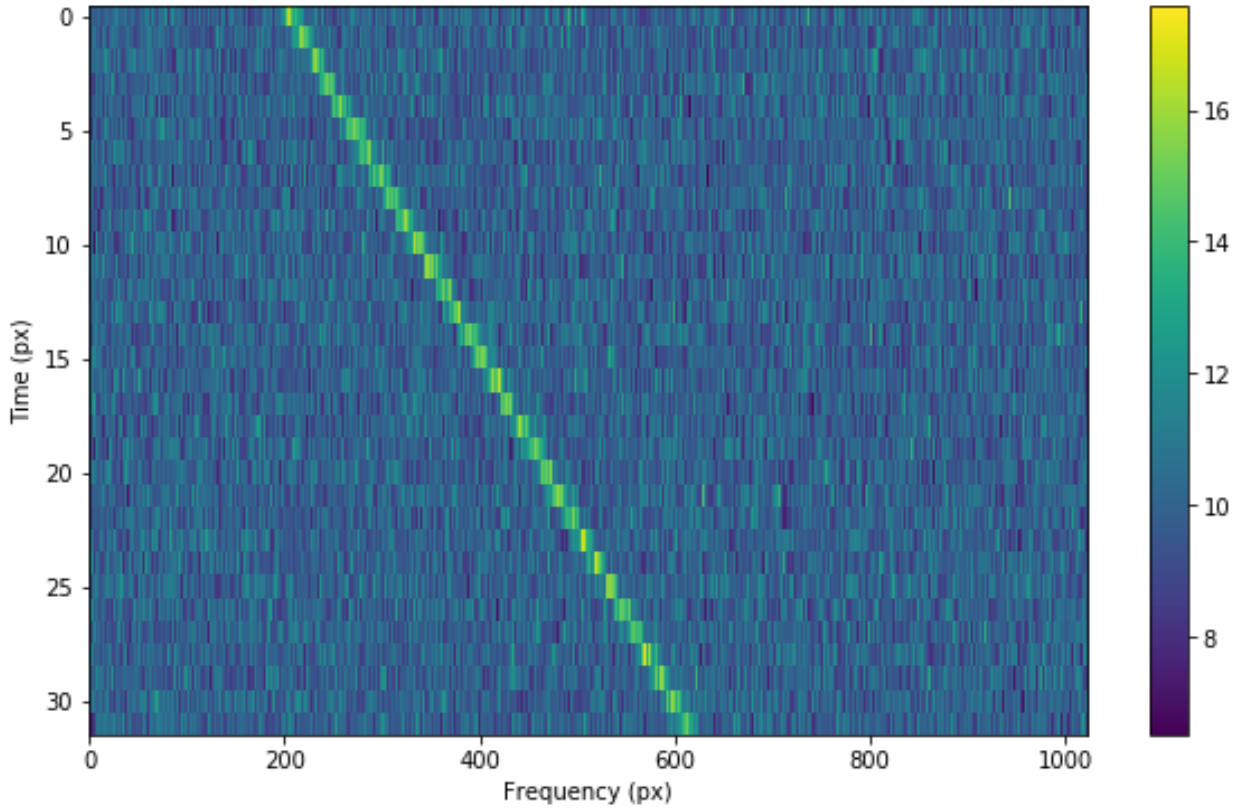
`integrate_t_profile` controls integration of the intensity variation with respect to time, `t_profile`. If your `t_profile` varies on timescales shorter than the time resolution dt, then it could make sense to integrate to get more appropriate intensities.

`integrate_f_profile` controls integration of the intensity variation with respect to frequency, `f_profile`. If your `f_profile` varies on spectral scales shorter than the frequency resolution df, then it could make sense to integrate to get more appropriate intensities.

Note that since integration requires make multiple calculations per pixel, it can increase signal computation time significantly. Be sure to evaluate whether it's actually necessary to integrate, or whether the default add_signal computation is sufficient for your use cases.

Here is an example of integration in action:

```
frame = stg.Frame(fchans=1024*u.pixel,
                  tchans=32*u.pixel,
                  df=2.7939677238464355*u.Hz,
                  dt=18.253611008*u.s,
                  fch1=6095.214842353016*u.MHz)
frame.add_noise(x_mean=10)

frame.add_signal(stg.constant_path(f_start=frame.get_frequency(200),
                                   drift_rate=2*u.Hz/u.s),
                 stg.constant_t_profile(level=frame.get_intensity(snr=30)),
                 stg.gaussian_f_profile(width=40*u.Hz),
                 stg.constant_bp_profile(level=1),
                 integrate_path=True,
                 integrate_t_profile=True,
                 integrate_f_profile=True,
                 t_subsamples=10,
                 f_subsamples=10)
```



1.4.2 Creating custom observational noise distributions

If you are interested in simulating observations of different resolutions and frequency bands, the underlying noise statistics may certainly differ from the included C-band distributions used by `frame.add_noise_from_obs`. In these cases, it may be best to generate your own parameter distribution arrays from your own observations, and feed those into `frame.add_noise_from_obs` yourself. It is worth mentioning that while you can just inject signals into observational frames directly, real observations may contain real signals as well. By estimating noise parameter distributions from observations, you can generate synthetic chi-squared or Gaussian noise with similar noise statistics as real observations, thereby resembling real data while excluding real signals.

To do this, we can use *`get_parameter_distributions()`*:

```
import setigen as stg
waterfall_fn = 'path/to/data.fil'
# Number of frequency channels per frame
fchans = 1024
# Number of time samples per frame; optional
tchans = 32
x_mean_array, x_std_array, x_min_array = stg.get_parameter_distributions(waterfall_fn,
                                                                         fchans,
                                                                         ␣
↪tchans=tchans,
                                                                         f_shift=None)
```

This will iterate over an entire filterbank file, estimating the noise statistics and returning them as numpy arrays.

1.4.3 Creating an injected synthetic signal dataset using observations

We can create a dataset based on observations using the *`split_utils`* module. We can use *`split_waterfall_generator()`* to create a Python generator that returns `blimpy` Waterfall objects, from which we can create `setigen` Frames. The function *`get_parameter_distributions()`* actually uses this behind the scenes to iterate through observational data.

```
import setigen as stg
waterfall_fn = 'path/to/data.fil'
fchans = 1024
tchans = 32
waterfall_itr = stg.split_waterfall_generator(waterfall_fn,
                                              fchans,
                                              tchans=tchans,
                                              f_shift=None)
waterfall = next(waterfall_itr)
frame = stg.Frame(waterfall)
```

Here, `f_shift` is the number of indices in the frequency direction to shift before making another slice or split of size `fchans`. If `f_shift=None`, it defaults to shifting over by `fchans`, so that there is no overlap.

To construct a full dataset, we can then use the generator to iterate over slices of data and save out frames. As a simple example:

```
for i, waterfall in enumerate(waterfall_itr):
    frame = stg.Frame(waterfall=waterfall)

    start_index = np.random.randint(0, fchans)
    end_index = np.random.randint(0, fchans)
    drift_rate = frame.get_drift_rate(start_index, end_index)
```

(continues on next page)

(continued from previous page)

```
    signal = frame.add_constant_signal(f_start=frame.get_frequency(start_index),
                                       drift_rate=drift_rate,
                                       level=frame.get_intensity(snr=10),
                                       width=40,
                                       f_profile_type='gaussian')
    signal_props = {
        'start_index': start_index,
        'end_index': end_index,
        'snr': 10,
    }
    frame.add_metadata(signal_props)
    frame.save_pickle('save/path/frame{:06d}.pickle'.format(i))
```

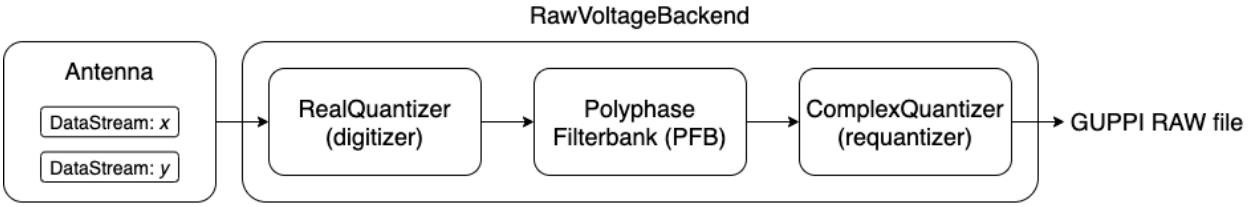
Depending on the application, it can also pay to save metadata out to a CSV file that tracks filenames / indices with corresponding properties.

1.5 Voltage synthesis (setigen.voltage)

The setigen.voltage module extends `setigen` to the voltage regime. Instead of directly synthesizing spectrogram data, we can produce real voltages, pass them through a software pipeline based on a polyphase filterbank, and record to file in GUPPI RAW format. As this process models actual hardware used by Breakthrough Listen for recording raw voltages, this enables lower level testing and experimentation.

A set of tutorial walkthroughs can be found here.

1.5.1 The basic pipeline structure



The basic layout of a voltage pipeline written using `setigen.voltage` is shown in the image.

First, we have an Antenna, which contains DataStreams for each polarization (1 or 2 total). Noise and signals are added to individual DataStreams, so that polarizations are unique and not necessarily correlated. These are added as functions, which accept an array of times in seconds and return an array of voltages, corresponding to random noise or defined signals. This allows us to obtain voltage samples on demand from each DataStream, and by extension from the Antenna.

The main backend elements are the digitizer, filterbank, and requantizer. The digitizer quantizes input voltages to a desired number of bits, and a desired full width at half maximum (FWHM) in the quantized voltage space. The filterbank implements a software polyphase filterbank, coarsely channelizing input voltages. The requantizer takes the resulting complex voltages, and quantizes each component to either 8 or 4 bits, suitable for saving into GUPPI RAW format.

All of these elements are wrapped into the RawVoltageBackend, which connects each piece together. The main method `RawVoltageBackend.record()` automatically retrieves real voltages as needed and passes them through each backend element, finally saving out the quantized complex voltages to disk.

A minimal working example of the pipeline is as follows:

```
from astropy import units as u
import setigen as stg

antenna = stg.voltage.Antenna(sample_rate=3e9*u.Hz,
                              fch1=6000e6*u.Hz,
                              ascending=True,
                              num_pols=1)

antenna.x.add_noise(v_mean=0,
                    v_std=1)

antenna.x.add_constant_signal(f_start=6002.2e6*u.Hz,
                              drift_rate=-2*u.Hz/u.s,
                              level=0.002)

digitizer = stg.voltage.RealQuantizer(target_fwhm=32,
                                      num_bits=8)

filterbank = stg.voltage.PolyphaseFilterbank(num_taps=8,
                                             num_branches=1024)

requantizer = stg.voltage.ComplexQuantizer(target_fwhm=32,
                                           num_bits=8)

rvb = stg.voltage.RawVoltageBackend(antenna,
                                    digitizer=digitizer,
                                    filterbank=filterbank,
                                    requantizer=requantizer,
                                    start_chan=0,
                                    num_chans=64,
                                    block_size=134217728,
                                    blocks_per_file=128,
                                    num_subblocks=32)

rvb.record(output_file_stem='example_1block',
           num_blocks=1,
           length_mode='num_blocks',
           header_dict={'HELLO': 'test_value',
                        'TELESCOP': 'GBT'},
           verbose=True)
```

1.5.2 Using GPU acceleration

The process of synthesizing real voltages at a high sample rate and passing through multiple signal processing steps can be very computationally expensive on a CPU. Accordingly, if you have access to a GPU, it is highly recommended to install CuPy, which performs the equivalent NumPy array operations on the GPU (https://docs.cupy.dev/en/stable/install.html). This is not necessary to run raw voltage generation, but will highly accelerate the pipeline.

Once you have CuPy installed, to enable GPU acceleration, you must set `SETIGEN_ENABLE_GPU` to '1' in the shell or in Python via `os.environ`. It can also be useful to set `CUDA_VISIBLE_DEVICES` to specify which GPUs to use. The following enables GPU usage and specifies to use the GPU indexed as 0.

In Bash:

```
export SETIGEN_ENABLE_GPU=1
export CUDA_VISIBLE_DEVICES=0
```

In Python:

```
import os
os.environ['SETIGEN_ENABLE_GPU'] = '1'
os.environ['CUDA_VISIBLE_DEVICES'] = '0'
```

1.5.3 Details behind classes

Adding noise and signal sources

If your application uses two polarizations, an Antenna's data streams are available via the `Antenna.x` and `Antenna.y` attributes. For one polarization, only the former is available. We can inject noise and signal sources to these individual data streams. Note that you can still add signal sources after the RawVoltageBackend is created; real voltages are only computed at execution time.

Real voltage noise is modeled as ideal Gaussian noise. Note that this actually stores a function with the DataStream that isn't evaluated until `get_samples()` is actually called:

```
antenna.x.add_noise(v_mean=0,
                    v_std=1)
```

For convenience, the `Antenna.streams` attribute is a list containing the available data streams for each polarization. So, to add a Gaussian noise source (with the same statistics) to each antenna, you can do:

```
for stream in antenna.streams:
    stream.add_noise(v_mean=0,
                     v_std=1)
```

This will adjust the `DataStream.noise_std` parameter for each polarization, which is also accessible using `DataStream.get_total_noise_std()`.

We can also add drifting cosine signals to each stream:

```
stream.add_constant_signal(f_start=6002.2e6,
                           drift_rate=-2*u.Hz/u.s,
                           level=0.002,
                           phase=0)
```

Here, *f_start* is the starting frequency, *drift_rate* is the change in frequency per time in Hz/s, *level* is the amplitude of the cosine signal, and *phase* is the phase offset in radians.

Custom signal sources

To add custom signal source functions, you can use the `add_signal` method:

```
stream.add_signal(my_signal_func)
```

Signal source functions are Python functions that accept an array of times, in seconds, and output a corresponding sequence of real voltages. A simple example showing how you might generate Gaussian noise "signal":

```
def my_noise_source(ts):
    return np.random.normal(0, 1, len(ts))

stream.add_signal(my_noise_source)
```

As custom signals are added, the `DataStream.noise_std` parameter may no longer be accurate. In these cases, you may run *`update_noise()`* to estimate the noise based on a few voltages calculated from all noise and signal sources. Then, the proper noise standard deviation can be produced via `DataStream.get_total_noise_std()`.

You may also check out these example notebooks: 03_custom_signals.ipynb and 04_custom_signals_estimate_noise.ipynb.

Quantizers

The quantization classes are RealQuantizer and ComplexQuantizer. The latter actually uses the former for quantizing real and imaginary components independently. Quantization is run per polarization and antenna.

The quantizers attempt to map the voltage distribution to an ideal quantized normal distribution with a target FWHM. Voltages that extend past the range of integers representable by *num_bits* are clipped. The standard deviation of the voltage distribution is calculated as they are collected, on a subset of *stats_calc_num_samples* samples. By default, this calculation is run on every pass through the pipeline, but can be limited to periodic calculations using the *stats_calc_period* initialization parameter. If this is set to anything besides a positive integer, the calculation will only be run on the first call and never again (which saves a lot of computation, but may not be the most accurate if the voltage distribution changes over time).

Polyphase filterbank

The PolyphaseFilterbank class implements and applies a PFB to quantized input voltages. A good introduction to PFBs is Danny C. Price 2016, "Spectrometers and Polyphase Filterbanks in Radio Astronomy" (http://arxiv.org/abs/1607.03579), as well as the accompanying Jupyter notebook.

The main things to keep in mind when initializing a PolyphaseFilterbank object are:

- *num_taps* controls the spectral profile of each individual coarse channel; the higher this is, the closer the spectral response gets to ideal
- *num_branches* controls the number of coarse channels; after the real FFT, we obtain *num_branches / 2* total coarse channels spanning the Nyquist range

Voltage backend

The RawVoltageBackend class connects the various components in the pipeline, allowing us to "record" only as much data as we currently need.

Behind the scenes, the backend actually uses a separate instance of each backend element per antenna and polarization. For example, if the backend is initialized with a single object instance for each the digitizer, filterbank, and requantizer, the backend object will make deep copies for each polarization in each antenna. This is done so that quantization (scaling) calculations are done independently for separate polarizations and antennas. Alternatively, you can initialize the backend with 2D lists of shape (num_antennas, num_pols) for each backend element, if, for example, there are variations in the desired *target_mean* and *target_fwhm* parameters.

1.5.4 Creating multi-antenna RAW files

To simulate interferometric pipelines, it may be useful to synthesize raw voltage data from multiple antennas. The MultiAntennaArray class supports exactly this, creating a list of sub-Antennas each with an associated integer delay (in time samples). In addition to the individual data streams that allow you to add noise and signals to each Antenna, there are "background" data streams `bg_x` and `bg_y` in MultiAntennaArray, representing common / correlated noise

or RFI that each Antenna can see, subject to the (relative) delay. If there are no delays, the background data streams will be perfectly correlated for each antenna.

Here's an example initialization for a 3 antenna array:

```
sample_rate = 3e9
delays = np.array([0, 1e-6, 2e-6]) * sample_rate
maa = stg.voltage.MultiAntennaArray(num_antennas=3,
                                    sample_rate=sample_rate,
                                    fch1=6*u.GHz,
                                    ascending=False,
                                    num_pols=2,
                                    delays=delays)
```

You can access both background data streams using the `MultiAntennaArray.bg_streams` attribute:

```
for stream in maa.bg_streams:
    stream.add_noise(v_mean=0,
                     v_std=1)
    stream.add_constant_signal(f_start=5998.9e6,
                               drift_rate=0*u.Hz/u.s,
                               level=0.0025)
```

Then, instead of passing a single Antenna into a RawVoltageBackend object, you pass in the MultiAntennaArray:

```
rvb = stg.voltage.RawVoltageBackend(maa,
                                    digitizer=digitizer,
                                    filterbank=filterbank,
                                    requantizer=requantizer,
                                    start_chan=0,
                                    num_chans=64,
                                    block_size=6291456,
                                    blocks_per_file=128,
                                    num_subblocks=32)
```

The RawVoltageBackend will get samples from each Antenna, accounting for the background data streams intrinsic to the MultiAntennaArray, subject to each Antenna's delays.

You may also check out this example notebook: 01_multi_antenna_raw_file_gen.ipynb.

1.5.5 Injecting signals at a desired SNR

With noise and multiple signal processing operations, including an FFT, it can be a bit tricky to choose the correct amplitude of a cosine signal at the beginning of the pipeline to achieve a desired signal-to-noise ratio (SNR) in the final finely channelized intensity data products. *`setigen.voltage.level_utils`* has a few helper functions to facilitate this, depending on the nature of the desired cosine signal.

Since the final SNR depends on the fine channelization FFT length and the time integration factor, as well as parameters inherent to the data production, we need external functions to help calculate an amplitude, or level, for our cosine signal.

First off, assume we are creating a non-drifting cosine signal. If the signal is at the center of a finely channelized frequency bin, *`get_level()`* gives the appropriate cosine amplitude to achieve a given SNR if the initial real Gaussian noise has a variance of 1:

```
fftlength = 1024
num_blocks = 1
```

(continues on next page)

(continued from previous page)

```
signal_level = stg.voltage.get_level(snr=10,
                                     raw_voltage_backend=rvb,
                                     fftlength=fftlength,
                                     num_blocks=num_blocks,
                                     length_mode='num_blocks')
```

If the noise in the DataStream doesn't have a variance of 1, we need to adjust this signal level by multiplying by `DataStream.get_total_noise_std()`. Note that this method also works for data streams within Antennas that are part of MultiAntennaArrays, since it will automatically account for the background noise in the array. Since the noise power is squared during fine channelization, the signal amplitude should go linearly as a function of the standard deviation of the noise.

If the signal is non-drifting, in general the spectral response will go as `1/sinc^2(x)`, where `x` is the fractional error off of the center of the spectral bin. To calculate the corresponding amount to adjust signal_level, you can use *`get_leakage_factor()`*. This technically calculates `1/sinc(x)`, which is inherently squared naturally along with the cosine signal amplitude during fine channelization.

To account for drift rates, it gets a bit more complicated; in general, if the drift rate is larger than a pixel by pixel slope of 1 in the final spectrogram data products, dividing the initial non-drifting power by that pixel by pixel slope will result in the new power. In other words, if *s* is the drift rate corresponding to a final pixel by pixel slope of 1, then a signal drifting by *2*s* will have half the SNR of the non-drifting signal. For a given RawVoltageBackend and reduced data product parameters *fftlength* and *int_factor* (integration factor), you can calculate *s* via *`get_unit_drift_rate()`*. However, the situation is much more complicated for drift rates between 0 and *s*, so setigen doesn't currently automatically calculate the requisite shift in power. Note that if you'd like to adjust the power for drift rates higher than *s*, you should adjust the amplitude (level) of the cosine signal by the square root of the relevant factor.

An example accounting for multiple effects like these:

```
f_start = 6003.1e6
leakage_factor = stg.voltage.get_leakage_factor(f_start, rvb, fftlength)
for stream in antenna.streams:
    level = stream.get_total_noise_std() * leakage_factor * signal_level
    stream.add_constant_signal(f_start=f_start,
                               drift_rate=0*u.Hz/u.s,
                               level=level)
```

You may also check out this example notebook: 05_raw_file_gen_snr.ipynb.

1.5.6 Injecting signals starting from existing RAW files

In addition to recording entirely synthetic voltage data, we can also inject signals onto existing RAW files. This approach is somehwat limited, since the data in existing RAW files have necessarily already been digitized, channelized, and requantized using hardware at the telescope; we cannot add the time series real voltage signals.

Instead, we can use parameters from the RAW data to create synthetic data streams, and add the corresponding complex RAW voltages together as our "injection". Of course, we want to make sure the synthetic data properties match those of the RAW files, so we have a helper function `get_raw_params` that returns a dictionary with relevant properties. Note that we still need to specify which coarse channel the recorded data starts from, since this isn't saved in the header.

```
start_chan = 0
input_file_stem = 'example_snr'

raw_params = stg.voltage.get_raw_params(input_file_stem=input_file_stem,
                                        start_chan=start_chan)
```

(continues on next page)

(continued from previous page)

```
antenna = stg.voltage.Antenna(sample_rate=sample_rate,
                              **raw_params)
```

To then create a RawVoltageBackend, we use the class method `RawVoltageBackend.from_data()`, where *input_file_stem* is the filename stem as used by `rawspec`.

```
rvb = stg.voltage.RawVoltageBackend.from_data(input_file_stem=input_file_stem,
                                              antenna_source=antenna,
                                              digitizer=digitizer,
                                              filterbank=filterbank,
                                              start_chan=start_chan,
                                              num_subblocks=32)
```

There are a few things to keep in mind here. Since we don't have access to the original noise distribution in real voltage space for the recorded RAW data (as it was quantized), it may be tough to inject at specific SNR levels. Also, if we create an Antenna with only cosine-like signals, the distribution of voltages will look highly non-Gaussian. So, if we attempt to digitize or requantize this normally, we risk distorting the data and introducing artifacts. To avoid this, if the Antenna has no injected Gaussian noise source, we can run `RawVoltageBackend.record()` with parameter `digitize=False`. Then, the signals will be channelized and quantized as if they were embedded in zero-mean Gaussian noise with standard deviation 1. Now, if there *is* a noise source, you can leave `digitize=True` (the default).

```
rvb.record(output_file_stem='example_snr_input',
           header_dict={'TELESCOP': 'GBT'},
           digitize=False,
           verbose=True)
```

In the `record()` call, if no *num_blocks* or *obs_length* is specified, data will be recorded matching the total length / size of the input data. You may specify these parameters to record a smaller amount of data (starting from the beginning of the input), but of course you can't produce a longer recording than what is present in the input.

Behind the scenes, at each iteration, the backend will read in a full data block from disk, and set requantizer statistics (target mean, target standard deviation) for each (antenna, polarization) pair for the real and imaginary quantizer components. Then, the synthetic data passing through the pipeline is requantized to the corresponding standard deviations in each complex component, but instead of centering to the target mean, they are centered to zero mean. This is so that when we add the synthetic data to the existing data, we don't change the overall voltage means. After these are added together, we finally requantize once more with the same requantizers, to the target mean and standard deviations. This procedure is done to match the existing data statistics and magnitudes as best as possible.

You may also check out this example notebook: 06_starting_from_existing_raw_files.ipynb.

1.6 setigen API reference

1.6.1 Signal parameter functions

setigen.funcs package

setigen.funcs.f_profiles module

Sample spectral profiles for signal injection.

For any given time sample, these functions map out the intensity in the frequency direction (centered at a particular frequency).

setigen.funcs.f_profiles.**box_f_profile**(*width*)
Square intensity profile in the frequency direction.

setigen.funcs.f_profiles.**gaussian_f_profile**(*width*)
Gaussian profile; width is the FWHM of the profile.

setigen.funcs.f_profiles.**multiple_gaussian_f_profile**(*width*)
Example adding multiple Gaussians in the frequency direction.

setigen.funcs.f_profiles.**lorentzian_f_profile**(*width*)
Lorentzian profile; width is the FWHM of the profile.

setigen.funcs.f_profiles.**voigt_f_profile**(*g_width*, *l_width*)
Voigt profile; g_width and l_width are the FWHMs of the Gaussian and Lorentzian profiles.

Further information here: https://en.wikipedia.org/wiki/Voigt_profile.

setigen.funcs.f_profiles.**sinc2_f_profile**(*width*, *trunc=True*)
Sinc squared profile; width is the FWHM of the squared normalized sinc function.

The trunc parameter controls whether or not the sinc squared profile is truncated at the first root (e.g. zeroed out for more distant frequencies).

setigen.funcs.paths module

Sample signal paths for signal injection.

For any given starting frequency, these functions map out the path of a signal as a function of time in time-frequency space.

setigen.funcs.paths.**constant_path**(*f_start*, *drift_rate*)
Constant drift rate.

setigen.funcs.paths.**squared_path**(*f_start*, *drift_rate*)
Quadratic signal path; drift_rate here only refers to the starting slope.

setigen.funcs.paths.**sine_path**(*f_start*, *drift_rate*, *period*, *amplitude*)
Sine path in time-frequency space.

setigen.funcs.paths.**simple_rfi_path**(*f_start*, *drift_rate*, *spread*, *spread_type='uniform'*, *rfi_type='stationary'*)
A crude simulation of one style of RFI that shows up, in which the signal jumps around in frequency. This example samples the center frequency for each time sample from either a uniform or normal distribution. 'spread' defines the range for center frequency variations.

Argument 'spread_type' can be either 'uniform' or 'normal'.

Argument 'rfi_type' can be either 'stationary' or 'random_walk'; 'stationary' only offsets with respect to a straight-line path, but 'random_walk' accumulates frequency offsets over time.

setigen.funcs.t_profiles module

Sample intensity profiles for signal injection.

These functions calculate the signal intensity and variation in the time direction.

setigen.funcs.t_profiles.**constant_t_profile**(*level=1*)
Constant intensity profile.

setigen.funcs.t_profiles.**sine_t_profile**(*period*, *phase=0*, *amplitude=1*, *level=1*)
Intensity varying as a sine curve, where level is the mean intensity.

setigen.funcs.t_profiles.**periodic_gaussian_t_profile**(*pulse_width*, *period*, *phase=0*, *pulse_offset_width=0*, *pulse_direction='rand'*, *pnum=3*, *amplitude=1*, *level=1*, *min_level=0*)
Intensity varying as Gaussian pulses, allowing for variation in the arrival time of each pulse.

period and *phase* give a baseline for pulse periodicity.

pulse_direction can be 'up', 'down', or 'rand', referring to whether the intensity increases or decreases from the baseline *level*. *amplitude* is the magnitude of each pulse. *min_level* is the minimum intensity, default is 0.

pulse_offset_width encodes the variation in the pulse period, whereas *pulse_width* is the width of individual pulses. Both are modeled as Gaussians, where 'width' refers to the FWHM of the distribution.

pnum is the number of Gaussians pulses to consider when calculating the intensity at each timestep. The higher this number, the more accurate the intensities.

setigen.funcs.bp_profiles module

setigen.funcs.bp_profiles.**constant_bp_profile**(*level=1*)
Constant bandpass profile.

setigen.funcs.func_utils module

setigen.funcs.func_utils.**gaussian**(*x*, *x0*, *sigma*)

setigen.funcs.func_utils.**lorentzian**(*x*, *x0*, *gamma*)

setigen.funcs.func_utils.**voigt**(*x*, *x0*, *sigma*, *gamma*)

setigen.funcs.func_utils.**voigt_fwhm**(*g_width*, *l_width*)
Accurate to 0.0003% for a pure Lorentzian profile, precise for a pure Gaussian.

Source: https://en.wikipedia.org/wiki/Voigt_profile.

1.6.2 setigen.frame module

class setigen.frame.**Frame**(*waterfall=None*, *fchans=None*, *tchans=None*, *df=None*, *dt=None*, *fch1=<Quantity 8. GHz>*, *ascending=False*, *data=None*)
Bases: object

Facilitate the creation of entirely synthetic radio data (narrowband signals + background noise) as well as signal injection into existing observations.

__init__(*waterfall=None*, *fchans=None*, *tchans=None*, *df=None*, *dt=None*, *fch1=<Quantity 8. GHz>*, *ascending=False*, *data=None*)
Initialize a Frame object either from an existing .fil/.h5 file or from frame resolution / size.

If you are initializing based on a .fil or .h5, pass in either the filename or the Waterfall object into the waterfall keyword.

Otherwise, you can initialize a frame by specifying the parameters fchans, tchans, df, dt, and potentially fch1, if it's important to specify frequencies (8*u.GHz is an arbitrary but reasonable choice otherwise).

The *data* keyword is only necessary if you are also preloading data that matches your specified frame dimensions and resolutions.

Parameters

- **waterfall** (*str or Waterfall, optional*) – Name of filterbank file or Waterfall object for preloading data
- **fchans** (*int, optional*) – Number of frequency samples
- **tchans** (*int, optional*) – Number of time samples
- **df** (*astropy.Quantity, optional*) – Frequency resolution (e.g. in u.Hz)
- **dt** (*astropy.Quantity, optional*) – Time resolution (e.g. in u.s)
- **fch1** (*astropy.Quantity, optional*) – Frequency of channel 1, as in filterbank file headers (e.g. in u.Hz). If ascending=True, fch1 is the minimum frequency; if ascending=False (default), fch1 is the maximum frequency.
- **ascending** (*bool, optional*) – Specify whether frequencies should be in ascending order, so that fch1 is the minimum frequency. Default is False, for which fch1 is the maximum frequency. This is overwritten if a waterfall object is provided, where ascending will be automatically determined by observational parameters.
- **data** (*ndarray, optional*) – 2D array of intensities to preload into frame

classmethod from_data(*df*, *dt*, *fch1*, *ascending*, *data*)
Instantiate Frame using a data array.

classmethod from_waterfall(*waterfall*)
Instantiate Frame using a filterbank file or blimpy Waterfall object.

zero_data()
Resets data to a numpy array of zeros.

mean()

std()

get_total_stats()

get_noise_stats()

add_noise(*x_mean*, *x_std=None*, *x_min=None*, *noise_type='chi2'*)
By default, synthesizes radiometer noise based on a chi-squared distribution. Alternately, can generate pure Gaussian noise.

Specifying noise_type='chi2' will only use x_mean, and ignore other parameters. Specifying noise_type='normal' or 'gaussian' will use all arguments (if provided).

When adding Gaussian noise to the frame, the minimum is simply a lower bound for intensities in the data (e.g. it may make sense to cap intensities at 0), but this is optional.

add_noise_from_obs(*x_mean_array=None*, *x_std_array=None*, *x_min_array=None*, *share_index=True*, *noise_type='chi2'*)
By default, synthesizes radiometer noise based on a chi-squared distribution. Alternately, can generate pure Gaussian noise.

If no arrays are specified from which to sample, noise samples will be drawn from saved GBT C-Band observations at (dt, df) = (1.4 s, 1.4 Hz) resolution, from frames of shape (tchans, fchans) = (32, 1024). These sample noise parameters consist of 126500 samples for mean, std, and min of each observation.

Specifying noise_type='chi2' will only use x_mean_array (if provided), and ignore other parameters. Specifying noise_type='normal' will use all arrays (if provided).

Note: this method will attempt to scale the noise parameters to match self.dt and self.df. This assumes that the observation data products are *not* normalized by the FFT length used to contstruct them.

Parameters

- **x_mean_array** (*ndarray*) – Array of potential means
- **x_std_array** (*ndarray*) – Array of potential standard deviations
- **x_min_array** (*ndarray, optional*) – Array of potential minimum values
- **share_index** (*bool*) – Whether to select noise parameters from the same index across each provided array. If share_index is True, then each array must be the same length.
- **noise_type** (*string*) – Distribution to use for synthetic noise; 'chi2', 'normal', 'gaussian'

add_signal(*path*, *t_profile*, *f_profile*, *bp_profile*, *bounding_f_range=None*, *integrate_path=False*, *integrate_t_profile=False*, *integrate_f_profile=False*, *t_subsamples=10*, *f_subsamples=10*)

Generates synthetic signal.

Adds a synethic signal using given path in time-frequency domain and brightness profiles in time and frequency directions.

Parameters

- **path** (*function, np.ndarray, list, float*) – Function in time that returns frequencies, or provided array or single value of frequencies for the center of the signal at each time sample
- **t_profile** (*function, np.ndarray, list, float*) – Time profile: function in time that returns an intensity (scalar), or provided array or single value of intensities at each time sample
- **f_profile** (*function*) – Frequency profile: function in frequency that returns an intensity (scalar), relative to the signal frequency within a time sample. Note that unlike the other parameters, this must be a function
- **bp_profile** (*function, np.ndarray, list, float*) – Bandpass profile: function in frequency that returns a relative intensity (scalar, between 0 and 1), or provided array or single value of relative intensities at each frequency sample
- **bounding_f_range** (*tuple*) – Tuple (bounding_min, bounding_max) that constrains the computation of the signal to only a range in frequencies
- **integrate_path** (*bool, optional*) – Option to average path along time to get a more accurate frequency position in t-f space. Note that this option only makes sense if the provided path can be evaluated at the sub frequency sample level (e.g. as opposed to returning a pre-computed array of frequencies of length *tchans*). Makes *t_subsamples* calculations per time sample.
- **integrate_t_profile** (*bool, optional*) – Option to integrate t_profile in the time direction. Note that this option only makes sense if the provided t_profile can be evaluated at the sub time sample level (e.g. as opposed to returning an array of intensities of length *tchans*). Makes *t_subsamples* calculations per time sample.
- **integrate_f_profile** (*bool, optional*) – Option to integrate f_profile in the frequency direction. Makes *f_subsamples* calculations per time sample.
- **t_subsamples** (*int, optional*) – Number of bins for integration in the time direction, using Riemann sums

- **f_subsamples** (*int, optional*) – Number of bins for integration in the frequency direction, using Riemann sums

Returns **signal** – Two-dimensional NumPy array containing synthetic signal data

Return type ndarray

Examples

Here's an example that creates a linear Doppler-drifted signal with chi-squared noise with sampled parameters:

```
>>> from astropy import units as u
>>> import setigen as stg
>>> fchans = 1024
>>> tchans = 32
>>> df = 2.7939677238464355*u.Hz
>>> dt = tsamp = 18.253611008*u.s
>>> fch1 = 6095.214842353016*u.MHz
>>> frame = stg.Frame(fchans=fchans,
                      tchans=tchans,
                      df=df,
                      dt=dt,
                      fch1=fch1)
>>> noise = frame.add_noise(x_mean=10)
>>> signal = frame.add_signal(stg.constant_path(f_start=frame.get_
↪frequency(200),
                                                drift_rate=2*u.Hz/u.s),
                              stg.constant_t_profile(level=frame.get_
↪intensity(snr=30)),
                              stg.gaussian_f_profile(width=40*u.Hz),
                              stg.constant_bp_profile(level=1))
```

Saving the noise and signals individually may be useful depending on the application, but the combined data can be accessed via frame.get_data(). The synthetic signal can then be visualized and saved within a Jupyter notebook using:

```
>>> %matplotlib inline
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure(figsize=(10, 6))
>>> frame.render()
>>> plt.savefig('image.png', bbox_inches='tight')
>>> plt.show()
```

To run within a script, simply exclude the first line: `%matplotlib inline`.

add_constant_signal(*f_start*, *drift_rate*, *level*, *width*, *f_profile_type='gaussian'*)

A wrapper around add_signal() that injects a constant intensity, constant drift_rate signal into the frame.

Parameters

- **f_start** (*astropy.Quantity*) – Starting signal frequency
- **drift_rate** (*astropy.Quantity*) – Signal drift rate, in units of frequency per time
- **level** (*float*) – Signal intensity
- **width** (*astropy.Quantity*) – Signal width in frequency units

- **f_profile_type** (*str*) – Can be 'box', 'sinc2', 'gaussian', 'lorentzian', or 'voigt', based on the desired spectral profile

Returns **signal** – Two-dimensional NumPy array containing synthetic signal data

Return type ndarray

get_index(*frequency*)
Convert frequency to closest index in frame.

get_frequency(*index*)
Convert index to frequency.

get_intensity(*snr*)
Calculates intensity from SNR, based on estimates of the noise in the frame.

Note that there must be noise present in the frame for this to make sense.

get_snr(*intensity*)
Calculate SNR from intensity.

Note that there must be noise present in the frame for this to make sense.

get_drift_rate(*start_index*, *end_index*)

get_info()

get_data(*use_db=False*)

get_metadata()

set_metadata(*new_metadata*)
Set custom metadata using a dictionary new_metadata.

update_metadata(*new_metadata*)
Append custom metadata using a dictionary new_metadata.

add_metadata(*new_metadata*)

render(*use_db=False*)

bl_render(*use_db=True*)

get_waterfall()
Return current frame as a Waterfall object. Note: some filterbank metadata may not be accurate anymore, depending on prior frame manipulations.

save_fil(*filename*, *max_load=1*)
Save frame data as a filterbank file (.fil).

save_hdf5(*filename*, *max_load=1*)
Save frame data as an HDF5 file.

save_h5(*filename*, *max_load=1*)
Save frame data as an HDF5 file.

save_npy(*filename*)
Save frame data as an .npy file.

load_npy(*filename*)
Load frame data from a .npy file.

save_pickle(*filename*)
Save entire frame as a pickled file (.pickle).

classmethod load_pickle(*filename*)
Load Frame object from a pickled file (.pickle), created with Frame.save_pickle.

1.6.3 setigen.split_utils module

setigen.split_utils.**split_waterfall_generator**(*waterfall_fn*, *fchans*, *tchans=None*, *f_shift=None*)
Creates a generator that returns smaller Waterfall objects by 'splitting' an input filterbank file according to the number of frequency samples.

Since this function only loads in data in chunks according to fchans, it handles very large observations well. Specifically, it will not attempt to load all the data into memory before splitting, which won't work when the data is very large anyway.

Parameters

- **waterfall_fn** (*str*) – Filterbank filename with .fil extension
- **fchans** (*int*) – Number of frequency samples per new filterbank file
- **tchans** (*int, optional*) – Number of time samples to select - will default from start of observation. If None, just uses the entire integration time
- **f_shift** (*int, optional*) – Number of samples to shift when splitting filterbank. If None, defaults to *f_shift=fchans* so that there is no overlap between new filterbank files

Returns **waterfall** – A blimpy Waterfall object containing a smaller section of the data

Return type Waterfall

setigen.split_utils.**split_fil**(*waterfall_fn*, *output_dir*, *fchans*, *tchans=None*, *f_shift=None*)
Creates a set of new filterbank files by 'splitting' an input filterbank file according to the number of frequency samples.

Parameters

- **waterfall_fn** (*str*) – Filterbank filename with .fil extension
- **output_dir** (*str*) – Directory for new filterbank files
- **fchans** (*int*) – Number of frequency samples per new filterbank file
- **tchans** (*int, optional*) – Number of time samples to select - will default from start of observation. If None, just uses the entire integration time
- **f_shift** (*int, optional*) – Number of samples to shift when splitting filterbank. If None, defaults to *f_shift=fchans* so that there is no overlap between new filterbank files

Returns **split_fns** – List of new filenames

Return type list of str

setigen.split_utils.**split_array**(*data*, *f_sample_num=None*, *t_sample_num=None*, *f_shift=None*, *t_shift=None*, *f_trim=False*, *t_trim=False*)
Splits NumPy arrays into a list of smaller arrays according to limits in frequency and time. This doesn't reduce/combine data, it simply cuts the data into smaller chunks.

Parameters **data** (*ndarray*) – Time-frequency data

Returns **split_data** – List of new time-frequency data frames

Return type list of ndarray

1.6.4 setigen.distributions module

setigen.distributions.**fwhm**(*sigma*)
Get full width at half maximum (FWHM) for a provided sigma / standard deviation, assuming a Gaussian distribution.

setigen.distributions.**gaussian**(*x_mean*, *x_std*, *shape*)

setigen.distributions.**truncated_gaussian**(*x_mean*, *x_std*, *x_min*, *shape*)
Samples from a normal distribution, but enforces a minimum value.

setigen.distributions.**chi2**(*x_mean*, *chi2_df*, *shape*)
Chi-squared distribution centered at a specific mean.

Parameters

- **x_mean** (*float*) –
- **chi2_df** (*int*) – Degrees of freedom for chi-squared
- **shape** (*list*) – Shape of output noise array

Returns **dist** – Array of chi-squared noise

Return type ndarray

1.6.5 setigen.waterfall_utils module

setigen.waterfall_utils.**max_freq**(*waterfall*)
Returns central frequency of the highest-frequency bin in a .fil file.

Parameters **waterfall** (*str or Waterfall*) – Name of filterbank file or Waterfall object

Returns **fmax** – Maximum frequency in data

Return type float

setigen.waterfall_utils.**min_freq**(*waterfall*)
Returns central frequency of the lowest-frequency bin in a .fil file.

Parameters **waterfall** (*str or Waterfall*) – Name of filterbank file or Waterfall object

Returns **fmin** – Minimum frequency in data

Return type float

setigen.waterfall_utils.**get_data**(*waterfall*, *use_db=False*)
Gets time-frequency data from filterbank file as a 2d NumPy array.

Note: when multiple Stokes parameters are supported, this will have to be expanded.

Parameters **waterfall** (*str or Waterfall*) – Name of filterbank file or Waterfall object

Returns **data** – Time-frequency data

Return type ndarray

setigen.waterfall_utils.**get_fs**(*waterfall*)
Gets frequency values from filterbank file.

Parameters **waterfall** (*str or Waterfall*) – Name of filterbank file or Waterfall object

Returns **fs** – Frequency values

Return type ndarray

setigen.waterfall_utils.**get_ts**(*waterfall*)
Gets time values from filterbank file.

Parameters **waterfall** (*str or Waterfall*) – Name of filterbank file or Waterfall object

Returns **ts** – Time values

Return type ndarray

1.6.6 setigen.sample_from_obs module

setigen.sample_from_obs.**sample_gaussian_params**(*x_mean_array*, *x_std_array*, *x_min_array=None*)
Sample Gaussian parameters (mean, std, min) from provided arrays.

Typical usage would be for select Gaussian noise properties for injection into data frames.

Parameters

- **x_mean_array** (*ndarray*) – Array of potential means
- **x_std_array** (*ndarray*) – Array of potential standard deviations
- **x_min_array** (*ndarray, optional*) – Array of potential minimum values

Returns

- *x_mean* – Selected mean of distribution
- *x_std* – Selected standard deviation of distribution
- *x_min* – If x_min_array provided, selected minimum of distribution

setigen.sample_from_obs.**get_parameter_distributions**(*waterfall_fn*, *fchans*, *tchans=None*, *f_shift=None*)
Calculate parameter distributions for the mean, standard deviation, and minimum of split filterbank data from real observations.

Parameters

- **waterfall_fn** (*str*) – Filterbank filename with .fil extension
- **fchans** (*int*) – Number of frequency samples per new filterbank file
- **tchans** (*int, optional*) – Number of time samples to select - will default from start of observation. If None, just uses the entire integration time
- **f_shift** (*int, optional*) – Number of samples to shift when splitting filterbank. If None, defaults to *f_shift=f_window* so that there is no overlap between new filterbank files

Returns

- *x_mean_array* – Distribution of means calculated from observations
- *x_std_array* – Distribution of standard deviations calculated from observations
- *x_min_array* – Distribution of minimums calculated from observations

setigen.sample_from_obs.**get_mean_distribution**(*waterfall_fn*, *fchans*, *tchans=None*, *f_shift=None*)
Calculate parameter distributions for the mean of split filterbank frames from real observations.

Parameters

- **waterfall_fn** (*str*) – Filterbank filename with .fil extension
- **fchans** (*int*) – Number of frequency samples per new filterbank file

- **tchans** (*int, optional*) – Number of time samples to select - will default from start of observation. If None, just uses the entire integration time
- **f_shift** (*int, optional*) – Number of samples to shift when splitting filterbank. If None, defaults to *f_shift=f_window* so that there is no overlap between new filterbank files

Returns Distribution of means calculated from observations

Return type x_mean_array

1.6.7 setigen.stats module

setigen.stats.**exclude_and_flatten**(*data*, *exclude=0*)

setigen.stats.**get_mean**(*data*, *exclude=0*)

setigen.stats.**get_std**(*data*, *exclude=0*)

setigen.stats.**get_min**(*data*, *exclude=0*)

setigen.stats.**compute_frame_stats**(*data*, *exclude=0*)

1.6.8 setigen.time_freq_utils module

setigen.time_freq_utils.**db**(*x*)
Converts to dB.

setigen.time_freq_utils.**integrate_frame**(*frame*, *normalize=False*)
Integrate over time using mean (not sum).

setigen.time_freq_utils.**integrate_frame_subdata**(*data*, *frame=None*, *normalize=False*)
Integrate a chunk of data assuming frame statistics using mean (not sum).

setigen.time_freq_utils.**normalize**(*data*, *cols=0*, *exclude=0.0*, *to_db=False*, *use_median=False*)
Normalize data per frequency channel so that the noise level in data is controlled; using mean or median filter.

Uses a sliding window to calculate mean and standard deviation to preserve non-drifted signals. Excludes a fraction of brightest pixels to better isolate noise.

Parameters

- **data** (*ndarray*) – Time-frequency data
- **cols** (*int*) – Number of columns on either side of the current frequency bin. The width of the sliding window is thus 2 * cols + 1
- **exclude** (*float, optional*) – Fraction of brightest samples in each frequency bin to exclude in calculating mean and standard deviation
- **to_db** (*bool, optional*) – Convert values to decibel equivalents *before* normalization
- **use_median** (*bool, optional*) – Use median and median absolute deviation instead of mean and standard deviation

Returns **normalized_data** – Normalized data

Return type ndarray

setigen.time_freq_utils.**normalize_by_max**(*data*)
Simple normalization by dividing out by the brightest pixel.

1.6.9 setigen.unit_utils module

This module contains a couple unit conversion utilities used in frame.Frame.

In general, we rely on astropy units for conversions, and note that float values are assumed to be in SI units (e.g. Hz, s).

setigen.unit_utils.**cast_value**(*value*, *unit*)

If value is already an astropy Quantity, then cast it into the desired unit. Otherwise, value is assumed to be a float and converted directly to the desired unit.

setigen.unit_utils.**get_value**(*value*, *unit=None*)

This function converts a value, which may be a float or astropy Quantity, into a float (in terms of a desired unit).

If we know that value is an astropy Quantity, then grabbing the value is simple (and we can cast this to a desired unit, if we need to change this.

If value is already a float, it simply returns value.

1.7 setigen.voltage API reference

1.7.1 setigen.voltage.backend module

class setigen.voltage.backend.**RawVoltageBackend**(*antenna_source*, *digitizer=<setigen.voltage.quantization.RealQuantizer object>*, *filterbank=<setigen.voltage.polyphase_filterbank.PolyphaseFilterba object>*, *requantizer=<setigen.voltage.quantization.ComplexQuantizer object>*, *start_chan=0*, *num_chans=64*, *block_size=134217728*, *blocks_per_file=128*, *num_subblocks=32*)

Bases: object

Central class that wraps around antenna sources and backend elements to facilitate the creation of GUPPI RAW voltage files from synthetic real voltages.

__init__(*antenna_source*, *digitizer=<setigen.voltage.quantization.RealQuantizer object>*, *filterbank=<setigen.voltage.polyphase_filterbank.PolyphaseFilterbank object>*, *requantizer=<setigen.voltage.quantization.ComplexQuantizer object>*, *start_chan=0*, *num_chans=64*, *block_size=134217728*, *blocks_per_file=128*, *num_subblocks=32*)

Initialize a RawVoltageBackend object, with an input antenna source (either Antenna or MultiAntennaArray), and backend elements (digitizer, filterbank, requantizer). Also, details behind the RAW file format and recording are specified on initialization, such as which coarse channels are saved and the size of recording blocks.

Parameters

- **antenna_source** (Antenna *or* MultiAntennaArray) – Antenna or MultiAntennaArray, from which real voltage data is created
- **digitizer** (RealQuantizer *or* ComplexQuantizer, *or list, optional*) – Quantizer used to digitize input voltages. Either a single object to be used as a template for each antenna and polarization, or a 2D list of quantizers of shape (num_antennas, num_pols).

- **filterbank** (PolyphaseFilterbank, *or list, optional*) – Polyphase filterbank object used to channelize voltages. Either a single object to be used as a template for each antenna and polarization, or a 2D list of filterbank objects of shape (num_antennas, num_pols).
- **requantizer** (ComplexQuantizer, *or list, optional*) – Quantizer used on complex channelized voltages. Either a single object to be used as a template for each antenna and polarization, or a 2D list of quantizers of shape (num_antennas, num_pols).
- **start_chan** (*int, optional*) – Index of first coarse channel to be recorded
- **num_chans** (*int, optional*) – Number of coarse channels to be recorded
- **block_size** (*int, optional*) – Recording block size, in bytes
- **blocks_per_file** (*int, optional*) – Number of blocks to be saved per RAW file
- **num_subblocks** (*int, optional*) – Number of partitions per block, used for computation. If 'num_subblocks'=1, one block's worth of data will be passed through the pipeline and recorded at once. Use this parameter to reduce memory load, especially when using GPU acceleration.

classmethod from_data(*input_file_stem*, *antenna_source*, *digitizer=<setigen.voltage.quantization.RealQuantizer object>*, *filterbank=<setigen.voltage.polyphase_filterbank.PolyphaseFilterbank object>*, *start_chan=0*, *num_subblocks=32*)

Initialize a RawVoltageBackend object, using existing RAW data as a background for signal insertion and recording. Compared to normal initialization, some parameters are inferred from the input data.

Parameters

- **input_file_stem** (*str*) – Filename or path stem to input RAW data
- **antenna_source** (Antenna *or* MultiAntennaArray) – Antenna or MultiAntennaArray, from which real voltage data is created
- **digitizer** (RealQuantizer *or* ComplexQuantizer, *or list, optional*) – Quantizer used to digitize input voltages. Either a single object to be used as a template for each antenna and polarization, or a 2D list of quantizers of shape (num_antennas, num_pols).
- **filterbank** (PolyphaseFilterbank, *or list, optional*) – Polyphase filterbank object used to channelize voltages. Either a single object to be used as a template for each antenna and polarization, or a 2D list of filterbank objects of shape (num_antennas, num_pols).
- **start_chan** (*int, optional*) – Index of first coarse channel to be recorded
- **num_subblocks** (*int, optional*) – Number of partitions per block, used for computation. If 'num_subblocks'=1, one block's worth of data will be passed through the pipeline and recorded at once. Use this parameter to reduce memory load, especially when using GPU acceleration.

Returns **backend** – Created backend object

Return type *RawVoltageBackend*

collect_data_block(*digitize=True*, *requantize=True*, *verbose=True*)

General function to actually collect data from the antenna source and return coarsely channelized complex voltages. Collects one block of data.

Parameters

- **digitize** (*bool, optional*) – Whether to quantize input voltages before the PFB

- **requantize** (*bool, optional*) – Whether to quantize output complex voltages after the PFB
- **verbose** (*bool, optional*) – Control whether tqdm prints progress messages

Returns **final_voltages** – Complex voltages formatted according to GUPPI RAW specifications; array of shape (num_chans * num_antennas, block_size / (num_chans * num_antennas))

Return type array

get_num_blocks(*obs_length*)

Calculate the number of blocks required as a function of observation length, in seconds. Note that only an integer number of blocks will be recorded, so the actual observation length may be shorter than the *obs_length* provided.

record(*output_file_stem*, *obs_length=None*, *num_blocks=None*, *length_mode='obs_length'*, *header_dict={}*, *digitize=True*, *verbose=True*)

General function to actually collect data from the antenna source and return coarsely channelized complex voltages. If input data is provided, only as much data as is in the input will be generated.

Parameters

- **output_file_stem** (*str*) – Filename or path stem; the suffix will be automatically appended
- **obs_length** (*float, optional*) – Length of observation in seconds, if in *obs_length* mode
- **num_blocks** (*int, optional*) – Number of data blocks to record, if in *num_blocks* mode
- **length_mode** (*str, optional*) – Mode for specifying length of observation, either *obs_length* in seconds or *num_blocks* in data blocks
- **header_dict** (*dict, optional*) – Dictionary of header values to set. Use to overwrite non-essential header values or add custom ones.
- **digitize** (*bool, optional*) – Whether to quantize input voltages before the PFB
- **verbose** (*bool, optional*) – Control whether tqdm prints progress messages

1.7.2 setigen.voltage.antenna module

class setigen.voltage.antenna.**Antenna**(*sample_rate=<Quantity 3. GHz>*, *fch1=<Quantity 0. GHz>*, *ascending=True*, *num_pols=2*, *t_start=0*, *seed=None*, ***kwargs*)

Bases: object

Models a radio antenna, with a DataStream per polarization (one or two).

__init__(*sample_rate=<Quantity 3. GHz>*, *fch1=<Quantity 0. GHz>*, *ascending=True*, *num_pols=2*, *t_start=0*, *seed=None*, ***kwargs*)

Initialize an Antenna object, which creates DataStreams for each polarization, under Antenna.x and Antenna.y (if there is a second polarization).

Parameters

- **sample_rate** (*float, optional*) – Physical sample rate, in Hz, for collecting real voltage data
- **fch1** (*astropy.Quantity, optional*) – Starting frequency of the first coarse channel, in Hz. If ascending=True, fch1 is the minimum frequency; if ascending=False (default), fch1 is the maximum frequency.

- **ascending** (*bool, optional*) – Specify whether frequencies should be in ascending or descending order. Default is True, for which fch1 is the minimum frequency.
- **num_pols** (*int, optional*) – Number of polarizations, can be 1 or 2
- **t_start** (*float, optional*) – Start time, in seconds
- **seed** (*int, optional*) – Integer seed between 0 and 2**32. If None, the random number generator will use a random seed.

set_time(*t*)
Set start time before next set of samples.

add_time(*t*)
Add time before next set of samples.

reset_start()
Reset the boolean that tracks whether this is the start of an observation.

get_samples(*num_samples*)
Retrieve voltage samples from each polarization.

Parameters **num_samples** (*int*) – Number of samples to get

Returns **samples** – Array of voltage samples, of shape (1, num_pols, num_samples)

Return type array

class setigen.voltage.antenna.**MultiAntennaArray**(*num_antennas*, *sample_rate=<Quantity 3. GHz>*, *fch1=<Quantity 0. GHz>*, *ascending=True*, *num_pols=2*, *delays=None*, *t_start=0*, *seed=None*, ***kwargs*)

Bases: object

Models a radio antenna array, with list of Antennas, subject to user-specified sample delays.

__init__(*num_antennas*, *sample_rate=<Quantity 3. GHz>*, *fch1=<Quantity 0. GHz>*, *ascending=True*, *num_pols=2*, *delays=None*, *t_start=0*, *seed=None*, ***kwargs*)
Initialize a MultiAntennaArray object, which creates a list of Antenna objects, each with a specified relative integer sample delay. Also creates background DataStreams to model coherent noise present in each Antenna, subject to that Antenna's delay.

Parameters

- **num_antennas** (*int*) – Number of Antennas in the array
- **sample_rate** (*float, optional*) – Physical sample rate, in Hz, for collecting real voltage data
- **fch1** (*astropy.Quantity, optional*) – Starting frequency of the first coarse channel, in Hz. If ascending=True, fch1 is the minimum frequency; if ascending=False (default), fch1 is the maximum frequency.
- **ascending** (*bool, optional*) – Specify whether frequencies should be in ascending or descending order. Default is True, for which fch1 is the minimum frequency.
- **num_pols** (*int, optional*) – Number of polarizations, can be 1 or 2
- **delays** (*array, optional*) – Array of integers specifying relative delay offsets per array with respect to the coherent antenna array background. If None, uses 0 delay for all Antennas.
- **t_start** (*float, optional*) – Start time, in seconds

- **seed** (*int, optional*) – Integer seed between 0 and 2**32. If None, the random number generator will use a random seed.

set_time(*t*)
Set start time before next set of samples.

add_time(*t*)
Add time before next set of samples.

reset_start()
Reset the boolean that tracks whether this is the start of an observation.

get_samples(*num_samples*)
Retrieve voltage samples from each antenna and polarization.

First, background data stream voltages are computed. Then, for each Antenna, voltages are retrieved per polarization and summed with the corresponding background voltages, subject to that Antenna's sample delay. An appropriate number of background voltage samples are cached with the Antenna, according to the delay, so that regardless of *num_samples*, each Antenna data stream has enough background samples to add.

Parameters **num_samples** (*int*) – Number of samples to get

Returns **samples** – Array of voltage samples, of shape (num_antennas, num_pols, num_samples)

Return type array

1.7.3 setigen.voltage.data_stream module

class setigen.voltage.data_stream.**DataStream**(*sample_rate=<Quantity 3. GHz>, fch1=<Quantity 0. GHz>, ascending=True, t_start=0, seed=None*)

Bases: object

Facilitate noise and signal injection in a real voltage time series data stream, for a single polarization. Noise and signal sources are functions, saved as properties of the DataStream, so that individual samples can be queried using get_samples().

__init__(*sample_rate=<Quantity 3. GHz>, fch1=<Quantity 0. GHz>, ascending=True, t_start=0, seed=None*)
Initialize a DataStream object with a sampling rate and frequency range.

By default, setigen.voltage does not employ heterodyne mixing and filtering to focus on a frequency bandwidth. Instead, the sensitive range is determined by these parameters; starting at the frequency *fch1* and spanning the Nyquist range *sample_rate / 2* in the increasing or decreasing frequency direction, as specified by *ascending*. Note that accordingly, the spectral response will be susceptible to aliasing, so take care that the desired frequency range is correct and that signals are injected at appropriate frequencies.

Parameters

- **sample_rate** (*float, optional*) – Physical sample rate, in Hz, for collecting real voltage data
- **fch1** (*astropy.Quantity, optional*) – Starting frequency of the first coarse channel, in Hz. If ascending=True, fch1 is the minimum frequency; if ascending=False (default), fch1 is the maximum frequency.
- **ascending** (*bool, optional*) – Specify whether frequencies should be in ascending or descending order. Default is True, for which fch1 is the minimum frequency.

- **t_start** (*float, optional*) – Start time, in seconds
- **seed** (*int, optional*) – Integer seed between 0 and 2**32. If None, the random number generator will use a random seed.

rng = None
Random number generator

set_time(*t*)
Set start time before next set of samples.

add_time(*t*)
Add time before next set of samples.

update_noise(*stats_calc_num_samples=10000*)
Replace self.noise_std by calculating out a few samples and estimating the standard deviation of the voltages.

Parameters **stats_calc_num_samples** (*int, optional*) – Maximum number of samples for use in estimating noise standard deviation

get_total_noise_std()
Get the standard deviation of the noise. If this DataStream is part of an array of Antennas, this will account for the background noise in the corresponding polarization.

Note that if this DataStream has custom signals or noise, it might not 'know' what the noise standard deviation is. In this case, one should run *update_noise()* to update the DataStream's estimate for the noise. Note that this actually runs *get_samples()* for the calculation, so if your custom signal functions have mutable properties, make sure to reset these (if necessary) before saving out data.

add_noise(*v_mean*, *v_std*)
Add Gaussian noise source to data stream. This essentially adds a lambda function that gets the appropriate number of noise samples to add to the voltage array when `get_samples()` is called. Updates noise property to reflect added noise.

Parameters

- **v_mean** (*float*) – Noise mean
- **v_std** (*float*) – Noise standard deviation

add_constant_signal(*f_start*, *drift_rate*, *level*, *phase=0*)
Adds a drifting cosine signal (linear chirp) as a signal source function.

Parameters

- **f_start** (*float*) – Starting signal frequency
- **drift_rate** (*float*) – Drift rate in Hz / s
- **level** (*float*) – Signal level or amplitude
- **phase** (*float*) – Phase, in radiations

add_signal(*signal_func*)
Wrapper function to add a custom signal source function.

get_samples(*num_samples*)
Retrieve voltage samples, based on noise and signal source functions.

If custom signals add complex voltages, the voltage array will be cast to complex type.

Parameters **num_samples** (*int*) – Number of samples to get

Returns **v** – Array of voltage samples

Return type array

class setigen.voltage.data_stream.**BackgroundDataStream**(*sample_rate=<Quantity 3. GHz>*, *fch1=<Quantity 0. GHz>*, *ascending=True*, *t_start=0*, *seed=None*, *antenna_streams=[]*)

Bases: *setigen.voltage.data_stream.DataStream*

Extends DataStream for background data in Antenna arrays.

__init__(*sample_rate=<Quantity 3. GHz>*, *fch1=<Quantity 0. GHz>*, *ascending=True*, *t_start=0*, *seed=None*, *antenna_streams=[]*)

Initialize a BackgroundDataStream object with a sampling rate and frequency range. The main extension is that we also pass in a list of DataStreams, belonging to all the Antennas within a MultiAntennaArray, for the same corresponding polarization. When noise is added to a BackgroundDataStream, the noise standard deviation gets propagated to each Antenna DataStream via the `DataStream.bg_noise_std` property.

Parameters

- **sample_rate** (*float, optional*) – Physical sample rate, in Hz, for collecting real voltage data
- **fch1** (*astropy.Quantity, optional*) – Starting frequency of the first coarse channel, in Hz. If ascending=True, fch1 is the minimum frequency; if ascending=False (default), fch1 is the maximum frequency.
- **ascending** (*bool, optional*) – Specify whether frequencies should be in ascending or descending order. Default is True, for which fch1 is the minimum frequency.
- **t_start** (*float, optional*) – Start time, in seconds
- **seed** (*int, optional*) – Integer seed between 0 and 2**32. If None, the random number generator will use a random seed.
- **antenna_streams** (*list of DataStream objects*) – List of DataStreams, which belong to the Antennas in a MultiAntennaArray, all corresponding to the same polarization

update_noise(*stats_calc_num_samples=10000*)

Replace self.noise_std by calculating out a few samples and estimating the standard deviation of the voltages. Further, set all child antenna background noise values.

Parameters **stats_calc_num_samples** (*int, optional*) – Maximum number of samples for use in estimating noise standard deviation

add_noise(*v_mean*, *v_std*)

Add Gaussian noise source to data stream. This essentially adds a lambda function that gets the appropriate number of noise samples to add to the voltage array when `get_samples()` is called. Updates noise property to reflect added noise. Further, set all child antenna background noise values.

Parameters

- **v_mean** (*float*) – Noise mean
- **v_std** (*float*) – Noise standard deviation

setigen.voltage.data_stream.**estimate_stats**(*voltages*, *stats_calc_num_samples=10000*)

Estimate mean and standard deviation, truncating to at most *stats_calc_num_samples* samples to reduce computation.

Parameters

- **voltages** (*array*) – Array of voltages

- **stats_calc_num_samples** (*int, optional*) – Maximum number of samples for use in estimating noise statistics

Returns

- **data_mean** (*float*) – Mean of voltages
- **data_sigma** (*float*) – Standard deviation of voltages

1.7.4 setigen.voltage.polyphase_filterbank module

class setigen.voltage.polyphase_filterbank.**PolyphaseFilterbank**(*num_taps=8*, *num_branches=1024*, *window_fn='hamming'*)

Bases: object

Implement a polyphase filterbank (PFB) for coarse channelization of real voltage input data.

Follows description in Danny C. Price, Spectrometers and Polyphase Filterbanks in Radio Astronomy, 2016. Available online at: http://arxiv.org/abs/1607.03579.

__init__(*num_taps=8*, *num_branches=1024*, *window_fn='hamming'*)

Initialize a polyphase filterbank object, with a voltage sample cache that ensures that consecutive sample retrievals get contiguous data (i.e. without introduced time delays).

Parameters

- **num_taps** (*int, optional*) – Number of PFB taps
- **num_branches** (*int, optional*) – Number of PFB branches. Note that this results in *num_branches / 2* coarse channels.
- **window_fn** (*str, optional*) – Windowing function used for the PFB

channelize(*x*, *use_cache=True*)

Channelize input voltages by applying the PFB and taking a normalized FFT.

Parameters **x** (*array*) – Array of voltages

Returns **X_pfb** – Post-FFT complex voltages

Return type array

setigen.voltage.polyphase_filterbank.**pfb_frontend**(*x*, *pfb_window*, *num_taps*, *num_branches*)

Apply windowing function to create polyphase filterbank frontend.

Follows description in Danny C. Price, Spectrometers and Polyphase Filterbanks in Radio Astronomy, 2016. Available online at: http://arxiv.org/abs/1607.03579.

Parameters

- **x** (*array*) – Array of voltages
- **pfb_window** (*array*) – Array of PFB windowing coefficients
- **num_taps** (*int*) – Number of PFB taps
- **num_branches** (*int*) – Number of PFB branches. Note that this results in *num_branches / 2* coarse channels.

Returns **x_summed** – Array of voltages post-PFB weighting

Return type array

setigen.voltage.polyphase_filterbank.**get_pfb_window**(*num_taps*, *num_branches*, *window_fn='hamming'*)

Get windowing function to multiply to time series data according to a finite impulse response (FIR) filter.

Parameters

- **num_taps** (*int*) – Number of PFB taps
- **num_branches** (*int*) – Number of PFB branches. Note that this results in *num_branches / 2* coarse channels.
- **window_fn** (*str, optional*) – Windowing function used for the PFB

Returns **window** – Array of PFB windowing coefficients

Return type array

setigen.voltage.polyphase_filterbank.**get_pfb_voltages**(*x*, *num_taps*, *num_branches*, *window_fn='hamming'*)

Produce complex raw voltage data as a function of time and coarse channel.

Parameters

- **x** (*array*) – Array of voltages
- **num_taps** (*int*) – Number of PFB taps
- **num_branches** (*int*) – Number of PFB branches. Note that this results in *num_branches / 2* coarse channels.
- **window_fn** (*str, optional*) – Windowing function used for the PFB

Returns **X_pfb** – Post-FFT complex voltages

Return type array

1.7.5 setigen.voltage.quantization module

class setigen.voltage.quantization.**RealQuantizer**(*target_mean=0*, *target_fwhm=32*, *num_bits=8*, *stats_calc_period=1*, *stats_calc_num_samples=10000*)

Bases: object

Implement a quantizer for input voltages.

__init__(*target_mean=0*, *target_fwhm=32*, *num_bits=8*, *stats_calc_period=1*, *stats_calc_num_samples=10000*)

Initialize a quantizer, which maps real input voltages to integers between -2**(num_bits - 1) and 2**(num_bits - 1) - 1, inclusive. Specifically, it estimates the mean and standard deviation of the voltages, and maps to 0 mean and a target full width at half maximum (FWHM). Voltages that extend past the quantized voltage range are clipped accordingly.

The mean and standard deviation calculations can be limited to save computation using the *stats_calc_period* and *stats_calc_num_samples* parameters. The former is an integer that specifies the period of computation; if 1, it computes the stats every time. If set to a non-positive integer, like -1, the computation will run once during the first call and never again. The latter specifies the maximum number of voltage samples to use in calculating the statistics; depending on the nature of the input voltages, a relatively small number of samples may be sufficient for capturing the general distribution of voltages.

Parameters

- **target_fwhm** (*float, optional*) – Target FWHM

- **num_bits** (*int, optional*) – Number of bits to quantize to. Quantized voltages will span -2**(num_bits - 1) to 2**(num_bits - 1) - 1, inclusive.
- **stats_calc_period** (*int, optional*) – Sets the period for computing the mean and standard deviation of input voltages
- **stats_calc_num_samples** (*int, optional*) – Maximum number of samples for use in estimating noise statistics

quantize(*voltages*, *custom_std=None*)

Quantize input voltages. Cache voltage mean and standard deviation, per polarization and per antenna.

Parameters

- **voltages** (*array*) – Array of real voltages
- **custom_std** (*float*) – Custom standard deviation to use for scaling, instead of automatic calculation. The quantizer will go from custom_std to self.target_std.

Returns **q_voltages** – Array of quantized voltages

Return type array

digitize(*voltages*, *custom_std=None*)

Quantize input voltages. Wrapper for `quantize()`.

class setigen.voltage.quantization.**ComplexQuantizer**(*target_mean=0*, *target_fwhm=32*, *num_bits=8*, *stats_calc_period=1*, *stats_calc_num_samples=10000*)

Bases: `object`

Implement a quantizer for complex voltages, using a pair of RealQuantizers.

__init__(*target_mean=0*, *target_fwhm=32*, *num_bits=8*, *stats_calc_period=1*, *stats_calc_num_samples=10000*)

Initialize a complex quantizer, which maps complex input voltage components to integers between -2**(num_bits - 1) and 2**(num_bits - 1) - 1, inclusive. Uses a pair of RealQuantizers to quantize real and imaginary components separately.

Parameters

- **target_fwhm** (*float, optional*) – Target FWHM
- **num_bits** (*int, optional*) – Number of bits to quantize to. Quantized voltages will span -2**(num_bits - 1) to 2**(num_bits - 1) - 1, inclusive.
- **stats_calc_period** (*int, optional*) – Sets the period for computing the mean and standard deviation of input voltages
- **stats_calc_num_samples** (*int, optional*) – Maximum number of samples for use in estimating noise statistics

quantize(*voltages*, *custom_stds=None*)

Quantize input complex voltages. Cache voltage means and standard deviations, per polarization and per antenna.

Parameters

- **voltages** (*array*) – Array of complex voltages
- **custom_stds** (*float, list, or array*) – Custom standard deviation to use for scaling, instead of automatic calculation. Each quantizer will go from custom_stds values to self.target_std. Can either be a single value or an array-like object of length 2, to set the custom standard deviation for real and imaginary parts.

Returns **q_voltages** – Array of complex quantized voltages

Return type array

setigen.voltage.quantization.**quantize_real**(*x*, *target_mean=0*, *target_std=13.589148804608305*, *num_bits=8*, *data_mean=None*, *data_std=None*, *stats_calc_num_samples=10000*)

Quantize real voltage data to integers with specified number of bits and target statistics.

Parameters

- **x** (*array*) – Array of voltages
- **target_mean** (*float, optional*) – Target mean for voltages
- **target_std** (*float, optional*) – Target standard deviation for voltages
- **num_bits** (*int, optional*) – Number of bits to quantize to. Quantized voltages will span -2**(num_bits - 1) to 2**(num_bits - 1) - 1, inclusive.
- **data_mean** (*float, optional*) – Mean of input voltages, if already known
- **data_std** (*float, optional*) – Standard deviation of input voltages, if already known. If None, estimates mean and standard deviation automatically.
- **stats_calc_num_samples** (*int, optional*) – Maximum number of samples for use in estimating noise statistics

Returns **q_voltages** – Array of quantized voltages

Return type array

setigen.voltage.quantization.**quantize_complex**(*x*, *target_mean=0*, *target_std=13.589148804608305*, *num_bits=8*, *stats_calc_num_samples=10000*)

Quantize complex voltage data to integers with specified number of bits and target FWHM range.

Parameters

- **x** (*array*) – Array of complex voltages
- **target_mean** (*float, optional*) – Target mean for voltages
- **target_std** (*float, optional*) – Target standard deviation for voltages
- **num_bits** (*int, optional*) – Number of bits to quantize to. Quantized voltages will span -2**(num_bits - 1) to 2**(num_bits - 1) - 1, inclusive.
- **stats_calc_num_samples** (*int, optional*) – Maximum number of samples for use in estimating noise statistics

Returns **q_c** – Array of complex quantized voltages

Return type array

1.7.6 setigen.voltage.level_utils module

setigen.voltage.level_utils.**get_unit_drift_rate**(*raw_voltage_backend*, *fftlength*, *int_factor=1*)

Calculate drift rate corresponding to a 1x1 pixel shift in the final data product. This is equivalent to dividing the fine channelized frequency resolution with the time resolution.

Parameters

- **raw_voltage_backend** (RawVoltageBackend) – Backend object to infer observation parameters
- **fftlength** (*int*) – FFT length to be used in fine channelization
- **int_factor** (*int, optional*) – Integration factor to be used in fine channelization

Returns **unit_drift_rate** – Drift rate in Hz / s

Return type float

setigen.voltage.level_utils.**get_level**(*snr*, *raw_voltage_backend*, *fftlength*, *obs_length=None*, *num_blocks=None*, *length_mode='obs_length'*)

Calculate required signal level as a function of desired SNR, assuming initial noise variance of 1. This is calculated for a single polarization. This further assumes the signal is non-drifting and centered on a finely channelized bin.

Parameters

- **snr** (*float*) – Signal-to-noise ratio (SNR)
- **raw_voltage_backend** (RawVoltageBackend) – Backend object to infer observation parameters
- **fftlength** (*int, optional*) – FFT length to be used in fine channelization
- **obs_length** (*float, optional*) – Length of observation in seconds, if in *obs_length* mode
- **num_blocks** (*int, optional*) – Number of data blocks to record, if in *num_blocks* mode
- **length_mode** (*str, optional*) – Mode for specifying length of observation, either *obs_length* in seconds or *num_blocks* in data blocks

Returns **level** – Level, or amplitude, for a real voltage cosine signal

Return type float

setigen.voltage.level_utils.**get_leakage_factor**(*f_start*, *raw_voltage_backend*, *fftlength*)

Get factor to scale up signal amplitude from spectral leakage based on the position of a signal in a fine channel. This calculates an inverse normalized sinc value based on the position of the signal with respect to finely channelized bins. Since intensity goes as voltage squared, this gives a scaling proportional to 1/sinc^2 in finely channelized data products; this is the standard fine channel spectral response.

Parameters

- **f_start** (*float*) – Signal frequency, in Hz
- **raw_voltage_backend** (RawVoltageBackend) – Backend object to infer observation parameters
- **fftlength** (*int, optional*) – FFT length to be used in fine channelization

Returns **leakage_factor** – Factor to multiply to signal level / amplitude

Return type float

setigen.voltage.level_utils.**get_total_obs_num_samples**(*obs_length=None*, *num_blocks=None*, *length_mode='obs_length'*, *num_antennas=1*, *sample_rate=3000000000.0*, *block_size=134217728*, *num_bits=8*, *num_pols=2*, *num_branches=1024*, *num_chans=64*)

Calculate number of required real voltage time samples for as given *obs_length* or *num_blocks*, without directly using a *RawVoltageBackend* object.

Parameters

- **obs_length** (*float, optional*) – Length of observation in seconds, if in *obs_length* mode
- **num_blocks** (*int, optional*) – Number of data blocks to record, if in *num_blocks* mode
- **length_mode** (*str, optional*) – Mode for specifying length of observation, either *obs_length* in seconds or *num_blocks* in data blocks
- **num_antennas** (*int*) – Number of antennas
- **sample_rate** (*float*) – Sample rate in Hz
- **block_size** (*int*) – Block size used in recording GUPPI RAW files
- **num_bits** (*int*) – Number of bits in requantized data (for saving into file). Can be 8 or 4.
- **num_pols** (*int*) – Number of polarizations recorded
- **num_branches** (*int*) – Number of branches in polyphase filterbank
- **num_chans** (*int*) – Number of coarse channels written to file

Returns **num_samples** – Number of samples

Return type int

1.7.7 setigen.voltage.waterfall module

setigen.voltage.waterfall.**get_pfb_waterfall**(*pfb_voltages_x*, *pfb_voltages_y=None*, *fftlength=256*, *int_factor=1*)

Perform fine channelization on input complex voltages after filterbank, for single or dual polarizations.

Parameters

- **pfb_voltages_x** (*array*) – Complex voltages in first polarization, of shape (time_samples, num_chans)
- **pfb_voltages_y** (*array, optional*) – Complex voltages in second polarization, of shape (time_samples, num_chans)
- **fftlength** (*int*) – FFT length to be used in fine channelization
- **int_factor** (*int, optional*) – Integration factor to be used in fine channelization

Returns **XX_psd** – Finely channelized voltages

Return type array

setigen.voltage.waterfall.**get_waterfall_from_raw**(*raw_filename*, *block_size*, *num_chans*, *int_factor=1*, *fftlength=256*)

Produces waterfall data array from the first block of a dual-polarized, 8 bit RAW file. Lightweight function mainly for testing.

Parameters

- **raw_filename** (*str*) – Filename of GUPPI RAW file
- **block_size** (*int*) – Number of bytes in a data block
- **num_chans** (*int*) – Number of coarse channels saved in RAW file
- **fftlength** (*int*) – FFT length to be used in fine channelization
- **int_factor** (*int, optional*) – Integration factor to be used in fine channelization

Returns **XX_psd** – Finely channelized voltages

Return type array

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

s

Index

Symbols

A

B

C

D

E

F

G

I

L

M

N

P

Q

R

S

T

U

V

Z