setigen Documentation

Bryan Brzycki

Mar 08, 2020

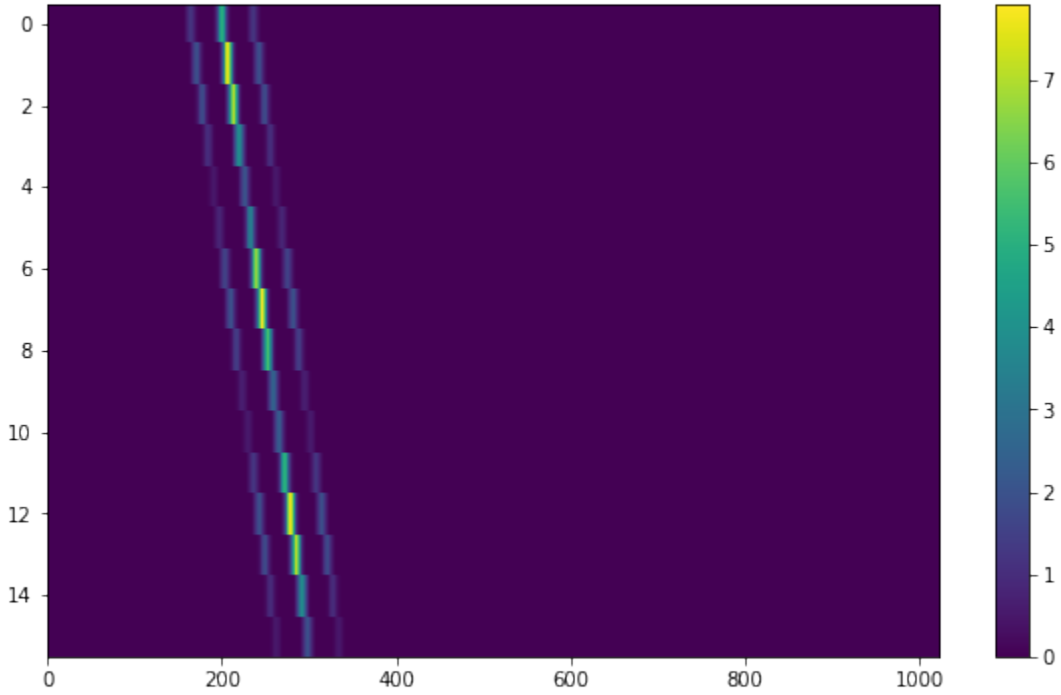
Contents

1 Table of Contents 3

2 Indices and tables 21

Python Module Index 23

Index 25



setigen is a Python library for generating and injecting artificial narrow-band signals into time-frequency data. *setigen* interfaces primarily with data saved in two-dimensional NumPy arrays or filterbank files (`.fil` extension).

setigen allows the user to generate synthetic signals in the time-frequency domain. Furthermore, the user may inject these synthetic signals into real observational data using tools that rely on the `blimpy` package (maintained by Breakthrough Listen based at UC Berkeley).

CHAPTER 1

Table of Contents

1.1 Installation

At the command line, execute:

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

Or, you can use pip to install the package automatically:

```
pip install setigen
```

1.2 Cookbook

1.2.1 Generating synthetic signals

Generating a basic signal

The main method that generates signals is *generate()*. We need to pass in an array of times, frequencies, and functions 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.

```
import setigen as stg
import numpy as np

# Define time and frequency arrays, essentially labels for the 2D data array
tsamp = 18.25361108
fch1 = 6095.214842353016
```

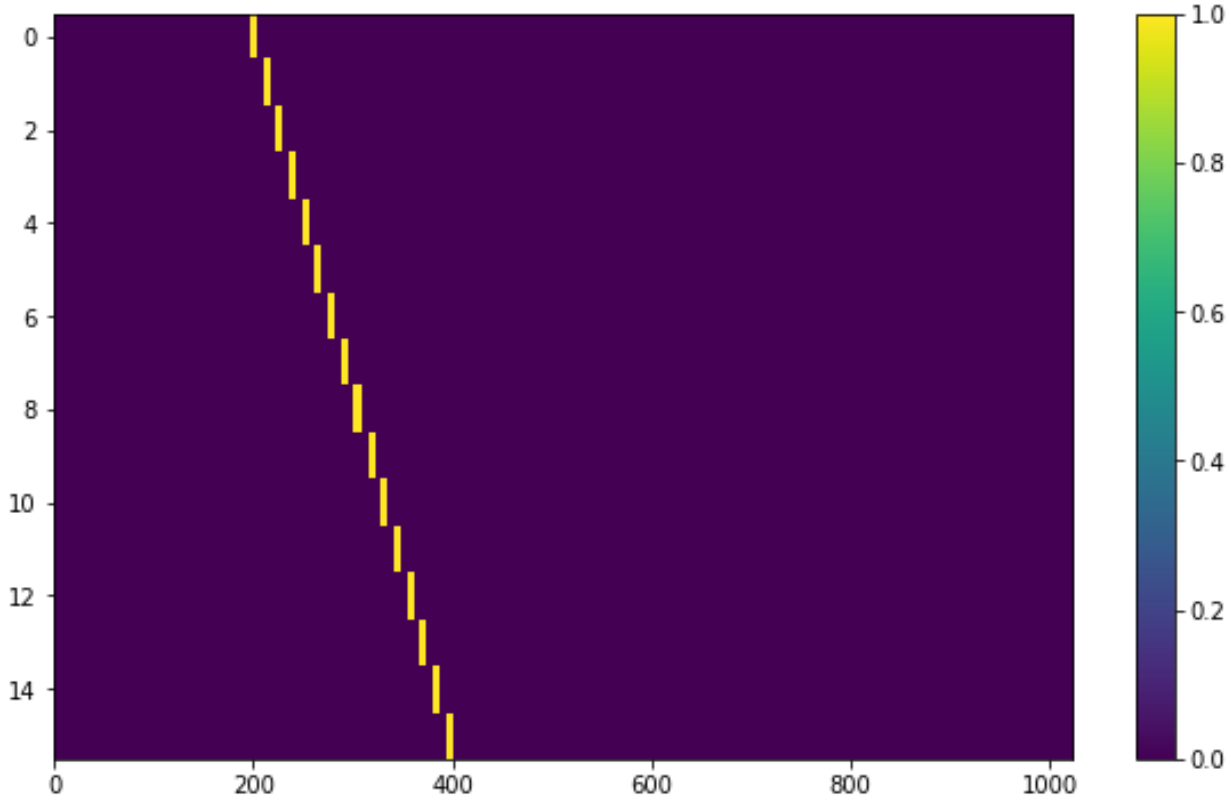
(continues on next page)

(continued from previous page)

```
df = -2.7939677238464355e-06
fchans = 1024
tchans = 16
fs = np.arange(fch1, fch1 + fchans * df, df)
ts = np.arange(0, tchans * tsamp, tsamp)

# Generate the signal
signal = stg.generate(ts,
                      fs,
                      stg.constant_path(f_start = fs[200], drift_rate = -0.000002),
                      stg.constant_t_profile(level = 1),
                      stg.box_f_profile(width = 0.00001),
                      stg.constant_bp_profile(level = 1))
```

`signal` is a 2D NumPy array with the resulting time-frequency data. To visualize this, we use `matplotlib.pyplot.imshow()`:

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



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 mixing and matching these parameters!

Here are some examples of pre-written signal functions. To avoid needless repetition, each example script will assume the same basic setup:

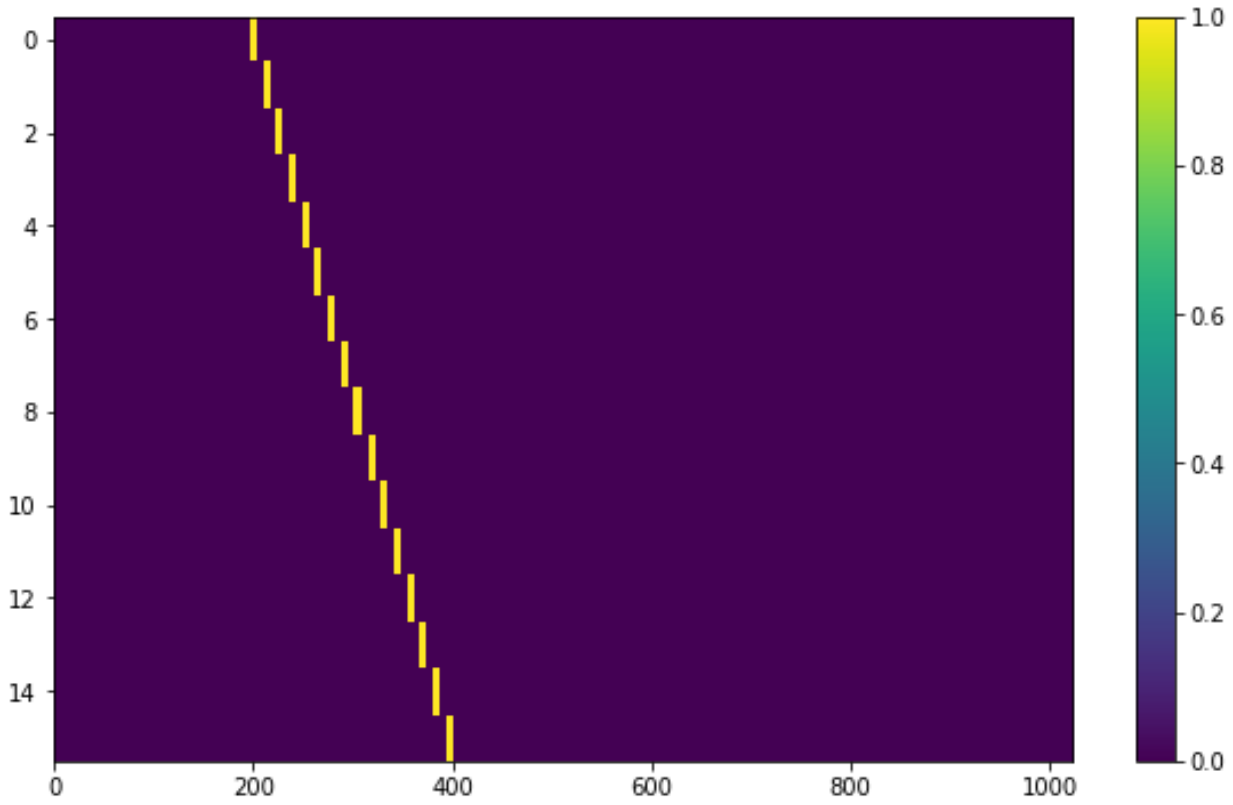
```
import setigen as stg
import numpy as np
import matplotlib.pyplot as plt

# Define time and frequency arrays, essentially labels for the 2D data array
tsamp = 18.25361108
fch1 = 6095.214842353016
df = -2.7939677238464355e-06
fchans = 1024
tchans = 16
fs = np.arange(fch1, fch1 + fchans * df, df)
ts = np.arange(0, tchans * tsamp, tsamp)
```

paths

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):

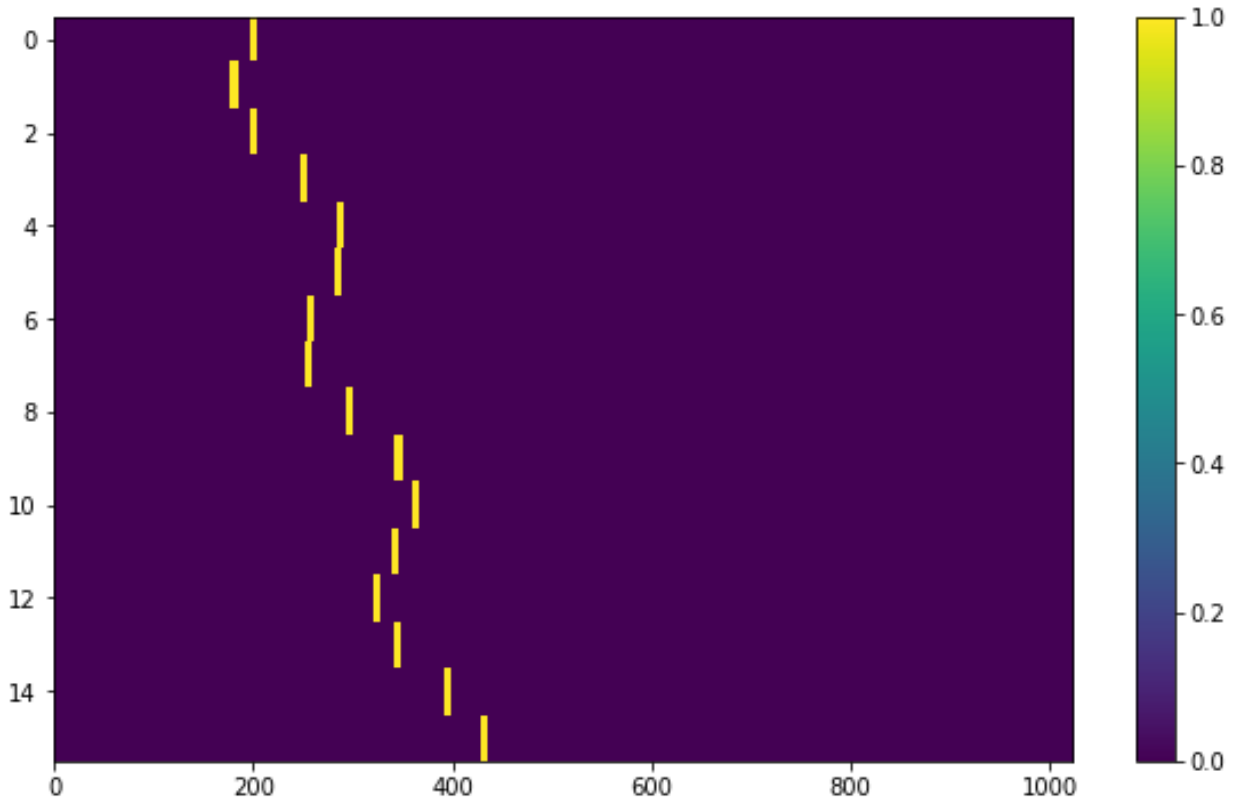
```
signal = stg.generate(ts,
                      fs,
                      stg.constant_path(f_start = fs[200], drift_rate = -0.000002),
                      stg.constant_t_profile(level = 1),
                      stg.box_f_profile(width = 0.00001),
                      stg.constant_bp_profile(level = 1))

fig = plt.figure(figsize=(10,6))
plt.imshow(signal, aspect='auto')
plt.colorbar()
fig.savefig("basic_signal.png", bbox_inches='tight')
```



Sine path

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

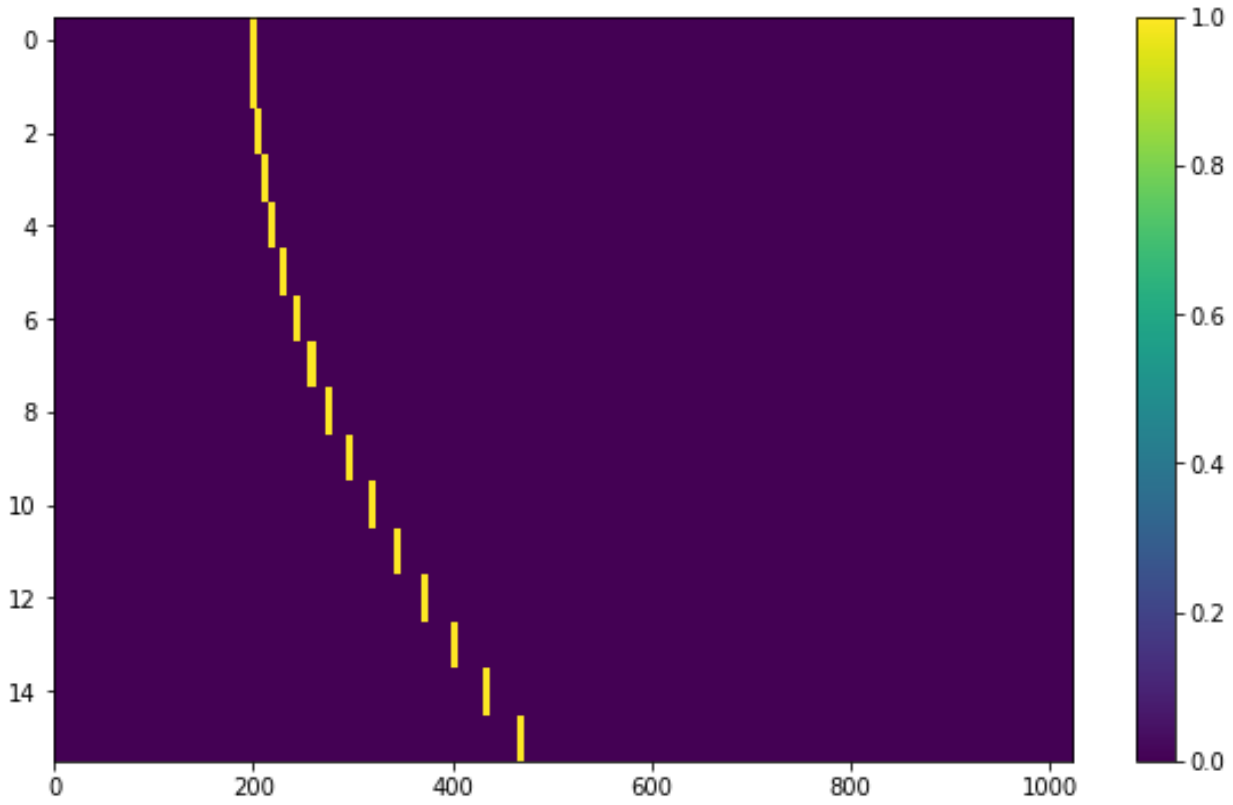
```
signal = stg.generate(ts,
                      fs,
                      stg.sine_path(f_start = fs[200], drift_rate = -0.000002,
                                    period = 100, amplitude = 0.0001),
                      stg.constant_t_profile(level = 1),
                      stg.box_f_profile(width = 0.00001),
                      stg.constant_bp_profile(level = 1))

fig = plt.figure(figsize=(10,6))
plt.imshow(signal, aspect='auto')
plt.colorbar()
fig.savefig("sine_signal.png", bbox_inches='tight')
```



Squared path

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

```
signal = stg.generate(ts,
                      fs,
                      stg.squared_path(f_start = fs[200],
                                       drift_rate = -0.00000001),
                      stg.constant_t_profile(level = 1),
                      stg.box_f_profile(width = 0.00001),
                      stg.constant_bp_profile(level = 1))

fig = plt.figure(figsize=(10,6))
plt.imshow(signal, aspect='auto')
plt.colorbar()
fig.savefig("squared_signal.png", bbox_inches='tight')
```

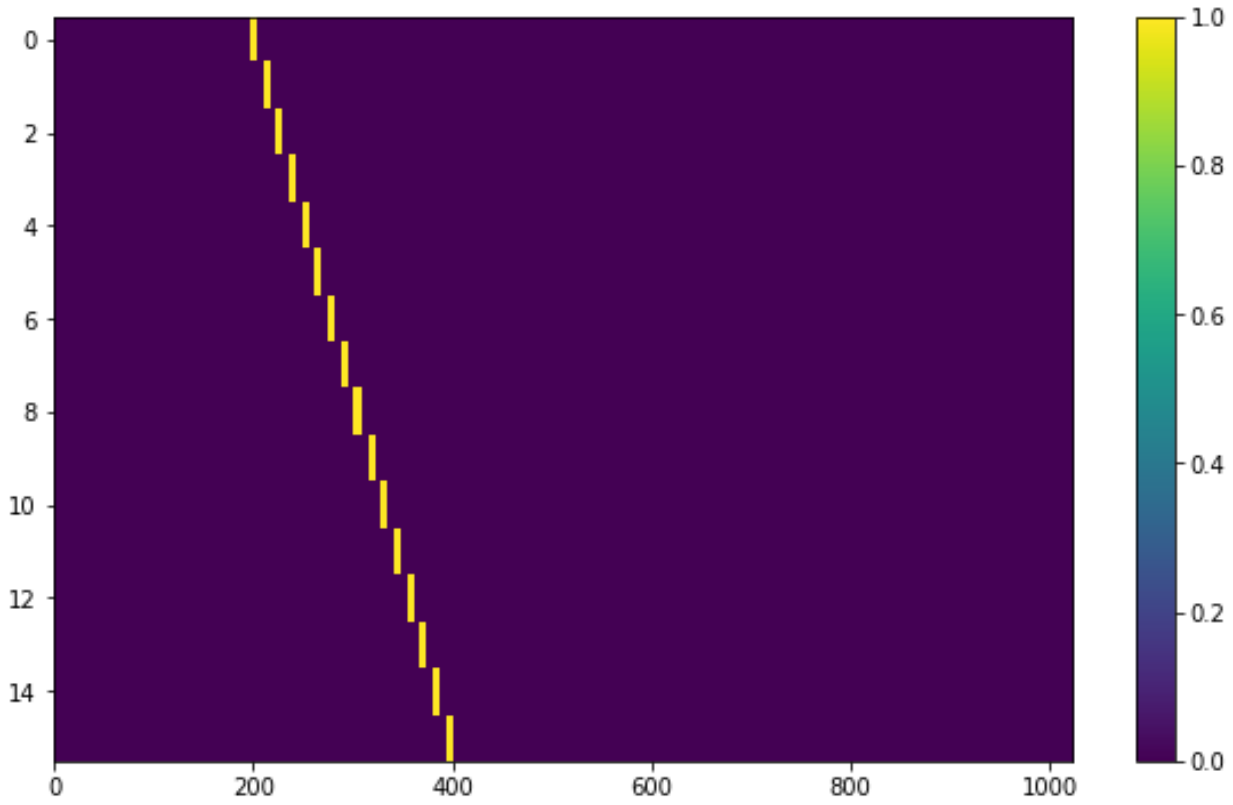


t_profiles

Constant intensity

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

```
signal = stg.generate(ts,
                      fs,
                      stg.constant_path(f_start = fs[200], drift_rate = -0.000002),
                      stg.constant_t_profile(level = 1),
                      stg.box_f_profile(width = 0.00001),
                      stg.constant_bp_profile(level = 1))

fig = plt.figure(figsize=(10,6))
plt.imshow(signal, aspect='auto')
plt.colorbar()
fig.savefig("basic_signal.png", bbox_inches='tight')
```

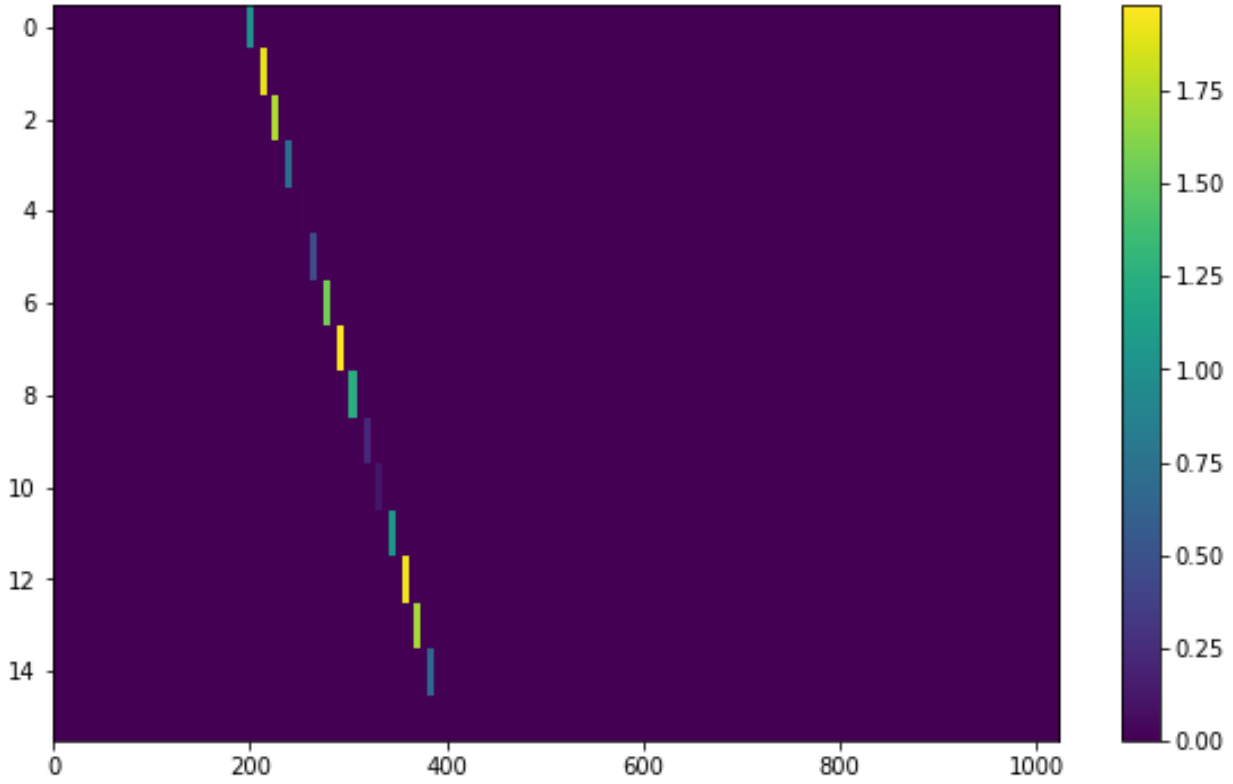


Sine intensity

To generate a signal with sinuisoidal 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.

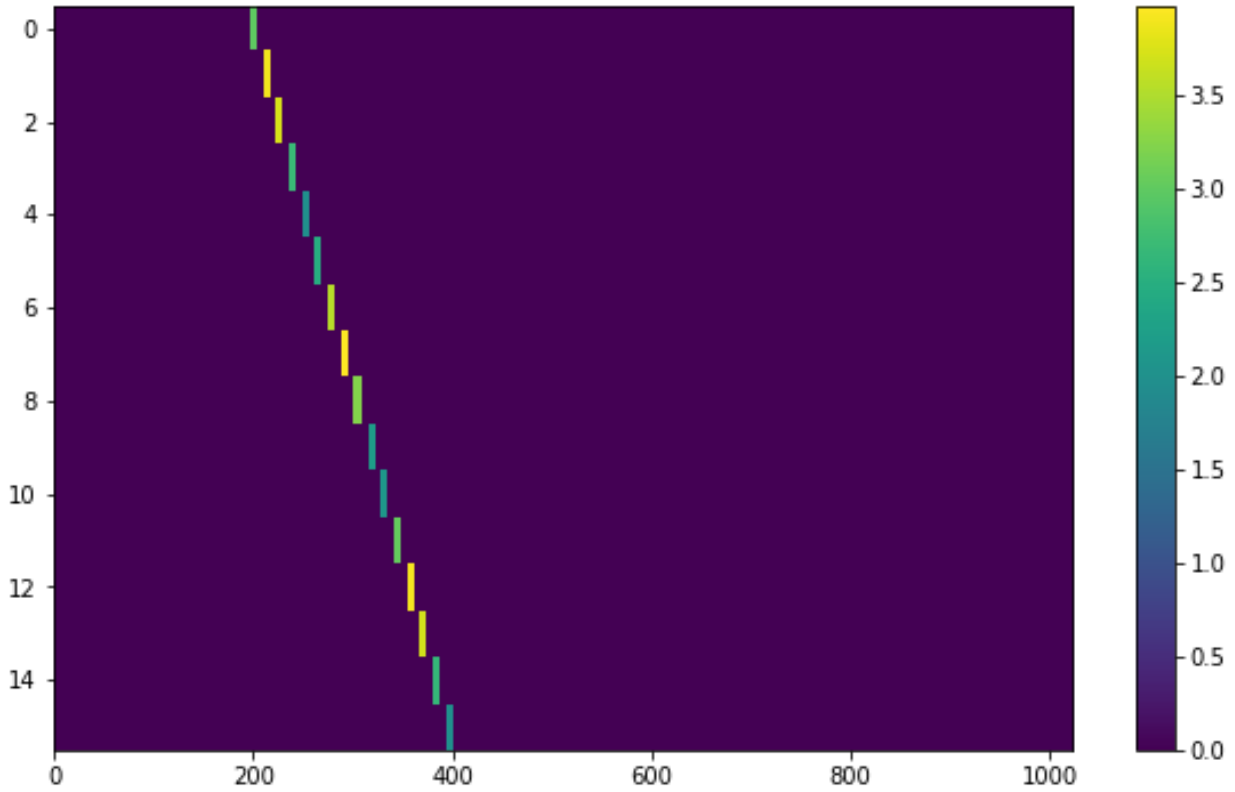
Here's an example with equal level and amplitude:

```
signal = stg.generate(ts,
                      fs,
                      stg.constant_path(f_start = fs[200], drift_rate = -0.000002),
                      stg.sine_t_profile(period = 100, amplitude = 1, level = 1),
                      stg.box_f_profile(width = 0.00001),
                      stg.constant_bp_profile(level = 1))

fig = plt.figure(figsize=(10,6))
plt.imshow(signal, aspect='auto')
plt.colorbar()
fig.savefig("sine_intensity_1_1.png", bbox_inches='tight')
```



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

```
signal = stg.generate(ts,
                      fs,
                      stg.constant_path(f_start = fs[200], drift_rate = -0.000002),
                      stg.sine_t_profile(period = 100, amplitude = 1, level = 3),
                      stg.box_f_profile(width = 0.00001),
                      stg.constant_bp_profile(level = 1))

fig = plt.figure(figsize=(10,6))
plt.imshow(signal, aspect='auto')
plt.colorbar()
fig.savefig("sine_intensity_1_3.png", bbox_inches='tight')
```

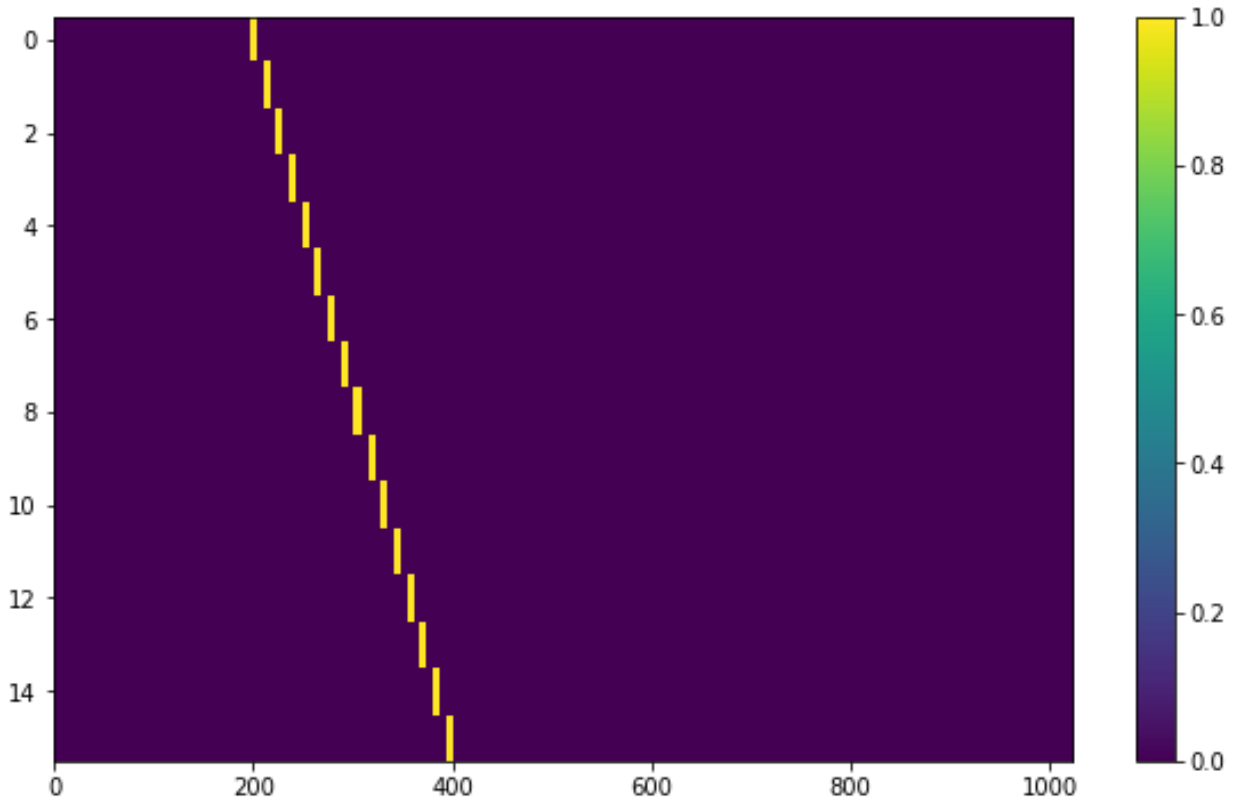


f_profiles

Box / square intensity profile

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

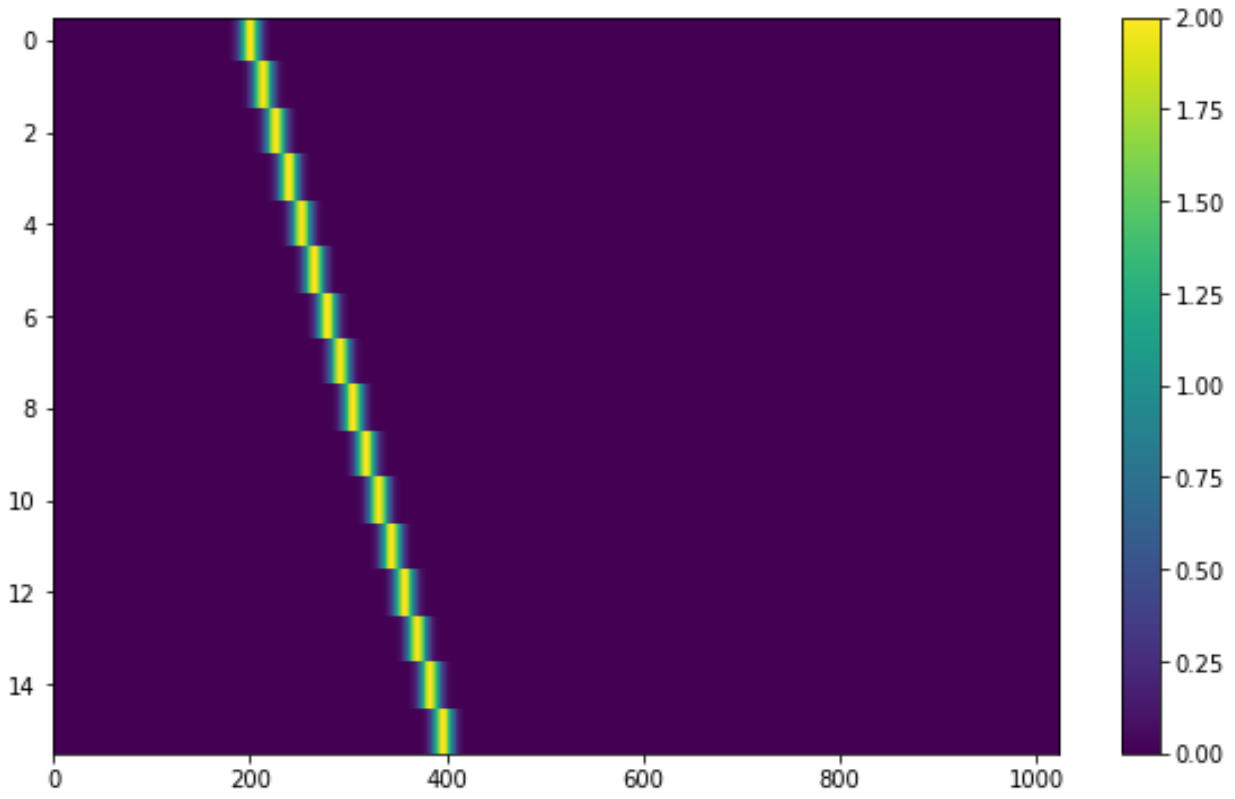
```
signal = stg.generate(ts,
                      fs,
                      stg.constant_path(f_start = fs[200], drift_rate = -0.000002),
                      stg.constant_t_profile(level = 1),
                      stg.box_f_profile(width = 0.00001),
                      stg.constant_bp_profile(level = 1))

fig = plt.figure(figsize=(10,6))
plt.imshow(signal, aspect='auto')
plt.colorbar()
fig.savefig("basic_signal.png", bbox_inches='tight')
```



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:

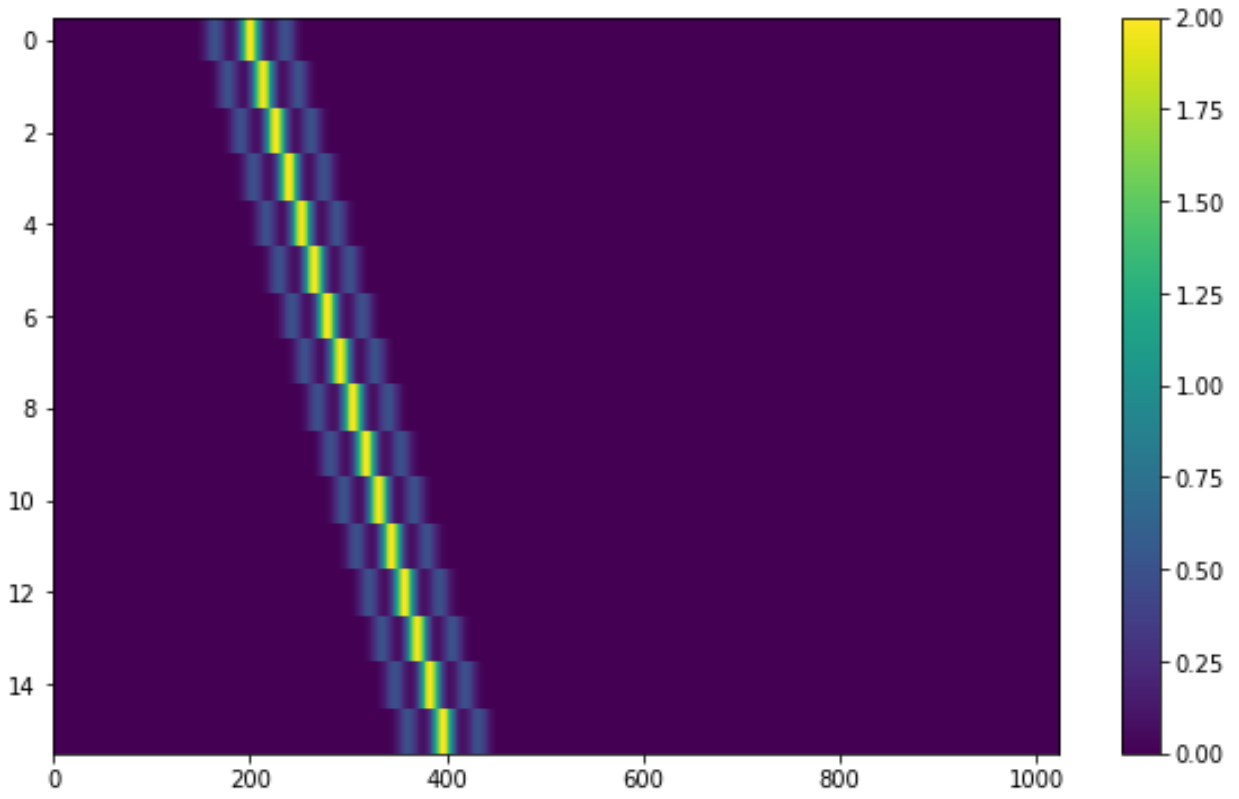
```
signal = stg.generate(ts,
                  fs,
                  stg.constant_path(f_start = fs[200], drift_rate = -0.000002),
                  stg.constant_t_profile(level = 1),
                  stg.gaussian_f_profile(width = 0.00002),
                  stg.constant_bp_profile(level = 2))


fig = plt.figure(figsize=(10,6))
plt.imshow(signal, aspect='auto')
plt.colorbar()
fig.savefig("gaussian_profile.png", bbox_inches='tight')
```



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.

```
signal = stg.generate(ts,
                  fs,
                  stg.constant_path(f_start = fs[200], drift_rate = -0.000002),
                  stg.constant_t_profile(level = 1),
                  stg.multiple_gaussian_f_profile(width = 0.00002),
                  stg.constant_bp_profile(level = 2))


fig = plt.figure(figsize=(10,6))
plt.imshow(signal, aspect='auto')
plt.colorbar()
fig.savefig("multiple_gaussian_profile.png", bbox_inches='tight')
```



Writing custom signal functions

You can easily go beyond *setigen*'s pre-written signal functions by writing your own. For each *generate()* parameter (`path`, `t_profile`, `f_profile`, and `bp_profile`), you can pass in your own custom functions.

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

```
def 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 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
```

It's important that the function you pass into each parameter has the correct input and output. Specifically:

`path` Takes in time `t` and outputs a frequency

`t_profile` Takes in time `t` and outputs an intensity

`f_profile` Takes in frequency `f` and a reference central frequency `f_center`, and outputs an intensity

`bp_profile` Takes in frequency `f` and outputs an intensity

To generate synthetic signals, *generate()* uses these functions to compute intensity for each time, frequency pair in the data.

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

1.2.2 Working with filterbank files

To work with time-frequency data, we use filterbank files (as used in `sigproc` and `blimpy`).

Accessing data

To visualize filterbank data, we use `blimpy.Waterfall`.

Splitting filterbank files

We can split a single filterbank file into a series of smaller filterbank files for different applications - searching through a more manageable frame size, machine learning with more dataframes, etc.

1.3 setigen package

1.3.1 Subpackages

setigen.funcs package

Submodules

setigen.funcs.bp_profiles module

setigen.funcs.bp_profiles.**constant_bp_profile**(*level=1*)

setigen.funcs.f_profiles module

setigen.funcs.f_profiles.**box_f_profile**(*width=1e-05*)

setigen.funcs.f_profiles.**gaussian_f_profile**(*width=1e-05*)

setigen.funcs.f_profiles.**multiple_gaussian_f_profile**(*width=1e-05*)

setigen.funcs.paths module

setigen.funcs.paths.**choppy_rfi_path**(*f_start*, *drift_rate*, *spread*, *spread_type='uniform'*)

setigen.funcs.paths.**constant_path**(*f_start*, *drift_rate*)

setigen.funcs.paths.**sine_path**(*f_start*, *drift_rate*, *period*, *amplitude*)

setigen.funcs.paths.**squared_path**(*f_start*, *drift_rate*)

setigen.funcs.t_profiles module

setigen.funcs.t_profiles.**constant_t_profile**(*level=1*)

setigen.funcs.t_profiles.**periodic_gaussian_t_profile**(*period*, *phase*, *sigma*, *pulse_dir*, *width*, *pnum=1*, *amplitude=1*, *level=0*)

setigen.funcs.t_profiles.**sine_t_profile**(*period*, *phase=0*, *amplitude=1*, *level=1*)

Module contents

1.3.2 Submodules

1.3.3 setigen.fil_utils module

setigen.fil_utils.**get_data**(*input*)
Gets time-frequency data from filterbank file as a 2d NumPy array.

Parameters **input** (*str*) – Name of filterbank file

Returns **data** – Time-frequency data

Return type ndarray

setigen.fil_utils.**get_fs**(*input*)
Gets frequency values from filterbank file.

Parameters **input** (*str*) – Name of filterbank file

Returns **fs** – Frequency values

Return type ndarray

setigen.fil_utils.**get_ts**(*input*)
Gets time values from filterbank file.

Parameters **input** (*str*) – Name of filterbank file

Returns **ts** – Time values

Return type ndarray

setigen.fil_utils.**maxfreq**(*input*)
Return central frequency of the highest-frequency bin in a .fil file.

setigen.fil_utils.**minfreq**(*input*)
Return central frequency of the lowest-frequency bin in a .fil file.

1.3.4 setigen.generate_signal module

setigen.generate_signal.**generate**(*ts*, *fs*, *path*, *t_profile*, *f_profile*, *bp_profile*, *integrate=False*, *samples=10*)
Generates synthetic signal.

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

Parameters

- **ts** (*ndarray*) – Time samples

- **fs** (*ndarray*) – Frequency samples
- **path** (*function*) – Function in time that returns frequencies
- **t_profile** (*function*) – Time profile: function in time that returns an intensity (scalar)
- **f_profile** (*function*) – Frequency profile: function in frequency that returns an intensity (scalar), relative to the signal frequency within a time sample
- **bp_profile** (*function*) – Bandpass profile: function in frequency that returns an intensity (scalar)
- **integrate** (*bool, optional*) – Option to integrate t_profile in the time direction
- **samples** (*int, optional*) – Number of bins to integrate t_profile in the time direction, using Riemann sums

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

Return type ndarray

Examples

A simple example that creates a linear Doppler-drifted signal:

```
>>> import setigen as stg
>>> import numpy as np
>>> tsamp = 18.25361108
>>> fch1 = 6095.214842353016
>>> df = -2.7939677238464355e-06
>>> fchans = 1024
>>> tchans = 16
>>> fs = np.arange(fch1, fch1 + fchans * df, df)
>>> ts = np.arange(0, tchans * tsamp, tsamp)
>>> signal = stg.generate(ts,
                          fs,
                          stg.constant_path(f_start = fs[200], drift_rate = -0.
↪000002),
                          stg.constant_t_profile(level = 1),
                          stg.box_f_profile(width = 0.00001),
                          stg.constant_bp_profile(level = 1))
```

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))
>>> plt.imshow(signal, aspect='auto')
>>> plt.colorbar()
>>> fig.savefig("image.png", bbox_inches='tight')
```

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

1.3.5 setigen.split_utils module

setigen.split_utils.**split_data**(*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

setigen.split_utils.**split_fil**(*input_fn*, *output_dir*, *f_sample_num*, *f_shift=None*)

Creates a set of new filterbank files by 'splitting' an input filterbank file according to the number of frequency samples.

Parameters

- **input_fn** (*str*) – Filterbank filename with .fil extension
- **output_dir** (*str*) – Directory for new filterbank files
- **f_sample_num** (*int*) – Number of frequency samples per new filterbank file
- **f_shift** (*int, optional*) – Number of samples to shift when splitting filterbank. If None, defaults to *f_shift=f_sample_num* so that there is no overlap between new filterbank files

Returns **split_fns** – List of new filenames

Return type list of str

1.3.6 setigen.time_freq_utils module

setigen.time_freq_utils.**db**(*x*)

Convert to dB

setigen.time_freq_utils.**gaussian_noise**(*data*, *mean*, *sigma*)

Create an array of Gaussian noise with the same dimensions as an input data array

setigen.time_freq_utils.**inject_noise**(*data*, *modulate_signal=False*, *modulate_width=0.1*, *background_noise=True*, *noise_sigma=1*)

Normalize data per frequency channel so that the noise level in data is controlled.

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
- **modulate_signal** (*bool, optional*) – Modulate signal itself with Gaussian noise (multiplicative)
- **modulate_width** (*float, optional*) – Standard deviation of signal modulation
- **background_noise** (*bool, optional*) – Add gaussian noise to entire image (additive)
- **noise_sigma** (*float, optional*) – Standard deviation of background Gaussian noise

Returns **noisy_data** – Data with injected noise

Return type ndarray

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

1.3.7 Module contents

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

setigen, 19
setigen.fil_utils, 16
setigen.funcs, 16
setigen.funcs.bp_profiles, 15
setigen.funcs.f_profiles, 15
setigen.funcs.paths, 15
setigen.funcs.t_profiles, 16
setigen.generate_signal, 16
setigen.split_utils, 17
setigen.time_freq_utils, 18

Index

B

box_f_profile() (*in module setigen.funcs.f_profiles*), 15

C

choppy_rfi_path() (*in module setigen.funcs.paths*), 15
constant_bp_profile() (*in module setigen.funcs.bp_profiles*), 15
constant_path() (*in module setigen.funcs.paths*), 15
constant_t_profile() (*in module setigen.funcs.t_profiles*), 16

D

db() (*in module setigen.time_freq_utils*), 18

G

gaussian_f_profile() (*in module setigen.funcs.f_profiles*), 15
gaussian_noise() (*in module setigen.time_freq_utils*), 18
generate() (*in module setigen.generate_signal*), 16
get_data() (*in module setigen.fil_utils*), 16
get_fs() (*in module setigen.fil_utils*), 16
get_ts() (*in module setigen.fil_utils*), 16

I

inject_noise() (*in module setigen.time_freq_utils*), 18

M

maxfreq() (*in module setigen.fil_utils*), 16
minfreq() (*in module setigen.fil_utils*), 16
multiple_gaussian_f_profile() (*in module setigen.funcs.f_profiles*), 15

N

normalize() (*in module setigen.time_freq_utils*), 18
normalize_by_max() (*in module setigen.time_freq_utils*), 19

P

periodic_gaussian_t_profile() (*in module setigen.funcs.t_profiles*), 16

S

setigen (*module*), 19
setigen.fil_utils (*module*), 16
setigen.funcs (*module*), 16
setigen.funcs.bp_profiles (*module*), 15
setigen.funcs.f_profiles (*module*), 15
setigen.funcs.paths (*module*), 15
setigen.funcs.t_profiles (*module*), 16
setigen.generate_signal (*module*), 16
setigen.split_utils (*module*), 17
setigen.time_freq_utils (*module*), 18
sine_path() (*in module setigen.funcs.paths*), 15
sine_t_profile() (*in module setigen.funcs.t_profiles*), 16
split_data() (*in module setigen.split_utils*), 17
split_fil() (*in module setigen.split_utils*), 18
squared_path() (*in module setigen.funcs.paths*), 15