The Astronomical Journal, 163:222 (11pp), 2022 May https://doi.org/10.3847/1538-3881/ac5e3d

Setigen: Simulating Radio Technosignatures for the Search for Extraterrestrial Intelligence

Bryan Brzycki[1], Andrew P. V. Siemion[2,3,4,5], Imke de Pater[1], Steve Croft[1,3], John Hoang[1], Cherry Ng[1,3,6], Danny C. Price[1,7], Sofia Sheikh[1], and Zihe Zheng[8]
[1] Department of Astronomy, University of California Berkeley, Berkeley, CA 94720, USA; bbrzycki@berkeley.edu
[2] Breakthrough Listen, University of California Berkeley, Berkeley, CA 94720, USA
[3] SETI Institute, Mountain View, CA 94043, USA
[4] Department of Physics and Astronomy, University of Manchester, Manchester, UK
[5] University of Malta, Institute of Space Sciences and Astronomy, Malta
[6] Dunlap Institute for Astronomy & Astrophysics, University of Toronto, 50 St. George Street, Toronto, ON M5S 3H4, Canada
[7] International Centre for Radio Astronomy Research, Curtin University, Bentley, WA 6102, Australia
[8] Goergen Institute for Data Science, University of Rochester, Rochester, NY 14627, USA
Received 2022 January 5; accepted 2022 March 14; published 2022 April 20

Abstract

The goal of the search for extraterrestrial intelligence (SETI) is the detection of nonhuman technosignatures, such as technology-produced emission in radio observations. While many have speculated about the character of such technosignatures, radio SETI fundamentally involves searching for signals that not only have never been detected, but also have a vast range of potential morphologies. Given that we have not yet detected a radio SETI signal, we must make assumptions about their form to develop search algorithms. The lack of positive detections also makes it difficult to test these algorithms' inherent efficacy. To address these challenges, we present `setigen`, a Python-based, open-source library for heuristic-based signal synthesis and injection for both spectrograms (dynamic spectra) and raw voltage data. `setigen` facilitates the production of synthetic radio observations, interfaces with standard data products used extensively by the Breakthrough Listen project, and focuses on providing a physically motivated synthesis framework compatible with real observational data and associated search methods. We discuss the core routines of `setigen` and present existing and future use cases in the development and evaluation of SETI search algorithms.

Unified Astronomy Thesaurus concepts: Technosignatures (2128); Search for extraterrestrial intelligence (2127); Astrobiology (74); Astronomy data modeling (1859)

1. Introduction

Since the inception of radio search for extraterrestrial intelligence (SETI) in the 1960s, technosignature searches have greatly expanded to cover more sky area, wider frequency ranges, and a larger variety of signal morphologies (Drake 1961; Werthimer et al. 1985; Tarter 2001; Siemion et al. 2013; Wright et al. 2014; MacMahon et al. 2018; Price et al. 2018; Gajjar et al. 2021). Arguably the most developed branch of radio SETI is the search for narrowband technosignatures, with signal bandwidths under 1 kHz, for which search algorithms are constantly being produced and improved (Siemion et al. 2013; Enriquez et al. 2017; Pinchuk et al. 2019; Margot et al. 2021). These algorithms operate on either voltage time series data or time-frequency spectrogram data (i.e., dynamic spectra, waterfall plots).

The incoherent tree deDoppler method is the primary search strategy for Doppler-accelerated narrowband signals in radio spectrograms (Taylor 1974; Siemion et al. 2013; Enriquez et al. 2017; Margot et al. 2021). An ideal sinusoidal emitter will appear to exhibit a frequency drift over time due to relative acceleration between the emitter and receiving telescope (Sheikh et al. 2019). Under a constant relative acceleration, such a signal will have a linear drift or slope in a spectrogram of Stokes I intensities. The tree deDoppler algorithm efficiently integrates spectra over potential drift rates and identifies signals above a threshold signal-to-noise ratio (S/N). Breakthrough Listen (BL), the most comprehensive SETI search program to date (Worden et al. 2017), developed `turboSETI`,[9] an open-source implementation of the deDoppler algorithm that serves as the backbone of many technosignature searches (Enriquez et al. 2017; Enriquez & Price 2019; Price et al. 2020; Sheikh et al. 2020; Gajjar et al. 2021).

This method works well for signals with high duty cycles and linear drift rates, but it can struggle to properly detect more complex signals (Pinchuk et al. 2019). This is particularly problematic given the increasingly complex radio frequency interference (RFI) environment within which these searches are conducted. Moreover, the lack of robust, labeled, narrowband signal data sets can make it difficult to quantify a given implementation's detection accuracy, especially in light of RFI and variable bandpass responses.

For more complex signal morphologies, machine-learning (ML) algorithms have been proposed that use computer vision techniques to classify image-like spectrograms. However, the same lack of labeled, narrowband signal data makes creating supervised ML models difficult. Zhang et al. (2018a) used a self-supervised approach in which spectrogram data were divided in time into two halves, for which the ML task was to predict the second half given the first. For an ML-based

[9] https://github.com/UCBerkeleySETI/turbo_seti

direction-of-origin filter, Pinchuk & Margot (2022) used a separate non-ML method to detect signals and create an algorithmically labeled spectrogram data set. In most cases, however, supervised approaches have relied on generating synthetic signals of various classes in order to guarantee correct labels (Harp et al. 2019; Brzycki et al. 2020; Margot et al. 2021).

To address these issues, we present `setigen`, an open-source Python library[10] that facilitates the creation of synthetic narrowband signals and supports injection into observational data (Brzycki 2022). `setigen` is meant to provide a general-use heuristic framework for creating mock radio SETI data. A primary design aspect is ensuring that the synthesis process is grounded as much as possible in physical quantities to better interface with real observations and search algorithms. `setigen` makes heavy use of `NumPy`[11] for efficient matrix operations (Oliphant 2006; Harris et al. 2020) and `blimpy`[12] for interfacing with data products routinely used by the BL project (Price et al. 2019).

There are two main modules in `setigen`, "spectrogram" and "voltage," dedicated to the most common data formats used in radio SETI. The spectrogram module works with Stokes I (intensity) data stored as time-frequency arrays and is designed to be flexible and heuristic-based. It can be used to generate many small snippets of data containing synthetic signals for quick algorithm test cases or for full labeled data sets. The voltage module creates synthetic antenna voltages, follows these voltages through a software-based signal processing chain that models a standard single-dish signal pipeline, including quantization and a PFB, and saves the final complex voltages. This requires a lot more computational power, so voltage `setigen` routines can be optionally GPU-accelerated via `CuPy`[13] (Okuta et al. 2017). As the voltage module models the signal processing chain, it can be used to produce more "realistic" signals, test complex voltage processing software, and evaluate how each signal processing element affects the final signal sensitivity.

Radio SETI searches typically operate on data in spectrogram format, as it compresses data and enables visualization and analysis of broader signal morphology in the time-frequency space (Enriquez et al. 2017; Margot et al. 2018; Pinchuk et al. 2019; Price et al. 2020; Sheikh et al. 2020). As such, `setigen` was initially written to create large data sets of radio spectrograms for use in supervised ML search experiments. The library was later expanded to support synthesizing raw voltage-level data to complement existing use cases.

`setigen` has already been used in a variety of applications, such as the development and testing of search algorithms. It has been used to create synthetic data sets with position labels for ML localization tasks in single observations (Brzycki et al. 2020). `setigen` has also been used to inject synthetic signals within ON-OFF cadences; each comprised of six consecutive observations and was used as a direction-of-origin filter for SETI. Ma et al. (submitted) injected signals into ON-OFF cadences taken with the Robert C. Byrd Green Bank Telescope (GBT; MacMahon et al. 2018) to train a sophisticated variational autoencoder model that can classify cadences as potential SETI candidates. Similarly, `setigen` was used extensively to produce training and test data in BL's first Kaggle ML competition,[14] in which contestants were tasked with classifying synthetic technosignature candidates in ON-OFF cadences.

Outside of ML, synthetic `setigen` data are used in injection-recovery testing for `turboSETI` as well as for a new search code, `hyperseti`.[15] The voltage module has been used to test and upgrade parts of the Allen Telescope Array's (Welch et al. 2009) software signal processing pipeline. Furthermore, `setigen` has been used to test RFI rejection and detection techniques for the Parkes Multibeam Galactic Plane Survey SETI search, helping to discriminate terrestrial signals from different regions in the sky as SETI surveys with multiple antennas or beams become more popular (K. Perez et al. 2022, in preparation).

This paper is organized as follows. Section 2 outlines the standard signal-chain and processing pipeline used in single-dish radio SETI observations to motivate details behind `setigen`'s synthesis methods. Section 3 presents the code methodology: Section 3.1 describes the spectrogram module for producing and working with synthetic Stokes I time-frequency data, while Section 3.2 describes the voltage synthesis module in detail, connecting components of typical radio signal chains to software analogs used in `setigen`. In Section 4, we discuss current limitations of the library and future directions for signal synthesis for SETI.

2. Overview of Single-dish Signal Chains

To motivate the capabilities of `setigen`, we first give a broad overview of the standard single-dish data recording pipeline, as well as some details pertinent to the BL digital recorder (BL DR) system at the GBT (MacMahon et al. 2018).

In a single-dish radio telescope, incoming radiation is reflected off the dish surface toward a feed horn at the focus. The feed couples incident free-space electromagnetic radiation to voltages within the telescope's receiver system.

These voltages are passed to an analog down-conversion system containing a heterodyne mixer, which shifts the signal from the target RF range into an intermediate frequency (IF) range near baseband more suitable for receiver hardware. The resulting voltages are then digitized by analog–digital converters (ADC) to a specified number of bits $N_{\mathrm{bits,d}}$ at a given sampling rate f_s. The BL DR system digitizes voltages to 8 bit at a sampling rate of $f_s = 3$ GHz for each linear polarization (MacMahon et al. 2018).

Radio telescope pipelines commonly use polyphase filterbanks (PFBs; Bellanger et al. 1976; Harris & Haines 2011; Price 2021) to help partition the usable band and improve the spectral channel response of the system. For example, the BL DR system uses an eight-tap PFB to divide the 1.5 GHz Nyquist range into $N_{\mathrm{coarse}} = 512$ "coarse" spectral channels, which in turn are divided among eight compute nodes (MacMahon et al. 2018). This procedure performs a fast Fourier transform (FFT) with a length of $P = 2N_{\mathrm{coarse}} = 1024$. For receivers with wide bandwidths, such as the C band at 3.95–8.00 GHz, multiple copies of these elements, starting from the analog mixer, are employed to cover the full band (GBT Support Staff 2022).

[10] https://github.com/bbrzycki/setigen
[11] https://numpy.org/
[12] https://github.com/UCBerkeleySETI/blimpy
[13] https://cupy.dev/
[14] https://www.kaggle.com/c/seti-breakthrough-listen
[15] https://github.com/UCBerkeleySETI/hyperseti

The digital processing components of the BL DR system are done on custom signal processing boards using field-programmable gate arrays (FPGAs), provided by the Collaboration for Astronomy Signal Processing and Electronics Research (CASPER; Hickish et al. 2016). These boards use fixed point arithmetic and increase the numerical bit size when doing computations (MacMahon et al. 2018). Accordingly, both the real and imaginary components of the resulting complex voltages must be requantized (e.g., to $N_{\rm bits,r}$) before they are written to disk. The BL DR system records these as 8 bit signed integers in Green Bank Ultimate Pulsar Processing Instrument (GUPPI; DuPlain et al. 2008) raw format, based on FITS (Pence et al. 2010) and stored as `.raw` files (Lebofsky et al. 2019).

As raw voltage data come at the highest resolution possible given the ADC sampling rate, data volumes are large, especially during standard BL observing campaigns. Therefore, we finely channelize or "reduce" raw data into spectrograms (also known as dynamic spectra or "waterfall plots"), 2D arrays of intensity (Stokes I) as a function of time and frequency (Lebofsky et al. 2019). Multiple versions with different resolutions can be created from the same set of raw data by varying the FFT length $N_{\rm fine}$ and integration factor $N_{\rm int}$.

During fine channelization, an FFT of length $N_{\rm fine}$ is performed on complex raw voltages within individual coarse channels, resulting in $N_{\rm fine}$ fine channels each. So, we can express the full Nyquist bandwidth as

$$f_N = \frac{f_s}{2} = N_{\rm coarse} N_{\rm fine} \Delta f. \tag{1}$$

This gives us an expression for the spectrogram's frequency resolution:

$$\Delta f = \frac{f_s/2}{N_{\rm coarse} N_{\rm fine}}. \tag{2}$$

If the total observation length is τ and the number of time channels (pixels) in the final spectrogram is N_t, then

$$N_t = \frac{\tau}{\Delta t}, \tag{3}$$

assuming that τ is a multiple of the spectrogram's time resolution Δt. In practice, extraneous samples are truncated when necessary to satisfy this requirement.

The integration factor $N_{\rm int}$ is the number of spectra integrated in the time direction. To get an expression for Δt, we can think in terms of the total number of samples collected (for a single linear polarization):

$$N_s = \tau f_s. \tag{4}$$

The pipeline takes in N_s real samples in time and, via a P-point FFT, transforms the data into a complex 2D array in time-frequency space, with nonintegrated dimensions $N_t N_{\rm int} \times P N_{\rm fine}$.

$$N_s = N_t N_{\rm int} P N_{\rm fine} = 2 N_t N_{\rm int} N_{\rm coarse} N_{\rm fine}. \tag{5}$$

Note that, as the FFT is performed on real voltages, the unique frequency extent is ultimately halved per the Nyquist range.

Combining Equations (2)–(5), we get

$$N_s = \tau f_s = 2 N_t N_{\rm int} N_{\rm coarse} N_{\rm fine}, \tag{6}$$

$$= 2 \frac{\tau}{\Delta t} N_{\rm int} N_{\rm coarse} N_{\rm fine}, \tag{7}$$

$$\Delta t = \frac{2 N_{\rm int} N_{\rm coarse} N_{\rm fine}}{f_s} = \frac{N_{\rm int}}{\Delta f}. \tag{8}$$

Although $N_{\rm fine}$ and $N_{\rm int}$ must both be integers, we otherwise have fine control over Δf and Δt through Equations (2) and (8).

3. Code Methodology

As object-oriented software, `setigen` has a set of important classes and routines that are described below. For more technical details and examples of the application programming interface, see the full documentation.[16]

3.1. Spectrogram Module

The spectrogram module provides an interface for synthesizing Stokes I (waterfall) data in a format common to radio SETI and is oriented around the `Frame` class. A `Frame` object contains a 2D data array of intensities as a function of time and frequency, as well as accompanying metadata, such as starting frequency and time-frequency resolutions.

Data frames can be initialized from either saved observational data or frame parameters. Frames can extract Stokes I data and observational metadata from filterbank (`.fil`) or HDF5 files (`.h5`). The most important metadata for `setigen` are the physical parameters of the underlying intensity data: resolutions and ranges in both time and frequency. Empty frames can therefore be created simply by specifying these parameters along with desired data array dimensions.

3.1.1. Noise Synthesis

In most SETI applications, we search for statistically significant signals embedded in noise. As voltage noise in the absence of RFI approximately follows a zero-mean normal distribution (Thompson et al. 2017), the radiometer noise in spectrogram data follows a chi-squared distribution (McDonough & Whalen 1995; Nita et al. 2007). When the time and frequency resolutions are coarse enough, the spectrogram noise approaches a normal distribution by the central limit theorem.

In particular, suppose we have a sequence of input voltages $\{x_n\}$ following a Gaussian distribution with zero mean. During the coarse channelization process, the PFB applies, at its core, an FFT to bring the voltages into frequency space:

$$X_k = \sum_{n=0}^{N-1} w_n x_n e^{-2\pi i k n/N},\ k = 0,\ldots,N-1, \tag{9}$$

where N is the number of frequency bins, and $\{w_n\}$ are coefficients of a windowing function applied to improve the spectral response (Price 2021).

More specifically, the filterbank sums over M rows of P samples before a P-point FFT, so that the response of the rth row of P samples is:

$$X_{k,r} = \sum_{p=0}^{P-1} \left[\sum_{m=0}^{M-1} w_{n'} x_{n''} \right] e^{-2\pi i k p/P}, \tag{10}$$

where $n' = mP + p$ and $n'' = (r - M + m)P + p$ are indices of the windowing coefficients and voltages samples in terms of m and p. Here, we assume that the MP windowing coefficients are symmetric about the midpoint, so that $w_n = w_{MP-n-1}$.

[16] https://setigen.readthedocs.io/

Ignoring quantization for the moment, we store the complex components of the resulting FFT voltages, $\mathrm{Re}(X_k)$ and $\mathrm{Im}(X_k)$, as raw voltage data. As these are linear combinations of independent zero-mean Gaussian variables (i.e., x_n), they both follow zero-mean Gaussian distributions.

In the absence of a windowing function ($w_n = 1$), for each channel besides the real-valued DC and Nyquist bins, the variances of the real and imaginary components are equal (σ^2; McDonough & Whalen 1995). When a windowing function is used, the underlying statistics can change such that the variances of the complex components differ as a function of the spectral bin (Nita et al. 2007). However, for commonly chosen symmetrical windows (e.g., Hamming), this effect is negligible in most spectral bins.

For a single linear polarization, the power is given by

$$I_{x,k} = |X_k|^2 = \mathrm{Re}(X_k)^2 + \mathrm{Im}(X_k)^2. \tag{11}$$

Assuming both complex components have the same variance σ^2, the power follows a chi-squared distribution with two degrees of freedom:

$$I_{x,k} \sim \sigma^2 \chi^2(2). \tag{12}$$

During the fine channelization step, we integrate the N_{int} spectra in the time direction and combine the power from N_{pol} polarizations. Therefore, in the final Stokes I spectrogram, the total number of chi-squared degrees of freedom is given by:

$$\mathrm{DOF} = 2N_{\mathrm{pol}}N_{\mathrm{int}} = 2N_{\mathrm{pol}}\Delta f \Delta t, \tag{13}$$

$$I_k \sim \sigma^2 \chi^2(2N_{\mathrm{pol}}\Delta f \Delta t), \tag{14}$$

using Equation (8). For dual-polarization Stokes I data, $\mathrm{DOF} = 4\Delta f \Delta t$. This allows us to generate synthetic chi-squared noise with the correct number of degrees of freedom just from frame resolutions, which are either directly specified or inferred from observations. As noncalibrated intensity values are arbitrarily scaled, we can simply scale the magnitudes of synthetic chi-squared noise to match empirical observational noise distributions.

The main function for noise synthesis across a frame is `add_noise`, which adds random noise to every pixel in the data array. By default, it generates chi-squared noise with a user-specified mean intensity μ. As the mean of a chi-squared distribution equals the number of degrees of freedom, for dual-polarization data, we have

$$I_k \sim \left(\frac{\mu}{4\Delta f \Delta t}\right)\chi^2(4\Delta f \Delta t), \tag{15}$$

$$\langle I_k \rangle = \left(\frac{\mu}{4\Delta f \Delta t}\right) \cdot 4\Delta f \Delta t = \mu, \tag{16}$$

$$\mathrm{Var}(I_k) = \left(\frac{\mu}{4\Delta f \Delta t}\right)^2 \cdot 2 \cdot 4\Delta f \Delta t = \frac{\mu^2}{2\Delta f \Delta t}. \tag{17}$$

In addition to chi-squared noise, `add_noise` can also generate Gaussian noise. By the central limit theorem, as the degrees of freedom increase, a chi-squared distribution approaches a normal distribution. For example, $N_{\mathrm{int}} = 51$ for BL's standard high-spectral-resolution data product, so $\mathrm{DOF} = 204$ and the resulting background noise is close to Gaussian. Directly synthesizing Gaussian-distributed noise can save normalization steps in the data processing but should be used carefully when comparing with real observational data.

A useful extension of the noise synthesis function is `add_noise_from_obs`, which draws from archived observational statistics to set realistic intensity values. The observations were taken using the GBT at the *C* band and reduced to (1.4 s, 1.4 Hz) resolution. For example, for chi-squared noise, the function randomly selects an archived mean intensity, scales it to the appropriate frame resolution, and populates noise per Equation (15). An implementation detail of BL's fine channelization software, `rawspec`,[17] is that as part of the FFT, intensity values are scaled up by a factor of the FFT length N_{fine}. So, for observations going through the BL data pipeline (i.e., the same digitization and coarse channelization hardware):

$$\mu \propto N_{\mathrm{fine}}N_{\mathrm{int}}, \tag{18}$$

$$\propto N_{\mathrm{fine}}\Delta f \Delta t, \tag{19}$$

$$\propto \Delta t. \tag{20}$$

Alternatively, the function also accepts user-provided arrays of background noise intensity statistics from which to sample instead. This can be used for synthesizing data with intensity ranges from other telescopes (e.g., Parkes) or even GBT data at different frequency bands or sensitivities.

After noise synthesis, the frame will update class attributes storing the estimated mean μ_b and standard deviation σ_b of the background noise. For an empty frame, the first noise synthesis function will set these properties directly. For preloaded observational data and further noise injection, the frame estimates the background noise through iterative sigma clipping at the 3σ level to exclude outliers. For frames small enough that noise statistics do not change over the frequency bandwidth, this enables signal injection at desired S/N levels.

3.1.2. Signal Synthesis

For narrowband signal synthesis, the `add_signal` function creates heuristic, user-defined signals in spectrogram data. Our convention is that the spectrogram data have time on the *y*-axis and frequency on the *x*-axis.

In spectrogram `setigen`, narrowband signals have a "central" frequency at each time step and a unique spectral profile centered at that frequency. As such, there are four main heuristic descriptors for a narrowband signal in `setigen`:

1. `path`—$I_p(t)$: Central signal frequencies as a function of time, e.g., linear (constant) drift rate, quadratic drift rate
2. `t_profile`—$I_t(t)$: Signal intensity as a function of time, e.g., constant intensity, Gaussian pulses
3. `f_profile`—$I_f(f, f_0)$: Spectral profile as a function of frequency (offset from central frequency), e.g., sinc^2 profile, Gaussian profile
4. `bp_profile`—$I_{bp}(f)$: Bandpass profile as a function of absolute frequency

These descriptors are parameters for `add_signal` and are Python functions by type. A set of common functions are provided with `setigen`, and others can be custom-written. The simplest and most ideal kind of narrowband signal has a constant intensity and drift rate; such signals can be created straightforwardly through the wrapper function `add_constant_signal`.

[17] https://github.com/UCBerkeleySETI/rawspec

For a pixel at (t, f) in the time-frequency spectrogram, the intensity of a synthetic signal is calculated as

$$I(t, f) = I_t(t) I_f(f, I_p(t)) I_{bp}(f). \quad (21)$$

As such, Equation (21) is computed for every pixel in the spectrogram, as there is no robust way to constrain arbitrary intensity profiles. For example, even an ideal Gaussian function is nonzero at all distances and defining a suitable range depends on the experiment. For large spectrograms, it can be inefficient to calculate intensities for pixels far from the main signal, so users can provide a custom frequency range to limit the signal calculation.

The signal calculation is fully heuristic, in that the calculation is completely user-specified and does not take other effects into account, such as FFT leakage or spectral responses. As intensity is treated as a function of time and frequency, this process can overlook how intensities are integrated in reality. As a partial solution, `add_signal` provides the option to separately subintegrate within each pixel in the time and frequency directions.

In a similar vein, a difficult effect to handle robustly is Doppler smearing, in which a highly drifting signal will have its power spread into multiple frequency channels within the same time channel (Sheikh et al. 2019). While an analytical form exists for the spectral profile of a linearly drifting cosine signal, the smearing effect will naturally apply to more complex signals. Variable spectral profiles are not yet supported in `setigen`, but from a user standpoint, it would be tedious to manually construct custom smearing profiles that change at each time step. Using a similar process to numerical integration, `add_signal` has the option to approximate Doppler smearing by computing and averaging a given number of copies of the signal, spaced evenly between signal center frequencies in adjacent time steps. For instance, for the ith time channel at $t = t_i$, copies of the signal centered at even spacings between $I_p(t_i)$ and $I_p(t_{i+1})$ are averaged together to get the ith spectral profile. This is done for all time channels, so that channels with smaller signal drifts will be brighter than those with larger signal drifts by the correct ratio, as long as the number of copies gives enough coverage over the channel with the largest signal drift.

Sometimes it can be difficult or unwieldy to wrap up a desired signal property into a separate function, or perhaps there is existing external code that produces such properties. In these cases, we can instead use NumPy arrays to describe these signals, rather than functions. As of now, the `path`, `t_profile`, and `bp_profile` arguments can be arrays.

3.1.3. Common Frame Operations

Besides supporting noise and narrowband signal injection, `setigen` comes with a set of tools for radio spectrogram analysis. These range from convenience functions for parameter calculations to frame-level data transformations.

For instance, estimating the S/N of a signal in an integrated spectrum is a common step in radio analysis. This can be done through a frame's `integrate` function, which can also be used along the frequency axis to produce an intensity time series array.

To inject a signal at a desired S/N, the `get_intensity` function calculates the requisite signal level as

$$I_t = \mathrm{S/N} \cdot \frac{\sigma_b}{N_t^{1/2}}, \quad (22)$$

assuming that the frame has background noise with standard deviation σ_b and that the S/N is measured by dividing the integrated signal maximum by the integrated noise deviation. As discussed in Section 3.1.1, each frame tracks an estimate of σ_b calculated using iterative sigma clipping and updates it when synthetic noise is injected.

It can be convenient to define signals in terms of the pixels they traverse rather than the frequencies. To convert between these for a given frame, one can use the `get_frequency` and `get_index` functions. We define the *unit drift rate* for a given spectrogram resolution to be the drift rate given by

$$\dot{\nu}_1 = \frac{\Delta f}{\Delta t}, \quad (23)$$

which can be accessed with the `unit_drift_rate` attribute. For a linearly drifting signal passing through the top and bottom of the frame, the corresponding drift rate can be calculated using the `get_drift_rate` function.

Given a frame with a linearly drifting signal, we can "de-drift" the frame using `setigen.dedrift`. This shifts each spectrum an appropriate amount along the frequency direction so that such a signal would, on average, appear to have zero frequency drift, making it simpler to calculate the S/N. In practice, empirical drift rates are not generally multiples of the unit drift rate, so de-drifted signals will not be perfectly aligned.

We can create a "slice" of a frame by specifying left and right frequency indices, analogous to NumPy array slicing, by using the frame's `get_slice` function. This results in a new frame with a truncated range, which can be helpful for isolating signals in the time-frequency space for further analysis.

If one is interfacing with other BL or astronomy code bases, outputting `setigen` frames to filterbank or HDF5 format can be very useful. These are done via the `save_fil` and `save_hdf5` functions. Frame objects can also be written and loaded with `pickle`, a convenient serialization method that can keep data and user-provided metadata together.

3.1.4. Demonstration: Spectrogram Module

We present a minimal working example of creating a data frame with synthetic noise and a drifting signal. First, we construct an empty frame with the desired resolution; here, we use parameters that match those of BL's high-frequency-resolution data product:

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

frame=stg.Frame(fchans=256,
                tchans=16,
                df=2.7939677238464355*u.Hz,
                dt=18.253611008*u.s,
                fch1=6095.214842353016*u.MHz)
```

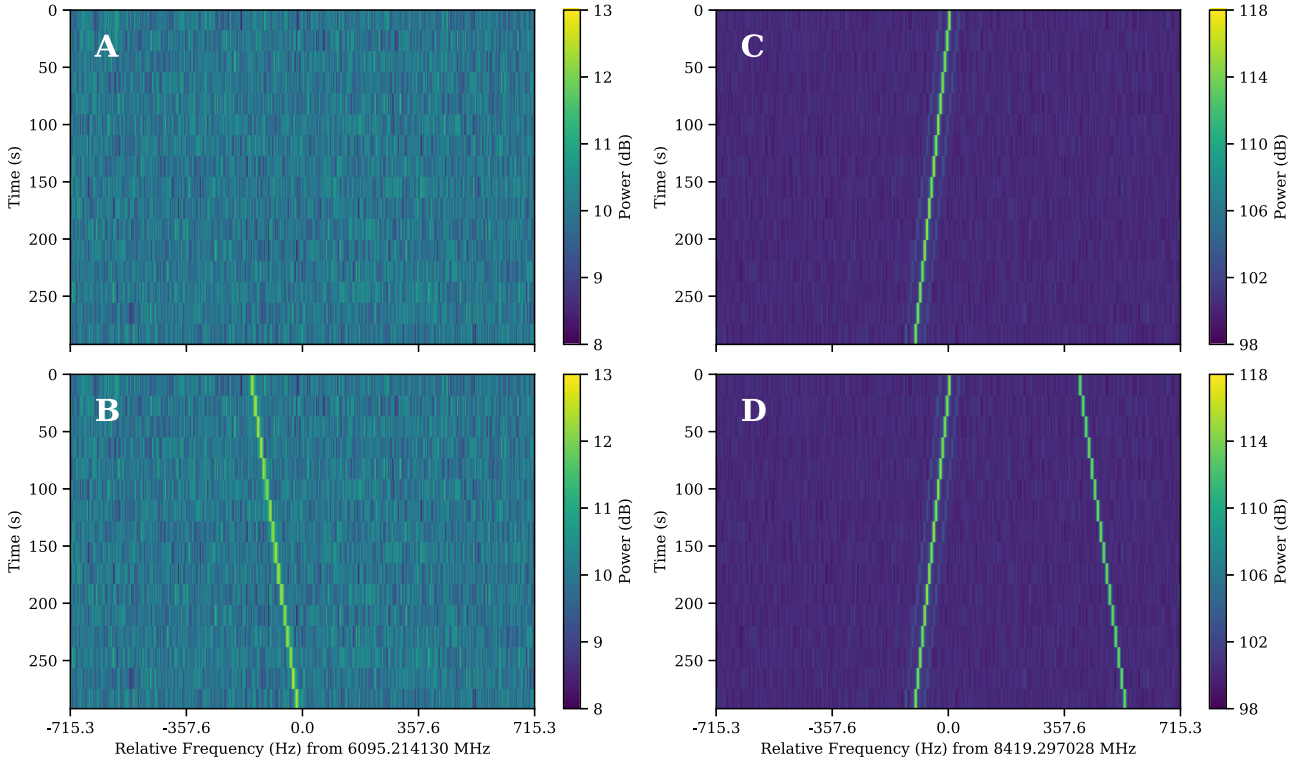

Figure 1. Radio spectrogram plots created from `setigen` frames. A: Frame with only synthetic chi-squared noise. B: Frame from panel (A) with an injected synthetic signal at $S/N = 30$. C: "Real" GBT observation of Voyager I carrier signal at the *X* band. D: Frame from panel (C) with an injected synthetic signal at $S/N = 1000$, with the same drift rate as the injected signal in panel (B).

Then, we add chi-squared noise with a desired mean, such as 10:

```
frame.add_noise(x_mean = 10, noise_type = 'chi2')
```

Finally, we add a simple drifting signal through our frame at $S/N = 30$ and plot the result in decibels (dB). The inputs to `add_signal` shown below are pre-written library functions that themselves return the functions described in Section 3.1.2. As they are indeed Python functions by type, the signal parameters allow for much more flexibility beyond this basic example.

```
frame.add_signal(
    stg.constant_path(
        f_start=frame.get_frequency(index=100),
        drift_rate=2*u.Hz/u.s
    ),
    stg.constant_t_profile(
        level=frame.get_intensity(snr=30)
    ),
    stg.gaussian_f_profile(width=10*u.Hz),
    stg.constant_bp_profile(level=1)
)

frame.bl_plot()
```

The frames after adding noise and after adding the signal are shown in Figures 1(A) and 1(B).

We also show an example with a signal detected from Voyager I in an *X*-band observation using the GBT, in Figure 1(C). Injecting a signal into the Voyager frame with the same drift rate as in the example (Figure 1(B)), now at $S/N = 1000$, we get Figure 1(D).

3.2. Raw Voltage Module

The raw voltage module is designed for synthesizing complex voltage data, providing a set of classes that model the signal processing pipeline described in Section 2. Instead of directly synthesizing spectrogram data, we can produce real voltages, pass them through a virtual pipeline, and record to file in GUPPI raw format. As this process models actual hardware used by BL for recording raw voltages, this enables lower level testing and experimentation.

The basic signal flow is shown in Figure 2. At the lowest level, a `DataStream` can accept noise and signal sources (as Python functions) and generate real voltages on demand. An `Antenna` models an antenna or dish used in radio telescopes and has one or two `DataStream` objects, corresponding to linear polarizations that are unique and not necessarily correlated. As described in Section 2, the sampled real voltages are passed to a processing pipeline, which consists, at its core, of a digitizer, a PFB, and a requantizer. In hardware, processing is done in fixed point arithmetic on an FPGA, but for simplicity, we use floating point. The digitizer quantizes input voltages to a specified number of bits and a target FWHM in the quantized voltage space. The filterbank implements a software PFB, 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.

The `RawVoltageBackend` object wraps around these elements and connects the full pipeline together. Given an observation length in seconds or a number of data recording

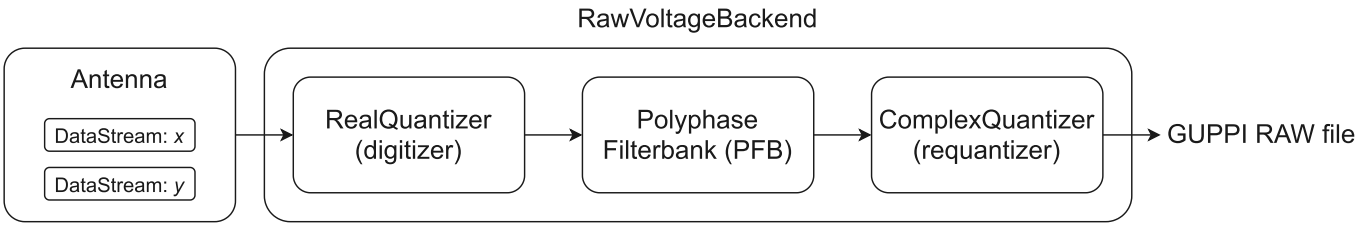

Figure 2. Basic layout of a voltage pipeline written using `setigen.voltage`.

"blocks," the main function `record` retrieves real voltage samples as needed and passes them through each backend element, finally saving the quantized complex voltages out to disk.

As voltage data are taken with very high sample rates, e.g., Gigasamples s^{-1} (Gsps), the voltage module is much more computationally expensive than the spectrogram module. To increase efficiency, most of the data manipulations are done with matrix operations, allowing for GPU acceleration with `CuPy` (Okuta et al. 2017).

3.2.1. Antennas and Data Streams

The `DataStream` class represents a stream of real voltage data for a single polarization and antenna. A data stream has an associated sample rate f_s, such as 3 GHz for the BL DR. As of now, the voltage module does not implement heterodyne mixing or bandpass filtering. Instead, data streams use a reference frequency `fch1` and frequency sign (ascending or descending from `fch1`) for voltage calculations.

The `Antenna` class is similarly defined by a sample rate, reference frequency, and frequency sign. For two linear polarizations, an `Antenna`'s data streams are available via the `x` and `y` attributes. For one polarization, only the former is available. For convenience, the `streams` attribute gets the list of available data streams for an antenna. One can add noise and signal sources to these individual data streams.

Real voltage noise is modeled as ideal Gaussian noise and added through the `add_noise` function. Note that this actually stores a Python function to the data stream that is only evaluated when `get_samples` is called. It also updates the data stream's `noise_std` attribute, which keeps track of the standard deviation of the voltages in that data stream. This is useful for injecting signals at target spectrogram S/Ns.

Drifting cosine signals can be added to a data stream using `add_constant_signal`. For more complex signals, one can write custom voltage functions to add using `add_signal`. Voltage signal sources are Python functions that accept an array of time stamps and output a corresponding sequence of real voltages. Here is a simple example that adds a nondrifting cosine signal with frequency `f_start`:

```
def cosine_signal(ts):
    delta_f=f_start---antenna.x.fch1
    return np.cos(2 * np.pi * delta_f * ts)

    antenna.x.add_signal(cosine_signal)
```

As custom signals are added, the `noise_std` parameter may no longer accurately reflect the background noise. In these cases, one can run the data stream's `update_noise` function to estimate noise empirically. This is not done by default to save computation, especially when there are multiple well-behaved voltage sources (e.g., Gaussian noise, cosine signals).

3.2.2. Quantization

The quantization process takes a continuous input voltage distribution and scales it to a target distribution that can be described by N_{bits} bits. As real voltage noise can be modeled by a Gaussian process, we can define this scaling in terms of the standard deviation or FWHM.

For real voltages $\{v\}$, target bit size N_{bits}, target mean μ_q (ideally 0), and target standard deviation σ_q, the quantized voltages v_q are given by:

$$v_s = \left\lfloor \frac{\sigma_q}{\sigma_v}(v - \langle v \rangle) + \mu_q \right\rfloor, \tag{24}$$

$$v_q = \min\left(\max(-2^{N_{\text{bits}}-1}, v_s), 2^{N_{\text{bits}}-1} - 1\right). \tag{25}$$

We can define quantizers in terms of a target FWHM w_q, in which case $\sigma_q = \frac{w_q}{2\sqrt{2\ln 2}}$.

The digitizer quantizes real voltages, while the requantizer receives complex voltages and quantizes per complex component. Quantization is run per polarization and antenna, and background statistics can be cached to save computation in subsequent calls. This is facilitated by the `RealQuantizer` and `ComplexQuantizer` classes.

3.2.3. Polyphase Filterbank

The `PolyphaseFilterbank` class implements and applies a PFB to quantized input voltages. Instead of directly applying a P-point FFT, a PFB first splits incoming voltages between P branches and lets M samples accumulate in each branch (Price 2021). A windowing function is applied over the $M \times P$ samples, the samples are summed over the M so-called polyphase taps, and finally a P-point FFT is taken of the result to get complex raw voltages in $N_{\text{coarse}} = P/2$ coarse channels. Further samples are read in groups of P and split between the PFB branches; accumulated samples step forward to the next tap to make room. PFBs have a better channel response than standard FFTs, especially as M increases, and are common in high-spectral-resolution radio back ends (Price 2021).

The two main parameters for a `PolyphaseFilterbank` are the number of taps M and the number of branches P. As the PFB works on MP samples at once, the object continuously caches samples for on-demand computation. The PFB also accepts a symmetric windowing function as an argument (Hamming, by default) and generates MP coefficients up front (Blackman & Tukey 1958).

3.2.4. Combining Components and Recording Data

The `RawVoltageBackend` class contains the full machinery to collect, process, and write complex voltage data to

GUPPI raw files, as in the standard pipeline shown in Figure 2. Nevertheless, as the individual signal processing components are all exposed as part of the voltage module, custom pipelines can be written by chaining them in different ways.

A `RawVoltageBackend` takes in components external to the data recording process as parameters, such as the antenna, digitizer, PFB, and requantizer. All other parameters and functions are specific to data recording and actually obtaining data from the external components.

As described by Lebofsky et al. (2019), the block size $N_{\mathrm{blocksize}}$ refers to the number of bytes in a single block of data in GUPPI format. Each data block has an associated header with observing metadata, such as target and frequency information. The number of blocks per file must also be specified to size individual raw files; multiple raw files may be associated with a single pointing. For standard 5 minute GBT observations, BL DR uses $N_{\mathrm{blocksize}} = 134217728$ with 128 blocks per file.

To specify the coarse channels that are actually recorded to disk, we can set the starting index and the number of consecutive channels N_{chan} to ultimately save. Purely for computational efficiency, we always perform a full FFT and truncate to obtain the desired coarse channels, instead of directly doing the transform operation on the subset of coarse channels. Especially when using a GPU to accelerate synthesis, this can fill up memory rather quickly, potentially to the point of overflow. Therefore, the `RawVoltageBackend` has an additional option to divide individual data blocks into a given number of subblocks, such that each subblock will fully fit in memory.

For a single antenna, the number of bytes $N_{\mathrm{blocksize}}$ in a block can be related to the number of time channels $N_{t,\mathrm{block}}$ corresponding to a single block in (nonintegrated) spectrogram format as

$$N_{\mathrm{blocksize}} = 2N_{\mathrm{pol}}\left(\frac{N_{\mathrm{bits},r}}{8}\right)N_{\mathrm{chan}}N_{t,\mathrm{block}}, \tag{26}$$

$$= \frac{1}{4}N_{\mathrm{pol}}N_{\mathrm{bits},r}N_{\mathrm{chan}}N_{t,\mathrm{block}}, \tag{27}$$

based on the structure of raw files as described by Lebofsky et al. (2019).

3.2.5. Multiantenna Support

To simulate voltage data for interferometric pipelines, it can be useful to synthesize raw voltage data from multiple antennas. `setigen` supports synthesizing multiantenna output through the `MultiAntennaArray` class, which creates a list of N_{ant} antennas each with an associated integer delay (in time samples). In addition to the individual data streams that allow the user to add noise and signals to each antenna, there are "background" data streams `bg_x` and `bg_y` in `MultiAntennaArray`, representing correlated noise or RFI that is detected at each antenna, subject to the (relative) delays. Signals and noise can therefore be added to the background across all array elements as well as to individual antennas.

The only difference in the pipeline is instead of supplying a `Antenna` as input to a `RawVoltageBackend`, one would supply a `MultiAntennaArray`. Then, the output is saved as a multiantenna extension of the GUPPI raw format.

3.2.6. Creating Signals at a Target Spectrogram S/N

During the course of the full signal processing pipeline, an injected cosine signal passes through multiple quantization and FFT steps. In many SETI experiments, a signal's S/N in spectrogram data is used for thresholding and analysis, so it is important to be able to estimate this S/N given pipeline parameters.

Suppose that we have a cosine signal with amplitude A at a frequency corresponding to the center of a fine spectral channel, and that this signal is injected onto a background of Gaussian noise $\mathcal{N}(0, \sigma_v^2)$. As the voltage data are real-valued, the signal magnitude becomes $A/2$ in the frequency space. As the voltages pass through the coarse and fine channelization steps, the signal magnitude picks up factors of P and N_{fine}, respectively, compared to the background noise.

The background noise will follow a chi-squared distribution with $\mathrm{DOF} = 2N_{\mathrm{pol}}N_{\mathrm{int}}$ (Section 3.1.1), scaled by multiplicative factors arising from quantization and FFT calculations. As the input voltage noise has variance σ_v^2, the standard deviation of the noise power σ_b will be proportional to the standard deviation $\sigma_{b,0}$ of a chi-squared distribution with mean σ_v^2. The time integration step to get the S/N will reduce this noise by a factor of $N_t^{1/2}$.

To get an expression for N_t given observation parameters, suppose our synthetic observation has N_{block} total blocks and that the time covered by a single block is τ_{block}. Then, we have the following equations:

$$\Delta t = \frac{N_{\mathrm{int}}}{\Delta f} = \frac{P}{f_s}N_{\mathrm{fine}}N_{\mathrm{int}}, \tag{28}$$

$$\tau_{\mathrm{block}} = N_{t,\mathrm{block}}\Delta t, \tag{29}$$

$$N_t = \frac{N_{\mathrm{block}}\tau_{\mathrm{block}}}{N_{\mathrm{int}}\Delta t} = \frac{N_{\mathrm{block}}N_{t,\mathrm{block}}}{N_{\mathrm{int}}}. \tag{30}$$

Combining all of these factors, we can express the final S/N of the signal as the ratio between the integrated (mean) signal power and the integrated background noise standard deviation as

$$\sigma_{b,0} = \sigma_v^2\left(\frac{2}{\mathrm{DOF}}\right)^{1/2}, \tag{31}$$

$$\mathrm{S/N} = \frac{I}{\sigma_b} = \frac{(A/2)^2 PN_{\mathrm{fine}}}{\sigma_{b,0}/N_t^{1/2}}. \tag{32}$$

This yields the amplitude or signal level in terms of the target S/N:

$$A = \left(\mathrm{S/N}\cdot\frac{4\sigma_{b,0}}{PN_{\mathrm{fine}}N_t^{1/2}}\right)^{1/2}. \tag{33}$$

Notice that A has a linear dependence on the standard deviation σ_v of the real voltage noise in a data stream, which can arise from multiple sources, especially in a multiantenna array. Given pipeline parameters, the `get_level` function can be used to calculate A/σ_v.

For a nondrifting cosine signal, we can also approximate the effect of spectral leakage between fine channels by comparing the signal frequency to the nearest channel central frequency. A signal with amplitude A centered at a frequency δf away from

the center of the closest fine spectral channel will have its power I attenuated by[18]

$$\frac{I'}{I} = \mathrm{sinc}^2\left(\frac{|\delta f|}{\Delta f}\right). \tag{34}$$

As intensity goes as voltage squared, we provide a function `get_leakage_factor` to calculate an amplitude adjustment factor f_l to easily scale from A to a new amplitude A' that corresponds to the nonattenuated intensity:

$$f_l = \frac{1}{\mathrm{sinc}\left(\frac{|\delta f|}{\Delta f}\right)}, \tag{35}$$

$$A' = f_l A. \tag{36}$$

Finally, for a linearly drifting cosine signal, if the drift rate $\dot{\nu}$ exceeds the unit drift rate $\dot{\nu}_1$, signal power will be smeared across multiple frequency bins in spectrogram data. This is a linear effect in spectrogram data, so cosine amplitudes should be increased by a factor of $(\dot{\nu}/\dot{\nu}_1)^{1/2}$ to counteract the apparent loss in power.

3.2.7. Injecting Synthetic Signals into Raw Voltage Data

In addition to creating fully synthetic complex voltage data from scratch, the `RawVoltageBackend` supports injecting or adding synthetic data to existing observational GUPPI raw data. The pipeline remains mostly the same, except for a few important differences that we detail below.

In order to get meaningful results, we must know and match details about the specific signal processing pipeline that produced the existing raw data. `setigen` provides a helper function called `get_raw_params` to extract header information from the raw data file, but other information must be provided separately by the user, such as the sampling rate and PFB parameters.

As recorded voltage data have already gone through multiple quantization steps, we cannot directly add time series voltages together (i.e., at the original ADC sampling rate). Instead, we choose to synthesize complex voltage data separately, add them to the recorded voltage data, and apply a final quantization step to match the initial distribution as best as possible.

However, this process requires that we create and process signals that are not necessarily embedded in noise. In typical narrowband signal injection scenarios, we wish to synthesize and inject signals whose distributions are non-Gaussian (e.g., a cosine signal). As the quantization steps assume that the input and output voltage distributions are both Gaussian, attempting to quantize bare narrowband signals will cause distortion and introduce clipping artifacts. Furthermore, without a reference noise distribution, quantization can scale the magnitude of processed signals in undesired ways, making S/N estimation difficult.

To address these issues, we approach the quantization steps differently. If there is already a synthetic noise source, we proceed normally through all steps in the pipeline. Otherwise, we skip the initial digitization step before the PFB, and instead treat the input voltages as if they followed a zero-mean Gaussian distribution with variance 1. Using a reference distribution allows us to set signal magnitudes with the `get_level` function to achieve target S/N levels. We then estimate the post-PFB mean and standard deviation of the reference Gaussian voltages and quantize the synthetic voltages based on these values instead of those from the "real synthetic distribution. This way, if the synthesized voltages were actually embedded in $\mathcal{N}(0, 1)$ noise, the resulting signal quantization would be very similar.

For each data block in the recorded raw file, the `RawVoltageBackend` will set requantizer statistics (target mean μ_q and target standard deviation σ_q) calculated from the existing data for each combination of antenna, polarization, and complex component. The synthetic voltages are 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 quantized synthetic data to the existing data, we do not change the overage voltage mean. After these are added together, we finally requantize once more to the target mean and target standard deviation to match the existing data statistics and magnitudes as best as possible.

3.2.8. Demonstration: Voltage Module

Here, we present a simple pipeline created with the raw voltage module to inject a drifting cosine signal in Gaussian noise. First, we create the signal processing elements:

```
from astropy import units as u
from setigen.voltage import *

d=RealQuantizer(target_fwhm=32,
                num_bits=8)

f=PolyphaseFilterbank(num_taps=8,
                      num_branches=1024)

r=ComplexQuantizer(target_fwhm=32,
                   num_bits=8)
```

Then, we create the antenna, setting the sampling rate and reference frequency. With two polarizations, we can add Gaussian noise and a constant amplitude, Doppler drifting cosine signal to both data streams:

```
a=Antenna(sample_rate=3*u.GHz,
          fch1=6000*u.MHz,
          ascending=True,
          num_pols=2)

for s in a.streams:
    s.add_noise(v_mean=0,
                v_std=1)

 s.add_constant_signal(f_start=6002.1*u.MHz,
                       drift_rate=-2*u.Hz/u.s,
                       level=0.004)
```

We connect these components through the recording backend, defining the dimensions and size of the final raw voltage data product, and record a block of data to file.

```
rvb=RawVoltageBackend(a,
                      digitizer=d,
                      filterbank=f,
                      requantizer=r,
```

[18] $\mathrm{sinc}\, x = \sin x/x$.

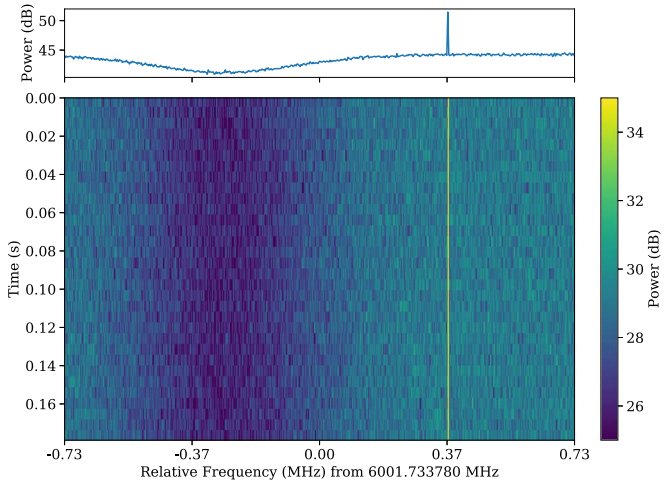

Figure 3. Spectrogram derived from synthetic raw voltages, showing the edge of the coarse channel bandpass shape and a bright, slightly drifting cosine signal. The top panel shows an integrated profile, showing the PFB scalloping loss toward the left and the synthetic signal toward the right.

(Continued)

```
                    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='TELESCOP': 'GBT',
           verbose=True)
```

After saving the raw voltages to disk, we reduce using `rawspec` with $N_{\rm fine} = 1024$ and $N_{\rm int} = 4$. A snippet of the resulting spectrogram output is shown in Figure 3, where intensities are plotted on a decibel scale. The signal is readily apparent, as is the frequency bandpass shape arising from the PFB.

4. Discussion

4.1. Limitations

While `setigen` is a flexible library that enables quick narrowband data set generation, it is important to discuss the limitations when using it for science.

First and foremost, `setigen` relies on heuristic, user-defined signals, rather than simulations from first principles. The search for technosignatures is necessarily informed by human bias, specifically applied via our assumptions about a technosignature's potential characteristics and morphology. It is possible that radiation from an extraterrestrial intelligence will exist in a form that we have not considered or designed searches for. Even when we consider only the problem of excision of anthropogenic RFI, we have to be careful when applying algorithms developed using the simplest of narrowband signals. Although there might never be a way of fully emulating the breadth and variety of the RFI environment, `setigen` can still be used to generate labeled, complex signals to test the efficacy of new and existing algorithms.

In a similar vein, the spectrogram module enables users to quickly generate signals that "look" like the narrowband signals we see in observations. However, as spectrogram signal injection does not have access to phase information, it is impossible to replicate the "correct" intensity statistics when adding a signal to integrated Stokes I noise. For example, adding a perfect cosine signal to zero-mean Gaussian noise in the voltage domain results in a noncentral chi-squared intensity distribution in Stokes I data, but adding a signal with constant intensity directly to chi-squared noise in a spectrogram does not result in the same distribution (over the pixels occupied by that signal; McDonough & Whalen 1995). While this effect is negligible for high S/N signals, algorithms developed to target low S/N signals may suffer from intrinsic inaccuracies in the intensity statistics.

Signal injection in the complex voltage domain also has limitations as we are not able, in software, to directly add signals in the real (analog) voltage stage. Raw data are quantized multiple times in hardware, so the injection step has to take place using complex voltages that are quantized in a similar way. While fundamental steps in the pipeline are linear, such as PFB operations (Equation (10)), quantization inherently breaks this linearity. Because of this, summing real and synthetic voltages that are independently processed can lead to artifacts and intensity discrepancies that would not arise if we could inject at the start of the signal processing pipeline.

4.2. Future Directions

`setigen` is written and developed with the needs of SETI researchers in mind, so new functionality and improvements are constantly being added. Here, we describe some potential enhancements that may be added in the near future.

As it stands, the spectrogram module is especially targeted at producing small frames with synthetic signals rather than injecting into large, broadband observations. While this suffices in many cases, it may be useful to inject within large data files in which frequency bandpass shapes significantly change the background intensities. For instance, for use in S/N estimation, `setigen` calculates background noise statistics over an entire frame rather than localized around the target signal injection frequency. For a large enough frame, this is both an inefficient and inaccurate calculation due to variable bandpass shapes. An improvement would be to localize the noise calculation to a window around the target injection site, as well as to similarly localize the signal injection calculation to prevent unnecessary computation.

The spectrogram module is also currently designed expressly to synthesize narrowband signals. There are many similarities in both signal processing and experimental design between technosignature searches and searches for time-varying phenomena such as pulsars and fast radio bursts (FRBs); `setigen` could thus be expanded to include broadband signal injection (Zhang et al. 2018b; Gajjar et al. 2021).

An exciting potential addition is to use parameterized ML methods to create labeled, realistic signals. By taking ideas from style transfer, a synthetic RFI signal could be created by specifying heuristic parameters and having an ML model generate such a signal with RFI-like properties (Gatys et al. 2016; Dai et al. 2017). While generative adversarial networks (GANs) have been used before to create radio spectrograms (Zhang et al. 2018a), conditional GANs that accept input parameters might help produce more specific, labeled signals, which can be better for certain SETI experiments. Furthermore, better RFI modeling could help improve ML-based searches for

astrophysical phenomena like FRBs in the presence of different classes of RFI.

Some of these enhancements may use a lot more computational power than the current synthesis process, so the option to GPU-accelerate the standard spectrogram module would be critical. Some of these enhancements may require a more careful look at file input/output methods when reading and writing large observational data files to avoid unnecessary or slow operations.

The raw voltage module can also be expanded to support alternate radio telescope configurations and back ends, such as those behind interferometers like MeerKAT (Jonas 2009). While `setigen` already has basic multiantenna functionality, it could be helpful to build on this with general-use utilities, such as routines that predict how a given injected signal would appear across multiple antennas or beams. The voltage module could also support additional requantization and recording modes, such as 2 and 16 bit. As interferometer usage in modern radio SETI continues to grow, `setigen` capabilities can be extended to help test signal detection in commensal and beamformed observations (Czech et al. 2021).

5. Summary

In this paper, we presented `setigen`, an open-source Python library for the creation and injection of synthetic narrowband radio signals. `setigen` can produce both finely channelized spectrogram data and coarsely channelized complex voltage data. The spectrogram module is designed to be intuitive and quick to use to facilitate the construction of synthetic data sets for SETI experiments and testing. While the voltage module is more complex and computationally intensive, it enables analysis of signals that pass through a software-defined pipeline, which can be helpful in understanding the implications of the instrumentation pipeline itself in SETI searches.

`setigen` is constantly being improved with the needs of SETI research in mind. As open-source software, the library is freely available, and we encourage the SETI community to use and contribute to it.

Breakthrough Listen is managed by the Breakthrough Initiatives, sponsored by the Breakthrough Prize Foundation. The Green Bank Observatory is a facility of the National Science Foundation, operated under cooperative agreement by Associated Universities, Inc. We thank the staff at the Green Bank Observatory for their operational support.

Software: `NumPy` (Oliphant 2006), `CuPy` (Okuta et al. 2017), `SciPy` (Virtanen et al. 2020), `Astropy` (Robitaille et al. 2013), `Blimpy` (Price et al. 2019), `H5py` (Collette et al. 2017), `Matplotlib` (Hunter 2007).

ORCID iDs

Bryan Brzycki https://orcid.org/0000-0002-7461-107X
Andrew P. V. Siemion https://orcid.org/0000-0003-2828-7720
Imke de Pater https://orcid.org/0000-0002-4278-3168
Steve Croft https://orcid.org/0000-0003-4823-129X
John Hoang https://orcid.org/0000-0001-5591-5927
Cherry Ng https://orcid.org/0000-0002-3616-5160
Danny C. Price https://orcid.org/0000-0003-2783-1608
Sofia Sheikh https://orcid.org/0000-0001-7057-4999
Zihe Zheng https://orcid.org/0000-0003-3248-2174

References

Bellanger, M., Bonnerot, G., & Coudreuse, M. 1976, ITASS, 24, 109
Blackman, R. B., & Tukey, J. W. 1958, BSTJ, 37, 485
Brzycki, B., Siemion, A. P., Croft, S., et al. 2020, PASP, 132, 114501
Brzycki, B. 2022, setigen, v2.3.1, Zenodo, doi:10.5281/zenodo.6410676
Collette, A., Tocknell, J., Caswell, T. A., et al. 2017, h5py, v2.4.0, Zenodo, doi:10.5281/zenodo.400660
Czech, D., Isaacson, H., Pearce, L., et al. 2021, PASP, 133, 064502
Dai, B., Fidler, S., Urtasun, R., & Lin, D. 2017, in Proc. of the IEEE Int. Conf. on Computer Vision (New York: IEEE), 2970
Drake, F. D. 1961, PhT, 14, 40
DuPlain, R., Ransom, S., Demorest, P., et al. 2008, Proc. SPIE, 7019, 70191D
Enriquez, E., & Price, D. 2019, turboSETI: Python-based SETI search algorithm, Astrophysics Source Code Library, ascl:1906.006
Enriquez, J. E., Siemion, A., Foster, G., et al. 2017, ApJ, 849, 104
Gajjar, V., Perez, K. I., Siemion, A. P. V., et al. 2021, AJ, 162, 33
Gatys, L. A., Ecker, A. S., & Bethge, M. 2016, in Proc. of the IEEE Conf. on Computer Vision and Pattern Recognition (New York: IEEE), 2414
GBT Support Staff 2022, Green Bank Observatory Proposer's Guide for the Green Bank Telescope (Green Bank, WV: GBO), https://www.gb.nrao.edu/scienceDocs/GBTpg.pdf
Harp, G., Richards, J., Shostak, S., et al. 2019, arXiv:1902.02426
Harris, C., & Haines, K. 2011, PASA, 28, 317
Harris, C. R., Millman, K. J., van der Walt, S. J., et al. 2020, Natur, 585, 357
Hickish, J., Abdurashidova, Z., Ali, Z., et al. 2016, JAI, 5, 1641001
Hunter, J. D. 2007, CSE, 9, 90
Jonas, J. L. 2009, IEEEP, 97, 1522
Lebofsky, M., Croft, S., Siemion, A. P. V., et al. 2019, PASP, 131, 124505
MacMahon, D. H. E., Price, D. C., Lebofsky, M., et al. 2018, PASP, 130, 044502
Margot, J.-L., Greenberg, A. H., Pinchuk, P., et al. 2018, AJ, 155, 209
Margot, J.-L., Pinchuk, P., Geil, R., et al. 2021, AJ, 161, 55
McDonough, R. N., & Whalen, A. D. 1995, Detection of Signals in Noise (New York: Academic Press)
Nita, G. M., Gary, D. E., Liu, Z., Hurford, G. J., & White, S. M. 2007, PASP, 119, 805
Okuta, R., Unno, Y., Nishino, D., Hido, S., & Loomis, C. 2017, in Proc. of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conf. on Neural Information Processing Systems (NIPS) (Red Hook, NY: Curran Associates Inc.), http://learningsys.org/nips17/assets/papers/paper_16.pdf
Oliphant, T. E. 2006, A Guide to NumPy, Vol. 1 (USA: Trelgol Publishing)
Pence, W. D., Chiappetti, L., Page, C. G., Shaw, R. A., & Stobie, E. 2010, A&A, 524, A42
Pinchuk, P., & Margot, J.-L. 2022, AJ, 163, 76
Pinchuk, P., Margot, J.-L., Greenberg, A. H., et al. 2019, AJ, 157, 122
Price, D., Enriquez, J., Chen, Y., & Siebert, M. 2019, JOSS, 4, 1554
Price, D. C. 2021, The WSPC Handbook of Astronomical Instrumentation: 1: Radio Astronomical Instrumentation (Singapore: World Scientific), 159
Price, D. C., Enriquez, J. E., Brzycki, B., et al. 2020, AJ, 159, 86
Price, D. C., MacMahon, D. H. E., Lebofsky, M., et al. 2018, PASA, 35, 41
Robitaille, T. P., Tollerud, E. J., Greenfield, P., et al. 2013, A&A, 558, A33
Sheikh, S. Z., Siemion, A., Enriquez, J. E., et al. 2020, AJ, 160, 29
Sheikh, S. Z., Wright, J. T., Siemion, A., & Enriquez, J. E. 2019, ApJ, 884, 14
Siemion, A. P. V., Demorest, P., Korpela, E., et al. 2013, ApJ, 767, 94
Tarter, J. 2001, ARA&A, 39, 511
Taylor, J. H. 1974, A&AS, 15, 367
Thompson, A. R., Moran, J. M., & Swenson, G. W. J. 2017, Interferometry and Synthesis in Radio Astronomy (3rd ed.; Cham: Springer)
Virtanen, P., Gommers, R., Oliphant, T. E., et al. 2020, NatMe, 17, 261
Welch, J., Backer, D., Blitz, L., et al. 2009, IEEEP, 97, 1438
Werthimer, D., Tarter, J., & Bowyer, S. 1985, Symposium-International Astronomical Union, Vol. 112 (Cambridge: Cambridge Univ. Press), 421
Worden, S. P., Drew, J., Siemion, A., et al. 2017, AcAau, 139, 98
Wright, S. A., Werthimer, D., Treffers, R. R., et al. 2014, Proc. SPIE, 9147, 91470J
Zhang, Y. G., Gajjar, V., Foster, G., et al. 2018b, ApJ, 866, 149
Zhang, Y. G., Won, K. H., Son, S. W., Siemion, A., & Croft, S. 2018a, in 2018 IEEE Global Conf. on Signal and Information Processing (GlobalSIP) (New York: IEEE), 1114