FLAG Spectral Line Software

Nickolas M. Pingel

December 29, 2020

1 Introduction

This document provides descriptions and cookbook style examples of the usage and output of the codes contained within the `SpectralFiller` python software package for use with data from the Focal L-Band Array for the Green Bank Telescope (FLAG) — a cryogenically cooled, 19 element, dual-polarization phased array feed (PAF). This suite contains scripts to perform data monitoring, post-correlation beamforming, calibration, and imaging from raw covariance matrices. FLAG is optimized to operate between frequencies ranging between 1 GHz and 2 GHz, with a focus on spectral line mapping of the 1420.406 MHz transition of neutral hydrogen (HI), pulsar timing, and pulsar/transient detection.

The following will focus on how to monitor, calibrate, and image data specific to the spectral line mapping and calibration modes of FLAG. This document is to be used in conjunction with documentation on the usage of the backend modes[1] while observing. This document is organized as follows: Section 2 outlines the formats of the raw and final data products; Section 3 describes the logic, usage, and examples of the scripts that perform the post-correlation beamforming to transition the raw covariance data into SDFITS files; Section 5 provides examples of observing configuration and Astrid scripts; Section 6 describes and provides examples for scripts that are to be used to monitor FLAG calibration and spectral line observations; and Section 7 summarizes and provides examples of the data calibration, reduction, and imaging scripts.

2 Raw Data Formats and SDFITS

This section outlines the format of the raw covaraince matrices and the final form of the SDFITS files.

2.1 Raw Data

The backend for the PAF was developed in collaboration with the Brigham Young University (BYU), West Virginia University (WVU) and the Green Bank Observatory (GBO). It consists of five high performance computing nodes (HPCs), each equipped with two Nvidia GeForce Titan X Graphical Processing Units (GPUs). Each HPC was

[1] `https://radioastron.groups.et.byu.net/dokuwiki/lib/exe/fetch.php?media=flag_public:flag_user_manual.pdf`

connected to the Reconfigurable Open Architecture Computing Hardware (ROACH)[2] Field Programmable Gate Arrays (FPGA) boards and received one fifth of the total bandwidth of 151.59 MHz. Each HPC can run in three basic modes: (1) the calibration correlator mode (CALCORR) wherein the bandpass was made up of 500 discrete 'coarse' channels each 0.30318 MHz wide; (2) The polyphase filter bank (PFB) correlator mode (PFBCORR) where a 30.318 MHz (100 coarse channels) section of the original bandpass is selected to be sent through a PFB to obtain finer channelization. In this mode, a contiguous set of five coarse channels is output to 160 'fine' channels for a final frequency resolution of 9.47 kHz; (3) the real-time beamformer mode (RTBF) where precomputed beamformer weights are read in and applied to save beamformed spectra to disk. This mode is designed to be used to detect transient sources such as pulsars and fast radio bursts. The remaining discussion will focus solely on the CALCORR and the PFBCORR modes.

In both correlator modes, each GPU runs two correlator threads making use of the xGPU library[3], which is optimized to work on FLAG system parameters. Each correlator thread handles one-twentieth of the total bandwidth made up of either 25 *non-contiguous* coarse frequency channels or 160 contiguous fine channels and writes the raw output to disk in a FITS[4] file format. The data acquisition software used to save these data to disk was borrowed from development code based for the Versatile GBT Astronomical Spectrometer (VEGAS) engineering FITS format. The output FITS file from each correlator thread is considered a 'bank' with a unique X-engine ID (XID; i.e., the correlator thread) ranging from 0 to 19 that is stored in the primary header of the FITS binary table.

The raw data output for both correlator modes are the covariance matrices denoting the covariance between individual dipole elements. However, due to xGPU limitations, the covariance matrices are shaped 64×64 (with elements for row and column over 40 being set to zero) and flattened to a one-dimensional data vector whose length depends on the specific correlator mode. An example of how the covariance values are ordered is illustrated in Figure 1. Here, $R_{\mathrm{k}}^{\mathrm{i,j}}$ corresponds to the covariance between dipole i and j at frequency channel k. When in CALCORR mode, the bank file corresponding to XID ID 0 contains covariances matrices for frequency channels 0 to 4, 100 to 104, ..., 400 to 404; the XID 1 bank file stores covariance matrices for frequency channels, 5 to 9, 105 to 109, ..., 405 to 409. However in PFBCORR mode, the covariance matrices for channels 0 to 159 are stored in the bank file corresponding to XID 0 and continue in a contiguous fashion such that the bank file corresponding to XID 19 stores data for frequency channels 3039 to 3199. The logic during data reduction is to process each frequency channel individually, then sort the result into the final bandpass based on the XID and mode in which the data were taken. The methods employed to construct the two-dimensional form of the covariance matrices and sort the frequency channels are discussed in depth in Section 3.2.

[2] `https://casper.berkeley.edu/wiki/ROACH-2_Revision_2`

[3] `https://github.com/GPU-correlators/xGPU/tree/master/src`

[4] `https://fits.gsfc.nasa.gov/standard40/fits_standard40aa-le.pdf`

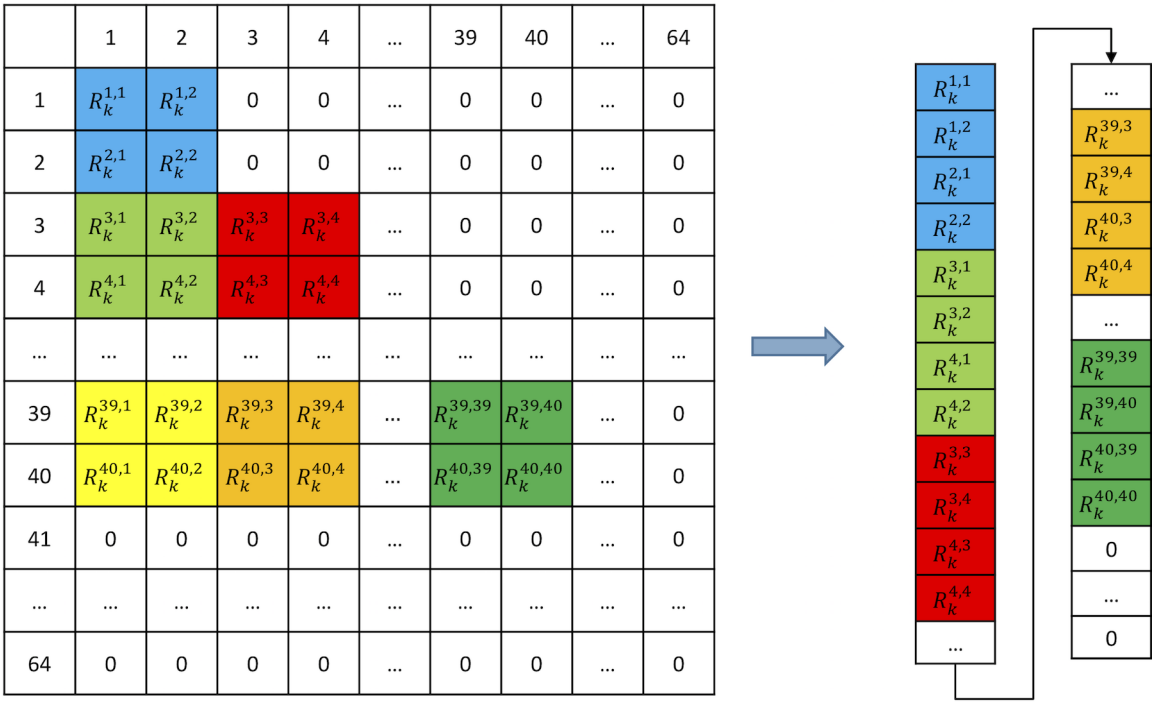


Figure 1: The structure of a covariance matrix used in beamforming. The numbers preceding each row/column correspond to the dipole element. Each element of the matrix stores the covariance between dipole elements i and j for a single frequency channel, k. The output is ordered in a flattened one-dimensional array that needs to be reshaped into a 40×40 matrix before beamforming weights can be applied. Additionally, due to xGPU limitations, the output size is 64×64, which results in many zeros that need to be thrown away in data processing.

2.2 SDFITS

A Single-Dish FITS (SDFITS) file is a combination of binary table FITS tables that describe a collection of integrations taken with the GBT. The data stored in these binary tables are specific for the GBO computing environment such that the files can be read into and manipulated by GBTIDL[5]. The primary goal of this python package is to transform the raw dipole covariances into beamformed spectra that represent power as a function of frequency channels, which can subsequently be calibrated with reference scans. For more information on the SDFITS format, see the following links:

`https://safe.nrao.edu/wiki/bin/view/Main/SdfitsDetails`
`https://casa.nrao.edu/aips2_docs/notes/236/node14.html`

3 Core Software

The scripts that drive the creation of an SDFITS file are `PAF_Filler.py` — in essence the 'main' function of the program — and the two modules `metaData- Module.py` and `beamformerModule.py`. The following subsections describe the logic flow for each component and script that is used to convers the binary files containing the complex weights to FITS files.

[5] `http://gbtidl.nrao.edu/`

FITS file (WVU)	Binary file (BYU)
0 (boresight)	3 (boresight)
1	0
2	1
3	4
4	6
5	5
6	2

Table 1: Conversion between beam name conventions

3.1 WeightFiller.py

The foremost step in the filling and calibration process of FLAG data is to convert a set of 20 binary files (one for each bank) that store the complex beamforming weight data to FITS files. The binary table structure of a FITS file allows for the weights to be readily accessible to the primary filling software for the application to the raw correlations to create beamformed bandpasses. Figure 2 (courtesy of Richard Black) summarizes the format of these binary files. The first component of the file contains (in sequential order) the real and imaginary component of the complex weight for each polarization, beam, coarse frequency channel, and dipole element. Note that due to the limitations of xGPU — the GPU based FX correlator software that drives the throughput processes of the backend — the correlation matrices are of shape 64×64. Note here that while we are only interested in the first 40 elements (corresponding to the number of dipoles multiplied by the two linear polarizations plus two spare data channels), for ease of formatting the binary table, the irrelevant correlations are kept at this stage to be thrown out by subsequent processing codes. It is also very important to note that the naming convention between beams changes from the binary to FITS version of the file. The change is implemented so the typical seven beam configuration will mimic the convention of other multi-beam receivers (e.g., Arecibo's ALFA); that is, Beam 0 in the FITS file format refers to the boresight beam, Beam 1 is the upper left beam, and the subsequent beam numbers increase in a clockwise fashion. Table 1 summarizes how the beam names are altered from the binary (BYU) to FITS (WVU) convention. All subsequent references to the beam names in this document refer to the WVU convention.

For the typical seven beam pattern, the total size of these binary files are 179392 bytes (4×2 total bytes per float pair representing a complex weight value × 2 polarizations × 7 beams × 25 coarse frequency channels × 64 total elements = 179200 bytes + 192 bytes for the header payload). The script uses the built-in `struct` package of python to unpack these binary files and arrange the complex weight values into a `numpy` array of 14×3200 (7 beams × 2 polarizations)×(64 elements×polarizations × 25 frequency channels×2 for the complex pair).

Before writing out the final FITS file, the metadata that contains the beam offsets, calibration set filenames, beamforming algorithm, and XID must also be sorted. Note that the 'beam offsets' refer to the cross-elevation and elevation offset (in units of degs) from the telescope pointing center recorded in the ancillary Antenna FITS file. An individual FITS file is written to disk for each bank. It contains a primary

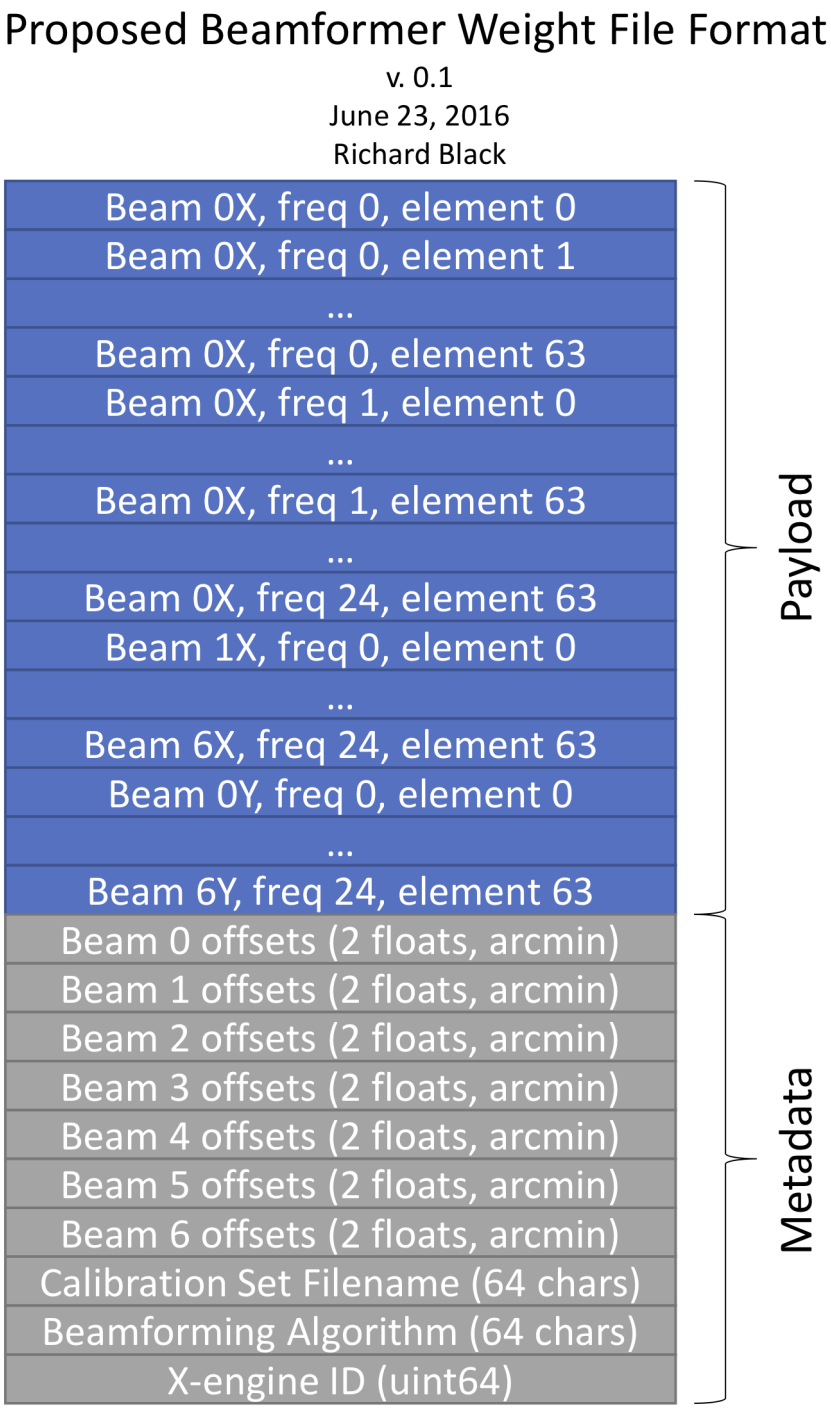


Figure 2: A summary of the structure of the binary files that contain the complex beamforming weights.

header that lists the calibration set filename, beamforming algorithm used, and XID, with the beamforming weights and offsets available in the first (and only) binary table extension 'Beam Weights and Offsets'. The name of the weight columns is formatted as Beam`NumberPolariztion` (e.g., Beam5X). The final two columns of this binary table extension hold the beam offsets and are either BeamOff_XEL or BeamOff_EL. The length of the returned data vector are equal to the length of beam weight columns, though only values with indices between 0 and the total number of beams contain meaningful data; the remainder of the vector is padded with zeros. Note that the beam for which an offset value is associated with corresponds to the index within the returned data vector; that is, the 0th indexed beam offset value corresponds to the 0th beam, the 1st indexed beam offset value corresponds to the 1st beam, etc. The generated FITS files have the naming convention `w_PROJECT_BANKNUM.FITS`, and are stored in the directory `weight_files` that is created within the directory where the script is ran. The only argument is the directory path to the binary files. An example call that successfully generates complex weights in the form of FITS files is:

```
$ weightFiller /lustre/projects/flag/AGBT18B_358_03/BF/mat
```

3.2 PAF_ Filler.py

As stated above, this script plays the role of the 'main' program that ultimately drives the generation of the beamformed SDFITS file. An example of a syntactically correct call is:

```
$ ipython --c="run PAF_Filler.py /home/gbtdata/AGBT17B_360_01
↪ /lustre/project/flag/AGBT17B_360_03/BF/weight\_files/ 1420.405e6
↪ 1450e6 -b 2018_08_01:00:00 2018_08_01:52:00 -o NGC891 -g
↪ 2018_08_01:05:00 2018_08_01:06:00 2018_08_01:07:00 -m 1 3"
```

The first and second inputs are paths to your GBO data project (that will generally begin with `/home/gbtdata/`) and the path to the FITS files that contain the complex weights. It is recommended that all weight FITS files are contained within this specified directory. The third and fourth input are respectively the rest and LO frequency in units of Hz. The rest and LO frequency available in ancillary FITS files generated by the telescope corresponds to the topocentric central frequency of the full 150 MHz band, so it is recommended the user sets these values explicitly for now. These paths and frequency values are the only required user inputs. Space delimited list of timestamps after the '`-b` flag represent bad scans that, for whatever reason, are not required for your subsequent analysis. If an indexing error is returned while constructing the final binary SDFITS table, it's likely that one needs to specify a bad time stamp. If the '`-o`' flag is specified, only scans associated with the listed objects will be processed. In this example, only the mapping scans for NGC891 will be processed. The timestamps proceeding the `the -g` flag will be the only scans for which SDFITS files will be created. Finally, the integers listed after the `-m flag` denote only the beams one wishes to create SDFITS for. If any of these options are missing, all available timestamps will be processed (e.g., ALL timestamps from an associated session will be processed if no flags are listed after the project ID; or all timestamps associated the explicitly listed observed objects will be processed if no other flags are provided).

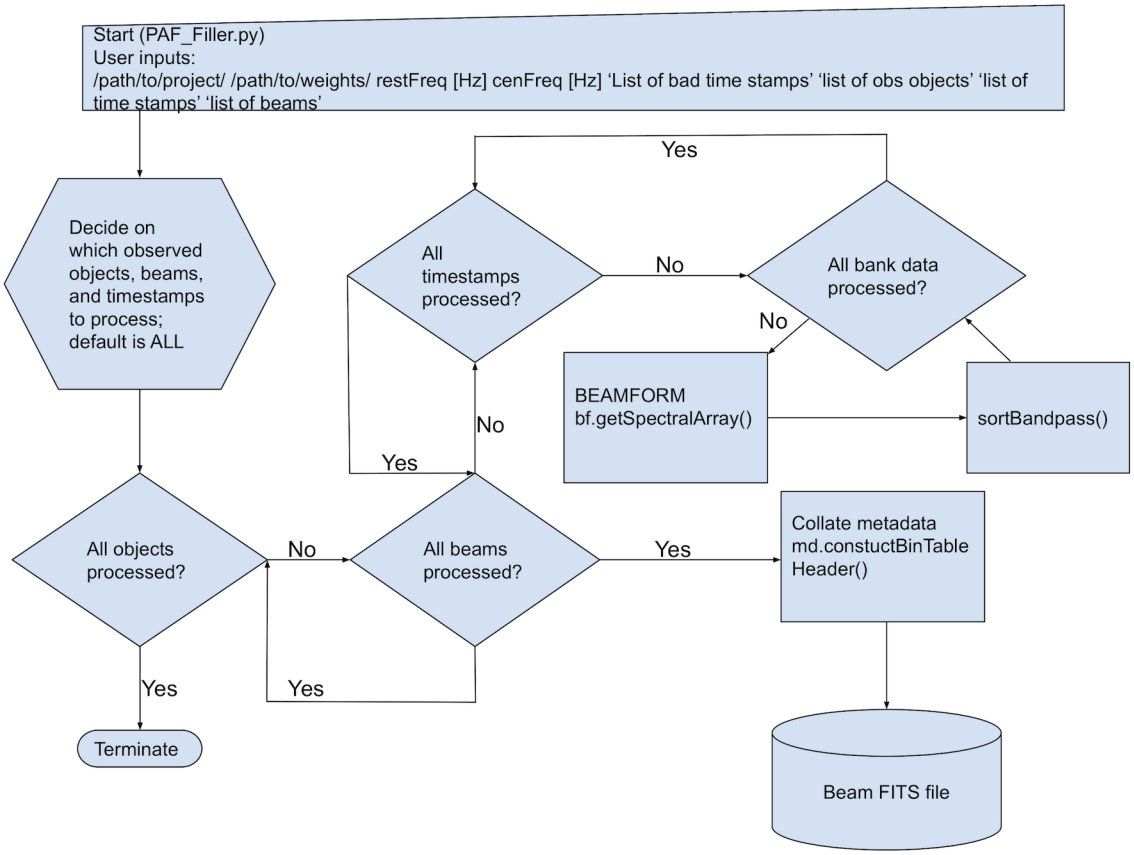


Figure 3: Logic flow of the `PAF_Filler.py` script, which contains the 'main' function that drives the beamforming generation of an SDFITS file per beam per observed object.

Figure 3 illustrates the logic flow of the program. After parsing the flags to determine what sources and beams the user wishes to fill, the program collates all the associated scans of an observed source (e.g. all of the scans that targeted NGC891) for lowest numbered requested beam (default is to begin boresight beam numbered 0). For each processed scan, a global data buffer for the XX and YY polarizations is created with dimensions integrations×frequency channels to hold the beamformed data. As described in the previous section, the backend creates 20 separate bank files that contain 1/20th of the total bandwidth. One-by-one, these bank FITS files are unpacked and passed to beamformer object created by `beamformingModule.py` (see 3.3) through the object function `getSpectralArray()`. This function within the beamformer module applies the complex weights to create beamformed spectra (see Section 3.3). The beamformed XX and YY bandpasses are returned and, based on the observing mode (i.e., CALCORR or PFBCORR mode), the frequency channels contained in the 1/20th bandpass chunk are sorted into the global data buffers though the internal `sortBandpass()` method. Once all scans for the current beam have been beamformed and sorted, the global data buffers and other pertinent information (i.e., a list of scan time stamps) are passed to a method called `sortLists()`. Because the same observed object could have been observed in both CALCORR and PFBCORR modes — such as when a standard flux calibrator source is used for flux and beamformer weight calibration — the data contained in the global buffers must be sorted in order to generate separate binary FITS tables contained within an individual Header Data Unit (HDU) object that can be appended unto the same HDUList, a list-like collection of HDUs.

After `sortLists()` determines how many individual HDU objects to generate based on the number of mode changes assocaited with a given object, the collection of associated scans are passed to the metadata object (see Section 3.4 through the `constuctBinTableHeader()` object function. Finally, once all FITS binary tables have been constructed for a given beam, they are appended to the same HDUList object and a FITS file (in SDFITS format) is written to disk in whichever directory the call to `PAF_Filler.py` was made. The process then repeats until all beams for all observed objects are processed.

3.3 `beamformingModule.py`

This python object module performs a majority of the processing. An instance of this object is initialized in by passing the path of the raw data and weights. Other default attributes of the object that get defined upon initialization are a reference vector that holds the indices to sort the order of the raw correlations from their 1D format to the covariance matrix format (see Figure 1) and the project ID. The method that drives the transformation from raw correlations between dipoles to (un-calibrated) beamformed bandpasses is `getSpectralArray()`. `PAF_Filler.py` supplies this method with the name of the FITS file that is currently being processed, the raw data array, the beam number that is being processed, and the XID.

The first action is to open and sort the complex weights using the internal `getWeights()` method, which requires the XID, beam, and number of frequency channels (i.e., 25 or 160 depending on CALCORR or PFBCORR). This particular method opens the associated weight FITS file and formats the weights to be organized in a 2D

numpy array of complex type with the rows representing the 25 coarse frequency channels, while the columns represent the correlations of the 40 dipoles.

Once the complex weights are in the correct format, a loop to process individual integrations is initiated. The internal method getCorrelationCube() will sort the raw correlations for each integration to be in the format of a 3D numpy array of complex type with rows and columns both representing the correlations between dipoles and the third axis containing these correlation matrices for the associated frequency channel. More explicitly, for a 2D plane of this data cube, the first element of the first column holds the auto-correlation (ρ) between the first dipole (dp) $\rho_{dp1,dp1}$, the second element is the cross-correlation between dipoles 1 and 2, or $\rho_{dp1,dp2}$, the third holds $\rho_{dp1,dp3}$, ..., and the last holds $\rho_{dp1,dp40}$. The sorting is performed by first grabbing a chunk of 2112 total correlation pairs (every frequency channel regardless of mode will ALWAYS have 2112 correlation pairs) from the input raw 1D data vector and using an index look-up vector located in SpectralFiller/misc/gpuToNativeMap.dat that is generated with the script SpectralFiller/utils/GpuToCovarMatrix_Map.py. Since the correlations are redundant, the corresponding transposed element in a row is the simply the conjugate value of the column value. The final returned cube will have dimensions of 40×40×channels, where channels will either be 25 or 160, depending on whether we are in CALCORR or PFBCORR mode. Recall two important aspects: (1) irrelevant correlations caused by xGPU limitations are thrown away at this stage; (2) some rows and columns be contain zeros as they correspond to two unused data streams.

Once a correlation cube has been constructed for the integration being processed, the method, getSpectralArray(), will loop over each frequency channel (ν) and call the internal method processPol() and feed it the 2D plane of dipole correlations ($\boldsymbol{R}$) and associated 1D weight vector ($\boldsymbol{w}$). This method applies

$$P(\nu) = \boldsymbol{w}^{\boldsymbol{H}} \cdot \boldsymbol{R} \cdot \boldsymbol{w}, \tag{1}$$

where the $\boldsymbol{H}$ superscript denotes the Hermition operator. The returned value, $P(\nu)$, is the the beamformed power value at frequency ν. Recall that for the CALCORR mode, the channels are *not* contiguous, and will be sorted in PAF- _Filler.py. If we are in PFBCORR mode (spectral line mode), the corresponding set of five complex weight values are selected for processing each chunk of 32 contiguous fine channels. Regardless of mode, a 2D array of raw, beamformed spectra is returned to PAF_Filler.py with the shape integrations×frequency to be subequently sorted into global data buffers that hold the power as a function of frequency channel.

3.4 metaDataModule.py

The metaDataModule.py is a python object that contains several internal methods to collate all associated metadata saved in ancillary FITS files during a given observing session. For example, the sky positions of the antenna must be associated with individual integrations for imaging after data calibration. The instantiation of the object occurs after all bank FITS files associated with an observed object and beam have been processed by the beamforming module. The necessary inputs to initialize this object are:

1. project name
2. path to the weight FITS files
3. a list of time stamps
4. rest frequency (in Hz)
5. topocentric LO frequency (in Hz)
6. a list of all bank FITS files
7. a list of the number of banks associated with a given time stamp
8. global XX data buffer
9. global YY data buffer
10. beam number
11. boolean flag denoting whether current mode is CALCORR or PFBCORR
12. a boolean flag denoting whether to generate a primary HDU and HDUList, or just a HDU object

The method, `constructBinTableHeader()`, drives the creation of the full binary FITS table that gets passed back to `PAF_Filler.py` before writing the final SDFITS file to disk.

The philosophy of this module is very much object oriented. Essentially, the `metaDataModule` object contains several attributes (e.g., a `valueArrr`, or FITS Column) that will be updated while the metadata is collected for each SDFITS keyword processed. Once all of the metadata have been collected for a given SDFITS keyword, the constructed column contained in the `metaDataModule` object is added to the a list of FITS columns that make up the full binary FITS table.

The first step taken by `constructBinTableHeader()` is the creation of a blank primary FITS header if this is the first call of the method from `PAF_Filler.py`. If this is a subsequent call, this step is skipped. A list of SDFITS keyword located in `SpectralFiller/misc/sdKeywords.txt` is then read in. Each keyword has an associated parameter (e.g., the keyword 'TTYPE1' is associated with the 'OBJECT' parameter) contained within the dictionary, `keyToParamDict`. These parameters are associated with specific ancillary FITS files generated throughout a GBT observation. For example, the OBJECT parameter is queried by this package in the GO FITS file written out by Astrid. A dictionary called `funcDict` associates a parameter to an an internal `metaDataModule` object method that collects the necessary information. The internal object methods to collect the metadata for specific parameters are `getGOFITSParam()`, `getArbParam()`, `getLOFITSParam()`, `getSMKey()`, `getAntFITSParam()`, and `getModeDepParams()`.

All of these internal methods follow similar logic when collecting the metadata associated with a parameter: (1) a parameter is passed to the associated metadata collection method mentioned above; (2) each file contained within the module `fileList`

attribute is read in and the number of scans is calculated; this is important since each row in the final SDFITS binary table is associated with specific integration and scan number; (3) the internal method, `initArr()`, is called to initialize a global data buffer that stores all necessary information as the FITS column is filled in. Once all of the necessary metadata for a specific parameter has been collated, the object's attributes `Column.param`, `Column.comment`, and `Column.valueArr` are filled in and returned to the
`constructBinTableHeader()` function. Once returned, these object attributes are added to a list of FITS column objects and reset to process the next SDFITS keyword.

There are several more steps to take once all of the metadata associated with all SDFITS keywords has been collected and stored within individual FITS columns. Firstly, the beam offsets are stored in the engineering Horizontal coordinate frame (i.e., Cross-Elevation and Elevation), as opposed to the preferred Equatorial coordinate frame. The internal method,
`offsetCorrection()` is called with the current state of the binary table HDU passed to it.

The `pyslalib` package, a python wrapper for the `slalib`[6] library, is utilized to apply the beam offsets. The antenna positions recorded during the observation are collected, in addition to required other metadata such as the humidity, temperature, and LST. The calculation begins by converting the recorded J2000 coordinates from the Antenna FITS file to the geoapparent reference frame (center of Earth) using the `pyslalib.sla_map` method. The observed hour angle is then calculated from the LST and provided to `pyslalib.sla_e2h` along with the geoapparent Declination and latitude of the GBT (38.4331294°) to convert to local horizontal coordinates.

Once in horizontal coordinates of Azimuth (Az) and Elevation (El), the beam offsets ($Az_{\rm off}$ and $El_{\rm off}$) and refraction correction are applied using the equations

$$El' = El - \Delta El_{\rm off} + n \tag{2}$$

$$Az' = Az - \frac{\Delta Az_{\rm off}}{\cos(El')}, \tag{3}$$

where n is the atmospheric refraction correction available in the Antenna FITS file. The inverse cosine factor accounts for the conversion between cross-elevation and Az. The coordinate transformation is then complete if the observations were taken in engineering coordinates (e.g., for a calibration grid). Otherwise, the new corresponding J2000 values are computed by first computing the topocentric equatorial coordinates through `pyslalib.sla_h2e`, changing reference frame from the GBT to geocentric via `pyslalib.sla_oap`, before finally using `pyslali- b.sla_amp` to precess the positions back to the mean 2000.0 epoch. If the observations were taken the Galactic coordinate system, these modified J2000 values are then converted to the corresponding Galatic longitude and latitude values.

As of 2019, no association between the local oscillator (LO) and GBO IF system exists for FLAG observing. This means the frequencies recorded for any observation are in the topocentric reference frame. Now that the J2000 values of each beam

[6] `http://star-www.rl.ac.uk/docs/sun67.htx/sun67.html`

pointing center are available, a proper Doppler correction can be calculated. *Currently the Doppler correction will be applied such that the IF frequencies will be in the Heliocentric reference frame with the OPTICAL velocity definition.* Translations to other reference frames and velocity definitions (e.g., LSRK in the RADIO definition) can be performed in GBTIDL during data reduction. The current binary table HDU is passed to the internal method, `radVelCorrection()`, for the correction. This function makes use of `RadVelCorr.py`, which is an edited version of Frank Ghigo's radial velocity correction calculator[7]. Within this method, the updated J2000 values, as well as the corresponding Universal Time and Date, are passed into the `correctVel()` method. The radial correction is then returned. From this value, the correct central frequency is computed and updated within the binary table HDU. Once all values for each scan and integration are computed, the CRVAL1, OBSFREQ and VELDEF parameters are updated before returning the binary table HDU to the main `constructBinTableHeader()` method. Once all corrections to the spatial and spectral coordinates have been made, the binary table HDU is combined with the primary HDU and returned to `PAF_Filler.py`.

4 Installation

This package is designed to access raw data stored on FLAG's dedicated `lustre` file system and other ancillary FITS files. Therefore, on a GBO machine the paths `/users/npingel/FLAG/SpectralFiller/` and `/users/npingel/FLAG/SpectralFiller/utils` need to be appended to one's `PYTHONPATH` environment variable. Note also that the most recent build has been tested for python versione 3.5 and up. The non-standard python package dependencies are `numpy`, `astropy`, and `pyslalib`.

5 Observing Scripts

Scripts related to observing are described in this section. Example configuration, calibration, and science observing scripts are presented in Figures 4- 7.

5.1 `CalcObsTime.py`

The parameters of the PFBCORR observing mode used for spectral line mapping with FLAG are not currently available in the GBT Mapping Calculator[8]. When provided with the map size in degrees, final spectral channel width in kHz, desired rms per spectral channel in K, and scan direction (Long or Lat) this script will calculate the necessary mapping time.

The mapping time is calculated by first determining the number of beams ($N_{\rm beams}$) contained within the user provided map area by taking the ratio of the map area in deg^2 and area of a typical GBT beam at 1.4 GHz (0.026 deg^2). Based on the provided scan direction, the number of total scans ($N_{\rm scans}$) and integrations ($N_{\rm ints}$) in one map area are then determined, assuming data are saved to disk every 1.67′ to ensure adequate spatial Nyquist sampling and rows/columns are spaced every 3′. Assum-

[7] `http://www.gb.nrao.edu/GBT/setups/radvelcalc.html`

[8] `http://www.gb.nrao.edu/~rmaddale/GBT/GBTMappingCalculator.html`

ing the standard FLAG integration time ($t_{\rm int}$) time of 0.5 s for PFBCORR mode, the total time to complete one map is

$$t_{\rm map} = N_{\rm scans} t_{\rm scans}, \tag{4}$$

where $t_{\rm scans} = N_{\rm ints} t_{\rm int}$. The total signal integration time ($t_{\rm On}$) per map is given by the ratio of $t_{\rm map}$ to $N_{\rm beams}$. While mapping, we generate a reference spectrum at the edge of our map, where there is generally no signal, by using the first four and last four integrations of a given mapping scan. Using the edge of the map as a reference position is an advantage because we are able to obtain a reference spectrum for each R.A./decl. scan without sacrificing telescope time to slew off source. The time spent on the reference signal is therefore $t_{\rm Off} = 8 t_{\rm int} = 4$ s. The effective integration time per map is therefore

$$t_{\rm eff,map} = \frac{t_{\rm On} t_{\rm Off}}{t_{\rm On} + t_{\rm Off}}. \tag{5}$$

The ideal radiometer equation in terms of the SEFD and adjusted for 7 identical formed beams is

$$\sigma[\rm Jy] = \frac{\rm SEFD}{\sqrt{\rm N_{pol} N_b \Delta\nu t_{eff}}}, \tag{6}$$

where $N_{\rm pol}$ is the number of polarizations (in our case 2), $N_{\rm b}$ is the number of formed beams (typically 7), and $\Delta\nu$ is the width of a typical frequency channel in Hz. Assuming a typical SEFD of 10 Jy (Pingel et al. 2019; in prep), gain of 1.86 K/Jy (Pingel et al. 2018), and target sensitivity, the script solves Equation 6 for the necessary $t_{\rm eff}$. The ratio between the necessary $t_{\rm eff}$ and $t_{\rm eff,map}$ determines the total number of maps an observer must make to get down their desired noise level per spectral channel. The total mapping time is therefore the total number of maps multiplied by $t_{\rm map}$.

For example, if one wishes to propose to map a 2x×2 deg^2 region around a nearby external galaxy down to an rms level of 1 mK (roughly a 5σ column density detection limit of $\sim 10^{17}$ cm^{-2}) over a 20 km s^{-1} line after smoothing to a final velocity resolution of 5 km s^{-1}), the necessary mapping time with FLAG is 146.9 hours, excluding overhead.

```
$ ipython calcObsTime.py 2 2 0.002 24.414
$ Number of beams per map: 153.47
$ Time per map (excluding overhead): 1476.00 [sec]
$ Total effective integration time in a single map: 2.83 [sec/beam]
$ Necessary effective integration time: 1012.18 [sec]
$ Total number of maps: 358.29
$ TOTAL OBSERVING TIME (EXCLUDING OVERHEAD): 146.90 [hours]
```

5.2 `GpuToCovMatrix.py`

The raw covariance values between dipoles are in a non-standard order and stored in a 1D data array (see Section 2.1). The script, `GpuToCovMatrix.py`, will generate a text file called `gputoNative.txt` that contains, for a single spectral channel, the indices that map the raw covariance values to their position in the covariance matrix (see Figure 1. For example, the value at index 3 (0-index based) in this generated data vector is

the covariance to be placed at the position of row 3, column 0 in the covariance matrix. Several filling and data monitoring scripts read a version of this text file located in the `SpectralFiller/utils` directory. The script will print to the terminal the indices of the autocorrelations for each data channel. This output is useful an observer needs to update the mapping between DDL data channel blades and actual dipoles. There are required inputs from the user. An example call and terminal output:

```
$ ipython GputToCovMatrix.py
$ Data Channel: 1, Index: 0
$ Data Channel: 2, Index: 3
$ Data Channel: 3, Index: 8
$ Data Channel: 4, Index: 11
$ ...
$ Data Channel: 40, Index: 839
```

5.3 Configuration

Because there is currently no true manager for FLAG, the communication and configuration of the LO must take place through Astrid scripts. The configuration shown in Figure 4 sets the LO to be at a topocentric frequency of 1450 MHz with an associated test-tone at 1500 MHz at a level of -50 dB. This specific setting effectively turns the test-tone off. For bit/byte/word locking, the test-tone level should be set to a level of 0 dB. This can be done by changing the variable 'Tonelevel'. The rest frequency and test-tone frequency can be set by changing the variables 'RestFreq' and 'ToneFreq', respectively. Note that MHz is the assumed units for these variables. The two functions, 'setRestFreq' and 'setTestTone', communicate with the LO manager to set the necessary parameters.

5.4 Calibration Scripts

5.4.1 7-PtCal

Figure 5 provides an example script that performs a 'seven-point calibration', which allows for an observer to derive weights for seven distinct beams. The beam power pattern consists of a central beam is surrounded by six outer beams in a hexagonal pattern with the beam responses overlapping at the half-power points. The beam widths are assumed to be $9.1'$. In general, the beams are labeled such that the boresight is beam '0', beam '1' is the upper left beam, and the subsequent beam numbers increase in a clockwise fashion. A reference scan is performed first that is -2 degrees away in cross-elevation from the boresight and at the same elevation as beam 5. The calibration source (generally one of the standard GBT flux L-Band flux calibators) is then centered within the boresight beam; next, the telescope will dwell at the center of beams 2, 3, 4, 5, 6, and 1; the final scan is another reference position again -2 deg away from the boresight but now at the same elevation of beam 1. Dwelling the telescope at the desired beam centers will characterize the response of each beam to the calibrator in that direction to facilitate the derivation of the complex beamformer weights. The length of the dwell can be set by setting the variable 'minVal' to some fraction of minutes.

```
import time

# function to set restfreq in LO1A
def setRestFreq(freq, level):
    LO1A = 1
    LO1B = 0
    LOvalues = {
        'loConfig'           : 'TrackA_BNotUsed',
       "subsystemSelect,LO1A" : LO1A,
       "subsystemSelect,LO1B" : LO1B,
        'receiver'          : 'NoiseSource',
        'restFrequency'       : freq,
        'restFrequency_A'   : freq,
        'loPowerLevel'         : level,
        'ifCenterFreq'          : 0,
        'ifCenterFreq_A'      : 0,
        'S9'                  : 1,
        'S2'                  : 'thru',
        'S4'                  : 1,
        'S5'                  : 1,
        'S1'                  : 'cross',
        'S3'                  : 3,
        'S11'                 : 'cross',
        'S12'                 : 4,
        'S13'                 : 1
        }
    SetValues("LO1", LOvalues)
    SetValues("LO1", {"state": "prepare"})

# function to test-tone in LO1B
def setTestTone( freq, level) :
    # set boolean for subsystem selection
    LO1A = 0
    LO1B = 1
    LOvalues = {
        'loConfig' :  'TrackA_TToneB',
        'subsystemSelect,LO1A' : 0,
        'subsystemSelect,LO1B' : 1,
        'receiver'                  : 'NoiseSource',
        'testToneFreq'          : freq,
        'restFrequency_B'   : freq,
        'testTonePowerLevel' : level,
       'ifCenterFreq'          : 0,
        'ifCenterFreq_A'      : 0,
        'S9'                  : 1,
        'S2'                  : 'thru',
        'S4'                  : 1,
        'S5'                  : 1,
        'S1'                  : 'cross',
        'S3'                  : 3,
        'S11'                 : 'cross',
        'S12'                 : 4,
        'S13'                 : 1
  }
    SetValues("LO1", LOvalues)
    SetValues("LO1", {"state": "prepare"})

# rest frequency and level [MhZ, dBm]
RestFreq=1450
LO1level=-14.0

# test tone frequency and level [MHz, dBm]
ToneFreq=1500
Tonelevel=-50.0

# minimum PAF config
PAF="""
receiver="RcvrArray1_2"
restfreq=1450.0
beam = 'B1'
"""

# configuring PAF
Comment('Configuring PAF')
Configure(PAF)

Comment('Configuring LO to set rest frequency to 1450 MHz')
# set LO1A to the rest frequency. with the LO1B subsystem disabled
setRestFreq(RestFreq, LO1level)


Comment('Configuring LO to set test tone')
# set L01B to generate the test tone, with the LO1A subsystem disabled
setTestTone(ToneFreq, Tonelevel)
```

Figure 4: A summary of the structure of the binary files that contain the complex beamforming weights.

```
# 01/15/18
# Script to obtain seven discrete pointings to derive beamforming weights.
# Two reference offs (-2.0 in cross-el at at the same elevation as the lower/upper
beams
# bracket the seven discrete pointings.
# written by: Nick Pingel
import math

# Catalogs
Catalog(fluxcal)
Catalog(lband_pointing)

# minutes for each track
minVal = 1.

#select calibrator source
src = '3C295'
# function to compute Cross-El/Eloffsets
def computeObsOffsets(beamNum):
    #beam width
    beamWidth = 9.1/2/60 # deg

    # possible offset angles
    offAngle = 60 # deg
    offAngleRad = math.radians(offAngle)

    # determine based on beam number
    if beamNum == 1:
        AzOffSet = math.cos(offAngleRad) * beamWidth
        ElOffSet = math.sin(offAngleRad) * beamWidth
    elif beamNum == 2:
        AzOffSet = beamWidth
        ElOffSet = 0
    elif beamNum == 3:
        AzOffSet = math.cos(offAngleRad) * beamWidth
        ElOffSet =(-1)*math.sin(offAngleRad) * beamWidth
    elif beamNum == 4:
        AzOffSet =(-1)*math.cos(offAngleRad) * beamWidth
        ElOffSet =(-1)*math.sin(offAngleRad) * beamWidth
    elif beamNum == 5:
        AzOffSet = (-1)*beamWidth
        ElOffSet = 0
    elif beamNum == 6:
        AzOffSet = (-1)*math.cos(offAngleRad) * beamWidth
        ElOffSet = math.sin(offAngleRad) * beamWidth

    print 'Current Offset: Cross-El = %s, El = %s'%(AzOffSet*60, ElOffSet*60)
    return AzOffSet, ElOffSet

def computeRefOff(refPoint):
    #beam width
    beamWidth = 9.1/2/60 # deg

    # possible offset angles
    offAngle = 60 # deg
    offAngleRad = math.radians(offAngle)
    if refPoint == 1:
        AzOffSet = -2.0
        ElOffSet =(-1) * math.sin(offAngleRad) * beamWidth
    elif refPoint == 2:
        AzOffSet = -2.0
        ElOffSet = math.sin(offAngleRad) * beamWidth
    print 'Current Offset: Cross-El = %s, El = %s'%(AzOffSet*60, ElOffSet*60)
    return AzOffSet, ElOffSet

#Slew to source
```

Figure 5: An example observing script of a discrete seven-pointing calibration.

```
Slew(src)

#determine offset for first reference off
crossElOffSet, ElOffSet = computeRefOff(1)

#track reference off
Track(src,None,minVal*60, fixedOffset=Offset('Encoder', crossElOffSet, ElOffSet,
cosv=True))

#Track boresight
Track(src, None, minVal*60)

# determine offsets for Beam 1
crossElOffSet, ElOffSet = computeObsOffsets(1)

# Track Beam 1
Track(src,None,minVal*60, fixedOffset=Offset('Encoder', crossElOffSet,ElOffSet,
cosv=True))

# determine offsets for beam 2
crossElOffSet, ElOffSet = computeObsOffsets(2)

# Track Beam 2
Track(src,None,minVal*60, fixedOffset=Offset('Encoder', crossElOffSet,ElOffSet,
cosv=True))

# determine offsets for beam 3
crossElOffSet, ElOffSet = computeObsOffsets(3)

# Track Beam 3
Track(src,None,minVal*60, fixedOffset=Offset('Encoder', crossElOffSet,ElOffSet,
cosv=True))

# determine offsets for beam 4
crossElOffSet, ElOffSet = computeObsOffsets(4)

# Track Beam 4
Track(src,None,minVal*60, fixedOffset=Offset('Encoder', crossElOffSet,ElOffSet,
cosv=True))

# determine offsets for beam 5
crossElOffSet, ElOffSet = computeObsOffsets(5)

# Track Beam 5
Track(src,None,minVal*60, fixedOffset=Offset('Encoder', crossElOffSet,ElOffSet,
cosv=True))

# determine offsets for beam 6
crossElOffSet, ElOffSet = computeObsOffsets(6)

# Track Beam 6
Track(src,None,minVal*60, fixedOffset=Offset('Encoder', crossElOffSet,ElOffSet,
cosv=True))

#determine offset for last reference off
crossElOffSet, ElOffSet = computeRefOff(2)

#track reference off
Track(src, None, minVal*60, fixedOffset=Offset('Encoder', crossElOffSet,ElOffSet,
cosv=True))
```

Figure 5: An example observing script of a discrete seven-pointing calibration (continued from previous page).

5.4.2 Calibration Grid

In some cases, it may be pertinent to characterize the response of the array over a larger field-of-view. In these cases, a the calibration grid can be performed. Generally, a 30×30 $\deg^2$ area around a standard point-source calibration source is mapped utilizing a RaLongMap procedure with the coordinates set to Encoder (i.e., horizontal frame). The rows are spaced every 1\10th of a beamwidth (∼0.9 arcminutes) for a total of 34 rows. Likewise, the scanning speed is set such that integrations are dumped every 1\10th. A total of six reference scans are performed throughout the procedure such that three are evenly spaced in elevation on each side. The entire procedure takes between 35-40 minutes to complete. The backend mode should be set to CALCORR for when running these observing scripts.

5.5 Mapping Scripts

The final observing script in Figure 7 demonstrates a typical science on-the-fly map an observer can make around an extended source with the backend in PFBCORR mode. It is very similar to the usual maps one would make with the traditional single-pixel feed. In this particular example, the map size is 2×2 deg (in J2000) with rows spaced every $3'$ (for a total of 41) and the time to map one row set to ∼36 seconds. The time to map a row is set as such to account for the backend default integration time of 0.5 s in PFBCORR mode. This ensures data is dumped every 1.67 arcminutes to be adequately spatially Nyquist sampled. This map will take about 30 minutes (including overhead) to complete.

6 Data Monitoring

This package comes with several utility scripts contained within the /utils/ directory that aid the analysis, reduction, and data monitoring of FLAG spectral line observations. This section summarizes the functionality and graphical output of available data monitoring scripts.

6.1 `plotCovariance.py`

The order of the covariance values between the different dipoles in the raw 1D data vector is non-trival (see Section 2.1). The script `plotCovariance.py` can be used to reorder the raw covariance values to display them in a more intuitive matrix form, where each row/column value is the time-averaged covariance value between individual data channels for a single scan. An example output is shown in Figure 8.

The autocorrelations are clearly visible along the diagonal. This plot is also useful to identify potential dead elements. For example, the blank row column demonstrates that data channel 21 is not connected. The user must provide the GBO project ID and scan timestamp. An example call is

```
$ ipython plotCovariance.py AGBT18B_358_03 2019_03_14_11:01:12
```

```
#continuous calibration grid with reference
#01/15/18
# written by: Nick Pingel
import numpy as np

# change this if necessary
Catalog(fluxcal)

# Set up custom scan input parameters
# source
src='3C295'

#slew
Slew(src)

# total map size and start parameter
RaSize=0.5
DecSize=0.5
startrow=1

# optimal row offset of 1/10 L-Band beam FWHM.
rowSpacing = 9.1/10/60 # in deg

# 32.97 sec scan time for desired scan rate of  1/10 L-Band beam FWHM
int = 30/(9.1/10)

# number of rows
nrows = 30
# perform RaLongMap, but stop after every five rows (thus alternativing
sides) and perform 10 second Track
# of an 'Off' pointing. The six total offs will be evenly distributed along
elevation
# with three on each side.

# set initial elevation offet of -0.25 deg
initElOffset = -0.25

for i in range(1,34,5):
    startRow = i #update startRow
    stopRow = i+4 #update endRow

    print('Starting Row: ' + str(startRow))
    print('Ending Row: ' + str(stopRow))
    RALongMap(src,Offset("Encoder",RaSize,0.0,cosv=True),
                Offset("Encoder",0.0,DecSize,cosv=True),
                Offset("Encoder",
0.0,rowSpacing,cosv=True),int,start=startRow, stop=stopRow)

    # if i is greater than 30, than we've already finished the 'off's, else
    # perform Track Offset -2.0 of arc in Az whereever previous scan finished
    if i < 30:
        # get current location (end of row)
        currLoc = GetCurrentLocation('Encoder')
        Comment('Going to reference position')

        # if stopRow is odd, the previous row finished at a postive offset,
        # requiring a positive cross-el offset for subsequent scan. If even,
        # a negative offset is needed
        if stopRow % 2 == 1:
            crossElOffset = 2.0
```

Figure 6: An example observing script of a calibration grid.

```
            elOffset = initElOffset + (stopRow-1)*rowSpacing
            print 'Cross-El offset: %s' % crossElOffset
            print 'El offset: %s' % elOffset
        else:
            crossElOffset = -2.0
            elOffset = initElOffset + (stopRow-1)*rowSpacing
            print 'Cross-El offset: %s' % crossElOffset
            print 'El offset: %s' % elOffset

        # setting cosv=True in offset object ensures we are in cross-el
        Track(src,None, scanDuration=10, fixedOffset=Offset('Encoder',
crossElOffset, elOffset,cosv=True))
        Comment('Continuing Grid')
```

Figure 6: An example observing script of a calibration grid (continued from previous page).

```
#DecLatMap for PAF
#01/15/18 Nick Pingel

Catalog(fluxcal)
Catalog(lband_pointing)
Catalog("/users/npingel/GBT17B-400/gbt17b_360.cat")

src='NGC6946'
inte = 120/3.33 # 2 deg at a scan rate of 3.33 arcmin/sec
rasize=2.0
decsize=2.0
startrow=1

#DecLatMap observation around source
Slew(src)

DecLatMap(src,Offset("J2000",rasize,0.0,cosv=True),
                  Offset("J2000",0.0,decsize,cosv=True),
                  Offset("J2000",
0.05,0.0,cosv=True),inte,start=startrow)
```

Figure 7: An example observing script of a on-the-fly map.

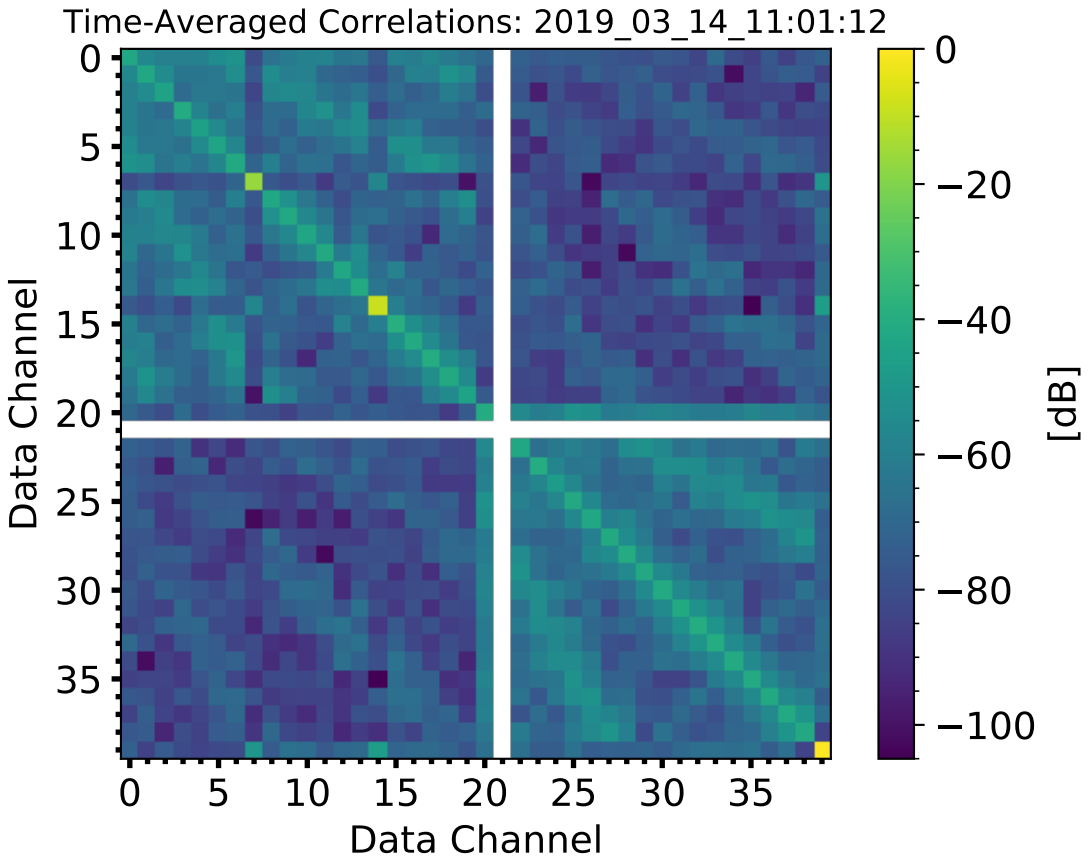


Figure 8: Example graphical output of plotCovariance.py. The time-averaged autocorrelations are clearly visible along the diagonal of the matrix. The blank row column correspond to the unconnected data channel 21.

6.2 `plotDipolePower.py`

This particular script plots the total power (in units of dB) as a function of scan time for each of the 19 dipoles (and polarization) in 5×4 panel plot.

There are only two user inputs required for this script with the first being the GBO project ID (e.g., AGBT1B8_4358_03) and second being the associated time stamp for the scan (e.g., 2018_03_14_10:14:40). An example output plot is shown in Figure 9. The black number in the top right corner denotes the dipole number, and the blue/orange solid lines denote total power from the YY/XX polarization. The particular scan being plotted came from a scan where the telescope was slewed in the cross-elevation direction across a strong calibrator source. There is a clear response in both polarizations from the inner dipoles (i.e., 1, 3, 4, 6, 7, 11, and 17), while outer dipoles such as dipole 13 showed little response. This plot is useful to ensure the live elements behave as expected, as well as checking to see if strong sources are detectable in individual dipoles. Note that no plot is inherently saved to disk. If the user wishes to save the plot, they must do so utilizing the python plotting GUI.

6.3 `plotBandpass.py`

This script allows the user to see the time averaged bandpass of a scan in units of raw counts for a specific dipole. In essence, this is plotting the autocorrelation for the indicated dipole. This scripts reads in the associated bank FITS files, sorts the channels based on XID, and averages over all integrations to produce the mean bandpass as a function of topocentric frequency. The input to this script are respectively the project ID, time stamp to process, and dipole element (integer 1 to 19).

An example output is shown in Figure 10. At the time of these observations, the PFB mode suffered from 'scalloping', where power at the edge of each five coarse channels set dropped by about 3 dB creating the distinct drop in counts seen across the bandpass. These are expected, and should be mitigated in a future implementation of the PFB mode. In general, an observer should keep an eye on the overall bandpass shape, the presence of Galactic HI at 1420.406 MHz, and the presence of RFI spikes. An example call to create the output in Figure 10 is:

```
$ ipython plotBandpass.py AGBT16B_400_14 2017_08_06_16:05:19 1
```

The Galactic HI line is clearly visible in both polarizations. Additionally, we can see RFI spikes near 1426.5 MHz and 1432 MHz in YY and XX, respectively. Recall that this is the raw power from an individual dipole; the subtle bandpass structure should be mitigated once beamforming weights are applied.

7 Reduction/Analysis Scripts

This section summarizes scripts that can be used post-observing to reduce, calibration, and image the raw beamformed SDFITS files. The final subsection contains a log of end-to-end commands used to generate a cube of M81 taken during the GBT18B_358.

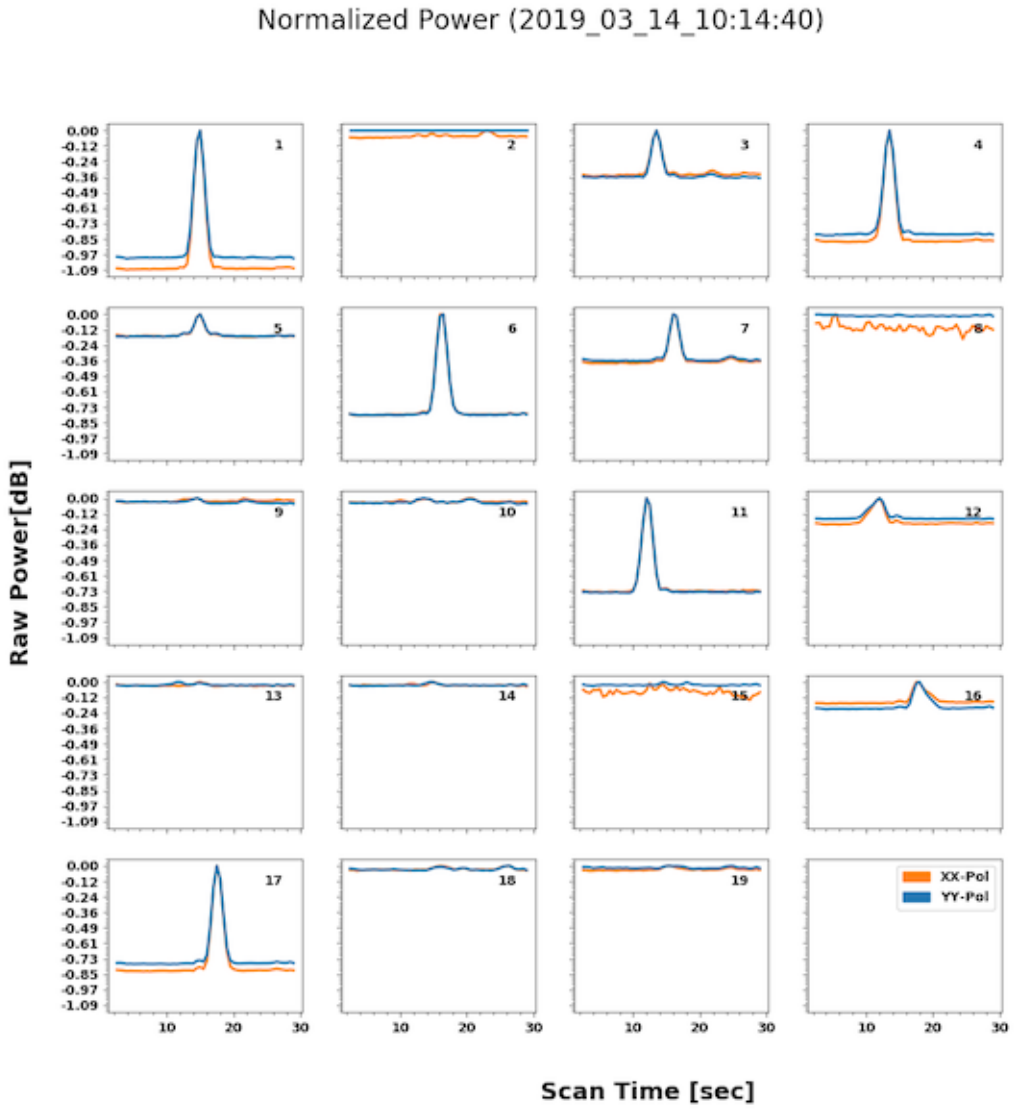


Figure 9: Example graphical output of plotDipolePower.py. Each panel represents the total power of a the associated dipole (black number) as a function of scan time for user provided scan. The blue and orange solid lines respectively represent the YY and XX polarizations.

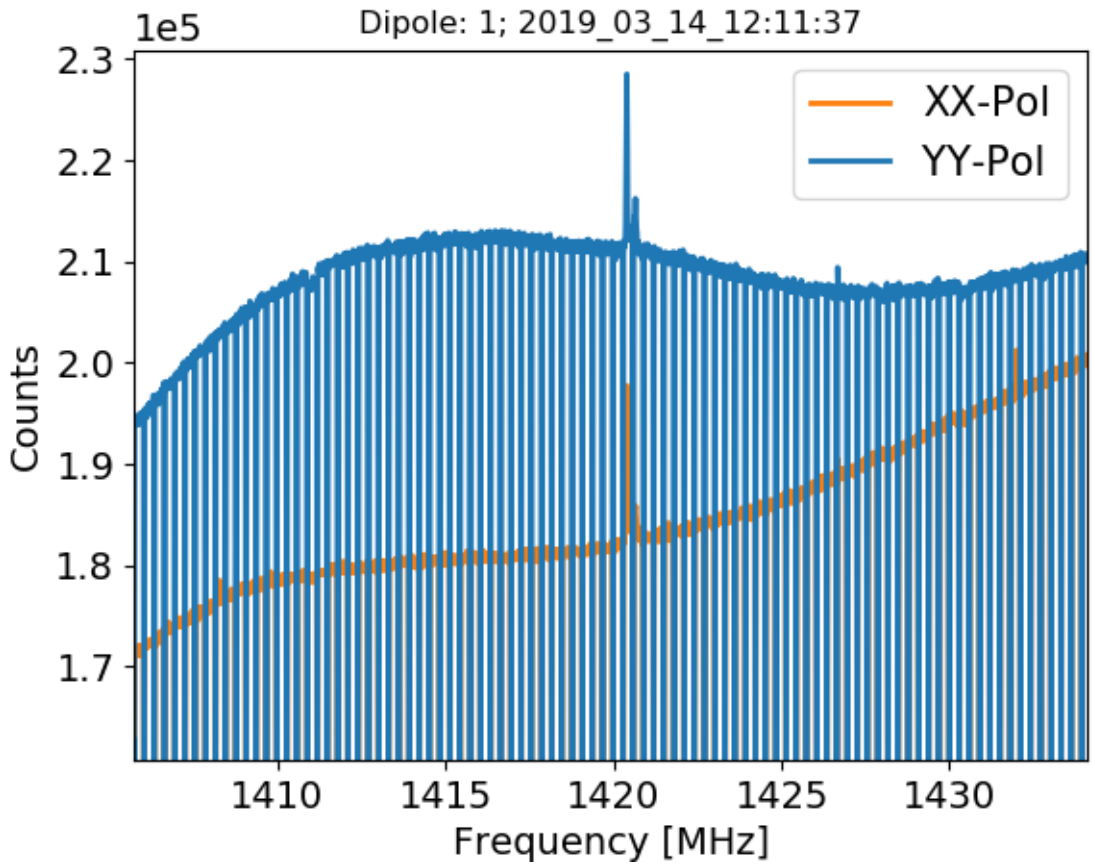


Figure 10: Example graphical output of `plotBandpass.py`. The XX and YY polarizations of the raw autocorrelation from the central dipole (dipole 1). Look to see the Galactic HI line is present as a positive indicator. This is also useful to look for signs of interference in the bandpass and the general shape of the raw autocorrelations.

7.1 plotTsys.py

7.2 `CreateLog.py`

A script to generate a log text file. The information contained within the generated logs can be used to determine the necessary input such as On/Off scan numbers to the post processing Matlab scripts. The columns in the log are:

1. Time stamp
2. Procedure name (e.g., Track, RaLongMap)
3. Scan #
4. Schedule block name
5. Current sequence number
6. Total Sequence number
7. Integration Length
8. Mode name

The only user input is the GBT project ID and session number. An example call is

```
$ ipython CreateLog.py AGBT16B_400_14
```

7.3 Using `calcSysFlux_Grid/7Pt.pro` to Derive System Equivalent Flux Densities

The first step in calibration of the raw beamformed spectra is to determine the scaling factor for the 'signal/reference' power ratios for each formed beam. While $T_{\rm sys}/\eta$ is a directly measurable quantity through the power ratio of the signal and noise correlation matrices, $T_{\rm sys}$ itself must take on some assumption of the true value of η. The flux density equivalent of $T_{\rm sys}$, or System Equivalent Flux Density (SEFD), makes no implicit assumption of η as it folds in the measurable ratio. The IDL scripts, `calcSysFlux_Grid.pro` and `calcSysFlux_7pt.pro` in the `utils` directory can be used to calculate the SEFD for grid and seven-point calibration scans, respectively.

Both scripts require six total inputs. The first input for both scripts is always a string of the calibrator source used. A user can choose from the sources: 3C48, 3C123, 3C38, 3C147, VirgoA, 3C286, 3C295, 3C353, and CygnusA. The script uses this string to look up dictionary values that store the coefficients used in Equation 1 of Perley & Butler (2017) to compute the source flux $S_{\rm src}$ (and statistical uncertainties) that is used as a flux scaling factor. The next four inputs are always sequentially increasing channel numbers used to define two channel regions — generally bracketing the HI line — whose power values are used to compute the statistics. The last input specifies the first reference scan of a seven point calibration scan when using `calcSysFlux_7pt.pro`, and the first reference scan in the calibration grid procedure for `calcSysFlux_Grid.pro`.

Both scripts are optimized to determine the calibrator flux and SEFD at 1420.406 MHz. The user can change the hard-coded value by adjusted the variable 'freqVal'.

The SEFD is given by

$$S_{\rm SEFD} = \frac{S_{\rm CalSrc} P_{\rm Off}}{P_{\rm sig} - P_{\rm ref}}, \tag{7}$$

where $P_{\rm sig}$ and $P_{\rm ref}$ are respectively the signal and reference power values taken at the frequency channel corresponding to 1420.406 MHz — usually channel 150 if the LO is set near 1450 MHz. Two distributions of signal and reference raw beamformed power values are built from the values contained within the two channel regions defined by the user and fit with separate Gaussian functions. For the seven-point procedure, the power value distributions are built from averaging the integrations of the corresponding Track scans. For the calibration grid, where the telescope is constantly slewing, only the single integration closest to the desired offset for a formed beam can be used. The distribution of signal power values therefore only consists of those within the bandpass of that specific integration. The reference distributions consists of the power values in the nearest (in angular distance) reference scan averaged over all integrations.

The respective uncertainties are the standard deviation returned by these Gaussian fits. If these fits fail to converge (e.g., due to a complex bandpass shape), the statistical standard deviation is used. All power values are corrected for atmospheric attenuation. The final uncertainty for the SEFD value is computed by propagating the statistical uncertainties of $P_{\rm ref}$, $P_{\rm ref}$, and $S_{\rm CalSrc}$. The script will report the final SEFD and associated propagated uncertainty. An example call within a `GBTIDL` prompt to determine the SEFD for beam 3 of a 7Pt-Cal scan that begins at scan number 7:

```
$ .compile calcSysFlux_7Pt
$ calcSysFlux_7Pt, '3C48', 100, 140, 160, 200 7
```

7.4 smooth_shift.pro

Just as we do in the custom GBTIDL pipeline developed to reduced on-the-fly mapping data from VEGAS, the spectra will be smoothed before calibration. All of FLAG's observing modes are considered to be 'total power'; that is, there are no separate sig, ref, or cal states (i.e., there is no noise diode firing). The native resolution of the PF-BCORR mode is 9.47 kHz. For extragalactic science, a good velocity resolution is 5.2 km s^{-1} (24.414 kHz); the smoothing kernel to set in smooth_shift.pro is therefore 24.414 kHz / 9.47 = 2.57695, or [0.288475, 1, 1, 0.288475]. An example call in GBTIDL is

```
$ smooth_shift, 'NGC6946', 'AGBT16B_400_12_NGC6946_Beam0_ss.fits', kernel
↪  = [0.288475, 1, 1, 0.288475]
```

7.5 PAF_edgeoffkeep.pro

After the user has taken the optional step of smoothing the raw beamformed spectra, calibration can be undertaken using the derived SEFD values from Section 7.3. An example of a raw, smoothed is shown in Figure 11. The nulls, or 'scalloping', seen every 303.18 kHz (every 32 fine frequency channels) is an artifact caused by the two stage

PFB architecture approach currently implemented in the backend. As the raw complex time series data are processed within the ROACHs, a response filter is applied in the coarse PFB such that the adjacent channels overlap at the 3 dB point to reduce spectral leakage. This underlying structure becomes readily apparent after the fine PFB implemented in the GPUs, however. The scalloping therefore traces the structure of each coarse channel across the bandpass. While the structure is somewhat mitigated in the calibrated data (since there is a division by a reference spectrum), power variations caused by spectral leakage (power from adjacent channels) in the transition bands of the coarse channel bandpass filter result in residual structure. Additionally, this scheme leads to signal aliasing stemming from the overlap in coarse channels. This does not hinder the performance of FLAG in terms of sensitivity, but a fix for the signal aliasing is a top priority going forward.

This code uses the first and last four integrations of a scan to create an average reference spectrum ($P_{\rm off}$) in units of counts. The bandpasses of each integration of each scan (and both polarizations) are then considered as the signal ($P_{\rm sig}$). Each integration is scaled such that

$$S_{\rm BP} = S_{\rm SEFD}\left(\frac{P_{\rm sig} - P_{\rm off}}{P_{\rm off}}\right), \tag{8}$$

where $S_{\rm SEFD}$ is calculated from Equation 7 and $S_{\rm BP}$ is the final calibrated bandpass in units of Jy. This code also utilizes a user provided channel channel range to fit a polynomial (usually of order 3) and remove residual baseline structure. Finally, an output file name is needed. An example call is

```
$ PAF_edgeoffkeep, 'NGC6946', Tsys_Y = 10, Tsys_X = 10, chanRange = [500,
↪  1500, 2000, 2500], order = 3, fileout =
↪  'AGBT16B_400_12_NGC6946_Beam0_edge_ss.fits'
```

7.6 PAF_chanBlank.pro & PAF_chanShift.pro

To mitigate the aliasing from the scallopping behavior, the affected frequency channels in the raw beamformed spectra can be blanked with
`PAF_chanBlank.pro` (i.e., before smoothing/calibration). Generally, these blanked data can now be smoothed/calibrated normally, although excessive dropouts across the bandpass could make a baseline fit difficult. The inputs are simply the source name and name of an output file. An example call is

```
$ PAF_chanBlank, 'NGC6946',
↪  fileout='AGBT16B_400_12_NGC6946_Beam0_chanBlank.fits'
```

Finally, before the smoothed, calibrated, and blanked data can be imaged, the channels in the data set corresponding to the higher LO setting must be shifted up by the equivalent of 151.59 MHz to align in channel space correctly. This corresponds to 6.2 channels when smoothed to 5.15 km s^{-1} as in the example in Section 7.4. The script,
`PAF_chanShift.pro` can be used to accomplish this last reduction step. There are three necessary inputs: (1) the source name; (2) an output file name; (3) and the value with which to shift the channels by. An example call is:

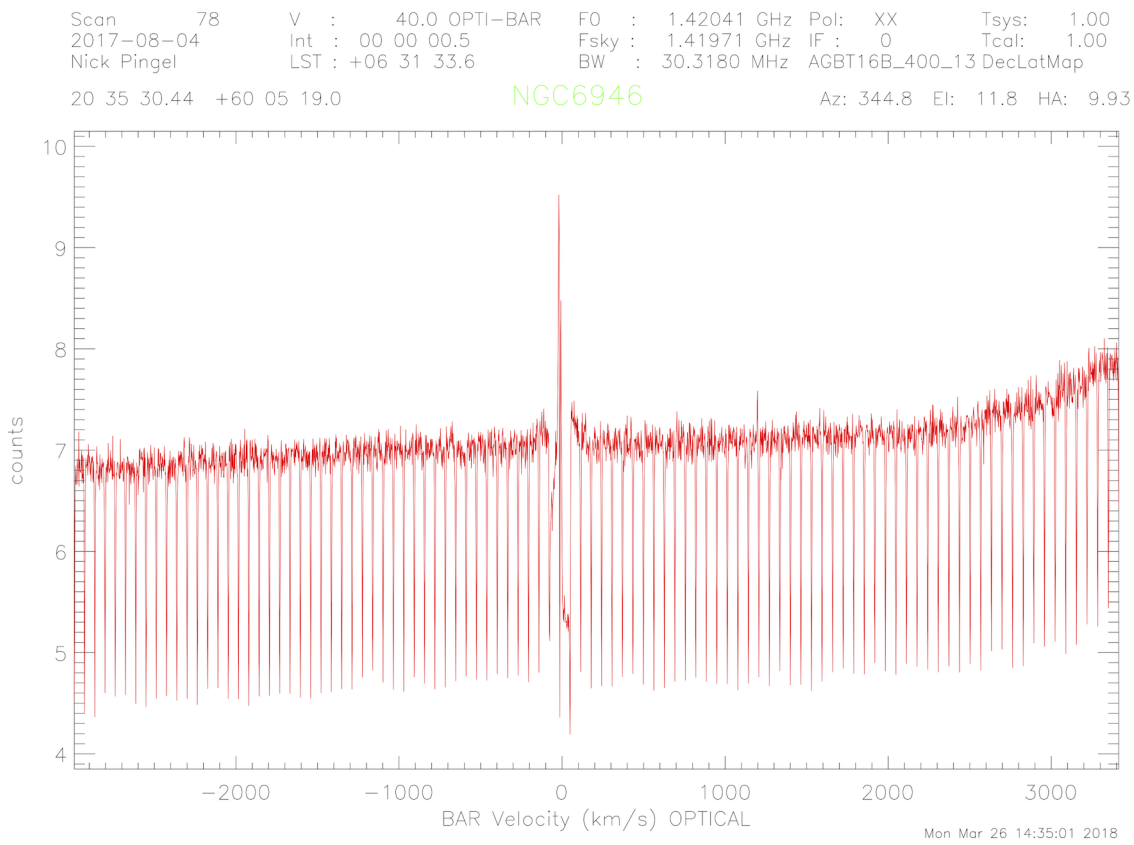


Figure 11: An example of an uncalibrated, beamformed spectrum as seen in a GBTIDL plotter window taken from the 35th integration of the 19th column of a *DecLatMap* scan of NGC6946. The 3 dB drop in power (i.e., 'scalloping') is an artifact of the two step PFB implementation of the backend (see text).

```
$ PAF_chanShift, 'NGC6946',
↪  fileout='AGBT16B_400_12_NGC6946_Beam0_edge_ss_chanBlank_shift.fits',
↪  chanShiftVal = 6.2
```

7.7 gbtgridBFCubes.sh

This is a bash script that calls the GBO program, `gbtgridder` in order to image each SDFITS file, as well as create a combined cube. It is set up to create cubes with Gaussian-Bessel convolution function, have 128×128 pixels each 105 arcseconds in extent, and a rest frequency of 1420.406 MHz. A single example call to the gridding program is:

```
$ gbtgridder AGBT16B_400_12_NGC6946_Beam0_edge_ss.fits
↪  --output=AGBT16B_400_12_NGC6946_Beam0_cube -k gaussbessel --mapcenter
↪  308.72 60.15 --pixelwidth 105 --restfreq 1420.406 --noweight --noline
↪  --nocont
```

The last options suppress a weight, line, and continuum image from being output. The user will need to change the 'projID', 'session', 'obj', 'ra', 'dec' to match the desired input file name, observed object, and major and minor coordinates of the map center (in either J2000 or Galactic), respectively.

7.8 plotBeamPatterns.py

This script constructs the formed beam patterns and provides a look at the beam profiles. The beam pattern of the ith beam at the kth frequency channel as a function of angle θ (I_k^i) is given by the equation

$$I_k^i = \left| w\left(\theta_i\right)^H \hat{a}_k\left(\theta_i\right) \right|^2, \tag{9}$$

where $w\left(\theta\right)$ is a maxSNR beamformer weight vector for a beam pointed in the direction θ_i and $\hat{a}_k\left(\theta_i\right)$ is the array steering vector. The steering vector is determined for a general pointing by the equation

$$\hat{a}_k\left(\theta\right) = \sqrt{\lambda_{\max,k}}\hat{\mathbf{R}}_{\text{off}}\tilde{\mathbf{v}}_k, \tag{10}$$

where $\tilde{\mathbf{v}}_k$ is the dominant eigenvector (corresponding to the largest eigenvalue, $\lambda_{\max,k}$) when solving the generalized eigenvalue equation.

The output consists of two plots: (1) a seven panel plot showing the formed beam patterns on the sky in units of dB (2) a seven panel plot showing perpendicular profiles with Gaussian fits. In addition to the output plots, four numpy arrays that are generate by the code are saved to disk in the form of a binary file via the package *pickle*. The four variables are the cross-elevation and elevation coordinates of the steering vectors and the YY and XX beam pattern responses at each of these points. The information stored in these arrays is enough to interpolate the pattern responses to a well-defined grid in order to image the patterns at some later time.

The script will report the FWHM of the Gaussian fits and the estimated area in square arcseconds. At the time of this writing, the script needs access to *MatLab* files produced by code written by BYU. Specifically, the aggregated grid, weights, and tsys.mat files, which contain the steering vectors for each beam and cross-elevation and elevation. The three inputs are project value (assuming a hard coded location), calibration scan time (either grid or seven), and a path to the directory that holds the weight FITS files. An example call is:

```
$ ipython plotBeamPatterns.py AGBT16B_400_12 grid
↪  ../../data/AGBT16B_400/AGBT16B_400_12/weight_files
/fits_files/scaledWeights/
```

Figure 12 gives an example output of this script when applied to a standard calibration grid performed on 3C295. The red x's in the plots showing the individual beam power patterns denote the beam pointing center (according to information stored in the weight FITS files) and the red dashed lines show the location of the cross-elevation/elevation profile cuts in the bottom plot. The intersection of the dashed lines represent the peak response of each formed beam. The central beam is quite Gaussian, while the outer beams show deviation from Gaussinity and have more prevalent sidelobe structure.

7.9 Graphical Interface for End - to - End Reduction

A graphical user interface (GUI) was made in Python using the `tkinter` package to simplify and streamline the above process for users who aren't necessarily expert-level

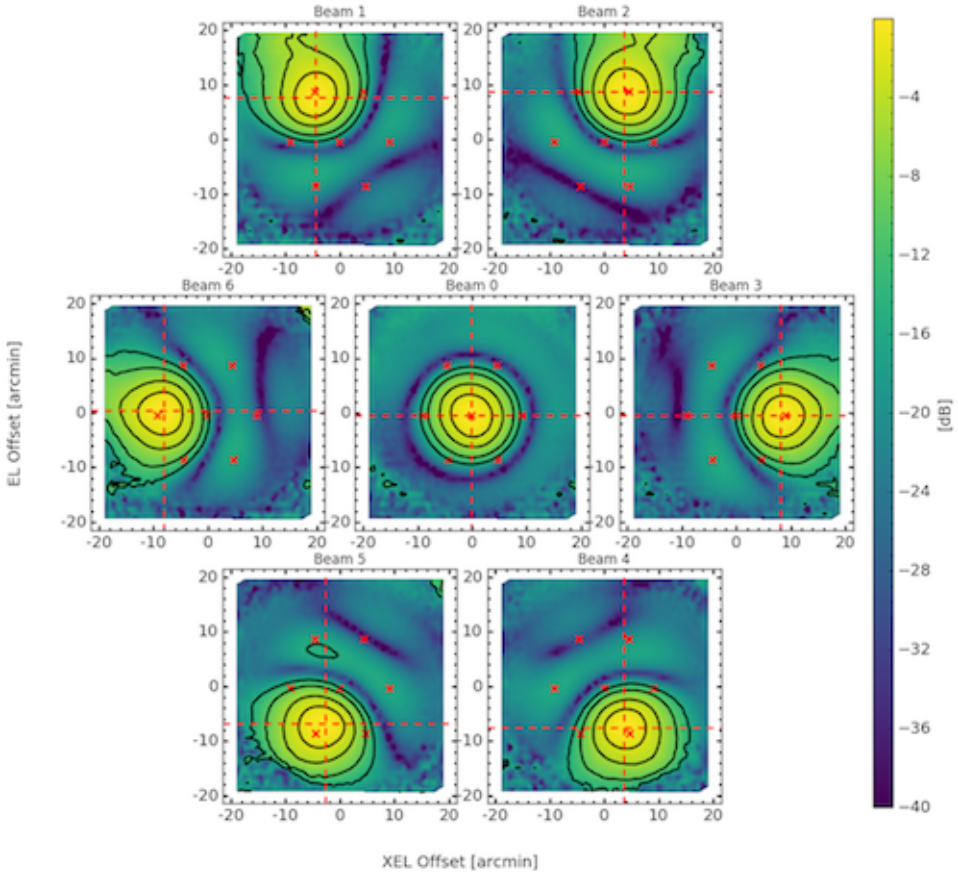
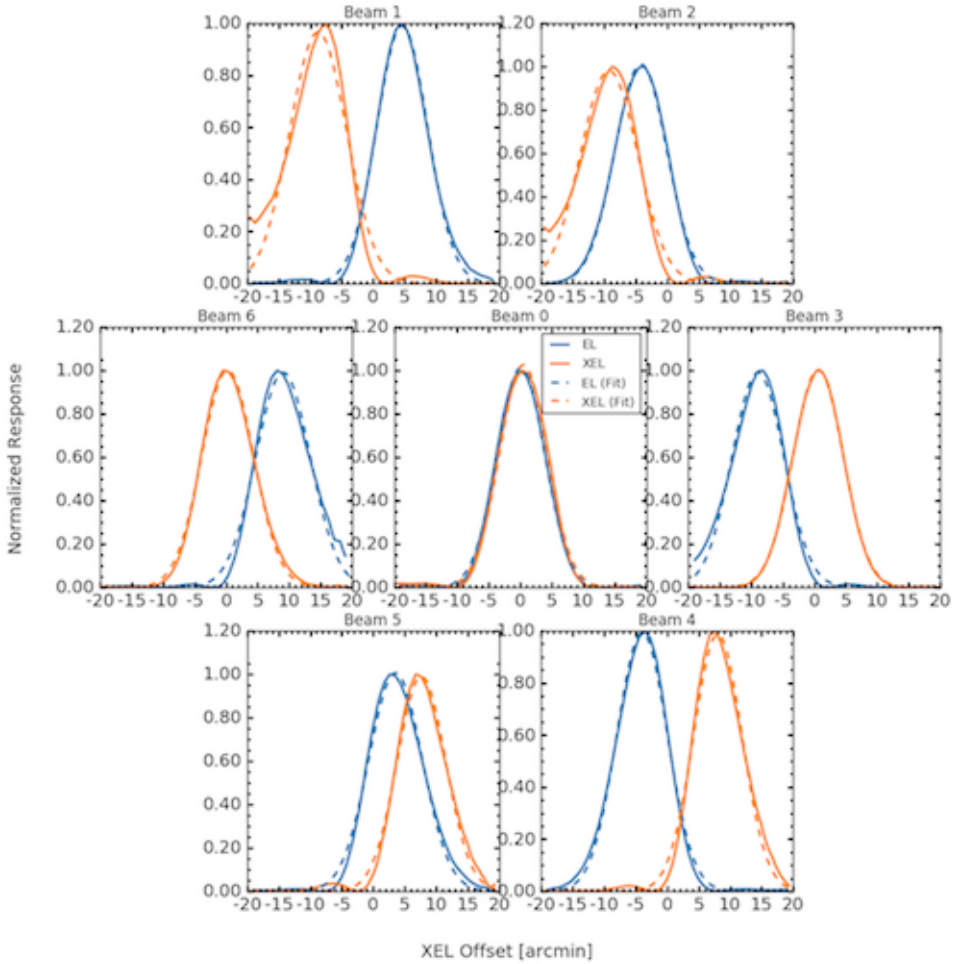


Figure 12: *above*: The formed beam pattern derived from a calibration grid The red x symbols denote the intended beam centers. The intersections of the vertical and horizontal dashed red lines denote the location of the peak response of each formed beam. The contours represent levels of −2, −5, −10, and −15 dB. *below*: Beam profiles along the dashed red lines in the first panel with Gaussian fits represented by dashed lines.


Beam 1
Beam 2
Beam 6
Beam 0
Beam 3
Beam 5
Beam 4
EL
XEL
EL (Fit)
XEL (Fit)
Normalized Response
XEL Offset [arcmin]

users. The interface is shown in Figure 13, with some options filled out for Eridanus observations. The code currently resides in `/home/groups/flag/scripts/pyFlag/evan/` so the following references to programs will assume this directory as a prefix. The interface is split up into two tabs, with the first being the "Weight Filling" tab and the second being the "Processing" tab. The former has functions for deriving beamweights and applying them to the covariance matrices of the data, while the latter deals with processing the filled data through `gbtidl` procedures for flux calibration and other necessary processing. The console window used to start the GUI will also display all printing statements from code executed by the interface.

7.9.1 Weight Filling

The "Weight Filling" tab starts with the initial weight derivation from calibration scans, which previously used a suite of Matlab codes[9]. These codes have since been translated into Python and implemented here. Before initializing the GUI, you must first activate the Python 3 environment used to write the program, as well as load paths to supporting scripts.

```
$ source activate /users/esmith/.conda/envs/py365

$ export
↪ PYTHONPATH=$PYTHONPATH:/home/groups/flag/scripts/pyFlag/evan/kernel/:
↪ /home/groups/flag/scripts/pyFlag/evan/patterns/:
↪ /home/groups/flag/scripts/pyFlag/evan/senstivity_maps/
```

And start the application with

```
$ python data_gui.py
```

The GUI is broken up into several `tkinter` Frames, with each one corresponding to a different step in the reduction and containing the relevant child widgets. The top frame contains info about the session and calibration type to use. The list of possible sessions is in session_info_gui.py, which contains a very long elif chain with information about every FLAG observing session. Find your session name in that file and type it into the "Enter session below:" dialogue box. The interface will save results to your scratch directory, as long as your username is typed into the "Enter your username:" dialogue box. Some scripts will save files to the present working directory, after which the interface moves them to the scratch directory. If these scripts are interrupted for any reason, be aware that the unfinished set of files will still reside in your present working directory and may cause errors. There is also an "All-in-one" button that does the whole weight-filling process - sensitivity_map, plot_beam_patterns, weightFiller, and PAF_filler all in one click. All of the buttons to run code have checks to make sure that all required inputs are filled out, but not that they are syntactically/functionally correct. The buttons are also named for the namesake program they call. The warnings

[9]The Matlab codes are found in /home/groups/flag/scripts/matFlag/, and my directory structure mimics this one as close as possible. The Matlab codes finish with writing out the binary weights files that go into weightFiller.py.

will show in the console window used to start the GUI. The data reduction steps taken for the "All-in-one" button take about 20-30 minutes to complete fully, with the longest script being PAF_filler.

The next frame down contains the sensitivity_map button, which runs the python translation of sensitivity_map.m. This is the first program in the weights derivation and data reduction process, and derives aggregate weights using the maxSNR algorithm. There's two boolean options for overwrite and quickmap (What does overwrite do again?)

Below that is the frame for the plot_beam_patterns button with a required input for the 'note'. This button calls the python translation of plot_beam_patterns.m This program writes out the binary weights files for each bank.

The next frame down calls programs from the SpectralFiller package[10]. The rest frequency and center frequency are auto-filled to 1420.405e6 and 1450e6, with the optional inputs below those. The checkbox next to an optional input must be checked in order for the program to take it as an input. The weightFiller button turns the binary weights files into FITS files, and the PAF_Filler button applies those weights to the spectroscopic data for the session. To finish out the data reduction process for the first tab, press weightFiller and then PAF_filler. PAF_filler has a variety of optional inputs to define good/bad time-stamped files, objects that were observed, and beams to fill.

7.9.2 Processing

The second tab in the GUI deals with processing the spectral line data after it has been filled. There are several steps, outlined in the sections above - to find the flux calibration and apply extra processing. The buttons have their required and optional available inputs next to them. The "Link IDL Scripts" button adds softlinks to the IDL programs used to your working directory so that they can be called from the command line. The "Inputs" button shows the inputs for chanBlank_parallel.py, chanShift_parallel.py, and PAF_edgeoffkeep_parallel.py in the console window.

7.9.3 Current Issues / Soon to come:

- For sensitivity_map, the quickmap option does nothing for now - in matlab it calls an alternate aggregate_banks_rb.m
- Why doesn't my modified plot_beam_patterns write the correct FITS files but weightFiller does

References

Perley, R. A., & Butler, B. J. 2017, ApJS, 230, 7

Pingel, N. M., Pisano, D. J., Heald, G., et al. 2018, ApJ, 865, 36

[10] `https://github.com/nipingel/SpectralFiller`

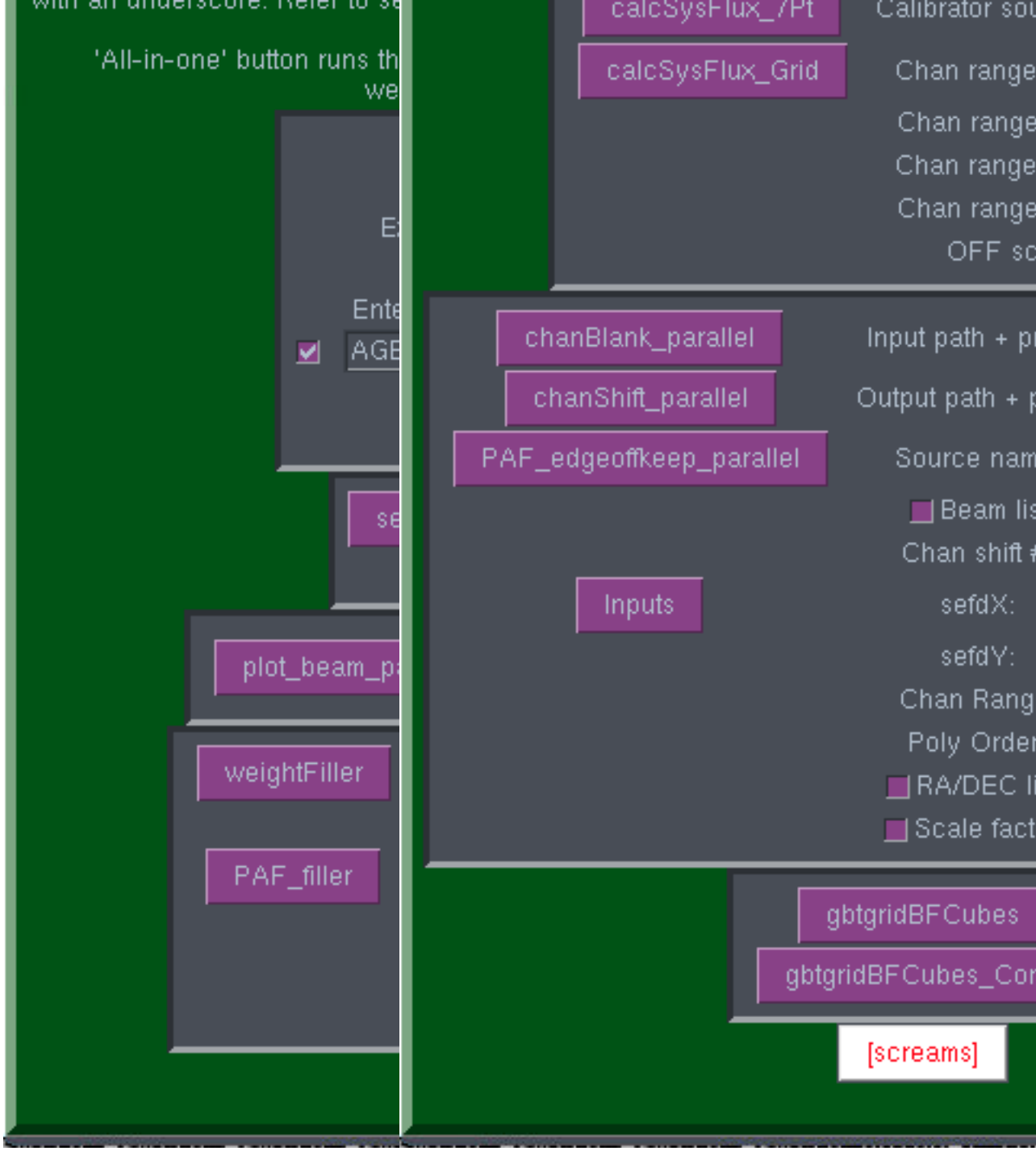


Figure 13: An image of the data reduction GUI with some options filled out.