FLAG Spectral Line Software

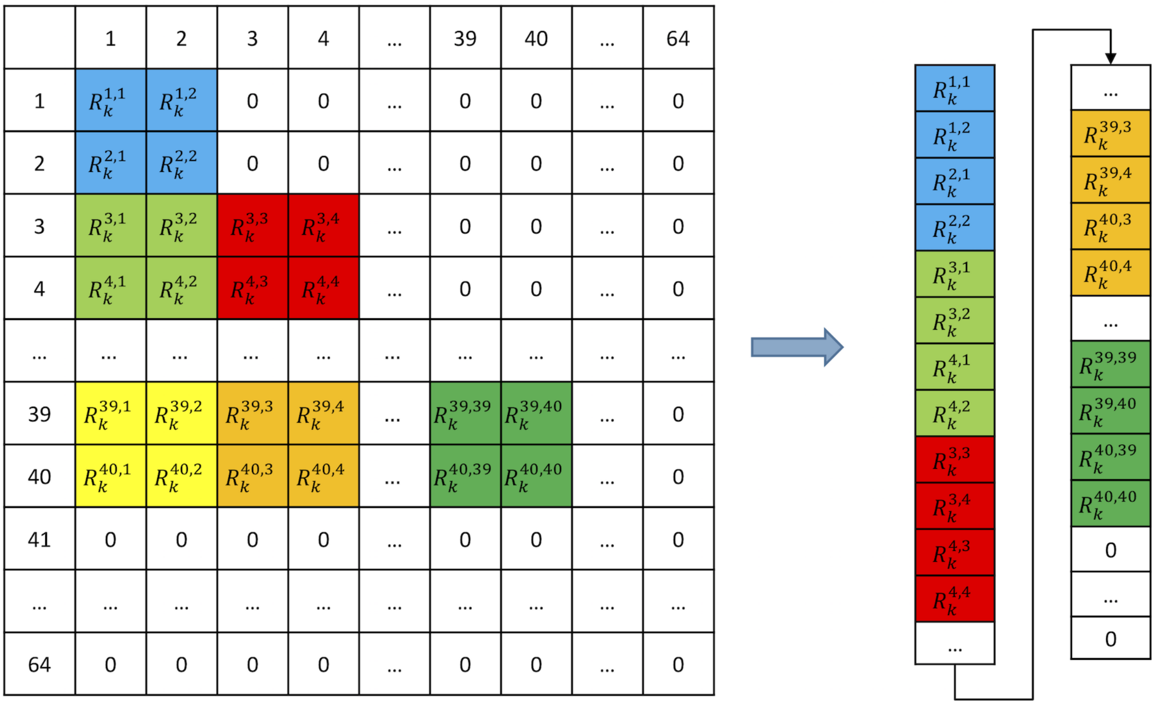
Nickolas M. Pingel

June 27, 2019

1 Introduction

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

The remainder of this document will focus on how to monitor, reduce, 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 documentation 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 and transition the data formats into their final formats that can be used in the GBO computing environment; Section 5 provides examples of observing configuration and Astrid scripts, Section 6 describes and provides examples for utility scripts that are to be used to monitor FLAG observations; and Section 7 summarizes and provides examples of the data calibration, reduction, and imaging scripts.



	1	2	3	4	...	39	40	...	64
1	$R_k^{1,1}$	$R_k^{1,2}$	0	0	...	0	0	...	0
2	$R_k^{2,1}$	$R_k^{2,2}$	0	0	...	0	0	...	0
3	$R_k^{3,1}$	$R_k^{3,2}$	$R_k^{3,3}$	$R_k^{3,4}$	...	0	0	...	0
4	$R_k^{4,1}$	$R_k^{4,2}$	$R_k^{4,3}$	$R_k^{4,4}$	...	0	0	...	0
...	...	...	...	...	...	...	...	...	...
39	$R_k^{39,1}$	$R_k^{39,2}$	$R_k^{39,3}$	$R_k^{39,4}$	...	$R_k^{39,39}$	$R_k^{39,40}$	...	0
40	$R_k^{40,1}$	$R_k^{40,2}$	$R_k^{40,3}$	$R_k^{40,4}$	...	$R_k^{40,39}$	$R_k^{40,40}$	...	0
41	0	0	0	0	...	0	0	...	0
...	...	...	...	...	...	...	...	...	...
64	0	0	0	0	...	0	0	...	0

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 correlation 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 Raw Data Formats and SDFITS

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 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 correlations of individual dipole elements. However, due to

[1] `https://safe.nrao.edu/wiki/bin/view/Beamformer/`

[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`

xGPU limitations, the covariance matrices are of size of 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 correlations are ordered is illustrated in Figure 1. Here, $\mathrm{R}_{\mathrm{k}}^{\mathrm{i,j}}$ corresponds to the correlation 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. When 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 linear and 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 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 correlations 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 metaDataModule.py and beamformerModule.py. The following subsections describe the logic flow for each component, in addition to the script that is used to unpack the binary files that contain the complex weights and convert them to FITS files.

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

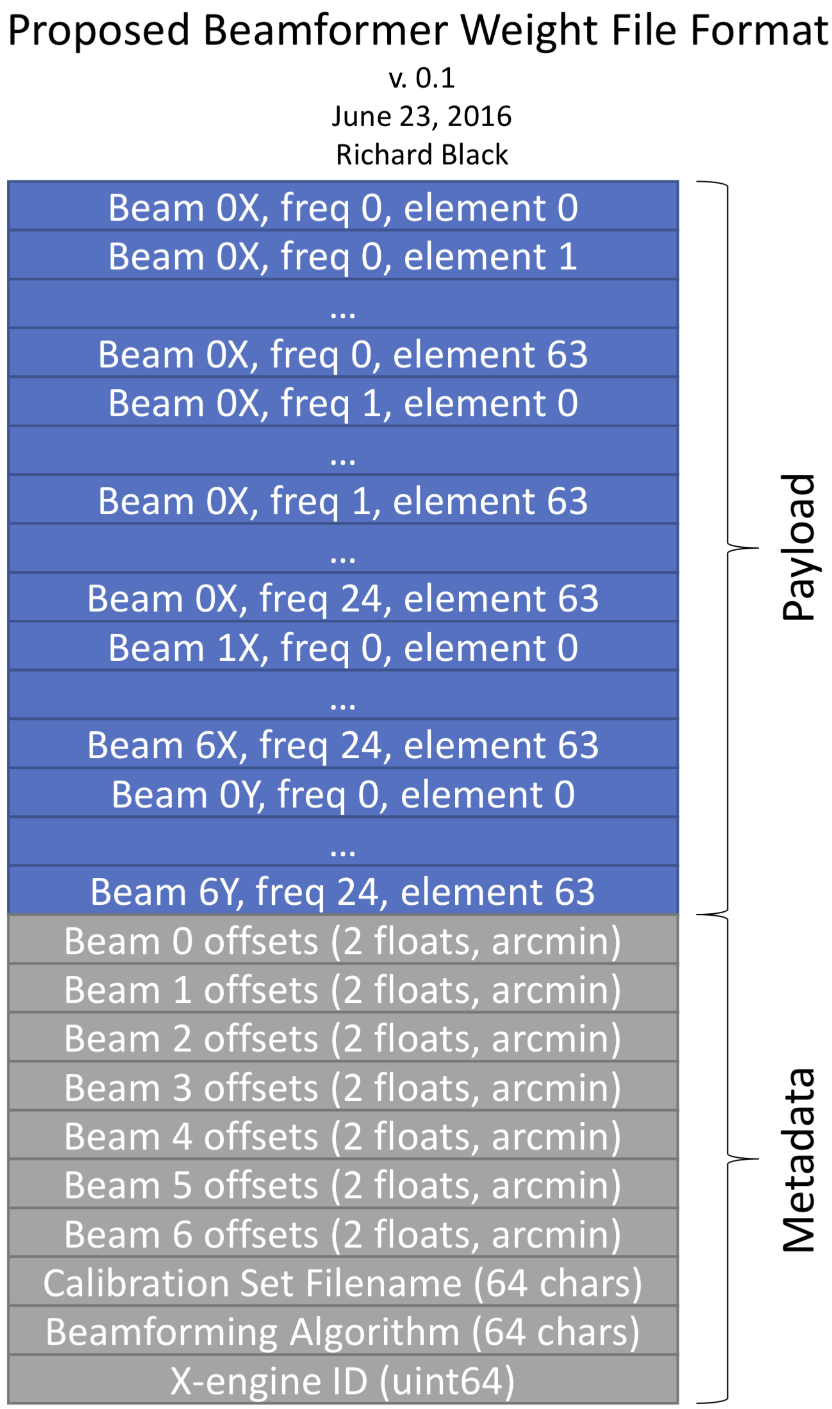
FITS file (WVU)	Binary file (BYU)
0 (boresight)	3 (boresight)
1	0
2	1
3	4
4	6
5	5
6	2

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) contain the 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 Dr. 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 38 elements (corresponding to the number of dipoles multiplied by the two linear polarizations), 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. Subsequently, all 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 to-



Proposed Beamformer Weight File Format
v. 0.1
June 23, 2016
Richard Black

Beam 0X, freq 0, element 0
Beam 0X, freq 0, element 1
…
Beam 0X, freq 0, element 63
Beam 0X, freq 1, element 0
…
Beam 0X, freq 1, element 63
…
Beam 0X, freq 24, element 63
Beam 1X, freq 0, element 0
…
Beam 6X, freq 24, element 63
Beam 0Y, freq 0, element 0
…
Beam 6Y, freq 24, element 63
Beam 0 offsets (2 floats, arcmin)
Beam 1 offsets (2 floats, arcmin)
Beam 2 offsets (2 floats, arcmin)
Beam 3 offsets (2 floats, arcmin)
Beam 4 offsets (2 floats, arcmin)
Beam 5 offsets (2 floats, arcmin)
Beam 6 offsets (2 floats, arcmin)
Calibration Set Filename (64 chars)
Beamforming Algorithm (64 chars)
X-engine ID (uint64)

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

tal 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 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 reminder of the vector is padded by 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/AGBT17B_360_02/BF/
```

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. Figure 3 illustrates the logic flow of the program. An example of a syntactically correct call is:

```
$ ipython PAF_Filler.py /home/gbtdata/AGBT17B_360_01
 ↪  /lustre/project/flag/AGBT17B_360_03/BF/weight\_files/w_17B_360_02_*.FITS
 ↪  1420.405e6 -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
```

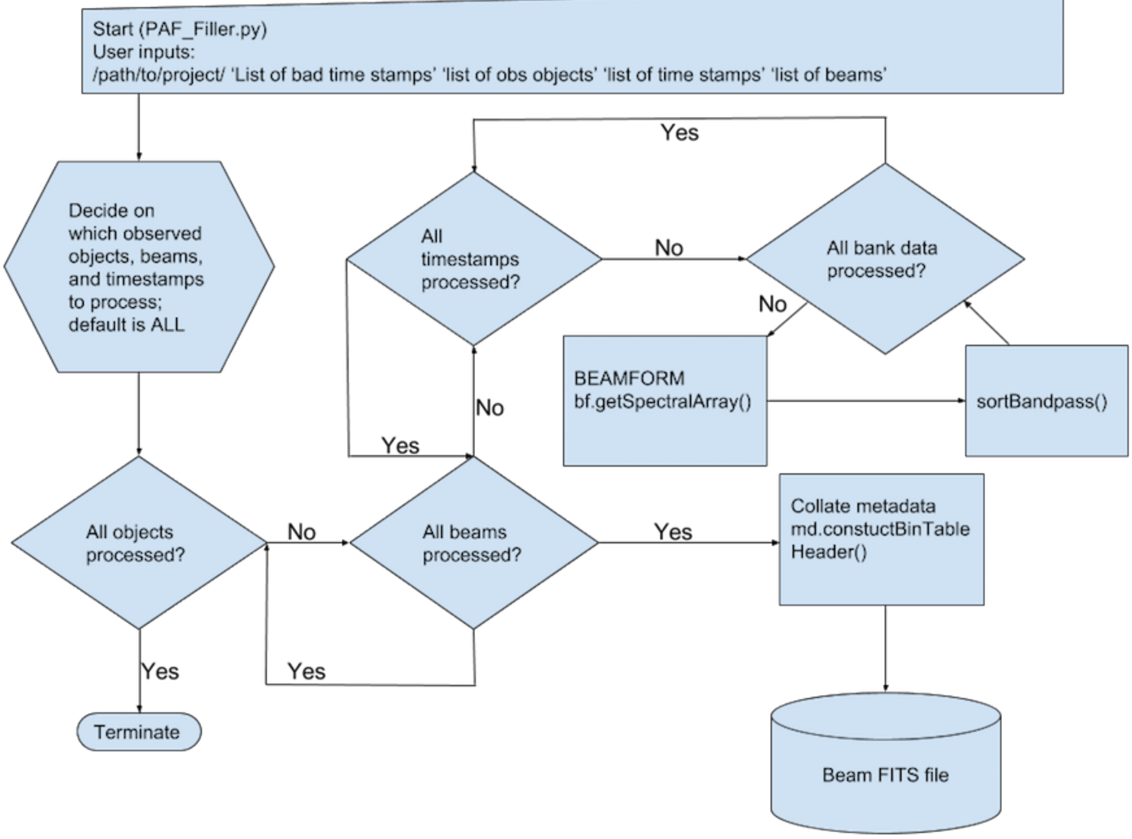

Start (PAF_Filler.py)
User inputs:
/path/to/project/ 'List of bad time stamps' 'list of obs objects' 'list of time stamps' 'list of beams'
Decide on which observed objects, beams, and timestamps to process; default is ALL
All objects processed?
Yes
Terminate
No
All beams processed?
Yes
No
All timestamps processed?
Yes
No
All bank data processed?
Yes
No
BEAMFORM bf.getSpectralArray()
sortBandpass()
Yes
Collate metadata md.constuctBinTable Header()
Beam FITS file

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.

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 one replaces the bank letter with the wildcard character '*'. The third input is the rest frequency in units of Hz. The rest 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 the value explicitly for now. These paths and rest frequency value are the only inputs that are required by the user. Space delimited list of timestamps after the '`-b` flag represent bad scans that, for whatever reason, are not required for your subsequent analysis. 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 -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).

After parsing the flags to determine what sources and beams the user wishes to fill, the program collate all the associated scans to 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. 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 about the collection of scans are passed to the metadata object (see 3.4 through the `constuctBinTableHeader()` object function. Finally, once the FITS binary table has been constructed for a given beam, the 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 PAF_Filler.py by passing the path to 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 to a more suitable 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 calibration or fine channelized mode). 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 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/GpuToFISHFITS_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 that two important aspects: (1) irrelevant correlations caused by xGPU limitations are thrown away at this stage; (2) dipoles will generally be zero 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 is a python object that contains several internal methods to collate all associated metadata that is 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 initialization of the object occurs after all bank FITS files associated with an observed object and beam have been processed by the beamforming module. The project name, path to the raw data, path to the weight FITS files, a list of bank files, rest frequency, a list of number of banks processed per scan, global XX and YY data buffers, the beam number, and a boolean flag that denotes whether we are in CALCORR or PFBCORR are passed upon object initialization. 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. 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 module function that collects the necessary information. The internal functions 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 paramter 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 a given parameter 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 Header Data Unit (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 reference frame is then changed to that of the GBT through the `pyslalib.sla_aop` using the latitude and longitude available in the GBT observers guide. The change from equatorial to horizontal coordinates is then computed utilizing the `pyslalib.sla_e2h` method. 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 Azimuth. After the application of the beam offsets, the new corresponding

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

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 `pyslalib.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 Winter 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 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 RA and Dec 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

5 Observing Scripts

Example configuration, calibration, and science observing scripts are presented in Figures 4- 7.

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

5.1 CalcObsTime.py

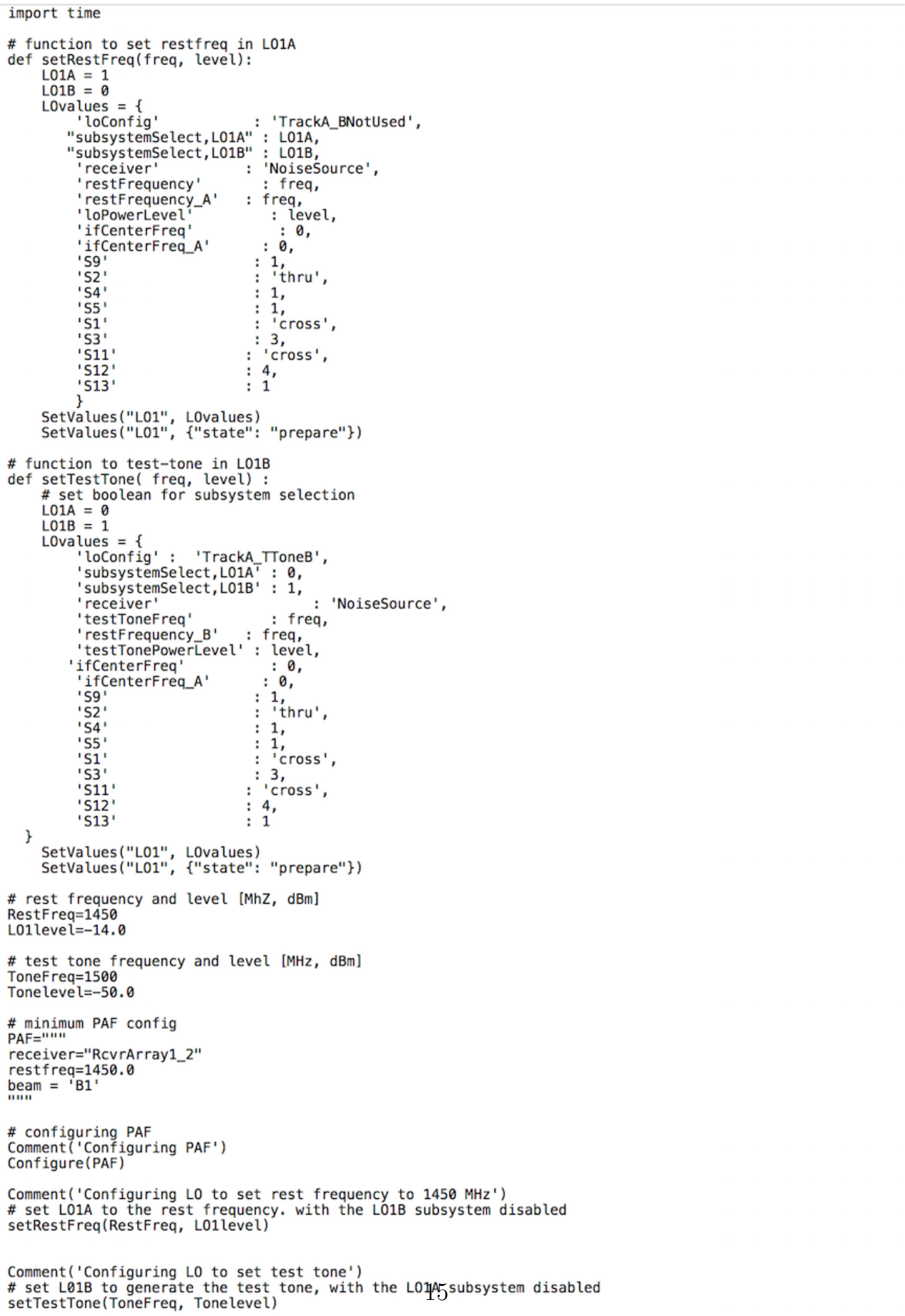
5.2 Configuration

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.

5.3 Calibration Scripts

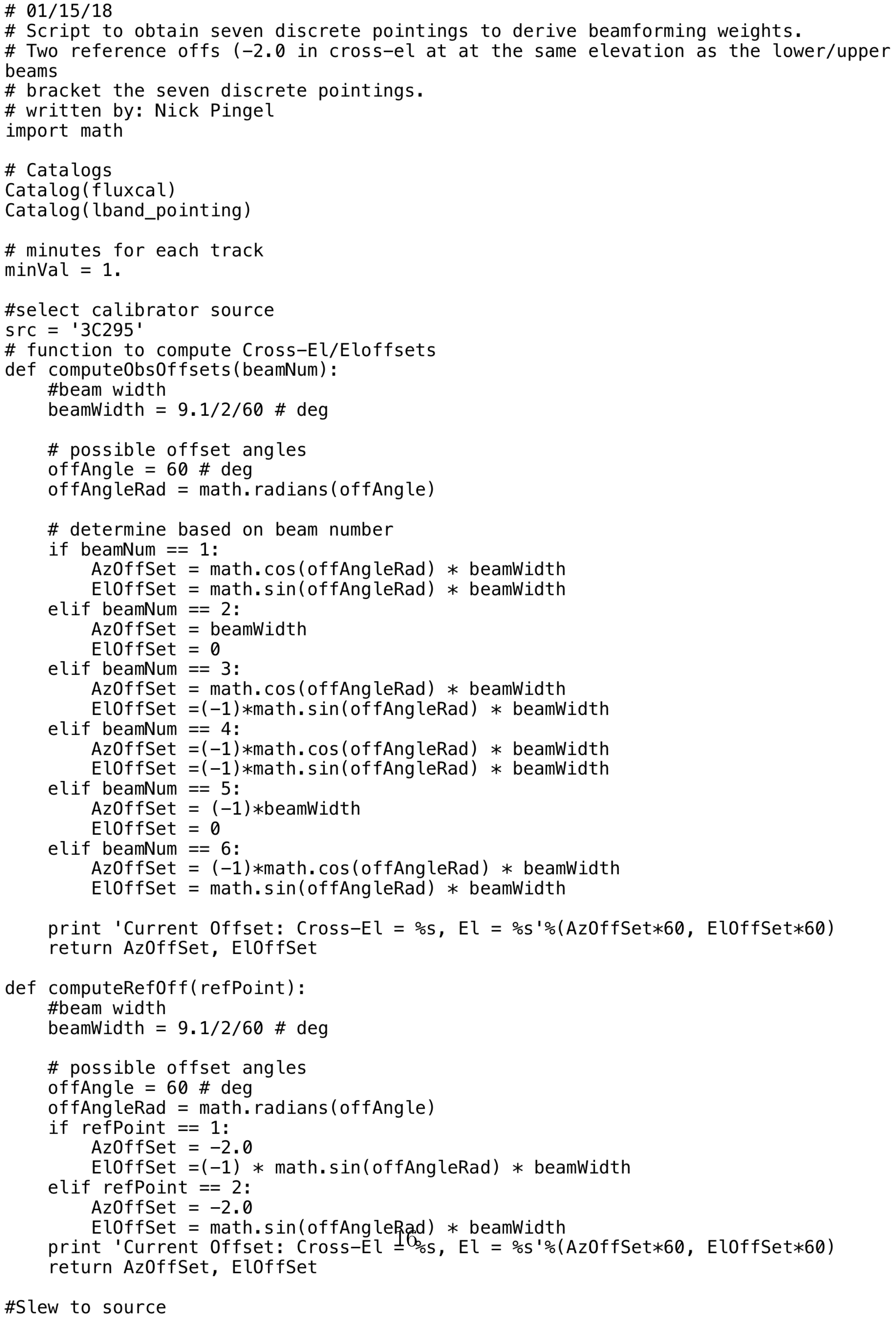
5.3.1 7-PtCal

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.

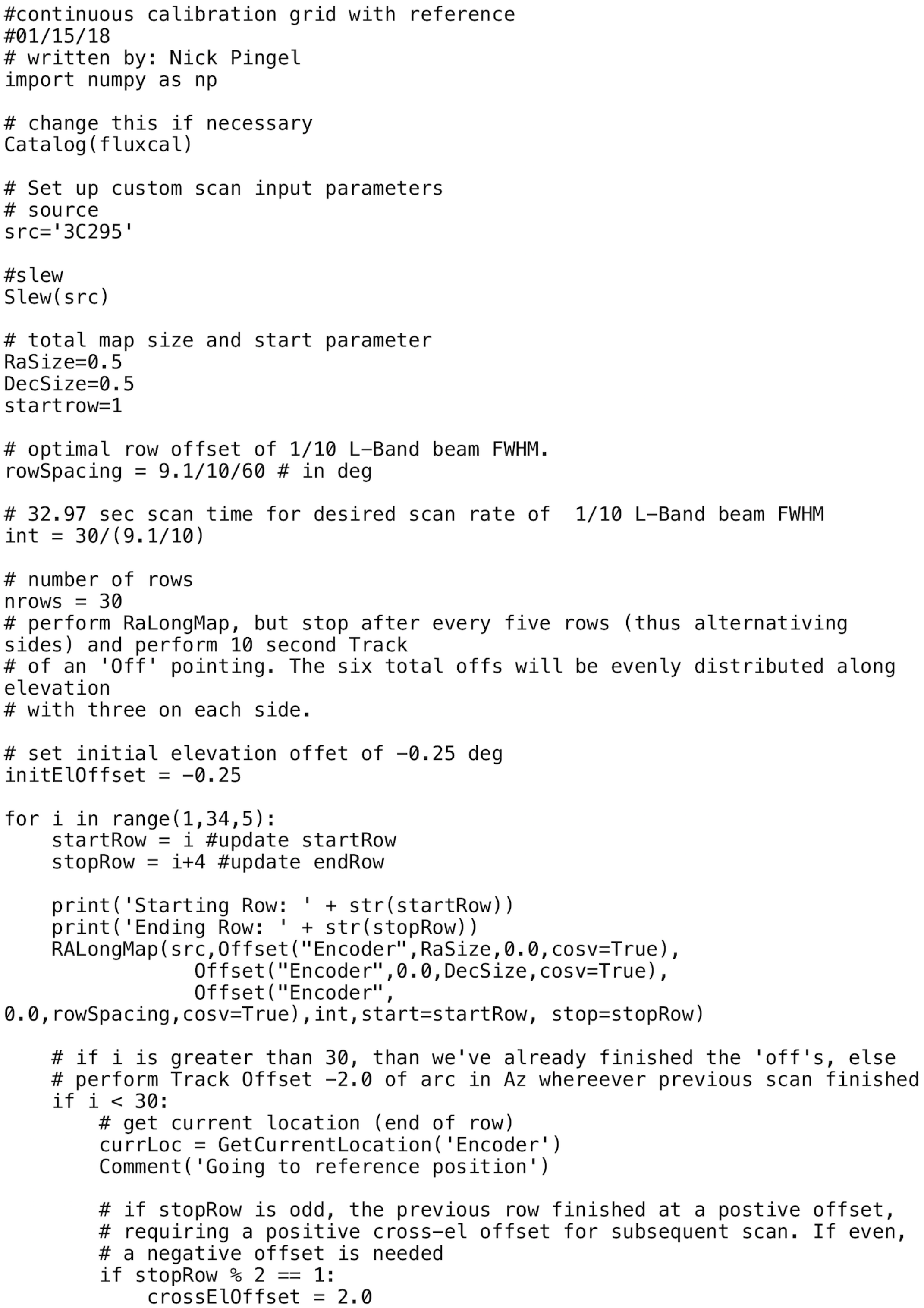


```
# 01/15/18
# Script to obtain seven discrete pointings to derive beamforming weights.
# Two reference offs (-2.0 in cross-el at at the same elevation as the lower/upper
beams
# bracket the seven discrete pointings.
# written by: Nick Pingel
import math

# 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).



```
#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')
```

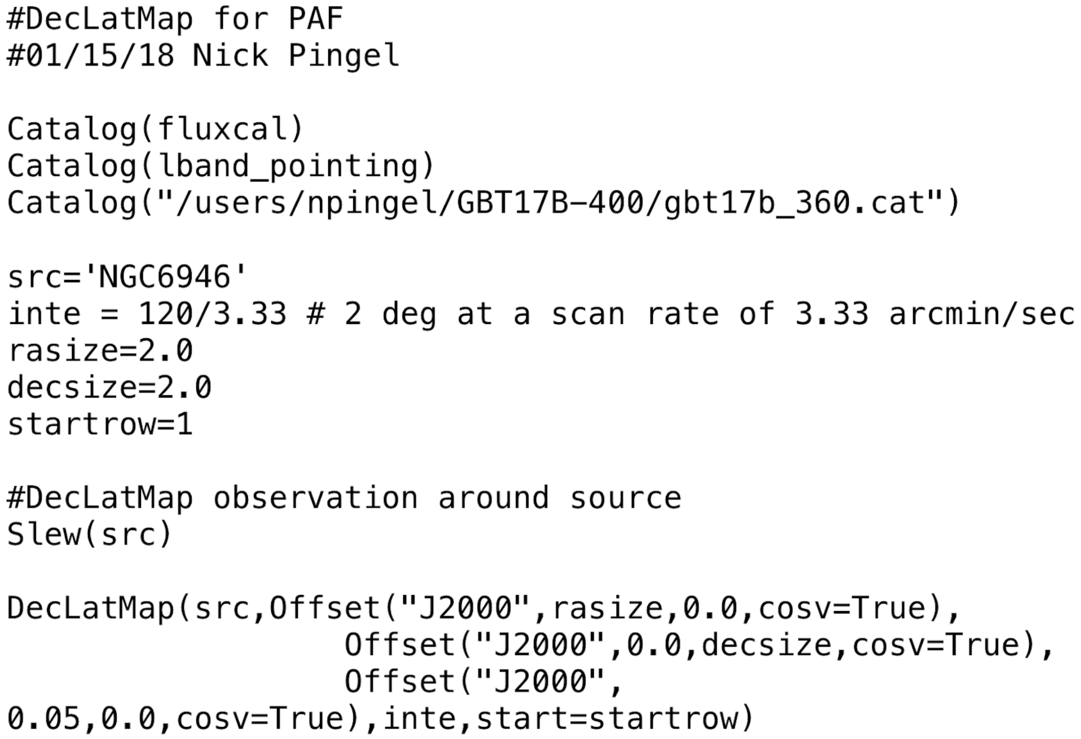
Figure 6: An example observing script of a calibration grid (continued from previous page).

5.3.2 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 ($\sim$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.

5.4 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 $\sim$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.



```
#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.

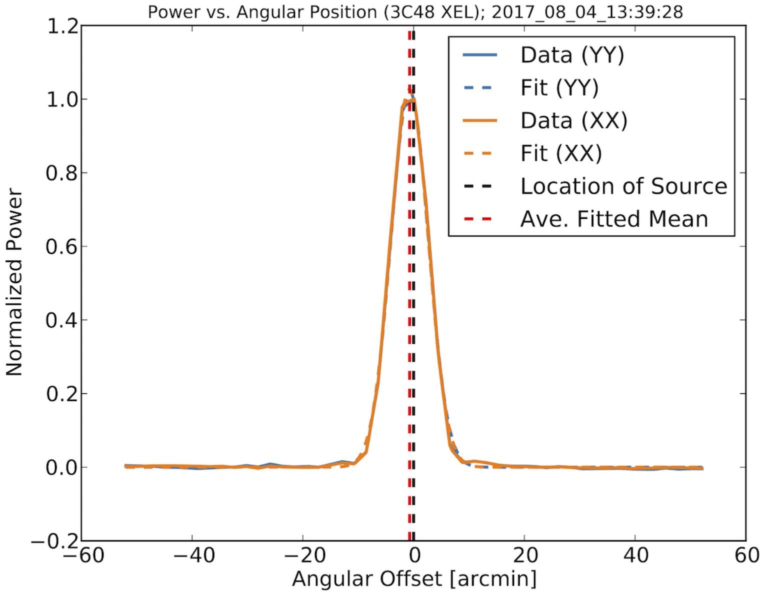


Figure 8: Example graphical output of PAF_Peak.py. A scan sequence consists of two scans that travel a total angular distance of +/- 130 arcminutes with the source centered in the cut. The solid lines represent the data, while the Gaussian fits are denoted by dashed lines in the equivalent color. The location of the source and average of the two return fitted means are shown by vertical dashed black and red lines, respectively.

6 Data Monitoring Scripts

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 will first summarize the functionality and provide examples of data monitoring scripts before moving on to discuss useful reduction/analysis scripts.

6.1 Data Monitoring Scripts

6.1.1 PAF_Peak.py

Many aspects of an observation can affect the ability of a radio telescope to point at the desired sky position including atmospheric effects, gravitational forces that vary with elevation, and the relation between different celestial coordinate systems. To ensure the GBT is pointed correctly, several distinct models that consider atmospheric conditions and the unique structural irregularities of the GBT make up the overall pointing model (Prestage et al. 2004). While these are automatically implemented for any observation, there are still arcminute (at L-Band) corrections that are required. Since FLAG has its own unique backend, the continuum receiver is not available thus requiring custom software and observing scripts whenever we are observing with FLAG. It is recommended to perform a pointing observation in both cardinal coordinates (cross-elevation and elevation) before beginning your science observations.

The custom observing scripts replicate the movement of the default 'Peak' scans available through astrid. The telescope is moved in +/- directions for a total of 130 acrminutes with the source centered in the path. There is a scan for both cross-elevation and elevation cuts. Each scan direction takes 30 total seconds dumping data out every 0.5 seconds (default for CALCORR mode).

After a scan is completed, the python program, PAF_Peak.py, will plot the normalized total power of the user provided dipole element (usually one wants to use central dipole 1) as a function of scan time with a Gaussian fit overlaid for the XX and YY polarizations. The returned fitted mean (in units of arcminutes) is the required Local Pointing Correction (LPC) that should be applied by either asking the operator or through the Cleo Device Manager.

To run this script, the user must provide several inputs. The first input is the project and session name (e.g., AGBT16B_400_01), the second is the timestamp of the cross-elevation/elevation cut (e.g. 01_18_18_00:00:00), the third must be either XEL or EL to determine direction (so it knows what FITS header keywords to look up while determining the antenna pointing), the fourth being the dipole and must be an integer between 1 and 19, and finally the name of the calibrator source observed. The flux models of Perley & Butler (2017), assumed L-Band gain of 1.86 [K/Jy], and the signal-to-noise ratio between the peak and mean baseline noise is used to estimate a system temperature (T_{sys}) in Kelvin.

An example call and the subsequent text output to the terminal is given below:

```
$ ipython PAF_Peak.py AGBT16B_400_13 2017_08_04_13:39:28 XEL 1 3C48

-----Fit Statistics (XX Pol)-----

Mean [arcmin]: -0.702110936942+/-0.00985090853533

FWHM [arcmin]: 7.89267743008+/-0.0230746550722
```

```
Reduced Chi-Sq: 0.0611814952157

Peak Power: 195250265.081

Estimated Tsys [K]: 26.7521799843

-----Fit Statistics (XX Pol)-----

Mean [arcmin]: -0.747401766135+/-0.00971459062521

FWHM [arcmin]: 7.86587937985+/-0.0227553408582

Reduced Chi-Sq: 0.0619439811324

Peak Power: 178499066.518

Estimated Tsys [K]: 26.8164265199
```

An example of the associated graphical output from the executed command is shown in Figure 8. See the caption for details on the various lines, though it is obvious that only a small LPC is required in the XEL direction. Please note that the estimated T_{sys} value assumes an aperture efficiency of $\eta = 0.6$ Anish Roshi et al. (2017). The plot is saved as a pdf in whichever directory made the call to PAF_Peak.py with the form
'PeakScan_Source_Direction_Timestamp'

6.1.2 plotDipolePower.py

The remaining routines are used to monitor the incoming data through the system. This particular script plots the total power 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., AGBT16B_400_14) and second being the associated time stamp for the scan (e.g., 2017_08_06_15:42:48). An example output plot is shown in Figure 9. The red number in the top right corner denotes the dipole number, and the green/blue solid lines denote total power from the YY/XX polarization. The particular scan being plotted came from a Peak scan as discussion in the previous subsection. We can see the nice response of the inner dipoles (i.e., 1, 3, 4, 6, 7), while dipoles such as dipole 8 showed little response due to a dead data channel. This plot is useful to ensure the live ele-

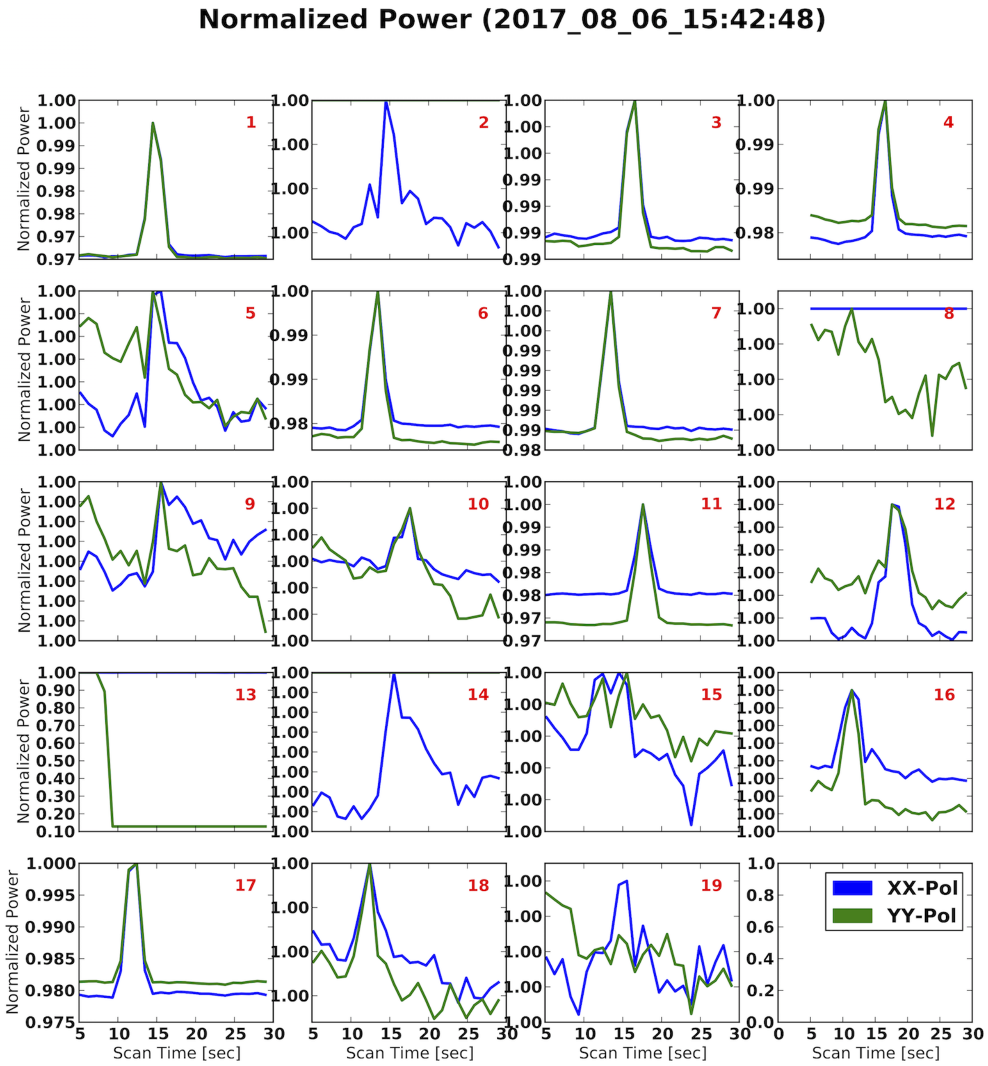


Figure 9: Example graphical output of plotDipolePower.py. Each panel represents the total power of a the associated dipole (red number) as a function of scan time for user provided scan. The green and blue solid lines respectively represent the YY and XX polarizations.

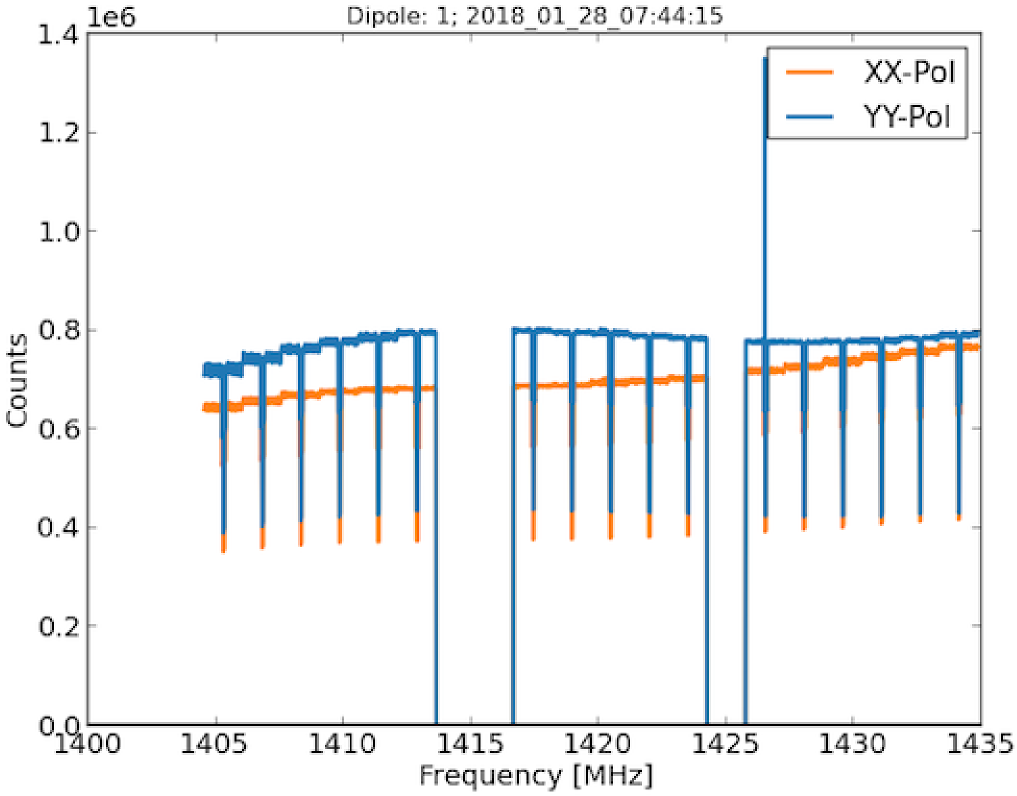


Figure 10: Example graphical output of plotBandpass.py. The XX and YY polarizations of the raw autocorrelation from the central dipole (dipole 1). Look to see the Galactic HI line is present as a positive indicator. This is also useful to look for signs of interference in the bandpass and the general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

ments 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.

6.1.3 plotBandpass.py

This script allows the user to see the time averaged bandpass (i.e., averaged over all integrations contained within an individual 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 finally averages over all integrations to produced 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 3dB 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 interference in either (or both) polarizations. Furthormore, this scan had several bank dropouts as evidenced by the drop in power near 1415 and 1425 MHz. An example call to create the output in Figure 10 is:

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

7 Reduction/Analysis Scripts

7.1 calcSysFlux.pro (Grid/7pt)

The first step in calibration of the raw beamformed spectra is to determine the scaling factor for the 'On/Off' power ratios for each individually 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` 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 Off scan of a seven point calibration scan when using `calcSysFlux_7pt.pro`, and the first Off 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

$$S_{\text{SEFD}} = \frac{S_{\text{CalSrc}} P_{\text{Off}}}{P_{\text{On}} - P_{\text{Off}}}, \tag{4}$$

where P_{On} and P_{Off} are respectively the On and Off 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 On and Off 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 On power values therefore only consists of those within the bandpass of that specific integration. The off 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 poor 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_{On}, P_{Off}, and S_{CalSrc}. The script will report the final SEFD and associated propagated uncertainty.

7.2 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 PFBCORR 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]
```

7.3 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 7.1. 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

$$S_{\rm BP} = S_{\rm SEFD}\left(\frac{P_{\rm sig} - P_{\rm off}}{P_{\rm off}}\right), \tag{5}$$

where $S_{\rm SEFD}$ is calculated from Equation 4 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'
```

7.4 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 7.2. 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:

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

7.5 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.

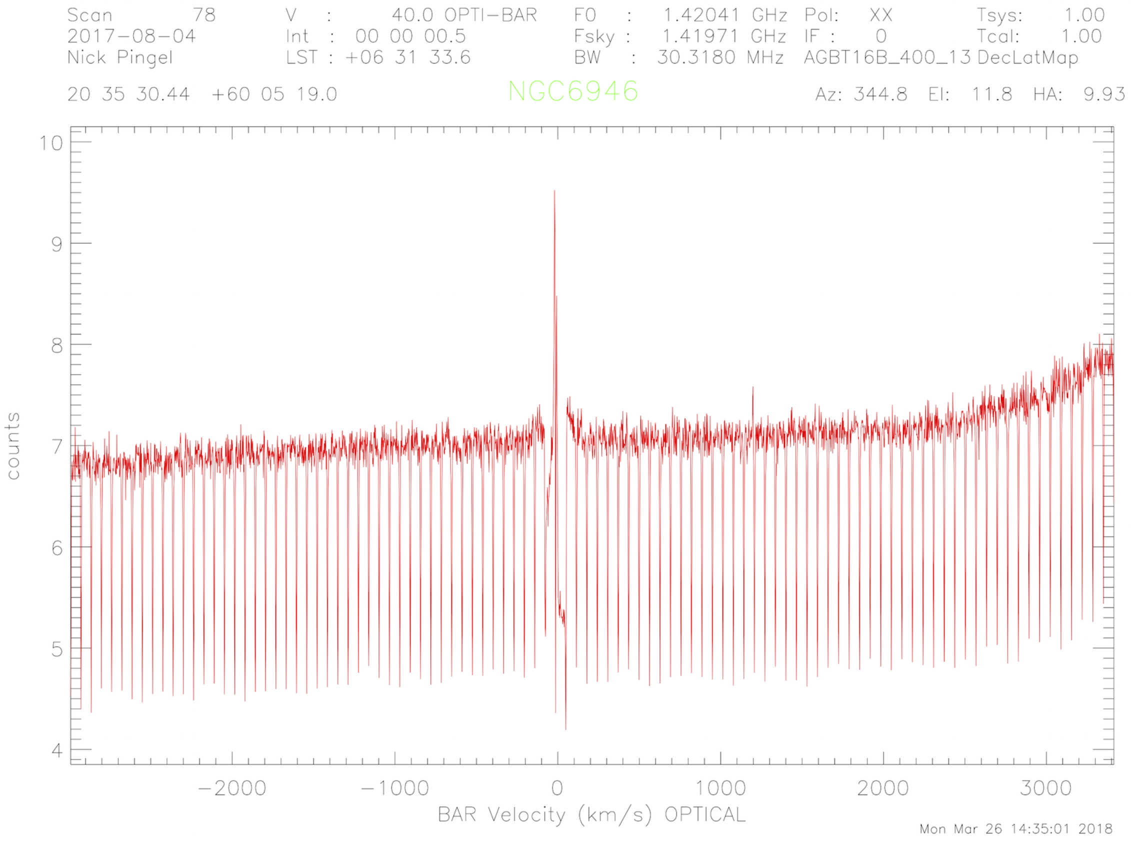


Figure 11: An example of an uncalibrated, beamformed spectrum as seen in a GBTIDL plotter window taken from the 35th integration of the 19th column of a *DecLatMap* scan of NGC6946. The 3 dB drop in power (i.e., 'scalloping') is an artifact of the two step PFB implementation of the backend (see text).

7.6 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

$$I_k^i = \left| w\left(\theta_i\right)^H \hat{a}_k\left(\theta_i\right) \right|^2, \tag{6}$$

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

$$\hat{a}_k\left(\theta\right) = \sqrt{\lambda_{\mathrm{max},k}}\hat{\mathbf{R}}_{\mathrm{off}}\tilde{\mathbf{v}}_k, \tag{7}$$

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

References

Anish Roshi, D., Shillue, W., Fisher, J. R., et al. 2017, ArXiv e-prints, arXiv:1711.02204

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

Prestage, R. M., Constantikes, K. T., Balser, D. S., & Condon, J. J. 2004, in Proc. SPIE, Vol. 5489, Ground-based Telescopes, ed. J. M. Oschmann, Jr., 1029–1040

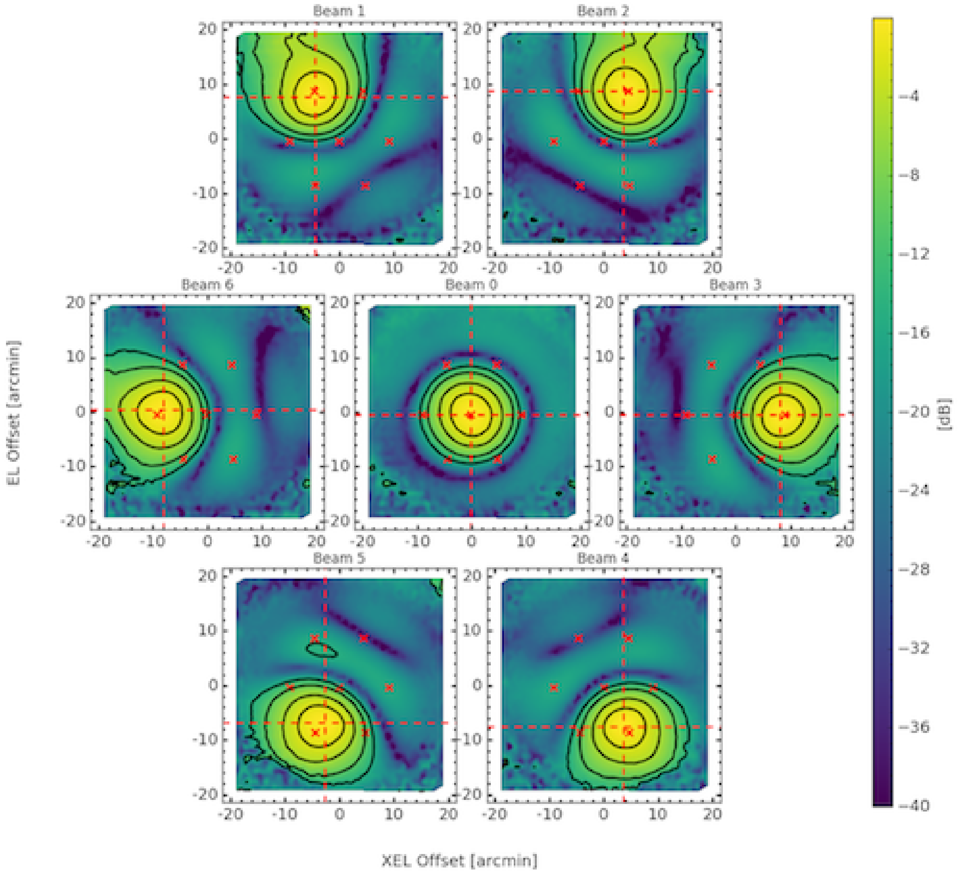
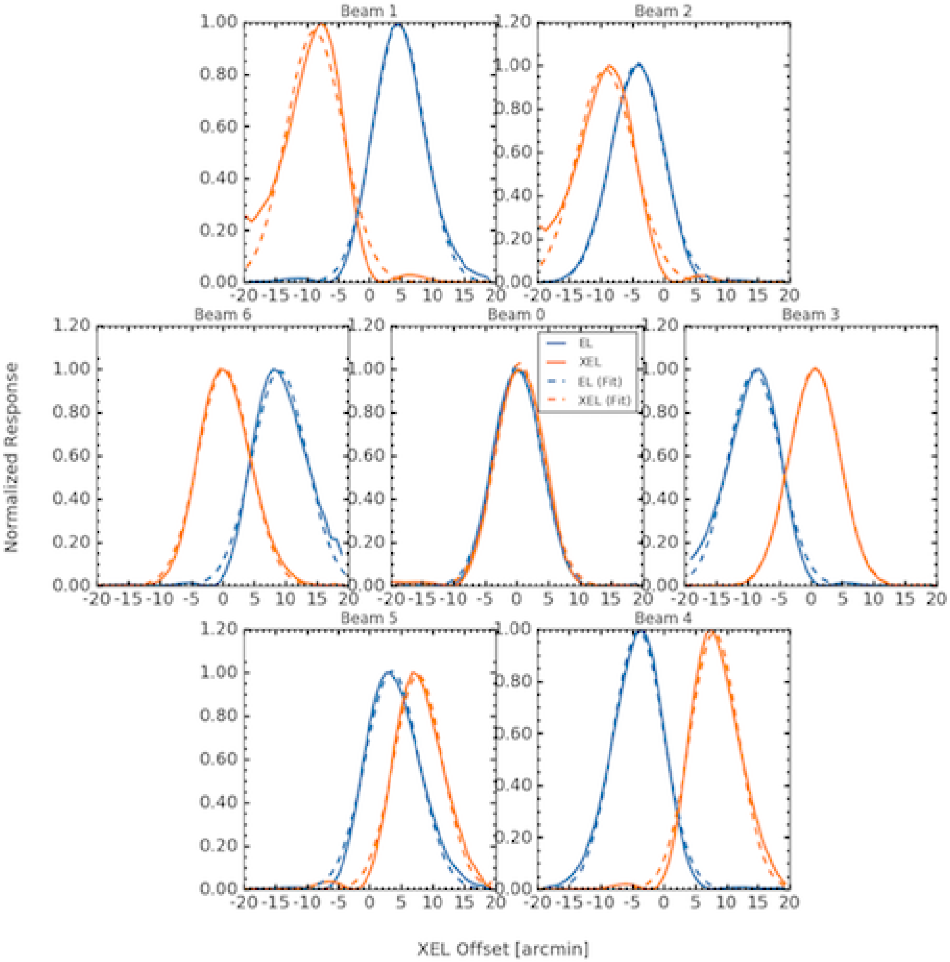


Figure 12: *above*: The formed beam pattern derived from a calibration grid The red x symbols denote the intended beam centers. The intersections of the vertical and horizontal dashed red lines denote the location of the peak response of each formed beam. The contours represent levels of −2, −5, −10, 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]
0.00
0.20
0.40
0.60
0.80
1.00
1.20
-20
-15
-10
-5
0
5
10
15
20