**USEFUL SCRIPTS FROM NIRUJ**

Currently, all work is done in /data/mohan/. Each data is downloaded into a directory named by its CBID with a symlink that describes the data. All analysis is done within these directories.

Green boxes are primarily code and yellow boxes are primarily output.

[**1. Downloading data from archive**](#_ac3ia75vi8l1) **2**

[**2. Writing data out to disk**](#_qj7sygw2netv) **2**

[**3. Correlator test scripts**](#_e6m8yyrsi2gw) **4**

[**4. Calibration within CASA**](#_igv0k9ev9ogy) **8**

[**5. Plotting CASA calibration files**](#_oa4b2vbj7zhp) **8**

[**6. Calibration with SDP**](#_wv8hwgfzspv) **10**

[**7. Compare data with bpcal model**](#_cpoalqa53645) **11**

[**8. Find jumps in time**](#_akh0u3h5atei) **12**

[**9. Stonehenge effect of Sun**](#_3y5f3xg7vfxn) **14**

[**10. Moving autocorr dips in time-channel via Radon transform**](#_qwjx6hwpbghh) **18**

[**11. Calibratability of 64-channel dips**](#_9r5yhljq5zta) **20**

[**12. Bands and channel flags**](#_dy0akpe1qg7k) **23**

[**13. List of scripts**](#_y0hnd1lg1p2b) **24**

I have uploaded some of my scripts referred [here](https://drive.google.com/drive/folders/1JoxONFHAxcp4lZJ2cDQ1sZc1vRXiUHIb?usp=sharing) (note that these are being updated regularly with small changes!)

Most of the scripts described below assume that basedir = ‘/data/mohan’ and that all files for a CBID are under *basedir/CBID*. All this would need to be changed depending on how you set up your system.

Note that all scripts are for python3 except those for CASA, which are for python2. Many of the diagnostic plots that are created may seem superfluous but each of them were necessitated by issues with specific datasets, and are useful for debugging.

## 1. Downloading data from archive

|  |
| --- |
| *cd /data/mohan*  python getdata.py <CBID> <DESCRIPTION> simple/full  *Example:*  *python getdata.py 1595971406 julcorr\_long\_l\_1k simple*  *Simple will download just the basic version of rdb and full will download all of it.* |

This creates, for example,

|  |
| --- |
| ls -l /data/mohan/1595971406  -rw-rw-r-- 1 nramanujam nramanujam 183667 Jul 29 02:29 1595971406\_sdp\_l0.rdb  -rw-rw-r-- 1 nramanujam nramanujam 0 Jul 31 18:29 'name=julcorr\_long\_l\_1k'  -rw-rw-r-- 1 nramanujam nramanujam 4228 Jul 31 18:29 printf |

The output of print(katdal.open(*rdbfile*)) is in the file print and the symlink name is also listed as a 0 byte file for reference. Other directories by the scripts below, as well as plots files, are written here as well.

## 2. Writing data out to disk

Since (1) some data will need to be read multiple times for various scripts and (2) we need to read all baselines data at the moment to access just the autocorrelations or a subset of data, we read in the dataset and write it out to disk on a scan by scan basis. This can take some time for 32K datasets. We can do this from within corrtests.py as well, as is described in the next section.

|  |
| --- |
| cd /data/mohan/1595971406  ipython  > import mylibs # this is a library of my routines  > mylibs.pickle\_new\_data('1595971406', extn='none')  *def pickle\_new\_data(num, nsave=True, getflags=True, dotimeavg=True, dofullauto=True, \*  *dofullcross=True, onlyauto=False, verbose=True, extn=None, basedir='/data/mohan/',*  *bchan=0, echan=0):*  *Read in a rdb file and writes out scan based files. Writes out numpy files by default. Writes out only XX and YY data for now.*  *num = CBID*  *nsave=True (default) writes out numpy files, else pickle files*  *getflags=True (default) will read in flags and 1. write out data after applying them and 2. write out flags*  *dotimeavg=True (default) will write out scan averaged auto and visibility data files separately*  *dofullauto=True (default) will write out the full auto corr data for each scan as well*  *dofullcross=True (default) will write out the full visibility data for each scan as well*  *onlyauto=False (default) will write out visibility data as well*  *verbose=True (default) will print out more info*  *extn=None (default) will assume the full rdb file. For simple file use extn='none'*  *basedir is the directory where the CBID folder is (default is /data/mohan)*  *bchan, echan are the channel range to write out data (default is full range)* |

For default parameters, it creates and writes out a structure like this.

|  |
| --- |
| > ls -ltr /data/mohan/1595971406  -rw-rw-r-- 1 nramanujam nramanujam 183667 Jul 29 02:29 1595971406\_sdp\_l0.rdb  -rw-rw-r-- 1 nramanujam nramanujam 0 Jul 31 18:29 'name=julcorr\_long\_l\_1k'  -rw-rw-r-- 1 nramanujam nramanujam 4228 Jul 31 18:29 printf  drwxrwxr-x 20 nramanujam nramanujam 4096 Jul 31 19:27 1595971406\_pickle  drwxrwxr-x 2 nramanujam nramanujam 10 Jul 31 19:11 calib  drwxrwxr-x 2 nramanujam nramanujam 4096 Aug 2 19:02 plots  > ls -ltr /data/mohan/1595971406/1595971406\_pickle/  drwxrwxr-x 2 nramanujam nramanujam 282 Jul 31 19:16 scan\_1  drwxrwxr-x 2 nramanujam nramanujam 282 Jul 31 19:17 scan\_2  drwxrwxr-x 2 nramanujam nramanujam 282 Jul 31 19:17 scan\_3  drwxrwxr-x 2 nramanujam nramanujam 282 Jul 31 19:18 scan\_4  etc  > ls -lhtr /data/mohan/1595971406/1595971406\_pickle/scan\_1  -rw-rw-r-- 1 nramanujam nramanujam 119M Jul 31 19:16 scan\_1\_cross\_scanav.npy  -rw-rw-r-- 1 nramanujam nramanujam 3.9M Jul 31 19:16 scan\_1\_auto\_scanav.npy  -rw-rw-r-- 1 nramanujam nramanujam 2.2G Jul 31 19:16 scan\_1\_fullcross.npy  -rw-rw-r-- 1 nramanujam nramanujam 2.2M Jul 31 19:16 scan\_1\_fullcrossdc.npy  -rw-rw-r-- 1 nramanujam nramanujam 2.2G Jul 31 19:16 scan\_1\_fullcrossflags.npy  -rw-rw-r-- 1 nramanujam nramanujam 36M Jul 31 19:16 scan\_1\_fullauto.npy  -rw-rw-r-- 1 nramanujam nramanujam 36K Jul 31 19:16 scan\_1\_fullautodc.npy  -rw-rw-r-- 1 nramanujam nramanujam 72M Jul 31 19:16 scan\_1\_fullautoflags.npy  The file names are self explanatory |

|  |
| --- |
| for fn in fns:  ...: d = N.load(fn)  ...: print(fn, d.shape)    scan\_1\_cross\_scanav.npy (2, 2, 1024, 1891)  Scan\_1\_auto\_scanav.npy (2, 2, 1024, 62)  scan\_1\_fullcross.npy (2, 74, 1024, 1891)  scan\_1\_fullcrossdc.npy (2, 74, 1891)  scan\_1\_fullcrossflags.npy (2, 74, 1024, 1891)  scan\_1\_fullauto.npy (2, 74, 1024, 62)  scan\_1\_fullautodc.npy (2, 74, 62)  scan\_1\_fullautoflags.npy (2, 74, 1024, 62)  Each data value is complex for visibility data.  Num of pol = 2  Number of dumps = 74  Number of channels = 1024  Number of antennas = 62  N\*(N-1)/2 = 1891  The first two files that are scan averages have the first index for the mean and rms respectively (ie, mean,rms X npol X nchan X nbl) |

## 3. Correlator test scripts

The single script that reads the numpy files, does a whole lot of analysis and outputs the result into a doc file automatically is now modularised for easy understanding and customisation. The value for basedir (‘/data/mohan/’ usually) is currently assigned inside the script which is called from *basedir/CBID*.

|  |
| --- |
| > python corrtests.py 1595971406 write/nowrite  Or run it inside ipython  Parameters are taken from corrtests\_config.py, which has  *doc = True # write docx file*  *basedir = '/data/mohan/' # where all the action happens*  *ring\_nmax = 10 # plot 10 worst offenders for 2-chan and 2-dump ringing*  *ring\_win = 31 # window length for detrending for 2-chan/dump ringing*  *fft\_win = 51 # window length for detrending for FFT to detect periodicities*  *fft\_thresh = 10.0 # threshold in FFT for detecting true periodicities*  *fft\_flag = 10 # flag the first fft\_flag indices in (shifted) fft before looking for peaks*  *fft\_nmax = 15 # plot the fft\_nmax highest peaks in FFT*  *detrend\_win = 51 # detrend each individual spectrum, used by many routines*  *cc\_thresh = 3.0 # threshold for flagging each scan av s.edpectrum before doing X-Y corr coeff*  *cc\_minlen = 10 # minimum length of flagged scan av.ed spectra for passing to X-Y corr coeff*  *comb\_mnn = 2 # minimum value of fold for comb analysis*  *comb\_nmax = 100 # maximum value of fold for comb analysis*  *comb\_nsearch = 1 # number of cycles to 'clean' the comb output of fold,start*  *comb\_minfold = 5 # go down to minfold while cleaning effects of all factors of fold of peak*  *comb\_thresh = 5.0 # threshold for SNR* |

If ‘write’ is used, then the data will be read from the rdb file and written out into scan based numpy arrays. This part is also described in the previous section, and is needed for the rest of the scripts to run. Each of the correlator tests are called sequentially in corrtests.py and almost all the tests are performed scan by scan, and on the scan averaged data. There is tremendous potential for parallelisation, which is not attempted yet.

This script will output a number of images in the /plot/ directory as well as two reports, and all plots in the reports can be found in the plots directory.

|  |
| --- |
| > ls -lh /data/mohan/1595971406/\*docx  -rw-rw-r-- 1 nramanujam nramanujam 3.4M Aug 2 19:02 1595971406\_fullreport.docx  -rw-rw-r-- 1 nramanujam nramanujam 37K Aug 2 19:02 1595971406\_summaryreport.docx  > ls -h /data/mohan/1595971406/plots  -rw-rw-r-- 1 nramanujam 166320 Aug 2 18:41 1595971406\_2dumpring\_dc.png  -rw-rw-r-- 1 nramanujam 59592 Aug 2 19:02 1595971406\_2powerplot.png  -rw-rw-r-- 1 nramanujam 48006 Aug 2 18:41 1595971406\_auto\_dc\_ringind.png  -rw-rw-r-- 1 nramanujam 137543 Aug 2 19:02 1595971406\_auto\_fold64n\_avspec.png  -rw-rw-r-- 1 nramanujam 75092 Aug 2 18:32 1595971406\_auto\_mean\_scan-ant-pol.png  .  .  etc |

These reports have outputs for the following (all of these are done for auto correlations and visibilities)

1. Summary of data, print(katdal.open(<file>))
2. Summary of flagging done
3. Spectral mean and rms for amp and phase
4. Ringing index over channels
5. Ringing index over time for DC
6. Spectral periodicities
7. XX-YY correlation in spectra
8. 64-channel folded spectra
9. Comb analysis

Look at corrtests.py for how to call each of these tests and their pre-requisites. A brief description of each section output in CBID\_fullreport.docx is given below. Note that all these tests are done on the uncalibrated visibility amplitudes (and auto correlations) and the phases are not looked at.

|  |
| --- |
| **Data parameters**  Lists the shape of the data array, and the number of antennas, baselines, channels and scans.  **Contents**  Lists the output of print(katdal.open(filename.rdb)). It also lists the bands used (see corr\_test\_chans.py).  **Flagging summary**  It lists the percentage of data flagged by SDP for each pol, for autocorrelations and visibilities. For both data, it plots the percentage of SDP flags as a function of channel, antenna/baseline and time (with the two pols overplotted).  **Spectral mean and variance**  Four plots are in this section - the mean and rms of the scan averaged autocorrelation and visibility amplitudes. Each plot has the mean/rms as a function of antenna/baseline index and scan number for each band (in corr\_test\_chans.py) and the two pols. The statistics are computed over channels (after detrending). The scan indices are actual indices from katdal.  **2-channel ringing**  A ringing index is defined as follows. Each sample is replaced by 1 if it is larger than the previous sample, and by -1 if smaller. Next, the even and the odd samples are separated out and added to each other. If there was ringing that was stronger than the noise, then every number in this sum would be zero. For a random series, we can predict the number of 0, 2 and -2 that is expected. If the total number of samples is N, and the number of zeros in the sum is Z, we define the ringing index R = 6Z/N - 2. The value of R will be 0 for no ringing, +1 for perfect ringing, and larger negative numbers for periods higher than 2. We assume that R>0.67 indicates the presence of ringing. The number of spectra in the scan averaged data that show 2-channel ringing are output as well as images of the ringing index as a function of antenna/baseline index and scan.  **2-dump ringing in DC**  Similarly, the ringing index is computed for the DC time series and the 10 examples with the highest ringing indices are also plotted along with the image of the index as before.  **Spectral periodicities**  For the scan averaged data, each detrended spectrum is FFTed and peaks more than a threshold (default is 10 sigma) are identified. The number of such spectra showing strong periodicities is output. Note that only periods that are less than the window size of the polynomial filter for detrending will be detected. For both auto and visibility amplitudes, the 15 spectra with the strongest periodicities (highest SNR in the FFT) are plotted along with their FFT amplitudes.  Next, the period (corresponding to the maximum FFT amplitude above the threshold) is plotted as a function of antenna/baseline index and scan for each band. No significant periodicity is represented by NaN. Next, a histogram of these significant periodicities is plotted, followed by plots of the FFT amplitude SNRs versus the periodicities (which are bounded by the detrending window length).  **Detrending**  The detrending window length for polynomial filtering for each individual spectrum is output on the doc and this detrending is performed (and this takes time). Note that this is done for scan averaged data). This detrended spectra are referred to hereafter.  **2-chan ringing in average detrended spectrum**  The ringing index for the average of the detrended spectra (after averaging over all baselines, times and pols)  **X-Y pol correlation in spectra**  Outputs the mean and median correlation coefficient between X and Y polarisation for auto and vis amplitudes over channels. The corr coeff over channel as a function of antenna/baseline are plotted. Before doing so, all data which are more than 3 sigma away from the median are flagged for each spectrum and this is to prevent RFI, spikes or dips to increase the correlation coefficient. Note that high correlation can be also caused by uncalibrated source structure in the field. The corr coeff as a function of antenna/baseline and scan number are also imaged. Finally, the average amplitude over antenna/baseline and time for X and Y are plotted against each other, as well as after 64-channel folding.  **64-channel folded spectra**  For each scan, a plot is created that shows the 64-folded spectrum averaged over all antenna/baseline and time (colours are the two pols), as well as for 128-fold and 256-fold, all on the left panel. On the right panel, the 128-folded spectra are plotted one below the other in two 64-channel parts, with a shift. Finally, the spectrum after averaging over all scans is also plotted after 128 folding.  **Comb function analysis**  For each pol, a comb analysis is done, where for every combination of start channel and fold period, we calculate the sum of the corresponding comb function multiplied by the (detrended antenna/baseline and time averaged) spectrum, normalised for equal variance. The resultant triangular matrix shows the most prominent period of the comb function and its phase, along with that of its harmonics, which is shown in the left panel in the figure below. The right panel shows the same data after removing all the harmonics. We expect that the most prominent pixel in the left panel is for a fold period of 64, and a start channel corresponding to the first channel (if present), and that the right panel is consistent with noise. The following figure shows the same effect but for combs of periods that are powers of two, following Thomas Abbott’s method. |

## 4. Calibration within CASA

First, we need to convert from rdb to MS.

|  |
| --- |
| > mvftoms.py -a --flags='static,cam,data\_lost' 1595971406\_sdp\_l0.rdb |

I only include CAM flags while converting to MS, and do not include the SDP flags. For 32K datasets, I sometimes include only channels 17500 to 25000 (-c 17500,25000) to save space and time.

The MS files are used to create bandpass and gaincal tables within CASA through a script that does the following, based on the inputs in **casa\_config.py**

1. Runs on *msfile*
2. Removes the WEIGHT\_SPECTRUM column if asked. Note that if already removed, doing it again will cause an error.
3. Flag certain antennas and scans based on the files flagsants and flagscans
4. Run gaincal every 8 seconds averaged over the channel range *gaincal\_bchan* to *gaincal\_echan* and using *myref* as refant
5. Run bandpass every 8 sec or every scan, depending on the parameter *bp*

|  |
| --- |
| CASA <1>: run -i docasa\_basiccal.py |

This creates the sn\_table\_\* and bandpass\_table\_\* files.

## 5. Plotting CASA calibration files

The gaincal and bandpass tables written by CASA can be plotted in many more useful ways than what CASA can do.

|  |
| --- |
| > mylibs.plot\_gaincal(caltable) |

This will create a number of images with the name of the gaincal table (*caltable*) in the filename, which plots

|  |
| --- |
| *1598912766\_Amp\_sn\_sdp.png and 1598912766\_Phase\_(-med)\_sn\_sdp.png*  The amplitudes and phases of each antenna gain (polarisations overplotted), with phases after median subtraction, all to the same scale.  *1598912766\_sn\_sdp\_figs.png*  Images of amp and phase gains as function of time and antenna (and phase after median subtraction as well).  *1598912766\_sn\_sdp\_phasecc.png*  The correlation coefficient between gain phases of every pair of antennas versus baseline length of that pair  *1598912766\_sn\_sdp\_plots.png*  The rms of gain amps and phases over antenna and over time  *1598912766\_sn\_sdp\_pol0\_highuv.png and 1598912766\_sn\_sdp\_pol1\_highuv.png*  The gain phases of the two antennas for pol X and Y for the 16 highest baseline lengths  *1598912766\_sn\_sdp\_pol0\_randomuv\_0.png, 1598912766\_sn\_sdp\_pol0\_randomuv\_1.png*  *1598912766\_sn\_sdp\_pol0\_randomuv\_2.png and 1598912766\_sn\_sdp\_pol0\_randomuv\_3.png*  The gain phases of the two antennas for pol Y for 16 random pairs within the baseline length ranges of [0,1000],[1500,2500],[3000,4000], [5500,8000] respectively. Similarly for pol Y for files with ‘pol1’  *1598912766\_sn\_sdpuv\_cc\_pol0.png and 1598912766\_sn\_sdpuv\_cc\_pol1.png*  The pairs of antennas in uv-place with the colour of each dot representing the correlation coefficient of the gain phases of the two constituent antennas for pol X and Y respectively.  *1598912766\_sn\_sdpxy\_cc\_pol0.png and 1598912766\_sn\_sdpxy\_cc\_pol1.png*  Antenna positions marked and (if > 1 km) connected by blue lines if the correlation coefficient of the gain phases of the two antennas are > 0.5 and red if < -0.5, for pol X and Y respectively. |

The bandpass table (caltable) of CASA is plotted as follows.

|  |
| --- |
| mylibs.plot\_bandpass(caltable) |

Which then produces the following plots,

|  |
| --- |
| *1595971406\_bandpass\_table\_scan\_avbp\_all.png*  This overplots the bandpass amplitudes of each antenna averaged over time  *1595971406\_bandpass\_table\_scan\_avbandpass.png*  This plots the   * bandpass amplitude averaged over all time, antennas and polarisations, * its detrended version, * the running mean (over 100 channels) of the detrended version, * the antenna-pol average of the rms of the bandpass amplitudes over time * Its running mean (over 100 channels)     *1595971406\_bandpass\_table\_scan\_bandpassplot\_N.png*  This plots for antenna N, the   * The image of the amp and phase of the bandpasses as a function of time channel and time (scan/dump) for both pols (first 4 plots) * The amp and phase as a function of time averaged over a range of channels * The amp and phase as a function of channel averaged over all time   The left panels are for pol X and the right panels are for pol Y |

Building on these two scripts, we also have *plot\_gain\_bandpass* which reads the bandpass table and sends the chan-averaged data to be plotted as gaincal solns instead.

|  |
| --- |
| mylibs.plot\_gain\_bandpass(caltable, chan) |

*plot\_twogains\_bandpass* reads the bandpass tables caltable1 and caltable2, and plot gains from chan1 averaged and chan2 averaged solutions respectively on top of each other.

|  |
| --- |
| mylibs.plot\_twogains\_bandpass(caltable1, caltable2, chan1, chan2, title) |

Lastly, *gainphasephase* reads the gaincal table, FFTs the gain phases and plots the phase of the detected ripples in any across antennas. This is useful to determine if the ionosphere is causing well defined ripples across time.

|  |
| --- |
| mylibs.gainphasephase(caltable) |

This creates the plot *CBID*\_phase\_ripple\_gainphase\_antposn.png.

## 6. Calibration with SDP

We can run gaincal and bandpass with SDP’s tools as well. Codes from Kim McAlpine have been used to build these scripts. sdp\_gaincal calculates the gains for a range of channels.

|  |
| --- |
| > run sdp\_gaincal.py *rdbfilename bchan dchan gaincal refant*  *rdbfilename = name of the rdb file in the directory*  *bchan = beginning channel number to average to calculate gains*  *dchan = average from bchan to bchan+dchan*  *gaincal*  *= dump => calculate gain for every dump*  *= scan => calculate gain for every scan*  *= int => calculate gain for solint of gaincal dumps*  *refant = refant antenna to use* |

The script calculates the gains and sends them, with metadata, to mylibs.plot\_gaincal which then creates the plots described in the previous section.

Similarly, the script sdp\_bandpass.py calculates the bandpasses using SDP routines for either every scan, every dump, or a certain number of dumps.

|  |
| --- |
| > run sdp\_bandpass.py *rdbfilename gaincal refant norm*  *rdbfilename = name of the rdb file in the directory*  *gaincal*  *= dump => calculate gain for every dump*  *= scan => calculate gain for every scan*  *= int => calculate gain for solint of gaincal dumps*  *refant = refant antenna to use*  *norm = False/True for normalising the bandpasses or not. Recommend False.* |

The script calculates the bandpasses and sends them, with metadata, to mylibs.plot\_bandpass which then creates the plots described in the previous section.

## 7. Compare data with bpcal model

It is now known that J1939 and J0408 need a detailed source model to accurately predict the observed visibilities. The script below does not utlise the model for calibration (which would be useful indeed), but just overplots the model visibilities over the data. This is useful to check whether the variations in the data are due to the source structure or not. This code can be parallelised over the model components since execution is slow.

|  |
| --- |
| > run calibmodel.py *rdbfilename source band bchan echan modeldir*  Eg: run ~/python3/tosend/calibmodel.py 1595971406\_sdp\_l0.rdb 0408 L 600 608 /home/nramanujam/python3/  *Source: 0408 or 1939*  *Band: L or U*  *Bchan, echan: channel range to average for model and data*  *Modeldir: directory where src\_band\_fl and src\_band\_posn files are*  Note that the script does not check if the data actually corresponds to the source mentioned. |

|  |
| --- |
| Once the script is run, plots of the model overlaid on the data (for amplitudes and phases) can be created using  > plotbl(bls, k)  plotbl is a function defined inside calibmodel.py  *bls: list of baseline indices to plot (eg, [1,30,100,300,1000,1300]*  *k: id for the figure filename plotted*  Creates  1595971406\_visph\_time\_fieldmodel\_Amp\_*k*.png and  1595971406\_visph\_time\_fieldmodel\_Phase\_*k*.png |

## 8. Find jumps in time

Some datasets exhibit sudden jumps in visibility amplitudes with time, and this could be either a spike or a dip, and last for either one dump or a few. This script detects a sharp jump in amplitudes for each baseline, averaged over a few channels, plots these time series, and also calculates and plots the statistics of their occurrences.

Each time series (pol and baseline) averaged over bchan-echan is detrended and examined for a jump that is statistically significant. Jumps are searched for a robust statistical deviation of 7, 6, and 5 stds successively, till a non-zero number of jumps are detected (if at all). Note that a large fraction of the total bandwidth needs to be averaged in order to detect these jumps clearly though they can be seen in the dynamical spectrum at times.

|  |
| --- |
| > run finddrops.py *rdbfilename bchan echan* |

This creates a number of plots and outputs.

|  |
| --- |
| **Output**:  The script is run twice, once for peaks and once for dips. For each of them, the average signal strength of the detected jumps and their occupancy fraction are output on the screen (and also on the plots themselves).  *1598912766\_up\_baddump\_file, 1598912766\_down\_baddump\_file*  This is an ascii file of all the baselines that have statistically significant jumps and each line has  “Scan number, pol (0/1), timestamp, baseline index, antenna1, antenna2”  Up and down are for spiked and dips respectively.  *1598912766\_up\_baddumps.pickle, 1598912766\_down\_baddumps.pickle*  This is a pickle file and for python3, you may need to  d= pickle.load(open('1598912766\_baddumps.pickle', 'rb'), encoding='bytes')  This has 6 components which are  d[0]: array of npol X nbaseline X ntime - 1 if that baseline had a jump, else 0  d[1]: array of npol X nants X ntime - 1 for every baseline with that antenna that had a jump, else 0  d[2]: names of the antennas used  d[3]: array of num of bad baselines X 15 - amplitude of [jump-2:jump+13] for each bad baseline  d[4]: list of list of lists; nscans X npol X nbadbaseline, where each inner list has the amplitudes of the jumps of all the detrended baselines for that scan and pol  d[5]: array of nscan X npol X baseline - has the median of vis amps over channels and time  Up and down are for spiked and dips respectively.  **Plots**:  *1598912766\_up\_dump\_scan\_N\_all.png, 1598912766\_down\_dump\_scan\_N\_all.png*  A plot of all detrended baseline time series for scan N (pol X and pol Y in two panels) and the mean overplotted as a black line. ‘Up’ is for spikes and ‘down’ is for dips.  *1598912766\_up\_dump\_scan\_N\_bad.png, 1598912766\_down\_dump\_scan\_N\_bad.png*  A plot of only those detrended baseline time series that have statistically significant detections of jumps for up/down. These jumps are also marked as red dots, the mean curve is the black line, and if no jumps are found, then that polarisation panel is blank.  *1598912766\_up\_baddumps\_templates\_av.png, 1598912766\_down\_baddumps\_templates\_av.png*  An average of all snippets of time series that have a jump for the entire dataset, starting from 2 dumps before the peak/dip and ending 12 dumps after.  *1598912766\_up\_jumpdumps\_baseline\_antenna\_vs\_time\_pol0.png, & also for down, and pol1*  Each of these are plots with the upper panel showing which time-baseline combinations show the number of detected jumps for up/down and pol 0/1. The bottom panel is the same but decomposed into antennas instead. Time-baselines with jumps will be shown as yellow.  *1598912766\_up\_jumpdumps\_num\_vs\_ant.png, 1598912766\_down\_jumpdumps\_num\_vs\_ant.png*  This is the same as the bottom panel of the previous figures, collapsed over the time axis. The two colours are the two pols. |

## 9. Stonehenge effect of Sun

There are a couple of scripts to analyse the effect of the Sun on data. The first script assumes that the data consists of multiple scans on the same source (eg J1939) whereas the later ones work scan by scan instead.

|  |
| --- |
| > run uhf\_sun\_process.py *g1 g2 b1 b2 id*  g1 and g2 are the start and end dump indices for the times not affected by the Sun and b1 and b2 correspond to the times affected by the Sun. id is the tag for the output file names.  If g1, g2, b1, b2, id are not present, then the times are calculated inside the ascript and id is taken as ‘def’ (default). Note that for long tracks which have more than one sunrise/sunset, the internally calculated g1, g2, b1, b2 might not be optimal and it is best to supply them as inputs. This can be used for L band too, if the value of chans is changed.  There are some input parameters that are currently inside the code, which needs to be exposed to the user. These are, with default values,  up=300; the zoomed-in plots of uv coverage are from -up to up  pw = 1.5; the power to which the rms is raised for plotting (see power)  nplot = 25; nplots highest rms values are plotted  maxbl = 1000; only baselines with index less than nplot are plotted  power = False; If True, the size of the circles is rms^pw, else rms  chans = 6000 for 32K UHF; data around this channel number are chosen (dchan = 150) |

|  |
| --- |
| The following plots are created in *plots/*  *1583247661\_stonehenge\_def\_el\_time.png*  The top plot shows the elevation of the Sun (deg) as a function of time in UTC hours. Note that we do not check for data which crosses 24 hrs. The bottom plot shows the elevation versus time dump index. The good times and bad times used in the script (either input as g1, g2, b1, b2 or calculated within) are marked in green and red respectively.  *1583247661\_stonehenge\_def\_pol0\_uv\_rms.png, 1583247661\_stonehenge\_def\_pol1\_uv\_rms.png*  The top row is for the bad times (sun affects the data) and the bottom row is for good times (sun below the horizon), that is, b1, b2, g1, g2. The left panels are the uv-coverage averaged over the time interval. The middle plot is a zoomed-in version. The right panels show the amplitude rms versus baseline index. Either the rms of rms^pw is used depending on whether power is True or False. The nplot baselines with the highest rms are plotted in orange. If the Sun affects the data, the orange dots in the top panel will lie along a line (and point towards the Sun).  *1583247661\_stonehenge\_def\_pol0\_bad\_good\_spec.png, 1583247661\_stonehenge\_def\_pol1\_bad\_good\_spec.png*  The nplot baselines with the highest rms are plotted for each pol. The blue plots are for the bads times (sun affects the data) and the orange ones are for the good times, for the same baseline.  *1583247661\_stonehenge\_def\_inneruv\_PA.png*  The inner core is plotted with the antenna positions and the nplot baselines with the highest rms are connected by red lines. If the effect of the Sun is strong, these red lines will be in the same direction. The right panel shows the histogram of the position angles of these baselines, and the position angle towards the Sun is marked by the horizontal black line, which should coincide with peak in the histogram.  *1583247661\_stonehenge\_def\_rms\_overchan\_fortime\_badbl.png*  The rms of amplitudes over channel (*chans-50:chans+50)* is plotted against the time for the *nplots* baselines and the sunrise/sunset time is marked by a vertical black line, The bad times (b1-b2) is also marked a horizontal red line. Since the nplots worst baselines depends on b1 and b2, this plot too will depend on b1 and b2. The corresponding plot for the script *sun\_byscan.py* plots all baselines and hence does not suffer from this.  *1583247661\_stonehenge\_def\_uvcov\_forsun.png*  This is the most useful plot of all. This plots the inner uv-plane as projected towards the Sun, with each point taken as the average for the bad times b1 to b2. The colours of the dots represent the rms for each baseline. The uv corresponding to 1.3 times the optical solar diameter is drawn as a black circle. For data affected by the Sun, the dots inside the circle would have higher rms (towards red) and the surrounding dots would be lower (blue).  *1583247661\_stonehenge\_def\_angspacing\_badbl\_time.png*  This plot is now redundant because of the previous plot and can be ignored. |

The next script is also meant for data that are scans for a single source, or for one target in a multi-source data. This script plots various parameters as a function on scan number.

|  |
| --- |
| > run sun\_sdp\_process.py  This script reads inputs from sun\_sdp\_config.py which has  *bchan and echan* - if not found, they are chosen by the script depending on the band and number of channels, unlike the previous script.  *srcnum =* The source number to do the analysis for. If not found, it is 0 (first/all)  *antmax* = maximum number of antennas to use. If not found, takes all  *uvmax =* maximum uv distance to consider. If not found, takes 8000 m  *scanav =* This is a list of number of scans to average over and plots are made for each scanav. If not found, it is taken as [1,2,3,4] => plots are made after averaging over 1, 2, 3, and 4 scans. This is useful if the scan duration is very short etc.  It also reads *flagscans.py* which has  flagscans = 1,3,4,5 and is a list of sdp scan numbers to flag.  The computation is done inside sun\_sdp.lib.py |

The outputs are as follows

|  |
| --- |
| The plots are of the form 1573754530/stonehenge\_ampph\_polP\_rms\_scanav\_A\_n.png, where P is 0, 1; n is 1, 2 and 3 and A runs through scanavs. In all these plots, **note** that the uv coordinates are those projected towards the Sun.  n=1  This plots the rms of the amplitudes (top) and phase (bottom) and their zoomed-in version (right), coloured by rms (blue<red). The three circles mark the uv corresponding to 0.7, 1.0 and 1.3 times the optical solar diameter.  n = 2  This plots the rms of the amplitudes (top) and phases (bottom) as a function of uv distance (right is zoomed-in). The three vertical lines are 0.7, 1.0. 1.3 times the solar radius. The points are coloured by scan.  n = 3  This plots the rms of all baselines for amplitude (top) and phase (bottom) as a function of scan (or averaged scans), coloured by uv distance (blue< red). This plot is also useful to figure out if some scans or antennas need to be flagged. |

Finally, we have the script that is used for data which can have any number of sources and the analysis is done per scan independently. This has been used to analyse the data taken at UHF and L with varying distance to the Sun and Sun’s elevation. The details of the analysis and example plots are shown in the relevant report.

|  |
| --- |
| > run sun\_by\_scan.py id flag/noflag  *id* is the identifier for the file names and if the second argument is *flag* then myflags are read in, else no flags are applied.  Two parameters are currently coded inside the script which need to be exposed to the user.  *chans* = centre of the RFI free channel range  *dchan* = twice the number of channels on either side of chans to analyse  It also reads uhf\_sun\_config.py which has the form  flagants = [‘m000’, ‘m033’]  If not found, then no antennas are flagged.  All these inputs need to be put in one place, of course. |

A text file is output which has the following format (note that all uv distances are as projected towards the Sun hereafter)

|  |
| --- |
| Each line is for one scan, and has  *(name, el, num, slope, height, pa, snr, xs2, xs3, num3, scan) x 2*  once for each pol, in a single line, where  name = name of the source  el = Elevation of the Sun for that scan  num = number of baselines less than 50 m (projected)  slope = fitted slope of rms vs uv distance for a power law  height = height of the fitted power law  pa = position angle of the Sun for that scan  snr = SNR of the height of the fit  xs2 = uv distance corresponding to 2 sigma above the median rms value  xs3 = same for 3 sigma  num3 = number of baselines within xs3 metres  scan = scan number |

|  |
| --- |
| For example, for scan 167,  *1609465615\_stonehenge\_norm\_pol0\_scan167\_uv\_rms.png, 1609465615\_stonehenge\_norm\_pol1\_scan167\_uv\_rms.png*  The projected uv coverage and the rms versus uv distance (all projected)  *1609465615\_stonehenge\_norm\_pol0\_scan167\_uvcov\_forsun.png , 1609465615\_stonehenge\_norm\_pol1\_scan167\_uvcov\_forsun.png*  The inner part of the projected uv plane, with the rms being colour coded, and the circle corresponding to 1.3 optical sun.  *1609465615\_stonehenge\_norm\_scan167\_rms\_overchan\_fortime\_badbl.png*  The amplitude rms over channels as a function of time for the two pols (top and bottom).  *1609465615\_stonehenge\_norm\_scan167\_profile.png*  The rms versus projected uv distance for the two pols (top and bottom) in blue, with the fits as orange. The right panels are in log scale. The vertical and horizontal lines correspond to num3 and xs3. |

It also outputs plots that summarise the entire observation, which are more useful.

|  |
| --- |
| Output plots for the entire dataset  *1609465615\_stonehenge\_norm\_el\_time.png*  The top panel shows the Sun’s elevation versus UTC and the bottom panel shows the Sun’s elevation versus dump index. Note that the former does not account for crossing 2400.  *1609465615\_stonehenge\_slopes\_heights.png*  The top panels are for pol XX and the bottom, for YY. The left panels shows the fitted slope to the rms versus projected uv distance as colour coded dots as a function of distance to the Sun and the Sun’s elevation. The right panels show the fitted heights for the same.  *1609465615\_stonehenge\_heights\_snr.png*  The top panels are for pol XX and the bottom, for YY. The left panels shows the fitted heights to the rms versus projected uv distance as colour coded dots in log scale as a function of distance to the Sun and the Sun’s elevation. The right panels show the equivalent SNR, in log scale, for the same  *1609465615\_stonehenge\_av\_heights\_log.png*  Similar plot as before, but plots the slopes, heights and SNR (the latter in log scale), averaged over the two pols.  *1609465615\_stonehenge\_uvcutoff2.png*  *1609465615\_stonehenge\_uvcutoff.png*  *1609465615\_stonehenge\_uvcutoff1.png*  *1609465615\_stonehenge\_av\_heights.png*  *1609465615\_stonehenge\_xsigma2.png*  *1609465615\_stonehenge\_xsigma3.png*  *1609465615\_stonehenge\_num3.png*  *1609465615\_stonehenge\_uvcutoff3.png*  *1609465615\_stonehenge\_uvcutoff2\_2sigma.png*  *1609465615\_stonehenge\_uvcutoff2\_3sigma.png*  *1609465615\_stonehenge\_num\_pa.png*  *1609465615\_stonehenge\_corr\_pol0.png*  *1609465615\_stonehenge\_corr\_pol1.png* |

## 10. Moving autocorr dips in time-channel via Radon transform

In 2019, many datasets showed periodic dips in the autocorrelation of antennas as a function of time, which also moved in channel space. Some of these would move along a curve in the time-channel plots and even double back to form a U-shaped curve. This script detects all such dips using a version of the pulsar dedispersion algorithm, or a simpler version of the Radon Transform, wherein sections in the time-channel space are de-dispersed for various ‘DMs’ and strong signals which indicate a quasi-linear movement in this space are detected. This can be used to look for similar signals in all data (including visibility amplitudes with some modifications).

For reasons lost in antiquity, the script is capable of doing the analysis for groups of scans separately. This is because different time periods of a dataset could exhibit different types of tracks in the time-channel plane. Given the prominence of RFI in the wide band, the entire analysis is done for distinct sub-bands taken from corr\_test\_chans.py (see later section).

The parameters for the Radon transform are read from a file. The current implementation is parallelised over groups using pprocess but can be undone easily.

|  |
| --- |
| **Parameters**  Input parameters for the Radon transform are read from delaytracks\_config.py, which overwrites the default values within the script delaytracks\_detect.py if present. These are (with defaults)  *docalib=True: normalise by the average response along channel and time for each baseline*  *bx=None, ex=None: beginning and end of time index to process*  *flatten=True: If True then piecewise 2-d 2nd order polynomials are fit to boxsizeXboxsize squares of the dynamic spectra after robust flagging, and subtracted, for detrending.*  *boxsize=50: Size of the box over which to fit a 2d polynomial and subtract, to detrend*  *thresh1=3.0: Pixels > thresh times robust estimator of rms are replaced from a normal distribution before fitting with a 2d polynomial*  *threshneg=-4.0: pixels below this threshold for robust estimator (after detrending) are dips*  *maxdm = 100: max displacement of spectrum of last dump w.r.t. first dump, in channels*  *peaksnr=10.0: minimum (robust) SNR for peak in the Radon transform to be considered*  *minsnr=20: minimum value for (normalised) peak in the Radon transform to be considered*  *robust=False: get statistics for each Radon transform plane using mean/rms or robust stats*  *niter=3: number of iterations for kappa-sigma clipping to obtain robust statistics*  *doabs=False: search only for positive outliers in kappa-sigma clipping for robust statistics*  *dmdiff=3: Nearby points above sidesnr that differ from main peak DM by dmdiff are associated with the same peak*  *sidesnr=3.0: SNR down to which associate points near a peak in the Radon transform with the peak itself (and remove), within dmdiff*  *doflag=True: After applying my mask (corr\_test\_chans.py), flag again for narrow band RFI*  *doplot=False: If True, create a plot of Radon transform for each scan and band*  **Running the scripts**  > mylibs.write\_antnames(CBID, extn)  This writes out the ant name file in CBID\_pickle.  If extn=’none’, it will look for the abridged rdb file else it will look for the full file.  > mylibs.sep\_auto\_ant(CBID, scans, extn)  Writes out the autocorrelation data per antenna instead of per scan, in CBID\_pickle.  If extn=’none’, it will look for the abridged rdb file else the full file.  If scans=None, it will read all scans together in one group. If multiple groups of scans are needed, they can be specified as scans = [[0,1,2], [3,4,5,6,7]] etc.  > run delaytracks\_detect.py *rdbfilename maxDM s1 s2 s3 s4 ...*  *Needs write\_antnames and sep\_auto\_ant to have been run first.*  rdbfilename is the name of the rdb file  maxDM is the maximum slope (DM) to search for dips in the time-channel space  If s1 etc are present then only these scan numbers are processed, else all scans are processed. |

This creates a number of plots, one set for each group. For each group, we have,

|  |
| --- |
| 1564042678\_delaydipslopes\_groupN\_bandM\_polK.png  where  N is the number of the group of scans (0, 1, …)  M is the sub-band that is analysed (0, 1, ...)  K is the pol (0/1)  Each plot has panels only for those antennas whose autocorrs show these dips, and each such dip in the Radon transform is marked by a dot. The x-axis is time in dumps and the y-axis is the slope of the dips in sec/MHz. Note that the maximum detectable slope is set by maxdm. |

## 

## 11. Calibratability of 64-channel dips

This script was originally meant to see if the 64-channel dips could be calibrated (yes, they can be). However, they can also be used to see the efficacy of bandpass calibration as a function of the time interval between the bpcal scan and the target scan. The script has been run on data that consists of only long tracks on a single calibrator but it can be used for any dataset (see config files for details).

This script is to be run inside CASA and it first runs gaincal and applies it and splits each scan into a separate file. It then calculates bandpasses for each scan. Next, each bandpass table is applied to every scan, and the average amplitude spectrum (scalar average of just the amplitudes) and its rms are calculated for each scan-bandpass combination. These spectra are detrended using a polynomial filter and also folded over 64 channels, and written into files. Another script then reads these files and creates plots.

Note that the script docasa\_64chan\_proper.py is in python2 since I have been using CASA <6. Also be aware that his script takes a long time to execute (~1 day) if the number of scans is large (need to calculate N bandpasses and N^2 applycal). This can only be parallelised if the bandpasses are first calculated, copied into multiple directories, and multiple CASAs are run for groups of scans. This is not painless.

The last plot described below can be used to determine how often must one bandpass calibrate as a function of increase in noise that one is okay with (see caveat).

|  |
| --- |
| CASA <1>: run -i docasa\_64chan\_proper.py  This reads docasa\_64chan\_config.py which has  *fname = '1596017061\_sdp\_l0.ms' # name of MS file*  *bscan = 1 # start scan to process*  *escan = 6 # end scan to process*  *win = 100 # window for detrending by polynomial filter as number of channels*  *bchan = 10000 # Beginning channel number of MS file if split (for flagging known RFI)*  *gbchan = 0 # bchan for gaincal with respect to MS file channels*  *gechan = 64 # echan for gaincal with respect to MS file channels*  This outputs a set of files names op\_64chan\_\* and ispec\_64chan\_\* (scalar and vector average respectively). |

The output files are then read by a python script called read\_avspec\_casa64chan.py (which is in python3). This assumes that the original dataset came from either 32K or 4K data. Parameters are read from read\_avspec\_casa64chan\_config.py. Note that currently, the output files are written as pickle files, and hence the syntax of reading them will depend on python2 versus 3. The scripts here assume that they are written in python2 and read in python3. When the script plots the rms as a function of the time interval separating the bandpass scan and the data scan, it assumes that all scans are of equal duration. This will need to be modified when that is no longer so.

|  |
| --- |
| In [1]: run read\_avspec\_casa64chan.py op/ispec  op = scalar average of amplitudes and ispec = vector average  This reads the file read\_avspec\_casa64chan\_config.py which has  nch = 32 # Nk chans in original dataset, can be 32/4  cbid = '1586078244' # CBID of dataset  bad1 = [] # list of scans to flag (for both the bandpass and applied-to scan)  bad2 = [] # list of scans to flag in 2nd run (for both the banspass and applied-to scan)  scanl = 10.0 # scan length in min (to calculate time between two scans; assumed constant) |

Bad1 and bad2 are lists of scans to ignore (for both the banspass and applied-to scan) and there are two of them for historic reasons. Plots are created for bad1 and bad2 separately (but only if they are non-empty lists). If you don’t like it, assign them as = []. Usually these bad scans are chosen from the outliers in *CBID\_rms\_scans\_bp\_inhrs\_end\_flag0\_flag0.png* and other plots below.

A number of plots are created.

|  |
| --- |
| In the figure names below., flag0 represents bad1 (and flag1 for bad2)  *1596017061\_64chan\_dip\_SNR\_RMS\_end\_pol0\_flag0.png, and \_pol1\_*  This is the main plot to be looked at. The top panel plots the SNR of the 64 channel dip in the 64-folded detrended amplitude spectrum averaged over all baselines as a function of scan. The blue line marks the SNR in the raw scan without bandpass correction. If the 64-channel dips are present, this curve would be around -10 to -15. The orange plot is for each scan after applying its own bandpass, which is always around zero since the 64 channel dips are closure quantities. The other curves are after applying a particular scan bandpass. That these curves are also around zero indicates that the 64 channel dips are calibratable.  The bottom panel shows the RMS of each detrended amplitude spectrum averaged over all baselines as a function of scan (but not 64-folded). These are normalised by the variation of rms as a function of scan. As expected, the blue curve for the raw uncalibrated spectra is the highest, the orange curve for the scan after applying its own bandpass is the lowest, and the other curves corresponding to applying the bandpasses of each scan are in between the two. Each of the latter curves will touch the orange curve for the ‘self’ bandpass.  *1596017061\_rawrms\_scan.png*  This plots the rms of the detrended amplitude spectrum averaged over all baselines as a function of scan, with the two colours being the two polarisations. The bottom panel of the first two figures above are normalised for this variation.  *1596017061\_avspec\_raw\_afterflag\_flag0.png*  Each panel shows the spectrum of the detrended amplitude spectrum averaged over all baselines for a scan (and the two colours are for the two pols), all to the same scale. The rms of these spectra are what are plotted in the bottom panel in blue, in the first two figures.  *1596017061\_avspec\_selfbp\_afterflag\_flag0.png*  This is similar to the previous plot except that this shows the spectra of each scan after applying its own bandpass to itself. The rms of these spectra are what are plotted in the bottom panel in orange, in the first two figures.  *1596017061\_rms\_matrix\_flag0.png*  This shows a matrix (for the two polarisations) representing the rms (plotted in the first two figures) of each scan (y-axis) after applying the bandpass of each scan (x-axis). As expected, the diagonal (applying a scan’s bandpass to itself) is much lower than the rest.  *1596017061\_avspec\_rawself\_afterflag\_fold\_pol0\_flag0.png and \_pol1\_*  The top left panel plots the raw (un-bandpass calibrated) detrended baseline-averaged spectra for each scan, and the average is over-plotted in a thick black line. The top tight plots shows the same but for the spectra of each scan after applying its own bandpass, with the average overplotted as a thick black line. The bottom plots show the average plots of the top panels after folding over 64 channels. If the 64-channel dips are present, this would be seen clearly in the bottom left plot (and not in the bottom right plot since it is antenna-decomposable/closure).  *1596017061\_rms\_scans\_bp\_inhrs\_end\_flag0\_flag0.png*  This summarises the efficacy of bandpass calibration as a function of duration between the bpcal scan and target scan. The previous plots showed the rms of the final spectrum of each of the N scans after applying each of the N bandpasses. The non-self N(N-1) RMSes are plotted as a function of the time separating the bandpass scan and the applied-to scan, in hours. The right panel is a zoom-in of the left panel. This can be used to determine how often must one perform bandpass calibration. As expected, it is roughly constant for about 2 hours, worsens over 2-8 hours, and is then roughly constant hence. Scans to be flagged (bad1, bad2) are chosen from the outlier scans in this plot.  **Caveat**: we plot the rms of the detrended amplitude spectra averaged over baselines (and not the amplitude of the vector average). |

## 12. Bands and channel flags

All scripts described above use specific bands for analysis and also use channel masks for known RFI. These are taken from corr\_test\_chans.py. Though this file has bands and masks for 1K, 4K and 32K separately, the scripts read those corresponding to 32K and convert them for 1K and 4K. These are only available for L band currently though.

|  |
| --- |
| **mynband**['l'][32] = 2  **mybands**['l'][32] = [[4200,8360],[17500,25000]]  **mybandflags**['l'][32] = [[[6530,6790],[7010,7160],[7460,7540],[7850,8010]],[[19920,20320],[21440,21760],[24250,24420],[21090,21110],[19170,19270],[19420,19450],[20950,21050],[18600,18650]]]  **myfullband**['l'][32] = [17500,25000]  **myfullbandflags**['l'][32] = [[19920,20320],[21440,21760],[24250,24400],[21090,21110],[19170,19270],[19420,19450],[20950,21050],[18600,18650]]  **mybands**: the two sub bands used for some of the analysis above (band0 and band1 in corrtests.py, e.g.)  **myfullband:** the standard band used for most analysis (including a lot of corrtests.py)  **myfullbandflags:** list of [bchan,echan] channels that should be flagged due to known RFI. |

## 13. List of scripts

**Stand-alone scripts**

1. **calibmodel.py**: Calculates the visibilities for an input calibrator source model and overplots on data
2. **corrtests.py:** Runs all the tests for correlator test and outputs a docx report with plots
3. **delaytracks\_detect.py**: Detects drifting tracks in the time-channel plane for auto correlations using Radon transforms.
4. **finddrops.py:** Detects sudden temporal spikes or dips in the visibility amplitudes and plots them as well as antenna, baseline and time occupancy of these jumps.
5. **sdp\_bandpass.py:** Calculates bandpass table using SDP routines and calls the plotting scripts (see below)
6. **sdp\_gaincal.py:** Calculates gaincal table using SDP routines and calls the plotting scripts (see below)

**CASA scripts**

1. **docasa\_64chan\_proper.py**: Bandpass calibrates every scan, applies each bandpass table to every scan and writes out corresponding spectrum (used by **read\_avspec\_casa64chan.py**)
2. **read\_avspec\_casa64chan.py**: Runs in ipython, reads output of **docasa\_64chan\_proper.py** and creates plots for the calibratability of 64-channel dips and applicability of bandpass tables with time
3. **docasa\_basiccal.py**: Does gaincal and bandpass cal inside CASA (see next section for plotting them)

**Inside mylibs.py - major functions**

1. **gainphasephase:** reads gaincal table, FFTs phases and plots the phase of the detected ripples across antennas
2. **comb\_analysis:** does comb analysis for a 1d array for various values of start and fold period
3. **comb\_analysis\_zoom:** does comb analysis for non-integer fold periods
4. **pickle\_new\_data:** read an rdb file and write out multiple numpy (or pickle) files for each scan. Is used by corrtests.py
5. **plot\_bandpass:** reads bandpass table from CASA or SDP and created pretty yet useful plots
6. **plot\_gain\_bandpass:** reads bandpass table and send the channel averaged data to be plotted as gaincal solns
7. **plot\_gaincal:** reads gaincal table from CASA or SDP and creates pretty yet useful plots
8. **plot\_twogains\_bandpass:** reads bandpass tables caltable1 and caltable2, plot gains from chan1 averaged and chan2 averaged solns respectively on top of each other