Cotter

Cotter is the name of André Offringa's pre-processing pipeline for MWA data. A description of cotter can be found in: The Low-Frequency Environment of the Murchison Widefield Array: Radio-Frequency Interference Analysis and Mitigation , Offringa et al., 2015, PASA, 32, e008. Cotter is used by the MWA ASVO to run conversion jobs and is also used by various MWA science teams to pre-process raw visibilities before passing the data onto their pipelines.

The most important actions Cotter performs are (in order):

- Apply mapping from input nr to tile number/polarisation index and from coarse channel to frequency
- Correct the phases for the cable length delay
- Calculate u,v,w coordinates and other required meta data
- Shift the phases from zenith to the pointing centre
- Correct for digital gains of the coarse channels and (try to) correct the coarse-channel filter shapes created by the poly-phase filters.
- Flag invalid channels (the DC channels and the outer most channels of each coarse channel) and the first 4 seconds (corrupt because of delay in tile initialization)
- RFI detection (for which it uses the AOFlagger library)
- Calculate some statistics
- (Optionally) average in time or frequency direction, calculate weights

Cotter is primarily used to convert the raw output of the MWA telescope ("gpubox files" containing visibilities) into an intermediate format for calibration; either the CASA measurement set or UVFITS. Bear in mind that André's "calibrate" program requires measurement sets for calibration. It is possible to convert UVFITS files to a measurement set format via CASA.

Source Code

You can find the source code on github: MWATelescope/cotter

Usage

Just run "cotter" to see the up-to-date help information. This is also a (crude) way to determine if Cotter is installed and working. A more complicated example:

OBSID=1065880128

MEASUREMENT_SET=$OBSID.ms

TIME_AVERAGE=4

FREQ_AVERAGE=40

EDGEWIDTH=160

MAX_MEMORY_GIGABYTES=40

```
cotter -absmem $MAX_MEMORY_GIGABYTES \
       -o "$MEASUREMENT_SET" \
       -m "$OBSID".metafits \
       -timeres $TIME_AVERAGE \
       -freqres $FREQ_AVERAGE \
       -edgewidth $EDGEWIDTH \
       -allowmissing \
       ./??????????_*gpubox*.fits
```

After you have created a measurement set with Cotter, you can view its associated statistics with the 'aoqplot' tool (part of the AOFlagger package). You can run it like:

```
aoqplot preprocessed.ms
```

You can combine the statistics of multiple sets by specifying multiple measurement sets on the command line, e.g.:

```
aoqplot *.ms
```

Compression

Cotter4 can make use of 'Dysco' compression, which I'll describe briefly here. To make use of compression, Dysco needs to be separately installed, available at https://github.com/aroffringa/dysco . On Github you can also find some info on the Dysco Wiki and there's a paper describing the technique. Compression is only used when explicitly requested. The Dysco software is not needed when running Cotter without requesting compression. One can request compression by adding '-use-dysco' to the commandline. A run with default compression settings looks like:

```
cotter4 -use-dysco -m 1077974936_metafits_ppds.fits -timeres 4 -freqres 80 *gpubox*.fits
```

The advanced compression parameters can be set with a call like this:

```
cotter4 -use-dysco -dysco-config 12 12 TruncatedGaussian 1.5 RF -m 1077974936_metafits_ppds.fits -timeres 4 -
freqres 80 *gpubox*.fits
```

This uses the following settings: use 12 bits per datafloat, 12 bits per weight float, optimize for a Gaussian distribution truncated at 1.5 sigma and use row-frequency normalization -- see cotter4 options & Dysco wiki for more info about these).

The compression works best on high time and frequency resolution, and as such it is ideal for archival data. In some cases it can also be used to speed up IO. The Dysco compression technique works by a combination of normalization, quantization and dithering, which together adds uncorrelated noise to the data. This noise behaves like system noise. Therefore, as long as the added noise is insignificant compared to the system noise in a single snapshot, it won't affect long integrations either. The default settings (first example command above) compresses with approximately a factor of 4, which is good enough to not affect the noise by more than a couple of percent in any observation and with averaging to <= 4 sec and <= 80 kHz. At the highest time/freq res, the added noise is less than 0.1%.

Flag-only output

Cotter can output "mwaf" files, which are specially-formatted FITS files that only contain the flags found by the AOFlagger. The RTS will be able to read these files. This mode can be selected by outputting to a file with extension .mwaf. The filename has to include '%%', which will be replaced by the gpubox number, for example: "cotter -o myobservation-%%.mwaf -m ....".

About the file format: the files contain FITS headers with some meta info. The flag file will not have separate flags for separate polarizations; each sample represents all 4 polarizations, as they are kept the same anyway. The information is stored in a binary table in HDU 2. Each table row contains the flags for all channels for a specific combination of baseline and timestep. There are nbaseline x ntimesteps rows in the file, where timestep is the most-significant index.

Example code for reading the flag files in python:

```
#!/usr/bin/env python
import pyfits,numpy
hdulist=pyfits.open('1078413440_01.mwaf')
# binary table is in 2nd HDU, first HDU is just metadata
d = hdulist[1].data['FLAGS']

nchan=hdulist[0].header['NCHANS']
nant =hdulist[0].header['NANTENNA']
ntime=hdulist[0].header['NSCANS']
nbl  = nant*(nant+1)/2

# shape of data is (ntimes*nbaselines,nchan)
# where nbaselines includes autos. Time index varies most slowly
dr = d.reshape(ntime,nbl,nchan)

# dump first time step of flags to file
dr[0,:,:].tofile('/tmp/dumppy.dat')

# get the mean flagging per antenna
dbaseline = dr.mean(axis=2).mean(axis=0)

# this will have nant x nant entries, including
# flags for the autocorrelations
dtile=numpy.zeros((nant,nant))
k=0
for i in xrange(nants):
    for j in xrange(i,nants):
        dtile[i,j]=dbaseline[k]
        dtile[j,i]=dbaseline[k]
        k+=1
```

References

The following conference paper gives a broad overview on the steps taken in AOFlagger. The numbers in it are a bit outdated, but the pipeline is still the same and helpful to get a first impression: - "A LOFAR RFI detection pipeline and its first results" http://arxiv.org/abs/1007.2089

The flagger contains two algorithmic steps which differentiates it from other flaggers. The first one is the sumthreshold algorithm, which is a combinatorial thresholding algorithm that looks for lines in the field. This is combined with a high-pass filter to not flag on the signal. Both background and sumthreshold algorithms are described and tested in this paper: - "Post-correlation radio frequency interference classification methods" http://arxiv.org/abs/1002.1957 There's also a technical report on how to implement the SumThreshold[?] in a fast vectorized way: http://www.astro.rug.nl/~offringa/SumThreshold.pdf

Another algorithmic step does a morphological selection. It works like a dilation, but is scale invariant. This algorithm is described & validated in the following paper. It also describes a fast way to do this; we hand't found the faster algorithm yet when we wrote the first quoted paper. - "A morphological algorithm for improving radio-frequency interference detection" http://arxiv.org/abs/1201.3364

Finally, the following paper describes the result of applying the full pipeline on LOFAR data. The full pipeline is also analysed further, e.g. the false-positive rate is determined. - "The LOFAR radio environment" http://www.aanda.org/articles/aa/abs/2013/01/aa20293-12/aa20293-12.html

I plan a similar "MWA radio environment" paper at some point, where Cotter will be in described too.

Cotter used to use values that I measured using a 32-tile commissioning array calibrator scan, back in 2012, by stacking subbands and normalizing the resulting passband. We are currently (April 2014) planning to change this to exact values derived from the pfb coefficients by Alan Levine. These are his memo and his gains:

- MEMO_CascadedFT_2012_05_25.pdf
- MEMO_PFB_Effects_2012_06_04.pdf
- sim2n_totpwr_dat.txt: Alan Levine's bandpass gains used to create his relative power figure (dead link)
- MwaPfbProtoFilterCoeff2009_512x8.dat - Filter coefficients used to calculate the PFB gains

The first column in sim2n_totpwr_dat.txt is the fine frequency channel, and the third column is the bandpass gains in db. Cotter takes relative power, so a conversion must be used to get the bandpass gains in the right format.