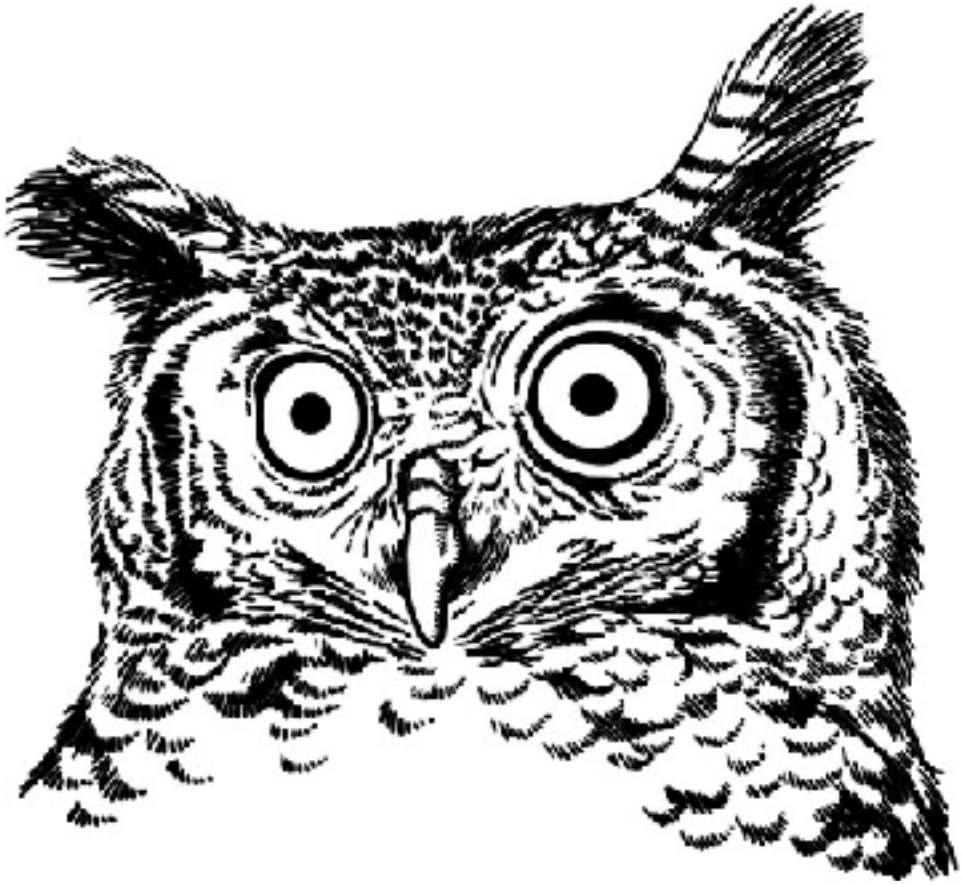
1st Edition



RTS

IN A NUTSHELL

A Desktop Quick Reference

RTS Processing Pipeline in a Nutshell

Installation

The RTS processing pipeline should be stored in the users local bin path (with appropriate read and execute permissions):

    $HOME/bin/mwa_pipe.py

The local environment should be set up as per the sample .bashrc shown at the end of this document - this ensures that paths are set correctly and the appropriate modules are loaded.

A default global sky catalogue should be made available in $RTS_CAT_PATH with the name rtscat.txt. Alternate catalogues can also be stored here and accessed via the pipeline.

The Basic RTS workflow

The typical work flow involves three stages:

- Download the visibility data from the archive
- Calibrate the visibility data
- Image the visibility data

The RTS typically works on a per-snapshot basis and identifies snapshots by their obsid (sometimes referred to as a gpsid). To determine which snapshots to download, refer to the MWA archive (http://mwa-metadata01.pawsey.org.au).

Within a snapshot, the RTS processes coarse channels (1.28 MHz channels) on separate CPU or GPU nodes. Thus when processing the entire band a total of 24 + 1 nodes is required (the additional node is for the master).

Working with slurm

slurm is the queuing system on the galaxy cluster at pawsey.

sbatch script - used to submit a job to the queue.

sbatch —dependency=afterany:jobid script - used to submit a job to the queue, the job will only run after the job with the specified jobid is complete. This is useful for setting up a download-calibrate-image workflow where each step much complete before the next step is initiated.

squeue - used to list jobs submitted to the queue. This is useful to see if a job has started and to see what job id has been assigned to a submitted job.

scancel jobid - used to cancel a job.

Downloading data from the archive

The following is a sample shell script to start the download process (qload.sh):

```
#!/bin/bash -l
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --ntasks-per-node=1
#SBATCH --time=12:00:00
#SBATCH --partition=workq
#SBATCH --account=mwasci
#SBATCH --export=NONE
```

```
rm -f load.log
./load.py
```

Note that loading data only requires a single core. The RTS load script (load.py) has the following format:

```
#!/usr/bin/env python
from mwa_pipe import *

rts = MWAPipe("load")
# list of obsids to process
obslist = ["1166109584", "1166109704"]
for obsid in obslist:
     # Fetch the data from the archive
     rts.fetch_data(obsid)
     # Reflag the observation
     rts.reflag(obsid, 1.3)
```

The fetch_data command requests the pipeline to download the data for the specified obsid. The reflag command decompresses the Cotter flag files for the specified obsid, checks the flagging occupancy for each channel and flags any channels that have increased levels of flagging. The value specified after the obsid indicates a factor above the median number visibilities per channel flagged above which a channel will be entirely flagged.

To initiate the download process simply submit to the queue as follows:

```
sbatch qload.sh
```

Calibration

The following is a sample shell script to start the calibration process (qcal.sh):

```
#!/bin/bash -l
#SBATCH --nodes=25
#SBATCH --ntasks-per-node=1
#SBATCH --time=12:00:00
#SBATCH --partition=gpuq
#SBATCH --account=mwasci
#SBATCH --export=NONE
rm -f cal.log
./cal.py
```

Note that the calibration requires 25 nodes to process the entire observing band. Calibration can be performed in either GPU or CPU mode. If using the GPU then the gpuq queue must be used otherwise the workq queue must be used. The RTS calibration script (cal.in) has the following format:

```
#!/usr/bin/env python
from mwa_pipe import *

rts = MWAPipe("cal")

rts.update_cal = True
rts.ncalibrators = 1
rts.template_base = "rts_"
rts.cat_extent = 20.0
rts.cal_cadence = 64
rts.cal_srcs = 30
rts.rts_bin = "rts_gpu"
# Find the list of download obsids in the range specified
obslist = get_local_obs_list("1166109584", "1166109704")
for obsid in obslist:
     # Clean out any old files
     rts.clean(obsid)
     # Calibrate the obsid
     rts.calibrate(obsid)
```

The update_cal and ncalibrators parameters should be set to True and 1, respectively - these indicate that the calibration solutions will be updated and that one calibrator sources will be used, respectively. It is important to note that the calibrator source used

is actually a hybrid source that is created specifically for this obsid. The hybrid calibrator is created by extracting cal_srcs number of sources from the global sky model that are within cat_extent x cat_extent degree region centred on the beam former pointing centre. The template_base provides a prefix for the RTS configuration template files used to generate the actual RTS input files needed for calibration and/or imaging. cal_cadence specifies the number of time stamps within the data set to use for the purpose of calibration. rts_bin specifies the RTS binary to use for calibration, for GPU processing this should be set to rts_gpu, for CPU processing this should be set to rts_cpu.

The clean command cleans out any non-critical files that were left over from previous runs of the calibration/imaging process. The calibrate command initiates the calibration process. Internally, this generates a local sky model (based on the parameters set in cal_srcs and cat_extent), creates an RTS input file and then initiates the calibration process using the RTS.

The calibration phase works best if there is a single, central, simple and dominant source within the beam former pointing. Any departure from this will result in a less ideal solution. Worst case scenarios include bright sources near the edge of the field or complex extended sources anywhere within the field of view - such sources often result in calibration failure.

To initiate the calibration process simply submit to the queue as follows:

```
sbatch qcal.sh
```

At the end of the calibration process there should be a BandpassCalibration_node*.dat file and a DI_JonesMatrices_node*.dat file for each available coarse channel in the data set.

Checking Calibration

TODO: Mention the plotbpall.py script - generates bandpass plots. Look for amplitudes of ~1.0 for XX/YY and ~0.0 for XY/YX. Any spikes, wobbles or biases are probably due to bad calibration. Script tries to suggest tiles that might need to be flagged. Also, bpsummary.py provides a quick summary of the bandpass solution. If any tiles are believed to be bad then these are reported in decreasing priority. The tile numbers listed here can be recorded in a flagged_tiles.txt file (one tile number per line) and this file should be copied to each observation id so that the RTS can use it to forcibly flag those tiles before performing calibration and imaging. TODO: What to do if things go wrong - sky model might be insufficiently accurate (any fuzzy, blobby, resolved things might cause trouble). Sometimes calibration for a field fails if something pops in a sidelobe - in

this instance, perhaps copy over successful bandpass from other times or transfer solution from point-like calibrator source.

TODO: What to do if flagging information is not available? If flag files are not available for a data set then rts.use_flag should be set to False. Furthermore, a flagged_channels.txt file should be created with the fine channel numbers that need to be flagged and this file needs to be copied to each observation id. For 40 kHz bands, the flagged_channels.txt file should flag channel 0, 1, 16, 30, and 31 at the very least (these same channels will be flagged in each coarse channel). For 20 kHz bands this should be 0, 1, 2, 3, 32, 60, 61, 62, 63 and for 10 kHz bands this should be 0, 1, 2, 3, 4, 5, 6, 7, 64, 120, 121, 122, 123, 124, 125, 126, 127. Each fine channel number should be specified on a new line.

Basic imaging

The following is a sample shell script to start the imaging process (qimage.sh):

```
#!/bin/bash -l
#SBATCH --nodes=25
#SBATCH --ntasks-per-node=1
#SBATCH --time=1:00:00
#SBATCH --partition=gpuq
#SBATCH --account=mwasci
#SBATCH --export=NONE
rm -f image.log
./image.py
```

Note that the image process requires 25 nodes to process the entire observing band. Imaging can be performed in either GPU or CPU mode for natural weighting schemes but the uniform and robust weighting schemes are currently only supported in CPU mode. If using the GPU then the gpuq queue must be used otherwise the workq queue must be used. The RTS imaging script (image.in) has the following format:

```
#!/usr/bin/env python
from mwa_pipe import *

rts = MWAPipe("image")
rts.weighting = "natural"
rts.field_size = 8.0
rts.ncalibrators = 1
rts.template_base = "rts_"
#
rts.fscrunch = 1
rts.img_cadence = 96
#
rts.do_accumulate = False
#
rts.min_baseline = 50.0
rts.make_psf = False
rts.cat_extent = 6.0
rts.npeel = 0
rts.niono_cal = 0
rts.img_srcs = 100
rts.update_cal = False
```

```
#
# Test set for transient
#
rts.rts_bin = "rts_gpu"
rts.img_cat = None
# Find the list of download obsids in the range specified
obslist = get_local_obs_list("1166109584", "1166109704")
for obsid in obslist:
     # Image the obsid using in-field calibration performed
earlier
     rts.image(obsid, cal_id = None, ra_hrs = 1.6504278, dec_deg
= -17.9501, dest_image_path = "uvceti")
```

The template_base provides a prefix for the RTS configuration template files used to generate the actual RTS input files needed for imaging. img_cadence specifies the number of time stamps at a time to use within the data set to use for the purpose of imaging (if img_cadence is a multiple of the number of integrations in a snapshot then that multiple set of image cubes will be generated. rts_bin specifies the RTS binary to use for imaging, for GPU processing this should be set to rts_gpu, for CPU processing this should be set to rts_cpu. Note that only natural weighting is supported by the GPU version of the RTS.

The weighting parameter can be used to set the weighting scheme (natural, uniform or robust). For robust weighting the robustness can be set with the robustness parameter. do_accumulate specifies whether weights should be accumulated over the entire band (needed for uniform and robustness weighting if generating full band images). A separate accumulate step is needed to generate the weighting files before imaging if using uniform or robust weighting.

fscrunch specifies the number of neighbouring channels to average together. If set to 1 then the data will be imaged using the finest available channel resolution (usually either 40 kHz or 10 kHz depending on the observing set up). fscrunch can not be set to a value large than the number of channels in a coarse channel (generally 32 for 40 kHz channels and 128 for 10 kHz channels).

min_baseline is used to set the shortest baseline length (in wavelengths) to be used for imaging, max_baseline is used to set the longest baseline length (in wavelengths).

Peeling is enabled by setting the cat_extent (where the sources will be selected from a cat_extent x cat_extent degree region centred on the beam former pointing centre) and a local sky model of img_srcs sources will be generated. From this catalogue, npeel specifies the number of sources to directly subtract from the visibilities, niono_cal specifies the number of sources to use for full ionospheric calibration (flux and position determined) and subtraction.

The img_cat command specifies whether the local sky model should be regenerated or alternatively whether it should be copied from another specified obsid. This is useful if you want to maintain a consistent set of sky models for peeling.

The image command performs the imaging. By default, the correlated image centre is used as the image centre, alternatively, the image centre can be explicitly specified with ra_hrs and dec_deg. cal_id specifies the calibrator to use. If set to None then a previously determined calibration solution for that obsid is used. If an obsid is specified then a calibration solution is copied from that obsid instead (useful if the field is complex but a simpler calibrator field is available). The final parameter indicates the directory in to which the image cubes will be copied to (within this path each obsid will have its own directory within which the image cube is stored).

To initiate the imaging process simply submit to the queue as follows:

```
sbatch qimage.sh
```

At the end of the imaging process there should be an image cube for each of the requested obsids in the destination path provided.

Checking Imaging Results (TODO)

TODO: Use imstats.py to check noise levels (particularly Stokes V is a good indicator). Can integrate over frequency channels to generate higher SNR images and to ensure that noise levels reduce by ~sqrt(nChannels).

Different weighting schemes (TODO)

Natural

Uniform

Robust

Image Data (TODO)

Describe image data

How to integrate

HealPix data

Other Helper Tools (TODO)

bpsummary.py - reads in the bandpass files in the current observation id directory and lists a summary of potentially bad tiles that were not flagged in the metadata.

collapseobsids.py index <prefix> - create integrated full-Stokes images for the indexed obsid using only images with the specified prefix. If no prefix is provided then “2” is assumed (images generated from the RTS). If using reprojected images then use “r” as the prefix.

finddata.py <prefix> - a simple script to count the number of fits files with the specified prefix in each obsid sub-folder in the current path. By default, the script looks for 2*.fits files (RTS-generated images). It can also be used to see how many visibility files have been downloaded from the archive by providing the prefix “1”. Similarly, it can be used to check how many reprojected images there are by providing the prefix “r”.

imint.py destination image-list — integrate a series of images

imstats.py image-list — determine basic statistics for a series of images (minimum, maximum, std.dev).

plotallbp.py - create a summary document of the bandpass solutions generated by the RTS. The script should be run in the directory where the bandpass solutions are to be found and also requires the meta fits file to be in that directory. The output is a “BandpassSummary.pdf” document that contains an amplitude and phase plot for all instrumental polarisations for each coarse channel.

project.py index - reproject all channel images in the indexed obsid to the local template image.

rmsynth.py prefix startPhi endPhi dPhi doClean writeCube - perform RM synthesis on image files in the current directory with the specified prefix. The RM cube will have a phi range from startPhi to endPhi with dPhi resolution. If doClean=1 then the first peak will be deconvolved with the RMSF. If writeCube=1 then the resulting RM cube will be written. Output files are peak.fits (peak polarised intensity at each pixel of the RM cube), val.fits (the phi at which the peak occurs for each pixel), PI.fits and PIphi.fits (the absolute value of the RM cube and the polarisation angle of the RM cube). If doClean=1 then all output files will be prepended with “c”.

sflag.py index threshold — determine the median rms noise in Stokes V of the indexed obsid and flag any channel that has (channel noise > threshold * median).

Command Reference

Accumulate: *obsid1[-obsid2]*

Perform uv weighting accumulation for the specified obsid(s). This step is necessary if using non-natural weighting schemes and if the intention is to integrate images across the entire observing band.

calibrate(*obsid)*

Generate a local sky model for the specified obsid and calibrate based on this sky model.

clean(*obsid)*

Remove all temporary and/or generated files from the specified obsid e.g. calibration files, log files, images, etc.

fetch_data(*obsid)*

Download GPU box files, metadata and Cotter flag files from the archive for the specified obsid

fetch_metadata: *obsid*

Regenerate metadata for the specified obsid.

image(obsid, cal_id, dest_image_path, ra_hrs, dec_deg)

Generate image cubes for the specified obsid. If no image centre is supplied (i.e. rahrs and decdeg) then the pointing centre will be used. cal_id specifies the obsid to use for a previously generated calibration solution (via calibrate), alternatively if self is specified then the imager will use the previously generated calibration solution for the obsid i.e. in-field calibration.

reflag(*obsid, threshold)*

Analyse the Cotter flag files for the specified obsid and flag any channels that exceed a median flagging occupancy by more than a factor specified by the threshold.

pair_reflag: *obsid1 obsid2 threshold*

pair_reflag performs the same function as reflag but ensures that obsid1 is flagged in the same manner as obsid2. This is particularly useful for LST-matched difference imaging to ensure similar visibility coverage.

uv_dump: *obsid*

Output a uv-fits format data set for the specified obsid.

Parameter reference

cal_cadence: *integrations* (Default: 64)

The number of integrations to use for calibration

cal_cat: *obsid* | *None* (Default: None)

If None is specified a new local sky model is generated, otherwise a previously generated catalogue is copied from the specified obsid.

cal_srcs: *sources* (Default: 25)

The number of source components to include in the local sky model for calibration.

cat_extent: *degrees* (Default: 20)

The extent of the sky in degrees that should be included when generating a local sky model.

data_path: path (Default: None)

Sets an alternate location where the archival data is stored. All files generated by the pipeline will still be created in the working directory where the pipeline is started but archive data can be read from another location. This is useful if a common data directory is used by multiple users or if it is more suitable to store the archival data on a different file system.

do_accumulate: *True* | *False* (Default: False)

Accumulate weights across all coarse channels

end_at: *integration* (Default: 0)

The integration to end imaging at.

field_size: *degrees* (Default: 2.0)

The size (in degrees) of the image to generate during imaging.

fscrunch: *nchan* (Default: 4)

The number of fine channels to average when generating output images.

img_cadence: *integrations* (Default: 96)

The number of integrations to use for imaging.

img_cat: *obsid* | *None* (Default: None)

if None is specified a new local sky model is generated, otherwise a previously generated catalogue is copied from the specified obsid.

img_srcs: *sources* (Default: 300)

The number of sources to include in the local sky model for imaging.

long_taper: *wavelengths* (Default: None)

Set a Gaussian taper to down weight longer baselines.

make_psf: *False* | *True* (Default: False)

Generate PSF images instead of observed images.

make_stokes: *False* | *True* (Default: True)

Generate Full Stokes images in addition to instrumental polarisation images.

max_baseline: *wavelengths* (Default: None)

The maximum baseline length (in wavelengths) to be used for calibration/imaging.

min_baseline: *wavelengths* (Default: None)

The minimum baseline length (in wavelengths) to be used for calibration/imaging.

ncalibrators: *ncal* (Default: 1)

The number of calibrators to use during calibration (normally 1).

niono_cal: *ncal* (Default: 0)

The number of sources to include as part of the ionospheric calibration.

npeel: *npeel* (Default: 0)

The number of sources to peel.

nside: *n* (Default: 2048)

The Healpix NSIDE to use for regridding.

regrid: *False* | *True* (Default: False)

Regrid images into the HealPix frame.

regrid_method: 0-3 (Default: 3)

Method used to perform regridding (2 is faster, 3 is more accurate).

remove_inst_pols: *False* | *True* (Default: True)

Remove instrumental polarisation images if they are generated during imaging.

robustness: *robustness* (Default: +2.0)

The robustness to use for robust weighting.

rts_bin: *rts_cpu* | *rts_gpu* (Default: rts_gpu)

Specifies whether the RTS should be run in GPU mode or CPU mode.

short_taper: *wavelengths* (Default: None)

Set a Gaussian taper to down weight short baselines.

src_cat: *cat_file* (Default: rtscat.txt)

rtscat.txt

start_at: *integration* (Default: 0)

The integration to start imaging at.

template_base: *prefix* (Default: rts_)

The prefix used to find the RTS template files for calibration and imaging.

update_cal: *nup* (Default: False)

False

use_flag: *True* | *False* (Default: True)

Use the Cotter flagging data downloaded from the archive.

use_meta: *True* | *False* (Default: True)

Use the metadata from the metadata files retrieved from the archive (this is historical and should always be set to True).

weighting: n*atural* | u*niform* | r*obust* (Default: natural)

The image weighting scheme to use.

Sample .bashrc file for RTS set-up.

```
#
export HISTSIZE=10000
test -s ~/.alias && . ~/.alias || true
#
export MWA_OPS_DIR=/group/mwaops
export MWA_SCI_DIR=/group/mwasci
#
# switch to GCC/GNU enironment
module switch PrgEnv-cray PrgEnv-gnu
module unload gcc
# use gcc 4.8 for CUDA/RTS, use version 4.9 for boost
module load gcc/4.8.2
module load cray-libsci
module load cmake
module load scipy
module load lapack
module load cudatoolkit
module load astropy
module load cfitsio
module load boost
module load casacore
module load ephem
module load readline
module load psycopg2
module load gsl
module load matplotlib

# set path to CFITSIO library
export CFITSIO_DIR=/ivec/cle52/galaxy/devel/PrgEnv-gnu/5.2.25/cfitsio/3370
source ~/MIRRC.sh

# set path to RTS binary
export RTSDIR=$MWA_OPS_DIR/CODE/RTS
export RTSBIN=$RTSDIR/bin

# set path to path where global sky models can be found
export RTS_CAT_PATH="$HOME/catalog/"

# set up library paths
export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:$MWA_OPS_DIR/CODE/lib"
export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:$CRAY_LD_LIBRARY_PATH"
export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:${MIRLIB}"

# set up binary paths
export PATH="${PATH}:${MWA_OPS_DIR}/CODE/bin:${RTSBIN}:${MIRBIN}"
export PATH="${PATH}:${MWA_SCI_DIR}/code/MWA_Tools/scripts"

# set up Python paths
export PYTHONPATH=${PYTHONPATH}:$MWA_SCI_DIR/code/MWA_Tools/
export PYTHONPATH=${PYTHONPATH}:$MWA_SCI_DIR/code/MWA_Tools/scripts
export PYTHONPATH=${PYTHONPATH}:$MWA_SCI_DIR/code/MWA_Tools/mwapy
export PYTHONPATH=${PYTHONPATH}:$MWA_SCI_DIR/code/MWA_Tools/configs
export PYTHONPATH=${PYTHONPATH}:$HOME/bin

# env vars for psql client
export PGPASSWORD=BowTie
export PGHOST=mwa-metadata01.pawsey.org.au
export PGDATABASE=mwa
export PGUSER=mwa
```

Sample RTS calibration template: rts_cal.in

```
//---------//
FscrunchChan=4

SubBandIDs=

StorePixelMatrices=0

MaxFrequency=180
ImageOversampling=3

applyDIcalibration=1
doMWArxCorrections=1
doRFIflagging=1
useFastPrimaryBeamModels=1

CorrDumpTime=0.0
CorrDumpsPerCadence=0
NumberOfIntegrationBins=0
NumberOfIterations=

StartProcessingAt=0
StartIntegrationAt=0

// In correlator mode, Base File name is used to find correlator files.
BaseFilename=*_gpubox

doRawDataCorrections=1
ReadGpuboxDirect=1
UsePacketInput=0
UseThreadedVI=0

ArrayFile=array_file.txt
ArrayNumberOfStations=128

ChannelBandwidth=0.04
NumberOfChannels=32

ArrayPositionLat=-26.70331940
ArrayPositionLong=116.67081524

//time is needed to set lst, even if ObservationTimeBase is set.
ObservationTimeBase=2456519.20083
```

```
// --- observing stuff --- //
ReadAllFromSingleFile=
ObservationFrequencyBase=167.055

ObservationPointCentreHA=1.031
ObservationPointCentreDec=-25.93

ObservationImageCentreRA=
ObservationImageCentreDec=

// --- weighting stuff --- //
calBaselineMin=20.0
calShortBaselineTaper=50.0

// --- calibration stuff --- //
DoCalibration=
SourceCatalogueFile=catalogue.txt
NumberOfCalibrators=1
```

Sample RTS imaging template: rts_img.in

```
//---------//
//ImagePSF=1
FscrunchChan=4

SubBandIDs=
StorePixelMatrices=1

MaxFrequency=180
ImageOversampling=5

applyDIcalibration=1
doMWArxCorrections=1
doRFIflagging=0
useFastPrimaryBeamModels=1

CorrDumpTime=0.0

CorrDumpsPerCadence=0
NumberOfIntegrationBins=0
NumberOfIterations=

StartProcessingAt=0
StartIntegrationAt=0

// In correlator mode, Base File name is used to find correlator files.
ReadAllFromSingleFile=
BaseFilename=*_gpubox

doRawDataCorrections=1
ReadGpuboxDirect=1
UsePacketInput=0
UseThreadedVI=1

ArrayFile=array_file.txt
ArrayNumberOfStations=128

ChannelBandwidth=0.04
NumberOfChannels=32

ArrayPositionLat=-26.70331940
ArrayPositionLong=116.67081524

//time is needed to set lst, even if ObservationTimeBase is set.
ObservationTimeBase=2456519.20083
```

```
// --- observing stuff --- //
ObservationFrequencyBase=167.055
ObservationPointCentreHA=1.031
ObservationPointCentreDec=-25.93

ObservationImageCentreRA=
ObservationImageCentreDec=

// --- weighting stuff --- //
calBaselineMin=20.0
calShortBaselineTaper=50.0

// --- calibration stuff --- //
DoCalibration=
SourceCatalogueFile=catalogue_imaging.txt
NumberOfCalibrators=1
NumberOfSourcesToPeel=0
NumberOfIonoCalibrators=0
UpdateCalibratorAmplitudes=0
```

Sample u-v export RTS template: rts_uv.in (not tested!)

```
//---------//
FscrunchChan=4

SubBandIDs=

MaxFrequency=200
ImageOversampling=5
writeVisToUVFITS=1

applyDIcalibration=1
doMWArxCorrections=1
doRFIflagging=0
useFastPrimaryBeamModels=1

CorrDumpTime=0.0
CorrDumpsPerCadence=0
NumberOfIntegrationBins=0
NumberOfIterations=

StartProcessingAt=0
StartIntegrationAt=0

// In correlator mode, Base File name is used to find correlator files.
ReadAllFromSingleFile=
BaseFilename=*_gpubox

doRawDataCorrections=1

ReadGpuboxDirect=1
UsePacketInput=0
UseThreadedVI=1

ArrayFile=array_file.txt
ArrayNumberOfStations=128

ChannelBandwidth=0.04
NumberOfChannels=32

ArrayPositionLat=-26.70331940
ArrayPositionLong=116.67081524

//time is needed to set lst, even if ObservationTimeBase is set.
ObservationTimeBase=2456519.20083
```

```
// --- observing stuff --- //

ObservationFrequencyBase=167.055

ObservationPointCentreHA=1.031
ObservationPointCentreDec=-25.93

ObservationImageCentreRA=
ObservationImageCentreDec=

// --- weighting stuff --- //

calBaselineMin=20.0
calShortBaselineTaper=50.0

//imgLongBaselineTaper = 3000.0
imgBaselineMin = 100.0

// --- calibration stuff --- //

DoCalibration=

SourceCatalogueFile=catalogue_imaging.txt

NumberOfCalibrators=1
NumberOfSourcesToPeel=0
NumberOfIonoCalibrators=0
UpdateCalibratorAmplitudes=0
```