

---

# RASCIL Documentation

*Release 1.0.0*

**Tim Cornwell, Peter Wortmann, Bojan Nikolic, Feng Wang, Vlad S**

**Mar 11, 2024**



CONTENTS

1 The role of the RASCIL in SKA Science Data Processing (SDP) 3

1.1 Installation . . . . . 3

1.2 Examples . . . . . 13

1.3 Structure . . . . . 14

1.4 API . . . . . 46

1.5 RASCIL development . . . . . 120

Python Module Index 129

Index 131



The Radio Astronomy Simulation, Calibration and Imaging Library expresses radio interferometry calibration and imaging algorithms in python and numpy. The interfaces all operate with familiar data structures such as image, visibility table, gain table, etc.

Source code: <https://gitlab.com/ska-telescope/external/rascil-main>

As of version 1.0.0, the library mostly contains high-level workflows and pipelines, while the data models and a large number of processing components (functions) have been migrated to [ska-sdp-datamodels](#) and [ska-sdp-func-python](#), which are directly used within RASCIL.

As of version 0.2.0, the data classes are built on the [Xarray](#) library, offering a rich API for applications. For more details including how to update existing scripts, see [Use of xarray](#).

To achieve sufficient performance we take a dual pronged approach - using threaded libraries for shared memory processing, and the [Dask](#) library for distributed processing.



## THE ROLE OF THE RASCIL IN SKA SCIENCE DATA PROCESSING (SDP)

RASCIL was developed in SDP under the name ARL (Algorithm Reference Library) with the emphasis of creating reference versions of standard algorithms. The ARL was therefore designed to present primarily imaging algorithms in a simple Python-based form so that the implemented functions could be seen and understood easily. This also fulfilled the requirement of providing a simple test version where algorithms could be tested and compared as necessary.

For an overview of the SDP see the [SDP CDR documentation](#)

More details can be found at: [SKA1 SDP Algorithm Reference Library \(ARL\) Report](#)

Subsequent to the conclusion of the SDP project, it became clear that ARL could play a larger role than being limited to a reference library. Hence, it was renamed to the Radio Astronomy Simulation, Calibration and Imaging Library (RASCIL) and is undergoing continued development. The Algorithm Reference Library (ARL) is now frozen. The background motivation and requirements of the ARL/RASCIL are detailed further in [Background](#).

### 1.1 Installation

RASCIL can be run on a Linux or macOS machine or cluster of machines, using python 3.8 or 3.9. At least 16GB physical memory is necessary to run the full test suite. In general more memory is better. RASCIL uses Dask for multi-processing and can make good use of multi-core and multi-node machines.

#### 1.1.1 Installation via pip

If you just wish to run the package and do not intend to run simulations or tests, RASCIL can be installed using pip:

```
pip3 install --index-url=https://artefact.skao.int/repository/pypi-all/simple rascil
```

This will download the latest stable version.

For simulations, you must add the data in a separate step:

```
mkdir rascil_data
cd rascil_data
curl https://ska-telescope.gitlab.io/external/rascil/rascil_data.tgz -o rascil_data.tgz
tar xzf rascil_data.tgz
cd data
export RASCIL_DATA=`pwd`
```

If you wish to run the RASCIL examples or tests, use one of the steps below.

### 1.1.2 Installation via docker

If you are familiar with docker, an easy approach is to use docker:

#### Dockerfiles for RASCIL

RASCIL supports the publishing of various docker images. The related Dockerfiles can be found in the `docker` directory and its subdirectories. The images are based on a python wheel created from RASCIL.

Makefiles are also included, which support building, pushing, and tagging images. The images are named as specified in the `release` file of the docker image directory, and tagged by the RASCIL version stored in `rascil/version.py`.

There are various directories for docker files:

- `rascil-base`: A minimal RASCIL, without data
- `rascil-full`: Base with data
- `rascil-notebook`: Supports running jupyter notebook
- `rascil-imaging-qa`: Runs the Continuum Imaging Quality Assessment tool
- `rascil-rcal`: Supports running RCAL as consumer of SDP visibility receive data

#### Automatic publishing

The docker images are automatically built by the CI pipeline.

When the repository is tagged, and a new version of it is released, a versioned docker images of each type is published to the [Central Artifact Repository](#) (CAR). To find out what versions you can download, look for the relevant RASCIL docker image in the CAR. Example:

```
artefact.skao.int/rascil-base:0.6.0
```

Upon every merge to the master branch, docker images with the `latest` tag are published to the GitLab Registry. Note that this tag is always updated with the latest code on master, hence it changes very often:

```
registry.gitlab.com/ska-telescope/external/rascil/rascil-imaging-qa:latest
```

#### Build, push, and tag a set of Dockerfiles

If you want to build an image yourself, follow these steps:

- `cd` into one of the subdirectories
- Build the image with `make build`

Other useful make commands :

- `push` pushes the images to the docker registry
- `push_latest` pushes the `:latest` tag
- `push_version` pushes a version tag without the git SHA



Note, the above make commands use environment variables to determine the image name and repository. For a full list and defaults, please consult the [Makefile](#) in `docker/make/`.

Useful make commands that can be run from the `docker` directory:

- `build_all_latest` builds, and tags as latest, all the images
- `rm_all` removes all the images
- `ls_all` lists all the images

## Test the images

The `docker/Makefile` contains commands for testing all the images. These write results into the host `/tmp` area. For `docker`:

- `make test_base`
- `make test_full`
- `make test_notebook`
- `make test_imaging_qa`
- `make test_rcal`

And for `singularity`:

- `make test_base_singularity`
- `make test_full_singularity`
- `make test_notebook_singularity`
- `make test_imaging_qa_singularity`
- `make test_rcal_singularity`

## Generic RASCIL images

### `rascil-base` and `rascil-full`

The base and full images are available at:

```
artefact.skao.int/rascil-base
artefact.skao.int/rascil-full
```

`rascil-base` does not have the RASCIL test data but is smaller in size. However, for many of the tests and demonstrations the test data is needed, which are included in `rascil-full`.

To run RASCIL with your home directory available inside the image:

```
docker run -it --volume $HOME:$HOME artefact.skao.int/rascil-full:<version>
```

Now let's run an example. First it simplifies using the container if we do not try to write inside the container, and that's why we mapped in our `$HOME` directory. So to run the `/rascil/examples/scripts/imaging.py` script, we first change directory to the name of the `HOME` directory, which is the same inside and outside the container, and then give the full address of the script inside the container. This time we will show the prompts from inside the container:

```
% docker run -p 8888:8888 -v $HOME:$HOME -it artefact.skao.int/rascil-full:0.6.
↪0
rascil@d0c5fc9fc19d:/rascil$ cd /<your home directory>
rascil@d0c5fc9fc19d:/<your home directory>$ python3 /rascil/examples/scripts/
↪imaging.py
...
rascil@d0c5fc9fc19d:/<your home directory>$ ls -l imaging*.fits
-rw-r--r-- 1 rascil rascil 2102400 Feb 11 14:04 imaging_dirty.fits
-rw-r--r-- 1 rascil rascil 2102400 Feb 11 14:04 imaging_psf.fits
-rw-r--r-- 1 rascil rascil 2102400 Feb 11 14:04 imaging_restored.fits
```

In this example, we change directory to an external location (my home directory in this case, use yours instead), and then we run the script using the absolute path name inside the container.

## RASCIL Notebooks

The docker image to use with RASCIL Jupyter Notebooks is:

```
artefact.skao.int/rascil-notebook
```

Run Jupyter Notebooks inside the container:

```
docker run -it -p 8888:8888 --volume $HOME:$HOME artefact.skao.int/rascil-
↪notebook:0.6.0
cd /<your home directory>
jupyter notebook --no-browser --ip 0.0.0.0 /rascil/examples/notebooks/
```

The Jupyter server will start and output possible URLs to use:

```
[I 14:08:39.041 NotebookApp] Serving notebooks from local directory: /rascil/
↪examples/notebooks
[I 14:08:39.041 NotebookApp] The Jupyter Notebook is running at:
[I 14:08:39.042 NotebookApp] http://d0c5fc9fc19d:8888/?
↪token=f050f82ed0f8224e559c2bdd29d4ed0d65a116346bcb5653
[I 14:08:39.042 NotebookApp] or http://127.0.0.1:8888/?
↪token=f050f82ed0f8224e559c2bdd29d4ed0d65a116346bcb5653
[I 14:08:39.042 NotebookApp] Use Control-C to stop this server and shut down
↪all kernels (twice to skip confirmation).
[W 14:08:39.045 NotebookApp] No web browser found: could not locate runnable
↪browser.
```

The 127.0.0.1 is the one we want. Enter this address in your local browser. You should see the standard Jupyter directory page.

## Images of RASCIL applications

### Continuum imaging Quality Assessment tool (a.k.a imaging\_qa)

`imaging_qa` finds compact sources in a continuum image and compares them to the sources used in the simulation, thus revealing the quality of the imaging.

## DOCKER

Pull the image:

```
docker pull artefact.skao.int/rascil-imaging-qa:<version>
```

Run the image:

```
docker run -v ${PWD}:/myData -e DOCKER_PATH=${PWD} \
  -e CLI_ARGS='--ingest_fitsname_restored /myData/my_restored.fits \
  --ingest_fitsname_residual /myData/my_residual.fits' \
  --rm artefact.skao.int/rascil-imaging-qa:0.6.0
```

Run it from the directory where your images you want to check are. The output files will appear in the same directory. Update the CLI\_ARGS string with the command line arguments of the `imaging_qa` code as needed. DOCKER\_PATH is used to extract the path of the output files the app produced in your local machine, not in the docker container. This is used for generating the output file index files.

## SINGULARITY

Pull the image:

```
singularity pull rascil-imaging-qa.img docker://artefact.skao.int/
rascil-imaging-qa:0.6.0
```

Run the image:

```
singularity run \
  --env CLI_ARGS='--ingest_fitsname_restored test-imaging-pipeline-dask_
↪continuum_imaging_restored.fits \
  --ingest_fitsname_residual test-imaging-pipeline-dask_continuum_
↪imaging_residual.fits' \
  rascil-imaging-qa.img
```

Run it from the directory where your images you want to check are. The output files will appear in the same directory. If the singularity image you downloaded is in a different path, point to that path in the above command. Update the CLI\_ARGS string with the command line arguments of the `imaging_qa` code as needed.

## Providing input arguments from a file

You may create a file that contains the input arguments for the app. Here is an example of it, called *args.txt*:

```
::  
  
-ingest_fitsname_restored=/myData/test-imaging-pipeline-dask_continuum_imaging_restored.fits  
-ingest_fitsname_residual=/myData/test-imaging-pipeline-dask_continuum_imaging_residual.fits  
-check_source=True -plot_source=True
```

Make sure each line contains one argument, there is an equal sign between arg and its value, and that there aren't any trailing white spaces in the lines (and no empty lines). The paths to images and other input files has to be the absolute path within the container. Here, we use the DOCKER example of mounting our data into the /myData directory.

Then, calling `docker run` simplifies as:

```
docker run -v ${PWD}:/myData -e DOCKER_PATH=${PWD} -e CLI_ARGS='@/myData/args.  
↪txt' \  
--rm artefact.skao.int/rascil-imaging-qa:0.6.0
```

Here, we assume that your custom *args.txt* file is also mounted together with the data into /myData. Provide the absolute path to that file when you run the above command.

You can use an *args* file to run the singularity version with same principles, bearing in mind that singularity will automatically mount your filesystem into the container with paths matching those on your system.

## RCAL visibility receive consumer

The [rascil\\_rcal](#) directory contains the necessary extra code and Dockerfile to build a docker image that can be used as a consumer for the [visibility receive workflow](#). This workflow can be deployed in the SDP system. It receives data packets from the Correlator and Beam Former (CBF) or its emulator.

A prototype *rcal-consumer* has been added to the docker image. It formats the received data packets into objects that can be passed into a *VisibilityBucket*. A *VisibilityBucket* is filled up until full, i.e. when it received all frequency channel data for a single time sample. The resulting *Visibility* object is then passed to **RCAL**, which processes the data and produces the resulting gain solutions (and optional png images).

The docker image is available from the Central Artifact Repository (tagged with the release version number):

```
artefact.skao.int/rascil-rcal:<version>
```

and from the GitLab container registry (tagged with latest and updated upon merge to master):

```
registry.gitlab.com/ska-telescope/external/rascil/rascil-rcal:latest
```

## Running RASCIL as a cluster

The following methods of running RASCIL as a cluster, will provide a set of docker-based environments, which host a Dask scheduler, various Dask workers (numbers can be customized), and a Jupyter lab notebook, which directly connects to the scheduler.

### Kubernetes

RASCIL can be run as a cluster in [Kubernetes](#) using [helm](#) and [kubectl](#) (you need to have these two installed). If you want to run it in a local developer environment (e.g. a laptop), we recommend using [Minikube](#).

A custom `values.yaml` file is provided in `/rascil/docker/kubernetes`. It is meant to be used with a custom Dask Helm chart maintained by SKA developers, hosted in a [GitLab repository](#). The documentation and details of the SKA Dask Helm chart can be found at <https://developer.skao.int/projects/ska-sdp-helmdeploy-charts/en/latest/charts/dask.html>.

You can modify the `values.yaml` file, if needed, e.g. you can change the number of worker replicas, or the docker image used (e.g. the version that should be run). If you don't use a `PersistentVolumeClaim`, remove `mounts` and `volume` sections from the jupyter and worker entries. (See also [/rascil/docker/kubernetes/README.md](#))

Start Minikube and add the helm repository:

```
helm repo add ska-helm https://gitlab.com/ska-telescope/sdp/ska-sdp-helmdeploy-
↪charts/-/raw/master/chart-repo
helm repo update
```

cd into the `/rascil/docker/kubernetes` directory and install the RASCIL cluster:

```
helm install test ska-helm/dask -f values.yaml
```

Instructions on how to connect to the Dask dashboard and the Jupyter lab notebook are printed in the screen, please follow those. You can follow the deployment process and access logs using `kubectl` or via ``k9s` <https://k9scli.io/>`_`.

To uninstall the chart and clean out all pods, run:

```
helm uninstall test
```

Note: this will remove changes you might have made in the Jupyter notebooks.

### Singularity

[Singularity](#) can be used to load and run the docker images:

```
singularity pull RASCIL-full.img docker://artefact.skao.int/rascil-full:0.6.0
singularity exec RASCIL-full.img python3 /rascil/examples/scripts/imaging.py
```

As in docker, don't run from the `/rascil/` directory.

Inside a SLURM file singularity can be used by prefacing dask and python commands with "singularity exec". For example:

```
ssh $host singularity exec /home/<your-name>/workspace/RASCIL-full.img dask-
↪scheduler --port=8786 &
ssh $host singularity exec /home/<your-name>/workspace/RASCIL-full.img dask-
↪worker --host ${host} --nprocs 4 --nthreads 1 \
--memory-limit 100GB $scheduler:8786 &
CMD="singularity exec /home/<your-name>/workspace/RASCIL-full.img python3 ./
↪cluster_test_ritoy.py ${scheduler}:8786 | tee ritoy.log"
eval $CMD
```

## Customisability

The docker images described here are ones we have found useful. However, if you have the RASCIL code tree installed then you can also make your own versions working from these Dockerfiles.

## Important updates

Starting with version 0.3.0, RASCIL is installed as a package into the docker images and the repository is not cloned anymore. Hence, every python script (except the ones in the `examples` directory) within the image has to be called with the `-m` switch in the following format, when running within the docker container, e.g.:

```
python -m rascil.apps.rascil_advise <args>
```

### 1.1.3 Installation via git clone

Use of git clone is necessary if you wish to develop and possibly contribute RASCIL code. Installation should be straightforward. We strongly recommend the use of a python virtual environment.

RASCIL requires python 3.8+.

The installation steps are:

- Use git to make a local clone of the Github repository:

```
git clone https://gitlab.com/ska-telescope/external/rascil.git
```

- Change into that directory:

```
cd rascil
```

- Install the required python packages and RASCIL package (in an activated virtual environment). The following command uses pip to install all of the requirements, including test and docs:

```
make install_requirements
```

- RASCIL makes use of a number of data files. These can be downloaded using Git LFS:

```
pip3 install git-lfs
git lfs install
git-lfs pull
```

If git-lfs is not already available, then lfs will not be recognised as a valid option for git in the second step. In this case, git-lfs can be installed via `sudo apt install git-lfs` or from a [tar file](#)

- Put the following definitions in your `.bashrc`:

```
export RASCIL=/path/to/rascil
export PYTHONPATH=$RASCIL:$PYTHONPATH
```

`python setup.py install` installs in the correct site-packages location so the definition of `PYTHONPATH` is not needed if you only don't intend to update or edit rascil in place. If you do intend to make changes, you will need the definition of `PYTHONPATH`.

## 1.1.4 Installation on specific machines

### Installation of RASCIL on macos

RASCIL is well-suited to running under macos. Installation should be straightforward. Although the pip approach can be used, we recommend use of Anaconda <https://www.anaconda.com> It is necessary to install python-casacore in a separate step. The steps required are:

```
conda create -n rascil_env python=3.9
conda activate rascil_env
conda install -c conda-forge python-casacore
git clone https://gitlab.com/ska-telescope/external/rascil.git
cd rascil
make install_requirements
```

We have specified python 3.9 since this currently is the preferred and supported version.

Finally, put the following definitions in your `.bashrc`:

```
export RASCIL=/path/to/rascil
export PYTHONPATH=$RASCIL:$PYTHONPATH
```

### Installation of RASCIL on CSD3

RASCIL is well-suited to running on CSD3. Installation should be straightforward. We strongly recommend the use of a python virtual environment. Follow the generic installation steps.

We recommend that RASCIL be installed on one of the preferred storage systems e.g. `/rds/user/hpccorn1/hpc-work/rascil`

Installation of Git LFS:

```
pip install git-lfs
git lfs install
git-lfs pull
```

## Installation of RASCIL on galahad

RASCIL is well-suited to running on galahad. Installation should be straightforward. We strongly recommend the use of a python virtual environment. Be sure to load the bare python base module and the gcc modules (in the supported versions) before installing RASCIL, e.g.

```
module load python39base gcc920
```

Follow the generic installation steps.

We recommend that RASCIL be installed on one of the preferred storage systems e.g. `/share/nas/<your-login>/rascil/`

If you are using singularity containers, you will probably need to put the singularity cache somewhere other than your home directory:

```
mkdir /share/nas/<yourname>/.singularity
export SINGULARITY_CACHEDIR=/share/nas/<yourname>/.singularity
singularity pull RASCIL-full.img docker://timcornwell/rascil-full
```

## Installation of RASCIL on P3

RASCIL is well-suited to running on P3. Installation should be straightforward. We strongly recommend the use of a python virtual environment. Follow the generic installation steps.

### 1.1.5 Trouble-shooting

#### Testing

Check your installation by running a subset of the tests:

```
pip3 install pytest pytest-xdist
py.test -n 4 tests/processing_components
```

Or the full set:

```
py.test -n 4 tests
```

- Ensure that pip is up-to-date. If not, some strange install errors may occur.
- Check that the contents of the data directories have plausible contents. If `gif-lfs` has not been run successfully then the data files will just contain meta data, leading to strange run-time errors.
- There may be some dependencies that require either conda (or brew install on a mac).
- Ensure that you have made the directory `test_results` to store the test results. Or add the below in the test script:

```
if not os.path.isdir(test_results):os.makedirs(test_results)
```



## Casacore installation

RASCIL requires python-casacore to be installed. This is included in the requirements for the RASCIL install and so should be installed automatically via pip. In some cases there may not be a compatible binary install (wheel) available via pip. If not, pip will download the source code of casacore and attempt a build from source. The most common failure mode during the source build is that it cannot find the boost-python libraries. These can be installed via pip. If errors like this occur, once rectified, re-installing python-casacore separately via pip may be required, prior to re-commencing the RASCIL install. Trouble-shooting problems with a source install can be difficult. If available, this can be avoided by using anaconda as the base for an environment. It supports python-casacore which can be installed with “conda install python-casacore”. It may also be possible to avoid some of the more difficult issues with building python-casacore by downloading CASA prior to the RASCIL install.

## RASCIL data in notebooks

In some case the notebooks may not automatically find the RASCIL data directory, in which case explicitly setting the RASCIL\_DATA environment variable may be required: `%env RASCIL_DATA=~/.rascil_data/data`.

# 1.2 Examples

## 1.2.1 Running notebooks

The best way to get familiar with RASCIL is via jupyter notebooks. For example:

```
jupyter notebook examples/notebooks/imaging.ipynb
```

See the jupyter notebooks below:

Some functions initially developed for the LOFAR telescope pipeline are made available in RASCIL. The following notebooks show how the functions are integrated.

In addition, there are other notebooks in examples/notebooks that are not built as part of this documentation. In some cases it may be necessary to add the following to the notebook to locate the RASCIL data `%env RASCIL_DATA=~/.rascil_data/data`

## 1.2.2 Running scripts

Some example scripts are found in the directory examples/scripts.

## 1.2.3 SKA simulations

- `genindex`
- `modindex`

## 1.3 Structure

Those familiar with other calibration and imaging packages will find the following information useful in navigating the RASCIL. Not all functions are listed here but are contained in the API.

The long form of the name is given for all entries but all function names are unique so a given function can be accessed using the very top level import:

```
import rascil.processing_components
import rascil.workflows
import rascil.apps
```

### 1.3.1 Data containers used by RASCIL

RASCIL holds data in python Classes. The bulk data and attributes are usually kept in a `xarray.Dataset`. For each `xarray` based class there is an accessor which holds class specific methods and properties.

Note that the data models have been migrated into the [SKA SDP Python-based Data Models](#) directory. Please refer to the documentation there for more information.

### 1.3.2 Functions

NOTE: Some processing functions have been migrated to the [ska-sdp-func-python](#) repository, please refer to the documentation there for information. Functions on this page is an incomplete list.

#### Read existing Measurement Set

Casacore must be installed for MS reading and writing:

- List contents of a MeasurementSet: `rascil.processing_components.visibility.base.list_ms()`
- Creates a list of Visibilities, one per FIELD\_ID and DATA\_DESC\_ID: `rascil.processing_components.visibility.base.create_visibility_from_ms()`

#### Image

- Image operations: `rascil.processing_components.image.operations()`
- Import from FITS: `rascil.processing_components.image.operations.import_image_from_fits()`
- Re-project coordinate system: `rascil.processing_components.image.operations.reproject_image()`
- Smooth image: `rascil.processing_components.image.operations.smooth_image()`
- FFT: `rascil.processing_components.image.operations.fft_image_to_griddata_with_wcs()`
- Remove continuum: `rascil.processing_components.image.operations.remove_continuum_image()`

### 1.3.3 Workflows

Workflows coordinate processing using the data models, processing components, and processing library. These are high level functions, and are available in an `rsexecute` (i.e. `dask`) version and sometimes a scalar version.

#### Calibration workflows

- Calibrate workflow: `rascil.workflows.rsexecute.calibration.calibrate_list_rsexecute_workflow()`

#### Imaging workflows

- Invert: `rascil.workflows.rsexecute.imaging.invert_list_rsexecute_workflow()`
- Predict: `rascil.workflows.rsexecute.imaging.predict_list_rsexecute_workflow()`
- Deconvolve: `rascil.workflows.rsexecute.imaging.deconvolve_list_rsexecute_workflow()`

#### Pipeline workflows

- ICAL: `rascil.workflows.rsexecute.pipelines.ical_skymodel_list_rsexecute_workflow()`
- Continuum imaging: `rascil.workflows.rsexecute.pipelines.continuum_imaging_skymodel_list_rsexecute_workflow()`
- Spectral line imaging: `rascil.workflows.rsexecute.pipelines.spectral_line_imaging_skymodel_list_rsexecute_workflow()`
- MPCCAL: `rascil.workflows.rsexecute.pipelines.mpccal_skymodel_list_rsexecute_workflow()`

#### Simulation workflows

- Testing and simulation support: `rascil.workflows.rsexecute.simulation.simulate_list_rsexecute_workflow()`

#### Execution

- Execution framework (an interface to Dask): `rascil.workflows.rsexecute.execution_support()`

### 1.3.4 Apps

Apps are command line applications written using the data models, processing components, and processing library.

#### Imaging

##### `rascil_imager`

`rascil_imager` is a command line app written using RASCIL. It supports three ways of making an image:

- invert: Inverse Fourier Transform of the visibilities to make a dirty image (or point spread function)
- cip: The SKA Continuum Imaging Pipeline.
- ical: The SKA Iterative Calibration Pipeline (ICAL)

Notable features:

- Reads a CASA MeasurementSet and writes FITS files
- Image size can be a composite of 2, 3, 5
- Distribute processing across processors using Dask
- Multi Frequency Synthesis Multiscale CLEAN available, also with distribution of CLEAN over facets
- Distribution of restoration over facets
- Wide field imaging using the fast and accurate nifty gridder
- Modelling of bright sources by fitting with sub-pixel locations
- Selfcalibration available for atmosphere (T), complex gains (G), and bandpass (B)
- Selection of data by uv range and r range (where r is the distance of station/dish from array centre)

CLI arguments are grouped:

- `--mode` prefixed parameters controls which algorithm is run.
- `--imaging` prefixed parameters control the details of the imaging such as number of pixels, cellsize
- `--clean` prefixed parameters control the clean deconvolutions (active only for modes `cip` and `ical`)
- `--calibration` prefixed parameters control the calibration in the ICAL pipeline. (active only for mode `ical`)
- `--dask` prefixed parameters control the use of Dask/rsexecute for distributing the processing

### MeasurementSet ingest

Although a CASA MeasurementSet can hold heterogeneous observations, identified by data descriptors. `rascil-imager` can only process identical data descriptors from a MS. The number of channels and polarisation must be the same.

Each selected data descriptor is optionally split into a number of channels optionally averaged and placed into one Visibility.

For example, using the arguments:

```
--ingest_msname SNR_G55_10s.calib.ms --ingest_dd 0 1 2 3 --ingest_vis_nchan 64 \  
--ingest_chan_per_vis 8 --ingest_average_vis True
```

will read data descriptors 0, 1, 2, 3, each of which has 64 channels. Each set of 64 channels are split into blocks of 8 and averaged. We thus end up with 32 separate datasets in RASCIL, each of which is a Visibility and has 1 channel, for a total of 32 channels. If the argument `--ingest_average_vis` is set to `False`, each Visibility has eight channels, for a total of 256 channels.

### Selection

`rascil_imager` supports selection of data by uv range `--imaging_uvmin` `--imaging_uvmax`, and by dish/station based on distance from the array centre `--imaging_rmin` `--imaging_rmax`

## Imaging

To make an image from visibilities or to predict visibilities from a model, it is necessary to use a gridded. Nifty gridded ([https://gitlab.mpcdf.mpg.de/ift/nifty\\_gridded](https://gitlab.mpcdf.mpg.de/ift/nifty_gridded)) is currently the best gridded to use in RASCIL. It is written in c and uses OpenMP to distribute the processing across multiple threads. The Nifty Gridded uses an improved wstacking algorithm uses many fewer w-planes than w stacking or w projection. It is not necessary to explicitly set the number of w-planes.

The gridded is set by the `--imaging_context` argument. The default, `--imaging_context ng` is the Nifty Gridded.

## CLEAN

rascil-imager supports Hogbom CLEAN, MultiScale CLEAN, and Multi-Frequency Synthesis MultiScale Clean (also known as MM CLEAN). The first two work independently on different frequency channels, while MM CLEAN works jointly cross all channels using a Taylor Series expansion in frequency for the emission.

The clean methods support a number of processing speed enhancements:

- The multi-frequency-synthesis CLEAN works by fitting a Taylor series in frequency. The `--ingest_chan_per_vis` argument controls the aggregation of channels in the MeasurementSet to form image planes for the CLEAN. Within a Visibility the different channels are gridded together to form one image. Each image is then used in the mmclean algorithm. For example, a data set may have 256 channels spread over 4 data descriptors. We can split these into 32 BlockVisibilities and then run the mmclean over these 32 channels.
- Only a limited central region of the PSF will be subtracted during the minor cycles.
- The cleaning may be partitioned into overlapping facets, each of which is cleaned independently, and then merged with neighbours using a taper function. This works well for fields of compact sources but is likely to not perform well for extended emission.
- The restoration may be distributed via subimages. This requires that the subimages have significant overlap such that the clean beam can fit within the overlap area.

Bright compact sources can optionally be represented by discrete components instead of pixels.

- `--clean_component_threshold 0.5` All sources > 0.5 Jy to be fitted
- `--clean_component_method fit` non-linear last squares algorithm to find source parameters

The skymodel written at the end of processing will include both the image model and the skycomponents.

## Polarisation

The polarisation processing behaviour is controlled by `--image_pol`.

- `--image_pol stokesI` will image only the I Stokes parameter
- `--image_pol stokesIQUV` will image all Stokes parameters I, Q, U, V

Note that the combination of MM CLEAN and stokesIQUV imaging is not likely to be meaningful.

## Self-calibration

rascil-imager supports self-calibration as part of the imaging. At the end of each major cycle a calibration solution and application may optionally be performed.

Calibration uses the Hamaker Bregman Sault formalism with the following Jones matrices supported: T (Atmospheric phase), G (Electronics gain), B - (Bandpass).

An example consider the arguments:

```
calibration_T_first_selfcal = 2
calibration_T_phase_only = True
calibration_T_timeslice = None
calibration_G_first_selfcal = 5
calibration_G_phase_only = False
calibration_G_timeslice = 1200.0
calibration_B_first_selfcal = 8
calibration_B_phase_only = False
calibration_B_timeslice = 1.0e5
calibration_global_solution = True
calibration_calibration_context = "TGB"
```

These will perform a phase only solution of the T term after the second major cycle for every integration, solution of G after 5 major cycles with timescale of 1200s, and solution of B after 8 major cycles, integrating across all frequencies where appropriate. Note, that T and G terms are averages across frequency.

## SkyModel in ICAL

When running `rascil_imager` in mode `ical`, optionally, an initial SkyModel can be used. To do this, set `--use_initial_skymodel` to `True`. The SkyModel is made up of model images (created based on input Block-Visibilities), and SkyComponents. The kind of SkyComponent(s) to use in the initial SkyModel is controlled by the `--input_skycomponent_file` and `--num_bright_sources` arguments:

1. If no input file is provided, a point source at the phase centre, with brightness of 1 Jy is used as the component.
2. **If either an HDF file or a TXT file is provided, the components are read from the file.**
  - a. if `--num_bright_sources` is left as `None`, all of the components are used for the SkyModel
  - b. if `--num_bright_sources` is an integer  $n$  ( $n > 0$ ), then  $n$  number of the brightest components are used for the SkyModel

This SkyModel is then overwritten during the remaining cycles of the run.

By default, `--use_initial_skymodel` is set to `False`, and hence no initial SkyModel is used.

In addition, you can decide whether to reset the initial skymodel after first calibration, or not, by setting the `--calibration_reset_skymodel` either to `True` or `False`.

## Dask

Dask is used to distribute processing across multiple cores or nodes. The setup and execution of a set of workers is controlled by a scheduler. By default, rascil uses the process scheduler which sets up a number of processes each with a number of threads. If the host has 16 cores, the set up will be 4 processes each with 4 threads for a total of 16 Dask workers.

For distribution across a cluster, the Dask distributed processor is required. See *RASCIL and DASK* for more details.

## Example script

The following runs the cip on a data set from the CASA examples:

```
#!/bin/bash
# Run this in the directory containing SNR_G55_10s.calib.ms
python $RASCIL/rascil/apps/rascil_imager.py --mode cip \
--ingest_mname SNR_G55_10s.calib.ms --ingest_dd 0 1 2 3 --ingest_vis_nchan 64 \
--ingest_chan_per_vis 8 --ingest_average_vis True \
--imaging_npixel 1280 --imaging_cellsize 3.878509448876288e-05 \
--imaging_weighting robust --imaging_robustness -0.5 \
--clean_nmajor 5 --clean_algorithm mmclean --clean_scales 0 6 10 30 60 \
--clean_fractional_threshold 0.3 --clean_threshold 0.12e-3 --clean_nmoment 5 \
--clean_psf_support 640 --clean_restored_output integrated
```

## Command line arguments

RASCIL continuum imager

```
usage: rascil_imager.py [-h] [--mode MODE] [--logfile LOGFILE]
                        [--performance_file PERFORMANCE_FILE]
                        [--ingest_mname INGEST_MNAME]
                        [--ingest_dd [INGEST_DD ...]]
                        [--ingest_vis_nchan INGEST_VIS_NCHAN]
                        [--ingest_chan_per_vis INGEST_CHAN_PER_VIS]
                        [--ingest_average_vis INGEST_AVERAGE_VIS]
                        [--imaging_phasecentre IMAGING_PHASECENTRE]
                        [--imaging_pol IMAGING_POL]
                        [--imaging_nchan IMAGING_NCHAN]
                        [--imaging_context IMAGING_CONTEXT]
                        [--imaging_ng_threads IMAGING_NG_THREADS]
                        [--imaging_w_stacking IMAGING_W_STACKING]
                        [--imaging_flat_sky IMAGING_FLAT_SKY]
                        [--imaging_npixel IMAGING_NPIXEL]
                        [--imaging_cellsize IMAGING_CELLSIZE]
                        [--imaging_weighting IMAGING_WEIGHTING]
                        [--imaging_robustness IMAGING_ROBUSTNESS]
                        [--imaging_gaussian_taper IMAGING_GAUSSIAN_TAPER]
                        [--imaging_dopsf IMAGING_DOPSF]
                        [--imaging_dft_kernel IMAGING_DFT_KERNEL]
                        [--imaging_uvmax IMAGING_UVMAX]
```

(continues on next page)

(continued from previous page)

```
[--imaging_uvmin IMAGING_UVMIN]
[--imaging_rmax IMAGING_RMAX]
[--imaging_rmin IMAGING_RMIN]
[--perform_flagging PERFORM_FLAGGING]
[--flagging_strategy_name FLAGGING_STRATEGY_NAME]
[--calibration_reset_skymodel CALIBRATION_RESET_SKYMODEL]
[--calibration_T_first_selfcal CALIBRATION_T_FIRST_SELFCAL]
[--calibration_T_phase_only CALIBRATION_T_PHASE_ONLY]
[--calibration_T_timeslice CALIBRATION_T_TIMESLICE]
[--calibration_G_first_selfcal CALIBRATION_G_FIRST_SELFCAL]
[--calibration_G_phase_only CALIBRATION_G_PHASE_ONLY]
[--calibration_G_timeslice CALIBRATION_G_TIMESLICE]
[--calibration_B_first_selfcal CALIBRATION_B_FIRST_SELFCAL]
[--calibration_B_phase_only CALIBRATION_B_PHASE_ONLY]
[--calibration_B_timeslice CALIBRATION_B_TIMESLICE]
[--calibration_global_solution CALIBRATION_GLOBAL_SOLUTION]
[--calibration_context CALIBRATION_CONTEXT]
[--use_initial_skymodel USE_INITIAL_SKYMODEL]
[--input_skycomponent_file INPUT_SKYCOMPONENT_FILE]
[--num_bright_sources NUM_BRIGHT_SOURCES]
[--clean_algorithm CLEAN_ALGORITHM]
[--clean_beam CLEAN_BEAM CLEAN_BEAM CLEAN_BEAM]
[--clean_scales [CLEAN_SCALES ...]]
[--clean_nmoment CLEAN_NMOMENT]
[--clean_nmajor CLEAN_NMAJOR]
[--clean_niter CLEAN_NITER]
[--clean_psf_support CLEAN_PSF_SUPPORT]
[--clean_gain CLEAN_GAIN]
[--clean_threshold CLEAN_THRESHOLD]
[--clean_component_threshold CLEAN_COMPONENT_THRESHOLD]
[--clean_component_method CLEAN_COMPONENT_METHOD]
[--clean_fractional_threshold CLEAN_FRACTIONAL_THRESHOLD]
[--clean_facets CLEAN_FACETS]
[--clean_overlap CLEAN_OVERLAP]
[--clean_taper CLEAN_TAPER]
[--clean_restore_facets CLEAN_RESTORE_FACETS]
[--clean_restore_overlap CLEAN_RESTORE_OVERLAP]
[--clean_restore_taper CLEAN_RESTORE_TAPER]
[--clean_restored_output CLEAN_RESTORED_OUTPUT]
[--use_dask USE_DASK] [--dask_nthreads DASK_NTHREADS]
[--dask_memory DASK_MEMORY]
[--dask_memory_usage_file DASK_MEMORY_USAGE_FILE]
[--dask-nodes [DASK_NODES ...]]
[--dask_nworkers DASK_NWORKERS]
[--dask_scheduler DASK_SCHEDULER]
[--dask_scheduler_file DASK_SCHEDULER_FILE]
[--dask_tcp_timeout DASK_TCP_TIMEOUT]
[--dask_connect_timeout DASK_CONNECT_TIMEOUT]
[--dask_malloc_trim_threshold DASK_MALLOC_TRIM_THRESHOLD]
```



## Named Arguments

<b>--mode</b>	Processing cip   ical   invert   load Default: “cip”
<b>--logfile</b>	Name of logfile (default is to construct one from msname)
<b>--performance_file</b>	Name of json file to contain performance information
<b>--ingest_msname</b>	MeasurementSet to be read
<b>--ingest_dd</b>	Data descriptors in MS to read (all must have the same number of channels) Default: [0]
<b>--ingest_vis_nchan</b>	Number of channels in a single data descriptor in the MS
<b>--ingest_chan_per_vis</b>	Number of channels per vis (before any average) Default: 1
<b>--ingest_average_vis</b>	Average all channels in vis? Default: “False”
<b>--imaging_phasecentre</b>	Phase centre (in SkyCoord string format)
<b>--imaging_pol</b>	RASCIL polarisation frame for image Default: “stokesI”
<b>--imaging_nchan</b>	Number of channels per image Default: 1
<b>--imaging_context</b>	Imaging context i.e. the gridder used 2d   ng Default: “ng”
<b>--imaging_ng_threads</b>	Number of Nifty Gridder threads to use (4 is a good choice) Default: 4
<b>--imaging_w_stacking</b>	Use the improved w stacking method in Nifty Gridder? Default: True
<b>--imaging_flat_sky</b>	If using a primary beam, normalise to flat sky? Default: False
<b>--imaging_npixel</b>	Number of pixels in ra, dec: Should be a composite of 2, 3, 5
<b>--imaging_cellsize</b>	Cellsize (radians). Default is to calculate.
<b>--imaging_weighting</b>	Type of weighting uniform or robust or natural) Default: “uniform”
<b>--imaging_robustness</b>	Robustness for robust weighting Default: 0.0
<b>--imaging_gaussian_taper</b>	Size of Gaussian smoothing, implemented as taper in weights (rad)
<b>--imaging_dopsf</b>	Make the PSF instead of the dirty image? Default: “False”
<b>--imaging_dft_kernel</b>	DFT kernel: cpu_looped   gpu_raw

- imaging\_uvmax** Maximum uv (wavelengths)
- imaging\_uvmin** Minimum uv (wavelengths)
- imaging\_rmax** Maximum distance of dish/station from array center (wavelengths)
- imaging\_rmin** Minimum distance of dish/station from array center (wavelengths)
- perform\_flagging** If enabled, runs AOFlagger flagging strategy  
Default: “False”
- flagging\_strategy\_name** Contains the name of the flagging strategy to use when perform\_flagging is True. There are strategies available for different telescopes: AARTFAAC, ARECIBO, ARECIBO 305M, BIGHORNS, EVLA, JVLA, LOFAR, MWA, PARKES, PKS, ATPKSMB, WSRT. If the desired telescope is not listed here, you can use one of the strategies defined in the AOFlagger repository (<https://gitlab.com/aroffringa/aoflagger/-/tree/master/data/strategies>) or define a new strategy interactively using the AOFlagger rfigui ([https://aoflagger.readthedocs.io/en/latest/using\\_rfigui.html](https://aoflagger.readthedocs.io/en/latest/using_rfigui.html))  
Default: “generic”
- calibration\_reset\_skymodel** Reset the initial skymodel after initial calibration?  
Default: “True”
- calibration\_T\_first\_selfcal** First selfcal for T (complex gain). T is common to both receptors  
Default: 1
- calibration\_T\_phase\_only** Phase only solution  
Default: “True”
- calibration\_T\_timeslice** Solution length (s) 0 means minimum
- calibration\_G\_first\_selfcal** First selfcal for G (complex gain). G is different for the two receptors  
Default: 3
- calibration\_G\_phase\_only** Phase only solution?  
Default: “False”
- calibration\_G\_timeslice** Solution length (s) 0 means minimum
- calibration\_B\_first\_selfcal** First selfcal for B (bandpass complex gain). B is complex gain per frequency.  
Default: 4
- calibration\_B\_phase\_only** Phase only solution  
Default: “False”
- calibration\_B\_timeslice** Solution length (s)
- calibration\_global\_solution** Solve across frequency  
Default: “True”
- calibration\_context** Terms to solve (in order e.g. TGB)  
Default: “T”
- use\_initial\_skymodel** Whether to use an initial SkyModel in ICAL or not  
Default: False

- input\_skycomponent\_file** Input name of skycomponents file (in hdf or txt format) for initial Sky-Model in ICAL
- num\_bright\_sources** Number of brightest sources to select for initial SkyModel (if None, use all sources from input file)
- clean\_algorithm** Type of deconvolution algorithm (hogbom or ms-clean or mmclean)  
Default: “mmclean”
- clean\_beam** Clean beam: major axis, minor axis, position angle (deg)
- clean\_scales** Scales for multiscale clean (pixels) e.g. [0, 6, 10]  
Default: [0]
- clean\_nmoment** Number of frequency moments in mmclean (1 is a constant, 2 is linear, etc.)  
Default: 4
- clean\_nmajor** Number of major cycles in cip or ical  
Default: 5
- clean\_niter** Number of minor cycles in CLEAN (i.e. clean iterations)  
Default: 1000
- clean\_psf\_support** Half-width of psf used in cleaning (pixels)  
Default: 256
- clean\_gain** Clean loop gain  
Default: 0.1
- clean\_threshold** Clean stopping threshold (Jy/beam)  
Default: 0.0001
- clean\_component\_threshold** Sources with absolute flux > this level (Jy) are fit or extracted using skycomponents
- clean\_component\_method** Method to convert sources in image to skycomponents: ‘fit’ in frequency or ‘extract’ actual values  
Default: “fit”
- clean\_fractional\_threshold** Fractional stopping threshold for major cycle  
Default: 0.3
- clean\_facets** Number of overlapping facets in faceted clean (along each axis)  
Default: 1
- clean\_overlap** Overlap of facets in clean (pixels)  
Default: 32
- clean\_taper** Type of interpolation between facets in deconvolution (none or linear or tukey)  
Default: “tukey”
- clean\_restore\_facets** Number of overlapping facets in restore step (along each axis)  
Default: 1

- clean\_restore\_overlap** Overlap of facets in restore step (pixels)  
Default: 32
- clean\_restore\_taper** Type of interpolation between facets in restore step (none or linear or tukey)  
Default: “tukey”
- clean\_restored\_output** Type of restored image output: taylor, list, or integrated  
Default: “list”
- use\_dask** Use Dask processing? False means that graphs are executed as they are constructed.  
Default: “True”
- dask\_nthreads** Number of threads in each Dask worker (None means Dask will choose)
- dask\_memory** Memory per Dask worker (GB), e.g. 5GB (None means Dask will choose)
- dask\_memory\_usage\_file** File in which to track Dask memory use (using dask-memusage)
- dask-nodes** Node names for SSHCluster
- dask\_nworkers** Number of workers (None means Dask will choose)
- dask\_scheduler** Externally defined Dask scheduler e.g. 127.0.0.1:8786 or ssh for SSHCluster or existing for current scheduler
- dask\_scheduler\_file** Externally defined Dask scheduler file to setup dask cluster
- dask\_tcp\_timeout** Dask TCP timeout
- dask\_connect\_timeout** Dask connect timeout
- dask\_malloc\_trim\_threshold** Threshold for trimming memory on release (0 is aggressive)  
Default: 0

## rascil\_sensitivity

rascil\_sensitivity is a command line app written using RASCIL. It allows calculation of point source sensitivity (pss) and surface brightness sensitivity (sbs). The analysis is based on Dan Briggs’s PhD thesis <https://casa.nrao.edu/Documents/Briggs-PhD.pdf>

rascil\_sensitivity works by constructing a Visibility set and running invert to obtain the point spread function. The visibility weights in the Visibility are constructed to be equal to the time-bandwidth product each visibility sample. For natural weighting, these weights are used as the imaging weights. The sum of gridded weights therefore gives the total time-bandwidth of the observation. Given Tsys and efficiency it can then calculate the point source sensitivity. To obtain the surface brightness sensitivity, we calculate the solid angle of the clean beam fitted to the PSF, and divide the point source sensitivity by the solid angle.

Weighting schemes such as robust weighting and visibility tapering modify the imaging weights. The point source sensitivity always worsens compared to natural weighting but the surface brightness sensitivity may improve.

The robustness parameter and the visibility taper can be specified as single values or as a list of values to test.

The array configuration is specified by 2 parameters: *configuration* identifies a table with details of the available dishes, *subarray* names a json file listing the ids (i.e. row numbers in the configuration table) of the dishes to be used. If no subarray is specified then all dishes will be selected. The json format is:

```
{"ids": [64, 65, 66, 67, 68, 69, 70, ....etc.]}
```

The principal output is a CSV file, written by pandas in which all values of robustness and taper are tested, along with natural weighting.

The processing is distributed using Dask over all frequency channels specified.

### Example script

The following:

```
python $RASCIL/rascil/apps/rascil_sensitivity.py --results range_0.5_int_20 --time_range_
↪-0.25 0.25 \
    --integration_time 20 --msfile range_0.5_int_20.ms
```

produces the output:: Final results:

```
weighting robustness taper cleanbeam_bmaj cleanbeam_bmin cleanbeam_bpa ... pss_casa reltonat_casa
sa sbs tb sbs_casa
```

```
0 uniform 0.0 0.0 0.000124 0.000106 0.348636 ... 5.055773e-08 4.214877 5.290084e-12 4.844478e+06
7.435200e+13 9557.074591 1 robust -2.0 0.0 0.000125 0.000107 0.346705 ... 4.907281e-08 4.091084 5.423607e-
12 4.528158e+06 8.096404e+13 9048.003290 2 robust -1.5 0.0 0.000138 0.000119 0.366295 ... 4.237805e-08
3.532957 6.570541e-12 2.905383e+06 1.339994e+14 6449.703859 3 robust -1.0 0.0 0.000220 0.000209 19.006936 ...
3.168845e-08 2.641790 1.669975e-11 6.384441e+05 4.295821e+14 1897.540277 4 robust -0.5 0.0 0.000328 0.000316
40.826795 ... 2.208990e-08 1.841582 3.703912e-11 1.701715e+05 1.229183e+15 596.393758 5 robust 0.0 0.0
0.000454 0.000437 33.235117 ... 1.618849e-08 1.349596 7.111900e-11 5.956637e+04 2.721061e+15 227.625391 6
robust 0.5 0.0 0.000600 0.000577 30.284717 ... 1.360183e-08 1.133952 1.242658e-10 2.643972e+04 4.523710e+15
109.457521 7 robust 1.0 0.0 0.000729 0.000702 -149.373492 ... 1.228866e-08 1.024476 1.836020e-10 1.549264e+04
6.035397e+15 66.930950 8 robust 1.5 0.0 0.000791 0.000761 30.715780 ... 1.200501e-08 1.000829 2.160325e-
10 1.241103e+04 6.792939e+15 55.570408 9 robust 2.0 0.0 0.000802 0.000772 -149.271796 ... 1.199519e-08
1.000010 2.221125e-10 1.194877e+04 6.932970e+15 54.005008 10 natural 0.0 0.0 0.000804 0.000773 -149.270574
... 1.199506e-08 1.000000 2.228600e-10 1.189396e+04 6.950160e+15 53.823312
```

[11 rows x 24 columns]

### Command line arguments

Calculate relative sensitivity for MID observations

```
usage: rascil_sensitivity.py [-h] [--use_dask USE_DASK]
                             [--imaging_npixel IMAGING_NPIXEL]
                             [--msfile MSFILE]
                             [--imaging_cellsize IMAGING_CELLSIZE]
                             [--imaging_oversampling IMAGING_OVERSAMPLING]
                             [--imaging_weighting IMAGING_WEIGHTING]
                             [--imaging_robustness [IMAGING_ROBUSTNESS ...]]
                             [--imaging_taper [IMAGING_TAPER ...]] [--ra RA]
                             [--tsys TSYS] [--efficiency EFFICIENCY]
                             [--diameter DIAMETER] [--declination DECLINATION]
                             [--configuration CONFIGURATION]
                             [--subarray SUBARRAY] [--rmax RMAX]
                             [--frequency FREQUENCY]
                             [--integration_time INTEGRATION_TIME]
```

(continues on next page)

(continued from previous page)

```
[--time_range TIME_RANGE TIME_RANGE]
[--nchan NCHAN] [--channel_width CHANNEL_WIDTH]
[--verbose VERBOSE] [--results RESULTS]
```

## Named Arguments

<b>--use_dask</b>	Use dask processing? Default: "True"
<b>--imaging_npixel</b>	Number of pixels in ra, dec: Should be a composite of 2, 3, 5 Default: 1024
<b>--msfile</b>	Export Measurement file. Default: ""
<b>--imaging_cellsize</b>	Cellsize (radians). Default is to calculate.
<b>--imaging_oversampling</b>	Oversampling of synthesised_beam (Default 3.0) Default: 3.0
<b>--imaging_weighting</b>	Type of weighting: uniform or robust or natural
<b>--imaging_robustness</b>	Robustness for robust weighting, Default: [-2.0, -1.5, -1.0, -0.5, 0.0, 0.5, 1.0, 1.5, 2.0]
<b>--imaging_taper</b>	If set, use value for Gaussian taper, specified as radians in image plane
<b>--ra</b>	Right ascension (degrees) Default: 15.0
<b>--tsys</b>	System temperature (K) Default: 20.0
<b>--efficiency</b>	Correlator efficiency Default: 1.0
<b>--diameter</b>	MID antenna diameter (m) Default: 15.0
<b>--declination</b>	Declination (degrees) Default: -45.0
<b>--configuration</b>	Name of configuration or path: MID(=MIDR5), MIDR5, MEERKAT+ Default: "MIDR5"
<b>--subarray</b>	Name of json file describing subarray to be used, default is all antennas Default: ""
<b>--rmax</b>	Maximum distance of station from centre (m) Default: 200000.0

<b>--frequency</b>	Centre frequency (Hz) Default: 1360000000.0
<b>--integration_time</b>	Integration time (s) Default: 600
<b>--time_range</b>	Hour angle range in hours Default: [-4.0, 4.0]
<b>--nchan</b>	Number of channels Default: 1
<b>--channel_width</b>	Channel bandwidth (Hz) Default: 100000000.0
<b>--verbose</b>	Verbose output? Default: "False"
<b>--results</b>	Root name for output files Default: "rascil_sensitivity"

## rascil\_rcal

rascil\_rcal is a command line app written using RASCIL. It simulates the real-time calibration pipeline RCAL. In the SKA, an initial calibration is performed in real-time as the visibility data are accumulated. An accurate sky model is assumed to be available or a point source model is used.

In rascil\_rcal a MeasurementSet is read in and then iterated through in time-order solving for the gains. The gaintables are accumulated into a single gain table that is written as an HDF file.

There is also an additional plotting function that plots the gaintable values (gain amplitude, phase and residual) over time. If plotting is required, please make sure you have the correct path `-plot_dir` set up. The output file name will contain the datetime of the first time sample in the data.

## RFI Flagger

rascil\_rcal also implements reading RFI (Radio Frequency Interference) flags and using them as part of the pipeline. Flagging is optional and can be controlled with the `flag_rfi` argument.

RASCIL's Visibility object contains a "flags" data array with the same dimensions as the visibilities. This array is updated with the results of the SKA Processing Function Library [RFI Flagger](#), which uses the sum-threshold method for flagging. The RFI flagger requires *initial threshold* and *rho* values (both needed to provide a list of thresholds used for finding RFI signal in the data), which can be set via CLI arguments, though we recommend using the defaults at this stage.

## Example script

The following runs the real time calibration pipeline on an MS generated by the MID continuum imaging simulations (with an optional input components file):

```
#!/bin/bash
python3 $RASCIL/rascil/apps/rascil_rcal.py \
--ingest_msname SKA_MID_SIM_custom_B2_dec_-45.0_nominal_nchan100_actual.ms \
--ingest_components_file SKA_MID_SIM_custom_B2_dec_-45.0_nominal_nchan100_components.hdf
```

There are also additional options if you want the sky model to have primary beams applied. Currently we support internal beam from MID and LOW, or additional beam file (in FITS format). An example:

```
#!/bin/bash
python3 $RASCIL/rascil/apps/rascil_rcal.py \
--ingest_msname myms.ms \
--ingest_components_file my_components.hdf \
--apply_beam True --ingest_beam_file my_beam.fits \
```

## Command line arguments

RASCIL RCAL simulator

```
usage: rascil_rcal.py [-h] [--ingest_msname INGEST_MSNAME]
                    [--ingest_dd [INGEST_DD ...]] [--logfile LOGFILE]
                    [--ingest_components_file INGEST_COMPONENTS_FILE]
                    [--apply_beam APPLY_BEAM]
                    [--ingest_beam_file INGEST_BEAM_FILE] [--cal_type {T,G}]
                    [--do_plotting DO_PLOTTING] [--plot_dir PLOT_DIR]
                    [--use_previous_gaintable USE_PREVIOUS_GAINTABLE]
                    [--phase_only_solution PHASE_ONLY_SOLUTION]
                    [--solution_tolerance SOLUTION_TOLERANCE]
                    [--flag_rfi FLAG_RFI]
                    [--initial_threshold INITIAL_THRESHOLD] [--rho RHO]
```

## Named Arguments

<b>--ingest_msname</b>	MeasurementSet to be read
<b>--ingest_dd</b>	Data descriptors in MS to read (all must have the same number of channels) Default: [0]
<b>--logfile</b>	Name of logfile (default is to construct one from msname)
<b>--ingest_components_file</b>	Name of components file (HDF5/txt) format
<b>--apply_beam</b>	If yes, apply primary beam correction to the ingested components Default: False
<b>--ingest_beam_file</b>	Name of external beam file in FITS format



<b>--cal_type</b>	Possible choices: T, G Type of calibration to perform. T=Atmospheric Phase, G=Electronics Gain Default: "T"
<b>--do_plotting</b>	If yes, plot the gain table values over time Default: False
<b>--plot_dir</b>	Full path of the directory to save the gain plots into (default is the same directory the MS file is located)
<b>--use_previous_gaintable</b>	Use previous gaintable as starting point for solution Default: "False"
<b>--phase_only_solution</b>	Solution should be for phases only Default: "True"
<b>--solution_tolerance</b>	Tolerance for solution: stops iteration when changes below this level Default: 1e-12
<b>--flag_rfi</b>	Whether to run the RFI flagger (before obtaining calibration solutions), or not. Default: "False"
<b>--initial_threshold</b>	The initial threshold to be used by the flagger. Used for calculating a list of thresholds. Note: use default value since flagger is still under development Default: 8.0
<b>--rho</b>	The initial rho used by flagger. Used for calculating a list of thresholds. Note: use default value since flagger is still under development Default: 1.5

## rascil\_vis\_ms

rascil\_vis\_ms is a command line app written using RASCIL for simple visualisation of an MS. It's primary use is for the RFI simulations.

## Example script

The following runs the visualisation on an MS generated by the RFI simulations:

```
#!/bin/bash
# Run this in the directory containing ./simulate_rfi.ms
python3 $RASCIL/rascil/apps/rascil_vis_ms.py --ingest_msname ./simulate_rfi.ms
```

### Command line arguments

RASCIL ms visualisation

```
usage: rascil_vis_ms.py [-h] [--ingest_msname INGEST_MSNAME]
                        [--logfile LOGFILE]
```

### Named Arguments

<b>--ingest_msname</b>	MeasurementSet to be read
<b>--logfile</b>	Name of logfile (default is to construct one from msname)

### rascil\_advise

rascil\_advise is a command line app written using RASCIL. It provides advice on imaging parameters for a CASA MeasurementSet.

### Example script

The following provides advice on an MS generated by the MID continuum imaging simulations:

```
#!/bin/bash
# Run this in the directory containing SKA_MID_SIM_custom_B2_dec_-45.0_nominal_nchan100_
↪ nominal.ms
python3 $RASCIL/rascil/apps/rascil_advise.py --ingest_msname SKA_MID_SIM_custom_B2_dec_-
↪ 45.0_nominal_nchan100_nominal.ms
```

### Command line arguments

RASCIL imaging advise

```
usage: rascil_advise.py [-h] [--ingest_msname INGEST_MSNAME]
                        [--ingest_dd [INGEST_DD ...]] [--logfile LOGFILE]
                        [--guard_band_image GUARD_BAND_IMAGE]
                        [--oversampling_synthesised_beam OVERSAMPLING_SYNTHESISED_BEAM]
                        [--delat DELTA]
```

## Named Arguments

<b>--ingest_msname</b>	MeasurementSet to be read
<b>--ingest_dd</b>	Data descriptors in MS to read (all must have the same number of channels) Default: [0]
<b>--logfile</b>	Name of logfile (default is to construct one from msname)
<b>--guard_band_image</b>	Size of field of view in primary beams Default: 3.0
<b>--oversampling_synthesised_beam</b>	Pixels per synthesised beam Default: 3
<b>--dela</b>	Maximum allowed decorrelation Default: 0.02

## rascil\_image\_check

rascil\_image\_check is a command line app written using RASCIL. It allows simple check on an image statistics. The allowed fields are the statistics checked by qa\_image function within the Image class

## Example script

The following provides a check on the maximum of an image suitable for use in a shell script. The value returned is 0 if the constraint is obeyed and 1 if not:

```
python3 $RASCIL/rascil/apps/rascil_image_check.py --image $RASCIL/data/models/M31_
↪canonical.model.fits --stat max --min 0.0 --max 1.2
```

## Command line arguments

RASCIL image check

```
usage: rascil_image_check.py [-h] [--image IMAGE] [--stat STAT] [--min MIN]
                             [--max MAX]
```

## Named Arguments

<b>--image</b>	Image to be read
<b>--stat</b>	Image QualityAssessment field to check Default: "max"
<b>--min</b>	Minimum value
<b>--max</b>	Maximum value

## imaging\_qa

imaging\_qa is a command line app written using RASCIL. It uses the python package [PyBDSF](#) to find sources in an image and check with the original inputs. Currently it features the following:

- Reads FITS images.
- Finds sources above a certain threshold and outputs the catalogue (in CSV, FITS and skycomponents format). For multi-frequency images, the source detection can be performed on the central channel or average over all channels.
- Produces image statistics and diagnostic plots including: running mean plots of the residual, restored, background and sources and a histogram with fitted Gaussian and power spectrum of the residual are also plotted.
- Optional: Read in the sensitivity image and apply a primary beam correction to the fluxes.
- Optional: Estimate the spectral index by reading in frequency moment images (in FITS format) containing higher order Taylor terms.
- Optional: compares with input source catalogue : takes hdf5 and txt format. The source input should has columns of “RA(deg), Dec(deg), FluxI(Jy), FluxQ(Jy), FluxU(Jy), FluxV(Jy), Ref. Freq.(Hz), Spectral Index”.
- Optional: plot the comparison and error of positions and fluxes for input and output source catalogue.

### Example:

The following runs the a data set from the RASCIL test:

```
#!/bin/bash
# Run this in the directory containing both the
# restored and residual fits files:
python $RASCIL/rascil/apps/imaging_qa_main.py \
--ingest_fitsname_restored test-imaging-pipeline-dask_continuum_imaging_restored.fits \
--ingest_fitsname_residual test-imaging-pipeline-dask_continuum_imaging_residual.fits
```

If a source check is required:

```
#!/bin/bash
# This example deals with the multi-frequency image
python $RASCIL/rascil/apps/imaging_qa_main.py \
--ingest_fitsname_restored test-imaging-pipeline-dask_continuum_imaging_restored_cube.
↪fits \
--check_source True --plot_source True \
--input_source_filename test-imaging-pipeline-dask_continuum_imaging_components.hdf
```

If primary beam correction is required:

```
#!/bin/bash
# This example deals with the multi-frequency image
python $RASCIL/rascil/apps/imaging_qa_main.py \
--ingest_fitsname_restored test-imaging-pipeline-dask_continuum_imaging_restored_cube.
↪fits \
--check_source True --plot_source True --apply_primary True\
--ingest_fitsname_residual test-imaging-pipeline-dask_continuum_imaging_sensitivity.fits_
↪\
--input_source_filename test-imaging-pipeline-dask_continuum_imaging_components.hdf
```

### Supplying arguments from a file:

You can also load arguments into the app from a file.

Example arguments file, called *args.txt*:

```
--ingest_fitsname_restored=test-imaging-pipeline-dask_continuum_imaging_restored.fits
--ingest_fitsname_residual=test-imaging-pipeline-dask_continuum_imaging_residual.fits
--check_source=True
--plot_source=True
```

Make sure each line contains one argument, there is an equal sign between arg and its value, and that there aren't any trailing white spaces in the lines.

Then run the `imaging_qa` code as follows:

```
python imaging_qa_main.py @args.txt
```

Specifying the `@` sign in front of the file name will let the code know that you want to read the arguments from a file instead of directly from the command line.

What happens when the image files, the argument file, and the `imaging_qa` code are not all in the same directory? Let's take the following directory structure as an example:

```
- rascil # this is the root directory of the RASCIL git repository
  - rascil
    - apps
      imaging_qa_main.py
    - my_data
      my_restored_file.fits
      my_residual_file.fits
      args.txt
```

With such a setup, the best way to run the `imaging_qa` code is from the top-level `rascil` directory (the git root directory). Your `args.txt` file will need to contain either the relative or absolute path to your FITS files. E.g.:

```
--ingest_fitsname_restored=rascil/my_data/test-imaging-pipeline-dask_continuum_imaging_
↪restored.fits
--ingest_fitsname_residual=rascil/my_data/test-imaging-pipeline-dask_continuum_imaging_
↪residual.fits
--check_source=True
--plot_source=True
```

And you need to provide similarly the relative or absolute path both to the args file and the code you are running:

```
python rascil/apps/imaging_qa_main.py @rascil/args.txt
```

### Docker image

A Docker image is available at [artefact.skao.int/rascil-imaging-qa](https://artefact.skao.int/rascil-imaging-qa) which can be run with either Docker or Singularity. Instructions can be found at

under **Running the imaging\_qa** section.

### Output plots

A list of plots are generated to analyze the image as well as comparing the input and output source catalogues.

Plots for restored image:

```
..._restored_plot.png # Running mean of restored image
..._sources_plot.png # Running mean of the sources
..._background_plot.png # Running mean of background
..._restored_power_spectrum.png # Power spectrum of restored image
```

Plots for residual image:

```
..._residual_hist.png # Histogram and Gaussian fit of residual image
..._residual_power_spectrum.png # Power spectrum of residual image
```

Plots for position matching:

```
..._position_value.png # RA, Dec values of input and output sources
..._position_error.png # RA, Dec error (output-input)
..._position_distance.png # RA, Dec error with respect to distance from the centre
```

Plots for wide field accuracy:

```
..._position_quiver.png # Quiver plot of the movement of source positions
..._gaussian_beam_position.png # Gaussian fitted beam sizes for output sources
```

Plots for flux matching:

```
..._flux_value.png # Values of output flux vs. input flux of sources
..._flux_ratio.png # Ratio of flux out/flux in
..._flux_histogram.png # Histogram of flux comparison
..._flux_position.png # Flux vs. RA and Dec of the sources
```

Plots for spectral index:

```
..._spec_index.png # Spectral index of input vs output fluxes over frequency.
..._spec_index_diagnostics_dist.png # Spectral index out/in vs. distance to centre
..._spec_index_diagnostics_flux.png # Spectral index out/in vs. input sources flux
```

## Command line arguments

RASCIL continuum imaging checker

```
usage: imaging_qa_main.py [-h]
                        [--ingest_fitsname_restored INGEST_FITSNAME_RESTORED]
                        [--ingest_fitsname_residual INGEST_FITSNAME_RESIDUAL]
                        [--ingest_fitsname_sensitivity INGEST_FITSNAME_SENSITIVITY]
                        [--ingest_fitsname_moment INGEST_FITSNAME_MOMENT]
                        [--finder_beam_maj FINDER_BEAM_MAJ]
                        [--finder_beam_min FINDER_BEAM_MIN]
                        [--finder_beam_pos_angle FINDER_BEAM_POS_ANGLE]
                        [--finder_thresh_isl FINDER_THRESH_ISL]
                        [--finder_thresh_pix FINDER_THRESH_PIX]
                        [--finder_multichan_option FINDER_MULTICHAN_OPTION]
                        [--perform_diagnostics PERFORM_DIAGNOSTICS]
                        [--apply_primary APPLY_PRIMARY]
                        [--use_frequency_moment USE_FREQUENCY_MOMENT]
                        [--telescope_model TELESCOPE_MODEL]
                        [--check_source CHECK_SOURCE]
                        [--plot_source PLOT_SOURCE]
                        [--input_source_filename INPUT_SOURCE_FILENAME]
                        [--match_sep MATCH_SEP] [--flux_limit FLUX_LIMIT]
                        [--trim_image TRIM_IMAGE] [--trim_box TRIM_BOX]
                        [--quiet_bdsf QUIET_BDSF]
                        [--source_file SOURCE_FILE]
                        [--rascil_source_file RASCIL_SOURCE_FILE]
                        [--logfile LOGFILE]
                        [--savefits_rmsim SAVEFITS_RMSIM]
                        [--restart RESTART] [--use_dask USE_DASK]
                        [--dask_scheduler DASK_SCHEDULER]
                        [--dask_memory DASK_MEMORY]
                        [--dask_nworkers DASK_NWORKERS]
                        [--dask_nthreads DASK_NTHREADS]
```

## Named Arguments

- ingest\_fitsname\_restored** FITS file of the restored image to be read
- ingest\_fitsname\_residual** FITS file of the residual image to be read
- ingest\_fitsname\_sensitivity** FITS file of the sensitivity image to be read
- ingest\_fitsname\_moment** FITS file of the frequency moment images to be read (Note: Use the prefix of the fits files, e.g. if the restored image is test\_image\_restored.fits here should input test\_image)
- finder\_beam\_maj** Major axis of the restoring beam (degrees) (usually not needed, passed in restored image)  
Default: 1.0
- finder\_beam\_min** Minor axis of the restoring beam (degrees) (usually not needed, passed in restored image)

	Default: 1.0
<b>--finder_beam_pos_angle</b>	Positioning angle of the restoring beam (degrees) (usually not needed, passed in restored image)
	Default: 0.0
<b>--finder_thresh_isl</b>	Threshold to determine the size of the islands used in BDSF (Blob Detector and Source Finder)
	Default: 5.0
<b>--finder_thresh_pix</b>	Threshold to detect source (peak value) used in BDSF
	Default: 10.0
<b>--finder_multichan_option</b>	For multi-channel images, what mode to perform source detection on (single or average)
	Default: "single"
<b>--perform_diagnostics</b>	Whether to perform diagnostics of the images (restored and residual)
	Default: "False"
<b>--apply_primary</b>	Whether to divide by primary beam after BDSF to correct source flux
	Default: "False"
<b>--use_frequency_moment</b>	Whether to use frequency moment images after BDSF to correct spectral index
	Default: "False"
<b>--telescope_model</b>	The telescope to generate primary beam correction
	Default: "MID"
<b>--check_source</b>	Option to check with original input source catalogue
	Default: "False"
<b>--plot_source</b>	Option to plot position and flux errors for source catalogue
	Default: "False"
<b>--input_source_filename</b>	If use external source file, the file name of source file
<b>--match_sep</b>	Maximum separation in radians for the source matching
	Default: 1e-05
<b>--flux_limit</b>	Minimum flux where comparison plots are generated
	Default: 0.001
<b>--trim_image</b>	For spectral index calculation, do we trim the image to avoid the edge effects?
	Default: "False"
<b>--trim_box</b>	If trim_image is true, proportion of the box that is trimmed (default is 3%)
	Default: 0.03
<b>--quiet_bdsf</b>	If True, suppress bdsf.process_image() text output to screen. Output is still sent to the log file.
	Default: "False"
<b>--source_file</b>	Name of output source file



<b>--rascil_source_file</b>	Name of output RASCIL components hdf file
<b>--logfile</b>	Name of output log file
<b>--savefits_rmsim</b>	This parameter is a Boolean (default is False). If True, save background rms image as a FITS file. Default: “False”
<b>--restart</b>	If true, surpass BDSF when the output already exists. The checker will start from reading the BDSF csv file Default: “False”
<b>--use_dask</b>	Default: “False”
<b>--dask_scheduler</b>	Externally defined Dask scheduler e.g. 127.0.0.1:8786 or ssh for SSHCluster or existing for current scheduler
<b>--dask_memory</b>	Memory per Dask worker (GB), e.g. 5GB (None means Dask will choose)
<b>--dask_nworkers</b>	Number of workers (None means Dask will choose)
<b>--dask_nthreads</b>	Number of threads in each Dask worker (None means Dask will choose)

## performance\_analysis

performance\_analysis is a command line app written using RASCIL. It helps in analysis of performance files written by rascil\_imager.

The performance files can be obtained using a script to iterate over some parameter. For example:

```
#!/usr/bin/env bash
#
results_dir=${HOME}/results/5km_resource_modelling
for int_time in 2880 1440 720 360
do
    mshome=${HOME}/data/int_time${int_time}
    for npixel in 512 1024 2048 4096 8192
    do
        results_dir=${HOME}/data/int_time${int_time}_npixel${npixel}
        mkdir -p ${results_dir}
        python3 ${RASCIL}/rascil/apps/rascil_imager.py --mode cip \
            --clean_nmoment 3 --clean_facets 4 --clean_nmajor 10 \
            --clean_threshold 3e-5 --clean_restore_facets 4 --clean_restore_overlap 32 \
            --use_dask True --imaging_context ng --imaging_npixel ${npixel} --imaging_pol_
↪stokesI --clean_restored_output list \
            --imaging_cellsize 5e-6 --imaging_weighting uniform --imaging_nchan 1 \
            --ingest_vis_nchan 100 --ingest_chan_per_vis 16 \
            --ingest_msname ${mshome}/SKA_MID_SIM.ms \
            --performance_file ${results_dir}/performance_rascil_imager_${int_time}_${npixel}
↪.json
    done
done
```

In addition, the memory usage can be tracked using a dask plugin. Currently this requires setting up the dask scheduler with the plugin:

```
ssh $scheduler dask-scheduler --port=8786 --preload dask_memusage --memusage-csv \
./performance_rascil_imager_${1}_${2}.csv &
```

## Command line arguments

RASCIL performance analysis

```
usage: performance_analysis.py [-h] [--mode MODE]
                               [--performance_files PERFORMANCE_FILES ...]
                               [--memory_file MEMORY_FILE] [--tag TAG]
                               [--parameters PARAMETERS ...]
                               [--functions FUNCTIONS ...]
                               [--vis_nvis VIS_NVIS] [--verbose VERBOSE]
                               [--results RESULTS]
```

## Named Arguments

<b>--mode</b>	Processing mode: line   bar   contour   summary   fit Default: “summary”
<b>--performance_files</b>	Names of json performance files to analyse: default is all json files in working directory
<b>--memory_file</b>	Name of memusage csv file
<b>--tag</b>	Informational tag used in plot titles and file names Default: “”
<b>--parameters</b>	Name of parameters from cli_args e.g. imaging_npixel_sq, used for line (1 parameter) and contour plots (2 parameters) Default: [‘imaging_npixel_sq’, ‘vis_nvis’]
<b>--functions</b>	Names of values from dask_profile to plot e.g. skymodel_predict_calibrate Default: [‘skymodel_predict_calibrate’, ‘skymodel_calibrate_invert’, ‘invert_ng’, ‘restore_cube’, ‘image_scatter_facets’, ‘image_gather_facets’]
<b>--vis_nvis</b>	Number of visibilities for use if vis_nvis not in json files
<b>--verbose</b>	Verbose output? Default: “False”
<b>--results</b>	Directory for results, default is current directory Default: “./”

## Other

### 1.3.5 RASCIL and DASK

RASCIL uses Dask for distributed processing:

<http://dask.pydata.org/en/latest/>

<https://github.com/dask/dask-tutorial>

Running RASCIL and Dask on a single machine is straightforward. First define a graph and then compute it either by calling the compute method of the graph or by passing the graph to a dask client.

A typical graph will flow from a set of input visibility sets to an image or set of images. In the course of constructing a graph, we will need to know the data elements and the functions transforming between them. These are well-modeled in RASCIL.

In order that Dask.delayed processing can be switched on and off, and that the same code is used for Dask and non-Dask processing, we have wrapped Dask.delayed in `rascil.workflows.rsexecute.execution_support()`. An example is:

```
rsexecute.set_client(use_dask=True)
continuum_imaging_list = \
    continuum_imaging_list_rsexecute_workflow(vis_list, model_imagelist=self.model_
    ↪ imagelist, context='2d',
                                           algorithm='mmclean', facets=1,
                                           scales=[0, 3, 10],
                                           niter=1000, fractional_threshold=0.1,
                                           nmoments=2, nchan=self.freqwin,
                                           threshold=2.0, nmajor=5, gain=0.1,
                                           deconvolve_facets=8, deconvolve_overlap=16,
                                           deconvolve_taper='tukey')
clean, residual, restored = rsexecute.compute(continuum_imaging_list, sync=True)
```

By default, rsexecute is initialised to use the Dask process scheduler with one worker per core. This can be changed by a call to rsexecute.set\_client:

```
rsexecute.set_client(use_dask=True, nworkers=4)
```

If use\_dask is True then a Dask graph is constructed via calls to rsexecute.execute() for subsequent execution.

If use\_dask is False then the named function is called immediately, and the execution is therefore single threaded:

```
rsexecute.set_client(use_dask=False)
```

Note that debugging is easiest if Dask is switched off (use\_dask=False)

The pipeline workflow `rascil.workflows.rsexecute.pipelines.continuum_imaging_list_rsexecute_workflow()` is itself assembled using the execution framework (an interface to Dask): `rascil.workflows.rsexecute.execution_support()`.

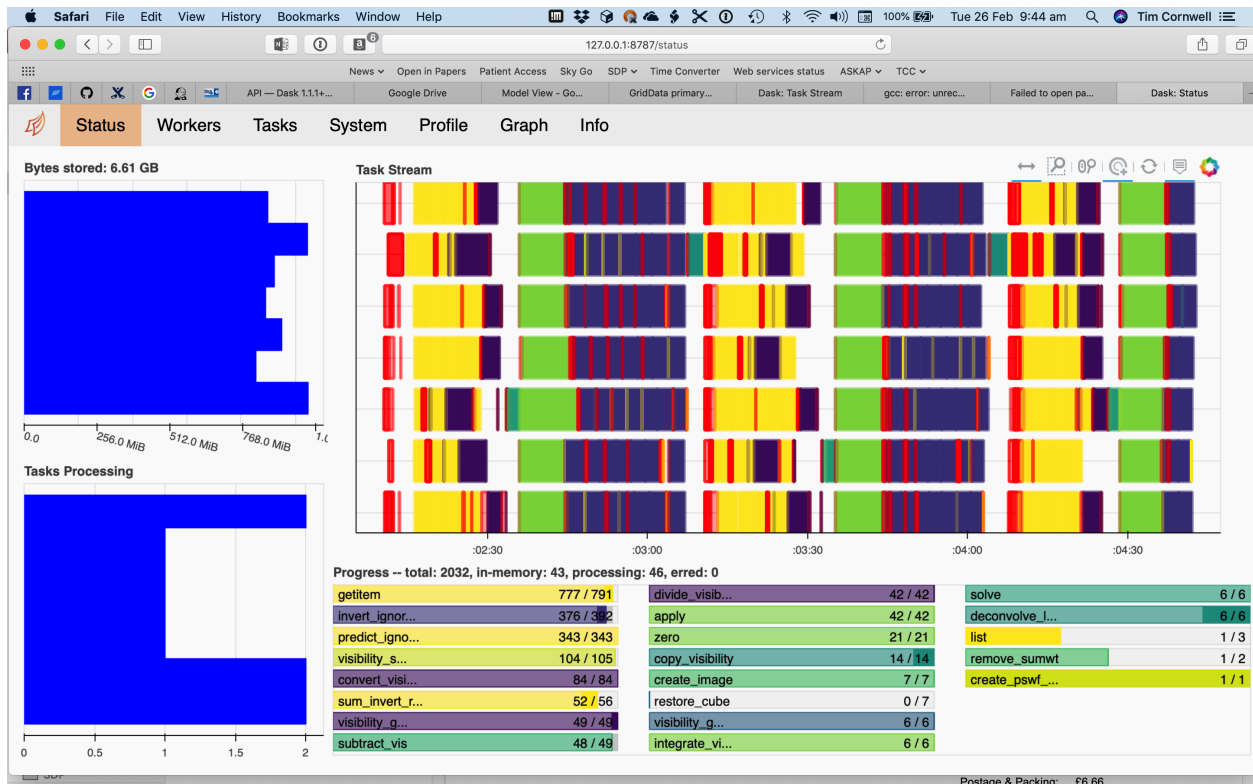
The functions for creating graphs are:

- Calibrate workflow: `rascil.workflows.rsexecute.calibration.calibrate_list_rsexecute_workflow()`
- Invert: `rascil.workflows.rsexecute.imaging.invert_list_rsexecute_workflow()`
- Predict: `rascil.workflows.rsexecute.imaging.predict_list_rsexecute_workflow()`
- Deconvolve: `rascil.workflows.rsexecute.imaging.deconvolve_list_rsexecute_workflow()`

- ICAL: `rascil.workflows.rsexecute.pipelines.ical_list_rsexecute_workflow()`
- Continuum imaging: `rascil.workflows.rsexecute.pipelines.continuum_imaging_list_rsexecute_workflow()`
- Spectral line imaging: `rascil.workflows.rsexecute.pipelines.spectral_line_imaging_list_rsexecute_workflow()`
- MPCCAL: `rascil.workflows.rsexecute.pipelines.mpccal_skymodel_list_rsexecute_workflow()`
- Testing and simulation support: `rascil.workflows.rsexecute.simulation.simulate_list_rsexecute_workflow()`

In addition there are notebooks that use components in workflows/notebooks.

These notebooks are scaled to run on a 2017-era laptop (4 cores, 16GB) but can be changed to larger scales. Both explicitly create a client and output the URL (usually <http://127.0.0.1:8787>) for the Dask diagnostics. Of these the status page is most useful, but the other pages are each worth investigating.



## Using RASCIL and Dask on a cluster

Running on a cluster is a bit more complicated. On a single node, Dask/rsexecute use a process-oriented scheduler. On a cluster, it is necessary to use the distributed scheduler.

You can start the distributed scheduler and workers by hand, using the `dask-ssh` command (more below). To communicate the IP address of the scheduler, set the environment variable `RASCIL_DASK_SCHEDULER` appropriately:

```
export RASCIL_DASK_SCHEDULER=192.168.2.10:8786
```

If you do this, remember to start the workers as well. `dask-ssh` is useful for this:

```
c=get_dask_client(timeout=30)
rsexecute.set_client(c)
```

get\_dask\_client will look for a scheduler via the environment variable RASCIL\_DASK\_SCHEDULER. If that does not exist, it will start a Client using the default Dask approach but that will be a single node scheduler.

Darwin and P3 uses SLURM for scheduling. There is python binding of DRMAA that could in principle be used to queue the processing. However a simple edited job submission script is also sufficient.

On P3, each node has 16 cores, and each core has 8GB. Usually this is sometimes insufficient for RASCIL and so some cores must be not used so the memory can be used by other cores. To run 8 workers and one scheduler on 8 nodes, the SLURM batch file should look something like:

```
#!/bin/bash
#!
#! Dask job script for P3
#! Tim Cornwell
#!
#! Name of the job:
#SBATCH -J IMAGING
#! Which project should be charged:
#SBATCH -A SKA-SDP
#! How many whole nodes should be allocated?
#SBATCH --nodes=8
#! How many (MPI) tasks will there be in total? (<= nodes*16)
#SBATCH --ntasks=8
#! Memory limit: P3 has roughly 107GB per node
#SBATCH --mem 107000
#! How much wallclock time will be required?
#SBATCH --time=23:59:59
#! What types of email messages do you wish to receive?
#SBATCH --mail-type=FAIL,END
#! Where to send email messages
#SBATCH --mail-user=realtimcornwell@gmail.com
#! Uncomment this to prevent the job from being requeued (e.g. if
#! interrupted by node failure or system downtime):
##SBATCH --no-requeue
#! Do not change:
#SBATCH -p compute

#SBATCH --exclusive

#! Modify the settings below to specify the application's environment, location
#! and launch method:

#! Optionally modify the environment seen by the application
#! (note that SLURM reproduces the environment at submission irrespective of ~/.bashrc):
module purge                                # Removes all modules still loaded

#! Set up python
# . $HOME/alaska-venv/bin/activate
export PYTHONPATH=$PYTHONPATH:$ARL
echo "PYTHONPATH is ${PYTHONPATH}"

echo -e "Running python: `which python`"
echo -e "Running dask-scheduler: `which dask-scheduler`"
```

(continues on next page)

(continued from previous page)

```

cd $SLURM_SUBMIT_DIR
echo -e "Changed directory to `pwd`.\\n"

JOBID=${SLURM_JOB_ID}
echo ${SLURM_JOB_NODELIST}

#! Create a hostfile:
scontrol show hostnames $SLURM_JOB_NODELIST | uniq > hostfile.$JOBID

scheduler=$(head -1 hostfile.$JOBID)
hostIndex=0
for host in `cat hostfile.$JOBID`; do
    echo "Working on $host ...."
    if [ "$hostIndex" = "0" ]; then
        echo "run dask-scheduler"
        ssh $host dask-scheduler --port=8786 &
        sleep 5
    fi
    echo "run dask-worker"
    ssh $host dask-worker --nprocs 1 --nthreads 8 --interface ib0 \
        --memory-limit 256GB --local-directory /mnt/storage-ssd/tim/dask-workspace/${host}
    ↪$scheduler:8786 &
        sleep 1
    hostIndex="1"
done
echo "Scheduler and workers now running"

#! We need to tell dask Client (inside python) where the scheduler is running
export ARL_DASK_SCHEDULER=${scheduler}:8786
echo "Scheduler is running at ${scheduler}"

CMD="python ../clean_ms_noniso.py --ngroup 1 --nworkers 0 --weighting uniform --context_
↪wprojectwstack --nwslabs 9 \
--mode pipeline --niter 1000 --nmajor 3 --fractional_threshold 0.2 --threshold 0.01 \
--amplitude_loss 0.02 --deconvolve_facets 8 --deconvolve_overlap 16 --restore_facets 4 \
--msname /mnt/storage-ssd/tim/Code/sim-low-imaging/data/noniso/GLEAM_A-team_EoR1_160_MHz_
↪no_errors.ms \
--time_coal 0.0 --frequency_coal 0.0 --channels 0 1 \
--plot False --fov 2.5 --single False | tee clean_ms.log"

eval $CMD

```

In the command CMD remember to shutdown the Client so the batch script will close the background dask-ssh and then exit.

Thw diagnostic pages can be tunneled. RASCIL emits the URL of the diagnostic page. For example:

```
http://10.143.1.25:8787
```

Then to tunnel the pages:

```
ssh hpccorn1@login.hpc.cam.ac.uk -L8080:10.143.1.25:8787
```

The diagnostic page is available from your local browser at:

```
127.0.0.1:8080
```

## Logging

Logging is difficult when using distributed processing. Here's a solution that works. At the beginning of your script or notebook, define a function to initialize the logger:

```
def init_logging():
    log.info("Logging to %s/clean_ms_dask.log" % cwd)
    logging.basicConfig(filename='%s/clean_ms_dask.log' % cwd,
                        filemode='a',
                        format='%(thread)s %(asctime)s.%(msecs)d %(name)s %(levelname)s
↳ %(message)s',
                        datefmt='%H:%M:%S',
                        level=logging.DEBUG)

log = logging.getLogger()
log.setLevel(logging.INFO)
log.addHandler(logging.StreamHandler(sys.stdout))
log.addHandler(logging.StreamHandler(sys.stderr))
init_logging()
...
rsexecute.run(init_logging)
```

To ensure that the Dask workers get the same setup, you will need to run `init_logging()` on each worker using the `rsexecute.run()` function:

```
rsexecute.run(init_logging)
```

or:

```
rsexecute.set_client(use_dask=True)
rsexecute.run(init_logging)
```

This will log to the same file. It is also possible to set up separate log file per worker by suitably changing `init_logger`.

### 1.3.6 Use of xarray

From release 0.2+, RASCIL has moved to use the [Xarray](#) library instead of numpy in the data classes. RASCIL data classes are now all derived from `xarray.Dataset`. This change is motivated by the large range of capabilities available from xarray. These include:

- Named dimensions and coordinates, allowing access via quantities such as time. frequency, polarisation, receptor
- Indexing, selection, iteration, and conditions
- Support of split-apply-recombine operations
- Interpolation in coordinates, including missing values
- Automatic invocation of Dask for array operations
- Arbitrary meta data as attributes

We have chosen to make the RASCIL data classes derive from `xarray.Dataset`. Instead of adding class methods to the RASCIL data class, which would introduce some interface fragility as `xarray` changes over time, we have used data accessors to control access to methods specific to the class. This design is suggested in the `xarray` documentation on extending `xarray`. Examples:

```
# Flagged visibility
vis.visibility_acc.flagged_vis

# UVW in wavelengths
vis.visibility_acc.uvw_lambda

# DataArray sizes
vis.visibility_acc.datasizes

# Phasecentre as an astropy.SkyCoord
im.image_acc.phasecentre

# Image RA, Dec grid
im.image_acc.ra_dec_mesh

# Gaintable number of receptors
gt.gaintable_acc.nrec
```

For examples of the capabilities afforded by `xarray` see the jupyter notebooks below:

Here is a simple example of how the capabilities of `xarray` can be used:

```
vis = create_visibility_from_ms(ms)[0]

# Don't squeeze out the unit dimensions because we will want
# them for the concat
chan_vis = [v[1] for v in vis.groupby_bins(dim, bins=2)]

# Predict visibility from a model.
chan_vis = [predict_ng(vis, model) in chan_vis]

# Now concatenate
newvis = xarray.concat(chan_vis, dim=dim, data_vars="minimal")
```

### 1.3.7 Conversion from previous data classes

The steps required are:

- For Image, GridData, and ConvolutionFunction, the name of the data variable is `pixels` so for example:

```
# The previous numpy format
im.data
# as an Xarray becomes
im["pixels"]
# as an numpy array or Dask array becomes
im["pixels"].data
# as an numpy array becomes
im["pixels"].values
```

(continues on next page)



(continued from previous page)

```
# The properties now require using the accessor class. For example:
im.nchan
# becomes
im.image_acc.nchan
# or directly to the attributes of the xarray.Dataset
im.attrs["nchan"]
```

- For Visibility, the various columns become data variables:

```
# The numpy format
bvis.data["vis"]
# becomes
bvis["vis"]
# as an numpy array or Dask array becomes
bvis["vis"].data
# as an numpy array becomes
bvis["vis"].values
# The properties now require using the accessor class. For example:
bvis.nchan
# becomes
bvis.visibility_acc.nchan
# or directly to the attributes of the xarray.Dataset
bvis.attrs["nchan"]
# The convenience methods for handling flags also require the accessor:
bvis.flagged_vis
# becomes
bvis.visibility_acc.flagged_vis
```

### 1.3.8 RASCIL and WAGG

RASCIL can use GPU-based version of nifty-gridder called WAGG for the gridding-degridding operations:

<https://gitlab.com/ska-telescope/sdp/ska-gridder-nifty-cuda/-/tree/sim-874-python-wrapper>

There are two function counterparts to `predict_ng` and `invert_ng` called `predict_wg` and `invert_wg`,

WAGG needs to be installed from its repository after the RASCIL installation. WAGG uses numpy to build the installation wheel, and it will download the recent one if numpy is absent in a system. The numpy version mismatch can cause the WAGG crash. By installing WAGG after RASCIL, we make sure it uses the numpy version that RASCIL requires for the build.

#### Installing WAGG module

To install WAGG it is required to clone the repository, switch to the python wrapper branch, change to `python` folder and run `pip install .`, i.e.:

```
git clone https://gitlab.com/ska-telescope/sdp/ska-gridder-nifty-cuda.git
cd ska-gridder-nifty-cuda
git checkout --track origin/sim-874-python-wrapper
cd python
pip install .
```

Alternatively, WAGG can be installed directly with `pip`:

```
pip install git+http://gitlab.com/ska-telescope/sdp/ska-gridder-nifty-cuda.git@sim-874-  
python-wrapper#subdirectory=python
```

## Using WAGG GPU-based predict and invert functions

WAGG module makes a use of Nvidia runtime system, called *NVRTC*. It is a runtime compilation library for CUDA C++. It accepts CUDA C++ source code in the form of a string, and outputs GPU-specific PTX (Parallel Thread Execution) instructions. The PTX code generated by NVRTC can be loaded and linked with other modules of the CUDA Driver API. More information on *NVRTC* can be found on CUDA website, <https://docs.nvidia.com/cuda/nvrtc/index.html>.

When the runtime support is installed, the functions `predict_wg` and `invert_wg` can be used as the CPU-based `predict_ng` and `invert_ng` since the parameters are the same. One can find an example on how to use the functions `predict_ng` and `invert_ng` in the Imaging and deconvolution demonstration Jupyter notebook in [Examples](#) section.

## 1.4 API

Here is a quick guide to the layout of the package:

- `rascil.processing_components`: Processing functions used in algorithms
- `rascil.workflows`: Distributed processing workflows
- `rascil.apps`: CLI apps
- `examples`: Example scripts and notebooks
- `tests`: Unit and regression tests
- `docs`: Complete documentation. Includes non-interactive output of examples.
- `rascil.data`: Data used for simulations

The API is specified in the `rascil` directory.

### 1.4.1 Processing Components

#### Calibration

Calibration is performed by fitting observed visibilities to a model visibility.

The scalar equation to be minimised is:

$$S = \sum_{t,f} \sum_{i,j} w_{t,f,i,j} \left| V_{t,f,i,j}^{\text{obs}} - J_i J_j^* V_{t,f,i,j}^{\text{mod}} \right|^2$$

The least squares fit algorithm uses an iterative substitution (or relaxation) algorithm from Larry D'Addario in the late seventies.

## rascil.processing\_components.calibration.iterators Module

GainTable iterators for iterating through a GainTable

### Functions

<code><i>gaintable_timeslice_iter</i>(gt, **kwargs)</code>	GainTable iterator
--	--------------------

### `gaintable_timeslice_iter`

`gaintable_timeslice_iter(gt: GainTable, **kwargs) → ndarray`

GainTable iterator

#### Parameters

- **gt** – GainTable
- **timeslice** – ‘auto’ or time in seconds
- **gaintable\_slices** – Number of slices (second in precedence to timeslice)

#### Returns

Boolean array with selected rows=True

## rascil.processing\_components.calibration.operations Module

Functions for calibration, including creation of gaintables, application of gaintables, and merging gaintables.

### Functions

<code><i>append_gaintable</i>(gt, othergt)</code>	Append othergt to gt
<code><i>create_gaintable_from_rows</i>(gt, rows[, make-copy])</code>	Create a GainTable from selected rows
<code><i>gaintable_plot</i>(gt[, cc, title, ants, ...])</code>	Standard plot of gain table

### `append_gaintable`

`append_gaintable(gt: GainTable, othergt: GainTable) → GainTable`

Append othergt to gt

#### Parameters

- **gt** –
- **othergt** –

#### Returns

GainTable gt + othergt

### create\_gaintable\_from\_rows

**create\_gaintable\_from\_rows**(*gt: GainTable, rows: ndarray, makecopy=True*) → Optional[GainTable]

Create a GainTable from selected rows

#### Parameters

- **gt** – GainTable
- **rows** – Boolean array of row selection
- **makecopy** – Make a deep copy (True)

#### Returns

GainTable

### gaintable\_plot

**gaintable\_plot**(*gt: GainTable, cc='T', title='', ants=None, channels=None, label\_max=0, min\_amp=1e-05, cmap='rainbow', \*\*kwargs*)

Standard plot of gain table

#### Parameters

- **gt** – Gaintable
- **cc** – Type of gain table e.g. 'T', 'G', 'B'
- **value** – 'amp' or 'phase' or 'residual'
- **ants** – Antennas to plot
- **channels** – Channels to plot
- **kwargs** –

#### Returns

## Flagging

### rascil.processing\_components.flagging.operations Module

Functions that Flagging Visibility and Visibility.

The flags of Visibility has axes [chan, pol, z, y, x] where z, y, x are spatial axes in either sky or Fourier plane. The order in the WCS is reversed so the grid\_WCS describes UU, VV, WW, STOKES, FREQ axes.

## Functions

<code>flagging_visibility(bvis[, baselines, ...])</code>	Flagging Visibility
<code>flagging_aoflagger(vis, strategy_name)</code>	Flagging Visibility using the AOFlagger package The flagging strategy can be telescope name (AARTFAAC, ARECIBO, ARECIBO 305M, BIGHORNS, EVLA, JVLA, LOFAR, MWA, PARKES, PKS, ATPKSMB, or WSRT) or a LUA file like the ones in <a href="https://gitlab.com/aroffringa/aoflagger/-/tree/master/data/strategies">https://gitlab.com/aroffringa/aoflagger/-/tree/master/data/strategies</a> .

## flagging\_visibility

**flagging\_visibility**(*bvis*, *baselines*=None, *antennas*=None, *channels*=None, *polarisations*=None)

Flagging Visibility

### Parameters

- **bvis** – Visibility
- **baselines** – The list of baseline numbers to flag
- **antennas** – The list of antenna number to flag
- **channels** – The list of Channel number to flag
- **polarisations** – The list of polarisations to flag

### Returns

Visibility

## flagging\_aoflagger

**flagging\_aoflagger**(*vis*, *strategy\_name*)

Flagging Visibility using the AOFlagger package The flagging strategy can be telescope name (AARTFAAC, ARECIBO, ARECIBO 305M, BIGHORNS, EVLA, JVLA, LOFAR, MWA, PARKES, PKS, ATPKSMB, or WSRT) or a LUA file like the ones in <https://gitlab.com/aroffringa/aoflagger/-/tree/master/data/strategies> . You can define a new strategy interactively using the AOFlagger rfigui ([https://aoflagger.readthedocs.io/en/latest/using\\_rfigui.html](https://aoflagger.readthedocs.io/en/latest/using_rfigui.html))

### Parameters

- **vis** – Visibility object
- **strategy\_name** – Strategy to use: can be a LUA file or a telescope name. If the strategy for the selected telescope is not available, a generic strategy is used.

### Returns

Visibility where the flags field has been updated

## Gridding Data

### rascil.processing\_components.griddata.convolution\_functions Module

Functions that define and manipulate ConvolutionFunctions.

The griddata has axes [chan, pol, z, dy, dx, y, x] where z, y, x are spatial axes in either sky or Fourier plane. The order in the WCS is reversed so the grid\_WCS describes UU, VV, DUU, DVV, WW, STOKES, FREQ axes.

GridData can be used to hold the Fourier transform of an Image or gridded visibilities. In addition, the convolution function can be stored in a GridData, most probably with finer spatial sampling.

## Functions

<code>calculate_bounding_box_convolutionfunction(cf)</code>	Calculate bounding boxes
<code>apply_bounding_box_convolutionfunction(cf[, ...])</code>	Apply a bounding box to a convolution function
<code>export_convolutionfunction_to_fits(cf[, ...])</code>	Write a convolution function to fits

### calculate\_bounding\_box\_convolutionfunction

**calculate\_bounding\_box\_convolutionfunction**(*cf*, *fractional\_level*=0.0001)

Calculate bounding boxes

Returns a list of bounding boxes where each element is (z, (y0, y1), (x0, x1))

These can be used in griddata/degridding.

#### Parameters

- **cf** –
- **fractional\_level** –

#### Returns

list of bounding boxes

### apply\_bounding\_box\_convolutionfunction

**apply\_bounding\_box\_convolutionfunction**(*cf*, *fractional\_level*=0.0001)

Apply a bounding box to a convolution function

#### Parameters

- **cf** –
- **fractional\_level** –

#### Returns

bounded convolution function

### export\_convolutionfunction\_to\_fits

**export\_convolutionfunction\_to\_fits**(*cf*: *ConvolutionFunction*, *fitsfile*: *str* = 'cf.fits')

Write a convolution function to fits

#### Parameters

- **cf** – ConvolutionFunction
- **fitsfile** – Name of output fits file in storage

#### Returns

None

#### See also

`rascil.processing_components.image.operations.import_image_from_array()`

## rascil.processing\_components.griddata.kernels Module

Functions that define and manipulate kernels

### Functions

<code>create_pswf_convolutionfunction(im[, ...])</code>	Fill an Anti-Aliasing filter into a ConvolutionFunction
<code>create_box_convolutionfunction(im[, ...])</code>	Fill a box car function into a ConvolutionFunction
<code>create_awterm_convolutionfunction(im[, ...])</code>	Fill AW projection kernel into a GridData.
<code>create_vpterm_convolutionfunction(im[, ...])</code>	Fill voltage pattern kernel projection kernel into a Grid-Data.

### create\_pswf\_convolutionfunction

**create\_pswf\_convolutionfunction**(*im*, *oversampling*=127, *support*=8, *polarisation\_frame*=None)

Fill an Anti-Aliasing filter into a ConvolutionFunction

Fill the Prolate Spheroidal Wave Function into a GridData with the specified oversampling. Only the inner non-zero part is retained

Also returns the griddata correction function as an image

#### Parameters

- **im** – Image template
- **oversampling** – Oversampling of the convolution function in uv space

#### Returns

griddata correction Image, griddata kernel as ConvolutionFunction

### create\_box\_convolutionfunction

**create\_box\_convolutionfunction**(*im*, *oversampling*=1, *support*=1, *polarisation\_frame*=None)

Fill a box car function into a ConvolutionFunction

Also returns the griddata correction function as an image

#### Parameters

- **im** – Image template
- **oversampling** – Oversampling of the convolution function in uv space

#### Returns

griddata correction Image, griddata kernel as ConvolutionFunction

### `create_awterm_convolutionfunction`

```
create_awterm_convolutionfunction(im, make_pb=None, nw=1, wstep=1000000000000000.0,
                                   oversampling=9, support=8, use_aaf=True, maxsupport=512,
                                   pa=None, normalise=True, polarisation_frame=None)
```

Fill AW projection kernel into a GridData.

#### Parameters

- **im** – Image template
- **make\_pb** – Function to make the primary beam model image (hint: use a partial)
- **nw** – Number of w planes
- **wstep** – Step in w (wavelengths)
- **oversampling** – Oversampling of the convolution function in uv space

#### Returns

griddata correction Image, griddata kernel as GridData

### `create_vp_term_convolutionfunction`

```
create_vp_term_convolutionfunction(im, make_vp=None, oversampling=8, support=6, use_aaf=False,
                                   maxsupport=512, pa=None, normalise=True,
                                   polarisation_frame=None)
```

Fill voltage pattern kernel projection kernel into a GridData.

The makes the convolution function for gridding polarised data with a voltage pattern.

#### Parameters

- **im** – Image template
- **make\_vp** – Function to make the voltage pattern model image (hint: use a partial)
- **oversampling** – Oversampling of the convolution function in uv space

#### Returns

griddata correction Image, griddata kernel as GridData

## Images

### `rascil.processing_components.image.gradients` Module

Image operations visible to the Execution Framework as Components



## Functions

<i>image_gradients</i> (im)	Calculate image first order gradients numerically
-----------------------------	---

### image\_gradients

**image\_gradients**(*im*: Image)

Calculate image first order gradients numerically

Two images are returned: one with respect to x and one with respect to y

Gradient units are (incoming unit)/pixel e.g. Jy/beam/pixel

#### Parameters

**im** – Image

#### Returns

Gradient images

## rascil.processing\_components.image.operations Module

Image operations visible to the Execution Framework as Components

## Functions

<i>add_image</i> (im1, im2)	Add two images
<i>average_image_over_frequency</i> (im)	Integrate image across frequency
<i>create_w_term_like</i> (im, w[, phasecentre, ...])	Create an image with a w term phase term in it:
<i>create_window</i> (template, window_type, **kwargs)	Create a window image using one of a number of methods
<i>fft_image_to_griddata_with_wcs</i> (im)	WCS-aware FFT of a canonical image
<i>import_image_from_fits</i> (fitsfile[, fixpol])	Read an Image from fits
<i>pad_image</i> (im, shape)	Pad an image to desired shape, adding equally to all edges
<i>sub_image</i> (im, shape)	Subsection an image to desired shape, cutting equally from all edges
<i>polarisation_frame_from_wcs</i> (wcs, shape)	Convert wcs to polarisation_frame
<i>remove_continuum_image</i> (im[, degree, mask])	Fit and remove continuum visibility in place
<i>reproject_image</i> (im, newwcs[, shape])	Re-project an image to a new coordinate system
<i>show_components</i> (im, comps[, npixels, fig, ...])	Show components against an image
<i>show_image</i> (im[, fig, title, pol, chan, cm, ...])	Show an Image with coordinates using matplotlib, optionally with components
<i>smooth_image</i> (model[, width, normalise])	Smooth an image with a 2D Gaussian kernel
<i>scale_and_rotate_image</i> (im[, angle, scale, order])	Scale and then rotate and image in x, y axes
<i>apply_voltage_pattern_to_image</i> (im, vp[, ...])	Apply a voltage pattern to an image

## add\_image

**add\_image**(*im1: Image, im2: Image*) → Image

Add two images

### Parameters

- **im1** – Image
- **im2** – Image

### Returns

Image

## average\_image\_over\_frequency

**average\_image\_over\_frequency**(*im: Image*) → Image

Integrate image across frequency

### Returns

Integrated image

## create\_w\_term\_like

**create\_w\_term\_like**(*im: Image, w, phasecentre=None, remove\_shift=False, dopol=False*) → Image

Create an image with a w term phase in it:

$$I(l, m) = e^{-2\pi j(w(\sqrt{1-l^2-m^2}-1))}$$

The phasecentre is used as the delay centre for the w term (i.e. where n==0)

### Parameters

- **im** – template image
- **phasecentre** – SkyCoord definition of phasecentre
- **w** – w value to evaluate
- **remove\_shift** –
- **dopol** – Do screen in polarisation?

### Returns

Image

## create\_window

**create\_window**(*template, window\_type, \*\*kwargs*)

Create a window image using one of a number of methods

The window is 1.0 or 0.0

### window types:

‘quarter’: Inner quarter of the image

‘no\_edge’: ‘window\_edge’ pixels around edge set to zero

‘threshold’: template image pixels < ‘window\_threshold’ absolute value set to zero

**Parameters**

- **template** – Template image
- **window\_type** – ‘quarter’ | ‘no\_edge’ | ‘threshold’

**Returns**

New image containing window

**See also**

`rascil.processing_components.image.deconvolution.deconvolve_cube()`

**fft\_image\_to\_griddata\_with\_wcs****fft\_image\_to\_griddata\_with\_wcs(im)**

WCS-aware FFT of a canonical image

**The only transforms supported are:**

RA–SIN, DEC–SIN <-> UU, VV XX, YY <-> KX, KY

For example:

```
from rascil.processing_components import create_test_image, fft_image_to_griddata_
↪with_wcs
im = create_test_image()
print(im)
Image:
  Shape: (1, 1, 256, 256)
  WCS: WCS Keywords
Number of WCS axes: 4
CTYPE : 'RA--SIN' 'DEC--SIN' 'STOKES' 'FREQ'
CRVAL : 0.0 35.0 1.0 1000000000.0
CRPIX : 129.0 129.0 1.0 1.0
PC1_1 PC1_2 PC1_3 PC1_4 : 1.0 0.0 0.0 0.0
PC2_1 PC2_2 PC2_3 PC2_4 : 0.0 1.0 0.0 0.0
PC3_1 PC3_2 PC3_3 PC3_4 : 0.0 0.0 1.0 0.0
PC4_1 PC4_2 PC4_3 PC4_4 : 0.0 0.0 0.0 1.0
CDELTA : -0.000277777791 0.000277777791 1.0 100000.0
NAXIS : 0 0
Polarisation frame: stokesI
print(fft_image_to_griddata_with_wcs(im))
Image:
  Shape: (1, 1, 256, 256)
  WCS: WCS Keywords
Number of WCS axes: 4
CTYPE : 'UU' 'VV' 'STOKES' 'FREQ'
CRVAL : 0.0 0.0 1.0 1000000000.0
CRPIX : 129.0 129.0 1.0 1.0
PC1_1 PC1_2 PC1_3 PC1_4 : 1.0 0.0 0.0 0.0
PC2_1 PC2_2 PC2_3 PC2_4 : 0.0 1.0 0.0 0.0
PC3_1 PC3_2 PC3_3 PC3_4 : 0.0 0.0 1.0 0.0
PC4_1 PC4_2 PC4_3 PC4_4 : 0.0 0.0 0.0 1.0
CDELTA : -805.7218610503596 805.7218610503596 1.0 100000.0
NAXIS : 0 0
Polarisation frame: stokesI
```

**Parameters****im** –**Returns****See also**

`ska_sdp_func_python.fourier_transforms.fft_support.fft()`      `ska_sdp_func_python.fourier_transforms.fft_support.ifft()`

**import\_image\_from\_fits**

**import\_image\_from\_fits**(*fitsfile: str, fixpol=True*) → Image

Read an Image from fits

**Parameters****fitsfile** – FITS file in storage**Returns**

Image

**pad\_image**

**pad\_image**(*im: Image, shape*)

Pad an image to desired shape, adding equally to all edges

Appropriate for standard 4D image with axes (freq, pol, y, x). Only pads in y, x

The wcs crpix is adjusted appropriately.

**Parameters**

- **im** – Image to be padded
- **shape** – Shape in 4 dimensions

**Returns**

Padded image

**sub\_image**

**sub\_image**(*im: Image, shape*)

Subsection an image to desired shape, cutting equally from all edges

Appropriate for standard 4D image with axes (freq, pol, y, x). Only works in y, x

The wcs crpix is adjusted appropriately.

**Parameters**

- **im** – Image to be padded
- **shape** – Shape in 4 dimensions

**Returns**

Padded image

## polarisation\_frame\_from\_wcs

**polarisation\_frame\_from\_wcs**(*wcs, shape*) → *PolarisationFrame*

Convert *wcs* to *polarisation\_frame*

See FITS definition in Table 29 of [https://fits.gsfc.nasa.gov/standard40/fits\\_standard40draft1.pdf](https://fits.gsfc.nasa.gov/standard40/fits_standard40draft1.pdf) or subsequent revision

1 I Standard Stokes unpolarized 2 Q Standard Stokes linear 3 U Standard Stokes linear 4 V Standard Stokes circular 1 RR Right-right circular 2 LL Left-left circular 3 RL Right-left cross-circular 4 LR Left-right cross-circular 5 XX X parallel linear 6 YY Y parallel linear 7 XY XY cross linear 8 YX YX cross linear

stokesI [1] stokesIQUV [1,2,3,4] circular [-1,-2,-3,-4] linear [-5,-6,-7,-8]

### For example::

```
pol_frame = polarisation_frame_from_wcs(im.image_acc.wcs, im["pixels"].data.shape)
```

### Parameters

- **wcs** – World Coordinate System
- **shape** – Shape corresponding to *wcs*

### Returns

*Polarisation\_Frame* object

## remove\_continuum\_image

**remove\_continuum\_image**(*im: Image, degree=1, mask=None*)

Fit and remove continuum visibility in place

Fit a polynomial in frequency of the specified degree where *mask* is *True* and remove it from the image

### Parameters

- **im** –
- **degree** – 1 is a constant, 2 is a slope, etc.
- **mask** – Frequency mask

### Returns

## reproject\_image

**reproject\_image**(*im: ~ska\_sdp\_datamodels.image.image\_model.Image, newwcs: ~astropy.wcs.wcs.WCS, shape=None*) -> (<class 'ska\_sdp\_datamodels.image.image\_model.Image'>, <class 'ska\_sdp\_datamodels.image.image\_model.Image'>)

Re-project an image to a new coordinate system

Currently uses the *reproject* python package. This seems to have some features do be careful using this method. For timeslice imaging *griddata* is used.

### Parameters

- **im** – Image to be reprojected
- **newwcs** – New WCS

- **shape** – Desired shape

**Returns**

Reprojected Image, Footprint Image

**show\_components**

**show\_components**(*im*, *comps*, *npixels=128*, *fig=None*, *vmax=None*, *vmin=None*, *title=""*)

Show components against an image

**Parameters**

- **im** –
- **comps** –
- **npixels** –
- **fig** –

**Returns**

**show\_image**

**show\_image**(*im: Image*, *fig=None*, *title: str = ""*, *pol=0*, *chan=0*, *cm='Greys'*, *components=None*, *vmin=None*, *vmax=None*, *vscale=1.0*)

Show an Image with coordinates using matplotlib, optionally with components

**Parameters**

- **im** – Image
- **fig** – Matplotlib figure
- **title** – String for title of plot
- **pol** – Polarisation to show (index)
- **chan** – Channel to show (index)
- **components** – Optional components to be overlaid
- **vmin** – Clip to this minimum
- **vmax** – Clip to this maximum
- **vscale** – scale max, min by this amount

**Returns**

## smooth\_image

**smooth\_image**(*model: Image, width=1.0, normalise=True*)

Smooth an image with a 2D Gaussian kernel

### Parameters

- **model** – Image
- **width** – Kernel width in pixels
- **normalise** – Normalise kernel peak to unity

## scale\_and\_rotate\_image

**scale\_and\_rotate\_image**(*im, angle=0.0, scale=None, order=5*)

Scale and then rotate an image in x, y axes

Applies scale then rotates

### Parameters

- **im** – Image
- **angle** – Angle in radians
- **scale** – Scale [scale\_x, scale\_y]
- **order** – Order of interpolation (0-5)

### Returns

## apply\_voltage\_pattern\_to\_image

**apply\_voltage\_pattern\_to\_image**(*im: Image, vp: Image, inverse=False, min\_det=0.1, \*\*kwargs*) → Image

Apply a voltage pattern to an image

For each pixel, the application is as follows:

$$I_{\text{corrected}}(l,m) = vp(l,m) I(l,m) jones(j,m).H$$

### Parameters

- **im** – Image to have jones applied
- **vp** – Jones image to be applied
- **inverse** – Apply the inverse (default=False)
- **min\_det** – Minimum determinant to correct

### Returns

new Image with Jones applied

## Imaging

### rascil.processing\_components.imaging.imaging\_params Module

#### Functions

<code>get_rowmap(col[, ucol])</code>	Map to unique cols
<code>get_polarisation_map(vis[, im])</code>	Get the mapping of visibility polarisations to image polarisations
<code>get_frequency_map(vis[, im])</code>	Map channels from visibilities to image

#### get\_rowmap

`get_rowmap(col, ucol=None)`

Map to unique cols

##### Parameters

- **col** – Data column
- **ucol** – Unique values in col

#### get\_polarisation\_map

`get_polarisation_map(vis: Visibility, im: Optional[Image] = None)`

Get the mapping of visibility polarisations to image polarisations

#### get\_frequency\_map

`get_frequency_map(vis, im: Optional[Image] = None)`

Map channels from visibilities to image

### rascil.processing\_components.imaging.primary\_beams Module

Functions to create primary beam and voltage pattern models



## Functions

<code>set_pb_header(pb[, use_local])</code>	Fill in PB header correctly for local coordinates.
<code>create_pb(model[, telescope, ...])</code>	Create an image containing the primary beam for a number of cases
<code>create_pb_generic(model[, pointingcentre, ...])</code>	Create a generic analytical model of the primary beam
<code>create_vp([model, telescope, ...])</code>	Create an image containing the dish voltage pattern for a number of cases
<code>create_vp_generic(model[, pointingcentre, ...])</code>	Create a generic analytical model of the voltage pattern
<code>create_vp_generic_numeric(model[, ...])</code>	Make an image like model and fill it with an analytical model of the primary beam
<code>create_low_test_beam(model[, use_local, azel])</code>	Create a test power beam for LOW
<code>create_low_test_vp(model[, use_local, azel])</code>	Create a test voltage beam for LOW
<code>create_mid_allsky([frequency, npixel, cellsize])</code>	Approximate all sky MID beam
<code>convert_azelvp_to_radec(vp, im, pa)</code>	Convert AZELGEO image to image coords at specific parallactic angle
<code>normalise_vp(vp)</code>	Normalise the vp in place so that the peak gain on axis for parallel pols is equal

### set\_pb\_header

**set\_pb\_header**(*pb*, *use\_local*=True)

Fill in PB header correctly for local coordinates.

There is no convention on how to represent primary beams. We use axes ‘AZELGEO long’ and ‘AZELGEO lati’

#### Parameters

**pb** –

#### Returns

### create\_pb

**create\_pb**(*model*, *telescope*='MID', *pointingcentre*=None, *use\_local*=True)

Create an image containing the primary beam for a number of cases

#### Parameters

- **model** – Template image
- **telescope** – ‘VLA’ or ‘ASKAP’

#### Returns

Primary beam image

### create\_pb\_generic

**create\_pb\_generic**(*model*, *pointingcentre*=None, *diameter*=25.0, *blockage*=1.8, *use\_local*=True)

Create a generic analytical model of the primary beam

Feed legs are ignored

#### Parameters

- **model** –
- **diameter** – Diameter of dish (m)
- **blockage** – Diameter of blockage

#### Returns

### create\_vp

**create\_vp**(*model*=None, *telescope*='MID', *pointingcentre*=None, *padding*=4, *use\_local*=True, *fixpol*=True)

Create an image containing the dish voltage pattern for a number of cases

#### Parameters

- **model** – Template image (Can be None for some cases)
- **telescope** –

#### Returns

Primary beam image

### create\_vp\_generic

**create\_vp\_generic**(*model*, *pointingcentre*=None, *diameter*=25.0, *blockage*=1.8, *use\_local*=True)

Create a generic analytical model of the voltage pattern

Feed legs are ignored

#### Parameters

- **model** –
- **diameter** – Diameter of dish (m)
- **blockage** – Diameter of blockage

#### Returns

### create\_vp\_generic\_numeric

**create\_vp\_generic\_numeric**(*model*, *pointingcentre*=None, *diameter*=15.0, *blockage*=0.0, *taper*='gaussian', *edge*=0.03162278, *zernikes*=None, *padding*=4, *use\_local*=True)

Make an image like model and fill it with an analytical model of the primary beam

The elements of the analytical model are: - dish, optionally blocked - Gaussian taper, default is -12dB at the edge - Offset to pointing centre (optional) - zernikes in a list of dictionaries. Each list element is of the form {“coeff”:0.1, “noll”:5}. See aotools for more details - Output image can be in RA, DEC coordinates or AZELGEO coordinates (the default). *use\_local*=True means to use AZELGEO coordinates centered on 0deg 0deg.

The dish is zero padded according to padding and FFT'ed to get the voltage pattern.

#### Parameters

- **model** –
- **pointingcentre** – SkyCoord of desired pointing centre
- **diameter** – Diameter of dish in metres
- **blockage** – Blockage of dish in metres
- **taper** – “Gaussian” or None
- **edge** – Value of taper at the end of the dish (default corresponds to -12dB)
- **zernikes** – Zernikes to be applied as phase across the dish (see above)
- **padding** – Pad the image by this amount
- **use\_local** – Use local frame (AZELGEO)?

#### Returns

### create\_low\_test\_beam

**create\_low\_test\_beam**(*model: Image, use\_local=True, azel=None*) → Image

Create a test power beam for LOW

This uses an approximation that ignores the antennas

#### Parameters

- **model** – Template image
- **use\_local** – Use az el coordinates instead of ra dec
- **azel** – Tuple (Azimuth, Elevation) radians

### create\_low\_test\_vp

**create\_low\_test\_vp**(*model: Image, use\_local=True, azel=None*) → Image

Create a test voltage beam for LOW

This uses an approximation that ignores the antennas

#### Parameters

- **model** – Template image
- **use\_local** – Use az el coordinates instead of ra dec
- **azel** – Tuple (Azimuth, Elevation) radians

#### Returns

Image

## create\_mid\_allsky

**create\_mid\_allsky**(*frequency=array([1.e+09])*, *npixel=512*, *cellsize=None*)

Approximate all sky MID beam

Unlocked 15m dish with no taper. Actual sidelobes are likely to be lower than this model implies.

### Parameters

- **frequency** – Frequencies to use array(float) (Hz) default is [1e9]
- **npixel** – Number of pixels per axis (int) Default is 512
- **cellsize** – Cellsize in radians. Default is pi/npixel

### Returns

Image

## convert\_azelpv\_to\_radec

**convert\_azelpv\_to\_radec**(*vp*, *im*, *pa*)

Convert AZELGEO image to image coords at specific parallactic angle

### Parameters

- **pb** – Primary beam or voltage pattern
- **im** – Template image
- **pa** – Parallactic angle (radians)

### Returns

## normalise\_vp

**normalise\_vp**(*vp*)

Normalise the vp in place so that the peak gain on axis for parallel pols is equal

### Parameters

**vp** –

### Returns

## Simulation

### rascil.processing\_components.simulation.atmospheric\_screen Module

Functions for tropospheric and ionospheric modeling : see [SDP Memo 97](#)

## Functions

<code>find_pierce_points(station_locations, ha, ...)</code>	Find the pierce points for a flat screen at specified height
<code>create_gaintable_from_screen(vis, sc, screen)</code>	Create gaintables from a screen calculated using ARatmospy
<code>grid_gaintable_to_screen(vis, gaintables, screen)</code>	Grid a gaintable to a screen image
<code>calculate_sf_from_screen(screen)</code>	Calculate structure function image from screen
<code>plot_gaintable_on_screen(vis, gaintables[, ...])</code>	Plot a gaintable on an ionospheric screen

### find\_pierce\_points

**find\_pierce\_points**(*station\_locations, ha, dec, phasecentre, height*)

Find the pierce points for a flat screen at specified height

A pierce point is where the line of site from a station or dish to a source passes through a thin screen

#### Parameters

- **station\_locations** – station locations [:3]
- **ha** – Hour angle
- **dec** – Declination
- **phasecentre** – Phase centre
- **height** – Height of screen

#### Returns

### create\_gaintable\_from\_screen

**create\_gaintable\_from\_screen**(*vis, sc, screen, height=None, vis\_slices=None, r0=5000.0, type\_atmosphere='ionosphere', reference\_component=None, jones\_type='B', \*\*kwargs*)

Create gaintables from a screen calculated using ARatmospy

Screen axes are ['XX', 'YY', 'TIME', 'FREQ']

#### Parameters

- **vis** –
- **sc** – Sky components for which pierce points are needed
- **screen** – Image or string (for fits file which will be memory mapped in
- **height** – Height (in m) of screen above telescope e.g. 3e5
- **r0** – r0 in meters
- **type\_atmosphere** – 'ionosphere' or 'troposphere'
- **reference** – Use the first component as a reference
- **jones\_type** – Type of calibration matrix T or G or B

#### Returns

### grid\_gaintable\_to\_screen

**grid\_gaintable\_to\_screen**(*vis*, *gaintables*, *screen*, *height*=300000.0, *gaintable\_slices*=None, *scale*=1.0, *r0*=5000.0, *type\_atmosphere*='ionosphere', *vis\_slices*=None, *\*\*kwargs*)

Grid a gaintable to a screen image

Screen axes are ['XX', 'YY', 'TIME', 'FREQ']

The phases are just averaged per grid cell, no phase unwrapping is performed.

#### Parameters

- **vis** –
- **gaintables** – input gaintables
- **screen** –
- **height** – Height (in m) of screen above telescope e.g. 3e5
- **r0** – r0 in meters
- **type\_atmosphere** – 'ionosphere' or 'troposphere'
- **scale** – Multiply the screen by this factor

#### Returns

gridded screen image, weights image

### calculate\_sf\_from\_screen

**calculate\_sf\_from\_screen**(*screen*)

Calculate structure function image from screen

Screen axes are ['XX', 'YY', 'TIME', 'FREQ']

#### Parameters

**screen** –

#### Returns

### plot\_gaintable\_on\_screen

**plot\_gaintable\_on\_screen**(*vis*, *gaintables*, *height*=300000.0, *gaintable\_slices*=None, *plotfile*=None)

Plot a gaintable on an ionospheric screen

Screen axes are ['XX', 'YY', 'TIME', 'FREQ']

#### Parameters

- **vis** –
- **gaintables** –
- **height** – Height (in m) of screen above telescope e.g. 3e5
- **scale** – Multiply the screen by this factor

#### Returns

gridded screen image, weights image

**rascil.processing\_components.simulation.noise Module**

Functions that add noise.

**Functions**

<code>calculate_noise_visibility</code> (bandwidth, ...)	Calculate noise rms per visibility [nchan, npol]
<code>addnoise_visibility</code> (vis[, t_sys, eta, seed])	Add noise to a visibility

**calculate\_noise\_visibility**

**calculate\_noise\_visibility**(*bandwidth, int\_time, diameter, t\_sys, eta*)

Calculate noise rms per visibility [nchan, npol]

**Parameters**

- **bandwidth** – (Hz)
- **int\_time** – Integration time (s)
- **diameter** – Diameter (m)
- **t\_sys** –  
(K)
- **eta** – Efficiency

**Returns**

Sigma [nrows, nchan]

**addnoise\_visibility**

**addnoise\_visibility**(*vis, t\_sys=None, eta=None, seed=None*)

Add noise to a visibility

TODO: Obtain sensitivity values from vis as a function of frequency

**Parameters**

- **vis** –
- **t\_sys** – System temperature
- **eta** – Efficiency

**Returns**

vis with noise added

## rascil.processing\_components.simulation.pointing Module

Functions for simulating pointing errors

### Functions

<code>simulate_gaintable_from_pointingtable(vis, ...)</code>	Create gaintables from a pointing table
<code>simulate_pointingtable_from_timeseries(pt[, ...])</code>	Create a pointing table with time series created from PSD.
<code>simulate_pointingtable(pt, pointing_error[, ...])</code>	Simulate a gain table

### simulate\_gaintable\_from\_pointingtable

**simulate\_gaintable\_from\_pointingtable**(*vis, sc, pt, vp, vis\_slices=None, scale=1.0, order=3, elevation\_limit=0.2617993877991494, jones\_type='G', \*\*kwargs*)

Create gaintables from a pointing table

Note that the column “nominal” is not used

#### Parameters

- **vis** –
- **sc** – Sky components for which pierce points are needed
- **pt** – Pointing table
- **vp** – Voltage pattern in AZELGEO frame
- **scale** – Multiply the screen by this factor
- **order** – order of spline (default is 3)
- **jones\_type** – Type of calibration matrix T or G or B

#### Returns

### simulate\_pointingtable\_from\_timeseries

**simulate\_pointingtable\_from\_timeseries**(*pt, type='wind', time\_series\_type='precision', pointing\_directory=None, reference\_pointing=False, seed=None*)

Create a pointing table with time series created from PSD.

#### Parameters

- **pt** – Pointing table to be filled
- **type** – Type of pointing: ‘tracking’ or ‘wind’
- **time\_series\_type** – Type of wind condition precision|standard|degraded
- **pointing\_directory** – Name of pointing file directory
- **reference\_pointing** – Use reference pointing?



**Returns****simulate\_pointingtable**

**simulate\_pointingtable**(*pt: PointingTable, pointing\_error, static\_pointing\_error=None, global\_pointing\_error=None, seed=None, \*\*kwargs*) → *PointingTable*

Simulate a gain table

**Parameters**

- **pointing\_error** – std of normal distribution (radians)
- **static\_pointing\_error** – std of normal distribution (radians)
- **global\_pointing\_error** – 2-vector of global pointing error (rad)
- **kwargs** –

**Returns**

*PointingTable*

**rascil.processing\_components.simulation.rfi Module**

Functions used to simulate RFI. Developed as part of SP-122/SIM.

The scenario is: \* There is a TV station at a remote location (e.g. Perth), emitting a broadband signal (7MHz) of known power (50kW). \* The emission from the TV station arrives at LOW stations with phase delay and attenuation. Neither of these are well known but they are probably static. \* The RFI enters LOW stations in a side-lobe of the main beam. Calculations by Fred Dulwich indicate that this provides attenuation of about 55 - 60dB for a source close to the horizon. \* The RFI enters each LOW station with fixed delay and zero fringe rate (assuming no e.g. ionospheric ducting) \* In tracking a source on the sky, the signal from one station is delayed and fringe-rotated to stop the fringes for one direction on the sky. \* The fringe rotation stops the fringe from a source at the phase tracking centre but phase rotates the RFI, which now becomes time-variable. \* The correlation data are time- and frequency-averaged over a timescale appropriate for the station field of view. This averaging de-correlates the RFI signal. \* We want to study the effects of this RFI on statistics of the images: on source and at the pole.

**Functions**

<i>calculate_averaged_correlation</i> (correlation, ...)	Average the correlation in time and frequency :param correlation: Correlation(ntimes, nant, nants, nchan] :param channel_width: Number of channels to average :param time_width: Number of integrations to average :return:
<i>simulate_rfi_block_prop</i> (bvis, ...[, ...])	Simulate RFI in a BlockVisility
<i>calculate_station_correlation_rfi</i> (...)	Form the correlation from the rfi at the station

### calculate\_averaged\_correlation

**calculate\_averaged\_correlation**(*correlation, time\_width, channel\_width*)

Average the correlation in time and frequency :param correlation: Correlation(ntimes, nant, nants, nchan) :param channel\_width: Number of channels to average :param time\_width: Number of integrations to average :return:

### simulate\_rfi\_block\_prop

**simulate\_rfi\_block\_prop**(*bvis, apparent\_emitter\_power, apparent\_emitter\_coordinates, rfi\_sources, rfi\_frequencies, low\_beam\_gain=None, apply\_primary\_beam=True*)

Simulate RFI in a BlockVisibility

#### Parameters

- **bvis** – input Visibility, to be updated with RFI
- **apparent\_emitter\_power** – RFI emitter power as received by an isotropic SKA antenna [nrfi\_sources x ntimes x nantennas x nchannels]
- **apparent\_emitter\_coordinates** – azimuth, elevation, distance information of RFI emitters [nrfi\_sources x ntimes x nantennas x 3]
- **rfi\_sources** – RFI source names or IDs
- **rfi\_frequencies** – frequency channels where there is RFI information length = nchannels
- **low\_beam\_gain** – beam gain data / information for Low. If provided, it is either a single value, or a numpy array with dimensions [nrfi\_sources x nstations x nchannels]; for Mid, use None
- **apply\_primary\_beam** – Apply the primary beam, not used for Low

#### Returns

Visibility

### calculate\_station\_correlation\_rfi

**calculate\_station\_correlation\_rfi**(*rfi\_at\_station, baselines*)

Form the correlation from the rfi at the station

#### Parameters

- **rfi\_at\_station** – [btimes, nchan, nants, nants]
- **baselines** – Visibility baselines object

#### Returns

correlation(ntimes, nbaselines, nchan) in Jy

**rascil.processing\_components.simulation.simulation\_helpers Module**

Functions that help with SKA simulations

**Functions**

<code>plot_visibility(vis_list[, colors, title, ...])</code>	Standard plot of visibility
<code>plot_visibility_pol(vis_list[, title, y, x, ...])</code>	Standard plot of visibility
<code>find_times_above_elevation_limit(...)</code>	Find all times for which a phasecentre is above the elevation limit
<code>plot_uvcoverage(vis_list[, ax, plot_file, title])</code>	Standard plot of uv coverage
<code>plot_uwcoverage(vis_list[, ax, plot_file, title])</code>	Standard plot of uw coverage
<code>plot_vwcoverage(vis_list[, ax, plot_file, title])</code>	Standard plot of vw coverage
<code>plot_configuration(config[, ax, plot_file, ...])</code>	Standard plot of uv coverage
<code>plot_azel(bvis_list[, plot_file])</code>	Standard plot of az el coverage
<code>plot_gaintable(gt_list[, title, value, ...])</code>	Standard plot of gain table
<code>plot_pointingtable(pt_list, plot_file, ...)</code>	Standard plot of pointing table
<code>find_pb_width_null(pdtype, frequency, **kwargs)</code>	Rough estimates of HWHM and null locations
<code>create_mid_simulation_components(...[, ...])</code>	Construct components for simulation
<code>plot_pa(bvis_list[, plot_file])</code>	Standard plot of parallactic angle coverage

**plot\_visibility**

**plot\_visibility**(*vis\_list*, *colors=None*, *title='Visibility'*, *y='amp'*, *x='uvdist'*, *plot\_file=None*, *chan=0*, *markersize=0.2*, *\*\*kwargs*)

Standard plot of visibility

**Parameters**

- **vis\_list** –
- **plot\_file** –
- **kwargs** –

**Returns****plot\_visibility\_pol**

**plot\_visibility\_pol**(*vis\_list*, *title='Visibility\_pol'*, *y='amp'*, *x='uvdist'*, *plot\_file=None*, *chan=0*, *\*\*kwargs*)

Standard plot of visibility

**Parameters**

- **vis\_list** –
- **plot\_file** –
- **kwargs** –

**Returns**

### `find_times_above_elevation_limit`

**find\_times\_above\_elevation\_limit**(*start\_times*, *end\_times*, *location*, *phasecentre*, *elevation\_limit*)

Find all times for which a phasecentre is above the elevation limit

#### Parameters

- **start\_times** –
- **end\_times** –
- **location** –
- **phasecentre** –
- **elevation\_limit** –

#### Returns

### `plot_uvcoverage`

**plot\_uvcoverage**(*vis\_list*, *ax=None*, *plot\_file=None*, *title='UV coverage'*, *\*\*kwargs*)

Standard plot of uv coverage

#### Parameters

- **vis\_list** –
- **plot\_file** –
- **kwargs** –

#### Returns

### `plot_uwcoverage`

**plot\_uwcoverage**(*vis\_list*, *ax=None*, *plot\_file=None*, *title='UW coverage'*, *\*\*kwargs*)

Standard plot of uw coverage

#### Parameters

- **vis\_list** –
- **plot\_file** –
- **kwargs** –

#### Returns

### plot\_vwcoverage

**plot\_vwcoverage**(*vis\_list*, *ax=None*, *plot\_file=None*, *title='VW coverage'*, *\*\*kwargs*)

Standard plot of vw coverage

#### Parameters

- **vis\_list** –
- **plot\_file** –
- **kwargs** –

#### Returns

### plot\_configuration

**plot\_configuration**(*config*, *ax=None*, *plot\_file=None*, *title='Configuration'*, *label=False*, *\*\*kwargs*)

Standard plot of uv coverage

#### Parameters

- **vis\_list** –
- **plot\_file** –
- **kwargs** –

#### Returns

### plot\_azel

**plot\_azel**(*bvis\_list*, *plot\_file=None*, *\*\*kwargs*)

Standard plot of az el coverage

#### Parameters

- **bvis\_list** –
- **plot\_file** –
- **kwargs** –

#### Returns

### plot\_gaintable

**plot\_gaintable**(*gt\_list*, *title=""*, *value='amp'*, *plot\_file=None*, *\*\*kwargs*)

Standard plot of gain table

#### Parameters

- **gt\_list** –
- **title** –
- **plot\_file** –
- **kwargs** –

**Returns****plot\_pointingtable****plot\_pointingtable**(*pt\_list*, *plot\_file*, *title*, *\*\*kwargs*)

Standard plot of pointing table

**Parameters**

- **pt\_list** –
- **plot\_file** –
- **title** –
- **kwargs** –

**Returns****find\_pb\_width\_null****find\_pb\_width\_null**(*pctype*, *frequency*, *\*\*kwargs*)

Rough estimates of HWHM and null locations

**Parameters**

- **pctype** –
- **frequency** –
- **kwargs** –

**Returns****create\_mid\_simulation\_components****create\_mid\_simulation\_components**(*phasecentre*, *frequency*, *flux\_limit*, *pbradius*, *pb\_npixel*, *pb\_cellsize*, *show=False*, *fov=10*, *polarisation\_frame=<ska\_sdp\_datamodels.science\_data\_model.polarisation\_model.PolarisationFrame object>*, *flux\_max=10.0*, *pb\_type='MID'*, *apply\_pb=True*)

Construct components for simulation

**Parameters**

- **context** – singlesource or null or s3sky
- **phasecentre** – Centre of components
- **frequency** – Frequency
- **pctype** – Type of primary beam
- **offset\_dir** – Offset in ra, dec degrees
- **flux\_limit** – Lower limit flux
- **pbradius** – Radius of components in radians
- **pb\_npixel** – Number of pixels in the primary beam model
- **pb\_cellsize** – Cellsize in primary beam model

- **fov** – FOV in degrees (used to select catalog)
- **flux\_max** – Maximum flux in model before application of primary beam
- **polarisation\_frame** –
- **apply\_pb** – Apply the primary beam to the output components
- **show** –

**Returns**

## plot\_pa

**plot\_pa**(*bvis\_list*, *plot\_file=None*, *\*\*kwargs*)

Standard plot of parallactic angle coverage

**Parameters**

- **bvis\_list** –
- **plot\_file** –
- **kwargs** –

**Returns**

## rascil.processing\_components.simulation.surface Module

Functions for dish surface modeling

## Functions

<a href="#"><i>simulate_gaintable_from_zernikes</i>(vis, sc, ...)</a>	Create gaintables for a set of zernikes
<a href="#"><i>simulate_gaintable_from_voltage_pattern</i>(vis, ...)</a>	Create gaintables from a list of components and voltage patterns

## simulate\_gaintable\_from\_zernikes

**simulate\_gaintable\_from\_zernikes**(*vis*, *sc*, *vp\_list*, *vp\_coeffs*, *vis\_slices=None*, *order=3*, *elevation\_limit=0.2617993877991494*, *jones\_type='B'*, *\*\*kwargs*)

Create gaintables for a set of zernikes

**Parameters**

- **vis** –
- **sc** – Sky components for which pierce points are needed
- **vp** – List of Voltage patterns in AZELGEO frame
- **vp\_coeffs** – Fractional contribution [nants, nvp]
- **order** – order of spline (default is 3)
- **jones\_type** – Type of calibration matrix T or G or B

## Returns

### simulate\_gaintable\_from\_voltage\_pattern

**simulate\_gaintable\_from\_voltage\_pattern**(vis, sc, vp, vis\_slices=None, order=3, elevation\_limit=0.2617993877991494, jones\_type='B', \*\*kwargs)

Create gaintables from a list of components and voltage patterns

#### Parameters

- **elevation\_limit** –
- **vis\_slices** –
- **vis** –
- **sc** – Sky components for which pierce points are needed
- **vp** – Voltage pattern in AZELGEO frame, can also be a list of voltage patterns, indexed alphabetically
- **order** – order of spline (default is 3)
- **jones\_type** – Type of calibration matrix T or G or B

#### Returns

### rascil.processing\_components.simulation.testing\_support Module

Functions that aid testing in various ways. A typical use would be:

```
lowcore = create_named_configuration('LOWBD2-CORE')
times = numpy.linspace(-3, +3, 13) * (numpy.pi / 12.0)

frequency = numpy.array([1e8])
channel_bandwidth = numpy.array([1e7])

# Define the component and give it some polarisation and spectral behaviour
f = numpy.array([100.0])
flux = numpy.array([f])

phasecentre = SkyCoord(ra=+15.0 * u.deg, dec=-35.0 * u.deg, frame='icrs', equinox='J2000
↪')
compabsdirection = SkyCoord(ra=17.0 * u.deg, dec=-36.5 * u.deg, frame='icrs', equinox=
↪'J2000')

comp = SkyComponent(flux=flux, frequency=frequency, direction=compabsdirection,
                    polarisation_frame=PolarisationFrame('stokesI'))
image = create_test_image(frequency=frequency, phasecentre=phasecentre,
                        cellsize=0.001,
                        polarisation_frame=PolarisationFrame(
↪'stokesI')

vis = create_visibility(lowcore, times=times, frequency=frequency,
                    channel_bandwidth=channel_bandwidth,
```

(continues on next page)



(continued from previous page)

```

phasecentre=phasecentre, weight=1,
polarisation_frame=PolarisationFrame('stokesI'),
integration_time=1.0)

```

## Functions

<code>create_low_test_image_from_gleam(npixel, ...)</code>	Create LOW test image from the GLEAM survey
<code>create_low_test_skycomponents_from_gleam([...])</code>	Create sky components from the GLEAM survey
<code>create_low_test_skymodel_from_gleam([...])</code>	Create LOW test skymodel from the GLEAM survey
<code>create_test_image([cellsize, frequency, ...])</code>	Create a useful test image
<code>create_test_image_from_s3(npixel, ...)</code>	Create MID test image from S3
<code>create_test_skycomponents_from_s3([...])</code>	Create test image from S3
<code>create_unittest_components(model, flux[, ...])</code>	
<code>create_unittest_model(vis, model_pol[, ...])</code>	
<code>ingest_unittest_visibility(config, ...[, ...])</code>	Make a standard visibility simulation
<code>insert_unittest_errors(vt[, seed, ...])</code>	Simulate gain errors and apply
<code>replicate_image(im[, polarisation_frame, ...])</code>	Make a new canonical shape Image, extended along third and fourth axes by replication.
<code>simulate_gaintable(gt[, phase_error, ...])</code>	Simulate a gain table

### create\_low\_test\_image\_from\_gleam

`create_low_test_image_from_gleam(npixel=512, polarisation_frame=<ska_sdp_datamodels.science_data_model.polarisation_model.PolarisationFrame object>, cellsize=1.5e-05, frequency=array([1.e+08]), channel_bandwidth=None, phasecentre=None, kind='cubic', applybeam=False, flux_limit=0.1, flux_max=inf, flux_min=-inf, radius=None, insert_method='Nearest') → Image`

Create LOW test image from the GLEAM survey

Stokes I is estimated from a cubic spline fit to the measured fluxes. The polarised flux is always zero.

See <http://www.mwatelescope.org/science/gleam-survey> The catalog is available from Vizier.

VIII/100 GaLactic and Extragalactic All-sky MWA survey (Hurley-Walker+, 2016)

GaLactic and Extragalactic All-sky Murchison Wide Field Array (GLEAM) survey. I: A low-frequency extragalactic catalogue. Hurley-Walker N., et al., Mon. Not. R. Astron. Soc., 464, 1146-1167 (2017), 2017MNRAS.464.1146H

#### Parameters

- **npixel** – Number of pixels
- **polarisation\_frame** – Polarisation frame (default PolarisationFrame(“stokesI”))
- **cellsize** – cellsize in radians
- **frequency** –
- **channel\_bandwidth** – Channel width (Hz)

- **phasecentre** – phasecentre (SkyCoord)
- **kind** – Kind of interpolation (see `scipy.interpolate.interp1d`) Default: linear
- **radius** – radius of search area in radians (Default is half-width of the diagonal)

**Returns**

Image

**create\_low\_test\_skycomponents\_from\_gleam**

```
create_low_test_skycomponents_from_gleam(flux_limit=0.1, polarisation_frame=<ska_sdp_datamodels.science_data_model.polarisation_model.PolarisationFrame object>, frequency=array([1.e+08]), kind='cubic', phasecentre=None, radius=1.0) → List[SkyComponent]
```

Create sky components from the GLEAM survey

Stokes I is estimated from a cubic spline fit to the measured fluxes. The polarised flux is always zero.

See <http://www.mwatelescope.org/science/gleam-survey> The catalog is available from Vizier.

VIII/100 GaLactic and Extragalactic All-sky MWA survey (Hurley-Walker+, 2016)

GaLactic and Extragalactic All-sky Murchison Wide Field Array (GLEAM) survey. I: A low-frequency extragalactic catalogue. Hurley-Walker N., et al., Mon. Not. R. Astron. Soc., 464, 1146-1167 (2017), 2017MNRAS.464.1146H

**Parameters**

- **flux\_limit** – Only write components brighter than this (Jy)
- **polarisation\_frame** – Polarisation frame (default PolarisationFrame(“stokesI”))
- **frequency** – Frequencies at which the flux will be estimated
- **kind** – Kind of interpolation (see `scipy.interpolate.interp1d`) Default: linear
- **phasecentre** – Desired phase centre (SkyCoord) default None implies all sources
- **radius** – Radius of sources selected around phasecentre (default 1.0 rad)

**Returns**

List of SkyComponents

**create\_low\_test\_skymodel\_from\_gleam**

```
create_low_test_skymodel_from_gleam(npixel=512, polarisation_frame=<ska_sdp_datamodels.science_data_model.polarisation_model.PolarisationFrame object>, cellsize=1.5e-05, frequency=array([1.e+08]), channel_bandwidth=array([1000000.]), phasecentre=None, kind='cubic', applybeam=True, flux_limit=0.1, flux_max=inf, flux_threshold=1.0, insert_method='Nearest', telescope='LOW', radius=None) → SkyModel
```

Create LOW test skymodel from the GLEAM survey

Stokes I is estimated from a cubic spline fit to the measured fluxes. The polarised flux is always zero.

See <http://www.mwatelescope.org/science/gleam-survey> The catalog is available from Vizier.

VIII/100 GaLactic and Extragalactic All-sky MWA survey (Hurley-Walker+, 2016)

GaLactic and Extragalactic All-sky Murchison Wide Field Array (GLEAM) survey. I: A low-frequency extragalactic catalogue. Hurley-Walker N., et al., Mon. Not. R. Astron. Soc., 464, 1146-1167 (2017), 2017MNRAS.464.1146H

#### Parameters

- **telescope** –
- **npixel** – Number of pixels
- **polarisation\_frame** – Polarisation frame (default PolarisationFrame(“stokesI”))
- **cellsize** – cellsize in radians
- **frequency** –
- **channel\_bandwidth** – Channel width (Hz)
- **phasecentre** – phasecentre (SkyCoord)
- **kind** – Kind of interpolation (see `scipy.interpolate.interp1d`) Default: cubic
- **applybeam** – Apply the primary beam?
- **flux\_limit** – Weakest component
- **flux\_max** – Maximum strength component to be included in components
- **flux\_threshold** – Split between components (brighter) and image (weaker)
- **insert\_method** – Nearest | PSWF | Lanczos
- **radius** – radius of search area in radians (Default is half-width of the axis)

#### Returns

SkyModel

### create\_test\_image

**create\_test\_image**(*cellsize=None, frequency=None, channel\_bandwidth=None, phasecentre=None, polarisation\_frame=None*) → Image

Create a useful test image

This is the test image M31 widely used in ALMA and other simulations. It is actually part of an Halpha region in M31.

#### Parameters

- **cellsize** –
- **frequency** – Frequency (array) in Hz
- **channel\_bandwidth** – Channel bandwidth (array) in Hz
- **phasecentre** – Phase centre of image (SkyCoord)
- **polarisation\_frame** – Polarisation frame

#### Returns

Image

## create\_test\_image\_from\_s3

```
create_test_image_from_s3(npixel=16384, polarisation_frame=<ska_sdp_datamodels.science_data_model.polarisation_model.PolarisationFrame
object>, cellsize=1.5e-05, frequency=array([1.e+08]),
channel_bandwidth=array([1000000.]), phasecentre=None, fov=20,
flux_limit=0.001) → Image
```

Create MID test image from S3

### The input catalog was generated using the following query::

Database: s3\_sex SQL: select \* from Galaxies where (pow(10, itot\_151)\*1000 > 1.0) and (right\_ascension between -5 and 5) and (declination between -5 and 5);

Number of rows returned: 29966

For frequencies < 610MHz, there are three tables to use:

data/models/S3_151MHz_10deg.csv, use fov=10
data/models/S3_151MHz_20deg.csv, use fov=20
data/models/S3_151MHz_40deg.csv, use fov=40

For frequencies > 610MHz, there are three tables:

data/models/S3_1400MHz_1mJy_10deg.csv,	use	flux_limit>=	1e-
3 data/models/S3_1400MHz_100uJy_10deg.csv,	use	flux_limit <	1e-
3 data/models/S3_1400MHz_10uJy_10deg.csv,	use	flux_limit <	1e-
4 data/models/S3_1400MHz_1mJy_18deg.csv,	use	flux_limit>=	1e-3
data/models/S3_1400MHz_100uJy_18deg.csv, use flux_limit < 1e-3			

The component spectral index is calculated from the 610MHz and 151MHz or 1400MHz and 610MHz, and then calculated for the specified frequencies.

If polarisation\_frame is not stokesI then the image will a polarised axis but the values will be zero.

### Parameters

- **npixel** – Number of pixels
- **polarisation\_frame** – Polarisation frame (default PolarisationFrame(“stokesI”))
- **cellsize** – cellsize in radians
- **frequency** –
- **channel\_bandwidth** – Channel width (Hz)
- **phasecentre** – phasecentre (SkyCoord)
- **fov** – fov 10 | 20 | 40
- **flux\_limit** – Minimum flux (Jy)

### Returns

Image

## create\_test\_skycomponents\_from\_s3

```
create_test_skycomponents_from_s3(polarisation_frame=<ska_sdp_datamodels.science_data_model.polarisation_model.PolarisationFrame>, frequency=array([1.e+08]),
channel_bandwidth=array([1000000.]), phasecentre=None, fov=20,
flux_limit=0.001, radius=None)
```

Create test image from S3

**The input catalog was generated using the following query::**

Database: s3\_sex SQL: select \* from Galaxies where (pow(10,itot\_151)\*1000 > 1.0) and (right\_ascension between -5 and 5) and (declination between -5 and 5);

Number of rows returned: 29966

For frequencies < 610MHz, there are three tables to use:

```
data/models/S3_151MHz_10deg.csv, use fov=10
data/models/S3_151MHz_20deg.csv, use fov=20
data/models/S3_151MHz_40deg.csv, use fov=40
```

For frequencies > 610MHz, there are three tables:

data/models/S3_1400MHz_1mJy_10deg.csv,	use	flux_limit>=	1e-
3 data/models/S3_1400MHz_100uJy_10deg.csv,	use	flux_limit <	1e-
3 data/models/S3_1400MHz_10uJy_10deg.csv,	use	flux_limit <	1e-
4 data/models/S3_1400MHz_1mJy_18deg.csv,	use	flux_limit>=	1e-3
data/models/S3_1400MHz_100uJy_18deg.csv,	use	flux_limit <	1e-3

The component spectral index is calculated from the 610MHz and 151MHz or 1400MHz and 610MHz, and then calculated for the specified frequencies.

If polarisation\_frame is not stokesI then the image will a polarised axis but the values will be zero.

### Parameters

- **polarisation\_frame** – Polarisation frame (default PolarisationFrame(“stokesI”))
- **frequency** –
- **channel\_bandwidth** – Channel width (Hz)
- **phasecentre** – phasecentre (SkyCoord)
- **fov** – fov 10 | 20 | 40
- **flux\_limit** – Minimum flux (Jy)
- **radius** – radius of search area in radians (Default is half-width of the axis)

### Returns

SkyComponents

### create\_unittest\_components

**create\_unittest\_components**(*model, flux, applypb=False, telescope='LOW', npixel=None, scale=1.0, single=False, symmetric=False, angular\_scale=1.0, offset=[0.0, 0.0]*)

### create\_unittest\_model

**create\_unittest\_model**(*vis, model\_pol, npixel=None, cellsize=None, nchan=1*)

### ingest\_unittest\_visibility

**ingest\_unittest\_visibility**(*config, frequency, channel\_bandwidth, times, vis\_pol, phasecentre, zerow=False, times\_are\_ha=True*)

Make a standard visibility simulation

#### Parameters

- **config** – Configuration
- **frequency** – Frequency (array in Hz)
- **channel\_bandwidth** – Channel bandwidth (array in Hz)
- **times** – Times (radians, utc or hour angle depending on times\_are\_ha)
- **vis\_pol** – Polarisation frame
- **phasecentre** – Phase centre (SkyCoord)
- **zerow** – Zero the w terms?
- **times\_are\_ha** – Are the times hourangles or utc (in radians)

#### Returns

### insert\_unittest\_errors

**insert\_unittest\_errors**(*vt, seed=1805550721, calibration\_context='TG', amp\_errors=None, phase\_errors=None*)

Simulate gain errors and apply

#### Parameters

- **vt** –
- **seed** – Random number seed, set to big integer repeat values from run to run
- **phase\_errors** – e.g. {'T': 1.0, 'G': 0.1, 'B': 0.01}
- **amp\_errors** – e.g. {'T': 0.0, 'G': 0.01, 'B': 0.01}

#### Returns

## replicate\_image

**replicate\_image**(*im*: ~ska\_sdp\_datamodels.image.image\_model.Image, *polarisation\_frame*=<ska\_sdp\_datamodels.science\_data\_model.polarisation\_model.PolarisationFrame object>, *frequency*=array([1.e+08])) → Image

Make a new canonical shape Image, extended along third and fourth axes by replication.

The order of the data is [chan, pol, dec, ra]

### Parameters

- **frequency** –
- **im** –
- **polarisation\_frame** – Polarisation\_frame

### Returns

Image

## simulate\_gaintable

**simulate\_gaintable**(*gt*: GainTable, *phase\_error*=0.1, *amplitude\_error*=0.0, *smooth\_channels*=1, *leakage*=0.0, *seed*=180550721, *\*\*kwargs*) → GainTable

Simulate a gain table

### Parameters

- **phase\_error** – std of normal distribution, zero mean
- **amplitude\_error** – std of log normal distribution
- **leakage** – std of cross hand leakage
- **smooth\_channels** – Use bspline over smooth\_channels
- **kwargs** –

### Returns

Gaintable

## Sky components

### rascil.processing\_components.skycomponent.plot\_skycomponent Module

Functions to manage plotting skycomponents in comparisons.

## Functions

<code>plot_skycomponents_positions(comps_test[, ...])</code>	Generate position scatter plot for two lists of skycomponents
<code>plot_skycomponents_position_distance(...[, ...])</code>	Generate position error plot vs distance for two lists of skycomponents
<code>plot_skycomponents_flux(comps_test, comps_ref)</code>	Generate flux scatter plot for two lists of skycomponents
<code>plot_skycomponents_flux_ratio(comps_test, ...)</code>	Generate flux ratio plot vs distance for two lists of skycomponents
<code>plot_skycomponents_flux_histogram(...[, ...])</code>	Generate flux ratio plot vs distance for two lists of skycomponents
<code>plot_skycomponents_position_quiver(...[, ...])</code>	Generate position error quiver diagram for two lists of skycomponents
<code>plot_gaussian_beam_position(comps_test, ...)</code>	Plot the major and minor size of beams for two lists of skycomponents :param comps_test: List of components to be tested :param comps_ref: List of reference components :param phasecentre: Centre of image in Sky-Cords :param image: Image to fit the skycomponents :param num: Number of the brightest sources to plot :param plot_file: Filename of the plot :param tol: Tolerance in rad
<code>plot_multifreq_spectral_index(comps_test, ...)</code>	Generate spectral index plot for two lists of multi-frequency skycomponents

### plot\_skycomponents\_positions

**plot\_skycomponents\_positions**(*comps\_test*, *comps\_ref*=None, *img\_size*=1.0, *plot\_file*=None, *tol*=1e-05, *plot\_error*=True, \*\*kwargs)

Generate position scatter plot for two lists of skycomponents

#### Parameters

- **comps\_test** – List of components to be tested
- **img\_size** – Cell size per pixel in the image to compare
- **comps\_ref** – List of reference components
- **plot\_file** – Filename of the plot
- **tol** – Tolerance in rad

:param plot\_error : If True, plot error, else just plot absolute values :return: [ra\_error, dec\_error]:

The error array for users to check



### plot\_skycomponents\_position\_distance

**plot\_skycomponents\_position\_distance**(*comps\_test*, *comps\_ref*, *phasecentre*, *img\_size*, *plot\_file=None*, *tol=1e-05*, *\*\*kwargs*)

Generate position error plot vs distance for two lists of skycomponents

#### Parameters

- **comps\_test** – List of components to be tested
- **comps\_ref** – List of reference components
- **plot\_file** – Filename of the plot
- **tol** – Tolerance in rad
- **phasecentre** – Centre of image in SkyCoords
- **img\_size** – Cell size per pixel in the image to compare

#### Returns

[*ra\_error*, *dec\_error*]: The error array for users to check

### plot\_skycomponents\_flux

**plot\_skycomponents\_flux**(*comps\_test*, *comps\_ref*, *plot\_file=None*, *tol=1e-05*, *refchan=None*, *\*\*kwargs*)

Generate flux scatter plot for two lists of skycomponents

#### Parameters

- **comps\_test** – List of components to be tested
- **comps\_ref** – List of reference components
- **plot\_file** – Filename of the plot
- **tol** – Tolerance in rad
- **refchan** – Reference channel for comparison, default is centre channel

#### Returns

[*flux\_in*, *flux\_out*]: The flux array for users to check

### plot\_skycomponents\_flux\_ratio

**plot\_skycomponents\_flux\_ratio**(*comps\_test*, *comps\_ref*, *phasecentre*, *plot\_file=None*, *tol=1e-05*, *refchan=None*, *max\_ratio=2*, *\*\*kwargs*)

Generate flux ratio plot vs distance for two lists of skycomponents

#### Parameters

- **comps\_test** – List of components to be tested
- **comps\_ref** – List of reference components
- **plot\_file** – Filename of the plot
- **tol** – Tolerance in rad
- **phasecentre** – Centre of image in SkyCoords
- **refchan** – Reference channel for comparison, default is centre channel

- **max\_ratio** – Maximum ratio to plot (default is 2.0)

**Returns**

[dist, flux\_ratio]: The flux array for users to check

### plot\_skycomponents\_flux\_histogram

**plot\_skycomponents\_flux\_histogram**(*comps\_test, comps\_ref, plot\_file=None, nbins=10, tol=1e-05, refchan=None, \*\*kwargs*)

Generate flux ratio plot vs distance for two lists of skycomponents

**Parameters**

- **comps\_test** – List of components to be tested
- **comps\_ref** – List of reference components
- **plot\_file** – Filename of the plot
- **tol** – Tolerance in rad
- **nbins** – Number of bins for the histogram
- **refchan** – Reference channel for comparison, default is centre channel

**Returns**

hist: The flux array for users to check

### plot\_skycomponents\_position\_quiver

**plot\_skycomponents\_position\_quiver**(*comps\_test, comps\_ref, phasecentre, num=100, plot\_file=None, tol=1e-05, \*\*kwargs*)

Generate position error quiver diagram for two lists of skycomponents

**Parameters**

- **comps\_test** – List of components to be tested
- **comps\_ref** – List of reference components
- **phasecentre** – Centre of image in SkyCoords
- **num** – Number of the brightest sources to plot
- **plot\_file** – Filename of the plot
- **tol** – Tolerance in rad

**Returns**

[ra\_error, dec\_error]: The error array for users to check

## plot\_gaussian\_beam\_position

**plot\_gaussian\_beam\_position**(*comps\_test, comps\_ref, phasecentre, image, num=100, plot\_file=None, tol=1e-05, \*\*kwargs*)

Plot the major and minor size of beams for two lists of skycomponents :param comps\_test: List of components to be tested :param comps\_ref: List of reference components :param phasecentre: Centre of image in SkyCoords :param image: Image to fit the skycomponents :param num: Number of the brightest sources to plot :param plot\_file: Filename of the plot :param tol: Tolerance in rad

### Returns

[bmaj, bmin]: The beam parameters for users to check

## plot\_multifreq\_spectral\_index

**plot\_multifreq\_spectral\_index**(*comps\_test, comps\_ref, phasecentre, plot\_file=None, tol=1e-05, flux\_limit=0.0, spec\_indx\_test=None, spec\_indx\_ref=None, plot\_diagnostics=False, \*\*kwargs*)

Generate spectral index plot for two lists of multi-frequency skycomponents

### Parameters

- **comps\_test** – List of components to be tested
- **comps\_ref** – List of reference components
- **phasecentre** – Centre of image in SkyCoords
- **plot\_file** – Filename of the plot
- **tol** – Tolerance in rad
- **flux\_limit** – Cutoff for plot (only components with central flux larger than this are plotted)
- **spec\_indx\_test** – Spectral index of comps\_test if provided (if None, fit from components)
- **spec\_indx\_ref** – Spectral index of comps\_ref if provided (if None, fit from components)
- **plot\_diagnostics** – Whether to plot diagnostics plot (flux in vs. spectral index out)

### Returns

[spec\_in, spec\_out]: The spectral index array for users to check

## Sky models

### rascil.processing\_components.skymodel.operations Module

Function to manage skymodels.

## Functions

<code>partition_skymodel_by_flux(sc, model[, ...])</code>	Partition skymodel according to flux
<code>show_skymodel(sms[, psf_width, cm, vmax, vmin])</code>	Show a list of SkyModels
<code>initialize_skymodel_voronoi(model, comps[, gt])</code>	Create a skymodel by Voronoi partitioning of the components, fill with components
<code>calculate_skymodel_equivalent_image(sm)</code>	Calculate an equivalent image for a skymodel
<code>update_skymodel_from_gaintables(sm, gt_list)</code>	Update a skymodel from a list of gaintables
<code>update_skymodel_from_image(sm, im[, damping])</code>	Update a skymodel for an image, applying damping factor
<code>expand_skymodel_by_skycomponents(sm, **kwargs)</code>	Expand a sky model so that all components and the image are in separate skymodels
<code>create_skymodel_from_skycomponents_gaintables(scs, gaintables)</code>	Create a list of sky model from lists of components and gaintables
<code>extract_skycomponents_from_skymodel(sm[, im])</code>	Extract the bright components from the image in a sky-model

### partition\_skymodel\_by\_flux

**partition\_skymodel\_by\_flux**(*sc, model, flux\_threshold=-inf*)

Partition skymodel according to flux

Bright skycomponents are put into a SkyModel as a list, and weak skycomponents are inserted into SkyModel as an image.

#### Parameters

- **sc** – List of skycomponents
- **model** – Model image
- **flux\_threshold** –

#### Returns

SkyModel

For example:

```
fluxes = numpy.linspace(0, 1.0, 11)
sc = [create_skycomponent(direction=phasecentre, flux=numpy.array([[f]]),
    ↪ frequency=frequency,
    ↪ fluxes]
    ↪ polarisation_frame=PolarisationFrame('stokesI')) for f in
    ↪ fluxes]

sm = partition_skymodel_by_flux(sc, model, flux_threshold=0.31)
assert len(sm.components) == 7, len(sm.components)
```

## show\_skymodel

**show\_skymodel**(*sms*, *psf\_width*=1.75, *cm*='Greys', *vmax*=None, *vmin*=None)

Show a list of SkyModels

### Parameters

- **sms** – List of SkyModels
- **psf\_width** – Width of PSF in pixels
- **cm** – matplotlib colormap
- **vmax** – Maximum in image display
- **vmin** – Minimum in image display

### Returns

## initialize\_skymodel\_voronoi

**initialize\_skymodel\_voronoi**(*model*, *comps*, *gt*=None)

Create a skymodel by Voronoi partitioning of the components, fill with components

### Parameters

- **model** – Model image
- **comps** – SkyComponents
- **gt** – Gaintable

### Returns

## calculate\_skymodel\_equivalent\_image

**calculate\_skymodel\_equivalent\_image**(*sm*)

Calculate an equivalent image for a skymodel

Uses the image from the first skymodel as the template for the image

### Parameters

**sm** – List of skymodels

### Returns

Image

## update\_skymodel\_from\_gaintables

**update\_skymodel\_from\_gaintables**(*sm*, *gt\_list*, *calibration\_context*='T', *damping*=0.5)

Update a skymodel from a list of gaintables

### Parameters

- **sm** – List of skymodels
- **gt\_list** – List of gain tables
- **calibration\_context** – Type of gaintable e.g. 'T', 'G'

**Returns**

List of skymodels

**update\_skymodel\_from\_image**

**update\_skymodel\_from\_image**(*sm, im, damping=0.5*)

Update a skymodel for an image, applying damping factor

**Parameters**

- **sm** – List of skymodels
- **im** – Image

**Returns**

List of SkyModels

**expand\_skymodel\_by\_skycomponents**

**expand\_skymodel\_by\_skycomponents**(*sm, \*\*kwargs*)

Expand a sky model so that all components and the image are in separate skymodels

The mask and gaintable are taken to apply for all new skymodels.

**Parameters**

**sm** – SkyModel

**Returns**

List of SkyModels

**create\_skymodel\_from\_skycomponents\_gaintables**

**create\_skymodel\_from\_skycomponents\_gaintables**(*components, gaintables, \*\*kwargs*)

Create a list of sky model from lists of components and gaintables

**Parameters**

**sm** – SkyModel

**Returns**

List of SkyModels

**extract\_skycomponents\_from\_skymodel**

**extract\_skycomponents\_from\_skymodel**(*sm, im=None, \*\*kwargs*)

Extract the bright components from the image in a skymodel

This produces one component per frequency channel

**Parameters**

- **sm** – skymodel
- **im** – image to be searched
- **kwargs** – Parameters for functions
- **component\_threshold** – (in kwargs) Threshold in Jy to be classified as a source

- **component\_method** – (in kwargs) Method to extract skycomponents: fit

**Returns**

Updated skymodel

**Utility****rascil.processing\_components.util.compass\_bearing Module****Functions**


---

<code>calculate_initial_compass_bearing(pointA, pointB)</code>	Calculates the bearing between two points.
--	--

---

**calculate\_initial\_compass\_bearing****calculate\_initial\_compass\_bearing**(*pointA*, *pointB*)

Calculates the bearing between two points.

**The formulae used is the following:**

$$= \text{atan2}(\sin(\text{long}).\cos(\text{lat2}), \cos(\text{lat1}).\sin(\text{lat2}) - \sin(\text{lat1}).\cos(\text{lat2}).\cos(\text{long}))$$
**Parameters**

- **pointA** – The tuple representing the latitude/longitude for the first point. Latitude and longitude must be in decimal degrees
- **pointB** – The tuple representing the latitude/longitude for the second point. Latitude and longitude must be in decimal degrees

**Returns**

The bearing in degrees

**Returns Type**

float

**rascil.processing\_components.util.installation\_checks Module**

Function to check the installation

**Functions**


---

<code>check_data_directory([verbose, fatal])</code>	Check the RASCIL data directory to see if it has been installed correctly
---	---

---

## check\_data\_directory

**check\_data\_directory**(*verbose=False, fatal=True*)

Check the RASCIL data directory to see if it has been installed correctly

## rascil.processing\_components.util.performance Module

Functions for monitoring performance

These functions can be used to write various configuration and performance information to JSON files for subsequent analysis. These are intended to be used by apps such as rascil-imager:

```
parser = cli_parser()
args = parser.parse_args()
performance_environment(args.performance_file, mode="w")
performance_store_dict(args.performance_file, "cli_args", vars(args), mode="a")
performance_store_dict(args.performance_file, "dask_profile", dask_info, mode="a")
performance_dask_configuration(args.performance_file, mode='a')
```

## Functions

<code>git_hash()</code>	Get the hash for this git repository.
<code>performance_store_dict(performance_file, key, s)</code>	Store dictionary in a file using json
<code>performance_qa_image(performance_file, key, im)</code>	Store image qa in a performance file
<code>performance_dask_configuration(...[, ...])</code>	Get selected Dask configuration info and write to performance file
<code>performance_read(performance_file)</code>	Read the performance file
<code>performance_environment(performance_file[, ...])</code>	Write the current processing environment to JSON file
<code>performance_read_memory_data(memory_file)</code>	Get the memusage data.
<code>performance_merge_memory(performance, mem)</code>	Merge memory data per function into performance data

## git\_hash

**git\_hash()**

Get the hash for this git repository.

Requires that the code tree was created using git

### Returns

string or “unknown”



## performance\_store\_dict

**performance\_store\_dict**(*performance\_file*, *key*, *s*, *indent*=2, *mode*='a')

Store dictionary in a file using json

### Parameters

- **performance\_file** – The (JSON) file to which the environment is to be written
- **key** – Key to use for the configuration info e.g. “restored”
- **s** – dictionary to be written
- **indent** – Number of characters indent in performance file
- **mode** – Writing mode: ‘w’ or ‘a’ for write and append

## performance\_qa\_image

**performance\_qa\_image**(*performance\_file*, *key*, *im*, *indent*=2, *mode*='a')

Store image qa in a performance file

### Parameters

- **performance\_file** – The (JSON) file to which the environment is to be written
- **key** – Key to use for the configuration info e.g. “restored”
- **im** – Image for which qa is to be calculated and written
- **indent** – Number of characters indent in performance file
- **mode** – Writing mode: ‘w’ or ‘a’ for write and append

## performance\_dask\_configuration

**performance\_dask\_configuration**(*performance\_file*, *rsexec*, *indent*=2, *mode*='a')

Get selected Dask configuration info and write to performance file

### Parameters

- **performance\_file** – The (JSON) file to which the environment is to be written
- **rsexec** – rsexecute passed in to avoid dependency
- **indent** – Number of characters indent in performance file
- **mode** – Writing mode: ‘w’ or ‘a’ for write and append

## performance\_read

**performance\_read**(*performance\_file*)

Read the performance file

### Parameters

**performance\_file** –

### Returns

Dictionary

## performance\_environment

**performance\_environment**(*performance\_file*, *indent*=2, *mode*='a')

Write the current processing environment to JSON file

### Parameters

- **performance\_file** – The (JSON) file to which the environment is to be written
- **indent** – Number of characters indent in performance file
- **mode** – Writing mode: 'w' or 'a' for write and append

## performance\_read\_memory\_data

**performance\_read\_memory\_data**(*memory\_file*)

Get the memusage data.

An example of the csv file: task\_key,min\_memory\_mb,max\_memory\_mb create\_visibility\_from\_ms-6d4df60d-244b-4a45-8dca-a7d96b676455,219.80859375,7651.37109375 getitem-ab6cb10a048f6d5efce69194feafa125,0,0 performance\_visibility-2dfe2b3a-e160-4724-a5e6-aed82bf0721c,0,0 create\_visibility\_from\_ms-724c98e9-279b-44ef-92d6-06e689b037a2,223.72265625,7642.13671875

The task\_key is split into task and key. The memory values are converted to GB.

### Parameters

**memory\_file** – Dictionary containing sequences of maximum and minimum memory for each function sampled

### Returns

## performance\_merge\_memory

**performance\_merge\_memory**(*performance*, *mem*)

Merge memory data per function into performance data

The memory usage information comes from the optional use of the dask-memusage scheduler plugin

### Parameters

- **performance** – Performance data dictionary
- **mem** – Memory data dictionary

### Returns

## Visibility

### rascil.processing\_components.visibility.base Module

Base simple visibility operations, placed here to avoid circular dependencies

## Functions

<code>create_visibility_from_ms(msname[, channum, ...])</code>	Minimal MS to Visibility converter
<code>create_visibility_from_uvfits(fitsname[, ...])</code>	Minimal UVFIT to Visibility converter
<code>export_visibility_to_ms(msname, vis_list[, ...])</code>	Minimal Visibility to MS converter
<code>extend_visibility_to_ms(msname, bvis)</code>	
<code>list_ms(msname[, ack])</code>	List sources and data descriptors in a MeasurementSet
<code>generate_baselines(nant)</code>	Generate mapping from antennas to baselines

### create\_visibility\_from\_ms

**create\_visibility\_from\_ms**(*msname*, *channum*=None, *start\_chan*=None, *end\_chan*=None, *ack*=False, *datacolumn*='DATA', *selected\_sources*=None, *selected\_dds*=None, *average\_channels*=False)

Minimal MS to Visibility converter

The MS format is much more general than the RASCIL Visibility so we cut many corners. This requires casacore to be installed. If not an exception ModuleNotFoundError is raised.

Creates a list of Visibility's, split by field and spectral window

Reading of a subset of channels is possible using either *start\_chan* and *end\_chan* or *channum*. Using *start\_chan* and *end\_chan* is preferred since it only reads the channels required. *Channum* is more flexible and can be used to read a random list of channels.

#### Parameters

- **msname** – File name of MS
- **channum** – range of channels e.g. range(17,32), default is None meaning all
- **start\_chan** – Starting channel to read
- **end\_chan** – End channel to read
- **ack** – Ask casacore to acknowledge each table operation
- **datacolumn** – MS data column to read DATA, CORRECTED\_DATA, or MODEL\_DATA
- **selected\_sources** – Sources to select
- **selected\_dds** – Data descriptors to select
- **average\_channels** – Average all channels read

#### Returns

List of Visibility

For example:

```
selected_sources = ['1302+5748', '1252+5634']
bvis_list = create_visibility_from_ms('../data/3C277.1_avg.ms', datacolumn=
    ↪ 'CORRECTED_DATA',
                                selected_sources=selected_sources)
sources = numpy.unique([bv.source for bv in bvis_list])
```

(continues on next page)

(continued from previous page)

```
print(sources)
['1252+5634' '1302+5748']
```

### `create_visibility_from_uvfits`

**create\_visibility\_from\_uvfits**(*fitsname*, *channum=None*, *ack=False*, *antnum=None*)

Minimal UVFIT to Visibility converter

The UVFITS format is much more general than the RASCIL Visibility so we cut many corners.

Creates a list of Visibility's, split by field and spectral window

#### Parameters

- **fitsname** – File name of UVFITS
- **channum** – range of channels e.g. range(17,32), default is None meaning all
- **antnum** – the number of antenna

#### Returns

### `export_visibility_to_ms`

**export\_visibility\_to\_ms**(*msname*, *vis\_list*, *source\_name=None*)

Minimal Visibility to MS converter

The MS format is much more general than the RASCIL Visibility so we cut many corners. This requires casacore to be installed. If not an exception ModuleNotFoundError is raised.

Write a list of Visibility's to a MS file, split by field and spectral window

#### Parameters

- **msname** – File name of MS
- **vis\_list** – list of Visibility
- **source\_name** – Source name to use
- **ack** – Ask casacore to acknowledge each table operation

#### Returns

### `extend_visibility_to_ms`

**extend\_visibility\_to\_ms**(*msname*, *bvis*)

## list\_ms

**list\_ms**(*msname*, *ack=False*)

List sources and data descriptors in a MeasurementSet

### Parameters

- **msname** – File name of MS
- **ack** – Ask casacore to acknowledge each table operation

### Returns

sources, data descriptors

### For example::

```
print(list_ms('3C277.1_avg.ms')) ([ '1302+5748', '0319+415', '1407+284', '1252+5634', '1331+305'], [0,
1, 2, 3])
```

## generate\_baselines

**generate\_baselines**(*nant*)

Generate mapping from antennas to baselines

Note that we need to include autocorrelations since some input measurement sets may contain autocorrelations

### Parameters

**nant** –

### Returns

## rascil.processing\_components.visibility.visibility\_fitting Module

Visibility fitting

## Functions

---

<i>fit_visibility</i> ( <i>vis</i> , <i>sc</i> [, <i>tol</i> , <i>niter</i> , ...])	Fit a single component to a visibility
---	--

---

## fit\_visibility

**fit\_visibility**(*vis*, *sc*, *tol=1e-06*, *niter=20*, *verbose=False*, *method='trust-exact'*, *\*\*kwargs*)

Fit a single component to a visibility

Uses the `scipy.optimize.minimize` function.

### Parameters

- **vis** – visibility
- **sc** – Initial component
- **tol** – Tolerance of fit
- **niter** – Number of iterations

- **verbose** –
- **method** – ‘CG’, ‘BFGS’, ‘Powell’, ‘trust-ncg’, ‘trust-exact’, ‘trust-krylov’: default ‘trust-exact’
- **kwargs** –

**Returns**

SkyComponent, convergence info as a dictionary

## 1.4.2 Workflows

Workflows are higher level functions that make use of the processing components, and processing library, operating on data models.

### rsexecute

rsexecute workflows can be used in two modes

- delayed using `Dask.delayed`
- serially executed immediately on definition,

Distribution is achieved by working on lists of data models, such as lists of BlockVisibilities.

For example:

```
from rascil.workflows import continuum_imaging_list_rsexecute_workflow, rsexecute
rsexecute.set_client(use_dask=True, threads_per_worker=1,
    memory_limit=32 * 1024 * 1024 * 1024, n_workers=8,
    local_dir=dask_dir, verbose=True)
continuum_imaging_list = continuum_imaging_list_rsexecute_workflow(vis_list,
    model_imagelist=model_list,
    context='wstack', vis_slices=51,
    scales=[0, 3, 10], algorithm='mmclean',
    nmoment=3, niter=1000,
    fractional_threshold=0.1, threshold=0.1,
    nmajor=5, gain=0.25,
    psf_support=64)

deconvolved_list, residual_list, restored_list = rsexecute.compute(continuum_imaging_
    ↪list,
    sync=True)
```

The call to `continuum_imaging_list_rsexecute_workflow` does not execute immediately just generates a `Dask.delayed` object that can be computed subsequently. The higher level functions such as `continuum_imaging_list_rsexecute_workflow` are built from lower level functions such as `invert_list_rsexecute_workflow`.

In this example, changing `use_dask` to `False` will cause the definitions to be executed immediately.

The rsexecute framework relies upon a singleton object called `rsexecute`. This is documented below as the class `_rsexecutebase`.

## rascil.workflows.rsexecute.calibration Package

Workflows for calibration

### Functions

---

<i>calibrate_list_rsexecute_workflow</i> (vis_list, ...)	Create a set of components for (optionally global) calibration of a list of visibilities
--	--

---

### calibrate\_list\_rsexecute\_workflow

**calibrate\_list\_rsexecute\_workflow**(vis\_list, model\_vislist, gt\_list=None, calibration\_context='TG', controls=None, global\_solution=True, \*\*kwargs)

Create a set of components for (optionally global) calibration of a list of visibilities

If global solution is true then visibilities are gathered to a single visibility data set which is then self-calibrated. The resulting gaintable is then effectively scattered out for application to each visibility set. If global solution is false then the solutions are performed locally.

#### Parameters

- **vis\_list** – list of visibilities (or graph)
- **model\_vislist** – list of model visibilities (or graph)
- **calibration\_context** – String giving terms to be calibrated e.g. ‘TGB’
- **controls** – Calibration controls dictionary
- **global\_solution** – Solve for global gains
- **kwargs** – Parameters for functions in components

#### Returns

list of calibrated vis, list of dictionaries of gaintables

## rascil.workflows.rsexecute.image Package

Workflows for operating on images

### Functions

---

<i>image_gather_channels_rsexecute</i> (image_list)	Gather a set of images in frequency, using a tree reduction or directly
<i>image_rsexecute_map_workflow</i> (im, imfunction)	Apply a function across an image: scattering to subimages, applying the function, and then gathering
<i>sum_images_rsexecute</i> (image_list[, split])	Sum a set of images, using a tree reduction

---

## image\_gather\_channels\_rsexecute

**image\_gather\_channels\_rsexecute**(*image\_list*, *split=0*)

Gather a set of images in frequency, using a tree reduction or directly

### Parameters

- **image\_list** – List of images
- **split** – Order of split i.e. 2 is binary, 0 is list

### Returns

graph for summed image

## image\_rsexecute\_map\_workflow

**image\_rsexecute\_map\_workflow**(*im*, *imfunction*, *facets=1*, *overlap=0*, *taper=None*, *\*\*kwargs*)

Apply a function across an image: scattering to subimages, applying the function, and then gathering

### Parameters

- **im** – Image to be processed
- **imfunction** – Function to be applied
- **facets** – See `image_scatter_facets`
- **overlap** – `image_scatter_facets`
- **taper** – `image_scatter_facets`
- **kwargs** – kwargs for `imfunction`

### Returns

graph for output image

For example:

```
rsexecute.set_client(use_dask=True)
model = create_test_image(frequency=frequency, phasecentre=phasecentre, cellsize=0.
↪001,
                           polarisation_frame=PolarisationFrame('stokesI'))
def imagerooter(im, **kwargs):
    im["pixels"].data = numpy.sqrt(numpy.abs(im["pixels"].data))
    return im
root_graph = image_rsexecute_map_workflow(model, imagerooter, facets=16)
root_image = rsexecute.compute(root_graph, sync=True)
```

## sum\_images\_rsexecute

**sum\_images\_rsexecute**(*image\_list*, *split=2*)

Sum a set of images, using a tree reduction

### Parameters

**image\_list** – List of images

### Returns

graph for summed image



**rascil.workflows.rsexecute.imaging Package****Functions**

<i>deconvolve_list_channel_rsexecute_workflow(...)</i>	Create a graph for deconvolution by channels, adding to the model
<i>deconvolve_list_rsexecute_workflow(...[, ...])</i>	Create a graph for deconvolution, adding to the model
<i>invert_list_rsexecute_workflow(vis_list, ...)</i>	Sum results from invert, iterating over the scattered image and vis_list
<i>predict_list_rsexecute_workflow(vis_list, ...)</i>	Predict, iterating over both the scattered vis_list and image
<i>residual_list_rsexecute_workflow(vis, ...[, ...])</i>	Create a graph to calculate (list or graph) of residual images
<i>restore_centre_rsexecute_workflow(...[, ...])</i>	Create a graph to calculate the restored image
<i>restore_list_rsexecute_workflow(...[, ...])</i>	Create a graph to calculate the restored image
<i>subtract_list_rsexecute_workflow(vis_list, ...)</i>	Initialise vis to zero
<i>sum_invert_results_rsexecute(image_list)</i>	Sum a set of invert results with appropriate weighting
<i>sum_predict_results_rsexecute(bvis_list[, split])</i>	Sum a set of predict results
<i>taper_list_rsexecute_workflow(vis_list, ...)</i>	Taper to desired size
<i>threshold_list_rsexecute(imagelist[, prefix])</i>	Find actual threshold for list of results
<i>weight_list_rsexecute_workflow(vis_list, ...)</i>	Weight the visibility data
<i>zero_list_rsexecute_workflow(vis_list[, copy])</i>	Creates a new vis_list and initialises all to zero

**deconvolve\_list\_channel\_rsexecute\_workflow**

**deconvolve\_list\_channel\_rsexecute\_workflow**(*dirty\_list, psf\_list, model\_imagelist, subimages, \*\*kwargs*)

Create a graph for deconvolution by channels, adding to the model

Does deconvolution channel by channel.

**Parameters**

- **dirty\_list** – list or graph of dirty images
- **psf\_list** – list or graph of psf images. The psfs must be the size of a facet
- **model\_imagelist** – list of graph of models
- **subimages** – Number of channels to split into
- **kwargs** – Parameters for functions in components

**Returns**

list of updated models (or graphs)

### deconvolve\_list\_rsexecute\_workflow

**deconvolve\_list\_rsexecute\_workflow**(dirty\_list, psf\_list, model\_imagelist, sensitivity\_list=None, prefix="", mask=None, \*\*kwargs)

Create a graph for deconvolution, adding to the model

note dirty\_list and psf\_list must have sumwt trimmed before calling this function

#### Parameters

- **dirty\_list** – list of dirty images (or graph)
- **psf\_list** – list of psfs (or graph)
- **model\_imagelist** – list of models (or graph)
- **prefix** – Informative prefix to log messages
- **mask** – Mask for deconvolution
- **kwargs** – Parameters for functions

#### Returns

graph for the deconvolution

For example:

```
dirty_imagelist = invert_list_rsexecute_workflow(vis_list, model_imagelist, context=
↳ '2d',
                                                dopsf=False, normalise=True)
psf_imagelist = invert_list_rsexecute_workflow(vis_list, model_imagelist, context=
↳ '2d',
                                                dopsf=True, normalise=True)
dirty_imagelist = rsexecute.persist(dirty_imagelist)
psf_imagelist = rsexecute.persist(psf_imagelist)
dec_imagelist = deconvolve_list_rsexecute_workflow(dirty_imagelist, psf_imagelist,
                                                    model_imagelist, niter=1000, fractional_threshold=0.01,
                                                    scales=[0, 3, 10], algorithm='mmclean', nmoment=3, nchan=freqwin,
                                                    threshold=0.1, gain=0.7)
dec_imagelist = rsexecute.persist(dec_imagelist)
```

### invert\_list\_rsexecute\_workflow

**invert\_list\_rsexecute\_workflow**(vis\_list, template\_model\_imagelist, context, dopsf=False, normalise=True, \*\*kwargs)

Sum results from invert, iterating over the scattered image and vis\_list

#### Parameters

- **vis\_list** – list of vis (or graph)
- **template\_model\_imagelist** – list of template models (or graph)
- **dopsf** – Make the PSF instead of the dirty image
- **normalise** – normalise by sumwt
- **context** – Imaging context
- **kwargs** – Parameters for functions in components

**Returns**

List of (image, sumwt) tuples, one per vis in vis\_list

For example:

```
model_list = [rsexecute.execute(create_image_from_visibility)
              (v, npixel=npixel, cellsize=cellsize, polarisation_frame=pol_frame)
              for v in vis_list]

model_list = rsexecute.persist(model_list)
dirty_list = invert_list_rsexecute_workflow(vis_list, template_model_
→imagelist=model_list, context='wstack',
                                vis_slices=51)
dirty_sumwt_list = rsexecute.compute(dirty_list, sync=True)
dirty, sumwt = dirty_sumwt_list[centre]
```

**predict\_list\_rsexecute\_workflow**

**predict\_list\_rsexecute\_workflow**(vis\_list, model\_imagelist, context, \*\*kwargs)

Predict, iterating over both the scattered vis\_list and image

The visibility and image are scattered, the visibility is predicted on each part, and then the parts are assembled.

**Parameters**

- **vis\_list** – list of vis (or graph)
- **model\_imagelist** – list of models (or graph)
- **context** – Type of processing e.g. 2d, ng
- **kwargs** – Parameters for functions in components

**Returns**

List of vis\_lists

For example:

```
dprepb_model = [rsexecute.execute(create_low_test_image_from_gleam)
                (npixel=npixel, frequency=[frequency[f]], channel_bandwidth=[channel_
→bandwidth[f]],
                cellsize=cellsize, phasecentre=phasecentre, polarisation_
→frame=PolarisationFrame("stokesI"),
                flux_limit=3.0, applybeam=True)
                for f, freq in enumerate(frequency)]

dprepb_model_list = rsexecute.persist(dprepb_model_list)
predicted_vis_list = predict_list_rsexecute_workflow(vis_list, model_
→imagelist=dprepb_model_list,
                context='wstack', vis_slices=51)
predicted_vis_list = rsexecute.compute(predicted_vis_list, sync=True)
```

### **residual\_list\_rsexecute\_workflow**

**residual\_list\_rsexecute\_workflow**(*vis*, *model\_imagelist*, *context*='2d', *\*\*kwargs*)

Create a graph to calculate (list or graph) of residual images

#### **Parameters**

- **vis** – List of vis (or graph)
- **model\_imagelist** – Model used to determine image parameters
- **context** – Imaging context e.g. '2d', 'ng'
- **kwargs** – Parameters for functions in components

#### **Returns**

list of (image, sumwt) tuples or graph

### **restore\_centre\_rsexecute\_workflow**

**restore\_centre\_rsexecute\_workflow**(*model\_imagelist*, *psf\_imagelist*, *residual\_imagelist*=None, *\*\*kwargs*)

Create a graph to calculate the restored image

This does the following: - Takes the centre frequency slice of the model - Integrates the residual across the band  
- Fits to the band-integrated PSF - Restores the model, clean\_beam, and residual

This will not give any information on the spectral behaviour, use `residual_list_rsexecute_workflow` for that purpose.

#### **Parameters**

- **model\_imagelist** – Model list (or graph)
- **psf\_imagelist** – PSF list (or graph)
- **residual\_imagelist** – Residual list (or graph)
- **kwargs** – Parameters for functions in components

#### **Returns**

list of restored images (or graphs)

### **restore\_list\_rsexecute\_workflow**

**restore\_list\_rsexecute\_workflow**(*model\_imagelist*, *psf\_imagelist*, *residual\_imagelist*=None, *restore\_facets*=1, *restore\_overlap*=8, *restore\_taper*='tukey', *clean\_beam*=None, *\*\*kwargs*)

Create a graph to calculate the restored image

#### **Parameters**

- **model\_imagelist** – Model list (or graph)
- **psf\_imagelist** – PSF list (or graph)
- **residual\_imagelist** – Residual list (or graph)
- **kwargs** – Parameters for functions in components
- **restore\_facets** – Number of facets used per axis (used to distribute)

- **restore\_overlap** – Overlap in pixels (0 is best)
- **restore\_taper** – Type of taper between facets

**Returns**

list of restored images (or graph)

**subtract\_list\_rsexecute\_workflow**

**subtract\_list\_rsexecute\_workflow**(*vis\_list, model\_vislist*)

Initialise vis to zero

**Parameters**

- **vis\_list** – List of vis (or graph)
- **model\_vislist** – Model to be subtracted (or graph)

**Returns**

List of vis or graph

**sum\_invert\_results\_rsexecute**

**sum\_invert\_results\_rsexecute**(*image\_list*)

Sum a set of invert results with appropriate weighting

Note that in the case of a single element of *image\_list* a copy is made

**Parameters**

**image\_list** – List of (image, sum weights) tuples

**Returns**

image, sum of weights

**sum\_predict\_results\_rsexecute**

**sum\_predict\_results\_rsexecute**(*bvis\_list, split=2*)

Sum a set of predict results

**Parameters**

- **bvis\_list** – List of (image, sum weights) tuples
- **split** – Split into

**Returns**

BlockVis

### **taper\_list\_rsexecute\_workflow**

**taper\_list\_rsexecute\_workflow**(*vis\_list*, *size\_required*)

Taper to desired size

#### **Parameters**

- **vis\_list** – List of vis (or graph)
- **size\_required** – Size in radians

#### **Returns**

List of vis (or graph)

### **threshold\_list\_rsexecute**

**threshold\_list\_rsexecute**(*imagelist*, *prefix*="", *\*\*kwargs*)

Find actual threshold for list of results

#### **Parameters**

- **prefix** – Prefix in log messages
- **imagelist** – List of images

#### **Returns**

### **weight\_list\_rsexecute\_workflow**

**weight\_list\_rsexecute\_workflow**(*vis\_list*, *model\_imagelist*, *weighting*='uniform', *robustness*=0.0, *\*\*kwargs*)

Weight the visibility data

This is done collectively so the weights are summed over all *vis\_lists* and then corrected

#### **Parameters**

- **vis\_list** –
- **model\_imagelist** – Model required to determine weighting parameters
- **weighting** – Type of weighting
- **kwargs** – Parameters for functions in graphs

#### **Returns**

List of *vis\_graphs*

For example:

```
vis_list = weight_list_rsexecute_workflow(vis_list, model_list, weighting='uniform')
```

## zero\_list\_rsexecute\_workflow

**zero\_list\_rsexecute\_workflow**(*vis\_list*, *copy=True*)

Creates a new *vis\_list* and initialises all to zero

### Parameters

- **vis\_list** – List of vis (or graph)
- **copy** – Make a new copy?

### Returns

List of vis (or graph)

## rascil.workflows.rsexecute.pipelines Package

### Functions

---

<code>continuum_imaging_skymodel_list_rsexecute_workflow(...)</code>	Create graph for the continuum imaging pipeline.
--	--

---

<code>ical_skymodel_list_rsexecute_workflow(...)</code>	Create graph for ICAL pipeline using SkyModel
---	---

---

<code>spectral_line_imaging_skymodel_list_rsexecute_workflow(...)</code>	Create graph for spectral line imaging pipeline
--	---

---

## continuum\_imaging\_skymodel\_list\_rsexecute\_workflow

**continuum\_imaging\_skymodel\_list\_rsexecute\_workflow**(*vis\_list*, *model\_imagelist*, *context*, *skymodel\_list=None*, *\*\*kwargs*)

Create graph for the continuum imaging pipeline.

Same as ICAL but with no selfcal.

### Parameters

- **vis\_list** – List of vis (or graph)
- **model\_imagelist** – List of models (or graph)
- **skymodel\_list** – list of SkyModels
- **context** – Imaging context
- **skymodel\_list** – list of SkyModels
- **kwargs** – Parameters for functions in components

### Returns

### **ical\_skymodel\_list\_rsexecute\_workflow**

**ical\_skymodel\_list\_rsexecute\_workflow**(*vis\_list*, *model\_imagelist*, *context*, *skymodel\_list*=None, *calibration\_context*='TG', *controls*=None, *do\_selfcal*=True, *pipeline\_name*='ical', \*\**kwargs*)

Create graph for ICAL pipeline using SkyModel

#### **Parameters**

- **vis\_list** – List of vis (or graph)
- **model\_imagelist** – list of models (or graph)
- **skymodel\_list** – list of SkyModels
- **context** – imaging context e.g. '2d'
- **calibration\_context** – Sequence of calibration steps e.g. TGB
- **do\_selfcal** – Do the selfcalibration?
- **perform\_flagging** – Run flagging strategy
- **kwargs** – Parameters for functions in components

#### **Returns**

### **spectral\_line\_imaging\_skymodel\_list\_rsexecute\_workflow**

**spectral\_line\_imaging\_skymodel\_list\_rsexecute\_workflow**(*vis\_list*, *model\_imagelist*, *context*, *continuum\_model\_imagelist*=None, \*\**kwargs*)

Create graph for spectral line imaging pipeline

Uses the continuum imaging rsexecute pipeline after subtraction of a continuum model

#### **Parameters**

- **vis\_list** – List of vis (or graph)
- **model\_imagelist** – List of Spectral line model (or graph)
- **continuum\_model\_imagelist** – Continuum model list (or graph)
- **context** – Imaging context e.g. ng or 2d
- **kwargs** – Parameters for functions in components

#### **Returns**

list of (deconvolved model, residual, restored) or graph



## rascil.workflows.rsexecute.simulation Package

### Functions

<code>corrupt_list_rsexecute_workflow(vis_list[, ...])</code>	Create a graph to apply gain errors to a vis_list
<code>create_atmospheric_errors_gaintable_rsexecute_workflow</code>	Create gaintable for atmospheric errors
<code>create_heterogeneous_gaintable_rsexecute_workflow</code>	Create gaintable for polarisation effects
<code>create_pointing_errors_gaintable_rsexecute_workflow</code>	Create gaintable for pointing errors
<code>create_polarisation_gaintable_rsexecute_workflow</code>	Create gaintable for polarisation effects
<code>create_standard_low_simulation_rsexecute_workflow</code>	Create the standard LOW simulation
<code>create_standard_mid_simulation_rsexecute_workflow</code>	Create the standard MID simulation
<code>create_surface_errors_gaintable_rsexecute_workflow</code>	Create gaintable for surface errors :param band: B1, B2 or Ku :param sub_bvis_list: List of vis (or graph) :param sub_components: List of components (or graph) :param vp_directory: Location of voltage patterns :param elevation_sampling: Sampling in elevation (degrees) :return: (list of error-free gaintables, list of error gaintables) or graph
<code>create_voltage_pattern_gaintable_rsexecute_workflow</code>	Create gaintable for nominal voltage pattern
<code>simulate_list_rsexecute_workflow([config, ...])</code>	A component to simulate an observation

### corrupt\_list\_rsexecute\_workflow

**corrupt\_list\_rsexecute\_workflow**(vis\_list, gt\_list=None, jones\_type='T', \*\*kwargs)

Create a graph to apply gain errors to a vis\_list

#### Parameters

- **vis\_list** – List of vis (or graph)
- **gt\_list** – Optional gain table graph
- **jones\_type** – Type of calibration matrix T or G or B
- **kwargs** –

#### Returns

list of vis (or graph)

### create\_atmospheric\_errors\_gaintable\_rsexecute\_workflow

**create\_atmospheric\_errors\_gaintable\_rsexecute\_workflow**(sub\_bvis\_list, sub\_components, r0=5000.0, screen=None, height=300000.0, type\_atmosphere='iono', reference\_component=None, jones\_type='B', \*\*kwargs)

Create gaintable for atmospheric errors

#### Parameters

- **sub\_bvis\_list** – List of vis (or graph)
- **sub\_components** – List of components (or graph)

- **r0** – r0 in m
- **screen** –
- **height** – Height (in m) of screen above telescope e.g. 3e5
- **type\_atmosphere** – ‘ionosphere’ or ‘troposphere’
- **jones\_type** – Type of calibration matrix T or G or B

**Returns**

(list of error-free gaintables, list of error gaintables) or graph

**create\_heterogeneous\_gaintable\_rsexecute\_workflow**

**create\_heterogeneous\_gaintable\_rsexecute\_workflow**(*band, sub\_bvis\_list, sub\_components, get\_vp, default\_vp='MID'*)

Create gaintable for polarisation effects

Compare with nominal and actual voltage patterns

**Parameters**

- **band** – B1, B2 or Ku
- **sub\_bvis\_list** – List of vis (or graph)
- **sub\_components** – List of components (or graph)

**Returns**

(list of error-free gaintables, list of error gaintables) or graph

**create\_pointing\_errors\_gaintable\_rsexecute\_workflow**

**create\_pointing\_errors\_gaintable\_rsexecute\_workflow**(*sub\_bvis\_list, sub\_components, sub\_vp\_list, pointing\_error=0.0, static\_pointing\_error=None, global\_pointing\_error=None, time\_series="", time\_series\_type="", seed=None, pointing\_directory=None*)

Create gaintable for pointing errors

**Parameters**

- **sub\_bvis\_list** – List of vis (or graph)
- **sub\_components** – List of components (or graph)
- **sub\_vp\_list** – List of model voltage patterns (or graph)
- **pointing\_error** – rms pointing error
- **static\_pointing\_error** – static pointing error
- **global\_pointing\_error** – global pointing error
- **time\_series** – Time series PSD file
- **time\_series\_type** – Type of time series ‘wind’|’
- **seed** – Random number seed

- **pointing\_directory** – Location of pointing files

**Returns**

(list of error-free gaintables, list of error gaintables) or graph

**create\_polarisation\_gaintable\_rsexecute\_workflow**

**create\_polarisation\_gaintable\_rsexecute\_workflow**(*band, sub\_bvis\_list, sub\_components, get\_vp, normalise=True*)

Create gaintable for polarisation effects

Compare with nominal and actual voltage patterns

**Parameters**

- **band** – B1, B2 or Ku
- **sub\_bvis\_list** – List of vis (or graph)
- **sub\_components** – List of components (or graph)
- **normalise** – Normalise peak of each receptor

**Returns**

(list of error-free gaintables, list of error gaintables) or graph

**create\_standard\_low\_simulation\_rsexecute\_workflow**

**create\_standard\_low\_simulation\_rsexecute\_workflow**(*band, rmax, phasecentre, time\_range, time\_chunk, integration\_time, polarisation\_frame=None, zerow=False*)

Create the standard LOW simulation

**Parameters**

- **band** – B
- **rmax** – Maximum distance from array centre
- **phasecentre** – Phase centre (SkyCoord)
- **time\_range** – Hour angle (in hours)
- **time\_chunk** – Chunking of time in seconds
- **integration\_time** –
- **polarisation\_frame** – Desired polarisation frame
- **zerow** – Set w to zero (False)

**Returns**

### `create_standard_mid_simulation_rsexecute_workflow`

`create_standard_mid_simulation_rsexecute_workflow`(*band, rmax, phasecentre, time\_range, time\_chunk, integration\_time, polarisation\_frame=None, zerow=False, configuration='MID'*)

Create the standard MID simulation

#### Parameters

- **band** – B1, B2, or Ku
- **rmax** – Maximum distance from array centre
- **phasecentre** – Phase centre (SkyCoord)
- **time\_range** – Hour angle (in hours)
- **time\_chunk** – Chunking of time in seconds
- **integration\_time** –
- **polarisation\_frame** – Desired polarisation frame
- **zerow** – Set w to zero (False)

#### Returns

### `create_surface_errors_gaintable_rsexecute_workflow`

`create_surface_errors_gaintable_rsexecute_workflow`(*band, sub\_bvis\_list, sub\_components, vp\_directory, elevation\_sampling=5.0*)

Create gaintable for surface errors :param band: B1, B2 or Ku :param sub\_bvis\_list: List of vis (or graph) :param sub\_components: List of components (or graph) :param vp\_directory: Location of voltage patterns :param elevation\_sampling: Sampling in elevation (degrees) :return: (list of error-free gaintables, list of error gaintables) or graph

### `create_voltage_pattern_gaintable_rsexecute_workflow`

`create_voltage_pattern_gaintable_rsexecute_workflow`(*band, sub\_bvis\_list, sub\_components, get\_vp, normalise=True*)

Create gaintable for nominal voltage pattern

Compare with nominal and actual voltage patterns

#### Parameters

- **band** – B1, B2 or Ku
- **sub\_bvis\_list** – List of vis (or graph)
- **sub\_components** – List of components (or graph)
- **normalise** – Normalise peak of each receptor

#### Returns

(list of error-free gaintables, list of error gaintables) or graph

## simulate\_list\_rsexecute\_workflow

**simulate\_list\_rsexecute\_workflow**(*config*='LOWBD2', *phasecentre*=<SkyCoord (ICRS): (ra, dec) in deg (15., -60.)>, *frequency*=None, *channel\_bandwidth*=None, *times*=None, *polarisation\_frame*=<ska\_sdp\_datamodels.science\_data\_model.polarisation\_model.PolarisationFrame object>, *order*='frequency', *format*='vis', *rmax*=1000.0, *zerow*=False, *skip*=1)

A component to simulate an observation

The simulation step can generate a single Visibility or a list of Visibility's. The parameter keyword determines the way that the list is constructed. If *order*='frequency' then len(*frequency*) Visibility's with all times are created. If *order*='time' then len(*times*) Visibility's with all frequencies are created. If *order* = 'both' then len(*times*) \* len(*times*) Visibility's are created each with a single time and frequency. If *order* = None then all data are created in one Visibility.

The output format can be either 'vis' (for calibration) or 'vis' (for imaging)

### Parameters

- **config** – Name of configuration: def LOWBDS-CORE
- **phasecentre** – Phase centre def: SkyCoord(ra=+15.0 \* u.deg, dec=-60.0 \* u.deg, frame='icrs', equinox='J2000')
- **frequency** – def [1e8]
- **channel\_bandwidth** – def [1e6]
- **times** – Observing times in radians: def [0.0]
- **polarisation\_frame** – def PolarisationFrame("stokesI")
- **order** – 'time' or 'frequency' or 'both' or None: def 'frequency'
- **format** – 'vis' or 'vis': def 'vis'
- **zerow** – Set w to zero
- **skip** – Number of dishes/stations to skip

### Returns

graph of vis\_list with different frequencies in different elements

## rascil.workflows.rsexecute.skymodel Package

### Functions

<code>deconvolve_skymodel_list_rsexecute_workflow(...)</code>	Deconvolve using a skymodel
<code>invert_skymodel_list_rsexecute_workflow(...)</code>	Calibrate and invert from a skymodel, iterating over the skymodel
<code>predict_skymodel_list_rsexecute_workflow(...)</code>	Predict from a list of skymodels
<code>restore_centre_skymodel_list_rsexecute_workflow(...)</code>	Create a graph to calculate the restored skymodel at the centre channel
<code>restore_skymodel_list_rsexecute_workflow(...)</code>	Create a graph to calculate the restored image

### deconvolve\_skymodel\_list\_rsexecute\_workflow

**deconvolve\_skymodel\_list\_rsexecute\_workflow**(*dirty\_image\_list*, *psf\_list*, *skymodel\_list*, *prefix=""*,  
*fit\_skymodel=False*, *\*\*kwargs*)

Deconvolve using a skymodel

This will either fit for the brightest components and add those to the skymodel components or use (optionally faceted) CLEAN based deconvolution

#### Parameters

- **dirty\_image\_list** – List of dirty images (or graphs)
- **psf\_list** – List of corresponding psf images (or graphs)
- **skymodel\_list** – list of skymodels (or graph)
- **prefix** – Informational prefix for logging messages
- **fit\_skymodel** – Fit the skymodel?
- **kwargs** –

#### Returns

list of skymodels (or graph)

### invert\_skymodel\_list\_rsexecute\_workflow

**invert\_skymodel\_list\_rsexecute\_workflow**(*vis\_list*, *skymodel\_list*, *\*\*kwargs*)

Calibrate and invert from a skymodel, iterating over the skymodel

The function `get_pb` should have the signature:

`get_pb(Visibility, Image)`

and should return the primary beam for the visibility.

The return is a graph for a set of tuples of (dirty, sensitivity image)

#### Parameters

- **vis\_list** – List of Visibility data models
- **skymodel\_list** – skymodel list
- **kwargs** – Parameters for functions in components

#### Returns

List of (image, weight) tuples)

### predict\_skymodel\_list\_rsexecute\_workflow

**predict\_skymodel\_list\_rsexecute\_workflow**(*obsvis*, *skymodel\_list*, *\*\*kwargs*)

Predict from a list of skymodels

If *obsvis* is a list then we pair *obsvis* element and *skymodel\_list* element and predict. If *obsvis* is `Visibility` then we calculate `Visibility` for each skymodel

#### Parameters

- **obsvis** – Observed Block Visibility or list or graph

- **skymodel\_list** – skymodel list
- **kwargs** – Parameters for functions in components

**Returns**

List of vis\_lists

**restore\_centre\_skymodel\_list\_rsexecute\_workflow**

```
restore_centre_skymodel_list_rsexecute_workflow(skymodel_list, psf_imagelist,
                                                residual_imagelist=None, clean_beam=None,
                                                **kwargs)
```

Create a graph to calculate the restored skymodel at the centre channel

**Parameters**

- **skymodel\_list** – Skymodel list (or graph)
- **psf\_imagelist** – PSF list (or graph)
- **residual\_imagelist** – Residual list (or graph)
- **kwargs** – Parameters for functions in components
- **clean\_beam** – Clean beam e.g. {"bmaj":0.1, "bmin":0.05, "bpa":-60.0}. Units are deg, deg, deg

**Returns**

list of restored images (or graph)

**restore\_skymodel\_list\_rsexecute\_workflow**

```
restore_skymodel_list_rsexecute_workflow(skymodel_list, psf_imagelist, residual_imagelist=None,
                                           restore_facets=1, restore_overlap=8, restore_taper='tukey',
                                           clean_beam=None, **kwargs)
```

Create a graph to calculate the restored image

**Parameters**

- **model\_imagelist** – Model list (or graph)
- **psf\_imagelist** – PSF list (or graph)
- **residual\_imagelist** – Residual list (or graph)
- **clean\_beam** – Clean beam e.g. {"bmaj":0.1, "bmin":0.05, "bpa":-60.0}. Units are deg, deg, deg
- **kwargs** – Parameters for functions in components
- **restore\_facets** – Number of facets used per axis (used to distribute)
- **restore\_overlap** – Overlap in pixels (0 is best)
- **restore\_taper** – Type of taper between facets

**Returns**

list of restored images (or graph)

## rascil.workflows.rsexecute.execution\_support Package

### Functions

---

<code>get_dask_client([timeout, n_workers, ...])</code>	Get a Dask.distributed Client to be used in rsexecute
---	---

---

#### get\_dask\_client

**get\_dask\_client**(*timeout=30, n\_workers=None, threads\_per\_worker=None, processes=True, create\_cluster=False, memory\_limit=None, local\_dir='.', with\_file=False, scheduler\_file='./scheduler.json', dashboard\_address=':8787'*)

Get a Dask.distributed Client to be used in rsexecute

The default operation of rsexecute.set\_client is to create a set of workers on one node. Hence if you want to use a cluster it is necessary to use get\_dask\_client.

The environment variable RASCIL\_DASK\_SCHEDULER is interpreted as pointing to the Dask distributed scheduler. and a client using that scheduler is returned. Otherwise a client for a LocalCluster is created.

The environment variable RASCIL\_DASK\_SCHEDULER\_FILE is interpreted as pointing to the Dask scheduler file and a client using that scheduler is returned. If RASCIL\_DASK\_SCHEDULER\_FILE is set, with\_file option is set to true and scheduler\_file name is overridden with the RASCIL\_DASK\_SCHEDULER\_FILE

#### Parameters

- **timeout** – Time out for creation (30s)
- **n\_workers** – Number of workers (cores available)
- **threads\_per\_worker** – 1
- **processes** – Use processes instead of threads (True)
- **create\_cluster** – Create a LocalCluster (True)
- **memory\_limit** – Memory limit per worker (bytes e.g. 8e9) (None)
- **scheduler\_file** – Scheduler file for Dask ('./scheduler.json')
- **dashboard\_address** – Port used for diagnostics (':8787')

#### Returns

Dask client

### Classes

The rsexecute framework relies upon a singleton object called rsexecute. This is documented below as the class \_rsexecutibase. Note that by design it is not possible to create more than one \_rsexecutibase object.

**class \_rsexecutibase**(*use\_dask=True, use\_dlg=False, verbose=False, optimize=True*)

Initialise rsexecute framework

A singleton of this class is created and is available globally as rsexecute. Hence it is not necessary to declare an instance of \_rsexecutibase.

For example:



```

from rascil.workflows import continuum_imaging_list_rsexecute_workflow, rsexecute
rsexecute.set_client(use_dask=True,
    memory_limit=32 * 1024 * 1024 * 1024, n_workers=8,
    local_dir=dask_dir, verbose=True)
continuum_imaging_list = continuum_imaging_list_rsexecute_workflow(vis_list,
    model_imagelist=model_list,
    context='wstack', vis_slices=51,
    scales=[0, 3, 10], algorithm='mmclean',
    nmoment=3, niter=1000,
    fractional_threshold=0.1, threshold=0.1,
    nmajor=5, gain=0.25,
    psf_support=64)

deconvolved_list, residual_list, restored_list = rsexecute.compute(continuum_
    →imaging_list,
    sync=True)

```

#### Parameters

- **use\_dask** – Use dask (True)
- **use\_dlg** – Use daluige (False)
- **verbose** – Be verbose in printing messages
- **optimize** – Optimize if using dask (True)

**execute**(*func*, \**args*, \*\**kwargs*)

Wrap for immediate or deferred execution

Passes through if dask is not being used

#### Parameters

- **args** –
- **kwargs** –

#### Returns

delayed func or func

**type()**

Get the name of the execution system

#### Returns

**set\_client**(*client=None*, *use\_dask=True*, *use\_dlg=False*, *verbose=False*, *optim=True*, \*\**kwargs*)

Set the Dask/DALiuGE client to be used

If you want to customise the Client or use an externally defined Scheduler use `get_dask_client` and pass it in.

#### Parameters

- **use\_dask** – Use Dask?
- **client** – If None and `use_dask` is True, a client will be created otherwise the client is None
- **use\_dlg** – Use Daliuge to execute graphs?
- **verbose** – Be verbose in output

- **optim** – Use `dask.optimize` via `rsexecute.optimize` function.

#### Returns

**compute**(*value*, *sync=False*)

Get the actual value

If not using dask then this returns the value directly since it already is computed. If using dask and `sync=True` then this waits and returns the actual value. If using dask and `sync=False` then this returns a future, on which you will need to call `.result()`

#### Parameters

- **value** –
- **sync** – Return synchronously? (False)

#### Returns

**persist**(*graph*, *\*\*kwargs*)

Persist graph data on workers

The graphs are placed on the workers but not computed

No-op if not using `_dask`

#### Parameters

**graph** –

#### Returns

**scatter**(*graph*, *\*\*kwargs*)

Scatter graph data to workers

The data are placed on the workers

No-op if not using `_dask` :param graph: :return:

**gather**(*graph*)

Gather graph from workers

The data are gathered from the workers

No-op if not using `_dask`

#### Parameters

**graph** –

#### Returns

**run**(*func*, *\*args*, *\*\*kwargs*)

Run a function on the client

#### Parameters

**func** –

#### Returns

**optimize**(*\*args*, *\*\*kwargs*)

Run Dask optimisation of graphs

Only does something when using dask

#### Parameters

- **args** – for `Dask.optimize`

- **kwargs** – for Dask.optimize

**Returns****close()**

Close the client

**init\_statistics()**

Initialise the profile and task stream info

rsexecute can save the Dask profile and Task Stream information for later saving

**Returns****save\_statistics(name='dask')**

Save the statistics to html files

rsexecute can save the Dask profile and Task Stream information for later saving. This saves the current statistics to html files.

**Parameters**

**name** – prefix to name e.g. dask

**memusage(memusage\_file='memusage.csv')**

Install the dask-memusage plugin

[https://github.com/itamarst/dask-memusage/blob/master/dask\\_memusage.py](https://github.com/itamarst/dask-memusage/blob/master/dask_memusage.py)

Note that there can only be one dask thread per process.

This only works for the process scheduler. For the distributed scheduler, preload the plugin. For example:

dask-scheduler --port=8786 --preload dask\_memusage --memusage-csv ./memusage.csv

**Parameters**

**memusage\_file** – Name of mem-usage file produced by dask-memusage plugin

**Returns****property client**

Client being used

**Returns**

client

**property using\_dask**

Is dask being used?

**Returns****property using\_dlg**

Is daluige being used?

**Returns****property optimizing**

Is Dask optimisation being performed?

**Returns**

### 1.4.3 Apps

The following command line apps are available.

- `genindex`
- `modindex`

## 1.5 RASCIL development

RASCIL is part of the SKA telescope organisation on GitLab <https://gitlab.com/ska-telescope/external/rascil.git> and development is ongoing. We welcome merge requests submitted via GitLab. Guidelines and instructions for contributing to code and documentation can be found [here](#).

### 1.5.1 Developing in RASCIL

Use the SKA Python Coding Guidelines (<http://developer.skatelescope.org/en/latest/development/python-codeguide.html>).

We recommend using a tool to help ensure PEP 8 compliance. PyCharm does a good job at this and other code quality checks.

#### Process

- Use git to make a local clone of the Github repository:

```
git clone https://gitlab.com/ska-telescope/rascil
```

- Make a branch. Use a descriptive name e.g. `feature_improved_gridding`, `bugfix_issue_666`
- Make whatever changes are needed, including documentation.
- Always add appropriate test code in the `tests` directory.
- Consider adding to the examples area.
- Push the branch to gitlab. It will then be automatically built and tested on gitlab: <https://gitlab.com/ska-telescope/rascil/-/pipelines>
- Once it builds correctly, submit a merge request.

#### Design

The RASCIL has been designed in line with the following principles:

- Data are held in Classes.
- The Data Classes correspond to familiar concepts in radio astronomy packages e.g. `visibility`, `gaintable`, `image`.
- The data members of the Data Classes are directly accessible by name e.g. `.data`, `.name`, `.phasecentre`.
- Direct access to the data members is envisaged.
- There are no methods attached to the data classes apart from variant constructors as needed.
- Standalone, stateless functions are used for all processing.

Additions and changes should adhere to these principles.

## Submitting code

RASCIL is part of the SKA telescope organisation on GitLab. <https://gitlab.com/ska-telescope/rascil.git>.

We welcome merge requests submitted via GitLab. Please note that we use Black to keep the python code style in good shape. The first step in the CI pipeline checks that the code complies with black formatting style, and will fail if that is not the case.

## 1.5.2 Documenting RASCIL

- The primary documentation is written in `reStructuredText` (rst).
- We use `Sphinx` to extract code documentation.
- We use the package `sphinx_automodapi` to build the API information.
- For this to work, all of the code must be loadable into python. To facilitate this, we make use of the dreaded `from somewhere import *`. This means that modules must use `__all__` to only export those names that are delivered by that module, as opposed to the other names used in the module.

## 1.5.3 Build and Release process

### Automatic builds

RASCIL is built automatically via a GitLab CI pipeline, which can be triggered by:

- on schedule
- commit to any branch
- merge/commit to master
- a tag is pushed to the repository

The following stages/jobs run, depending on the trigger mechanism:

- **on schedule: the `compile_requirements` job runs, whose sole purpose is to regularly update the requirements files with the latest package versions. It also runs the `.post` stage.**
- **commit to a branch: it runs the `linting` and `test` stages, as well as the `prepost` and `.post` ones.** The latter two creates and posts the `ci_metrics` data.
- **merge/commit to master:**
  - `linting`, and `test` stages run
  - `build` stage runs with the data and `build_package` jobs. The first builds and saves the RASCIL data to GitLab, while the second builds the RASCIL python package for later consumption
  - the `publish` stage's `docker_latest` job runs, which builds, tags and publishes the latest docker images to the Central Artefact Repository. This stage also runs the `pages` job, which publishes the documentation and rebuilds the data.
  - `prepost` and `.post` stages run
- **commit tag: tagging the repository is manual (see below), which triggers the following parts of the pipeline**
  - `linting` stage
  - `build` stage's `build_package` job, which builds the RASCIL python package

- publish stage’s `publish_to_car` and `docker_release` jobs. The first publishes the python package, while the second publishes the release-tagged (i.e. tagged with the package version) docker image to the Central Artefact Repository
- `.post` stage

The above process makes sure that new code is automatically tested at every point of the development process, and that the correct version of the python package and the docker images are published with the appropriate tag and at the appropriate time.

### Releasing a new version

The release process:

- Overall based on: <https://developer.skao.int/en/latest/tools/software-package-release-procedure.html>
- Use semantic versioning: <https://semver.org>
- Follow the packaging process in: <https://packaging.python.org/tutorials/packaging-projects/>

The release of a new package happens in two stages:

- a release tag is pushed to the repository (manually by a maintainer)
- the CI pipeline’s relevant stages publish the new package.

Note: while commits are allowed directly to master by maintainers of the repository, this should not be used as an option, but rather update the code via Merge Requests. This is only allowed for releasing a new version of the package.

### Steps:

- Ensure that the current master builds on GitLab: <https://gitlab.com/ska-telescope/external/rascil/-/pipelines>
- Decide whether a release is warranted and what semantic version number it should be: <https://semver.org>
- Check if the documentation has been updated. If not, create a new branch, update the documentation, create a merge request and merge that to master (after approval).
- Check out master and pull the latest version of it.
- Update CHANGELOG.md for the relevant changes in this release, putting newer description at the top.
- Commit the changes (do not push!)
- Bump the version using the Makefile:

```
make release-[patch|minor|major]
```

Note: `bumpver` needs to be installed. This step automatically commits the new version tag to the repository.

- Review the pipeline build for success
- Create a new virtualenv and try the install by using `pip3` install `rascil`:

```
virtualenv test_env
. test_env/bin/activate
pip3 install --index-url=https://artefact.skao.int/repository/pypi-all/simple rascil
python3
>>> import rascil
```

### 1.5.4 Managing requirements

RASCIL requirements are stored in three files:

- `requirements.in` Python requirements for the main code base
- `requirements-test.in` Python requirements to run the tests
- `requirements-docs.in` Python requirements to build the documentation

`pip-compile` is used to generate the corresponding `.txt` files. `pip-compile` resolves all dependencies and saves them with their resolved versions in the `.txt` files.

This method is used to make sure we do not update requirements with every build, but rather install them from the `.txt` files, where they are pinned. We also have to make sure we regularly update these versions, by running `pip-compile` on the `.in` files, which ideally do not contain version pins.

#### Manually updating the requirements

The Makefile of RASCIL contains three options to work with requirements on your local machine:

- `make requirements` This will update the requirements in the `.txt` file, but will not install them
- `make install_requirements` This will install the existing requirements from the `.txt` files, but not update them
- `make update_requirements` This will first update all requirements, then install them (i.e it runs the first two commands)

The first and third commands change the `.txt` files, but do not commit the changes. Still, it is worth running them from a branch, and not directly from master.

#### Process automation

Regularly updating the requirements manually is prone to be forgotten, which can result in packages being out-of-date very quickly. Hence we set up a semi-automatic process using the GitLab CI pipeline with a job run on a schedule.

The scheduled pipeline only runs one job, with the following steps:

- `run make requirements`
- check if there are changes compared to the existing remote files
- if there, create and check out a new branch
- commit and push the changes to the new branch
- create a Merge Request (MR) of the new branch into the source branch
- assign the MR
- if there aren't any changes, do nothing

The tests are not run as part of this pipeline, because the MR created at the end of will have the tests run as part of its own pipeline.

The assignee now has the responsibility of keeping track how the pipeline of this new MR does. If it succeeds, then it should be merged to master. If it fails, then the failing tests should be checked and the reasons for failure should be fixed. Packages should not be pinned within the `.in` files, just because tests are failing, unless there is a very good reason for it. Packages pinned in the `.in` files should be regularly revisited and if possible, unpinned.

### 1.5.5 Background

This outlines the original motivation for the ARL. Some shift in emphasis has occurred as a result of the expansion of RASCIL beyond the original purpose of a reference library.

#### Core motivations

- In many software packages, the only function specification is the application code itself. Although the underlying algorithm may be published, the implementation tends to diverge over time, making this method of documentation less effective. The algorithm reference library is designed to present imaging algorithms in a simple Python-based form. This is so that the implemented functions can be seen and understood without resorting to interpreting source code shaped by real-world concerns such as optimisations.
- Maintenance of the reference library over time is a choice for operations and we do not discuss it further here.
- Desire for simple test version: for example, scientists may wish to understand how the algorithm works and see it tested in particular circumstances. Or a software developer wish to compare it to production code.

#### Purpose

- Documentation: The primary purpose of the library is to be easily understandable to people not familiar with radio interferometry imaging. This means that the library should be broken down into a number of small, well-documented functions. Aside from the code itself, these functions will be further explained by documentation as well as material demonstrating its usage. Where such efforts would impact the clarity of the code itself it should be kept separate (e.g. example notebooks).
- Testbed for experimentation: One purpose for the library is to facilitate experimentation with the algorithm without touching the production code. Production code may be specialised due to the need for optimization, however the reference implementation should avoid any assumptions not actually from the theory of interferometry imaging.
- Publication e.g. via github: All algorithms used in production code should be known and published. If the algorithms are available separately from the production code then others can make use of the published code for small projects or to start on an improved algorithm.
- Conduit for algorithms into SKA: The library can serve as a conduit for algorithms into the SKA production system. A scientist can provide Python Version of an algorithm which then can be translated into optimized production code by the SKA computer team.
- Algorithm unaffected by optimization: Production code is likely to be obscured by the need to optimize in various ways. The algorithms in the library will avoid this as much as possible in order to remain clear and transparent. Where algorithms need to be optimised in order to remain executable on typical hardware, we might opt for providing multiple equivalent algorithm variants.
- Deliver algorithms for construction phase: The algorithm reference library Will also serve as a resource for the delivery of algorithms to the construction phase. It is likely that much of the production code will be written by people not intimately familiar with radio astronomy. Experience shows that such developers can often work from a simple example of the algorithm.
- Reference for results: The library will also serve to provide reference results for the production code. This is not entirely straightforward because the algorithms in both cases work in different contexts. Code that establishes interoperability with external code will have to be kept separate to not clutter the core implementation. This means that we will not be able to guarantee comparability in all cases. In that case, it will be the responsibility other developers of the production code to establish it - for example by using suitably reduced data sets.



## Stakeholders

- SDP design team: The principal stakeholders for the algorithm reference library are the SDP Design Team. They will benefit from having cleared descriptions of algorithms for all activities such as resource estimation, parameter setting, definition of pipelines, and so on.
- SKA Project Scientists: The SKA project scientists must be able to understand the algorithms used in the pipelines. This is essential if they are going to be assured that the processing is as desired, and relay that to the observers.
- External scientists: External scientists and observers using the telescope will benefit in two ways. First, in understanding the processing taking place in the pipelines, and second, being able to bring new algorithms for deployment into the pipelines.
- SDP contractors: Depending upon the procurement model, SDP may be developed by a team without very much domain knowledge. While we expect the documentation of the entire system to be in good shape after CDR, the algorithms are the very core of the system and must be communicated clearly and concisely. We can expect that any possible contractors considering a bid would be reassured by the presence of algorithm reference library.
- Outreach: Finally, outreach may be a consumer of the library. For example, the library could be made available to students at various levels to introduce them to astronomical data-processing concepts.

## Prior art

LAPACK is an example of a library that mutated into a reference library. The original code was written in straightforward FORTRAN but now many variants have been spawned including for example Versions optimized for particular hardware, or using software scheduling techniques such as DAGs to arrange their internal processing. The optimized variants must always agree with the reference code.

## Requirements

- Minimal implementation: The implementation should be minimal making use of as few external libraries as possible. Python is a good choice for the implementation because the associated libraries are powerful and well-defined.
- Use numpy whenever possible: Some form of numeric processing is inevitably necessary. There is also need for efficient bulk data transfer between functions. For consistency, we choose to adopt the numpy library for both algorithm and interface definition.
- Take algorithms with established provenance: While the purpose of the library is to define the algorithms clearly, the algorithms themselves should have well-defined provenance. Acceptable forms of provenance include publication in a peer-reviewed journal, publication in a well-defined memo series, and use in a well-defined production system. In time we might expect that the algorithm reference library will itself provide sufficient provenance. This depends upon the processes to maintain the library being stringently defined and applied.
- No optimization: No optimization should be performed on algorithms in the library if doing so obscures the fundamentals of the algorithm. Runtime of the testsuite should not be considered except in so far as it prevents effective use.
- V&V begins here: Validation and verification of the pipeline processing begins in the algorithm reference library. That means that it should be held to high standards of submission, testing, curation, and documentation.
- Single threaded: All algorithms should be single threaded unless multi-threading is absolutely required to achieve acceptable performance. However, as distributed execution is going to be vital for the SDP, special care should be taken to document and demonstrate parallelism opportunities.
- Memory limit: The memory used should be compatible with execution on a personal computer or laptop.

- How we maintain the requirements: *Managing requirements*

### Algorithms to be defined

The following list gives an initial set of algorithms to be defined. It is more important to have the overall framework in place expeditiously than to have each algorithm be state-of-the-art.

- Simulation
  - Station/Antenna locations
  - Illumination/Primary beam models
  - Generation of visibility data
  - Generation of gain tables
- Calibration
  - Calibration solvers
    - \* Stefcal
  - Calibration application
    - \* Gain interpolation
    - \* Gain application
  - Self-calibration
- Visibility plane
  - Convolution kernels
    - \* Standard
    - \* W Projection
    - \* AW Projection
    - \* AWI Projection
  - Degriding/Gridding
    - \* 2D
    - \* W projection
    - \* W slices
    - \* W snapshots
  - Preconditioning/Weighting
    - \* Uniform
    - \* Briggs
- Visibility plane to/from Image plane
  - DFT
  - Faceting
  - Phase rotation
  - Averaging/deaveraging

- Major cycles
- Image plane
  - Source finding
  - Source fitting
  - Reprojection
  - Interpolation
  - MSClean minor cycle (for spectral line)
  - MSMFS minor cycle (for continuum)

To test and demonstrate completeness, the main pipelines will be implemented.

## Testing

- Testing philosophy: The essence of an algorithm reference library is that it should be used as the standard for the structure and execution of a particular algorithm. This can only be done if the algorithm and the associated code are tested exhaustively.
- We will use three ways of performing testing of the code
  - Unit tests of all functions:
  - Regression tests of the complete algorithm over a complete set of inputs.
  - Code reviews (either single person or group read-throughs).
- Test suite via Jenkins: The algorithm reference library will therefore come with a complete set of unit tests and regression tests. These should be run automatically, by, for example, a framework such as Jenkins, on any change to ensure their errors are caught quickly and not compounded.
- genindex
- modindex
- genindex
- modindex



## PYTHON MODULE INDEX

**r** 92

`rascil.processing_components.calibration.iterators`, 47

`rascil.processing_components.calibration.operations`, 47

`rascil.processing_components.flagging.operations`, 48

`rascil.processing_components.griddata.convolution_functions`, 49

`rascil.processing_components.griddata.kernels`, 51

`rascil.processing_components.image.gradients`, 52

`rascil.processing_components.image.operations`, 53

`rascil.processing_components.imaging.imaging_params`, 60

`rascil.processing_components.imaging.primary_beams`, 60

`rascil.processing_components.simulation.atmospheric_screen`, 64

`rascil.processing_components.simulation.noise`, 67

`rascil.processing_components.simulation.pointing`, 68

`rascil.processing_components.simulation.rfi`, 69

`rascil.processing_components.simulation.simulation_helpers`, 71

`rascil.processing_components.simulation.surface`, 75

`rascil.processing_components.simulation.testing_support`, 76

`rascil.processing_components.skycomponent.plot_skycomponent`, 83

`rascil.processing_components.skymodel.operations`, 87

`rascil.processing_components.util.compass_bearing`, 91

`rascil.processing_components.util.installation_checks`, 91

`rascil.processing_components.util.performance`, 92

`rascil.processing_components.visibility.base`, 94

`rascil.processing_components.visibility.visibility_fitting`, 97

`rascil.workflows.rsexecute.calibration`, 99

`rascil.workflows.rsexecute.execution_support`, 116

`rascil.workflows.rsexecute.image`, 99

`rascil.workflows.rsexecute.imaging`, 101

`rascil.workflows.rsexecute.pipelines`, 107

`rascil.workflows.rsexecute.simulation`, 109

`rascil.workflows.rsexecute.skymodel`, 113



## Symbols

- `_rsexecutbase` (class in `ras-cil.workflows.rsexecute.execution_support.rsexecute`), 116
- ### A
- `add_image()` (in module `ras-cil.processing_components.image.operations`), 54
- `addnoise_visibility()` (in module `ras-cil.processing_components.simulation.noise`), 67
- `append_gaintable()` (in module `ras-cil.processing_components.calibration.operations`), 47
- `apply_bounding_box_convolutionfunction()` (in module `ras-cil.processing_components.griddata.convolution_functions`), 50
- `apply_voltage_pattern_to_image()` (in module `ras-cil.processing_components.image.operations`), 59
- `average_image_over_frequency()` (in module `ras-cil.processing_components.image.operations`), 54
- ### C
- `calculate_averaged_correlation()` (in module `ras-cil.processing_components.simulation.rfi`), 70
- `calculate_bounding_box_convolutionfunction()` (in module `ras-cil.processing_components.griddata.convolution_functions`), 50
- `calculate_initial_compass_bearing()` (in module `ras-cil.processing_components.util.compass_bearing`), 91
- `calculate_noise_visibility()` (in module `ras-cil.processing_components.simulation.noise`), 67
- `calculate_sf_from_screen()` (in module `ras-cil.processing_components.simulation.atmospheric_screen`), 66
- `calculate_skymodel_equivalent_image()` (in module `ras-cil.processing_components.skymodel.operations`), 89
- `calculate_station_correlation_rfi()` (in module `rascil.processing_components.simulation.rfi`), 70
- `calibrate_list_rsexecute_workflow()` (in module `rascil.workflows.rsexecute.calibration`), 99
- `check_data_directory()` (in module `ras-cil.processing_components.util.installation_checks`), 92
- `client` (`_rsexecutbase` property), 119
- `close()` (`_rsexecutbase` method), 119
- `compute()` (`_rsexecutbase` method), 118
- `continuum_imaging_skymodel_list_rsexecute_workflow()` (in module `ras-cil.workflows.rsexecute.pipelines`), 107
- `convert_azelpv_to_radec()` (in module `ras-cil.processing_components.imaging.primary_beams`), 64
- `corrupt_list_rsexecute_workflow()` (in module `rascil.workflows.rsexecute.simulation`), 109
- `create_atmospheric_errors_gaintable_rsexecute_workflow()` (in module `ras-cil.workflows.rsexecute.simulation`), 109
- `create_awterm_convolutionfunction()` (in module `ras-cil.processing_components.griddata.kernels`), 52
- `create_box_convolutionfunction()` (in module `ras-cil.processing_components.griddata.kernels`), 51
- `create_gaintable_from_rows()` (in module `ras-cil.processing_components.calibration.operations`), 48
- `create_gaintable_from_screen()` (in module `ras-cil.processing_components.simulation.atmospheric_screen`), 65
- `create_heterogeneous_gaintable_rsexecute_workflow()` (in module `ras-`

<code>cil.workflows.rsexecute.simulation), 110</code>	<code>cil.workflows.rsexecute.simulation), 112</code>
<code>create_low_test_beam() (in module ras-</code>	<code>create_test_image() (in module ras-</code>
<code>cil.processing_components.imaging.primary_beams),</code>	<code>cil.processing_components.simulation.testing_support),</code>
63	79
<code>create_low_test_image_from_gleam()</code>	<code>create_test_image_from_s3() (in module ras-</code>
<code>(in module ras-</code>	<code>cil.processing_components.simulation.testing_support),</code>
<code>cil.processing_components.simulation.testing_support),</code>	80
77	<code>create_test_skycomponents_from_s3()</code>
<code>create_low_test_skycomponents_from_gleam()</code>	<code>(in module ras-</code>
<code>(in module ras-</code>	<code>cil.processing_components.simulation.testing_support),</code>
<code>cil.processing_components.simulation.testing_support),</code>	81
78	<code>create_unittest_components() (in module ras-</code>
<code>create_low_test_skymodel_from_gleam()</code>	<code>cil.processing_components.simulation.testing_support),</code>
<code>(in module ras-</code>	82
<code>cil.processing_components.simulation.testing_support),</code>	<code>create_unittest_model() (in module ras-</code>
78	<code>cil.processing_components.simulation.testing_support),</code>
<code>create_low_test_vp() (in module ras-</code>	82
<code>cil.processing_components.imaging.primary_beams),</code>	<code>create_visibility_from_ms() (in module ras-</code>
63	<code>cil.processing_components.visibility.base),</code>
<code>create_mid_allsky() (in module ras-</code>	95
<code>cil.processing_components.imaging.primary_beams),</code>	<code>create_visibility_from_uvfits() (in module ras-</code>
64	<code>cil.processing_components.visibility.base), 96</code>
<code>create_mid_simulation_components()</code>	<code>create_voltage_pattern_gaintable_rsexecute_workflow()</code>
<code>(in module ras-</code>	<code>(in module ras-</code>
<code>cil.processing_components.simulation.simulation_helpers),</code>	<code>cil.workflows.rsexecute.simulation), 112</code>
74	<code>create_vp() (in module ras-</code>
<code>create_pb() (in module ras-</code>	<code>cil.processing_components.imaging.primary_beams),</code>
<code>cil.processing_components.imaging.primary_beams),</code>	62
61	<code>create_vp_generic() (in module ras-</code>
<code>create_pb_generic() (in module ras-</code>	<code>cil.processing_components.imaging.primary_beams),</code>
<code>cil.processing_components.imaging.primary_beams),</code>	62
62	<code>create_vp_generic_numeric() (in module ras-</code>
<code>create_pointing_errors_gaintable_rsexecute_workflow()</code>	<code>cil.processing_components.imaging.primary_beams),</code>
<code>(in module ras-</code>	62
<code>cil.workflows.rsexecute.simulation), 110</code>	<code>create_vp_term_convolutionfunction()</code>
<code>create_polarisation_gaintable_rsexecute_workflow()</code>	<code>(in module ras-</code>
<code>(in module ras-</code>	<code>cil.processing_components.griddata.kernels),</code>
<code>cil.workflows.rsexecute.simulation), 111</code>	52
<code>create_pswf_convolutionfunction()</code>	<code>create_w_term_like() (in module ras-</code>
<code>(in module ras-</code>	<code>cil.processing_components.image.operations),</code>
<code>cil.processing_components.griddata.kernels),</code>	54
51	<code>create_window() (in module ras-</code>
<code>create_skymodel_from_skycomponents_gaintables()</code>	<code>cil.processing_components.image.operations),</code>
<code>(in module ras-</code>	54
<code>cil.processing_components.skymodel.operations),</code>	
90	<b>D</b>
<code>create_standard_low_simulation_rsexecute_workflow()</code>	<code>deconvolve_list_channel_rsexecute_workflow()</code>
<code>(in module ras-</code>	<code>(in module rascil.workflows.rsexecute.imaging),</code>
<code>cil.workflows.rsexecute.simulation), 111</code>	101
<code>create_standard_mid_simulation_rsexecute_workflow()</code>	<code>deconvolve_list_rsexecute_workflow() (in mod-</code>
<code>(in module ras-</code>	<code>ule rascil.workflows.rsexecute.imaging), 102</code>
<code>cil.workflows.rsexecute.simulation), 112</code>	<code>deconvolve_skymodel_list_rsexecute_workflow()</code>
<code>create_surface_errors_gaintable_rsexecute_workflow()</code>	<code>(in module ras-</code>
<code>(in module ras-</code>	<code>cil.workflows.rsexecute.skymodel), 114</code>



## E

`execute()` (`_rsexecutbase` method), 117

`expand_skymodel_by_skymodel_components()` (in module `ras-cil.processing_components.skymodel.operations`), 90

`export_convolutionfunction_to_fits()` (in module `ras-cil.processing_components.griddata.convolution_functions`), 50

`export_visibility_to_ms()` (in module `ras-cil.processing_components.visibility.base`), 96

`extend_visibility_to_ms()` (in module `ras-cil.processing_components.visibility.base`), 96

`extract_skymodel_components_from_skymodel()` (in module `ras-cil.processing_components.skymodel.operations`), 90

`generate_baselines()` (in module `ras-cil.processing_components.visibility.base`), 97

`get_dask_client()` (in module `ras-cil.workflows.rsexecute.execution_support`), 116

`get_frequency_map()` (in module `ras-cil.processing_components.imaging.imaging_params`), 60

`get_polarisation_map()` (in module `ras-cil.processing_components.imaging.imaging_params`), 60

`get_rowmap()` (in module `ras-cil.processing_components.imaging.imaging_params`), 60

`git_hash()` (in module `ras-cil.processing_components.util.performance`), 92

`grid_gaintable_to_screen()` (in module `ras-cil.processing_components.simulation.atmospheric_screen`), 66

## F

`fft_image_to_griddata_with_wcs()` (in module `ras-cil.processing_components.image.operations`), 55

`find_pb_width_null()` (in module `ras-cil.processing_components.simulation.simulation_helpers`), 74

`find_pierce_points()` (in module `ras-cil.processing_components.simulation.atmospheric_screen`), 65

`find_times_above_elevation_limit()` (in module `ras-cil.processing_components.simulation.simulation_helpers`), 72

`fit_visibility()` (in module `ras-cil.processing_components.visibility.visibility_fitting`), 97

`flagging_aoflagger()` (in module `ras-cil.processing_components.flagging.operations`), 49

`flagging_visibility()` (in module `ras-cil.processing_components.flagging.operations`), 49

`ical_skymodel_list_rsexecute_workflow()` (in module `rascil.workflows.rsexecute.pipelines`), 108

`image_gather_channels_rsexecute()` (in module `rascil.workflows.rsexecute.image`), 100

`image_gradients()` (in module `ras-cil.processing_components.image.gradients`), 53

`image_rsexecute_map_workflow()` (in module `ras-cil.workflows.rsexecute.image`), 100

`import_image_from_fits()` (in module `ras-cil.processing_components.image.operations`), 56

`ingest_unittest_visibility()` (in module `ras-cil.processing_components.simulation.testing_support`), 82

`init_statistics()` (`_rsexecutbase` method), 119

`initialize_skymodel_voronoi()` (in module `ras-cil.processing_components.skymodel.operations`), 89

`insert_unittest_errors()` (in module `ras-cil.processing_components.simulation.testing_support`), 82

## G

`gaintable_plot()` (in module `ras-cil.processing_components.calibration.operations`), 48

`gaintable_timeslice_iter()` (in module `ras-cil.processing_components.calibration.iterators`), 47

`invert_list_rsexecute_workflow()` (in module `ras-cil.workflows.rsexecute.imaging`), 102

`invert_skymodel_list_rsexecute_workflow()` (in module `rascil.workflows.rsexecute.skymodel`), 114

## L

`list_ms()` (in module `ras-`

## M

memusage() (*\_rsexecutebase method*), 119

module

    rascil.processing\_components.calibration.iterators, 47

    rascil.processing\_components.calibration.operations, 47

    rascil.processing\_components.flagging.operations, 48

    rascil.processing\_components.griddata.convolution.functions, 49

    rascil.processing\_components.griddata.kernels, 51

    rascil.processing\_components.image.gradients, 52

    rascil.processing\_components.image.operations, 53

    rascil.processing\_components.imaging.imaging\_params, 60

    rascil.processing\_components.imaging.primary\_beams, 60

    rascil.processing\_components.simulation.atmospheric\_screen, 64

    rascil.processing\_components.simulation.noise, 67

    rascil.processing\_components.simulation.pointing, 68

    rascil.processing\_components.simulation.rfi, 69

    rascil.processing\_components.simulation.simulation\_helpers, 71

    rascil.processing\_components.simulation.surface, 75

    rascil.processing\_components.simulation.testing\_support, 76

    rascil.processing\_components.skycomponent.plot\_skycomponent, 83

    rascil.processing\_components.skymodel.operations, 87

    rascil.processing\_components.util.compass\_bearing, 91

    rascil.processing\_components.util.installation\_checks, 91

    rascil.processing\_components.util.performance, 92

    rascil.processing\_components.visibility.base, 94

    rascil.processing\_components.visibility.visibility\_fitting, 97

    rascil.workflows.rsexecute.calibration, 99

    rascil.workflows.rsexecute.execution\_support, 116

    rascil.workflows.rsexecute.image, 99

    rascil.workflows.rsexecute.imaging, 101

    rascil.workflows.rsexecute.pipelines, 107

    rascil.workflows.rsexecute.simulation, 109

    rascil.workflows.rsexecute.skymodel, 113

## N

    normalize\_vp() (in module rascil.processing\_components.imaging.primary\_beams), 54

    normalize\_vp(), 54

## O

    optimize() (*\_rsexecutebase method*), 118

    optimizing (*\_rsexecutebase property*), 119

## P

    pad\_image() (in module rascil.processing\_components.image.operations), 56

    partition\_skymodel\_by\_flux() (in module rascil.processing\_components.skymodel.operations), 88

    performance\_dask\_configuration() (in module rascil.processing\_components.util.performance), 93

    performance\_environment() (in module rascil.processing\_components.util.performance), 94

    performance\_merge\_memory() (in module rascil.processing\_components.util.performance), 94

    performance\_qa\_image() (in module rascil.processing\_components.util.performance), 93

    performance\_read() (in module rascil.processing\_components.util.performance), 93

    performance\_read\_memory\_data() (in module rascil.processing\_components.util.performance), 94

    performance\_store\_dict() (in module rascil.processing\_components.util.performance), 93

    persist() (*\_rsexecutebase method*), 118

    plot\_azel() (in module rascil.processing\_components.simulation.simulation\_helpers), 73

    plot\_configuration() (in module rascil.processing\_components.simulation.simulation\_helpers), 73

`plot_gaintable()` (in module `ras-` `polarisation_frame_from_wcs()` (in module `ras-`  
`cil.processing_components.simulation.simulation_helpers`), `cil.processing_components.image.operations`),  
73 57  
`plot_gaintable_on_screen()` (in module `ras-` `predict_list_rsexecute_workflow()` (in module  
`cil.processing_components.simulation.atmospheric_screen`), `rascil.workflows.rsexecute.imaging`), 103  
66  
`plot_gaussian_beam_position()` (in module `ras-` (in module `ras-`  
`cil.processing_components.skycomponent.plot_skycomponent`), `rascil.workflows.rsexecute.skymodel`), 114  
87  
`plot_multifreq_spectral_index()` (in module `ras-` **R**  
`cil.processing_components.skycomponent.plot_skycomponent`), `rascil.processing_components.calibration.iterators`  
87 module, 47  
`plot_pa()` (in module `ras-` `rascil.processing_components.calibration.operations`  
`cil.processing_components.simulation.simulation_helpers`), module, 47  
75  
`plot_pointingtable()` (in module `ras-` `rascil.processing_components.flagging.operations`  
`cil.processing_components.simulation.simulation_helpers`), module, 48  
74  
`plot_skycomponents_flux()` (in module `ras-` `rascil.processing_components.griddata.convolution_function`  
`cil.processing_components.skycomponent.plot_skycomponent`), module, 49  
85  
`plot_skycomponents_flux_histogram()` `rascil.processing_components.griddata.kernels`  
(in module `ras-` module, 51  
`cil.processing_components.skycomponent.plot_skycomponent`), `rascil.processing_components.image.gradients`  
86 module, 52  
`plot_skycomponents_flux_ratio()` (in module `ras-` `rascil.processing_components.image.operations`  
`cil.processing_components.skycomponent.plot_skycomponent`), module, 53  
85  
`plot_skycomponents_position_distance()` `rascil.processing_components.imaging.imaging_params`  
(in module `ras-` module, 60  
`cil.processing_components.skycomponent.plot_skycomponent`), `rascil.processing_components.imaging.primary_beams`  
85 module, 60  
`plot_skycomponents_position_quiver()` `rascil.processing_components.simulation.atmospheric_screen`  
(in module `ras-` module, 64  
`cil.processing_components.skycomponent.plot_skycomponent`), `rascil.processing_components.simulation.noise`  
85 module, 67  
`plot_skycomponents_position_quiver()` `rascil.processing_components.simulation.pointing`  
(in module `ras-` module, 68  
`cil.processing_components.skycomponent.plot_skycomponent`), `rascil.processing_components.simulation.rfi`  
86 module, 69  
`plot_skycomponents_positions()` (in module `ras-` `rascil.processing_components.simulation.simulation_helpers`  
`cil.processing_components.skycomponent.plot_skycomponent`), module, 71  
84  
`plot_uvcoverage()` (in module `ras-` `rascil.processing_components.simulation.surface`  
`cil.processing_components.simulation.simulation_helpers`), module, 75  
72  
`plot_uwcoverage()` (in module `ras-` `rascil.processing_components.simulation.testing_support`  
`cil.processing_components.simulation.simulation_helpers`), module, 76  
72  
`plot_visibility()` (in module `ras-` `rascil.processing_components.skycomponent.plot_skycomponent`  
`cil.processing_components.simulation.simulation_helpers`), module, 83  
71  
`plot_visibility_pol()` (in module `ras-` `rascil.processing_components.skymodel.operations`  
`cil.processing_components.simulation.simulation_helpers`), module, 87  
71  
`plot_vwcoverage()` (in module `ras-` `rascil.processing_components.util.compass_bearing`  
`cil.processing_components.simulation.simulation_helpers`), module, 91  
73  
`plot_vwcoverage()` (in module `ras-` `rascil.processing_components.util.installation_checks`  
`cil.processing_components.simulation.simulation_helpers`), module, 91  
73  
`plot_vwcoverage()` (in module `ras-` `rascil.processing_components.util.performance`  
`cil.processing_components.simulation.simulation_helpers`), module, 92  
73  
`plot_vwcoverage()` (in module `ras-` `rascil.processing_components.visibility.base`  
`cil.processing_components.simulation.simulation_helpers`), module, 94  
73

rascil.processing\_components.visibility.visibility\_fitting  
     module, 97  
 rascil.workflows.rsexecute.calibration  
     module, 99  
 rascil.workflows.rsexecute.execution\_support  
     module, 116  
 rascil.workflows.rsexecute.image  
     module, 99  
 rascil.workflows.rsexecute.imaging  
     module, 101  
 rascil.workflows.rsexecute.pipelines  
     module, 107  
 rascil.workflows.rsexecute.simulation  
     module, 109  
 rascil.workflows.rsexecute.skymodel  
     module, 113  
 remove\_continuum\_image() (in module rascil.processing\_components.image.operations),  
     57  
 replicate\_image() (in module rascil.processing\_components.simulation.testing\_support),  
     83  
 reproject\_image() (in module rascil.processing\_components.image.operations),  
     57  
 residual\_list\_rsexecute\_workflow() (in module rascil.workflows.rsexecute.imaging), 104  
 restore\_centre\_rsexecute\_workflow() (in module rascil.workflows.rsexecute.imaging), 104  
 restore\_centre\_skymodel\_list\_rsexecute\_workflow()  
     (in module rascil.workflows.rsexecute.skymodel), 115  
 restore\_list\_rsexecute\_workflow() (in module rascil.workflows.rsexecute.imaging), 104  
 restore\_skymodel\_list\_rsexecute\_workflow()  
     (in module rascil.workflows.rsexecute.skymodel), 115  
 run() (*\_rsexecutebase* method), 118

**S**

save\_statistics() (*\_rsexecutebase* method), 119  
 scale\_and\_rotate\_image() (in module rascil.processing\_components.image.operations),  
     59  
 scatter() (*\_rsexecutebase* method), 118  
 set\_client() (*\_rsexecutebase* method), 117  
 set\_pb\_header() (in module rascil.processing\_components.imaging.primary\_beams),  
     61  
 show\_components() (in module rascil.processing\_components.image.operations),  
     58  
 show\_image() (in module rascil.processing\_components.image.operations),

show\_skymodel() (in module rascil.processing\_components.skymodel.operations),  
     89  
 simulate\_gaintable() (in module rascil.processing\_components.simulation.testing\_support),  
     83  
 simulate\_gaintable\_from\_pointingtable()  
     (in module rascil.processing\_components.simulation.pointing),  
     68  
 simulate\_gaintable\_from\_voltage\_pattern()  
     (in module rascil.processing\_components.simulation.surface),  
     76  
 simulate\_gaintable\_from\_zernikes()  
     (in module rascil.processing\_components.simulation.surface),  
     75  
 simulate\_list\_rsexecute\_workflow() (in module rascil.workflows.rsexecute.simulation), 113  
 simulate\_pointingtable() (in module rascil.processing\_components.simulation.pointing),  
     69  
 simulate\_pointingtable\_from\_timeseries()  
     (in module rascil.processing\_components.simulation.pointing),  
     68  
 simulate\_rfi\_block\_prop() (in module rascil.processing\_components.simulation.rfi),  
     70  
 smooth\_image() (in module rascil.processing\_components.image.operations),  
     59  
 spectral\_line\_imaging\_skymodel\_list\_rsexecute\_workflow()  
     (in module rascil.workflows.rsexecute.pipelines), 108  
 sub\_image() (in module rascil.processing\_components.image.operations),  
     56  
 subtract\_list\_rsexecute\_workflow() (in module rascil.workflows.rsexecute.imaging), 105  
 sum\_images\_rsexecute() (in module rascil.workflows.rsexecute.image), 100  
 sum\_invert\_results\_rsexecute() (in module rascil.workflows.rsexecute.imaging), 105  
 sum\_predict\_results\_rsexecute() (in module rascil.workflows.rsexecute.imaging), 105

**T**

taper\_list\_rsexecute\_workflow() (in module rascil.workflows.rsexecute.imaging), 106  
 threshold\_list\_rsexecute() (in module rascil.workflows.rsexecute.imaging), 106

`type()` (*\_rsexecutebase method*), [117](#)

## U

`update_skymodel_from_gaintables()`  
(*in module rascil.processing\_components.skymodel.operations*),  
[89](#)

`update_skymodel_from_image()` (*in module rascil.processing\_components.skymodel.operations*),  
[90](#)

`using_dask` (*\_rsexecutebase property*), [119](#)

`using_dlg` (*\_rsexecutebase property*), [119](#)

## W

`weight_list_rsexecute_workflow()` (*in module rascil.workflows.rsexecute.imaging*), [106](#)

## Z

`zero_list_rsexecute_workflow()` (*in module rascil.workflows.rsexecute.imaging*), [107](#)