developer.skatelescope.org Documentation

Release 0.1.0-beta

Marco Bartolini

Jan 30, 2024

HOME

1 Local development **3**
  1.1 Local development . . . 3

2 Direction-dependent effects **7**
  2.1 OSKAR cookbook . . . 7

3 Low-level RFI **29**
  3.1 Low-level RFI simulations . . . 29

Index **45**

This repository collects scripts used for various SKA-Low simulations.

CHAPTER

ONE

LOCAL DEVELOPMENT

1.1 Local development

1.1.1 Download

To clone the repository, use:

```
git clone https://gitlab.com/ska-telescope/ska-low-simulations.git
```

Or browse the files at https://gitlab.com/ska-telescope/ska-low-simulations

1.1.2 Install requirements

Local virtual environment

Note: the following information is to set up an environment that works with the RFI simulations only. Other scripts may need more packages to be added to requirements.txt.

Differences based on Operating System

Linux: you may create a virtual environment with conda, virtualenvwrapper, or other python-based virtual environment tool. Installing the requirements via pip should work in all.

MacOS: you will need to create the environemnt with conda. That is because `python-casacore` does not currently behave well, when trying to install it via `pip` into a standard python environment.

Create a virtual environment

virtualenvwrapper

To install and set up virtualenvwrapper follow this guide.

Create an environment: replace *my-environment* with the name you prefer, and replace *python3.7* with the path to your python3 installation. If the PYTHONPATH used by virtualenvwrapper is the python3 version you want to use, then you can omit the `-p` option.

```
mkvirtualenv -p python3.7 my-environment
```

Start environment:

```
workon my-environemnt
```

Deactivate environment:

```
deactivate
```

conda

To install and set up conda follow the conda guide.

Create an environemnt (replace *my-environment* with the name you prefer):

```
conda create --name my-environment python=3.7
```

Start environment:

```
conda activate my-environemnt
```

Deactivate environment:

```
conda deactivate
```

Install requirements

Linux

Once you have activated your environment and navigated into the ska-sim-low directory (i.e. the root directory of the git repository), run the following:

```
pip install -r requirements.txt --pre
```

`--pre` will allow you to download the latest beta versions of dependencies. This is necessary to get the latest RASCIL version from PyPi.

Depending on what python version you used to create the environment, the pip within that will be tied to that python version. This command should install all of the necessary requirements.

In addition, you will have to obtain RASCIL data. RASCIL will be installed via pip as part of the above command, however the additional setup described at RASCIL Installation is required. If you encounter with a `Tuple index out of range` error while running RASCIL-dependent code, you may also need to go through the `Git LFS` steps on the same page under "Installation via git clone".

You will also need OSKAR set up. On Linux, you may use the Singularity image.

MacOS

On MacOS, `python-casacore`, a dependency of RASCIL, does not behave well with `pip`, so you will need `conda` to install it.

Install `python-casacore` with conda:

```
conda install -c conda-forge python-casacore
```

Install the rest of the requirements using `pip`:

```
pip install -r requirements.txt --pre
```

`--pre` will allow you to download the latest beta versions of dependencies. This is necessary to get the latest RASCIL version from PyPi.

Depending on what python version you used to create the environment, the pip within that will be tied to that python version. This command should install all of the necessary requirements.

In addition, you will have to obtain RASCIL data. RASCIL will be installed via pip as part of the above command, however the additional setup described at RASCIL Installation is required. If you encounter with a `Tuple index out of range` error while running RASCIL-dependent code, you may also need to go through the `Git LFS` steps on the same page under "Installation via git clone".

You will also need OSKAR set up, which you can do via installing the binary version.

Docker container as Python interpreter

Note: the following instructions are still under development, as not all of the RFI code has been tested with this setup.

If you don not want to set up a complicated environment locally with all sorts of data also added to your machine, then you can create a Docker image, which then you can use as your python interpreter both from the command line and from *PyCharm* or *Visual Studio Code*.

Create a docker image

Create a Dockerfile, called *docker_python_env*, with the following information in it (do not add the file to git):

```
FROM nexus.engageska-portugal.pt/rascil-docker/rascil-base

WORKDIR /rascil/sim-low-rfi/

ADD requirements.txt requirements-test.txt .
ADD docs/requirements-docs.txt .

RUN pip install -r requirements.txt -r requirements-test.txt -r requirements-docs.txt
```

The starting image is `rascil-base`. This does not contain any RASCIL data. If you need RASCIL data as part of the image, you'll need to use `rascil-full`. Here you can read more about RASCIL container images.

Build the docker image (be in the ska-sim-low directory, where your personal dockerfile should also be:

```
docker build -t rfi-environment -f docker_python_env .
```

Docker as Python interpreter in IDE

Use this image as your Python interpreter in Pycharm Professional or Visual Studio Code. To set the environment up, please follow the links.

Develop locally and run code in Docker

You can also run your code, tests, bash scripts directly from the Docker container, while still accessing and changing the files on your machine with your favourite IDE or text editor.

Start the container:

```
docker run -it -v ${PWD}:/rascil/sim-low-rfi --rm rfi-environment:latest
```

This will take you inside the container. `--rm` will stop the container from running once you exit it. `${PWD}:/rascil/sim-low-rfi` will attach the directory where you start the container from, into a directory called `/rascil/sim-low-rfi` that is within the container. If you change something in this directory outside the container, the same changes will appear within the container. Make sure you start the container from the *ska-sim-low* directory, that way you can carry on changing those files and the changes will be present in the container as well. Now you can run, e.g., test within your container where you have a fully functioning python environment, while still developing on your local machine.

CHAPTER TWO

DIRECTION-DEPENDENT EFFECTS

Many of the simulations of direction-dependent effects on the sky make use of the OSKAR simulator. Scripts for these simulations are written specifically for each investigation.

A "cookbook" is provided here to help describe the examples that use OSKAR, and to provide a starting point for writing new scripts.

2.1 OSKAR cookbook

The sections below are intended to be read in order.

2.1.1 Basic concepts

OSKAR provides a toolbox of Python utilities for running simulations, and for making dirty or residual images to analyse the results. Since the potential parameter space for all possible simulations is very large, no single script could sensibly cater for them all - so in order to run a set of simulations for your own investigation, you will probably find it useful to write your own script. Don't panic though: if you're familiar with basic Python concepts, this is not hard. Depending on the simulations you want to run and the steps needed to analyse and present the results, the script to drive the simulations could be very simple.

This cookbook outlines some of the key concepts and provides some examples to show what is possible, which could be used as building blocks for your own scripts.

As shown in the sketch below, to produce simulated visibilities, OSKAR needs as inputs a sky model, a telescope model, and a few parameters to describe the observation.

Any simulation script will usually need to iterate over a number of simulated observations, changing parts of the sky model, the telescope model and/or the parameters in a systematic way for each run. Ways to set up the *sky model* and the *telescope model* are described on later pages, but the most commonly used settings parameters are outlined below.

Commonly used settings

The settings parameters used by OSKAR are generated from a Python dictionary of key-value pairs.

The following parameters will almost always need to be set appropriately when running any interferometer simulation with OSKAR. The values given below are examples only!

```
params = {
    'simulator': {
        'use_gpus': True  # or False
    },
```

(continues on next page)

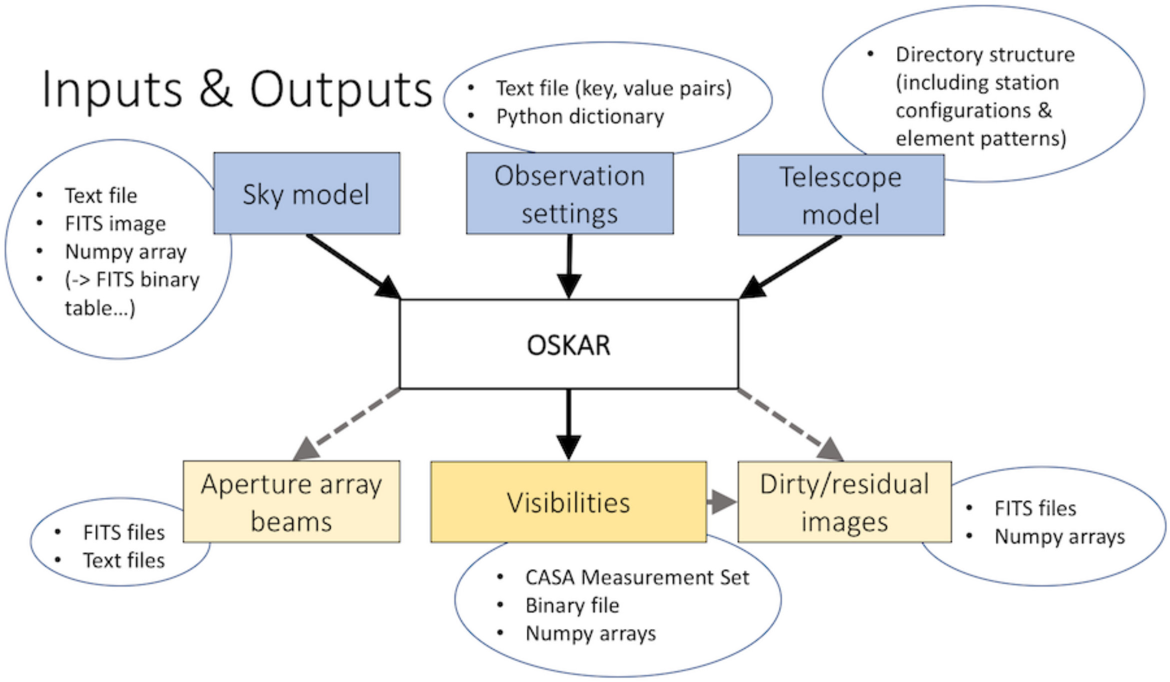


Fig. 1: Overview of OSKAR inputs and outputs

(continued from previous page)

```
    'observation': {
        'num_channels': 3,  # Simulate 3 frequency channels
        'start_frequency_hz': 100e6,  # First channel at 100 MHz
        'frequency_inc_hz': 20e6,  # Channel separation of 20 MHz
        'phase_centre_ra_deg': 20,
        'phase_centre_dec_deg': -30,
        'num_time_steps': 24,  # Simulate 24 correlator dumps
        'start_time_utc': '2000-01-01 12:00:00.000',
        'length': '12:00:00.000'  # 12 hours, or length in seconds
    },
    'telescope': {
        'input_directory': '/absolute/or/relative/path/to/a/telescope_model_folder.tm/'
    },
    'interferometer': {
        'channel_bandwidth_hz': 10e3,
        'time_average_sec': 1.0,
        'oskar_vis_filename': 'example.vis',
        'ms_filename': 'example.ms'
    }
}
```

The dictionary keys may be nested, as above, or flat if it is more convenient. The following is entirely equivalent to the above:

```
params = {
    'simulator/use_gpus': True,  # or False
```

(continues on next page)

(continued from previous page)

```
    'observation/num_channels': 3,  # Simulate 3 frequency channels
    'observation/start_frequency_hz': 100e6,  # First channel at 100 MHz
    'observation/frequency_inc_hz': 20e6,  # Channel separation of 20 MHz
    'observation/phase_centre_ra_deg': 20,
    'observation/phase_centre_dec_deg': -30,
    'observation/num_time_steps': 24,  # Simulate 24 correlator dumps
    'observation/start_time_utc': '2000-01-01 12:00:00.000',
    'observation/length': '12:00:00.000',  # 12 hours, or length in seconds
    'telescope/input_directory': '/absolute/or/relative/path/to/a/telescope_model_folder.
↪tm/',
    'interferometer/channel_bandwidth_hz': 10e3,
    'interferometer/time_average_sec': 1.0,
    'interferometer/oskar_vis_filename': 'example.vis',
    'interferometer/ms_filename': 'example.ms'
}
```

Using these example parameters, simulated visibility data will be written to a CASA Measurement Set specified by the `interferometer/ms_filename` settings key (in this case `example.ms`) and also a binary visibility data file specified by the `interferometer/oskar_vis_filename` key (here, `example.vis`).

Most of the rest of the parameters specify the time and frequency coverage of the observation, as well as the direction of the phase centre.

The hardest parameter to set is usually the start time. To help with this, a utility function called `get_start_time` is provided in the file `utils.py`, which calculates an optimal start time using the target Right Ascension, the observation length, and the longitude of the SKA-Low telescope. The observation will then be symmetric about the meridian.

The full list of settings parameters is shown in the OSKAR GUI for the `oskar_sim_interferometer` application, and also in the settings documentation.

Creating a settings tree and interferometer simulator

After defining parameters in a standard Python dictionary as above, an OSKAR `SettingsTree` should be created from it, and this can be used to instantiate other classes to run a simulation.

To set up an `oskar.Interferometer` simulator in Python, use the parameters for the `oskar_sim_interferometer` application, and then set them from the Python dictionary as follows:

```
settings = oskar.SettingsTree('oskar_sim_interferometer')
settings.from_dict(params)  # using the Python dictionary above.
```

This `settings` object can then be passed as a parameter to the constructor:

```
sim = oskar.Interferometer(settings=settings)
```

Setting the input data models

A sky model and telescope model can be defined either using the settings parameters, or set programmatically from Python - the latter option being useful, for example, if changing a model within a loop. The methods `oskar.Interferometer.set_sky_model()` and `oskar.Interferometer.set_telescope_model()` can be used for this. See the following pages for more details.

2.1.2 Setting up a sky model

Sky models used by OSKAR exist completely independently of any other simulation parameters. The sky model can be thought of as simply a table of source data, where each row of the table contains parameters for a single source. As a bare minimum (but often sufficient for many simulations), each source must specify a Right Ascension, Declination, and Stokes I flux as the first three columns. **Source coodinates must be specified in decimal degrees, and source fluxes in Jy.**

The class `oskar.Sky` is used to encapsulate data for a sky model. Useful class methods (which create and return a new sky model) include:

- **from_array(array, precision='double')**
  - to convert a numpy array to a sky model.
- **generate_grid(ra0_deg, dec0_deg, side_length, fov_deg, mean_flux_jy=1.0, std_flux_jy=0.0, seed=1, precision='double')**
  - to generate a grid of sources around a point.
- **load(filename, precision='double')**
  - to load a sky model from a text file.

Useful methods on the class include:

- **append(from_another)**
  - to append another sky model to this one.
- **create_copy()**
  - to create and return a copy of a sky model.
- **filter_by_flux(min_flux_jy, max_flux_jy)**
  - to remove sources from the sky model based on their Stokes I flux. (Sources with fluxes outside the specified range will be removed.)
- **filter_by_radius(inner_radius_deg, outer_radius_deg, ref_ra_deg, ref_dec_deg)**
  - to remove sources from the sky model based on their angular distance from a reference point. (Sources with distances outside the specified range will be removed.)
- **save(filename)**
  - to save the sky model to a text file.
- **to_array()**
  - to convert the sky model to a numpy array.

Example: Using the GLEAM catalogue

The GLEAM Extragalactic Catalogue can be downloaded as a FITS binary table from the VizieR service. To use data from a FITS binary table as a sky model, pull the data columns out into a new array using `astropy` and then create an OSKAR sky model from the array, as follows:

```
from astropy.io import fits
import numpy
import oskar

# Get the first HDU (a binary table) in the specified FITS file.
data = fits.getdata('GLEAM_EGC.fits', 1)

# Create a 3-column numpy array (RA, Dec, Stokes I).
sky_array = numpy.column_stack(
                (data['RAJ2000'], data['DEJ2000'], data['peak_flux_wide']))

# Create the sky model from the 3-column numpy array above.
sky = oskar.Sky.from_array(sky_array)

# Print the number of sources in the sky model.
print(sky.num_sources)
>>> 307455
```

Example: Filtering a sky model

It may be necessary to filter a sky model to remove sources inside or outside a certain radius from a specific point (such as the phase centre) as part of a simulation script.

For example, to keep sources only within 20 degrees of the point at (RA, Dec) = (0, 80) degrees, use:

```
ra0 = 0
dec0 = 80
sky.filter_by_radius(0, 20, ra0, dec0)
```

Similarly, to keep sources only *outside* a radius of 20 degrees from the same point, use instead:

```
sky.filter_by_radius(20, 180, ra0, dec0)
```

2.1.3 Setting up a telescope model

An OSKAR telescope model encapuslates all static (time-invariant) data needed to describe a telescope configuation.

Physically, a telescope model consists of a directory hierarchy which holds the data for each station in the telescope. Signals from stations at the root-level of the telescope model are cross-correlated, while elements and sub-stations (in sub-directories) are beam-formed first to generate each station beam.

It is often sufficient to set up the telescope model at the same time as the other settings parameters simply by specifying the input directory (and any other options), but it can sometimes be necessary to set it explicitly.

Example: Overriding element data

For example, to override the some values in the model after it has been loaded:

```
params = {
    # Set up all required simulation parameters here...
    ...
    'telescope': {
        'input_directory': 'telescope.tm'
    }
}
settings = oskar.SettingsTree('oskar_sim_interferometer')
settings.from_dict(params)

# Create the telescope model from the settings parameters.
tel = oskar.Telescope(settings=settings)

# Override element gains.
tel.override_element_gains(mean=1, std=0.03, seed=1)

# Override element cable length errors.
tel.override_element_cable_length_errors(std=0.015)
```

The telescope model can then be set programmatically using `oskar.Interferometer.set_telescope_model(tel)`.

2.1.4 Defining a parameter space and running simulations

An investigation may require a large number of simulations to be carried out in order to explore a parameter space, which typically means that a set of nested loops must be written in order to run all the simulations.

In many cases, only the simulation parameters in the settings tree need to be changed to run a new simulation, but sometimes the sky model and/or telescope model also needs to be changed within a loop. Defining the parameters that need to vary is the first thing to do when writing a new simulation script.

Example: A four-dimensional parameter space

A simple example script which iterates over a 4-dimensional parameter space is shown below. In this case, the observation length (3 values: short, medium, long), the target field (3 values: EoR0, EoR1, EoR2), the ionospheric phase screen (2 values: on, off) and the sky model (2 values: GLEAM and A-team only) were all varied for a total of 36 simulations, and a CASA Measurement Set was written for each case.

Note how nested Python dictionaries are used to define groups of parameters that need to change on each iteration, and the `update()` method is used to merge one dictionary into another. Each dimension is iterated using the general form:

```
# Iterate over a dimension.
for key_name, params_to_update in dictionary.items():

    # Update the current settings dictionary.
    current_settings.update(params_to_update)

    # Iterate over the next dimension...
```

These are all standard Python constructs.

After all the parameters have been set up in the settings tree, an instance of `oskar.Interferometer` is created using it. Finally, calling `oskar.Interferometer.run()` will run each simulation.

```python
#!/usr/bin/env python3
"""
Run simulations for SKA1-LOW direction-dependent effects.
https://confluence.skatelescope.org/display/SE/Simulations+with+Direction-
↪Dependent+Effects
https://jira.skatelescope.org/browse/SIM-489
"""

import copy
import os.path

from astropy.io import fits
import numpy
import oskar

from .utils import get_start_time


def bright_sources():
    """
    Returns a list of bright A-team sources.
    Does not include the Galactic Centre!
    """
    # For A: data from the Molonglo Southern 4 Jy sample (VizieR).
    # Others from GLEAM reference paper, Hurley-Walker et al. (2017), Table 2.
    return numpy.array(
        (
            [
                50.67375,
                -37.20833,
                528,
                0,
                0,
                0,
                178e6,
                -0.51,
                0,
                0,
                0,
                0,
            ],  # For
            [
                201.36667,
                -43.01917,
                1370,
                0,
                0,
                0,
                200e6,
```

(continues on next page)

(continued from previous page)

```
                -0.50,
                0,
                0,
                0,
                0,
            ],  # Cen
            [
                139.52500,
                -12.09556,
                280,
                0,
                0,
                0,
                200e6,
                -0.96,
                0,
                0,
                0,
                0,
            ],  # Hyd
            [
                79.95833,
                -45.77889,
                390,
                0,
                0,
                0,
                200e6,
                -0.99,
                0,
                0,
                0,
                0,
            ],  # Pic
            [
                252.78333,
                4.99250,
                377,
                0,
                0,
                0,
                200e6,
                -1.07,
                0,
                0,
                0,
                0,
            ],  # Her
            [
                187.70417,
                12.39111,
                861,
```

(continues on next page)

(continued from previous page)

```
                0,
                0,
                0,
                200e6,
                -0.86,
                0,
                0,
                0,
                0,
            ],  # Vir
            [
                83.63333,
                22.01444,
                1340,
                0,
                0,
                0,
                200e6,
                -0.22,
                0,
                0,
                0,
                0,
            ],  # Tau
            [
                299.86667,
                40.73389,
                7920,
                0,
                0,
                0,
                200e6,
                -0.78,
                0,
                0,
                0,
                0,
            ],  # Cyg
            [
                350.86667,
                58.81167,
                11900,
                0,
                0,
                0,
                200e6,
                -0.41,
                0,
                0,
                0,
                0,
            ],  # Cas
```

(continues on next page)

(continued from previous page)

```
        )
    )


def main():
    """Main function."""
    # Load GLEAM catalogue data as a sky model.
    sky_dir = "./"
    gleam = fits.getdata(sky_dir + "GLEAM_EGC.fits", 1)
    gleam_sky_array = numpy.column_stack(
        (gleam["RAJ2000"], gleam["DEJ2000"], gleam["peak_flux_wide"])
    )

    # Define common base settings.
    tel_dir = "./"
    tel_model = "SKA1-LOW_SKO-0000422_Rev3_38m_SKALA4_spot_frequencies.tm"
    common_settings = {
        "simulator/max_sources_per_chunk": 65536,
        "simulator/write_status_to_log_file": True,
        "observation/start_frequency_hz": 125e6,  # First channel at 125 MHz.
        "observation/frequency_inc_hz": 5e6,  # Channels spaced every 5 MHz.
        "observation/num_channels": 11,
        "telescope/input_directory": tel_dir + tel_model,
        "interferometer/channel_bandwidth_hz": 100e3,  # 100 kHz-wide channels.
        "interferometer/time_average_sec": 1.0,
        "interferometer/max_time_samples_per_block": 4,
    }

    # Define observations.
    observations = {
        "short": {
            "observation/length": 5 * 60,
            "observation/num_time_steps": 300,
            "telescope/external_tec_screen/input_fits_file": "screen_short_300_1.0.fits",
        },
        "medium": {
            "observation/length": 30 * 60,
            "observation/num_time_steps": 300,
            "telescope/external_tec_screen/input_fits_file": "screen_medium_300_6.0.fits",
        },
        "long": {
            "observation/length": 4 * 60 * 60,
            "observation/num_time_steps": 240,
            "telescope/external_tec_screen/input_fits_file": "screen_long_240_60.0.fits",
        },
    }

    # Define fields.
    fields = {
        "EoR0": {
            "observation/phase_centre_ra_deg": 0.0,
```

(continues on next page)

(continued from previous page)

```
            "observation/phase_centre_dec_deg": -27.0,
        },
        "EoR1": {
            "observation/phase_centre_ra_deg": 60.0,
            "observation/phase_centre_dec_deg": -30.0,
        },
        "EoR2": {
            "observation/phase_centre_ra_deg": 170.0,
            "observation/phase_centre_dec_deg": -10.0,
        },
    }

    # Define ionosphere settings.
    ionosphere = {
        "ionosphere_on": {"telescope/ionosphere_screen_type": "External"},
        "ionosphere_off": {"telescope/ionosphere_screen_type": "None"},
    }

    # Define sky model components.
    sky_models = {
        "A-team": oskar.Sky.from_array(bright_sources()),
        "GLEAM": oskar.Sky.from_array(gleam_sky_array),
    }

    # Loop over observations.
    for obs_name, obs_params in observations.items():
        # Copy the base settings dictionary.
        current_settings = copy.deepcopy(common_settings)

        # Update current settings with observation parameters.
        current_settings.update(obs_params)

        # Loop over fields.
        for field_name, field_params in fields.items():
            # Update current settings with field parameters.
            current_settings.update(field_params)

            # Update current settings with start time.
            ra0_deg = current_settings["observation/phase_centre_ra_deg"]
            length_sec = current_settings["observation/length"]
            start_time = get_start_time(ra0_deg, length_sec)
            current_settings["observation/start_time_utc"] = start_time

            # Loop over ionospheric screen on/off.
            for iono_name, iono_params in ionosphere.items():
                # Update current settings with ionosphere parameters.
                current_settings.update(iono_params)

                # Loop over sky model components.
                for sky_name, sky_model in sky_models.items():
                    # Update output MS filename based on current parameters.
                    ms_name = "SKA_LOW_SIM"
```

(continues on next page)

(continued from previous page)

```
                    ms_name += "_" + obs_name
                    ms_name += "_" + field_name
                    ms_name += "_" + iono_name
                    ms_name += "_" + sky_name
                    ms_name += ".MS"

                    # Check if the MS already exists (if so, skip).
                    if os.path.isdir(ms_name):
                        continue

                    # Create the settings tree.
                    settings = oskar.SettingsTree("oskar_sim_interferometer")
                    settings.from_dict(current_settings)
                    settings["interferometer/ms_filename"] = ms_name

                    # Set up the simulator and run it.
                    sim = oskar.Interferometer(settings=settings)
                    sim.set_sky_model(sky_model)
                    sim.run()


if __name__ == "__main__":
    main()
```

Example: An irregular frequency axis

Frequency channels which are regularly spaced can be run in one go (and written to a single Measurement Set if required) by specifying multiple channels in the settings. However, when running simulations at spot frequencies across a band, these will need to be run separately by explicitly looping over each one. All that is required is to define a list of frequencies and then loop over them, for example:

```
axis_freq_MHz = [50, 70, 110, 137, 160, 230, 320]
for freq_MHz in axis_freq_MHz:
    settings['observation/start_frequency_hz'] = freq_MHz * 1e6
    ...
```

2.1.5 Imaging visibility data sets

Simulated visibilities can be saved to a CASA Measurement Set, so any imager capable of working with Measurement Sets can be used to image them.

For convenience, OSKAR includes an imager which can be used if all that is required is a dirty (or residual) image, and it also provides the option to make images directly from data in numpy arrays. This can often be faster than writing visibilities out to a Measurement Set and loading them back again in order to make an image using another program.

An `oskar.Imager` instance can be created in Python using a settings tree for the `oskar_imager` application. For example:

```
params = {
    'image/fov_deg': 5.0,
    'image/size': 6144,
```

(continues on next page)

(continued from previous page)

```
    'image/algorithm': 'W-projection',  # default is FFT (2D only)
    # The following two options are recommended for large images
    # (particularly when using W-projection),
    # as long as you have enough GPU RAM to hold the complex visibility
    # grid as well as the convolution kernels.
    'image/fft/use_gpu': True,
    'image/fft/grid_on_gpu': True,
    'image/input_vis_data': 'example.vis',  # or 'example.ms'
    'image/root_path': 'example_image'  # Optional: see below
}
settings = oskar.SettingsTree('oskar_imager')
settings.from_dict(params)
imager = oskar.Imager(settings=settings)
```

This will generate an image using the visibility data in the file `example.vis` (or `example.ms` if a Measurement Set is specified instead), and will write a FITS image in Stokes I called `example_image_I.fits`.

The image will be centred on the phase centre used in the observation by default, but it can be re-centred on a different direction by adding the parameters:

```
params = {
    ...  (other parameters here)
    'image/direction': 'RA, Dec.',
    'image/direction/ra_deg': 12.34,  # Insert the required coordinates.
    'image/direction/dec_deg': 56.78
}
```

The full list of settings parameters is shown in the OSKAR GUI for the `oskar_imager` application, and also in the settings documentation.

After setting it up, call `oskar.Imager.run()` to make the image. If required, the image(s) can be returned directly to Python as a numpy array instead of (or as well as) writing a FITS file. Use `return_images=1` as an argument to `oskar.Imager.run()` and assign the return value to a variable. This will be a dictionary of arrays holding the image(s), which can be accessed using the 'images' dictionary key as follows:

```
output = imager.run(return_images=1)
image = output['images'][0]  # Stokes I image.
```

Making dirty or residual images automatically

Many simulation runs need to make either dirty images, or images of residual visibilities. The residuals are generated by subtracting a reference (or model) visibility data set first.

For convenience, the `ResidualImageSimulator` class, described below, can be used to make either dirty or residual images at the same time as running a simulation. It combines the functionality of `oskar.Interferometer` and `oskar.Imager`, so that residual visibilities can be generated as needed and processed on-the-fly as the simulation progresses, without needing to write out visibilities and load them back again to make each image. If generating residuals, only the reference visibilities need to be saved, and multiple subsequent runs can use the same reference data set.

This simulator is configured in the same way as `oskar.Interferometer`, and can optionally be passed an instance of an imager, and a filename containing the reference visibility data. Use it in place of `oskar.Interferometer` if you need to make an image of a simulated data set.

Similar to `oskar.Imager`, any images created using this simulator will be returned directly as numpy arrays from the `run()` method:

```
output = sim.run()
image = output['images'][0]  # Stokes I image.
```

See the notes and example in the class documentation (included below) for usage instructions.

class scripts.**ResidualImageSimulator**(**args: Any*, ***kwargs: Any*)

Interferometer simulator which generates both reference and residual visibilities, optionally imaging them.

This class inherits `oskar.Interferometer`, so it requires the same settings parameters. Each visibility block can be imaged if required in the overridden `process_block()` method.

To generate (and image) residual visibilities, two simulations must be run using separate instances of this class:

1. The first run generates the reference visibility data set, which must be saved to an OSKAR visibility data file.
2. The reference visibilities are then subtracted from the visibilities generated in the second run.

Dirty/residual images of the visibilities can be returned for either run by specifying an imager to use when constructing the simulator.

The following code shows a minimal but complete example of how this class could be used. Note that there are two separate simulators created.

```
import oskar

# Create a 9-by-9 unit-amplitude point-source sky model
# at the phase centre.
# First, define the phase centre coordinates.
ra0_deg = 0
dec0_deg = -30
grid_width_deg = 4.0  # Width of the source grid in degrees.
sky = oskar.Sky.generate_grid(ra0_deg, dec0_deg, 9, grid_width_deg)

# Create and set up an imager to make the residual images.
params_img = {
    'image/fov_deg': grid_width_deg + 0.5,
    'image/size': 6144,
    'image/fft/use_gpu': True
}
settings_img = oskar.SettingsTree('oskar_imager')
settings_img.from_dict(params_img)
imager = oskar.Imager(settings=settings_img)

# Define base parameters for a simulated observation
# using oskar.Interferometer.
obs_length_sec = 4 * 3600.0  # 4 hours, in seconds.
base_params_sim = {
    'observation/start_frequency_hz': 110e6,  # One channel at 110 MHz
    'observation/phase_centre_ra_deg': ra0_deg,
    'observation/phase_centre_dec_deg': dec0_deg,
    'observation/num_time_steps': 24,  # Simulate 24 correlator dumps
    'observation/start_time_utc': get_start_time(ra0_deg, obs_length_sec),
    'observation/length': obs_length_sec,
```

(continues on next page)

(continued from previous page)

```
    'interferometer/channel_bandwidth_hz': 10e3,
    'interferometer/time_average_sec': 1.0,
    # Ignore w-components only if the sky model allows for it,
    # with all sources well within the imaged field of view!
    # (W-smearing will be disabled for all sources.)
    'interferometer/ignore_w_components': True
}
settings_sim = oskar.SettingsTree('oskar_sim_interferometer')
settings_sim.from_dict(base_params_sim)

# Set the parameters for the reference simulation, including a
# reference telescope model, and the output visibility file name.
settings_sim['telescope/input_directory'] = '/path/to/reference_telescope_model_folder.tm'
settings_sim['interferometer/oskar_vis_filename'] = 'reference_data.vis'

# Run the reference simulation.
# No image is made at this point, but visibilities are saved to a file.
sim = ResidualImageSimulator(settings=settings_sim)
sim.set_sky_model(sky)
sim.run()

# Set the parameters for the comparison simulation, including a new
# telescope model. We don't need to save the residual visibilities,
# so the output file name is blank.
settings_sim['telescope/input_directory'] = '/path/to/comparison_telescope_model_folder.tm'
settings_sim['interferometer/oskar_vis_filename'] = ''

# Run the comparison simulation.
# Note that the pre-configured imager and the filename of the
# reference visibility data are both passed in the constructor.
sim = ResidualImageSimulator(
    imager=imager, settings=settings_sim, ref_vis='reference_data.vis')
sim.set_sky_model(sky)

# The residual image(s) is (are) returned by the run() method.
output = sim.run()
image = output['images'][0]  # Stokes I residual image
```

__init__(*imager=None*, *settings=None*, *ref_vis=None*)

Creates the simulator, storing a handle to the imager.

Parameters

- **imager** (*Optional[oskar.Imager]*) – Imager to use.
- **settings** (*Optional[oskar.SettingsTree]*) – Optional settings to use to set up the simulator.
- **ref_vis** (*Optional[str]*) – Pathname of reference visibility file.

finalise()

Called automatically by the base class at the end of run().

process_block(*block*, *block_index*)

Processes the visibility block.

Residual visibilities are generated if appropriate by subtracting the corresponding reference visibilities. The (modified) visibility block is also sent to the imager if one was set, and written to any open visibility data files defined in the settings.

Parameters

- **block** (*oskar.VisBlock*) – A handle to the block to be processed.
- **block_index** (*int*) – The index of the visibility block.

run()

Runs the interferometer simulator and imager, if set.

Any images will be returned in an array accessed by the 'images' dictionary key, for example:

```
output = sim.run()
image = output['images'][0]  # Stokes I image.
```

2.1.6 Making an animation

Sometimes it can be useful to make an animation to check whether a set of simulations worked, or to help give a demo. This section shows an example of how to use matplotlib and the OSKAR imager from within a loop to make each frame of an animation by iterating over time samples in a Measurement Set. The script below could either be used as-is, or adapted to a more complex use case. Each frame is generated by reading slices of visibility data in `Plotter._animate_func`, while the remainder of the script sets up the environment using calls to functions in matplotlib.

The script has the following command-line arguments:

```
usage: animate_ms.py [-h] [--fov_deg FOV_DEG] [--size SIZE] [--fps FPS]
                    [--out OUT] [--title TITLE]
                    MS [MS ...]

Make an animation from one or more Measurement Sets

positional arguments:
MS                  Measurement Set path(s)

optional arguments:
-h, --help          show this help message and exit
--fov_deg FOV_DEG   Field of view to image, in degrees (default: 0.5)
--size SIZE         Image side length, in pixels (default: 256)
--fps FPS           Frames per second in output (default: 10)
--out OUT           Output filename (default: out.mp4)
--title TITLE       Overall figure title (default: )
```

Download `animate_ms.py`:

```
#!/usr/bin/env python3
"""
Generate an animation by stepping through visibility time samples.
"""
import argparse
```

(continues on next page)

(continued from previous page)

```python
import copy

import matplotlib

matplotlib.use("Agg")
# pylint: disable=wrong-import-position
from mpl_toolkits.axes_grid1 import make_axes_locatable
from matplotlib import animation
import matplotlib.pyplot as plt
import numpy
import oskar


# pylint: disable=too-many-instance-attributes
class Plotter:
    """Generate an animation by stepping through visibility time samples."""

    def __init__(self):
        """Constructor."""
        self._artists = ()
        self._axes = None
        self._base_settings = {}
        self._fig = None
        self._ms_list = []
        self._ms_names = []
        self._num_frames = 0
        self._title = ""

    def animate(
        self, imager_settings, ms_names, title="", fps=10, filename="out.mp4"
    ):
        """Function to generate the animation.

        Args:
            imager_settings (dict): Base settings for OSKAR imager.
            ms_names (list[str]): List of Measurement Sets to image.
            title (str): Main figure title.
            fps (int): Frames-per-second.
            filename (str): Name of output MP4 file.
        """
        # Store arguments.
        self._base_settings = imager_settings
        self._ms_names = ms_names
        self._title = title
        self._ms_list.clear()

        # Work out the number of frames to generate.
        num_images = len(self._ms_names)
        self._num_frames = 0
        for i in range(num_images):
            ms = oskar.MeasurementSet.open(self._ms_names[i], readonly=True)
            num_rows = ms.num_rows
```

(continues on next page)

(continued from previous page)

```
            num_stations = ms.num_stations
            num_baselines = (num_stations * (num_stations - 1)) // 2
            self._num_frames = max(self._num_frames, num_rows // num_baselines)
            self._ms_list.append(ms)

        # Create the plot panels.
        num_cols = num_images
        if num_cols > 4:
            num_cols = 4
        num_rows = (num_images + num_cols - 1) // num_cols
        panel_size = 8
        if num_images > 1:
            panel_size = 5
        if num_images > 3:
            panel_size = 4
        fig_size = (num_cols * panel_size, num_rows * panel_size)
        fig, axes = plt.subplots(
            nrows=num_rows, ncols=num_cols, squeeze=False, figsize=fig_size
        )
        self._fig = fig
        self._axes = axes.flatten()

        # Call the animate function.
        anim = animation.FuncAnimation(
            self._fig,
            self._animate_func,
            init_func=self._init_func,
            frames=range(0, self._num_frames),
            interval=1000.0 / fps,
            blit=False,
        )

        # Save animation.
        anim.save(filename, writer="ffmpeg", bitrate=3500)
        plt.close(fig=fig)

    def _init_func(self):
        """Internal initialisation function called by FuncAnimation."""
        # Create an empty image.
        imsize = self._base_settings["image/size"]
        zeros = numpy.zeros((imsize, imsize))
        zeros[0, 0] = 1

        # Create list of matplotlib artists that must be updated each frame.
        artists = []

        # Iterate plot panels.
        for i in range(len(self._axes)):
            ax = self._axes[i]
            im = ax.imshow(zeros, aspect="equal", cmap="gnuplot2")
            divider = make_axes_locatable(ax)
            cax = divider.append_axes("right", size="5%", pad=0.05)
```

(continues on next page)

(continued from previous page)

```
            cbar = plt.colorbar(im, cax=cax)
            ax.invert_yaxis()
            ax.axes.xaxis.set_visible(False)
            ax.axes.yaxis.set_visible(False)
            if i < len(self._ms_names):
                ax.set_title(self._ms_names[i])
            else:
                cbar.set_ticks([])
                cbar.set_ticklabels([])
            artists.append(im)

        # Set figure title.
        self._fig.suptitle(self._title, fontsize=16, y=0.95)

        # Return tuple of artists to update.
        self._artists = tuple(artists)
        return self._artists

    def _animate_func(self, frame):
        """Internal function called per frame by FuncAnimation.

        Args:
            frame (int): Frame index.
        """
        # Iterate plot panels.
        num_panels = len(self._ms_list)
        for i in range(num_panels):
            # Read the visibility meta data.
            freq_start_hz = self._ms_list[i].freq_start_hz
            freq_inc_hz = self._ms_list[i].freq_inc_hz
            num_channels = self._ms_list[i].num_channels
            num_stations = self._ms_list[i].num_stations
            num_rows = self._ms_list[i].num_rows
            num_baselines = (num_stations * (num_stations - 1)) // 2

            # Read the visibility data and coordinates.
            start_row = frame * num_baselines
            if start_row >= num_rows or start_row + num_baselines > num_rows:
                continue
            (u, v, w) = self._ms_list[i].read_coords(start_row, num_baselines)
            vis = self._ms_list[i].read_column(
                "DATA", start_row, num_baselines
            )
            num_pols = vis.shape[-1]

            # Create settings for the imager.
            params = copy.deepcopy(self._base_settings)
            settings = oskar.SettingsTree("oskar_imager")
            settings.from_dict(params)

            # Make the image for this frame.
            print(
```

(continues on next page)

(continued from previous page)

```
                "Generating frame %d/%d, panel %d/%d"
                % (frame + 1, self._num_frames, i + 1, num_panels)
            )
            imager = oskar.Imager(settings=settings)
            imager.set_vis_frequency(freq_start_hz, freq_inc_hz, num_channels)
            imager.update(
                u, v, w, vis, end_channel=num_channels - 1, num_pols=num_pols
            )
            data = imager.finalise(return_images=1)

            # Update the plot panel and colourbar.
            self._artists[i].set_data(data["images"][0])
            self._artists[i].autoscale()


def main():
    """Main function."""
    parser = argparse.ArgumentParser(
        description="Make an animation from one or more Measurement Sets",
        formatter_class=argparse.ArgumentDefaultsHelpFormatter,
    )
    parser.add_argument(
        "ms_names", metavar="MS", nargs="+", help="Measurement Set path(s)"
    )
    parser.add_argument(
        "--fov_deg",
        type=float,
        default=0.5,
        help="Field of view to image, in degrees",
    )
    parser.add_argument(
        "--size", type=int, default=256, help="Image side length, in pixels"
    )
    parser.add_argument(
        "--fps", type=int, default=10, help="Frames per second in output"
    )
    parser.add_argument("--out", default="out.mp4", help="Output filename")
    parser.add_argument("--title", default="", help="Overall figure title")
    args = parser.parse_args()

    # Imager settings.
    imager_settings = {"image/fov_deg": args.fov_deg, "image/size": args.size}

    # Make animation.
    plotter = Plotter()
    plotter.animate(
        imager_settings, args.ms_names, args.title, args.fps, args.out
    )


if __name__ == "__main__":
    main()
```

Example: Single-station drift scan of Galactic plane

As an example, the following OSKAR parameter file will generate a simulated Measurement Set for a 24-hour drift-scan observation of the Galactic plane using a telescope model consisting of a single 38-metre diameter SKA-Low station of 256 isotropic elements.

Download `drift_scan_galaxy.ini`:

```
[General]
app=oskar_sim_interferometer
version=2.8.0

[simulator]
double_precision=false

[sky]
healpix_fits/file=haslam_nside_128.fits
healpix_fits/min_abs_val=30.0

[observation]
mode=Drift scan
start_frequency_hz=1.0e+08
start_time_utc=2000-01-01 09:30:00.0
length=24:00:00.0
num_time_steps=96

[telescope]
input_directory=single_station.tm
pol_mode=Scalar
station_type=Isotropic beam

[interferometer]
ms_filename=drift_scan_galaxy.ms
```

The animation below was then produced by running the `animate_ms.py` script with the following command-line arguments using the output Measurement Set:

```
./animate_ms.py --fov_deg=180 --fps=20 --title="OSKAR drift scan test" --out=drift_scan.
↪mp4 drift_scan_galaxy.ms
```

CHAPTER

THREE

LOW-LEVEL RFI

These provide scripts to simulate data containing propagated radio frequency interference (RFI) from terrestrial antennas.

3.1 Low-level RFI simulations

These simulations are designed to provide simulated data containing received signal from known Australian terrestrial transmitters. The aim of providing these simulations is to enable testing of RFI-mitigation techniques and specifically to understand the level of low-level RFI (radio frequency interference) likely to be present for SKA Low observations and the limitations of the standard mitigation software. This has particular relevance for the Epoch of Re-ionisation (EoR) Key Science Project and was in part motivated by the presence of such RFI in MWA EoR experiments.

There are several scripts provided in the **ska-sim-low/rfi** directory. For the main end-to-end simulation providing output images and measurement sets the bash script `rfi_sim.sh` should be used. This runs three python scripts which in combination will take input transmitter characteristics, calculate the propagation attenuation, the directional beam gain and then simulate observations outputting FITS images or measurement set files as required. Alternatively these scripts can be run individually via the command line. There is an additional script `rfi/power_spectrum.py`, which is not part of the main simulation but can optionally be used to calculate a power spectrum from the FITS images.

For local environments, we recommend running `rfi_sim_test.sh`, which is a version of the original bash script that is scaled to run on a laptop and executes the same three python scripts.

These simulations rely on Pycraf, OSKAR and RASCIL. Please see the relevant documentation for further information.

Details of the inputs required, models used and instructions of how to use the scripts can be found in the links below:

3.1.1 Propagation attenuation with Pycraf

These simulations use the Pycraf module in Python to calculate the propagation attenuation. The relevant pycraf-based scripts can be found in **rfi/pycraf_scripts** directory. Pycraf utilises the International Telecommunications Union (ITU) recommendation framework and specifically those of ITU-R P.452-16, P.676-10 and F.699-7 that describe the assumed models of the antennas and expected propagation effects. For a full description of the relevant models, please see the ITU documentation.

Propagation

The propagation of the transmission can be affected a number of ways including but not limited to, the effects of the terrain/line-of-sight, diffraction, tropospheric scatter and ducting. The Pycraf module utilises the relevant equations described in ITU-R P.452-16 and P.676-10 from the ITU recommendations to calculate the expected attenuation as a result of these factors between transmitter and receiver.

Terrain data

When calculating the propagation attenuation, the script will automatically download the relevant terrain data from the NASA Shuttle Radar Topography Mission (SRTM) provided by the Jet Propulsion Laboratory. This will be stored by default in the **rfi/data/srtm_data** directory.

Receivers

For the purposes of modelling the propagation attenuation and simulating the RFI, we are not using a Pycraf-based model of receivers, but rather they are assumed to be represented by the beam-formed station beam. A line-of-sight gain towards the transmitter is calculated for the station beam with OSKAR and used to correct the final propagation model in the next stage.

Simulation inputs

Transmitters

The Pycraf supporting script as well as the main propagation calculation script (see below) can be run using the default input, which represents the basic information for a single transmitter. For the full RFI simulation a CSV file containing information on multiple transmitters is recommended. Information on the digital television antennas in Western Australia is provided (`Filtered_DTV_list_plain.csv`). This represents a sub-sample of the transmitter information included in the full ACMA (Australian Communications and Media Authority) license list, which also contains more information with regards to each license than is needed for the simulation scripts. Each transmitter in the input CSV should be provided with minimally a name (which will be used alongside the ID to identify the transmitter specific files e.g. attenuation values), a location list (latitude, longitude in degrees), a power [W], a height [m], a central frequency [MHz] and a bandwidth [MHz].

For more information on the transmitter data, see *Terrestrial transmitter data*.

SKA Low configuration

An input configuration file (txt or equivalent) containing position information in longitude and latitude is necessary to run the main script. By default the Low configuration file used can be found in `rfi/data/telescope_files/SKA1-LOW_SKO-0000422_Rev3_38m_SKALA4_spot_frequencies.tm/layout_wgs84.txt`.

Calculate propagation attenuation

Propagation attenuation is calculated using `SKA_low_RFI_propagation.py`. Based on the input information described above, the script will model the transmitters and calculate the expected attenuation at each frequency increment for each requested SKA Low station. Note `SKA_low_RFI_propagation.py` calls `SKA_low_pycraf_propagation.py` to perform the core Pycraf calculation.

The attenuation values will then be used to calculate the apparent power of the emitter as an isotropic antenna would see it.

Use of Az/El calculations

When calculating attenuation values that are expected to be used in conjunction with OSKAR beam-gain values (i.e. if using the `rfi_sim.sh` bash script), it is advisable to use the default setup of `--az_calc=True` and `--non_below_el=True`. Though it is possible for signal to be received from a transmitter that is below the horizon to a given station (predominantly via atmospheric effects), OSKAR relies on a horizon limit and will return a beam gain of zero for any transmitters below the horizon. By default, at this stage a line-of-sight calculation is done to find the position of the transmitter with respect to the Low antennas and any transmitters below the horizon are discarded from the simulation. An updated transmitter file in CSV format is written for use in the RASCIL simulation stage. This prevents the simulation of essentially non-contributing transmitters in the final RASCIL script. (Note: it is possible to perform the RASCIL simulation using an input beam gain file or a single value input, which can be used to include these transmitters in the simulated data.)

HDF5 output

The results of the script are written into an HDF5 file, with the structure described at *Radio Frequency Interference (RFI) interface*. The code will still output all the azimuth-elevation txt files for use in OSKAR. The HDF5 file is the default input for the RASCIL-script. Note: the .hdf5 file can also be the input for the OSKAR-script, but it is not the default behaviour at the moment.

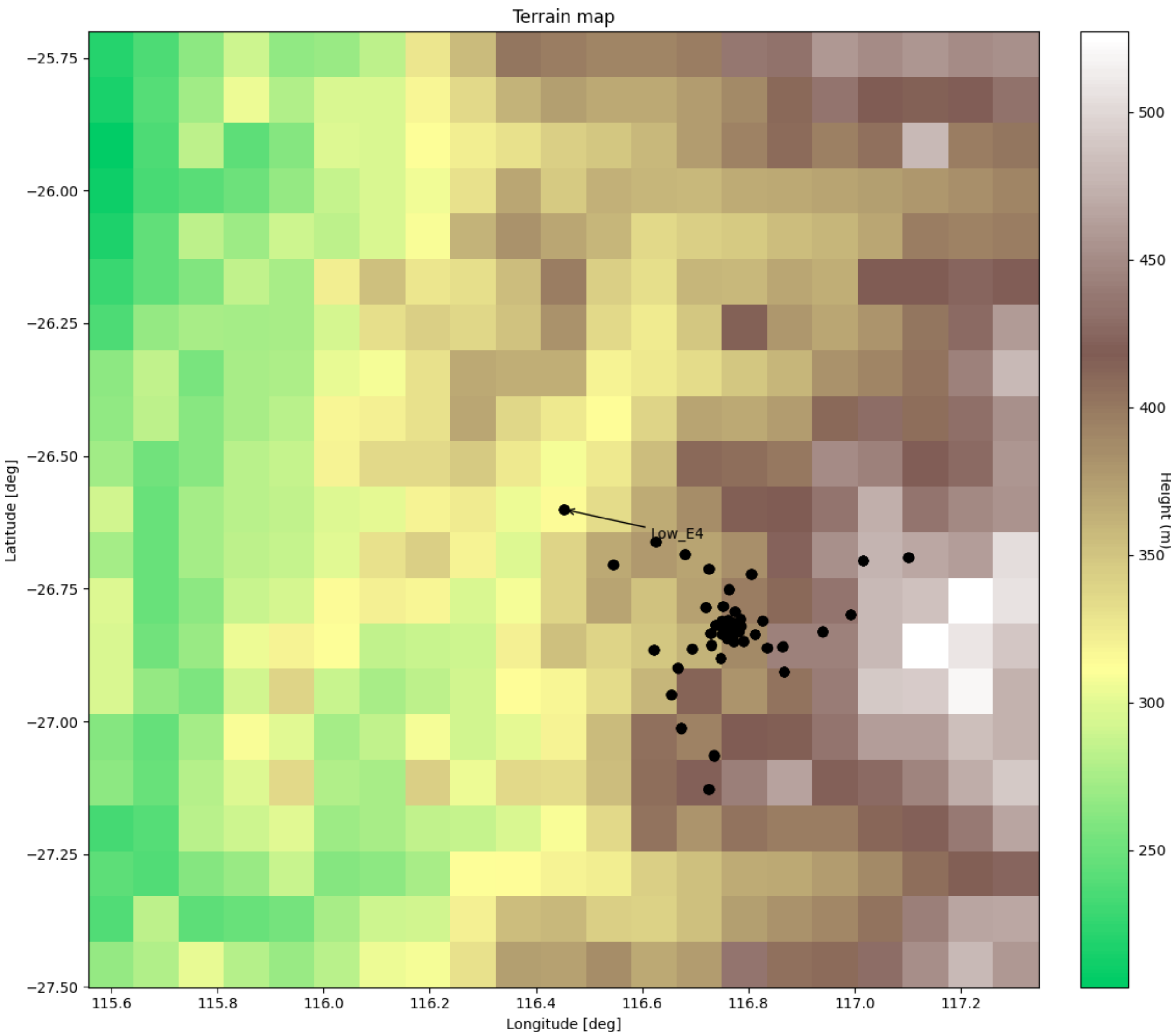
Command line arguments

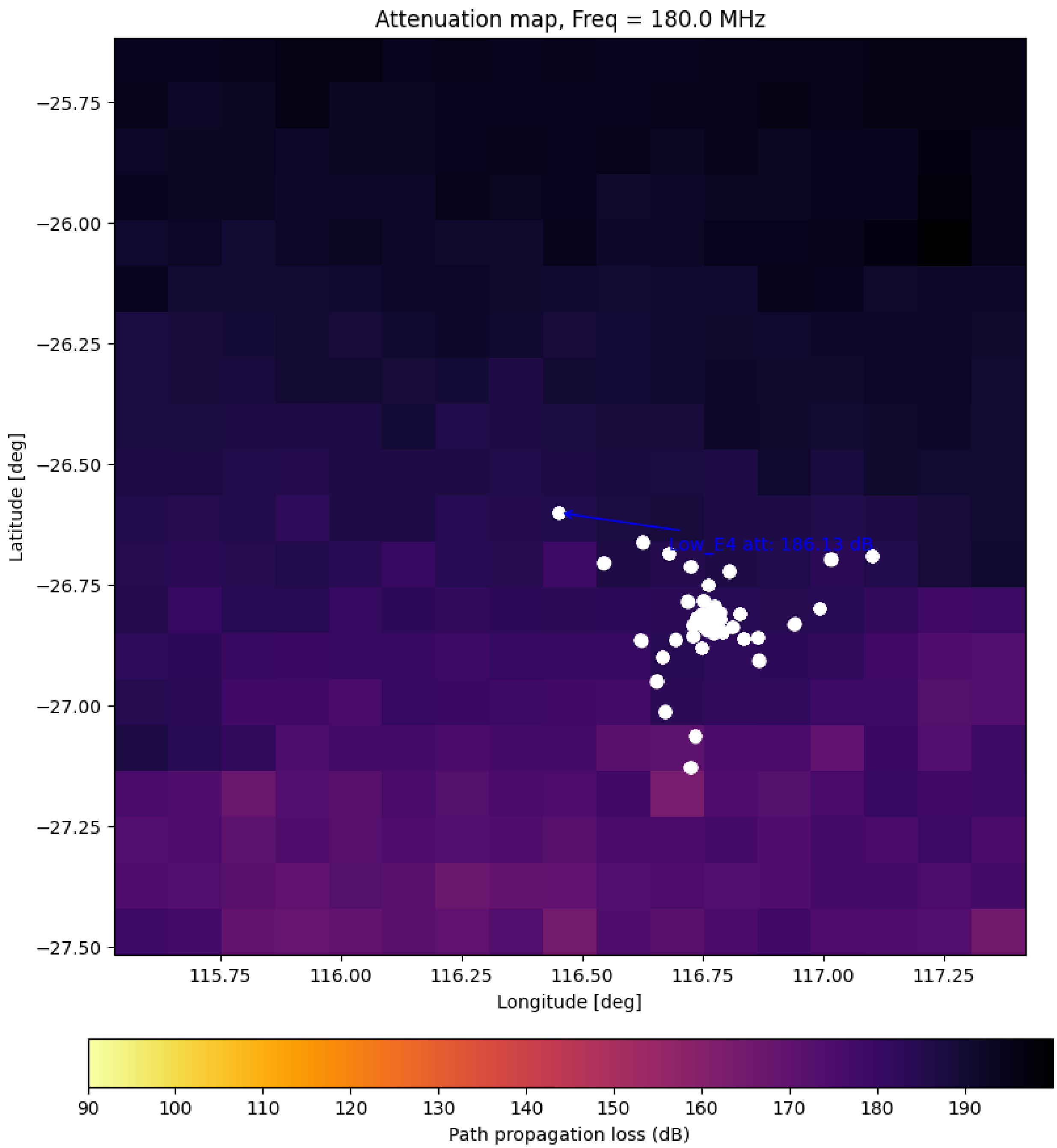
Pycraf Python script

Pycraf supporting scripts

Within the **rfi/pycraf_scripts** directory there is an additional script called `Test_pycraf_LOW_antenna.py`, which is provided to aid with understanding the Pycraf functionality and to enable more comprehensive visualisation of the models and calculations used.

The script is designed to be interactive and the primary input parameters should be easily identifiable within the script itself. This script performs the Pycraf path attenuation calculation for the SKA antennas provided in the SKA Low configuration file (see Simulation inputs for further information). There are additional options to perform and plot a map-based attenuation calculation for visualisation. These can be selected within the code and will produce a map of the relevant SRTM terrain data, a heatmap of the calculated attenuation and a combination of the terrain with overlaid attenuation contours. In each case the positions of the antennas and central array reference point (referred to as 'LOW_E4' in the example given) as well as the transmitter (where apppropriate) will be plotted and labelled. The relevant options (`do_map_solution`, `doplotAll` and `choose_resolution`) can be used to limit the images produced to either a small region around the array or the full distance between the array and transmitter. Example outputs are shown below:


Terrain map
−25.75
−26.00
−26.25
−26.50
−26.75
−27.00
−27.25
−27.50
Latitude [deg]
115.6
115.8
116.0
116.2
116.4
116.6
116.8
117.0
117.2
Longitude [deg]
Low_E4
500
450
400
350
300
250
Height (m)


Attenuation map, Freq = 180.0 MHz
Latitude [deg]
−25.75
−26.00
−26.25
−26.50
−26.75
−27.00
−27.25
−27.50
Longitude [deg]
115.75
116.00
116.25
116.50
116.75
117.00
117.25
Low_E4 att: 186.13 dB
90
100
110
120
130
140
150
160
170
180
190
Path propagation loss (dB)

3.1.2 Beam-gain calculation with OSKAR

The `oskar_sim_beam_gain_sing.py` script is used to run OSKAR to calculate the beam gain in the direction of each transmitter for one or more SKA stations. This can be run in one of two ways, using an installed version of OSKAR or using a containerised (Singularity) version of OSKAR. The python script will determine the version to use, by default trying first the Singularity image, and if that doesn't exist, reverting to the installed version. If the containerised version is used, it should be located in the **rfi** directory. This can be replaced by a newer version as required and can be downloaded from the OSKAR repository.

For further information please see the OSKAR documentation.

Simulation inputs

OSKAR telescope configuration

A telescope configuration file for OSKAR containing SKA Low station information. For full details see the OSKAR documentation. As appropriate those provided with OSKAR can be used and may provide more up-to-date configuration information. By default a copy of a configuration file is used in *rfi/data/telescope_files/SKA1-LOW_SKO-0000422_Rev3_38m_SKALA4_spot_frequencies.tm*.

HDF5 input and output

The script uses the HDF5 output file from *Propagation attenuation with Pycraf* script, with structure explained at *Radio Frequency Interference (RFI) interface*, and write the results into another HDF5 file, which is based on the following class:

class rfi.rfi_interface.rfi_data_cube.**BeamGainDataCube**(*ra: float*, *dec: float*, *obs_time: str*, *freq_chans: ndarray*, *rfi_ids: ndarray*, *nstations: int*)

Data Cube to contain Beam Gain information calculated by OSKAR.

Parameters

- **ra** – right ascension of observed source
- **dec** – declination of observed source
- **obs_time** – time of observation
- **freq_chans** – array of frequency channels
- **rfi_ids** – array of RFI source IDs
- **nstations** – number of SKA stations

property beam_gain

Beam gain value

export_to_hdf5(*filename*)

Save transformed data to HDF5

Parameters

filename – name of output file

This is true, as long as the transmitter HDF5 file name is supplied via the `--input_hdf_file` CLI argument. By default, it is set to *tv_transmitter_attenuation_cube.hdf5*, which is the default output generated by the *Propagation attenuation with Pycraf* script.

When the script is run this way, OSKAR performs calculations per SKA station. The OSKAR output files are only temporarily saved, then read back in so the data can be exported to HDF5. At the end of the run, the temporary files are removed. The HDF5 file also contains pointing information (i.e right ascension and declination), which is used as input for *Simulation of visibility with RASCIL*.

Transmitter data

Alternatively, one can supply an updated transmitter CSV file and individual azimuth-elevation files, created by the *Propagation attenuation with Pycraf* script, via the `--transmitters` and `--indir` CLI arguments. You also have to set the `--input_hdf_file` CLI argument to an empty string (`""`) explicitly, to avoid using the HDF5 set-up.

In this case, OSKAR calculates beam gains for the array centre, given by the single az-el input for each transmitter (instead of a value per transmitter per SKA station). For more information on the transmitter data, follow *Terrestrial transmitter data*.

The python script outputs beam-gain values as a function of frequency as txt files.

Command line arguments

OSKAR Python script

3.1.3 Simulation of visibility with RASCIL

The final stage of the three-stage RFI simulation, `simulate_low_rfi_visibility_propagation.py` uses RASCIL to calculate the visibility measured by SKA-Low (LOW) for a number of emitters, and generate output images or measurement sets. We are interested in the effects of RFI signals that cannot be detected in the visibility data. Therefore, in our simulations we add transmitter apparent power and beam-gain information calculated in the previous stages.

As before, we study the effects of a TV station located in Perth, AU, emitting a broadband signal of a known power (information stored in CSV files in **rfi/data/transmitters**). We presume the following scenario:

The emission from the TV station arrives at LOW stations with phase delay and attenuation. We calculate *Propagation attenuation with Pycraf*. The RFI enters LOW stations in a side-lobe of the station beam. We perform *Beam-gain calculation with OSKAR*, which, together with the pre-calculated apparent power values, is used as an input for the RASCIL script. The RFI enters each LOW station with fixed delay and zero fringe rate (assuming no e.g. ionospheric ducting or reflection from a plane). When tracking a source on the sky, the signal from one station is delayed and fringe-rotated. Fringe rotation stops the fringe from a source at the phase tracking centre but phase-rotates the RFI, which now becomes time-variable. To de-correlate the RFI signal, the correlation data are time- and frequency-averaged over a timescale appropriate for the station field of view.

We want to study the effects of this RFI on statistics of the visibilities, and on images made on source and at the pole. The `simulate_low_rfi_visibility_propagation.py` script averages the data producing baseline-dependent de-correlation and uses RASCIL functions and input data from the previous stages to produce FITS images, and un-averaged MeasurementSets (one per time chunk). The images are on signal channels and on pure noise channels, and for the source of interest. Distributed processing is implemented via Dask.

Simulation inputs

SKA Low configuration

An input configuration file (txt or equivalent, called as the "antenna_file") containing position information in longitude and latitude. The default configuration file (`rfi/data/telescope_files/SKA1-LOW_SKO-0000422_Rev3_38m_SKALA4_spot_frequencies.tm/layout_wgs84.txt`) is used by the code if the `--use_antfile` argument is set to `True`, else it uses the RASCIL equivalent. If `--use_antfile == True`, you can specify an alternative configuration file by setting the `--antenna_file` CLI argument (see *RASCIL Python script*).

Transmitter apparent power

The apparent power of the transmitter is calculated by *Pycraf Python script* and stored in an HDF5 file, together with other relevant information, such as time and station-dependent azimuth and elevation data. For more information, follow *Propagation attenuation with Pycraf*.

Beam gain data

Beam gain values as a function of frequency, calculated by *OSKAR Python script* and stored in an HDF5 file, together with pointing information (i.e. right ascension and declination). For more information, follow *Beam-gain calculation with OSKAR*.

Usage and command line arguments

RASCIL Python script

Power spectrum

The `power_spectrum.py` script can be used following the production of output FITS images from the simulation to produce power spectrum plots.

Usage and command line arguments: *Power Spectrum Python script*

A command line interface is also available to accommodate multiple different RFI sources (at the moment, TV antennas only), and produce a standardized output (in HDF5 format) of Propagation attenuation scripts, which can be consumed by visibility simulations.

3.1.4 Radio Frequency Interference (RFI) interface

Command line tool to standardize the output of RFI attenuation scripts into a format, which can be processed by visibility simulation pipelines. The agreed standard format is HDF5.

The following RFI sources are supported:

```
- TV Transmitter
```

Usage

RFI Interface

Input

The Interface is fully compatible with the Propagation attenuation scripts described in *Propagation attenuation with Pycraf*. At the moment, the following input arguments can be directly modified from the interface:

```
--transmitter_file   path to the CSV file containing the TV transmitter information.
--n_time_chunks      number of time samples to run the simulation for; default = 1.␣
↪Optional

--frequency_range    start and end of frequency range in MHz (specified as <freq_start>␣
↪and <freq_end>). Optional.
--n_channels         number of channels to run the simulation for (specified as <n_
↪channels>). Optional.
```

Note, if any of `<freq_start>, <freq_end>`, or `<n_channels>` is supplied, the other two also needs to be part of the input arguments.

Output

The RFI signal data are saved in an HDF5 file with the following structure:

- Source ID, string, dimensions: (nsources)
- Source type, string, dimensions: (nsources)
- Time samples, string, dimensions: (ntimes)
- Frequency channels, FP64, dimensions: (nfreqs), units: [Hz]
- SKA station ID, string, dimensions: (nstations)
- Apparent source coordinates in antenna rest frame, FP64, dimensions: (nsources, ntimes, nants, 3) These are [azimuth, elevation, distance], units: [degree, degree, m]
- Transmitter power as received by an isotropic antenna, FP64, dimensions: (nsources, ntimes, nants, nfreqs) This does not include the antenna beam pattern which will be applied in the visibility simulation pipeline. units: [dB]

Class description

DataCube

class `rfi.rfi_interface.rfi_data_cube.`**`DataCube`**(*times: list*, *freqs: list*, *station_ids: list*, *rmax=None*, *station_skip=None*)

Class to transform RFI data and save the result in an HDF5 file.

Parameters

- **`times`** – list of time samples the simulation ran for
- **`freqs`** – list of frequency channels the simulation ran for
- **`station_ids`** – list of station ids that were used in the simulation

- `rmax` – maximum distance of SKA station from its array centre
- `station_skip` – . . .

rmax and station_skip are needed for transferring information from the propagation script to the visibility simulation script

`append_data`(*new_rfi_data: DataCubePerSource*)

Append data from a DataCubePerSource object to the existing arrays.

Parameters
`new_rfi_data` – input DataCubePerSource object containing RFI data for a single source

`export_to_hdf5`(*filename*)

Save transformed data to HDF5

Parameters
`filename` – name of output file

`validate_input_data`(*input_data*)

Validate input data.

Data are valid if:

- source_id exists
- time samples of the input match the ones that the DataCube was initialized with
- frequency channels of the input match the ones that the DataCube was initialized with
- station ids of the input match the ones that the DataCube was initialized with

Parameters
`input_data` – input DataCubePerSource object containing RFI data for a single source

Additional information and the list of command line arguments of relevant scripts can be found here:

3.1.5 Supplemental Information

Supplement material to the RFI simulations scripts.

Terrestrial transmitter data

The example RFI simulations focus on digital television (DTV) transmitters and specifically a single antenna located in Perth, AU. However, there are a significant number of DTV as well as other terrestrial transmitters operating in Western Australia. Information has been gathered on the current broadcasting transmitters in Western Australia from several sources including the Australian Communications and Media Authority (ACMA) license register, Oz Digital TV and TX Australia. The ACMA is extensive and, if desired, a full list of transmitters is available from the link above (note the full file size will be several GBs). A CSV copy of the simplified ACMA information for only the Western Australia DTV antennas is included in the **data/transmitters** directory (`Filtered_DTV_list_plain.csv`) which is usable with theses simulations. Other example CSV files are also present in the directory, containing only a handful of transmitters, which can be used for testing. Alternatively, filtering the larger csv file for specific areas or frequencies can provide larger sub-groups to simulate.

The main characteristics of the DTV transmitters have been taken primarily from a copy of the current license register from ACMA, which provides power output, direction of polarisation (H, horizontal or V, vertical) as well as antenna type (e.g. omnidirectional or directional). A number of the transmitter types have beam pattern information available alongside the license information, which contains antenna gains for a number of azimuthal directions. Where no further information is available, the transmitter is assumed to be represented by a fixed-link antenna as described in F.699-7. The map below shows the locations of the SKA-Low stations and the DTV transmitters in Western Australia that transmit in the 50 - 350 MHz range. The map below shows the locations of the SKA-Low stations and the relevant transmitters in Western Australia that transmit in the 50 - 350 MHz range. DTV and DR (digital radio) transmitters are shown by default. FM transmitters can also be shown by opening the map. The colour range displays groups of transmitters based on their emitting power.

3.1.6 CLI

Below can be found the command line interface, usage and command line argument description, of the three main RFI simulation scripts and any additional relevant scripts.

Pycraf Python script

Calculate RFI propagation

```
usage: SKA_low_RFI_propagation.py [-h] [--transmitters TRANSMITTERS]
                                  [--set_freq SET_FREQ] [--freq FREQ]
                                  [--set_bandwidth SET_BANDWIDTH]
                                  [--bandwidth BANDWIDTH]
                                  [--n_channels N_CHANNELS]
                                  [--frequency_range FREQUENCY_RANGE FREQUENCY_RANGE]
                                  [--az_calc AZ_CALC] [--trans_out TRANS_OUT]
                                  [--non_below_el NON_BELOW_EL]
                                  [--srtm_directory SRTM_DIRECTORY]
                                  [--antenna_file ANTENNA_FILE] [--rmax RMAX]
                                  [--station_skip STATION_SKIP]
                                  [--output_dir OUTPUT_DIR]
                                  [--array_centre ARRAY_CENTRE]
                                  [--plot_attenuation PLOT_ATTENUATION]
                                  [--n_time_chunks N_TIME_CHUNKS]
                                  [--frequency_variable FREQUENCY_VARIABLE]
                                  [--time_variable TIME_VARIABLE]
                                  [--omega OMEGA] [--temperature TEMPERATURE]
                                  [--pressure PRESSURE]
                                  [--timepercent TIMEPERCENT]
                                  [--height_rg HEIGHT_RG] [--diam DIAM]
                                  [--zones ZONES] [--hprof_step HPROF_STEP]
```

Named Arguments

--transmitters	Location of input csv file containing transmitter properties.
--set_freq	Choose the central frequency with –freq, otherwise read it from the csv file
--freq	Central frequency (MHz)
--set_bandwidth	Choose the bandwidth with –bandwidth, otherwise read it from the csv file
--bandwidth	Bandwidth (MHz)

--n_channels Number of frequency channels. (Must match nchannels_per_chunk for RFI simulation run)

--frequency_range Frequency range (MHz)

--az_calc Calculate and output the Az/El of the transmitter

--trans_out Name of output transmitter list. File-type not required. If 'infile' will supplement the input name of the input transmitter file. If not full path, it will be written to output_dir directory.

--non_below_el If transmitter elevation to array centre < 0 deg, remove from list and do not simulate.

--srtm_directory Directory for the SRTM files required by pycraf for terrain information.

--antenna_file Location of text files with antenna locations

--rmax Maximum distance of station from centre (m)

--station_skip Decimate stations by this factor

--output_dir Default directory to write attenuation outputs

--array_centre List containing name, latitude (degs), longitude (degs) for the SKA Low array centre

--plot_attenuation Output plot of attenuation values for each transmitter at the array centre.

--n_time_chunks Number of time samples to simulate. (Same as the RASCIL-based part's –nintegrations_per_chunk arg.)

--frequency_variable Simulate frequency-variable RFI signal?

--time_variable Simulate time-variable RFI signal?

--omega Fraction of path over sea. See pycraf documentation.

--temperature Assumed temperature (K). See pycraf documentation.

--pressure Assumed pressure (hPa). See pycraf documentation.

--timepercent Time percent. See pycraf documentation and P.452 report.

--height_rg Assumed height of receiver above ground (m). See pycraf documentation.

--diam Assumed diameter of transmitter (m). See pycraf documentation.

--zones List of clutter types for transmitter and receiver, default is unknown. See pycraf documentation.

--hprof_step Distance resolution of the calculated solution. See pycraf documentation.

OSKAR Python script

Calculate beam gain for SKA1-Low

```
usage: oskar_sim_beam_gain_sing.py [-h] [--ra RA] [--declination DECLINATION]
                                   [--indir INDIR] [--outdir OUTDIR]
                                   [--oskar_path OSKAR_PATH]
                                   [--telescope_path TELESCOPE_PATH]
                                   [--input_hdf_file INPUT_HDF_FILE]
```

(continues on next page)

(continued from previous page)

```
                                   [--transmitters TRANSMITTERS]
                                   [--set_freq SET_FREQ] [--freq FREQ]
                                   [--set_bandwidth SET_BANDWIDTH]
                                   [--bandwidth BANDWIDTH]
                                   [--N_channels N_CHANNELS]
                                   [--frequency_range FREQUENCY_RANGE FREQUENCY_RANGE]
                                   [--choose_range CHOOSE_RANGE]
                                   [--beam_gain_out BEAM_GAIN_OUT]
```

Named Arguments

--ra Right Ascension (deg)

--declination Declination

--indir Directory where transmitter Az_El or HDF5 files are stored

--outdir Directory to store results

--oskar_path Path to the singularity SIF file for OSKAR

--telescope_path Path to telescope model directory

--input_hdf_file HDF5 file located in –indir, which contains necessary coordinate information for each RFI source.If not specified, use individual az/el and transmitter files located in –indir.

--transmitters CSV file containing transmitter properties; not used with HFD data

--set_freq Choose the central frequency with –freq, otherwise read it from the CSV file;not used when input is an HDF5 file.

--freq Central frequency (MHz); not used when input is an HDF5 file.

--set_bandwidth Choose the bandwidth with –bandwidth, otherwise read it from the CSV file; not used when input is an HDF5 file.

--bandwidth Bandwidth (MHz); not used when input is an HDF5 file.

--N_channels Number of frequency channels (must match nchannels_per_chunk for RFI simulation run); not used when input is an HDF5 file.

--frequency_range Frequency range (MHz); not used when input is an HDF5 file.

--choose_range use channels over full frequency range given. If False, default to only over specified bandwidth. If full frequency range larger than bandwidth number of output channels will be those within the bandwidth only. Not used when input is an HDF5 file.

--beam_gain_out Starting name of output beam gain file for each transmitter. Directory and file-type not required. Not used when input is an HDF5 file.

RASCIL Python script

Simulate RFI data with RASCIL

```
usage: simulate_low_rfi_visibility_propagation.py [-h] [--seed SEED]
                                                  [--noise NOISE] [--ra RA]
                                                  [--declination DECLINATION]
                                                  [--nchannels_per_chunk NCHANNELS_PER_
↪CHUNK]
                                                  [--channel_average CHANNEL_AVERAGE]
                                                  [--frequency_range FREQUENCY_RANGE␣
↪FREQUENCY_RANGE]
                                                  [--time_average TIME_AVERAGE]
                                                  [--integration_time INTEGRATION_TIME]
                                                  [--time_range TIME_RANGE TIME_RANGE]
                                                  [--input_file INPUT_FILE]
                                                  [--use_beamgain USE_BEAMGAIN]
                                                  [--beamgain_hdf_file BEAMGAIN_HDF_FILE]
                                                  [--beamgain_dir BEAMGAIN_DIR]
                                                  [--use_antfile USE_ANTFILE]
                                                  [--antenna_file ANTENNA_FILE]
                                                  [--write_ms WRITE_MS]
                                                  [--msout MSOUT]
                                                  [--output_dir OUTPUT_DIR]
                                                  [--use_dask USE_DASK]
```

Named Arguments

--seed Random number seed

--noise Add random noise to the visibility samples?

--ra Right Ascension (degrees)

--declination Declination (degrees)

--nchannels_per_chunk Number of channels in a chunk

--channel_average Number of channels in a chunk to average

--frequency_range Frequency range (Hz)

--time_average Number of integrations in a chunk to average

--integration_time Integration time (s)

--time_range Hourangle range (hours)

--input_file Full path to the HDF5 file, which contains necessary RFI information for each RFI source.

--use_beamgain Use beam gain values in calculation

--beamgain_hdf_file HDF5 file with beam gain, transmitter, frequency, and pointing (RA, DEC) information.

--beamgain_dir Folder containing multiple Numpy files or the HDF file with beam gain information.

--use_antfile Use the antenna file in the rfi data in calculation, otherwise use from RASCIL

--antenna_file txt file containing antenna locations

--write_ms Write measurement set?

--msout Name for MeasurementSet

--output_dir Output directory for storing files

--use_dask Use dask to distribute processing?

Power Spectrum Python script

Display power spectrum of image

```
usage: power_spectrum.py [-h] [--image IMAGE]
                         [--signal_channel SIGNAL_CHANNEL]
                         [--noise_channel NOISE_CHANNEL]
                         [--resolution RESOLUTION]
```

Named Arguments

--image Image name

--signal_channel Channel containing both signal and noise

--noise_channel Channel containing noise only

--resolution Resolution in radians needed for conversion to K

RFI Interface

```
Usage:
    rfi_source_signal_interface.py tv_antenna --transmitters=<transmitter-csv> [<n_
↪channels> <freq_start> <freq_end>]
    rfi_source_signal_interface.py aircraft
    rfi_source_signal_interface.py (-h | --help)

Arguments:
    # if tv_antenna
    --transmitters=<transmitter-csv>      Location of input CSV file containing TV␣
↪transmitter properties

Options:
    -h --help           Show this screen.

    # if any of the following is provided, all three has to be provided as a CLI argument
    <n_channels>        Number of frequency channels. Default: 3
    <freq_start>        Start of Frequency range [MHz]. Default: 170.5
    <freq_end>          End of Frequency range [MHz]. Default: 184.5
```

INDEX

Symbols

__init__() (*scripts.ResidualImageSimulator method*), 21

A

append_data() (*rfi.rfi_interface.rfi_data_cube.DataCube method*), 38

B

beam_gain (*rfi.rfi_interface.rfi_data_cube.BeamGainDataCube property*), 34
BeamGainDataCube (*class in rfi.rfi_interface.rfi_data_cube*), 34

D

DataCube (*class in rfi.rfi_interface.rfi_data_cube*), 37

E

export_to_hdf5() (*rfi.rfi_interface.rfi_data_cube.BeamGainDataCube method*), 34
export_to_hdf5() (*rfi.rfi_interface.rfi_data_cube.DataCube method*), 38

F

finalise() (*scripts.ResidualImageSimulator method*), 21

P

process_block() (*scripts.ResidualImageSimulator method*), 21

R

ResidualImageSimulator (*class in scripts*), 20
run() (*scripts.ResidualImageSimulator method*), 22

V

validate_input_data() (*rfi.rfi_interface.rfi_data_cube.DataCube method*), 38