hera_sim Documentation

Release 4.2.3.dev51+g57cde9b

HERA-Team

May 14, 2024

CONTENTS

Basic simulation package for HERA-like redundant interferometric arrays.

CHAPTER

ONE

FEATURES

- **Systematic Models:** Many models of instrumental systematics in various forms, eg. thermal noise, RFI, band-pass gains, cross-talk, cable reflections and foregrounds.
- **HERA-tuned:** All models have defaults tuned to HERA, with various default "sets" available (eg.H1C, H2C)
- **Interoperability:** Interoperability with `pyuvdata` datasets and `pyuvsim` configurations.
- **Ease-of-use:** High-level interface for adding multiple systematics to existing visibilities in a self-consistent way.
- **Visibility Simulation:** A high-level interface for visbility simulation that is compatible with the configuration definition from `pyuvsim` but is able to call multiple simulator implementations.
- **Convenience:** Methods for adjusting simulated data to match the times/baselines of a reference dataset.

CHAPTER

TWO

DOCUMENTATION

At ReadTheDocs. In particular, for a tutorial and overview of available features, check out the tour.

CHAPTER

THREE

INSTALLATION

3.1 Conda users

If you are using conda, the following command will install all dependencies which it can handle natively:

```
$ conda install -c conda-forge numpy scipy pyuvdata attrs h5py healpy pyyaml
```

If you are creating a new development environment, consider using the included environment file:

```
$ conda env create -f ci/tests.yaml
```

This will create a fresh environment with all required dependencies, as well as those required for testing. Then follow the pip-only instructions below to install `hera_sim` itself.

3.2 Pip-only install

Simply use `pip install -e .` or run `pip install git+git://github.com/HERA-Team/hera_sim`.

3.3 Developer install

For a development install (tests and documentation), run `pip install -e .[dev]`.

Other optional extras can be installed as well. To use baseline-dependent averaging functionality, install the extra `[bda]`. For the ability to simulate redundant gains, install `[cal]`. To enable GPU functionality on some of the methods (especially visibility simulators), install `[gpu]`.

As the repository is becoming quite large, you may also wish to perform a shallow clone to retrieve only the recent commits and history. This makes the clone faster and avoid bottleneck in CI pipelines.

Provide an argument `--depth 1` to the `git clone` command to copy only the latest revision of the repository.

```
git clone --depth [depth] git@github.com:HERA-Team/hera_sim.git
```

CHAPTER

FOUR

VERSIONING

We use semantic versioning (`major.minor.patch`) for the `hera_sim` package (see SemVer documentation). To briefly summarize, new `major` versions include API-breaking changes, new `minor` versions add new features in a backwards-compatible way, and new `patch` versions implement backwards-compatible bug fixes.

CHAPTER

FIVE

CONTENTS

5.1 Tutorials and FAQs

The following introductory tutorial will help you get started with `hera_sim`:

5.1.1 Tour of hera_sim

This notebook briefly introduces some of the effects that can be modeled with `hera_sim`.

[1]:
```
%matplotlib notebook
import aipy, uvtools
import numpy as np
import pylab as plt
```

[2]:
```
from hera_sim import foregrounds, noise, sigchain, rfi
```

```
/home/bobby/HERA/dev/fix_tutorial/hera_sim/hera_sim/visibilities/__init__.py:27:␣
↪UserWarning: PRISim failed to import.
  warnings.warn("PRISim failed to import.")
/home/bobby/HERA/dev/fix_tutorial/hera_sim/hera_sim/visibilities/__init__.py:33:␣
↪UserWarning: VisGPU failed to import.
  warnings.warn("VisGPU failed to import.")
/home/bobby/HERA/dev/fix_tutorial/hera_sim/hera_sim/__init__.py:36: FutureWarning:
In the next major release, all HERA-specific variables will be removed from the codebase.
↪ The following variables will need to be accessed through new class-like structures to␣
↪be introduced in the next major release:

noise.HERA_Tsky_mdl
noise.HERA_BEAM_POLY
sigchain.HERA_NRAO_BANDPASS
rfi.HERA_RFI_STATIONS

Additionally, the next major release will involve modifications to the package's API,␣
↪which move toward a regularization of the way in which hera_sim methods are interfaced␣
↪with; in particular, changes will be made such that the Simulator class is the most␣
↪intuitive way of interfacing with the hera_sim package features.
  FutureWarning)
```

[3]:
```
fqs = np.linspace(.1,.2,1024,endpoint=False)
lsts = np.linspace(0,2*np.pi,10000, endpoint=False)
```

(continues on next page)

(continued from previous page)

```
times = lsts / (2*np.pi) * aipy.const.sidereal_day
bl_len_ns = np.array([30., 0, 0])
```

Foregrounds

Diffuse Foregrounds

[4]:
```
Tsky_mdl = noise.HERA_Tsky_mdl['xx']
vis_fg_diffuse = foregrounds.diffuse_foreground(lsts, fqs, bl_len_ns, Tsky_mdl)
```

[5]:
```
MX, DRNG = 2.5, 3
plt.figure()
plt.subplot(211); uvtools.plot.waterfall(vis_fg_diffuse, mode='log', mx=MX, drng=DRNG);␣
↪plt.colorbar(); plt.ylim(0,4000)
plt.subplot(212); uvtools.plot.waterfall(vis_fg_diffuse, mode='phs'); plt.colorbar();␣
↪plt.ylim(0,4000)
plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

Point-Source Foregrounds

[6]:
```
vis_fg_pntsrc = foregrounds.pntsrc_foreground(lsts, fqs, bl_len_ns, nsrcs=200)
```

[7]:
```
plt.figure()
plt.subplot(211); uvtools.plot.waterfall(vis_fg_pntsrc, mode='log', mx=MX, drng=DRNG);␣
↪plt.colorbar()#; plt.ylim(0,4000)
plt.subplot(212); uvtools.plot.waterfall(vis_fg_pntsrc, mode='phs'); plt.colorbar(); plt.
↪ylim(0,4000)
plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
/home/bobby/anaconda3/envs/fix_tutorial/lib/python3.7/site-packages/uvtools/plot.py:40:␣
↪RuntimeWarning: divide by zero encountered in log10
  data = np.log10(data)
```

Diffuse and Point-Source Foregrounds

[8]:
```
vis_fg = vis_fg_diffuse + vis_fg_pntsrc
```

[9]:
```
plt.figure()
plt.subplot(211); uvtools.plot.waterfall(vis_fg, mode='log', mx=MX, drng=DRNG); plt.
↪colorbar(); plt.ylim(0,4000)
plt.subplot(212); uvtools.plot.waterfall(vis_fg, mode='phs'); plt.colorbar(); plt.ylim(0,
↪4000)
plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

Noise

[10]:
```
tsky = noise.resample_Tsky(fqs,lsts,Tsky_mdl=noise.HERA_Tsky_mdl['xx'])
t_rx = 150.
omega_p = noise.bm_poly_to_omega_p(fqs)
nos_jy = noise.sky_noise_jy(tsky + t_rx, fqs, lsts, omega_p)
```

[11]:
```
plt.figure()
plt.subplot(211); uvtools.plot.waterfall(nos_jy, mode='log', mx=MX, drng=DRNG); plt.
↪colorbar()#; plt.ylim(0,4000)
plt.subplot(212); uvtools.plot.waterfall(nos_jy, mode='phs'); plt.colorbar()#; plt.
↪ylim(0,4000)
plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[12]:
```
vis_fg_nos = vis_fg + nos_jy
```

[13]:
```
plt.figure()
plt.subplot(211); uvtools.plot.waterfall(vis_fg_nos, mode='log', mx=MX, drng=DRNG); plt.
↪colorbar(); plt.ylim(0,4000)
plt.subplot(212); uvtools.plot.waterfall(vis_fg_nos, mode='phs'); plt.colorbar(); plt.
↪ylim(0,4000)
plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

RFI

[14]:
```
rfi1 = rfi.rfi_stations(fqs, lsts)
rfi2 = rfi.rfi_impulse(fqs, lsts, chance=.02)
rfi3 = rfi.rfi_scatter(fqs, lsts, chance=.001)
rfi_all = rfi1 + rfi2 + rfi3
```

[15]:
```
plt.figure()
plt.subplot(211); uvtools.plot.waterfall(rfi_all, mode='log', mx=MX, drng=DRNG); plt.
↪colorbar(); plt.ylim(0,4000)
plt.subplot(212); uvtools.plot.waterfall(rfi_all, mode='phs'); plt.colorbar(); plt.
↪ylim(0,4000)
plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[16]:
```
vis_fg_nos_rfi = vis_fg_nos + rfi_all
```

[17]:
```
plt.figure()
plt.subplot(211); uvtools.plot.waterfall(vis_fg_nos_rfi, mode='log', mx=MX, drng=DRNG);␣
↪plt.colorbar(); plt.ylim(0,4000)
plt.subplot(212); uvtools.plot.waterfall(vis_fg_nos_rfi, mode='phs'); plt.colorbar();␣
↪plt.ylim(0,4000)
plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

Gains

[18]:
```
g = sigchain.gen_gains(fqs, [1,2,3])
plt.figure()
for i in g: plt.plot(fqs, np.abs(g[i]), label=str(i))
plt.legend(); plt.show()
gainscale = np.average([np.median(np.abs(g[i])) for i in g])
MXG = MX + np.log10(gainscale)
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[19]:
```
vis_total = sigchain.apply_gains(vis_fg_nos_rfi, g, (1,2))
plt.figure()
plt.subplot(211); uvtools.plot.waterfall(vis_total, mode='log', mx=MXG, drng=DRNG); plt.
↪colorbar(); plt.ylim(0,4000)
plt.subplot(212); uvtools.plot.waterfall(vis_total, mode='phs'); plt.colorbar(); plt.
↪ylim(0,4000)
plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

Crosstalk

[20]:
```
xtalk = sigchain.gen_whitenoise_xtalk(fqs)
vis_xtalk = sigchain.apply_xtalk(vis_fg_nos_rfi, xtalk)
vis_xtalk = sigchain.apply_gains(vis_xtalk, g, (1,2))
plt.figure()
plt.subplot(211); uvtools.plot.waterfall(vis_xtalk, mode='log', mx=MXG, drng=DRNG); plt.
↪colorbar(); plt.ylim(0,4000)
plt.subplot(212); uvtools.plot.waterfall(vis_xtalk, mode='phs'); plt.colorbar(); plt.
↪ylim(0,4000)
plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[ ]:
```
```

The following tutorial will help you learn how use the `Simulator` class:

5.1.2 The Simulator Class

The most convenient way to use `hera_sim` is to use the `Simulator` class, which builds in all the primary functionality of the `hera_sim` package in an easy-to-use interface, and adds the ability to consistently write all produced effects into a `pyuvdata.UVData` object (and to file).

This notebook provides a brief tutorial on basic use of the `Simulator` class, followed by a longer, more in-depth tutorial that shows some advanced features of the class.

Setup

[1]:
```
import tempfile
import time
from pathlib import Path

import matplotlib.pyplot as plt
import numpy as np
from astropy import units

import hera_sim
from hera_sim import Simulator, DATA_PATH, utils
from uvtools.plot import labeled_waterfall
```

We'll inspect the visibilities as we go along by plotting the amplitudes and phases on different axes in the same figure. Here's the function we'll be using:

[2]:
```
def waterfall(sim, antpairpol=(0,1,"xx"), figsize=(6,3.5), dpi=200, title=None):
    """Convenient plotting function to show amp/phase."""
    fig, (ax1, ax2) = plt.subplots(
```

(continues on next page)

(continued from previous page)

```
        nrows=2,
        ncols=1,
        figsize=figsize,
        dpi=dpi,
    )
    fig, ax1 = labeled_waterfall(
        sim.data,
        antpairpol=antpairpol,
        mode="log",
        ax=ax1,
        set_title=title,
    )
    ax1.set_xlabel(None)
    ax1.set_xticklabels(['' for tick in ax1.get_xticks()])
    fig, ax2 = labeled_waterfall(
        sim.data,
        antpairpol=antpairpol,
        mode="phs",
        ax=ax2,
        set_title=False,
    )
    return fig
```

Introduction

The `Simulator` class provides an easy-to-use interface for all of the features provided by `hera_sim`, though it is not to be confused with the `VisibilitySimulator` class, which is intended to be a universal interface for high-accuracy visibility simulators commonly used within the community. The `Simulator` class features:

- A universal `add` method for simulating any effect included in the `hera_sim` API.
  - More advanced users can create their own custom effects that the `Simulator` can use, as long as these custom effects abide by a certain syntax.
- A `get` method for retrieving any previously simulated effect.
  - Simulated effects are not cached, but rather the parameters characterizing the effect are recorded and the desired effect is re-simulated.
- Four modes of specifying how to configure the random state when simulating effects:
  - "redundant" ensures that the random state is set the same for each baseline within a redundant group;
  - "once" uses the same random state when simulating data for every baseline in the array;
  - "initial" only sets the random state initially, prior to simulating data for the array;
  - Similar to "once", the user can provide an integer to seed the random state, which is then used to set the random state for every baseline in the array.
- Convenient methods for writing the data to disk.
  - The `write` method writes the entire `UVData` object's contents to disk in the desired format.
  - The `chunk_sim_and_save` method allows for writing the data to disk in many files with a set number of integrations per file.

In order to enable a simple, single interface for simulating an effect, `hera_sim` employs a certain heirarchy to how it thinks about simulated effects. Every simulation effect provided in `hera_sim` derives from a single abstract base class, the `SimulationComponent`. A "component" is a general term for a category of simulated effect, while a "model" is an implementation of a particular effect. Components track all of the models that are particular implementations of the components, and this enables `hera_sim` to track every model that is created, so long as it is a subclass of a simulation component. To see a nicely formatted list of all of the components and their models, you can print the output of the `components.list_all_components` function, as demonstrated below.

[3]:
```
print(hera_sim.components.list_all_components())
```

```
array:
  lineararray
  hexarray
foreground:
  diffuseforeground | diffuse_foreground
  pointsourceforeground | pntsrc_foreground
noise:
  thermalnoise | thermal_noise
rfi:
  stations | rfi_stations
  impulse | rfi_impulse
  scatter | rfi_scatter
  dtv | rfi_dtv
gain:
  bandpass | gains | bandpass_gain
  reflections | reflection_gains | sigchain_reflections
crosstalk:
  crosscouplingcrosstalk | cross_coupling_xtalk
  crosscouplingspectrum | cross_coupling_spectrum | xtalk_spectrum
  whitenoisecrosstalk | whitenoise_xtalk | white_noise_xtalk
eor:
  noiselikeeor | noiselike_eor
```

The format for the output is structured as follows:

```
component_1:
  model_1_name | model_1_alias_1 | ... | model_1_alias_N
  ...
  model_N_name | model_N_alias_1 | ...
component_2:
  ...
```

So long as there are not name conflicts, `hera_sim` is able to uniquely identify which model corresponds to which name/alias, and so there is some flexibility in telling the `Simulator` how to find and simulate an effect for a particular model.

Basic Use

To get us started, let's make a `Simulator` object with 100 frequency channels spanning from 100 to 200 MHz, a half-hour of observing time using an integration time of 10.7 seconds, and a 7-element hexagonal array.

[4]:
```
# Define the array layout.
array_layout = hera_sim.antpos.hex_array(
    2, split_core=False, outriggers=0
)

# Define the timing parameters.
start_time = 2458115.9  # JD
integration_time = 10.7  # seconds
Ntimes = int(30 * units.min.to("s") / integration_time)

# Define the frequency parameters.
Nfreqs = 100
bandwidth = 1e8  # Hz
start_freq = 1e8  # Hz

sim_params = dict(
    Nfreqs=Nfreqs,
    start_freq=start_freq,
    bandwidth=bandwidth,
    Ntimes=Ntimes,
    start_time=start_time,
    integration_time=integration_time,
    array_layout=array_layout,
)

# Create an instance of the Simulator class.
sim = Simulator(**sim_params)
```

Overview

The `Simulator` class adds some attributes for conveniently accessing metadata:

[5]:
```
# Observed frequencies in GHz
sim.freqs[::10]
```

[5]:
```
array([0.1 , 0.11, 0.12, 0.13, 0.14, 0.15, 0.16, 0.17, 0.18, 0.19])
```

[6]:
```
# Observed Local Sidereal Times (LSTs) in radians
sim.lsts[::10]
```

[6]:
```
array([4.58108965, 4.58889222, 4.59669478, 4.60449734, 4.61229991,
       4.62010247, 4.62790503, 4.6357076 , 4.64351016, 4.65131273,
       4.65911529, 4.66691785, 4.67472042, 4.68252298, 4.69032555,
       4.69812811, 4.70593067])
```

[7]:
```
# Array layout in local East-North-Up (ENU) coordinates
sim.antpos
```

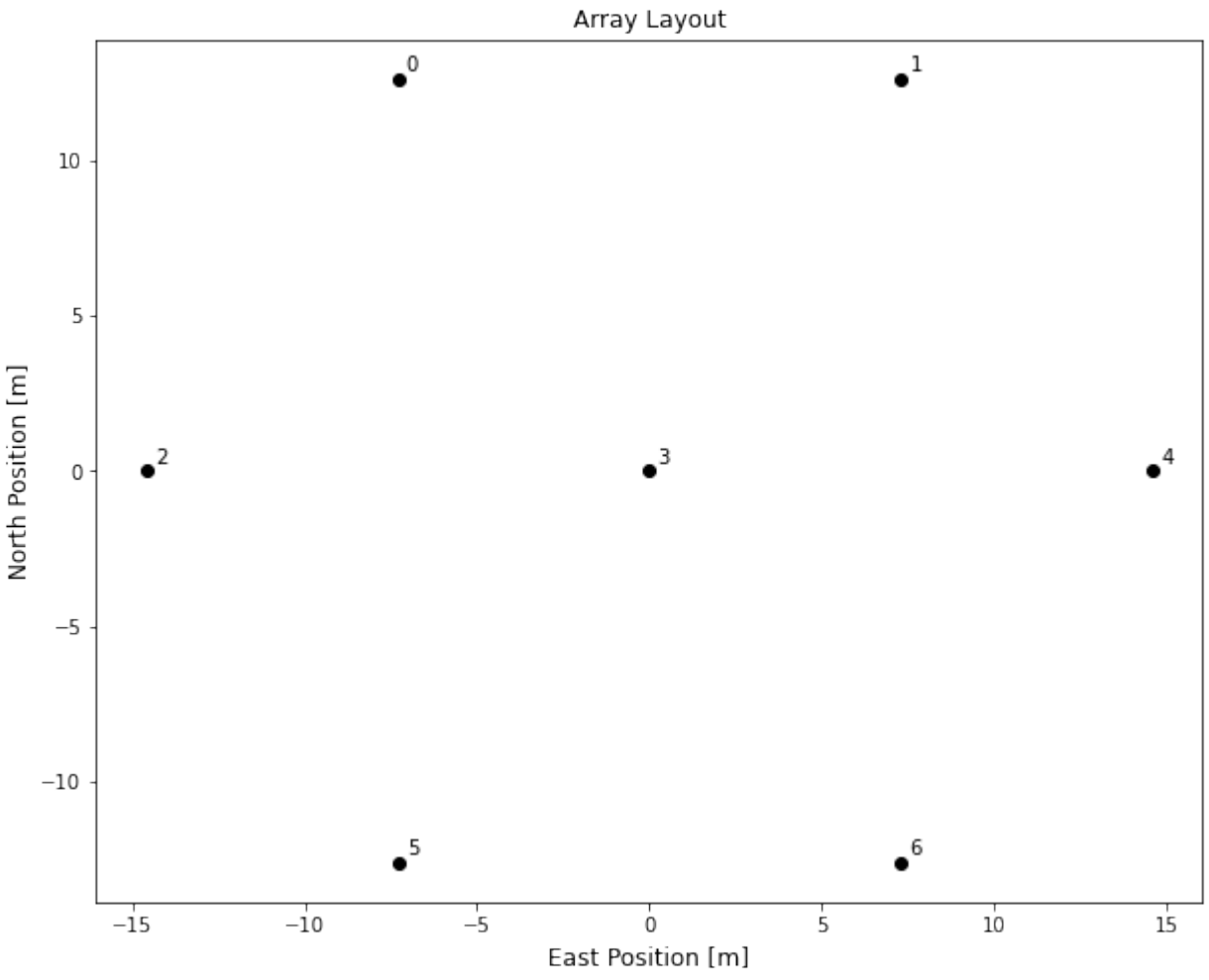
[7]:
```
{0: array([-7.30000000e+00,  1.26439709e+01, -4.36185665e-09]),
 1: array([ 7.30000000e+00,  1.26439709e+01, -3.99203159e-09]),
 2: array([-1.46000000e+01,  6.98581573e-09, -4.65185394e-09]),
 3: array([ 0.00000000e+00,  7.20559015e-09, -4.28202888e-09]),
 4: array([ 1.46000000e+01,  7.42536457e-09, -3.91220382e-09]),
 5: array([-7.30000000e+00, -1.26439709e+01, -4.57202631e-09]),
 6: array([ 7.30000000e+00, -1.26439709e+01, -4.20220125e-09])}
```

[8]:
```
# Polarization array
sim.pols
```

[8]:
```
['xx']
```

You can also generate a plot of the array layout using the `plot_array` method:

[9]:
```
fig = sim.plot_array()
```



The `data` attribute can be used to access the `UVData` object used to store the simulated data and metadata:

[10]:
```
type(sim.data)
```

[10]:
```
pyuvdata.uvdata.uvdata.UVData
```

Adding Effects

Effects may be added to a simulation by using the `add` method. This method takes one argument and variable keyword arguments: the required argument `component` may be either a string identifying the name of a `hera_sim` class (or an alias thereof, see below), or an appropriately defined callable class (see the section on using custom classes for exact details), while the variable keyword arguments parametrize the model. The `add` method simulates the effect and applies it to the entire array. Let's walk through an example.

We'll start by simulating diffuse foreground emission. For simplicity, we'll be using the H1C season default settings so that the number of extra parameters we need to specify is minimal.

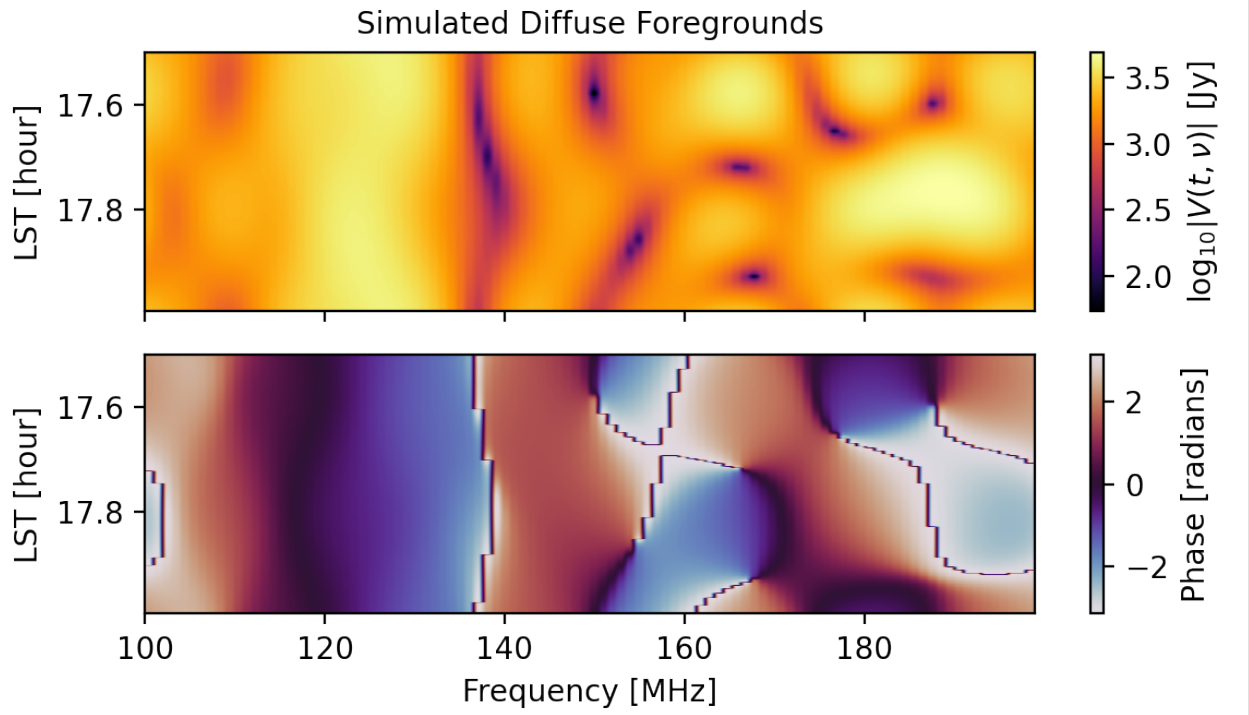
[11]:
```
# Use H1C season defaults.
hera_sim.defaults.set("h1c")
```

[12]:
```
# Start off by simulating some foreground emission.
sim.add("diffuse_foreground")
```

```
You have not specified how to seed the random state. This effect might not be exactly␣
↪recoverable.
```

[13]:
```
# Let's check out the data for the (0,1) baseline.
fig = waterfall(sim, title="Simulated Diffuse Foregrounds")
fig.tight_layout()
```

```
FixedFormatter should only be used together with FixedLocator
```



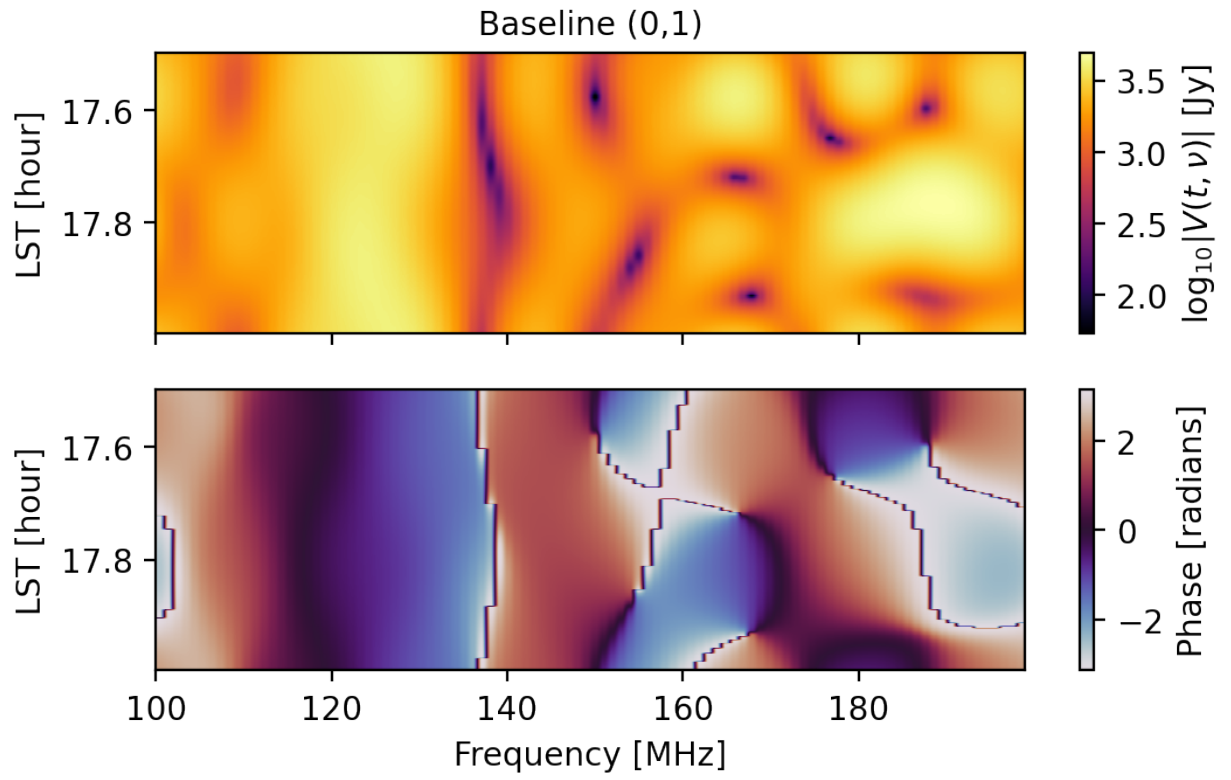
That was simple enough; however, basic use like this does not ensure that the simulation is as realistic as it can be with the tools provided by `hera_sim`. For example, data that should be redundant is not redundant by default:

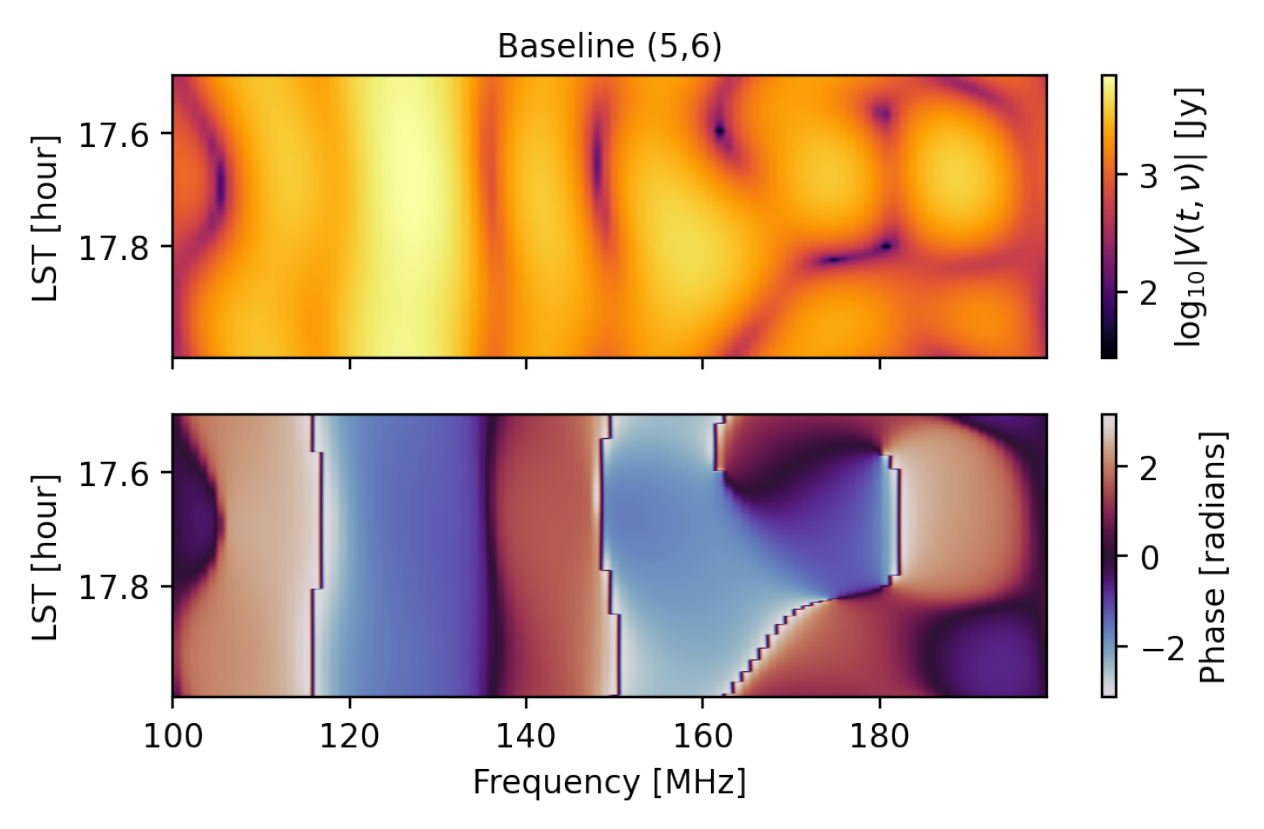
[14]:
```
# (0,1) and (5,6) are redundant baselines, but...
np.all(sim.get_data(0,1,"xx") == sim.get_data(5,6,"xx"))
```

[14]:
```
False
```

[15]:
```
# As an extra comparison, let's plot them
fig1 = waterfall(sim, antpairpol=(0,1,"xx"), title="Baseline (0,1)")
fig2 = waterfall(sim, antpairpol=(5,6,"xx"), title="Baseline (5,6)")
```

```
FixedFormatter should only be used together with FixedLocator
```





We can ensure that a simulated effect that should be redundant is redundant by specifying that we use a "redundant" seed:

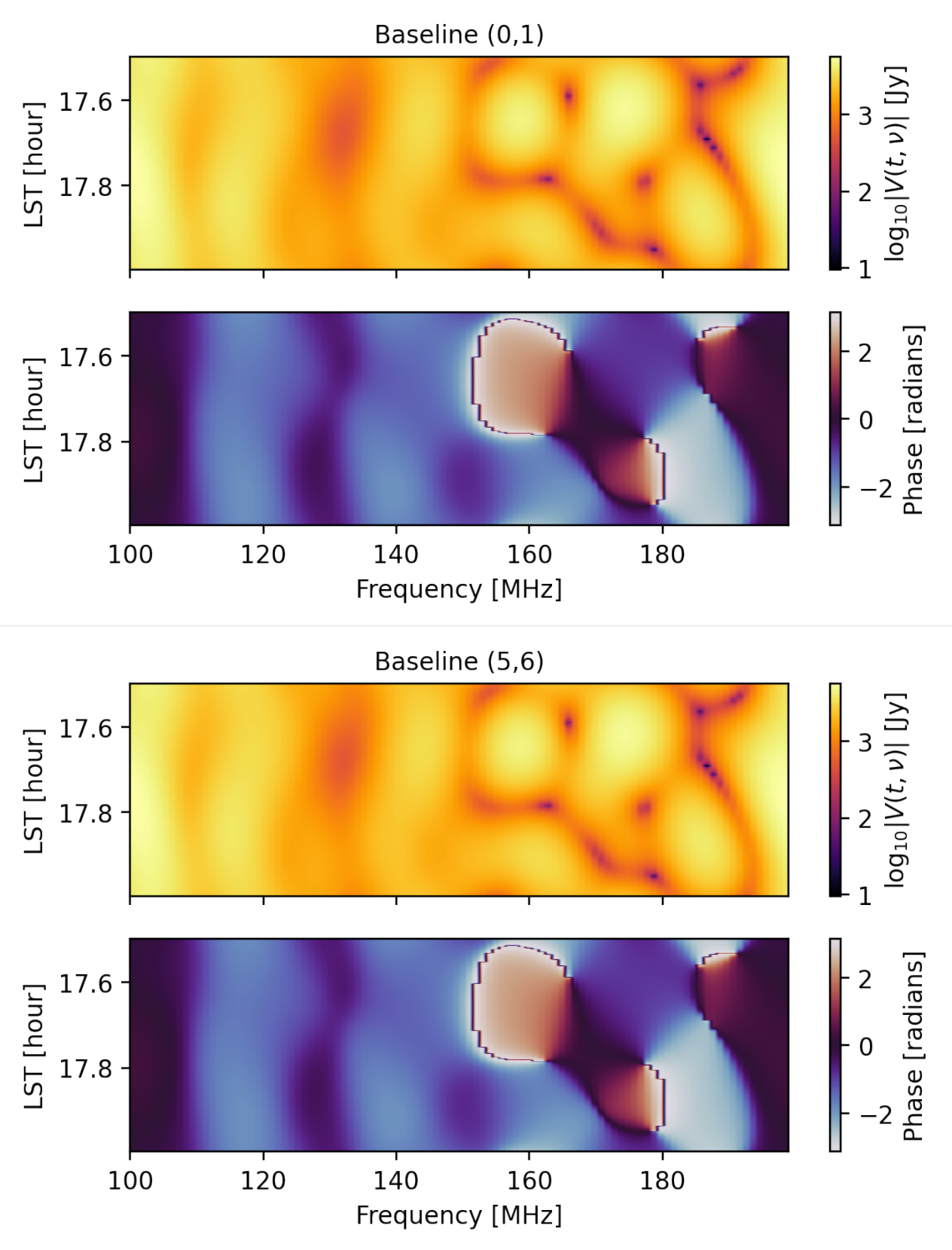
[16]:
```
sim.refresh()  # Clear the contents and zero the data
sim.add(
    "diffuse_foreground",
    seed="redundant",  # Use the same random state for each redundant group
)
```

[17]:
```
np.all(sim.get_data(0,1,"xx") == sim.get_data(5,6,"xx"))
```

[17]:
```
True
```

[18]:
```
# Let's plot them again in case the direct comparison wasn't enough.
fig1 = waterfall(sim, antpairpol=(0,1,"xx"), title="Baseline (0,1)")
fig2 = waterfall(sim, antpairpol=(5,6,"xx"), title="Baseline (5,6)")
```

```
FixedFormatter should only be used together with FixedLocator
```



It is worth noting that the `get_data` method used here is the `pyuvdata.UVData` method that has been exposed to the

`Simulator`, not to be confused with the `get` method of the `Simulator`, but more on that later.

[19]:
```
# Go one step further with this example.
all_redundant = True
for red_grp in sim.red_grps:
    if len(red_grp) == 1:
        continue  # Nothing to see here
    base_vis = sim.get_data(red_grp[0])
    # Really check that data is redundant.
    for baseline in red_grp[1::]:
        all_redundant &= np.allclose(base_vis, sim.get_data(baseline))
all_redundant
```

[19]:
```
True
```

Retrieving Simulated Effects

The parameter values used for simulating an effect are stored so that the effect may be recovered at a later time by re-simulating the effect. For effects that have a random element (like bandpass gains or thermal noise), you'll want to make sure that the `seed` parameter is specified to ensure that the random state can be configured appropriately.

[20]:
```
sim.refresh()
vis = sim.add(
    "noiselike_eor",
    eor_amp=1e-3,
    seed="redundant",
    ret_vis=True,  # Return a copy of the visibility for comparison later
)
sim.add(
    "gains",  # Apply a bandpass to the EoR data
    seed="once",  # Use the same seed each pass through the loop
)
```

As an extra note regarding the `"once"` setting used to set the random state in the above cell, this setting ensures that the same random state is used each iteration of the loop over the array. This isn't strictly necessary to use for multiplicative effects, since these are computed once before iterating over the array, but should be used for something like point-source foregrounds (which randomly populates the sky with sources each time the model is called).

[21]:
```
# The data has been modified by the gains, so it won't match the original.
np.allclose(vis, sim.data.data_array)
```

[21]:
```
False
```

[22]:
```
# We can recover the simulated effect if we wanted to, though.
np.allclose(vis, sim.get("noiselike_eor"))
```

[22]:
```
True
```

It is worth highlighting that the `get` method does *not* retrieve cached data (since simulated effects are not cached), but rather re-simulates the effect using the same parameter values and random states (since these are cached).

[23]:
```
# Make explicitly clear that the simulated effects are not cached
vis is sim.get("noiselike_eor")
```

[23]:
```
False
```

[24]:
```
# Show that the same effect really is recovered.
gains = sim.get("gains")
new_vis = vis.copy()
for ant1, ant2, pol, blt_inds, pol_ind in sim._iterate_antpair_pols():
    # Modify the old visibilities by the complex gains manually.
    total_gain = gains[(ant1, pol[0])] * np.conj(gains[(ant2, pol[1])])
    new_vis[blt_inds,0,:,pol_ind] *= total_gain
np.allclose(new_vis, sim.data.data_array)  # Now check that they match.
```

[24]:
```
True
```

Each time a component is added to the simulation, it is logged in the history:

[25]:
```
print(sim.data.history)
```

```
hera_sim v1.0.1.dev14+ga590958: Added noiselikeeor using parameters:
eor_amp = 0.001
fringe_filter_type = tophat
hera_sim v1.0.1.dev14+ga590958: Added bandpass using parameters:
bp_poly = <hera_sim.interpolators.Bandpass object at 0x7f124d22d550>
```

Saving A Simulation

Finally, we'll often want to write simulation data to disk. The simplest way to do this is with the `write` method:

[26]:
```
tempdir = Path(tempfile.mkdtemp())
filename = tempdir / "simple_example.uvh5"
sim.write(filename, save_format="uvh5")
```

[27]:
```
filename in tempdir.iterdir()
```

[27]:
```
True
```

[28]:
```
# Check that the data is the same.
sim2 = Simulator(data=filename)
is_equiv = True
# Just do a basic check.
for attr in ("data_array", "freq_array", "time_array", "antenna_positions"):
    is_equiv &= np.all(getattr(sim.data, attr) == getattr(sim2.data, attr))
is_equiv
```

```
Telescope hera_sim is not in known_telescopes.
```

[28]:
```
True
```

This concludes the section on basic use of the `Simulator`.

Advanced Use

The preceding examples should provide enough information to get you started with the `Simulator`. The rest of this notebook shows some of the more advanced features offered by the `Simulator`.

Singling Out Baselines/Antennas/Polarizations

It is possible to simulate an effect for only a subset of antennas, baselines, and/or polarizations by making use of the `vis_filter` parameter when using the `add` method. Below are some examples.

[29]:
```
sim.refresh()
vis_filter = [(0,1), (3,6)]  # Only do two baselines
sim.add("diffuse_foreground", vis_filter=vis_filter)
for baseline in sim.data.get_antpairs():
    if np.any(sim.get_data(baseline)):
        print(f"Baseline {baseline} has had an effect applied.")
```

```
Baseline (1, 0) has had an effect applied.
Baseline (6, 3) has had an effect applied.
```

```
You have not specified how to seed the random state. This effect might not be exactly␣
↪recoverable.
```

[30]:
```
vis_filter = [1,]  # Apply gains to only one antenna
vis_A = sim.get_data(0, 1)
vis_B = sim.get_data(3, 6)
sim.add("gains", vis_filter=vis_filter)
np.allclose(vis_A, sim.get_data(0, 1)), np.allclose(vis_B, sim.get_data(3, 6))
```

[30]:
```
(False, True)
```

[31]:
```
# Now something a little more complicated.
hera_sim.defaults.deactivate()  # Use the same initial params, but 2 pol
sim = Simulator(polarization_array=["xx", "yy"], **sim_params)
hera_sim.defaults.activate()

# Actually calculate the effects for the example.
vis_filter = [(0,1,"yy"), (0,2,"yy"), (1,2,"yy")]
sim.add("noiselike_eor", vis_filter=vis_filter)
for antpairpol, vis in sim.data.antpairpol_iter():
    if np.any(vis):
        print(f"Antpairpol {antpairpol} has had an effect applied.")
# Apply gains only to antenna 0
vis_A = sim.get_data(0, 1, "yy")
vis_B = sim.get_data(0, 2, "yy")
vis_C = sim.get_data(1, 2, "yy")
sim.add("gains", vis_filter=[0,])
(
    np.allclose(vis_A, sim.get_data(0, 1, "yy")),
    np.allclose(vis_B, sim.get_data(0, 2, "yy")),
    np.allclose(vis_C, sim.get_data(1, 2, "yy")),
)
```

```
You have not specified how to seed the random state. This effect might not be exactly␣
↪recoverable.
Antpairpol (1, 0, 'yy') has had an effect applied.
Antpairpol (2, 0, 'yy') has had an effect applied.
Antpairpol (2, 1, 'yy') has had an effect applied.
```

[31]:
```
(False, False, True)
```

For some insight into what's going on under the hood, this feature is implemented by recursively checking each entry in `vis_filter` and seeing if the current baseline + polarization (in the loop that simulates the effect for every baseline + polarization) matches any of the keys provided in `vis_filter`. If any of the keys are a match, then the filter says to simulate the effect—this is important to note because it can lead to some unexpected consequences. For example:

[32]:
```
seed = 12345  # Ensure that the random components are identical.
sim.refresh()
vis_filter = [(0,1), "yy"]
vis_A = sim.add(
    "diffuse_foreground",
    vis_filter=vis_filter,
    ret_vis=True,
    seed=seed,
)
sim.refresh()
vis_filter = [(0,1,"yy"),]
vis_B = sim.add(
    "diffuse_foreground",
    vis_filter=vis_filter,
    ret_vis=True,
    seed=seed,
)
np.allclose(vis_A, vis_B)
```

[32]:
```
False
```

[33]:
```
# In case A, data was simulated for both polarizations for baseline (0,1)
blt_inds = sim.data.antpair2ind(0, 1, ordered=False)
np.all(vis_A[blt_inds])
```

[33]:
```
True
```

[34]:
```
# As well as every baseline for polarization "yy"
np.all(vis_A[...,1])
```

[34]:
```
True
```

[35]:
```
# Only baseline (0,1) had an effect simulated for polarization "xx"
np.all(vis_A[...,0])
```

[35]:
```
False
```

[36]:
```
# Whereas for case B, only baseline (0, 1) with polarization "yy"
# had the simulated effect applied.
(
    np.all(vis_B[blt_inds,...,1]),  # Data for (0, 1, "yy")
```

(continues on next page)

(continued from previous page)

```
    np.all(vis_B[blt_inds]),  # Data for both pols of (0, 1)
    np.all(vis_B[...,1]),  # Data for all baselines with pol "yy"
)
```

[36]:
```
(True, False, False)
```

Here's an example of how the `vis_filter` parameter can be used to simulate a single antenna that is especially noisy.

[37]:
```
hera_sim.defaults.deactivate()  # Pause the H1C defaults for a moment...
sim = Simulator(**sim_params)  # Only need one polarization for this example.
hera_sim.defaults.activate()  # Turn the defaults back on for simulating effects.
# Start by adding some foregrounds.
sim.add("diffuse_foreground", seed="redundant")
# Then add noise with a receiver temperature of 100 K to all baselines
# except those that contain antenna 0.
v = sim.get_data(0, 1)
vis_filter = [antpair for antpair in sim.get_antpairs() if 0 not in antpair]
sim.add("thermal_noise", Trx=100, seed="initial", vis_filter=vis_filter)
# Now make antenna 0 dramatically noisy.
sim.add(
    "thermal_noise",
    Trx=5e4,
    seed="initial",
    vis_filter=[0,],
    component_name="noisy_ant",
)
```

[38]:
```
# Recall that baselines (0,1) and (2,3) are redundant,
# so the only difference here is the noise.
jansky_to_kelvin = utils.jansky_to_kelvin(
    sim.freqs,
    hera_sim.defaults('omega_p'),
)
Tsky_A = sim.get_data(0, 1) * jansky_to_kelvin
Tsky_B = sim.get_data(2, 3) * jansky_to_kelvin
np.std(np.abs(Tsky_A)), np.std(np.abs(Tsky_B))
```

[38]:
```
(50.151933214628464, 47.48252263186864)
```

Notice that the final call to the `add` method specified a value for the parameter `component_name`. This is especially useful when simulating an effect multiple times using the same class, as we have done here. As currently implemented, the `Simulator` would normally overwrite the parameters from the first application of noise with the parameters from the second application of noise; however, by specifying a name for the second application of noise, we can recover the two effects independently:

[39]:
```
vis_A = sim.get("thermal_noise")
vis_B = sim.get("noisy_ant")
blt_inds = sim.data.antpair2ind(0, 1)
(
    np.any(vis_A[blt_inds]),  # The first application give (0, 1) noise,
    np.all(vis_B[blt_inds]),  # but the second application did.
)
```

[39]:
```
(False, True)
```

With that, we conclude this section of the tutorial. This section should have provided you with sufficient examples to add effects to your own simulation in rather complicated ways—we recommend experimenting a bit on your own!

Naming Components

This was briefly touched on in the previous section, but it is interesting enough to get its own section. By using the `component_name` parameter, it is possible to use the same model multiple times for simulating an effect and have the `Simulator` be able to recover every different effect simulated using that model. Here are some examples:

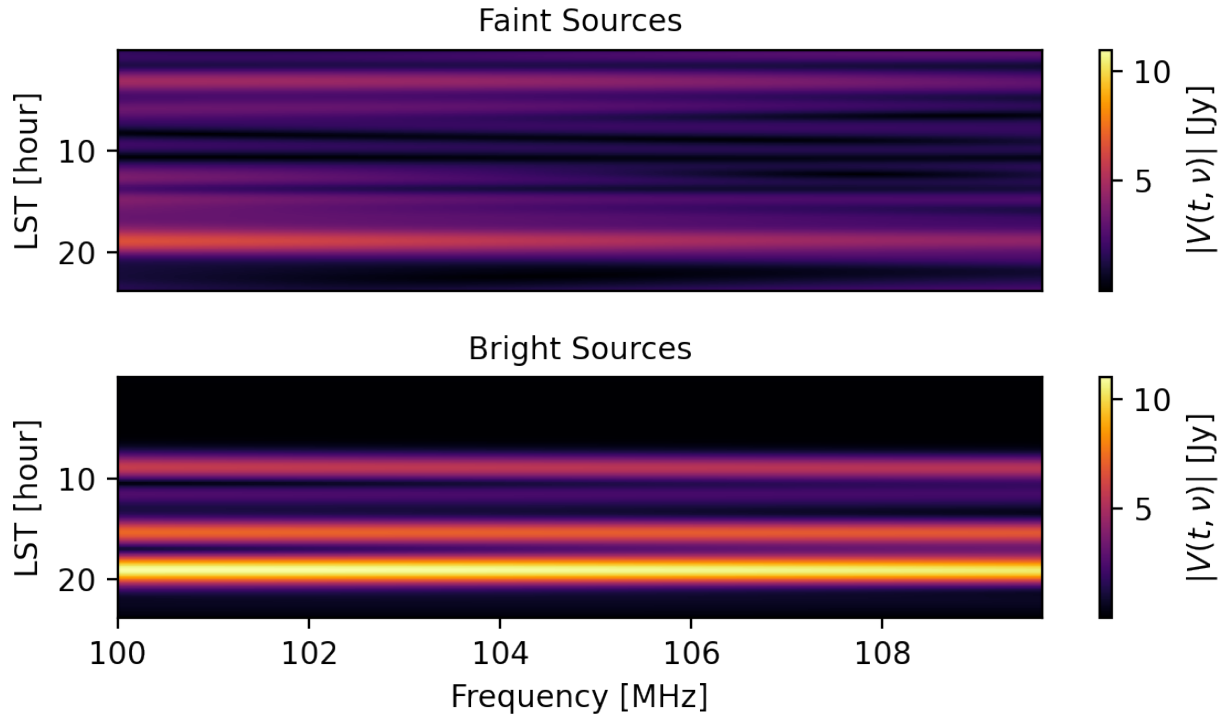
[40]:
```
hera_sim.defaults.set("debug")
sim = Simulator(
    Ntimes=6*24,  # Do a full day
    integration_time=10*60,  # In 10 minute increments
    Nfreqs=100,  # Just to make the plots look nicer
)
sim.add(
    "pntsrc_foreground",
    nsrcs=5000,  # A lot of sources
    Smin=0.01,
    Smax=0.5,  # That are relatively faint
    component_name="faint sources",
    seed="once",
)
sim.add(
    "pntsrc_foreground",
    nsrcs=10,  # Just a few sources
    Smin=0.5,
    Smax=10,  # That are bright
    component_name="bright_sources",
    seed="once",
)
```

[41]:
```
# Let's look at the data. First, we need to recover it.
faint_srcs = sim.get("faint sources", key=(0,1,"xx"))
bright_srcs = sim.get("bright_sources", key=(0,1,"xx"))

# Now let's get ready to plot.
fig, (ax1, ax2) = plt.subplots(nrows=2, ncols=1, figsize=(6,3.5), dpi=200)

# Use a common colorscale
vmin = min(np.abs(faint_srcs).min(), np.abs(bright_srcs).min())
vmax = max(np.abs(faint_srcs).max(), np.abs(bright_srcs).max())

# Actually make the plots
fig, ax1 = labeled_waterfall(
    faint_srcs,
    freqs=sim.freqs * units.GHz.to("Hz"),
    lsts=sim.lsts,
    set_title="Faint Sources",
    ax=ax1,
```

(continues on next page)

(continued from previous page)

```
    mode="abs",
    vmin=vmin,
    vmax=vmax,
)
ax1.xaxis.set_visible(False)
fig, ax2 = labeled_waterfall(
    bright_srcs,
    freqs=sim.freqs * units.GHz.to("Hz"),
    lsts=sim.lsts,
    set_title="Bright Sources",
    mode="abs",
    ax=ax2,
    vmin=vmin,
    vmax=vmax,
)
fig.tight_layout()
```



So, by setting a value for the `component_name` parameter when calling the `add` method, it is possible to use a single model for simulating multiple different effects and later on recover the individual effects.

Pre-Computing Fringe-Rate/Delay Filters

Some of the effects that are simulated utilize a fringe-rate filter and/or a delay filter in order to add to the realism of the simulation while keeping the computational overhead relatively low. Typically these filters are calculated on the fly; however, this can start to become an expensive part of the calculation for very large arrays. In order to address this, we implemented the ability to pre-compute these filters and use the cached filters instead. Here's an example.

[42]:
```
# Make a big array with a high degree of redundancy.
big_array = hera_sim.antpos.hex_array(6, split_core=False, outriggers=0)
hera_sim.defaults.set("debug")  # Use short time and frequency axes.
sim = Simulator(array_layout=big_array)
delay_filter_kwargs = {"standoff": 30, "delay_filter_type": "gauss"}
fringe_filter_kwargs = {"fringe_filter_type": "gauss", "fr_width": 1e-4}
seed = 12345
```

[43]:
```
# First, let's simulate an effect without pre-computing the filters.
vis_A = sim.add(
    "diffuse_foreground",
    delay_filter_kwargs=delay_filter_kwargs,
    fringe_filter_kwargs=fringe_filter_kwargs,
    seed=seed,
    ret_vis=True,
)
```

[44]:
```
# Now do it using the pre-computed filters.
sim.calculate_filters(
    delay_filter_kwargs=delay_filter_kwargs,
    fringe_filter_kwargs=fringe_filter_kwargs,
)
vis_B = sim.add("diffuse_foreground", seed=seed, ret_vis=True)
```

[45]:
```
# Show that we get the same result.
np.allclose(vis_A, vis_B)
```

[45]:
```
True
```

The above example shows how to use the filter caching mechanism and that it produces results that are identical to those when the filters are calculated on the fly, provided the filters are characterized the same way. Note, however, that this feature is still experimental and extensive benchmarking needs to be done to determine when it is beneficial to use the cached filters.

Using Custom Models

In addition to using the models provided by `hera_sim`, it is possible to make your own models and add an effect to the simulation using the custom model and the `Simulator`. If you would like to do this, then you need to follow some simple rules for writing the custom model:

- A component base class must be defined using the `@component` decorator.
- Models must inherit from the component base class.
- The model call signature must have `freqs` as a parameter. Additive components (like visibilities) must also take `lsts` as a parameter.
  - Additive components must return a `np.ndarray` with shape `(lsts.size, freqs.size)`.

  - Multiplicative components must have a class attribute `is_multiplicative` that is set to `True`, and they must return a dictionary mapping antenna numbers to `np.ndarrays` with shape `(freqs.size,)` or `(lsts.size, freqs.size)`.

- Frequencies will always be passed in units of GHz, and LSTs will always be passed in units of radians.

An example of how to do this is provided in the following section.

Registering Classes

[46]:
```
# Minimal example for using custom classes.
# First, make the base class that the custom model inherits from.
from hera_sim import component

@component
class Example:
    # The base class doesn't *need* a docstring, but writing one is recommended
    pass

class TestAdd(Example):
    def __init__(self):
        pass

    def __call__(self, lsts, freqs):
        return np.ones((len(lsts), len(freqs)), dtype=complex)

sim.refresh()
sim.add(TestAdd)  # You can add the effect by directly referencing the class
np.all(sim.data.data_array == 1)
```

```
You have not specified how to seed the random state. This effect might not be exactly␣
↪recoverable.
```

[46]:
```
True
```

[47]:
```
# Now for a multiplicative effect
class TestMult(Example):
    is_multiplicative = True
    def __init__(self):
        pass

    def __call__(self, freqs, ants):
        return {ant: i * np.ones_like(freqs) for i, ant in enumerate(ants)}

mult_model = TestMult()
sim.add(mult_model)  # You can also use an instance of the class
np.all(sim.get_data(1,2) == 2)
```

[47]:
```
True
```

[48]:
```
# You can also give the class an alias it may be referenced by
class TestAlias(Example):
    is_multiplicative = False
    _alias = ("bias",)
```

(continues on next page)

(continued from previous page)

```
    def __init__(self):
        pass

    def __call__(self, lsts, freqs, bias=10):
        return bias * np.ones((lsts.size, freqs.size), dtype=complex)

sim.refresh()
sim.add("bias")  # Or you can just use the name (or an alias) of the model
np.all(sim.data.data_array == 10)
```

[48]:
```
True
```

How does this work? Basically all that happens is that the `@component` decorator ensures that every model that is subclassed from the new simulation component is tracked in the component's `_models` attribute.

[49]:
```
Example._models
```

[49]:
```
{'testadd': __main__.TestAdd,
 'testmult': __main__.TestMult,
 'testalias': __main__.TestAlias,
 'bias': __main__.TestAlias}
```

It is also worth noting that you can make new models from components that already exist in `hera_sim`! We could, for example, make a new RFI model like so:

[50]:
```
class NewRFI(hera_sim.rfi.RFI):
    _alias = ("example_rfi",)
    """Doesn't do anything, just an example."""
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
    def __call__(self, lsts, freqs):
        """This class will be added to the list of known components, though!"""
        return np.zeros((lsts.size, freqs.size), dtype=complex)
```

In case it isn't clear: defining a new simulation component and making a model of that component makes `hera_sim` automatically aware of its existence, and so the `Simulator` is able to find it with ease (provided there isn't a name conflict). See the last entry of the following text.

[51]:
```
print(hera_sim.components.list_all_components())
```

```
array:
  lineararray
  hexarray
foreground:
  diffuseforeground | diffuse_foreground
  pointsourceforeground | pntsrc_foreground
noise:
  thermalnoise | thermal_noise
rfi:
  stations | rfi_stations
  impulse | rfi_impulse
  scatter | rfi_scatter
  dtv | rfi_dtv
  newrfi | example_rfi
```

(continues on next page)

(continued from previous page)

```
gain:
  bandpass | gains | bandpass_gain
  reflections | reflection_gains | sigchain_reflections
crosstalk:
  crosscouplingcrosstalk | cross_coupling_xtalk
  crosscouplingspectrum | cross_coupling_spectrum | xtalk_spectrum
  whitenoisecrosstalk | whitenoise_xtalk | white_noise_xtalk
eor:
  noiselikeeor | noiselike_eor
example:
  testadd
  testmult
  testalias | bias
```

Saving Data in Chunks

We finally get to the last of the advanced features of the `Simulator`: writing data to disk in a way that resembles how the correlator writes HERA data to disk. HERA data files typically only contain a few integrations, and follow a standard naming convention. With the `chunk_sim_and_save` method, you can write files to disk so that they are chunked into a set number of integrations per file and follow a particular naming scheme. Here's an example:

[52]:
```
# We'll use a new temporary directory
tempdir = Path(tempfile.mkdtemp())
sim.chunk_sim_and_save(
    save_dir=tempdir,  # Write the files to tempdir
    Nint_per_file=2,  # Include 2 integrations per file
    prefix="example",  # Prefix the file basename with "example"
    sky_cmp="custom",  # Tack on "custom" after the JD in the filename
    state="true",  # Tack on "true" after "custom" in the filename
    filetype="uvh5",  # Use the uvh5 file format
)
```

[53]:
```
print("\n".join(str(f) for f in tempdir.iterdir()))
```

```
/tmp/tmpbyom8sqz/example.2458119.50099.custom.true.uvh5
/tmp/tmpbyom8sqz/example.2458119.50173.custom.true.uvh5
/tmp/tmpbyom8sqz/example.2458119.50223.custom.true.uvh5
/tmp/tmpbyom8sqz/example.2458119.50198.custom.true.uvh5
/tmp/tmpbyom8sqz/example.2458119.50124.custom.true.uvh5
/tmp/tmpbyom8sqz/example.2458119.50050.custom.true.uvh5
/tmp/tmpbyom8sqz/example.2458119.50025.custom.true.uvh5
/tmp/tmpbyom8sqz/example.2458119.50000.custom.true.uvh5
/tmp/tmpbyom8sqz/example.2458119.50074.custom.true.uvh5
/tmp/tmpbyom8sqz/example.2458119.50149.custom.true.uvh5
```

The filename format is `save_dir/[{prefix}.]{jd_major}.{jd_minor}[.{sky_cmp}][.{state}].{filetype}`. Note that this hasn't been tested with other filetypes (e.g. `miriad`), so results may vary if you deviate from the `uvh5` format.

It's also possible to provide reference files for deciding how to chunk the files:

[54]:
```
sim.chunk_sim_and_save(
    save_dir=tempdir,
    ref_files=list(tempdir.iterdir()),
    prefix="new",
    sky_cmp="example",
    state="files",
    filetype="uvh5",
)
```

```
Telescope hera_sim is not in known_telescopes.
Telescope hera_sim is not in known_telescopes.
Telescope hera_sim is not in known_telescopes.
Telescope hera_sim is not in known_telescopes.
Telescope hera_sim is not in known_telescopes.
Telescope hera_sim is not in known_telescopes.
Telescope hera_sim is not in known_telescopes.
Telescope hera_sim is not in known_telescopes.
Telescope hera_sim is not in known_telescopes.
Telescope hera_sim is not in known_telescopes.
```

[55]:
```
print("\n".join(str(f) for f in tempdir.iterdir() if str(f).endswith("files.uvh5")))
```

```
/tmp/tmpbyom8sqz/new.2458119.50025.example.files.uvh5
/tmp/tmpbyom8sqz/new.2458119.50099.example.files.uvh5
/tmp/tmpbyom8sqz/new.2458119.50149.example.files.uvh5
/tmp/tmpbyom8sqz/new.2458119.50000.example.files.uvh5
/tmp/tmpbyom8sqz/new.2458119.50223.example.files.uvh5
/tmp/tmpbyom8sqz/new.2458119.50074.example.files.uvh5
/tmp/tmpbyom8sqz/new.2458119.50198.example.files.uvh5
/tmp/tmpbyom8sqz/new.2458119.50050.example.files.uvh5
/tmp/tmpbyom8sqz/new.2458119.50124.example.files.uvh5
/tmp/tmpbyom8sqz/new.2458119.50173.example.files.uvh5
```

The `run_sim` Method

The `Simulator` class also features the `run_sim` method, which allows you to run an entire simulation with a single method call. The idea behind this is that one might want to decide all of the simulation parameters beforehand, or have a configuration file specifying the simulation parameters, and then run the entire simulation in one go. Below are some examples of how to use the `run_sim` method.

Defining A Configuration Dictionary

We can specify a sequence of steps to be simulated by using a dictionary that maps models to dictionaries that specify their parameter values. This simulation will include diffuse foregrounds, a noiselike EoR signal, and bandpass gains.

[56]:
```
hera_sim.defaults.deactivate()
sim = Simulator(**sim_params)
config = {
    "diffuse_foreground":
        {
            "Tsky_mdl": hera_sim.defaults("Tsky_mdl"),
```

(continues on next page)

(continued from previous page)

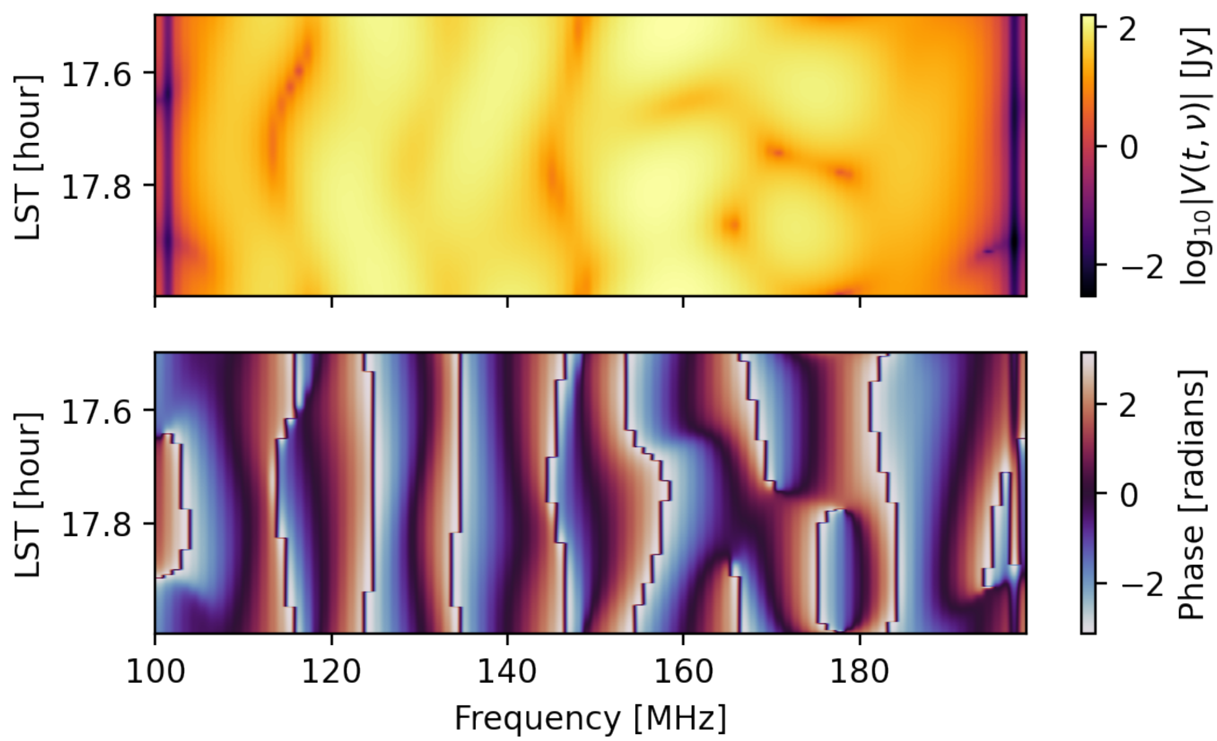
```
            "omega_p": np.ones_like(sim.freqs),
        },
    "noiselike_eor": {"eor_amp": 1e-4},
    "gains": {"dly_rng": (-50, 50)},
}
sim.run_sim(**config)
```

```
You have not specified how to seed the random state. This effect might not be exactly␣
↪recoverable.
```

[57]:

```
# Let's take a look at some of the data
fig = waterfall(sim)
```

```
FixedFormatter should only be used together with FixedLocator
```



[58]:

```
# Now let's verify that the simulation contains what we want.
print(sim.data.history)
```

```
hera_sim v1.0.1.dev14+ga590958: Added diffuseforeground using parameters:
Tsky_mdl = <hera_sim.interpolators.Tsky object at 0x7f124cc5b850>
omega_p = [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1.]
hera_sim v1.0.1.dev14+ga590958: Added noiselikeeor using parameters:
eor_amp = 0.0001
hera_sim v1.0.1.dev14+ga590958: Added bandpass using parameters:
```

(continues on next page)

(continued from previous page)

```
dly_rng = (-50, 50)
```

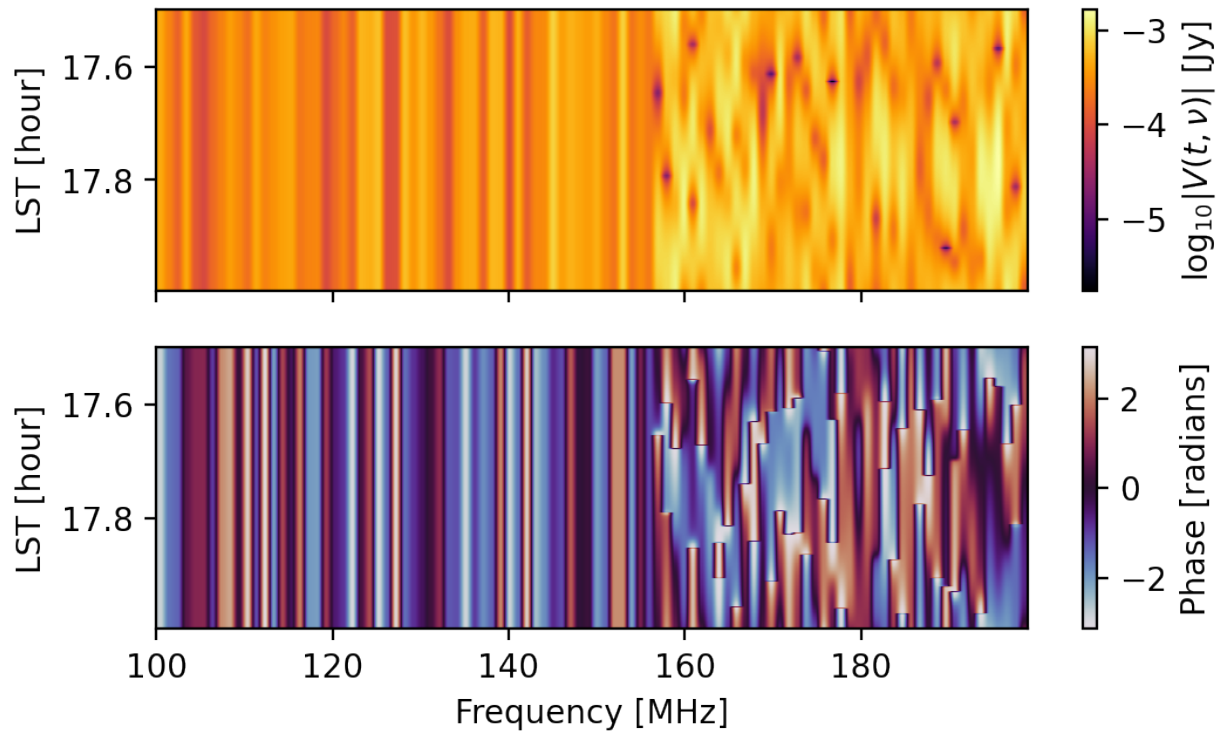
Using A Configuration File

Instead of using a dictionary to specify the parameters, we can instead use a configuration YAML file:

[59]:
```
cfg_file = tempdir / "config.yaml"
cfg_file.touch()
with open(cfg_file, "w") as cfg:
    cfg.write(
        """
        pntsrc_foreground:
            nsrcs: 10000
            Smin: 0.2
            Smax: 20
            seed: once
        noiselike_eor:
            eor_amp: 0.005
            seed: redundant
        reflections:
            amp: 0.01
            dly: 200
            seed: once
        """
    )
sim.refresh()
sim.run_sim(sim_file=cfg_file)
```

[60]:
```
# Again, let's check out some data.
fig = waterfall(sim)
```

```
FixedFormatter should only be used together with FixedLocator
```



[61]:
```
# Then verify the history.
print(sim.data.history)
```

```
hera_sim v1.0.1.dev14+ga590958: Added pointsourceforeground using parameters:
nsrcs = 10000
Smin = 0.2
Smax = 20
hera_sim v1.0.1.dev14+ga590958: Added noiselikeeor using parameters:
eor_amp = 0.005
hera_sim v1.0.1.dev14+ga590958: Added reflections using parameters:
amp = 0.01
dly = 200
```

[ ]:

The following tutorial will help you learn how to interface with the `defaults` module:

5.1.3 Guide for hera_sim Defaults and Simulator

This notebook is intended to be a guide for interfacing with the `hera_sim.defaults` module and using the `hera_sim.Simulator` class with the `run_sim` class method.

[1]:
```
import os
import numpy as np
import matplotlib.pyplot as plt
import yaml

import uvtools
import hera_sim
from hera_sim import Simulator
from hera_sim.data import DATA_PATH
from hera_sim.config import CONFIG_PATH
%matplotlib inline
```

```
/home/bobby/HERA/dev/fix_tutorial/hera_sim/hera_sim/visibilities/__init__.py:27:␣
↪UserWarning: PRISim failed to import.
  warnings.warn("PRISim failed to import.")
/home/bobby/HERA/dev/fix_tutorial/hera_sim/hera_sim/visibilities/__init__.py:33:␣
↪UserWarning: VisGPU failed to import.
  warnings.warn("VisGPU failed to import.")
/home/bobby/HERA/dev/fix_tutorial/hera_sim/hera_sim/__init__.py:36: FutureWarning:
In the next major release, all HERA-specific variables will be removed from the codebase.
↪ The following variables will need to be accessed through new class-like structures to␣
↪be introduced in the next major release:

noise.HERA_Tsky_mdl
noise.HERA_BEAM_POLY
sigchain.HERA_NRAO_BANDPASS
rfi.HERA_RFI_STATIONS

Additionally, the next major release will involve modifications to the package's API,␣
↪which move toward a regularization of the way in which hera_sim methods are interfaced␣
↪with; in particular, changes will be made such that the Simulator class is the most␣
↪intuitive way of interfacing with the hera_sim package features.
  FutureWarning)
```

We'll be using the `uvtools.plot.waterfall` function a few times throughout the notebook, but with a standardized way of generating plots. So let's write a wrapper to make these plots nicer:

[2]:
```
# let's wrap it so that it'll work on Simulator objects
def waterfall(sim, antpairpol):
    """
    For reference, sim is a Simulator object, and antpairpol is a tuple with the
    form (ant1, ant2, pol).
    """
    freqs = np.unique(sim.data.freq_array) * 1e-9 # GHz
    lsts = np.unique(sim.data.lst_array) # radians
    vis = sim.data.get_data(antpairpol)

    # extent format is [left, right, bottom, top], vis shape is (NTIMES,NFREQS)
    extent = [freqs.min(), freqs.max(), lsts.max(), lsts.min()]
```

(continues on next page)

(continued from previous page)

```
    fig = plt.figure(figsize=(12,8))
    axes = fig.subplots(2,1, sharex=True)
    axes[1].set_xlabel('Frequency [GHz]', fontsize=12)
    for ax in axes:
        ax.set_ylabel('LST [rad]', fontsize=12)

    fig.sca(axes[0])
    cax = uvtools.plot.waterfall(vis, mode='log', extent=extent)
    fig.colorbar(cax, label=r'$\log_{10}(V$/Jy)')

    fig.sca(axes[1])
    cax = uvtools.plot.waterfall(vis, mode='phs', extent=extent)
    fig.colorbar(cax, label='Phase [rad]')

    plt.tight_layout()
    plt.show()
```

[3]:
```
# while we're preparing things, let's make a dictionary of default settings for
# creating a Simulator object

# choose a modest number of frequencies and times to simulate
NFREQ = 128
NTIMES = 32

# choose the channel width so that the bandwidth is 100 MHz
channel_width = 1e8 / NFREQ

# use just two antennas
ants = {0:(20.0,20.0,0), 1:(50.0,50.0,0)}

# use cross- and auto-correlation baselines
no_autos = False

# actually make the dictionary of initialization parameters
init_params = {'n_freq':NFREQ, 'n_times':NTIMES, 'antennas':ants,
               'channel_width':channel_width, 'no_autos':no_autos}
```

[4]:
```
# turn off warnings; remove this cell in the future when this isn't a feature
hera_sim.defaults._warn = False
```

Defaults Basics

Let's first go over the basics of the `defaults` module. There are three methods that serve as the primary interface between the user and the `defaults` module: `set`, `activate`, and `deactivate`. These do what you'd expect them to do, but the `set` method requires a bit of extra work to interface with if you want to set custom default parameter values. We'll cover this later; for now, we'll see how you can switch between defaults characteristic to different observing seasons. Currently, we support default settings for the H1C observing season and the H2C observing season, and these may be accessed using the strings `'h1c'` and `'h2c'`, respectively.

[5]:
```
# first, let's note that the defaults are initially deactivated
hera_sim.defaults._override_defaults
```

[5]:
```
False
```

[6]:
```
# so let's activate the season defaults
hera_sim.defaults.activate()
hera_sim.defaults._override_defaults
```

[6]:
```
True
```

[7]:
```
# now that the defaults are activated, let's see what some differences are

# note that the defaults object is initialized with H1C defaults, but let's
# make sure that it's set to H1C defaults in case this cell is rerun later
hera_sim.defaults.set('h1c')

h1c_beam = hera_sim.noise._get_hera_bm_poly()
h1c_bandpass = hera_sim.sigchain._get_hera_bandpass()

# now change the defaults to the H2C observing season
hera_sim.defaults.set('h2c')

h2c_beam = hera_sim.noise._get_hera_bm_poly()
h2c_bandpass = hera_sim.sigchain._get_hera_bandpass()

# now compare them
print("H1C Defaults:\n \nBeam Polynomial: \n{}\nBandpass Polynomial: \n{}\n".format(h1c_
↪beam, h1c_bandpass))
print("H2C Defaults:\n \nBeam Polynomial: \n{}\nBandpass Polynomial: \n{}\n".format(h2c_
↪beam, h2c_bandpass))
```

```
H1C Defaults:

Beam Polynomial:
[ 8.07774113e+08 -1.02194430e+09  5.59397878e+08 -1.72970713e+08
  3.30317669e+07 -3.98798031e+06  2.97189690e+05 -1.24980700e+04
  2.27220000e+02]
Bandpass Polynomial:
[-2.04689451e+06  1.90683718e+06 -7.41348361e+05  1.53930807e+05
 -1.79976473e+04  1.12270390e+03 -2.91166102e+01]

H2C Defaults:

Beam Polynomial:
[ 1.36744227e+13 -2.55445530e+13  2.14955504e+13 -1.07620674e+13
  3.56602626e+12 -8.22732117e+11  1.35321508e+11 -1.59624378e+10
  1.33794725e+09 -7.75754276e+07  2.94812713e+06 -6.58329699e+04
  6.52944619e+02]
Bandpass Polynomial:
[ 1.56076423e+17 -3.03924841e+17  2.72553042e+17 -1.49206626e+17
  5.56874144e+16 -1.49763003e+16  2.98853436e+15 -4.48609479e+14
  5.07747935e+13 -4.29965657e+12  2.67346077e+11 -1.18007726e+10
  3.48589690e+08 -6.15315646e+06  4.88747021e+04]
```

(continues on next page)

(continued from previous page)

[8]:
```
# another thing
fqs = np.linspace(0.1,0.2,1024)
lsts = np.linspace(0,2*np.pi,100)

noise = hera_sim.noise.thermal_noise

hera_sim.defaults.set('h1c')
np.random.seed(0)
h1c_noise = noise(fqs,lsts)

hera_sim.defaults.set('h2c')
np.random.seed(0)
h2c_noise = noise(fqs,lsts)
np.random.seed(0)
still_h2c_noise = noise(fqs,lsts)

# passing in a kwarg
np.random.seed(0)
other_noise = noise(fqs,lsts,omega_p=np.ones(fqs.size))

# check things
print("H1C noise identical to H2C noise? {}".format(np.all(h1c_noise==h2c_noise)))
print("H2C noise identical to its other version? {}".format(np.all(h2c_noise==still_h2c_
↪noise)))
print("Either noise identical to other noise? {}".format(np.all(h1c_noise==other_noise)
                                                         or np.all(h2c_noise==other_
↪noise)))
```

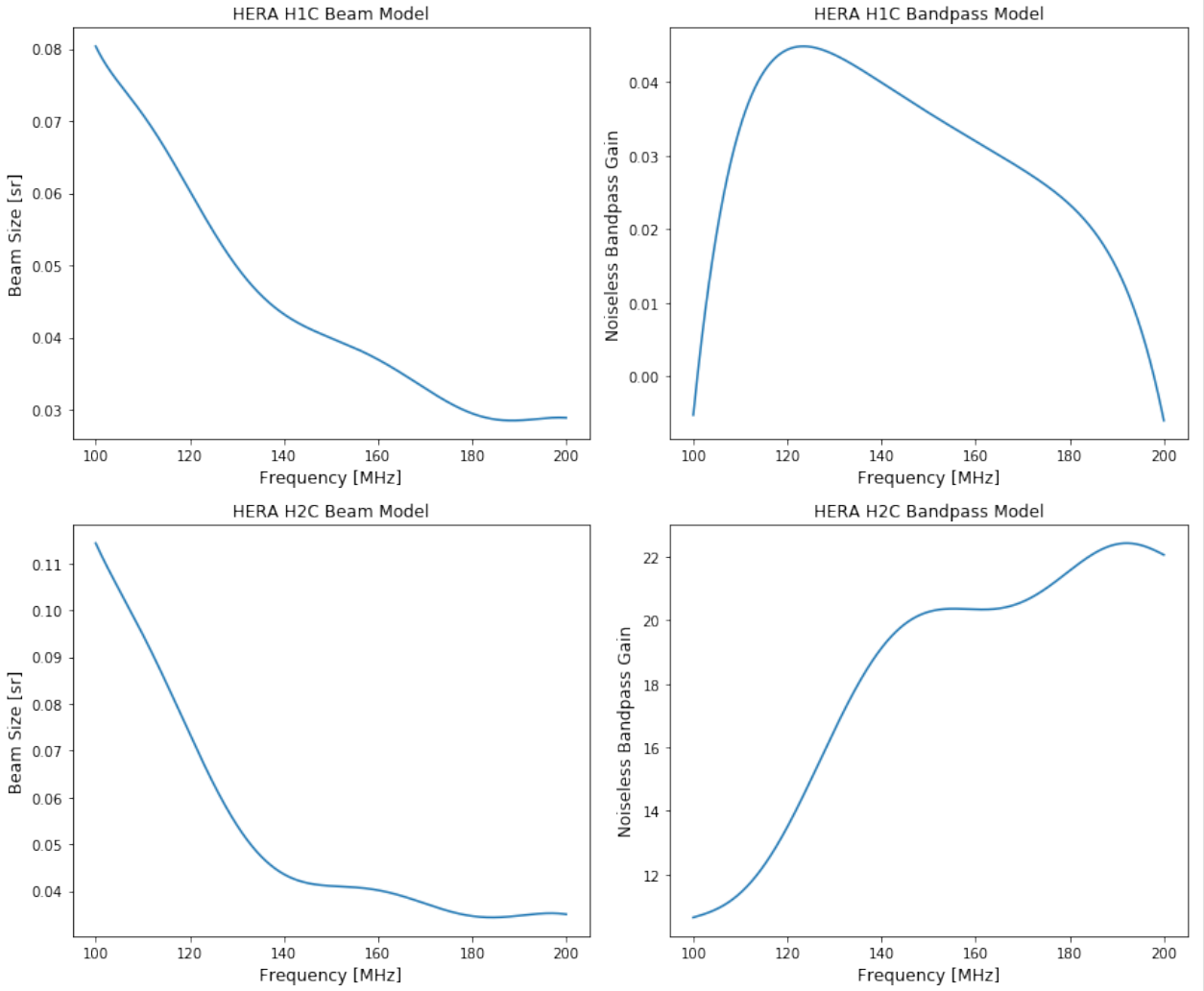
```
H1C noise identical to H2C noise? False
H2C noise identical to its other version? True
Either noise identical to other noise? False
```

[9]:
```
# check out what the beam and bandpass look like
seasons = ('h1c', 'h2c')
beams = {'h1c': h1c_beam, 'h2c': h2c_beam}
bps = {'h1c': h1c_bandpass, 'h2c': h2c_bandpass}
fig = plt.figure(figsize=(12,10))
axes = fig.subplots(2,2)
for j, ax in enumerate(axes[:,0]):
    seas = seasons[j]
    ax.set_xlabel('Frequency [MHz]', fontsize=12)
    ax.set_ylabel('Beam Size [sr]', fontsize=12)
    ax.set_title('HERA {} Beam Model'.format(seas.upper()), fontsize=12)
    ax.plot(fqs * 1e3, np.polyval(beams[seas], fqs))
for j, ax in enumerate(axes[:,1]):
    seas = seasons[j]
    ax.set_xlabel('Frequency [MHz]', fontsize=12)
    ax.set_ylabel('Noiseless Bandpass Gain', fontsize=12)
    ax.set_title('HERA {} Bandpass Model'.format(seas.upper()), fontsize=12)
    ax.plot(fqs * 1e3, np.polyval(bps[seas], fqs))
```

(continues on next page)

(continued from previous page)

```
plt.tight_layout()
plt.show()
```



[10]:
```
# let's look at the configuration that's initially loaded in
hera_sim.defaults._raw_config
```

[10]:
```
{'foregrounds': {'diffuse_foreground': {'Tsky_mdl': <hera_sim.interpolators.Tsky at␣
↪0x7f52e274cd90>,
   'omega_p': <hera_sim.interpolators.Beam at 0x7f52e27561d0>}},
 'io': {'empty_uvdata': {'start_freq': 46920776.3671875,
   'channel_width': 122070.3125,
   'integration_time': 8.59}},
 'noise': {'_get_hera_bm_poly': {'bm_poly': 'HERA_H2C_BEAM_POLY.npy'},
  'resample_Tsky': {'Tsky': 180.0, 'mfreq': 0.18, 'index': -2.5},
  'sky_noise_jy': {'inttime': 8.59},
  'thermal_noise': {'Tsky_mdl': <hera_sim.interpolators.Tsky at 0x7f52e275e810>,
   'omega_p': <hera_sim.interpolators.Beam at 0x7f52e275eb10>,
   'Trx': 0,
   'inttime': 8.59}},
```

(continues on next page)

(continued from previous page)

```
 'rfi': {'_get_hera_stations': {'rfi_stations': 'HERA_H2C_RFI_STATIONS.npy'},
  'rfi_impulse': {'chance': 0.001, 'strength': 20.0},
  'rfi_scatter': {'chance': 0.0001, 'strength': 10.0, 'std': 10.0},
  'rfi_dtv': {'freq_min': 0.174,
   'freq_max': 0.214,
   'width': 0.008,
   'chance': 0.0001,
   'strength': 10.0,
   'strength_std': 10.0}},
 'sigchain': {'_get_hera_bandpass': {'bandpass': 'HERA_H2C_BANDPASS.npy'},
  'gen_bandpass': {'gain_spread': 0.1},
  'gen_whitenoise_xtalk': {'amplitude': 3.0},
  'gen_cross_coupling_xtalk': {'amp': 0.0, 'dly': 0.0, 'phs': 0.0}}}
```

[11]:
```
# and what about the set of defaults actually used?
hera_sim.defaults._config
```

[11]:
```
{'Tsky_mdl': <hera_sim.interpolators.Tsky at 0x7f52e275e810>,
 'omega_p': <hera_sim.interpolators.Beam at 0x7f52e275eb10>,
 'start_freq': 46920776.3671875,
 'channel_width': 122070.3125,
 'integration_time': 8.59,
 'bm_poly': 'HERA_H2C_BEAM_POLY.npy',
 'Tsky': 180.0,
 'mfreq': 0.18,
 'index': -2.5,
 'inttime': 8.59,
 'Trx': 0,
 'rfi_stations': 'HERA_H2C_RFI_STATIONS.npy',
 'chance': 0.0001,
 'strength': 10.0,
 'std': 10.0,
 'freq_min': 0.174,
 'freq_max': 0.214,
 'width': 0.008,
 'strength_std': 10.0,
 'bandpass': 'HERA_H2C_BANDPASS.npy',
 'gain_spread': 0.1,
 'amplitude': 3.0,
 'amp': 0.0,
 'dly': 0.0,
 'phs': 0.0}
```

[12]:
```
# let's make two simulator objects
sim1 = Simulator(**init_params)
sim2 = Simulator(**init_params)
```

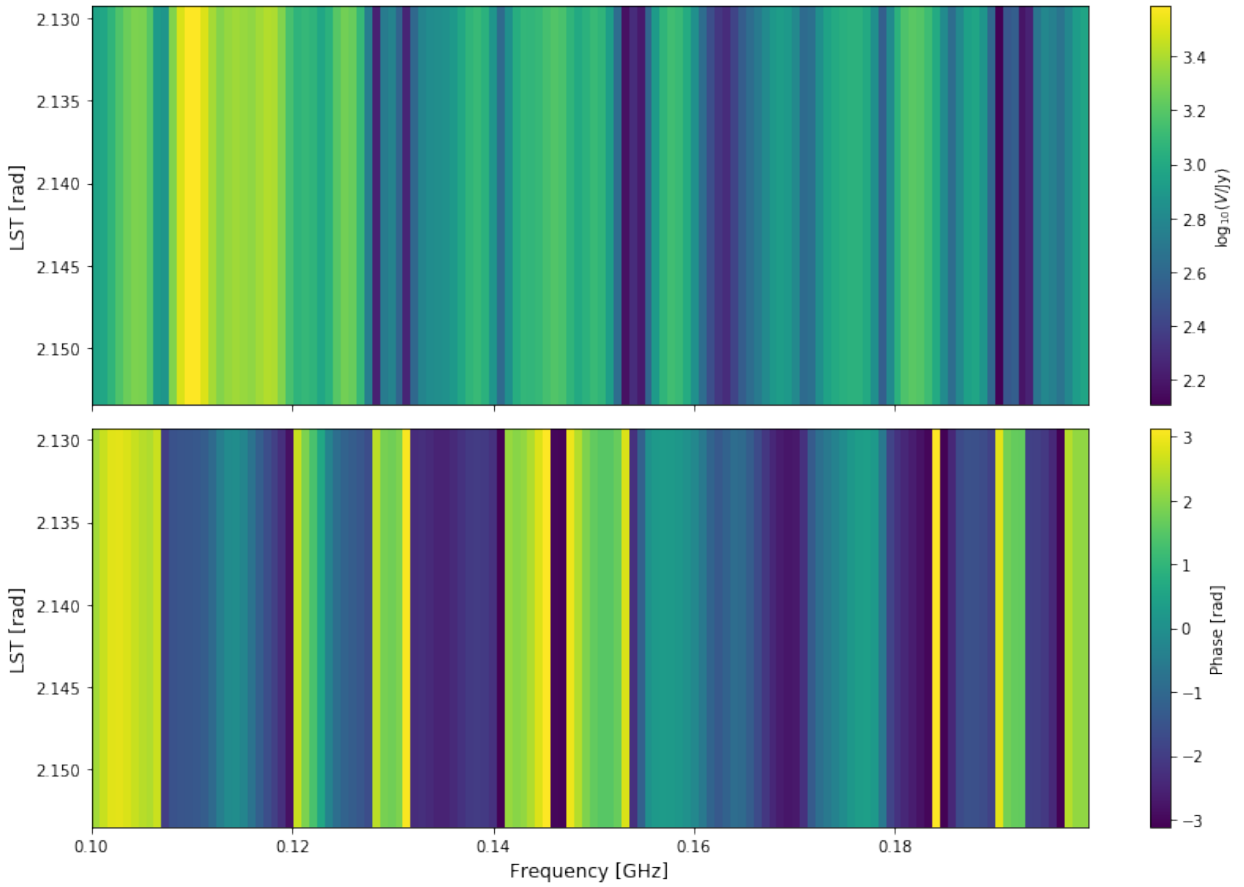
[13]:
```
# parameters for two different simulations
hera_sim.defaults.set('h1c')
sim1params = {'pntsrc_foreground': {},
              'noiselike_eor': {},
              'diffuse_foreground': {}}
```

(continues on next page)

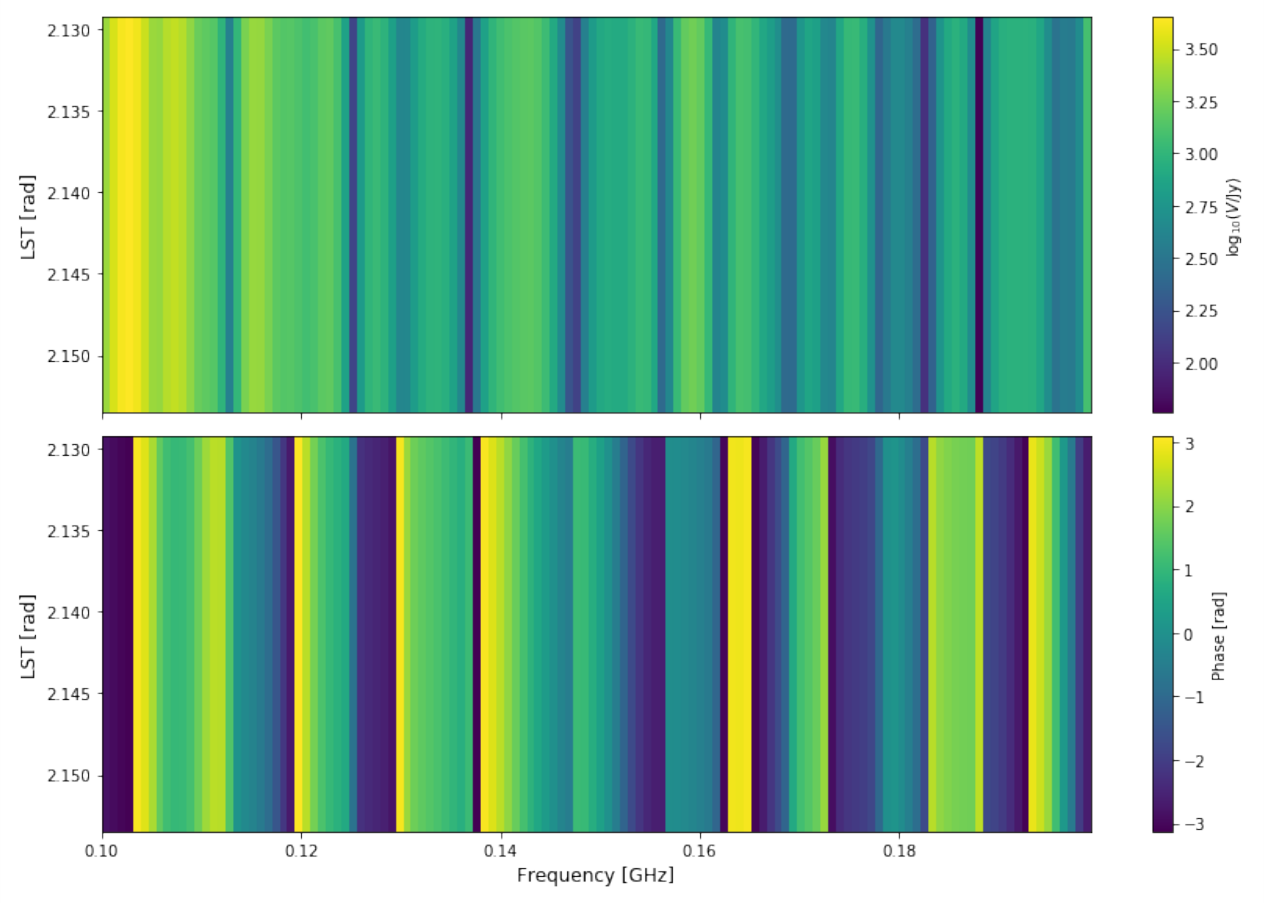
(continued from previous page)

```
hera_sim.defaults.set('h2c')
sim2params = {'pntsrc_foreground': {},
              'noiselike_eor': {},
              'diffuse_foreground': {}}
sim1.run_sim(**sim1params)
sim2.run_sim(**sim2params)
```

[14]:
```
antpairpol = (0,1,'xx')
waterfall(sim1, antpairpol)
```



[15]:
```
waterfall(sim2, antpairpol)
```



[ ]:

The following tutorial will give you an overview of how to use `hera_sim` from the command line to add instrumental systematics / noise to visibilities:

5.1.4 Running `hera_sim` from the command line

As of v0.2.0 of `hera_sim`, quick-and-dirty simulations can be run from the command line by creating a configuration file and using `hera_sim`'s `run` command to create simulated data in line with the configuration's specifications. The basic syntax of using `hera_sim`'s command-line interface is (this can be run from anywhere if `hera_sim` is installed):

```
$ hera-sim-simulate.py run --help
usage: hera-sim-simulate.py [-h] [-o OUTFILE] [-v] [-sa] [--clobber] config

Run a hera_sim-managed simulation from the command line.

positional arguments:
  config                Path to configuration file.

options:
  -h, --help            show this help message and exit
  -o OUTFILE, --outfile OUTFILE
                        Where to save simulated data. Overrides outfile
```

(continues on next page)

(continued from previous page)

```
                        specified in config.
  -v, --verbose         Print progress updates.
  -sa, --save_all       Save each simulation component.
  --clobber             Overwrite existing files in case of name conflicts.
```

An example configuration file can be found in the `config_examples` directory of the repo's top-level directory. Here are its contents:

```
$ cat -n ../config_examples/template_config.yaml
     1        # This document is intended to serve as a template for constructing new
     2        # configuration YAMLs for use with the command-line interface.
     3
     4        bda:
     5            max_decorr: 0
     6            pre_fs_int_time: !dimensionful
     7                value: 0.1
     8                units: 's'
     9            corr_FoV_angle: !dimensionful
    10                value: 20
    11                units: 'deg'
    12            max_time: !dimensionful
    13                value: 16
    14                units: 's'
    15            corr_int_time: !dimensionful
    16                value: 2
    17                units: 's'
    18        filing:
    19            outdir: '.'
    20            outfile_name: 'quick_and_dirty_sim.uvh5'
    21            output_format: 'uvh5'
    22            clobber: True
    23        # freq and time entries currently configured for hera_sim use
    24        freq:
    25            n_freq: 100
    26            channel_width: 122070.3125
    27            start_freq: 46920776.3671875
    28        time:
    29            n_times: 10
    30            integration_time: 8.59
    31            start_time: 2457458.1738949567
    32        telescope:
    33            # generate from an antenna layout csv
    34            # array_layout: 'antenna_layout.csv'
    35            # generate using hera_sim.antpos
    36            array_layout: !antpos
    37                array_type: "hex"
    38                hex_num: 3
    39                sep: 14.6
    40                split_core: False
    41                outriggers: 0
    42            omega_p: !Beam
    43                # non-absolute paths are assumed to be specified relative to the
```

(continues on next page)

(continued from previous page)

```
44                # hera_sim data path
45                datafile: HERA_H2C_BEAM_MODEL.npz
46                interp_kwargs:
47                    interpolator: interp1d
48                    fill_value: extrapolate
49                    # if you want to use a polynomial interpolator instead, then
50                    # interpolator: poly1d
51                    # kwargs not accepted for this; see numpy.poly1d documentation
52        defaults:
53            # This must be a string specifying an absolute path to a default
54            # configuration file or one of the season default keywords
55            'h2c'
56        systematics:
57            rfi:
58                # see hera_sim.rfi documentation for details on parameter names
59                rfi_stations:
60                    seed: once
61                    stations: !!null
62                rfi_impulse:
63                    impulse_chance: 0.001
64                    impulse_strength: 20.0
65                rfi_scatter:
66                    scatter_chance: 0.0001
67                    scatter_strength: 10.0
68                    scatter_std: 10.0
69                rfi_dtv:
70                    seed: once
71                    dtv_band:
72                        - 0.174
73                        - 0.214
74                    dtv_channel_width: 0.008
75                    dtv_chance: 0.0001
76                    dtv_strength: 10.0
77                    dtv_std: 10.0
78            sigchain:
79                gains:
80                    seed: once
81                    gain_spread: 0.1
82                    dly_rng: [-20, 20]
83                    bp_poly: HERA_H1C_BANDPASS.npy
84                sigchain_reflections:
85                    seed: once
86                    amp: !!null
87                    dly: !!null
88                    phs: !!null
89            crosstalk:
90                # only one of the two crosstalk methods should be specified
91                gen_whitenoise_xtalk:
92                    amplitude: 3.0
93                # gen_cross_coupling_xtalk:
94                    # seed: initial
95                    # amp: !!null
```

(continues on next page)

(continued from previous page)

```
    96                     # dly: !!null
    97                     # phs: !!null
    98             noise:
    99                 thermal_noise:
   100                     seed: initial
   101                     Trx: 0
   102         sky:
   103             Tsky_mdl: !Tsky
   104                 # non-absolute paths are assumed to be relative to the hera_sim
   105                 # data folder
   106                 datafile: HERA_Tsky_Reformatted.npz
   107                 # interp kwargs are passed to scipy.interp.RectBivariateSpline
   108                 interp_kwargs:
   109                     pol: xx # this is popped when making a Tsky object
   110             eor:
   111                 noiselike_eor:
   112                     eor_amp: 0.00001
   113                     min_delay: !!null
   114                     max_delay: !!null
   115                     seed: redundant # so redundant baselines see same sky
   116                     fringe_filter_type: tophat
   117             foregrounds:
   118                 # if using hera_sim.foregrounds
   119                 diffuse_foreground:
   120                     seed: redundant # redundant baselines see same sky
   121                     delay_filter_kwargs:
   122                         standoff: 0
   123                         delay_filter_type: tophat
   124                         normalize: !!null
   125                     fringe_filter_kwargs:
   126                         fringe_filter_type: tophat
   127                 pntsrc_foreground:
   128                     seed: once
   129                     nsrcs: 1000
   130                     Smin: 0.3
   131                     Smax: 300
   132                     beta: -1.5
   133                     spectral_index_mean: -1.0
   134                     spectral_index_std: 0.5
   135                     reference_freq: 0.5
   136                 # Note regarding seed_redundantly:
   137                 # This ensures that baselines within a redundant group see the␣
↪same sky;
   138                 # however, this does not ensure that the sky is actually␣
↪consistent. So,
   139                 # while the data produced can be absolutely calibrated, it cannot␣
↪be
   140                 # used to make sensible images (at least, I don't *think* it can␣
↪be).
   141
   142         simulation:
   143             # specify which components to simulate in desired order
```

(continues on next page)

(continued from previous page)

```
144           # this should be a complete list of the things to include if hera_sim
145           # is the simulator being used. this will necessarily look different
146           # if other simulators are used, but that's not implemented yet
147           #
148           components: [foregrounds,
149                        noise,
150                        eor,
151                        rfi,
152                        sigchain, ]
153           # list particular model components to exclude from simulation
154           exclude: [sigchain_reflections,
155                     gen_whitenoise_xtalk,]
```

The remainder of this tutorial will be spent on exploring each of the items in the above configuration file.

BDA

The following block of text shows all of the options that must be specified if you would like to apply BDA to the simulated data. Note that BDA is applied at the very end of the script, and requires the BDA package to be installed from http://github.com/HERA-Team/baseline_dependent_averaging.

```
$ sed -n 4,17p ../config_examples/template_config.yaml
bda:
    max_decorr: 0
    pre_fs_int_time: !dimensionful
        value: 0.1
        units: 's'
    corr_FoV_angle: !dimensionful
        value: 20
        units: 'deg'
    max_time: !dimensionful
        value: 16
        units: 's'
    corr_int_time: !dimensionful
        value: 2
        units: 's'
```

Please refer to the `bda.apply_bda` documentation for details on what each parameter represents. Note that practically each entry has the tag `!dimensionful`; this YAML tag converts the entries in `value` and `units` to an `astropy.units.quantity.Quantity` object with the specified value and units.

Filing

The following block of text shows all of the options that may be specified in the `filing` section; however, not all of these *must* be specified. In fact, the only parameter that is required to be specified in the config YAML is `output_format`, and it must be either `miriad`, `uvfits`, or `uvh5`. These are currently the only supported write methods for `UVData` objects.

```
$ sed -n 18,24p ../config_examples/template_config.yaml
filing:
    outdir: '.'
```

(continues on next page)

(continued from previous page)

```
    outfile_name: 'quick_and_dirty_sim.uvh5'
    output_format: 'uvh5'
    clobber: True
# freq and time entries currently configured for hera_sim use
freq:
```

Recall that `run` can be called with the option `--outfile`; this specifies the full path to where the simulated data should be saved and overrides the `outdir` and `outfile_name` settings from the config YAML. Additionally, one can choose to use the flag `-c` or `--clobber` in place of specifying `clobber` in the config YAML. Finally, the dictionary defined by the `kwargs` entry has its contents passed to whichever write method is chosen, and the `save_seeds` option should only be used if the `seed_redundantly` option is specified for any of the simulation components.

Setup

The following block of text contains three sections: `freq`, `time`, and `telescope`. These sections are used to initialize the `Simulator` object that is used to perform the simulation. Note that the config YAML shows all of the options that may be specified, but not all options are necessarily required.

```
$ sed -n 26,53p ../config_examples/template_config.yaml
    channel_width: 122070.3125
    start_freq: 46920776.3671875
time:
    n_times: 10
    integration_time: 8.59
    start_time: 2457458.1738949567
telescope:
    # generate from an antenna layout csv
    # array_layout: 'antenna_layout.csv'
    # generate using hera_sim.antpos
    array_layout: !antpos
        array_type: "hex"
        hex_num: 3
        sep: 14.6
        split_core: False
        outriggers: 0
    omega_p: !Beam
        # non-absolute paths are assumed to be specified relative to the
        # hera_sim data path
        datafile: HERA_H2C_BEAM_MODEL.npz
        interp_kwargs:
            interpolator: interp1d
            fill_value: extrapolate
            # if you want to use a polynomial interpolator instead, then
            # interpolator: poly1d
            # kwargs not accepted for this; see numpy.poly1d documentation
defaults:
    # This must be a string specifying an absolute path to a default
```

If you are familiar with using configuration files with `pyuvsim`, then you'll notice that the sections shown above look very similar to the way config files are constructed for use with `pyuvsim`. The config files for `run` were designed as an extension of the `pyuvsim` config files, with the caveat that some of the naming conventions used in `pyuvsim` are somewhat different than those used in `hera_sim`. For information on the parameters listed in the `freq` and `time`

sections, please refer to the documentation for `hera_sim.io.empty_uvdata`. As for the `telescope` section, this is where the antenna array and primary beam are defined. The `array_layout` entry specifies the array, either by specifying an antenna layout file or by using the `!antpos` YAML tag and specifying the type of array (currently only `linear` and `hex` are supported) and the parameters to be passed to the corresponding function in `hera_sim.antpos`. The `omega_p` entry is where the primary beam is specified, and it is currently assumed that the beam is the same for each simulation component (indeed, this simulator is not intended to produce super-realistic simulations, but rather perform simulations quickly and give somewhat realistic results). This entry defines an interpolation object to be used for various `hera_sim` functions which require such an object; please refer to the documentation for `hera_sim.interpolators.Beam` for more information. Future versions of `hera_sim` will provide support for specifying the beam in an antenna layout file, similar to how it is done by `pyuvsim`.

Defaults

This section of the configuration file is optional to include. This section gives the user the option to use a default configuration to specify different parameters throughout the codebase. Users may define their own default configuration files, or they may use one of the provided season default configurations, located in the `config` folder. The currently supported season configurations are `h1c` and `h2c`. Please see the `defaults` module/documentation for more information.

```
$ sed -n 54,57p ../config_examples/template_config.yaml
    # configuration file or one of the season default keywords
    'h2c'
systematics:
    rfi:
```

Systematics

This is the section where any desired systematic effects can be specified. The block of text shown below details all of the possible options for systematic effects. Note that currently the `sigchain_reflections` and `gen_cross_coupling_xtalk` sections cannot easily be worked with; in fact, `gen_cross_coupling_xtalk` does not work as intended (each baseline has crosstalk show up at the same phase and delay, with the same amplitude, but uses a different autocorrelation visibility). Also note that the `rfi` section is subject to change, pending a rework of the `rfi` module.

```
$ sed -n 58,96p ../config_examples/template_config.yaml
        # see hera_sim.rfi documentation for details on parameter names
        rfi_stations:
            seed: once
            stations: !!null
        rfi_impulse:
            impulse_chance: 0.001
            impulse_strength: 20.0
        rfi_scatter:
            scatter_chance: 0.0001
            scatter_strength: 10.0
            scatter_std: 10.0
        rfi_dtv:
            seed: once
            dtv_band:
                - 0.174
                - 0.214
            dtv_channel_width: 0.008
```

(continues on next page)

(continued from previous page)

```
            dtv_chance: 0.0001
            dtv_strength: 10.0
            dtv_std: 10.0
    sigchain:
        gains:
            seed: once
            gain_spread: 0.1
            dly_rng: [-20, 20]
            bp_poly: HERA_H1C_BANDPASS.npy
        sigchain_reflections:
            seed: once
            amp: !!null
            dly: !!null
            phs: !!null
    crosstalk:
        # only one of the two crosstalk methods should be specified
        gen_whitenoise_xtalk:
            amplitude: 3.0
        # gen_cross_coupling_xtalk:
            # seed: initial
            # amp: !!null
            # dly: !!null
```

Note that although these simulation components are listed under `systematics`, they do not necessarily need to be listed here; the configuration file is formatted as such just for semantic clarity. For information on any particular simulation component listed here, please refer to the corresponding function's documentation. For those who may not know what it means, `!!null` is how `NoneType` objects are specified using `pyyaml`.

Sky

This section specifies both the sky temperature model to be used throughout the simulation as well as any simulation components which are best interpreted as being associated with the sky (rather than as a systematic effect). Just like the `systematics` section, these do not necessarily need to exist in the `sky` section (however, the `Tsky_mdl` entry *must* be placed in this section, as that's where the script looks for it).

```
$ sed -n 97,130p ../config_examples/template_config.yaml
            # phs: !!null
    noise:
        thermal_noise:
            seed: initial
            Trx: 0
sky:
    Tsky_mdl: !Tsky
        # non-absolute paths are assumed to be relative to the hera_sim
        # data folder
        datafile: HERA_Tsky_Reformatted.npz
        # interp kwargs are passed to scipy.interp.RectBivariateSpline
        interp_kwargs:
            pol: xx # this is popped when making a Tsky object
    eor:
        noiselike_eor:
```

(continues on next page)

(continued from previous page)

```
            eor_amp: 0.00001
            min_delay: !!null
            max_delay: !!null
            seed: redundant # so redundant baselines see same sky
            fringe_filter_type: tophat
    foregrounds:
        # if using hera_sim.foregrounds
        diffuse_foreground:
            seed: redundant # redundant baselines see same sky
            delay_filter_kwargs:
                standoff: 0
                delay_filter_type: tophat
                normalize: !!null
            fringe_filter_kwargs:
                fringe_filter_type: tophat
        pntsrc_foreground:
            seed: once
            nsrcs: 1000
            Smin: 0.3
```

As of now, `run` only supports simulating effects using the functions in `hera_sim`; however, we intend to provide support for using different simulators in the future. If you would like more information regarding the `Tsky_mdl` entry, please refer to the documentation for the `hera_sim.interpolators.Tsky` class. Finally, note that the `seed_redundantly` parameter is specified for each entry in `eor` and `foregrounds`; this parameter is used to ensure that baselines within a redundant group all measure the same visibility, which is a necessary feature for data to be absolutely calibrated. Please refer to the documentation for `hera_sim.eor` and `hera_sim.foregrounds` for more information on the parameters and functions listed above.

Simulation

This section is used to specify which of the simulation components to include in or exclude from the simulation. There are only two entries in this section: `components` and `exclude`. The `components` entry should be a list specifying which of the groups from the `sky` and `systematics` sections should be included in the simulation. The `exclude` entry should be a list specifying which of the particular models should not be simulated. Here's an example:

```
$ sed -n -e 137,138p -e 143,150p ../config_examples/template_config.yaml
        # This ensures that baselines within a redundant group see the same sky;
        # however, this does not ensure that the sky is actually consistent. So,
    # specify which components to simulate in desired order
    # this should be a complete list of the things to include if hera_sim
    # is the simulator being used. this will necessarily look different
    # if other simulators are used, but that's not implemented yet
    #
    components: [foregrounds,
                 noise,
                 eor,
```

The entries listed above would result in a simulation that includes all models contained in the `foregrounds`, `noise`, `eor`, `rfi`, and `sigchain` dictionaries, except for the `sigchain_reflections` and `gen_whitenoise_xtalk` models. So the simulation would consist of diffuse and point source foregrounds, thermal noise, noiselike EoR, all types of RFI modeled by `hera_sim`, and bandpass gains, with the effects simulated in that order. It is important to make sure that effects which enter multiplicatively (i.e. models from `sigchain`) are simulated *after* effects that enter additively, since

the order that the simulation components are listed in is the same as the order of execution.

The following tutorials will give you an overview of how to simulate visibilities from sky models using `hera_sim` (first from an interpreter, then from the command line):

5.1.5 Visibility Simulator Examples

Although `hera_sim` is primarily aimed at simulating instrumental effects on top of existing visibility data, the package also has the `visibilities` module that offers a uniform visibility simulation interface to wrappers for several visibility simulators, as well as a few analytic beam models of the HERA antenna.

Visibility Simulation Interface

Starting from versions of `hera_sim >= 2.0.0`, wrappers for VisCPU, healvis, and pyuvsim visibility simulators are provided through the `VisCPU`, `Healvis`, and `UVSim` classes.

The new `ModelData` object serves as a container for the visiblity data and all information required to perform visibility simulation. A preferred method to initialize this object is to use the classmethod `from_config`, providing pyuvsim configuration files that specify the observation and telescope parameters, beam model, and sky model. Direct initialization is also supported and currently required to utilize analytic HERA beam models in the `hera_sim.beams` module (see Manual Initialization). Under the hood, ModelData wraps three objects used for facilitating simulation in pyuvsim, each encapsulating different information required for simulating visibility. * `pyuvdata.UVData` * A data container for observation parameters, array layout, telescope location, and visibility. * Accesible from the `uvdata` property. * Package reference: https://pyuvdata.readthedocs.io/en/latest/index.html * `pyradiosky.SkyModel` * A data container for sky models, including catalog of sources and diffuse sky maps in HEALPix. * Provide methods for conversion from a catalog of point sources to a HEALPix map and vice versa, which is used by the visibility simulator wrappers. * Accesible from the `sky_model` property. * Package reference: https://pyradiosky.readthedocs.io/en/latest/index.html * `pyuvsim.BeamList` and the associated `beam_ids` * Specify the antenna beam models and which beam model each antenna uses. (See Manual Initialization for supported beam types.) * Accesible from the `beams` and `beam_ids` properties. * Reference: https://pyuvsim.readthedocs.io/en/latest/classes.html#pyuvsim.BeamList

The `VisibilitySimulation` class is the new uniform visibility simulation interface. It takes an instance of a `VisibilitySimulator` class (of which `VisCPU`, `HealVis` and `UVSim` are examples) and a `ModelData` object as inputs. Visibility simulation can then be executed simply by calling the `simulate` method on an instance of a `VisibilitySimulation` class. This returns a Numpy array of visibilities in the same shape as in the `UVData.data_array` specification, which is also added to the object property `VisibilitySimulation.data_model.uvdata.data_array`.

The motivation behind the `VisibilitySimulation`-`VisibilitySimulator`-`ModelData` setup is to provide a single uniform interface to many simulators, and the ability to create wrappers for simulators that are not already provided. This interface is heavily based on the pyuvsim configuration specifications and infrastructures, and thus it provides the ability to perform simulation with any simulator given the same static configuration.

Example 1: Demonstrate the Uniform Visibility Simulator Interface

[1]:
```
"""This example simulates visiblities with VisCPU at 10 frequencies and 60
times with the 50 brightest GLEAM sources as a sky model and the HERA Phase I
array as the instrument."""
from hera_sim import DATA_PATH
from hera_sim.visibilities import VisCPU, ModelData, VisibilitySimulation
```

(continues on next page)

(continued from previous page)

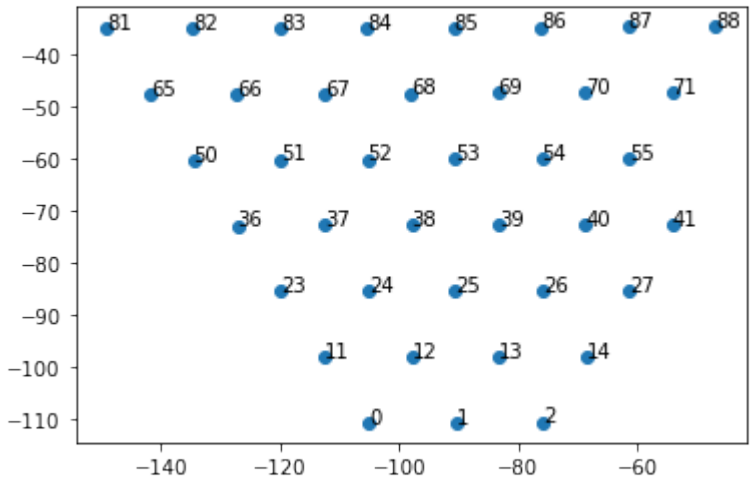
```
# Path to the example pyuvsim configuration files.
# These can be found in `hera_sim.DATA_PATH` to follow along.
config_file = (
    DATA_PATH /
    'tutorials_data/visibility_simulator/obsparam_hera_phase1_gleam_top50.yaml'
).as_posix()

# Initialize a ModelData object.
# The pyuvsim configuration files consist of a few yaml and csv files, and a
# sky model file readable by `pyradiosky`. `ModelData.from_config` takes the
# outermost file obsparam_*.yaml as an input.
data_model = ModelData.from_config(config_file)

# Initialize a VisCPU simulator.
simulator = VisCPU()
# simulator = VisCPU(use_pixel_beams=False)
# simulator = UVSim(quiet=True)

# Construct a VisibilitySimulation object.
simulation = VisibilitySimulation(data_model=data_model, simulator=simulator)

# Executing the simulation by calling `simulate` method.
visibility_array = simulation.simulate()
```

Accessing Simulation Parameters and Visibility

All simulation parameters and visibility data are contained in the `ModelData` object and can be retrieved from the corresponding object attributes and properties.

Example 2: Plot the array layout of the simulation in Example 1

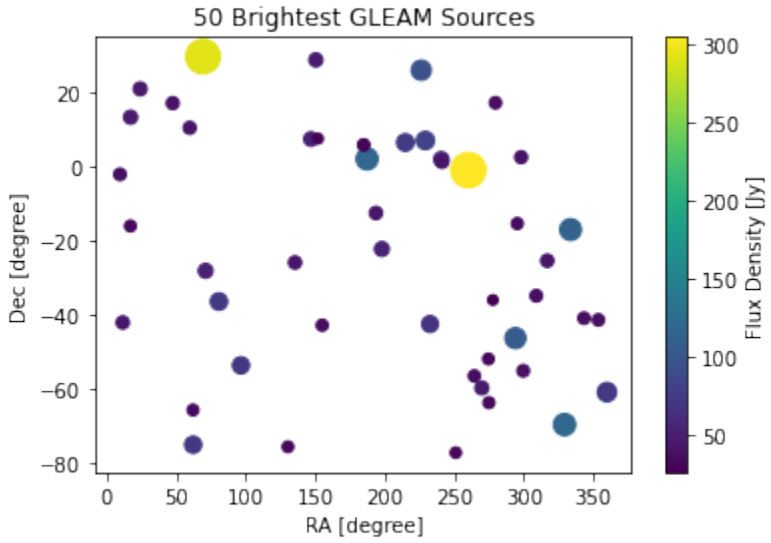
[2]:
```
import matplotlib.pyplot as plt
from pyuvdata import utils as uvutils

uvd = simulation.data_model.uvdata

# Get antennas positions and corresponding antenna numbers
antpos, antnum = uvd.get_ENU_antpos()

# Plot the EN antenna position.
plt.scatter(antpos[:, 0], antpos[:, 1])
for i, antnum in enumerate(uvd.antenna_numbers):
    plt.text(antpos[i, 0], antpos[i, 1], antnum)
plt.show()
```



Example 3: Plot the source positions in the SkyModel used in Example 1

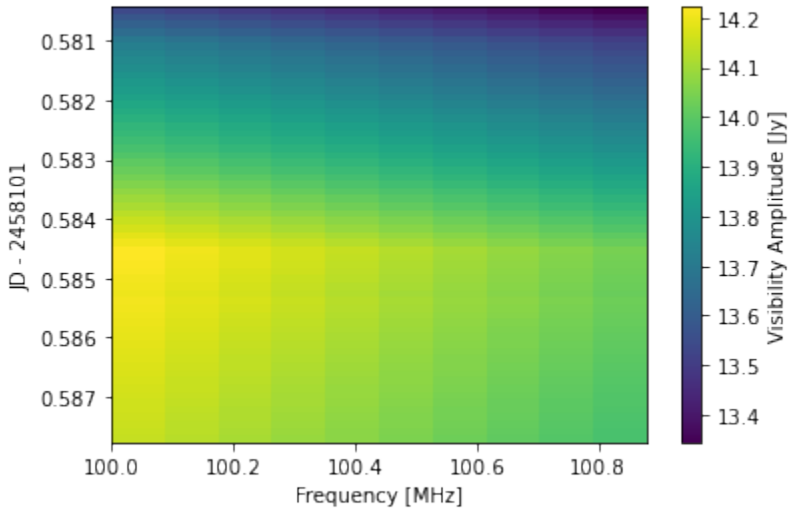
[3]:
```
sky_model = simulation.data_model.sky_model
print('Type of sky model:', sky_model.component_type)
print('Number of sources:', sky_model.Ncomponents)

# Extract the source positions and fluxes
ra = sky_model.ra
dec = sky_model.dec
flux = sky_model.stokes[0, 0, :]

# Plot the source positions with color and size showing source fluxes.
plt.scatter(ra, dec, s=flux, c=flux)
plt.colorbar(label='Flux Density [Jy]')
plt.xlabel('RA [degree]')
plt.ylabel('Dec [degree]')
plt.title('50 Brightest GLEAM Sources')
plt.show()
```

```
Type of sky model: point
Number of sources: 50
```



Example 4: Plot the waterfall of the visibility simulation in Example 1

[4]:
```
import numpy as np

bls = (0, 1, 'xx')
waterfall = uvd.get_data(bls)
extent = [uvd.freq_array[0, 0] / 1e6, uvd.freq_array[0, -1] / 1e6,
          uvd.time_array[-1] - 2458101, uvd.time_array[0] - 2458101]
plt.imshow(np.abs(waterfall), aspect='auto', interpolation='none',
           extent=extent)
plt.xlabel('Frequency [MHz]')
plt.ylabel('JD - 2458101')
plt.colorbar(label=f'Visibility Amplitude [{uvd.vis_units}]')
plt.show()
```



For more comprehensive visualization of visibility data (e.g. waterfalls of differences between visibilities, Fourier

Transform of a waterfall, and etc), users may be interested in plotting routines in the HERA uvtools package.

Initializing Model Data

As mentioned, the `ModelData` object contains all the data and information required to perform visibility simulation with `hera_sim`.

The prefered method to initlize a ModelData object is to generate a set pyuvsim configuration files (see https://pyuvsim.readthedocs.io/en/latest/parameter_files.html) and then pass the top-level obsparams_*.yaml file to the classmethod `from_config`.

```
data_model = ModelData.from_config(config_file)
```

If not using configuration files, the underlying objects can be first created and passed to the `ModelData` constructor for manual initialization.

Users may consider using `hera_sim.io.empty_uvdata`, `pyuvsim.simsetup.initialize_uvdata_from_params`, or `pyuvsim.simsetup.initialize_uvdata_from_keywords` to construct a UVData object.

See the next section on how to construct a sky model with `pyradiosky.SkyModel`.

The beam models given to the `beams` parameter can be: i) a `pyuvsim.BeamList` object, ii) a `pyuvdata.UVBeam`, or iii) a list of `pyuvsim.AnalyticBeam`, including its subclasses.

The `hera_sim.beams` module provides analytic models of the HERA antenna through the `PolyBeam`, `PerturbedPolayBeam`, and `ZernikeBeam` objects (see HERA Memo #081 and #101, and Choudhuri et al 2021). Although these beam models are subclasses of `pyuvsim.AnalyticBeam`, they are currently not regcogniziable by the pyuvsim configuration file parser, and thus can only be used by initializing `ModelData` manually.

Example 5: Manaully initialize a `ModelData` object with `PolyBeam`

[5]:
```
from pyuvsim.simsetup import initialize_uvdata_from_params, _complete_uvdata
from pyuvsim import BeamList
from pyradiosky import SkyModel
from hera_sim.beams import PolyBeam

# Initialize just the UVData and beam_ids from the pyuvsim configuraton files.
# The configuration file (same as in Example 1) uses the "airy" type
# `pyuvsim.AnalyticBeam` as the beam models. Here,
# `pyuvsim.simsetup.initialize_uvdata_from_param` returns `UVData`, `BeamList`,
# and `beam_ids`, but we will discard and replace the `BeamList` with a new
# `BeamList` that uses the `PolyBeam` model.
uvdata, _, beam_ids = initialize_uvdata_from_params(config_file)
# Fill `data_array` with zeros and commplete the UVData object.
_complete_uvdata(uvdata, inplace=True)

# Construct a new `BeamList` with polarized `PolyBeam`.
# `beam_coeffs` from Choudhuri et al 2020.
cfg_pol_beam = dict(
    ref_freq=1e8,
    spectral_index=-0.6975,
    beam_coeffs=[
        2.35088101e-01,
```

(continues on next page)

(continued from previous page)

```
        -4.20162599e-01,
        2.99189140e-01,
        -1.54189057e-01,
        3.38651457e-02,
        3.46936067e-02,
        -4.98838130e-02,
        3.23054464e-02,
        -7.56006552e-03,
        -7.24620596e-03,
        7.99563166e-03,
        -2.78125602e-03,
        -8.19945835e-04,
        1.13791191e-03,
        -1.24301372e-04,
        -3.74808752e-04,
        1.93997376e-04,
        -1.72012040e-05,
    ],
    polarized=True,
)
beams = BeamList([PolyBeam(**cfg_pol_beam)])

# Load the SkyModel. Same model used in Example 1.
sky_model_file = (
    DATA_PATH /
    'tutorials_data/visibility_simulator/gleam_top50.skyh5'
).as_posix()
sky_model = SkyModel()
sky_model.read_skyh5(sky_model_file)

# Construct ModelData
data_model_polybeam = ModelData(
    uvdata=uvdata,
    sky_model=sky_model,
    beam_ids=beam_ids,
    beams = beams
)
```

Constructing a SkyModel

`pyradiosky.SkyModel` is both a container and an interface for representing astrophysical radio sources.

A `SkyModel` object will usually consist of many components that represent either compact point sources or pixels of a HEALPix map. This choice can be set by specifying `component_type="point"` or `component_type="healpix"` when constructing the object. Doing so will determine which parameters are required as followed. * If `component_type="point"`, the `name` of each source and the source position (`ra` and `dec`) are required. * If `component_type="healpix"`, the `nside` parameter of the HEALPix map, the indices of the HEALPix pixel `hpx_inds`, and the HEALPix pixel ordering `hpx_order` must be set. * If `component_type` is not set, the type is infered from weather `nside` is given.

The component flux is given per frequency as an array of Stoke IQUV via the `stokes` argument (Shape: (4, Nfreqs, Ncomponents)). In addition, the `spectral_type` must be specified, which can be: * “flat” : Flat spectrum. * “full”

: Flux is defined at each frequency given in the `freq_array` attribute. * "subband" : Flux is defined at a set of band centers give in the `freq_array` attribute. * "spectral_index" : Flux is defined at a reference frequency set in the `reference_frequency` attribute and follows a power law given a spectral index set in the `spectral_index` attribute.

The method `at_frequencies` can be used to evaluate the stokes array of a SkyModel object at specified frequencies, converting the object to a "full" spectral type when `inplace=True`, or yielding a new instance of the object with a "full" spectral type if `inplace=False`.

The classmethods `pyradiosky.from_gleam_catalog`, `pyradiosky.from_votable_catalog`, and `pyradiosky.from_text_catalog` can be used to `SkyModel` object from the GLEAM EGC catalog file, a votable catalog file, or from a tab separated value file. The `pyuvsim.simsetup.create_mock_catalog` method can also be used to create a mock catalog of point source and/or diffuse sky

Loading a HEALPix map into a SkyModel object is currently only possible via the native skyh5 file format. Thus, a diffuse sky map in a HEALPix FITS format (e.g., Haslam 408 MHz) must be manually initialized from the HEALPix array values and indices.

A method for converting a `SkyModel` object from one component type to another is provided, which can be used to convert a "healpix" model to "point" and combine with another "point"-type model, for example. This conversion is used by `hera_sim` in its VisCPU and healvis wrappers.

Below, we show how to create a random point source model and a uniformly distrubuted diffuse model, and combine them into a single SkyModel object. Please refer to the pyradiosky developer API for details of the objects parameters and methods.

Example 6: Construct a random point source SkyModel

[6]:
```
from pyradiosky import SkyModel
from astropy.coordinates import Longitude, Latitude
from astropy import units as u

## Point Soruce Model ##
component_type = 'point'

# Each source is a component in a SkyModel.
# Create a sky model with 50 random sources (i.e. 50 point source components).
nsources = 100

# `ra` and `dec` must be Longitude and Latitude astropy quantity
ra = Longitude(np.random.uniform(0, 360, nsources) * u.degree)
dec = Latitude(np.random.uniform(-90, 90, nsources) * u.degree)

# `names` are required for point soruces
names = [f'rand{i:02d}' for i in range(nsources)]

# SkyModel supports several spectral type. We will use "spectral_index" here,
# where the flux of each source is describe by a power law given a reference
# frequency and a spectral index.
spectral_type = 'spectral_index'
spectral_index = np.random.uniform(-4, 3, nsources)
reference_frequency = 200e6 * np.ones(nsources) * u.Hz

# Source fluxes must be given in Stokes IQUV as an array of astropy quantity
# with an appropriate unit. Shape (4, Nfreqs, Ncomponents).
```

(continues on next page)

(continued from previous page)

```
# Nfreqs is 1 here because we are using "spctral_index".
stokes = np.zeros((4, 1, nsources)) * u.Jy
stokes[0, :, :] = np.random.uniform(10, 100, nsources) * u.Jy
# Let's also give our sources ~10% linar polarization
stokes[1, :, :] = np.random.uniform(1, 5, nsources) * u.Jy
stokes[2, :, :] = np.random.uniform(1, 5, nsources) * u.Jy

# Collect all parameters in a dictionary to be passed to a SkyModel constructor
source_params = {
    'component_type': component_type,
    'name': names,
    'ra': ra,
    'dec': dec,
    'stokes': stokes,
    'spectral_type': spectral_type,
    'reference_frequency': reference_frequency,
    'spectral_index': spectral_index,
    'history': ' Generate a random polarized source catalog with 50 sources.\n'
}

source_model = SkyModel(**source_params)

print('SkyModel History:\n', source_model.history)
print('Type of sky model:', source_model.component_type)
print('Number of components = number of sources:', source_model.Ncomponents)
```

```
SkyModel History:
  Generate a random polarized source catalog with 50 sources.
  Read/written with pyradiosky version: 0.1.2.
Type of sky model: point
Number of components = number of sources: 100
```

Example 7: Construct a uniformly distributed diffuse HEALPix SkyModel

[7]:
```
# Create a diffuse HEALPix SkyModel with a "rainbow" sky and flat spectra.
component_type = 'healpix'
nside = 2 ** 3
npix = 12 * nside ** 2
hpx_order = 'ring'
hpx_inds = np.arange(npix)

spectral_type = 'flat'

stokes = np.zeros((4, 1, npix)) * u.K
stokes[0, :, :] = np.random.uniform(400, 500, npix) * u.K

diffuse_params = {
    'component_type': component_type,
    'nside': nside,
    'hpx_inds': hpx_inds,
    'hpx_order': hpx_order,
```

(continues on next page)

(continued from previous page)

```
    'spectral_type' : spectral_type,
    'stokes': stokes,
    'history': ' Create a diffuse "rainbow" sky with a flat spectra.\n'
}
diffuse_model = SkyModel(**diffuse_params)
print('SkyModel History:\n', diffuse_model.history)
print('Type of sky model:', diffuse_model.component_type)
print('Nside:', diffuse_model.nside)
print('Number of components = number of healpix pixels:',
      diffuse_model.Ncomponents)
```

```
SkyModel History:
  Create a diffuse "rainbow" sky with a flat spectra.
  Read/written with pyradiosky version: 0.1.2.
Type of sky model: healpix
Nside: 8
Number of components = number of healpix pixels: 768
```

Example 8: Convert one SkyModel type to another and concatnate two SkyModel objects

One model type can be converted to another type and concatenated to form a composite model. The concatnation can only be done if `component_type`, `spectral_type`, and `freq_array` of both objects matches

[8]:
```
# Evaluate both sky models at frequencies in Example 1 simulation before
# concatenating.
frequencies = simulation.data_model.uvdata.freq_array[0] * u.Hz
source_model_full = source_model.at_frequencies(frequencies, inplace=False)
diffuse_model_full = diffuse_model.at_frequencies(frequencies, inplace=False)

# Drop spectral_index from the source model before combining.
# See GitHub issue https://github.com/RadioAstronomySoftwareGroup/pyradiosky/issues/160
source_model_full.spectral_index = None

# Convert the diffuse model to "point" type, making sure the flux unit
# is also converted to jansky
diffuse_model_full.healpix_to_point(to_jy=True)
composite_model = source_model_full.concat(diffuse_model_full, inplace=False)
```

Let's run some simulations with these sky models and other speciication the same as in Example 1 and compare the output visibilities.

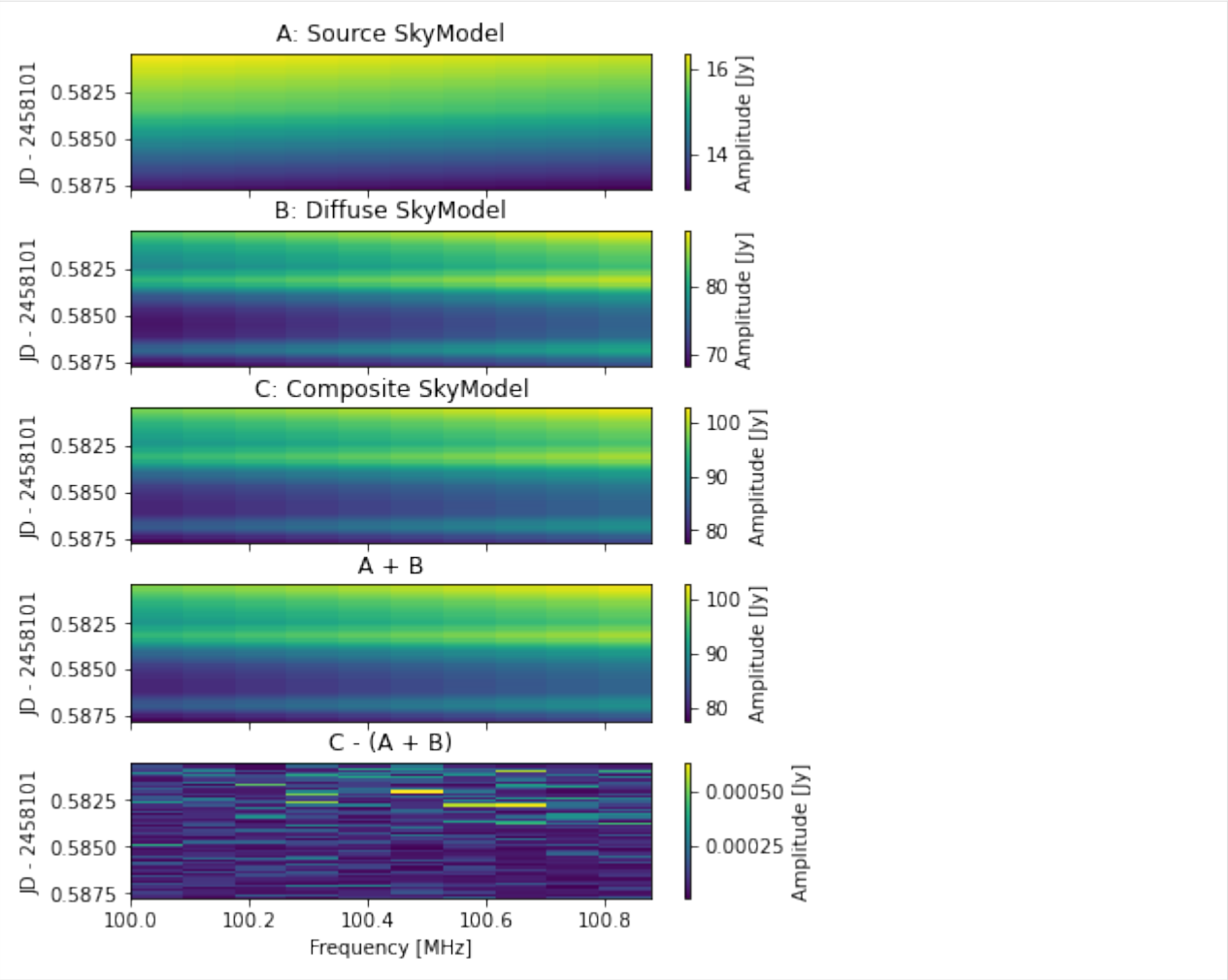
[9]:
```
# Define a function to run the simulation.
def run_simulation(sky_model):
    """Create and run a simulation given a sky model."""
    # Use the same config file as in Example 1 to initialize UVdata and beams.
    uvdata, beams, beam_ids = initialize_uvdata_from_params(config_file)
    _complete_uvdata(uvdata, inplace=True)

    # Initialize ModelData given the SkyModel
    data_model = ModelData(uvdata=uvdata, sky_model=sky_model,
                           beams=beams, beam_ids=beam_ids)
```

(continues on next page)

(continued from previous page)

```
    # Initialize a VisibilitySimulation instance, using VisCPU
    simulation = VisibilitySimulation(
        data_model=data_model,
        simulator=VisCPU(use_pixel_beams=True)
    )
    _  = simulation.simulate()

    return simulation


source_sim = run_simulation(source_model_full)
diffuse_sim = run_simulation(diffuse_model_full)
composite_sim = run_simulation(composite_model)
```

[10]:
```
# Define some labels for all the cases that we want to compare
waterfall_labels = ['A: Source SkyModel', 'B: Diffuse SkyModel',
                    'C: Composite SkyModel',
                    'A + B', 'C - (A + B)']

# Get waterfalls of the complex visibility,
# same baseline (0, 1, 'xx') as in Example 1.
waterfall_list = [sim.data_model.uvdata.get_data(bls)
                  for sim in [source_sim, diffuse_sim, composite_sim]]

# Make A + B, and C - (A + B), add them to the list
waterfall_list.append(waterfall_list[0] + waterfall_list[1])
waterfall_list.append(waterfall_list[2] - waterfall_list[3])

# Extent for the waterfall plot, same for all three simulations
extent = [source_sim.uvdata.freq_array[0, 0] / 1e6,
          source_sim.uvdata.freq_array[0, -1] / 1e6,
          source_sim.uvdata.time_array[-1] - 2458101,
          source_sim.uvdata.time_array[0] - 2458101]

# Plotting
fig, ax = plt.subplots(5, 1, sharex='all', sharey='all', figsize=(6, 8))
for i, waterfall in enumerate(waterfall_list):
    im = ax[i].imshow(np.abs(waterfall), aspect='auto', interpolation='none',
                      extent=extent)
    fig.colorbar(im, ax=ax[i], label='Amplitude [Jy]')
    ax[i].set_title(waterfall_labels[i])
    ax[i].set_ylabel('JD - 2458101')
ax[-1].set_xlabel('Frequency [MHz]')
fig.subplots_adjust(hspace=0.3)
```



Simulator Choices

`hera_sim.visibilities` provide wraper classes for three visibility simulators, allowing them to be used in the new uniform interface. These classes are actually subclasses of the `VisibilitySimulator` class, a new abstract class that serves as a template for wrapping visibility simulators to be used with the uninorm interface, making it easy to create your own wrapper for a different simulator.

Each of the simulator approaches the visibility calculation differently with its own merits.

`VisCPU` is the wrapper for the vis_cpu package, a Python/numpy-based simulator for interferometer visibilities developed by members from HERA collaboration. It models the sky as an ensemble of point sources, each with their own frequency spectrum. The code is capable of modelling polarized visibilities and primary beams, but currently only a Stokes I sky model.

The HERA Memo #98 describes the mathematical underlying of the `vis_cpu` simulator although be warned that some information might have already been obsolete due to the recent rapid development of the simulator. In summary, vis_cpu computes a geometric delay for each source and antenna, which is used to calculate the phase factor that is then multiplied by the source flux to obtain per-antenna visibility factor. The direction-dependent polarized antenna beam factor (Jones matrix) is computed by evaluating the complex (efield) antenna pattern at each source position on the azimutal-zenithal coordinates. This is usually done by calling the `interp` method of the UVBeam object, or by directly evaluating an analytic function in the case of using an analytic beam model. All of these are done per time at a given *single* frequency.

The `hera_sim` wrapper is default to used the "pixel beam" mode, which interpolates each anteanna beam once onto an (l, m) grid that is then used to constructs a 2D Bivariate spline. The spline object is evaluated given a source position to estimate the antenna beam pattern. This bypasses the need to compute the new (l, m) coordinates at every time, and, although less accurate, is usually faster for complex efield beams. However, analytic beam models such as the `PolyBeam` object can precisely and quickly give the beam value at any given az-za coordinate. Thus, pixel beam mode should *not* be used when using analytic beam models.

The `hera_sim` wrapper also adds a frequency loop on top of the internal time loop in vis_cpu codes and interfaces between the `ModelData` to vis_cpu input parameters, including transformation of a HEALPix SkyModel into a point source model. MPI parallelization over frequencies is also implemented on the `hera_sim` wrapper although shared memory is currently not supported.

The `HealVis` class provide a wrapper for the healvis simulator. healvis simulates visibility off of HEALPix shells by directly evaluating the radio interferometry measurement equation (RIME) at the healpix pixels, avoiding artifacts from sky projection (see Section 3. of Lanman et al. 2020 for the underlying mathematical formalism). Due to this unique calculation, a point source catalog must be gridded onto a healpix map to use in healvis. The `hera_sim` wrapper facilites this conversion through the interface between the `HealVis` wrapper and the `ModelData` object.

The `UVSim` class provides a `hera_sim` wrapper to pyuvsim, a comprehensive simulation package for radio interferometers that emphasizes accuracy and extensibility over speed. Under the hood, it execute `pyuvsim.uvsim.run_uvdata_uvsim` given the information in the `ModelData` object.

The `VisibilitySimulator` abstract base class defines the required class infrastructure for creating a custom simulator.

[11]:
```
from hera_sim.visibilities import VisibilitySimulator

VisibilitySimulator??
```

```
Init signature: VisibilitySimulator()
Source:
class VisibilitySimulator(metaclass=ABCMeta):
    """Base class for all hera_sim compatible visibility simulators."""

    #: Whether this particular simulator has the ability to simulate point
    #: sources directly.
    point_source_ability = True

    #: Whether this particular simulator has the ability to simulate diffuse
    #: maps directly.
    diffuse_ability = False

    __version__ = "unknown"

    @abstractmethod
    def simulate(self, data_model: ModelData) -> np.ndarray:
        """Simulate the visibilities."""
        pass

    def validate(self, data_model: ModelData):
        """Check that the data model complies with the assumptions of the simulator."""
        pass
File:           ~/src/miniconda3/envs/hera_sim_dev/lib/python3.8/site-packages/hera_sim/
↪visibilities/simulators.py
Type:           ABCMeta
Subclasses:     UVSim, VisCPU, HealVis
```

(continues on next page)

(continued from previous page)

A custom simulator can be made by subclassing the `VisibilitySimulator` class and overriding the `simulate` method, which must take a `ModelData` object as an input and must return a Numpy array with the same shap as `UVData.data_array`. The `validate` method may be override, and `point_source_ability` and `diffuse_abilityatrributes` may be set, as neccessary.

Let's take a look at the codes of the `UVSim` class that wraps the pyuvsim simulator as an example. It defines a `simulate` method that calls `pyuvsim.uvsim.run_uvdata_uvsim`, passing in a UVData object from the ModelData input, and returning the `data_array` from the UVData output.

[12]:
```
from hera_sim.visibilities import UVSim

UVSim??
```

```
Init signature: UVSim(quiet: bool = False)
Source:
class UVSim(VisibilitySimulator):
    """A wrapper around the pyuvsim simulator.

    Parameters
    ----------
    quiet
        If True, don't print anything.
    """

    def __init__(self, quiet: bool = False):
        self.quiet = quiet

    def simulate(self, data_model: ModelData):
        """Simulate the visibilities."""
        beam_dict = {
            ant: data_model.beam_ids[str(idx)]
            for idx, ant in enumerate(data_model.uvdata.antenna_names)
        }

        # TODO: this can be removed once
        # https://github.com/RadioAstronomySoftwareGroup/pyuvsim/pull/357
        # is mereged.
        if data_model.sky_model.name is not None:
            data_model.sky_model.name = np.array(data_model.sky_model.name)

        warnings.warn(
            "UVSim requires time-ordered data. Ensuring that order in UVData..."
        )
        data_model.uvdata.reorder_blts("time")

        # The UVData object must have correctly ordered pols.
        # TODO: either remove this when pyuvsim fixes bug with ordering
        # (https://github.com/RadioAstronomySoftwareGroup/pyuvsim/issues/370) or
        # at least check whether reordering is necessary once uvdata has that ability.
        if np.any(data_model.uvdata.polarization_array != np.array([-5, -6, -7, -8])):
            warnings.warn(
```

(continues on next page)

(continued from previous page)

```
                "In UVSim, polarization array must be in AIPS order. Reordering..."
            )
            data_model.uvdata.reorder_pols("AIPS")

        out_uv = pyuvsim.uvsim.run_uvdata_uvsim(
            input_uv=data_model.uvdata,
            beam_list=data_model.beams,
            beam_dict=beam_dict,
            catalog=pyuvsim.simsetup.SkyModelData(data_model.sky_model),
            quiet=self.quiet,
        )
        return out_uv.data_array
File:           ~/src/miniconda3/envs/hera_sim_dev/lib/python3.8/site-packages/hera_sim/
↪visibilities/pyuvsim_wrapper.py
Type:           ABCMeta
Subclasses:
```

[ ]:

5.1.6 Running `hera-sim-vis` from the command line

As of v2.4.0 of `hera_sim`, we have included a command-line interface for performing visibility simulations using *any* compatible visiblity simulator. The interface for the script is (this can be run from anywhere if `hera_sim` is installed):

```
$ hera-sim-vis.py --help
usage: hera-sim-vis.py [-h] [--object_name OBJECT_NAME] [--compress COMPRESS]
                       [--normalize_beams] [--fix_autos]
                       [--max-auto-imag MAX_AUTO_IMAG] [-d] [--run-auto-check]
                       [--profile] [--profile-funcs PROFILE_FUNCS]
                       [--profile-output PROFILE_OUTPUT]
                       [--log-level {INFO,ERROR,WARNING,CRITICAL,DEBUG}]
                       [--log-width LOG_WIDTH] [--log-plain-tracebacks]
                       [--log-absolute-time] [--log-no-mem]
                       [--log-mem-backend {tracemalloc,psutil}]
                       [--log-show-path]
                       obsparam simulator_config

Run vis_cpu via hera_sim given an obsparam.

positional arguments:
  obsparam              pyuvsim-formatted obsparam file.
  simulator_config      YAML configuration file for the simulator.

options:
  -h, --help            show this help message and exit
  --object_name OBJECT_NAME
                        Set object_name in the UVData
  --compress COMPRESS   Compress by redundancy. A file name to store the
                        cache.
```

(continues on next page)

(continued from previous page)

```
  --normalize_beams     Peak normalize the beams.
  --fix_autos           Check and fix non-real xx/yy autos
  --max-auto-imag MAX_AUTO_IMAG
                        Maximum fraction of imaginary/absolute for autos
                        before raising an error
  -d, --dry-run         If set, create the simulator and data model but don't
                        run simulation.
  --run-auto-check      whether to check autos are real

Options for line-profiling:
  --profile             Line-Profile the script
  --profile-funcs PROFILE_FUNCS
                        List of functions to profile
  --profile-output PROFILE_OUTPUT
                        Output file for profiling info.

Options for logging:
  --log-level {INFO,ERROR,WARNING,CRITICAL,DEBUG}
                        logging level to display.
  --log-width LOG_WIDTH
                        width of logging output
  --log-plain-tracebacks
                        use plain instead of rich tracebacks
  --log-absolute-time   show logger time as absolute instead of relative to
                        start
  --log-no-mem          do not show memory usage
  --log-mem-backend {tracemalloc,psutil}
  --log-show-path       show path of code in log msg
```

Two main configuration files are required. The first is an "obsparam" file, which should follow the formatting defined by pyuvsim. As described in the visibility simulator tutorial, the `hera_sim.visibilities` module provides a universal interface between this particular configuration setup and a number of particular simulators.

To specify the simulator to be used, the second configuration file must be provided. An example of this configuration file, defined for the `VisCPU` simulator, can be found in the repo's config_examples directory. Here are its contents:

```
$ cat -n ../config_examples/simulator.yaml
     1        simulator: MatVis
     2        precision: 2
     3        ref_time: mean
     4        correct_source_positions: true
```

Notice that the file contains a `simulator:` entry. This must be the name of a simulator derived from the base `VisibilitySimulator` class provided in `hera_sim`. Usually, this will be one of the built-in simulators listed in the API reference under "Built-In Simulators".

However, it may also be a custom simulator, defined outside of `hera_sim`, so long as it inherits from `VisibilitySimulator`. To use one of these via command line, you need to provide the dot-path to the class, so for example: `my_library.module.MySimulator`.

The remaining parameters are passed to the constructor for the given simulator. So, for example, for the `VisCPU` simulator, all available parameters are listed under the Parameters section of its class API. In general, the class may do some transformation of the YAML inputs before constructing its instance, using the `from_yaml_dict()` method. Check out the documentation for that method for your particular simulator to check if it requires any transformations (eg. if it

required data loaded from a file, it might take the filename from the YAML, and read the file in its `from_yaml_dict()` method before constructing the object with the full data).

5.2 API Reference

5.2.1 Modules

hera_sim.vis	Functions for producing white-noise redundant visibilities.
hera_sim.antpos	A module for creating antenna array configurations.
hera_sim.defaults	Class for dynamically changing hera_sim parameter defaults.
hera_sim.eor	A module for simulating EoR-like visibilities.
hera_sim.foregrounds	Visibility-space foreground models.
hera_sim.interpolators	This module provides interfaces to different interpolation classes.
hera_sim.io	Methods for input/output of data.
hera_sim.noise	Models of thermal noise.
hera_sim.rfi	Models of radio frequency interference.
hera_sim.sigchain	Models of signal-chain systematics.
hera_sim.simulate	Module containing a high-level interface for `hera_sim`.
hera_sim.utils	Utility module.
hera_sim.cli_utils	Useful helper functions and argparsers for running simulations via CLI.
hera_sim.components	A module providing discoverability features for hera_sim.

hera_sim.vis

Functions for producing white-noise redundant visibilities.

Functions

sim_red_data(reds[, gains, shape, gain_scatter])	Simulate thermal-noise-free random but redundant (up to gains) visibilities.

hera_sim.vis.sim_red_data

hera_sim.vis.**sim_red_data**(*reds*, *gains=None*, *shape=(10, 10)*, *gain_scatter=0.1*)

Simulate thermal-noise-free random but redundant (up to gains) visibilities.

Parameters

- **reds** (*list of list of tuples*) – list of lists of baseline-pol tuples where each sublist has only redundant pairs
- **gains** (*dict*) – pre-specify base gains to then scatter on top of in the {(index,antpol): ndarray} format. Default gives all ones.

- **shape** (*tuple*) – (Ntimes, Nfreqs).
- **gain_scatter** (*float*) – relative amplitude of per-antenna complex gain scatter

Returns

- *dict* – true gains used in the simulation in the {(index,antpol): np.array} format
- *dict* – true underlying visibilities in the {(ind1,ind2,pol): np.array} format
- *dict* – simulated visibilities in the {(ind1,ind2,pol): np.array} format

hera_sim.antpos

A module for creating antenna array configurations.

Input parameters vary between functions, but all functions return a dictionary whose keys refer to antenna numbers and whose values refer to the ENU position of the antennas.

Classes

Array(**kwargs)	Base class for constructing telescope array objects.
HexArray([sep, split_core, outriggers])	Build a hexagonal array configuration, nominally matching HERA.
LinearArray([sep])	Build a linear (east-west) array configuration.

hera_sim.antpos.Array

class hera_sim.antpos.**Array**(***kwargs*)

Base class for constructing telescope array objects.

This is an *abstract* class, and should not be directly instantiated. It represents a "component" – a modular part of a simulation for which several models may be defined. Models for this component may be defined by subclassing this abstract base class and implementing (at least) the *__call__()* method. Some of these are implemented within hera_sim already, but custom models may be implemented outside of hera_sim, and used on equal footing with the the internal models (as long as they subclass this abstract component).

As with all components, all parameters that define the behaviour of the model are accepted at class instantiation. The *__call__()* method actually computes the simulated effect of the component (typically, but not always, a set of visibilities or gains), by *default* using these parameters. However, these parameters can be over-ridden at call-time. Inputs such as the frequencies, times or baselines at which to compute the effect are specific to the call, and do not get passed at instantiation.

Methods

__call__(**kwargs)	Compute the component model.
get_aliases()	Get all the aliases by which this model can be identified.
get_model(mdl)	Get a model with a particular name (including aliases).
get_models([with_aliases])	Get a dictionary of models associated with this component.

hera_sim.antpos.Array.__call__

abstract Array.**__call__**(***kwargs*)

Compute the component model.

hera_sim.antpos.Array.get_aliases

classmethod Array.**get_aliases**() → tuple[str]

Get all the aliases by which this model can be identified.

hera_sim.antpos.Array.get_model

classmethod Array.**get_model**(*mdl: str*) → *SimulationComponent*

Get a model with a particular name (including aliases).

hera_sim.antpos.Array.get_models

classmethod Array.**get_models**(*with_aliases=False*) → dict[str, *hera_sim.components.SimulationComponent*]

Get a dictionary of models associated with this component.

Attributes

attrs_to_pull	Mapping between parameter names and Simulator attributes
is_multiplicative	Whether this systematic multiplies existing visibilities
is_randomized	Whether this systematic contains a randomized component
return_type	Whether the returned value is per-antenna, per-baseline, or the full array

hera_sim.antpos.Array.attrs_to_pull

Array.**attrs_to_pull: dict = {}**

Mapping between parameter names and Simulator attributes

hera_sim.antpos.Array.is_multiplicative

Array.**is_multiplicative: bool = False**

Whether this systematic multiplies existing visibilities

hera_sim.antpos.Array.is_randomized

Array.**is_randomized: bool = False**

Whether this systematic contains a randomized component

hera_sim.antpos.Array.return_type

Array.**return_type: str | None = None**

Whether the returned value is per-antenna, per-baseline, or the full array

hera_sim.antpos.HexArray

class hera_sim.antpos.**HexArray**(*sep=14.6*, *split_core=True*, *outriggers=2*)

Build a hexagonal array configuration, nominally matching HERA.

Parameters

- **sep** (*int, optional*) – The separation between adjacent grid points, in meters. Default separation is 14.6 meters.
- **split_core** (*bool, optional*) – Whether to fracture the core into tridents that subdivide a hexagonal grid. Loses N antennas. Default behavior is to split the core.
- **outriggers** (*int, optional*) – The number of rings of outriggers to add to the array. The outriggers tile with the core to produce a fully-sampled UV plane. The first ring corresponds to the exterior of a hex_num=3 hexagon. For R outriggers, $3R^2 + 9R$ antennas are added to the array.

Methods

__call__(hex_num, **kwargs)	Compute the positions of the antennas.
get_aliases()	Get all the aliases by which this model can be identified.
get_model(mdl)	Get a model with a particular name (including aliases).
get_models([with_aliases])	Get a dictionary of models associated with this component.

hera_sim.antpos.HexArray.__call__

HexArray.__call__(*hex_num*, ***kwargs*)

Compute the positions of the antennas.

Parameters

hex_num (*int*) – The hexagon (radial) number of the core configuration. The number of core antennas returned is $3N^2 - 3N + 1$.

Returns

antpos (*dict*) – Dictionary of antenna numbers and positions, in ENU coordinates. Antenna positions are given in units of meters.

hera_sim.antpos.HexArray.get_aliases

classmethod HexArray.**get_aliases**() → tuple[str]

Get all the aliases by which this model can be identified.

hera_sim.antpos.HexArray.get_model

classmethod HexArray.**get_model**(*mdl: str*) → *SimulationComponent*

Get a model with a particular name (including aliases).

hera_sim.antpos.HexArray.get_models

classmethod HexArray.**get_models**(*with_aliases=False*) → dict[str, *hera_sim.components.SimulationComponent*]

Get a dictionary of models associated with this component.

Attributes

attrs_to_pull	Mapping between parameter names and Simulator attributes
is_multiplicative	Whether this systematic multiplies existing visibilities
is_randomized	Whether this systematic contains a randomized component
return_type	Whether the returned value is per-antenna, per-baseline, or the full array

hera_sim.antpos.HexArray.attrs_to_pull

HexArray.**attrs_to_pull**: dict = {}

Mapping between parameter names and Simulator attributes

hera_sim.antpos.HexArray.is_multiplicative

HexArray.**is_multiplicative**: bool = False

Whether this systematic multiplies existing visibilities

hera_sim.antpos.HexArray.is_randomized

HexArray.**is_randomized**: bool = False

Whether this systematic contains a randomized component

hera_sim.antpos.HexArray.return_type

HexArray.**return_type**: str | None = None

Whether the returned value is per-antenna, per-baseline, or the full array

hera_sim.antpos.LinearArray

class hera_sim.antpos.**LinearArray**(*sep=14.6*)

Build a linear (east-west) array configuration.

Parameters

sep (*float, optional*) – The separation between adjacent antennas, in meters. Default separation is 14.6 meters.

Methods

__call__(nants, **kwargs)	Compute the antenna positions.
get_aliases()	Get all the aliases by which this model can be identified.
get_model(mdl)	Get a model with a particular name (including aliases).
get_models([with_aliases])	Get a dictionary of models associated with this component.

hera_sim.antpos.LinearArray.__call__

LinearArray.**__call__**(*nants*, ***kwargs*)

Compute the antenna positions.

Parameters

nants (*int*) – The number of antennas in the configuration.

Returns

antpos (*dict*) – Dictionary of antenna numbers and ENU positions. Positions are given in meters.

hera_sim.antpos.LinearArray.get_aliases

classmethod LinearArray.**get_aliases**() → tuple[str]

Get all the aliases by which this model can be identified.

hera_sim.antpos.LinearArray.get_model

classmethod LinearArray.**get_model**(*mdl: str*) → *SimulationComponent*

Get a model with a particular name (including aliases).

hera_sim.antpos.LinearArray.get_models

classmethod LinearArray.**get_models**(*with_aliases=False*) → dict[str, *hera_sim.components.SimulationComponent*]

Get a dictionary of models associated with this component.

Attributes

attrs_to_pull	Mapping between parameter names and Simulator attributes
is_multiplicative	Whether this systematic multiplies existing visibilities
is_randomized	Whether this systematic contains a randomized component
return_type	Whether the returned value is per-antenna, per-baseline, or the full array

hera_sim.antpos.LinearArray.attrs_to_pull

LinearArray.**attrs_to_pull**: dict = {}

Mapping between parameter names and Simulator attributes

hera_sim.antpos.LinearArray.is_multiplicative

LinearArray.**is_multiplicative**: bool = False

Whether this systematic multiplies existing visibilities

hera_sim.antpos.LinearArray.is_randomized

LinearArray.**is_randomized**: bool = False

Whether this systematic contains a randomized component

hera_sim.antpos.LinearArray.return_type

LinearArray.**return_type**: str | None = None

Whether the returned value is per-antenna, per-baseline, or the full array

hera_sim.defaults

Module for interfacing with package-wide default parameters.

hera_sim.eor

A module for simulating EoR-like visibilities.

EoR models should require lsts, frequencies, and a baseline vector as arguments, and may have arbitrary optional parameters. Models should return complex-valued arrays with shape (Nlsts, Nfreqs) that represent a visibility appropriate for the given baseline.

Classes

EoR(**kwargs)	Base class for fast EoR simualtors.
NoiselikeEoR([eor_amp, min_delay, ...])	Generate a noiselike, fringe-filtered EoR visibility.

hera_sim.eor.EoR

class hera_sim.eor.**EoR**(***kwargs*)

Base class for fast EoR simualtors.

This is an *abstract* class, and should not be directly instantiated. It represents a "component" – a modular part of a simulation for which several models may be defined. Models for this component may be defined by subclassing this abstract base class and implementing (at least) the *__call__()* method. Some of these are implemented within hera_sim already, but custom models may be implemented outside of hera_sim, and used on equal footing with the the internal models (as long as they subclass this abstract component).

As with all components, all parameters that define the behaviour of the model are accepted at class instantiation. The *__call__()* method actually computes the simulated effect of the component (typically, but not always, a set of visibilities or gains), by *default* using these parameters. However, these parameters can be over-ridden at call-time. Inputs such as the frequencies, times or baselines at which to compute the effect are specific to the call, and do not get passed at instantiation.

Methods

__call__(**kwargs)	Compute the component model.
get_aliases()	Get all the aliases by which this model can be identified.
get_model(mdl)	Get a model with a particular name (including aliases).
get_models([with_aliases])	Get a dictionary of models associated with this component.

hera_sim.eor.EoR.__call__

abstract EoR.**__call__**(***kwargs*)

Compute the component model.

hera_sim.eor.EoR.get_aliases

classmethod EoR.**get_aliases**() → tuple[str]

Get all the aliases by which this model can be identified.

hera_sim.eor.EoR.get_model

classmethod EoR.**get_model**(*mdl: str*) → *SimulationComponent*

Get a model with a particular name (including aliases).

hera_sim.eor.EoR.get_models

classmethod EoR.**get_models**(*with_aliases=False*) → dict[str, *hera_sim.components.SimulationComponent*]

Get a dictionary of models associated with this component.

Attributes

attrs_to_pull	Mapping between parameter names and Simulator attributes
is_multiplicative	Whether this systematic multiplies existing visibilities
is_randomized	Whether this systematic contains a randomized component
return_type	Whether the returned value is per-antenna, per-baseline, or the full array

hera_sim.eor.EoR.attrs_to_pull

EoR.**attrs_to_pull**: dict = {}

Mapping between parameter names and Simulator attributes

hera_sim.eor.EoR.is_multiplicative

EoR.**is_multiplicative**: bool = False

Whether this systematic multiplies existing visibilities

hera_sim.eor.EoR.is_randomized

EoR.**is_randomized**: bool = False

Whether this systematic contains a randomized component

hera_sim.eor.EoR.return_type

EoR.**return_type**: str | None = None

Whether the returned value is per-antenna, per-baseline, or the full array

hera_sim.eor.NoiselikeEoR

class hera_sim.eor.**NoiselikeEoR**(*eor_amp: float = 1e-05*, *min_delay: float | None = None*, *max_delay: float | None = None*, *fringe_filter_type: str = 'tophat'*, *fringe_filter_kwargs: dict | None = None*, *rng: Generator | None = None*)

Generate a noiselike, fringe-filtered EoR visibility.

Parameters

- **eor_amp** – The amplitude of the EoR power spectrum.
- **min_delay** – Minimum delay to allow through the delay filter. Default is -inf.
- **max_delay** – Maximum delay to allow through the delay filter. Default is +inf
- **fringe_filter_type** – The kind of filter to apply in fringe-space.
- **fringe_filter_kwargs** – Arguments to pass to the fringe filter. See `utils.rough_fringe_filter()` for possible arguments.

Notes

This algorithm produces visibilities as a function of time/frequency that have white noise structure, filtered over the delay and fringe-rate axes. The fringe-rate filter makes the data look more like EoR by constraining it to moving with the sky (given the baseline vector).

Methods

__call__(lsts, freqs, bl_vec, **kwargs)	Compute the noise-like EoR model.
get_aliases()	Get all the aliases by which this model can be identified.
get_model(mdl)	Get a model with a particular name (including aliases).
get_models([with_aliases])	Get a dictionary of models associated with this component.

hera_sim.eor.NoiselikeEoR.__call__

NoiselikeEoR.**__call__**(*lsts*, *freqs*, *bl_vec*, ***kwargs*)

Compute the noise-like EoR model.

hera_sim.eor.NoiselikeEoR.get_aliases

classmethod NoiselikeEoR.**get_aliases**() → tuple[str]

Get all the aliases by which this model can be identified.

hera_sim.eor.NoiselikeEoR.get_model

classmethod NoiselikeEoR.**get_model**(*mdl: str*) → *SimulationComponent*

Get a model with a particular name (including aliases).

hera_sim.eor.NoiselikeEoR.get_models

classmethod NoiselikeEoR.**get_models**(*with_aliases=False*) → dict[str, *hera_sim.components.SimulationComponent*]

Get a dictionary of models associated with this component.

Attributes

attrs_to_pull	Mapping between parameter names and Simulator attributes
is_multiplicative	Whether this systematic multiplies existing visibilities
is_randomized	Whether this systematic contains a randomized component
is_smooth_in_freq	
return_type	Whether the returned value is per-antenna, per-baseline, or the full array

hera_sim.eor.NoiselikeEoR.attrs_to_pull

NoiselikeEoR.**attrs_to_pull: dict = {'bl_vec': None}**

Mapping between parameter names and Simulator attributes

hera_sim.eor.NoiselikeEoR.is_multiplicative

NoiselikeEoR.**is_multiplicative: bool = False**

Whether this systematic multiplies existing visibilities

hera_sim.eor.NoiselikeEoR.is_randomized

NoiselikeEoR.**is_randomized: bool = True**

Whether this systematic contains a randomized component

hera_sim.eor.NoiselikeEoR.is_smooth_in_freq

NoiselikeEoR.**is_smooth_in_freq = False**

hera_sim.eor.NoiselikeEoR.return_type

NoiselikeEoR.**return_type**: str | None = **'per_baseline'**

Whether the returned value is per-antenna, per-baseline, or the full array

hera_sim.foregrounds

Visibility-space foreground models.

This module defines several cheap foreground models evaluated in visibility space.

Classes

DiffuseForeground([Tsky_mdl, omega_p, ...])	Produce a rough simulation of diffuse foreground-like structure.
Foreground(**kwargs)	Base class for foreground models.
PointSourceForeground([nsrcs, Smin, Smax, ...])	Produce a uniformly-random point-source sky observed with a truncated Gaussian beam.

hera_sim.foregrounds.DiffuseForeground

class hera_sim.foregrounds.**DiffuseForeground**(*Tsky_mdl=None*, *omega_p=None*, *delay_filter_kwargs=None*, *fringe_filter_kwargs=None*, *rng=None*)

Produce a rough simulation of diffuse foreground-like structure.

Parameters

- **Tsky_mdl** (*interpolation object*) – Sky temperature model, in units of Kelvin. Must be callable with signature Tsky_mdl(lsts, freqs), formatted so that lsts are in radians and freqs are in GHz.
- **omega_p** (*interpolation object or array-like of float*) – Beam size model, in units of steradian. If passing an array, then it must be the same shape as the frequency array passed to the `freqs` parameter.
- **delay_filter_kwargs** (*dict, optional*) – Keyword arguments and associated values to be passed to *rough_delay_filter()*. Default is to use the following settings: `standoff : 0.0`, `delay_filter_type : tophat`.
- **fringe_filter_kwargs** (*dict, optional*) – Keyword arguments and associated values to be passed to *rough_fringe_filter()*. Default is to use the following settings: `fringe_filter_type : tophat`.
- **rng** (*np.random.Generator, optional*) – Random number generator.

Notes

This algorithm provides a rough simulation of visibilities from diffuse foregrounds by using a sky temperature model. The sky temperature models provided in this package are appropriate for the HERA H1C observing season, and are only valid for frequencies between 100 MHz and 200 MHz; anything beyond this range is just a copy of the value at the nearest edge. Simulated autocorrelations (i.e. zero magnitude `bl_vec`) are returned as complex arrays, but have zero imaginary component everywhere. For cross-correlations, the sky model is convolved with white noise (in delay/fringe-rate space), and rough delay and fringe filters are applied to the visibility. As a standalone component model, this is does not produce consistent simulated visibilities for baselines within a redundant group (except for autocorrelations); however, the *Simulator* class provides the functionality to ensure that redundant baselines see the same sky. Additionally, visibilities simulated with this model are not invariant under complex conjugation and baseline conjugation, since the delay filter applied is symmetric; however, the *Simulator* class is aware of this and ensures invariance under complex conjugation and baseline conjugation.

Methods

__call__(lsts, freqs, bl_vec, **kwargs)	Compute the foregrounds.
get_aliases()	Get all the aliases by which this model can be identified.
get_model(mdl)	Get a model with a particular name (including aliases).
get_models([with_aliases])	Get a dictionary of models associated with this component.

hera_sim.foregrounds.DiffuseForeground.__call__

DiffuseForeground.**__call__**(*lsts*, *freqs*, *bl_vec*, ***kwargs*)

Compute the foregrounds.

Parameters

- **lsts** (*array-like of float*) – Array of LST values in units of radians.
- **freqs** (*array-like of float*) – Array of frequency values in units of GHz.
- **bl_vec** (*array-like of float*) – Length-3 array specifying the baseline vector in units of ns.

Returns

vis (*ndarray of complex*) – Array of visibilities at each LST and frequency appropriate for the given sky temperature model, beam size model, and baseline vector. Returned in units of Jy with shape (lsts.size, freqs.size).

hera_sim.foregrounds.DiffuseForeground.get_aliases

classmethod DiffuseForeground.**get_aliases**() → tuple[str]

Get all the aliases by which this model can be identified.

hera_sim.foregrounds.DiffuseForeground.get_model

classmethod DiffuseForeground.**get_model**(*mdl: str*) → *SimulationComponent*

Get a model with a particular name (including aliases).

hera_sim.foregrounds.DiffuseForeground.get_models

classmethod DiffuseForeground.**get_models**(*with_aliases=False*) → dict[str, *hera_sim.components.SimulationComponent*]

Get a dictionary of models associated with this component.

Attributes

attrs_to_pull	Mapping between parameter names and Simulator attributes
is_multiplicative	Whether this systematic multiplies existing visibilities
is_randomized	Whether this systematic contains a randomized component
is_smooth_in_freq	
return_type	Whether the returned value is per-antenna, per-baseline, or the full array

hera_sim.foregrounds.DiffuseForeground.attrs_to_pull

DiffuseForeground.**attrs_to_pull: dict = {'bl_vec': None}**

Mapping between parameter names and Simulator attributes

hera_sim.foregrounds.DiffuseForeground.is_multiplicative

DiffuseForeground.**is_multiplicative: bool = False**

Whether this systematic multiplies existing visibilities

hera_sim.foregrounds.DiffuseForeground.is_randomized

DiffuseForeground.**is_randomized: bool = True**

Whether this systematic contains a randomized component

hera_sim.foregrounds.DiffuseForeground.is_smooth_in_freq

DiffuseForeground.**is_smooth_in_freq = True**

hera_sim.foregrounds.DiffuseForeground.return_type

DiffuseForeground.**return_type: str | None = 'per_baseline'**

Whether the returned value is per-antenna, per-baseline, or the full array

hera_sim.foregrounds.Foreground

class hera_sim.foregrounds.**Foreground**(***kwargs*)

Base class for foreground models.

This is an *abstract* class, and should not be directly instantiated. It represents a "component" – a modular part of a simulation for which several models may be defined. Models for this component may be defined by subclassing this abstract base class and implementing (at least) the *__call__()* method. Some of these are implemented within hera_sim already, but custom models may be implemented outside of hera_sim, and used on equal footing with the the internal models (as long as they subclass this abstract component).

As with all components, all parameters that define the behaviour of the model are accepted at class instantiation. The *__call__()* method actually computes the simulated effect of the component (typically, but not always, a set of visibilities or gains), by *default* using these parameters. However, these parameters can be over-ridden at call-time. Inputs such as the frequencies, times or baselines at which to compute the effect are specific to the call, and do not get passed at instantiation.

Methods

__call__(**kwargs)	Compute the component model.
get_aliases()	Get all the aliases by which this model can be identified.
get_model(mdl)	Get a model with a particular name (including aliases).
get_models([with_aliases])	Get a dictionary of models associated with this component.

hera_sim.foregrounds.Foreground.__call__

abstract Foreground.**__call__**(***kwargs*)

Compute the component model.

hera_sim.foregrounds.Foreground.get_aliases

classmethod Foreground.**get_aliases**() → tuple[str]

Get all the aliases by which this model can be identified.

hera_sim.foregrounds.Foreground.get_model

classmethod Foreground.**get_model**(*mdl: str*) → *SimulationComponent*

Get a model with a particular name (including aliases).

hera_sim.foregrounds.Foreground.get_models

classmethod Foreground.**get_models**(*with_aliases=False*) → dict[str, *hera_sim.components.SimulationComponent*]

Get a dictionary of models associated with this component.

Attributes

attrs_to_pull	Mapping between parameter names and Simulator attributes
is_multiplicative	Whether this systematic multiplies existing visibilities
is_randomized	Whether this systematic contains a randomized component
return_type	Whether the returned value is per-antenna, per-baseline, or the full array

hera_sim.foregrounds.Foreground.attrs_to_pull

Foreground.**attrs_to_pull: dict = {}**

Mapping between parameter names and Simulator attributes

hera_sim.foregrounds.Foreground.is_multiplicative

Foreground.**is_multiplicative: bool = False**

Whether this systematic multiplies existing visibilities

hera_sim.foregrounds.Foreground.is_randomized

Foreground.**is_randomized**: bool = **False**

Whether this systematic contains a randomized component

hera_sim.foregrounds.Foreground.return_type

Foreground.**return_type**: str | None = **None**

Whether the returned value is per-antenna, per-baseline, or the full array

hera_sim.foregrounds.PointSourceForeground

class hera_sim.foregrounds.**PointSourceForeground**(*nsrcs=1000*, *Smin=0.3*, *Smax=300*, *beta=-1.5*, *spectral_index_mean=-1*, *spectral_index_std=0.5*, *reference_freq=0.15*, *rng=None*)

Produce a uniformly-random point-source sky observed with a truncated Gaussian beam.

Parameters

- **nsrcs** (*int, optional*) – Number of sources to place on the sky. Point sources are simulated to have a flux-density drawn from a power-law distribution specified by the `Smin`, `Smax`, and `beta` parameters. Additionally, each source has a chromatic flux-density given by a power law; the spectral index is drawn from a normal distribution with mean `spectral_index_mean` and standard deviation `spectral_index_std`.
- **Smin** (*float, optional*) – Lower bound of the power-law distribution to draw flux-densities from, in units of Jy.
- **Smax** (*float, optional*) – Upper bound of the power-law distribution to draw flux-densities from, in units of Jy.
- **beta** (*float, optional*) – Power law index for the source counts versus flux-density.
- **spectral_index_mean** (*float, optional*) – The mean of the normal distribution to draw source spectral indices from.
- **spectral_index_std** (*float, optional*) – The standard deviation of the normal distribution to draw source spectral indices from.
- **reference_freq** (*float, optional*) – Reference frequency used to make the point source flux densities chromatic, in units of GHz.
- **rng** (*np.random.Generator, optional*) – Random number generator.

Methods

__call__(lsts, freqs, bl_vec, **kwargs)	Compute the point source foregrounds.
get_aliases()	Get all the aliases by which this model can be identified.
get_model(mdl)	Get a model with a particular name (including aliases).
get_models([with_aliases])	Get a dictionary of models associated with this component.

hera_sim.foregrounds.PointSourceForeground.__call__

PointSourceForeground.**__call__**(*lsts*, *freqs*, *bl_vec*, ***kwargs*)

Compute the point source foregrounds.

Parameters

- **lsts** (*array-like of float*) – Local Sidereal Times for the simulated observation, in units of radians.
- **freqs** (*array-like of float*) – Frequency array for the simulated observation, in units of GHz.
- **bl_vec** (*array-like of float*) – Baseline vector for the simulated observation, given in East-North-Up coordinates in units of nanoseconds. Must have length 3.

Returns

vis (*np.ndarray of complex*) – Simulated observed visibilities for the specified LSTs, frequencies, and baseline. Complex-valued with shape (lsts.size, freqs.size).

Notes

The beam used here is a Gaussian with width hard-coded to HERA's width, and truncated at the horizon.

This is a *very* rough simulator, use at your own risk.

hera_sim.foregrounds.PointSourceForeground.get_aliases

classmethod PointSourceForeground.**get_aliases**() → tuple[str]

Get all the aliases by which this model can be identified.

hera_sim.foregrounds.PointSourceForeground.get_model

classmethod PointSourceForeground.**get_model**(*mdl: str*) → *SimulationComponent*

Get a model with a particular name (including aliases).

hera_sim.foregrounds.PointSourceForeground.get_models

classmethod PointSourceForeground.**get_models**(*with_aliases=False*) → dict[str, *hera_sim.components.SimulationComponent*]

Get a dictionary of models associated with this component.

Attributes

attrs_to_pull	Mapping between parameter names and Simulator attributes
is_multiplicative	Whether this systematic multiplies existing visibilities
is_randomized	Whether this systematic contains a randomized component
return_type	Whether the returned value is per-antenna, per-baseline, or the full array

hera_sim.foregrounds.PointSourceForeground.attrs_to_pull

PointSourceForeground.**attrs_to_pull: dict = {'bl_vec': None}**

Mapping between parameter names and Simulator attributes

hera_sim.foregrounds.PointSourceForeground.is_multiplicative

PointSourceForeground.**is_multiplicative: bool = False**

Whether this systematic multiplies existing visibilities

hera_sim.foregrounds.PointSourceForeground.is_randomized

PointSourceForeground.**is_randomized: bool = True**

Whether this systematic contains a randomized component

hera_sim.foregrounds.PointSourceForeground.return_type

PointSourceForeground.**return_type: str | None = 'per_baseline'**

Whether the returned value is per-antenna, per-baseline, or the full array

hera_sim.interpolators

This module provides interfaces to different interpolation classes.

Classes

Bandpass(datafile, **interp_kwargs)	Bandpass interpolation object.
Beam(datafile, **interp_kwargs)	Beam interpolation object.
FreqInterpolator(datafile[, obj_type])	Frequency interpolator.
Interpolator(datafile, **interp_kwargs)	Base interpolator class.
Reflection(datafile, **interp_kwargs)	Complex reflection coefficient interpolator.
Tsky(datafile, **interp_kwargs)	Sky temperature interpolator.

hera_sim.interpolators.Bandpass

class hera_sim.interpolators.**Bandpass**(*datafile*, ***interp_kwargs*)

Bandpass interpolation object.

Parameters

- **datafile** (*str*) – Passed to the superclass constructor.
- **interp_kwargs** (*unpacked dict, optional*) – Passed to the superclass constructor.

Methods

__call__(freqs)	Evaluate the interpolation object at the given frequencies.

hera_sim.interpolators.Bandpass.__call__

Bandpass.**__call__**(*freqs*)

Evaluate the interpolation object at the given frequencies.

hera_sim.interpolators.Beam

class hera_sim.interpolators.**Beam**(*datafile*, ***interp_kwargs*)

Beam interpolation object.

Parameters

- **datafile** (*str*) – Passed to the superclass constructor.
- **interp_kwargs** (*unpacked dict, optional*) – Passed to the superclass constructor.

Methods

__call__(freqs)	Evaluate the interpolation object at the given frequencies.

hera_sim.interpolators.Beam.__call__

Beam.**__call__**(*freqs*)

Evaluate the interpolation object at the given frequencies.

hera_sim.interpolators.FreqInterpolator

class hera_sim.interpolators.**FreqInterpolator**(*datafile*, *obj_type=None*, ***interp_kwargs*)

Frequency interpolator.

Parameters

- **datafile** (*str*) – Passed to the superclass constructor.
- **interp_kwargs** (*unpacked dict, optional*) – Extends superclass interp_kwargs parameter by checking for the key 'interpolator' in the dictionary. The 'interpolator' key should have the value 'poly1d' or 'interp1d'; these correspond to the *np.poly1d* and *scipy.interpolate.interp1d* objects, respectively. If the 'interpolator' key is not found, then it is assumed that a *np.poly1d* object is to be used for the interpolator object.

Raises

`AssertionError` – This is raised if the choice of interpolator and the required type of the ref_file do not agree (i.e. trying to make a 'poly1d' object using a .npz file as a reference). An AssertionError is also raised if the .npz for generating an 'interp1d' object does not have the correct arrays in its archive.

Methods

__call__(freqs)	Evaluate the interpolation object at the given frequencies.

hera_sim.interpolators.FreqInterpolator.__call__

`FreqInterpolator.__call__`(*freqs*)

Evaluate the interpolation object at the given frequencies.

hera_sim.interpolators.Interpolator

`class hera_sim.interpolators.Interpolator`(*datafile*, ***interp_kwargs*)

Base interpolator class.

Parameters

- **datafile** (*str*) – Path to the file to be used to generate the interpolation object. Must be either a .npy or .npz file, depending on which type of interpolation object is desired. If path is not absolute, then the file is assumed to exist in the *data* directory of `hera_sim` and is modified to reflect this assumption.
- **interp_kwargs** (*unpacked dict, optional*) – Passed to the interpolation method used to make the interpolator.

Methods

hera_sim.interpolators.Reflection

`class hera_sim.interpolators.Reflection`(*datafile*, ***interp_kwargs*)

Complex reflection coefficient interpolator.

Methods

__call__(freqs)	Evaluate the interpolation object at the given frequencies.

hera_sim.interpolators.Reflection.__call__

Reflection.**__call__**(*freqs*)

Evaluate the interpolation object at the given frequencies.

hera_sim.interpolators.Tsky

class hera_sim.interpolators.**Tsky**(*datafile*, ***interp_kwargs*)

Sky temperature interpolator.

Parameters

- **datafile** (*str*) – Passed to superclass constructor. Must be a `.npz` file with the following archives:
  - `tsky`: Array of sky temperature values in units of Kelvin; must have shape=(NPOLS, NLSTS, NFREQS).
  - `freqs`: Array of frequencies at which the tsky model is evaluated, in units of GHz; must have shape=(NFREQS,).
  - `lsts`: Array of LSTs at which the tsky model is evaulated, in units of radians; must have shape=(NLSTS,).
  - `meta`: Dictionary of metadata describing the data stored in the npz file. Currently it only needs to contain an entry 'pols', which lists the polarizations such that their order agrees with the ordering of arrays along the tsky axis-0. The user may choose to also save the units of the frequency, lst, and tsky arrays as strings in this dictionary.
- **interp_kwargs** (*unpacked dict, optional*) – Extend interp_kwargs parameter for superclass to allow for the specification of which polarization to use via the key 'pol'. If 'pol' is specified, then it must be one of the polarizations listed in the 'meta' dictionary.

Variables

- **freqs** (*np.ndarray*) – Frequency array used to construct the interpolator object. Has units of GHz and shape=(NFREQS,).
- **lsts** (*np.ndarray*) – LST array used to construct the interpolator object. Has units of radians and shape=(NLSTS,).
- **tsky** (*np.ndarray*) – Sky temperature array used to construct the interpolator object. Has units of Kelvin and shape=(NPOLS, NLSTS, NFREQS).
- **meta** (*dict*) – Dictionary containing some metadata relevant to the interpolator.
- **pol** (*str, default 'xx'*) – Polarization appropriate for the sky temperature model. Must be one of the polarizations stored in the 'meta' dictionary.

Raises

AssertionError: – Raised if any of the required npz keys are not found or if the tsky array does not have shape=(NPOLS, NLSTS, NFREQS).

Methods

__call__(lsts, freqs)	Evaluate the Tsky model at the specified lsts and freqs.

hera_sim.interpolators.Tsky.__call__

Tsky.**__call__**(*lsts*, *freqs*)

Evaluate the Tsky model at the specified lsts and freqs.

Attributes

freqs	Frequencies of the interpolation data.
lsts	Times at which the sky temperature is measured.
meta	Metadata about the measured/model sky temperature.
tsky	Measured values of the sky temperature to be interpolated.

hera_sim.interpolators.Tsky.freqs

property Tsky.**freqs: ndarray**

Frequencies of the interpolation data.

hera_sim.interpolators.Tsky.lsts

property Tsky.**lsts: ndarray**

Times at which the sky temperature is measured.

hera_sim.interpolators.Tsky.meta

property Tsky.**meta**

Metadata about the measured/model sky temperature.

hera_sim.interpolators.Tsky.tsky

property Tsky.**tsky: ndarray**

Measured values of the sky temperature to be interpolated.

hera_sim.io

Methods for input/output of data.

Functions

chunk_sim_and_save(sim_uvd, save_dir[, ...])	Chunk the simulation data to match the reference file and write to disk.
empty_uvdata([Ntimes, start_time, ...])	Create an empty UVData object with given specifications.

hera_sim.io.chunk_sim_and_save

hera_sim.io.**chunk_sim_and_save**(*sim_uvd*, *save_dir*, *ref_files=None*, *Nint_per_file=None*, *prefix=None*, *sky_cmp=None*, *state=None*, *filetype='uvh5'*, *clobber=True*)

Chunk the simulation data to match the reference file and write to disk.

Chunked files have the following naming convention: `save_dir/[{prefix}.]{jd_major}.{jd_minor}[.{sky_cmp}][.{state}].{filetype}`. The entires in brackets are optional and may be omitted.

Parameters

- **sim_uvd** (`pyuvdata.UVData`) – `pyuvdata.UVData` object containing the simulation data to chunk and write to disk.
- **save_dir** (*str or path-like object*) – Path to the directory where the chunked files will be saved.
- **ref_files** (*iterable of str*) – Iterable of filepaths to use for reference when chunking. This must be specified if `Nint_per_file` is not specified. This determines (and overrides, if also provided) `Nint_per_file` if provided.
- **Nint_per_file** (*int, optional*) – Number of integrations per chunked file. This must be specified if `ref_files` is not specified.
- **prefix** (*str, optional*) – Prefix of file basename. Default is to add no prefix.
- **sky_cmp** (*str, optional*) – String denoting which sky component has been simulated. Should be one of the following: ('foregrounds', 'eor', 'sum').
- **state** (*str, optional*) – String denoting whether the file is the true sky or corrupted.
- **filetype** (*str, optional*) – Format to use when writing files to disk. Must be a filetype supported by `pyuvdata.UVData`. Default is uvh5.
- **clobber** (*bool, optional*) – Whether to overwrite any existing files that share the new filenames. Default is to overwrite files.

hera_sim.io.empty_uvdata

hera_sim.io.**empty_uvdata**(*Ntimes=None*, *start_time=2456658.5*, *integration_time=None*, *array_layout: dict[int, collections.abc.Sequence[float]] = None*, *Nfreqs=None*, *start_freq=None*, *channel_width=None*, *n_freq=None*, *n_times=None*, *antennas=None*, *conjugation=None*, ***kwargs*)

Create an empty UVData object with given specifications.

Parameters

- **Ntimes** (*int, optional*) – NUmber of unique times in the data object.
- **start_time** (*float, optional*) – Starting time (Julian date) by default 2456658.5
- **array_layout** (*dict, optional*) – Specify an array layout. Keys should be integers specifying antenna numbers, and values should be length-3 sequences of floats specifying ENU positions.
- **Nfreqs** (*int, optional*) – Number of frequency channels in the data object
- **start_freq** (*float, optional*) – Lowest frequency channel, by default None
- **channel_width** (*float, optional*) – Channel width, by default None
- **n_freq** (*int, optional*) – Alias for `Nfreqs`
- **n_times** (*int, optional*) – Alias for `Ntimes`.
- **antennas** (*dict, optional*) – Alias for array_layout for backwards compatibility.
- ****kwargs** – Passed to `pyuvsim.simsetup.initialize_uvdata_from_keywords()`

Returns

UVData – An empty UVData object with given specifications.

hera_sim.noise

Models of thermal noise.

Functions

resample_Tsky(lsts, freqs[, Tsky_mdl, Tsky, ...])	Evaluate an array of sky temperatures.
sky_noise_jy(lsts, freqs, **kwargs)	Generate thermal noise at particular LSTs and frequencies.
white_noise(*args, **kwargs)	Generate white noise in an array.

hera_sim.noise.resample_Tsky

hera_sim.noise.**resample_Tsky**(*lsts*, *freqs*, *Tsky_mdl=None*, *Tsky=180.0*, *mfreq=0.18*, *index=-2.5*)

Evaluate an array of sky temperatures.

Parameters

- **lsts** (*array-like of float*) – LSTs at which to sample the sky tmeperature.
- **freqs** (*array_like of float*) – The frequencies at which to sample the temperature, in GHz.

- **Tsky_mdl** (*callable, optional*) – Callable function of `(lsts, freqs)`. If not given, use a power-law defined by the next three parameters.
- **Tsky** (*float, optional*) – Sky temperature at `mfreq`. Only used if `Tsky_mdl` not given.
- **mfreq** (*float, optional*) – Reference freq for sky temperature. Only used if `Tsky_mdl` not given.
- **index** (*float, optional*) – Spectral index of sky temperature model. Only used if `Tsky_mdl` not given.

Returns
ndarray – The sky temperature as a 2D array, first axis LSTs and second axis freqs.

hera_sim.noise.sky_noise_jy

hera_sim.noise.**sky_noise_jy**(*lsts: ndarray*, *freqs: ndarray*, ***kwargs*)

Generate thermal noise at particular LSTs and frequencies.

Parameters
- **lsts** (*array_like*) – LSTs at which to compute the sky noise.
- **freqs** (*array_like*) – Frequencies at which to compute the sky noise.
- ****kwargs** – Passed to *ThermalNoise*.

Returns
ndarray – 2D array of white noise in LST/freq.

hera_sim.noise.white_noise

hera_sim.noise.**white_noise**(**args*, ***kwargs*)

Generate white noise in an array.

Deprecated. Use `utils.gen_white_noise` instead.

Classes

Noise(**kwargs)	Base class for thermal noise models.
ThermalNoise([Tsky_mdl, omega_p, ...])	Generate thermal noise based on a sky model.

hera_sim.noise.Noise

class hera_sim.noise.**Noise**(***kwargs*)

Base class for thermal noise models.

This is an *abstract* class, and should not be directly instantiated. It represents a "component" – a modular part of a simulation for which several models may be defined. Models for this component may be defined by subclassing this abstract base class and implementing (at least) the *__call__()* method. Some of these are implemented within hera_sim already, but custom models may be implemented outside of hera_sim, and used on equal footing with the the internal models (as long as they subclass this abstract component).

As with all components, all parameters that define the behaviour of the model are accepted at class instantiation. The *__call__()* method actually computes the simulated effect of the component (typically, but not always, a set of visibilities or gains), by *default* using these parameters. However, these parameters can be over-ridden at call-time. Inputs such as the frequencies, times or baselines at which to compute the effect are specific to the call, and do not get passed at instantiation.

Methods

__call__(**kwargs)	Compute the component model.
get_aliases()	Get all the aliases by which this model can be identified.
get_model(mdl)	Get a model with a particular name (including aliases).
get_models([with_aliases])	Get a dictionary of models associated with this component.

hera_sim.noise.Noise.__call__

abstract Noise.**__call__**(***kwargs*)

Compute the component model.

hera_sim.noise.Noise.get_aliases

classmethod Noise.**get_aliases**() → tuple[str]

Get all the aliases by which this model can be identified.

hera_sim.noise.Noise.get_model

classmethod Noise.**get_model**(*mdl: str*) → *SimulationComponent*

Get a model with a particular name (including aliases).

hera_sim.noise.Noise.get_models

classmethod Noise.**get_models**(*with_aliases=False*) → dict[str, *hera_sim.components.SimulationComponent*]

Get a dictionary of models associated with this component.

Attributes

attrs_to_pull	Mapping between parameter names and Simulator attributes
is_multiplicative	Whether this systematic multiplies existing visibilities
is_randomized	Whether this systematic contains a randomized component
return_type	Whether the returned value is per-antenna, per-baseline, or the full array

hera_sim.noise.Noise.attrs_to_pull

Noise.**attrs_to_pull**: dict = {}

Mapping between parameter names and Simulator attributes

hera_sim.noise.Noise.is_multiplicative

Noise.**is_multiplicative**: bool = False

Whether this systematic multiplies existing visibilities

hera_sim.noise.Noise.is_randomized

Noise.**is_randomized**: bool = False

Whether this systematic contains a randomized component

hera_sim.noise.Noise.return_type

Noise.**return_type**: str | None = None

Whether the returned value is per-antenna, per-baseline, or the full array

hera_sim.noise.ThermalNoise

class hera_sim.noise.**ThermalNoise**(*Tsky_mdl=None*, *omega_p=None*, *integration_time=None*, *channel_width=None*, *Trx=0*, *autovis=None*, *antpair=None*, *rng=None*)

Generate thermal noise based on a sky model.

Parameters

- **Tsky_mdl** (*callable, optional*) – A function of `(lsts, freq)` that returns the integrated sky temperature at that time/frequency. If not provided, assumes a power-law temperature with 180 K at 180 MHz and spectral index of -2.5.
- **omega_p** (*array_like or callable, optional*) – If callable, a function of frequency giving the integrated beam area. If an array, same length as given frequencies.
- **integration_time** (*float, optional*) – Integration time in seconds. By default, use the average difference between given LSTs.

- **channel_width** (*float, optional*) – Channel width in Hz, by default the mean difference between frequencies.
- **Trx** (*float, optional*) – Receiver temperature in K
- **autovis** (*float, optional*) – Autocorrelation visibility amplitude. Used if provided instead of `Tsky_mdl`.
- **antpair** (*tuple of int, optional*) – Antenna numbers for the baseline that noise is being simulated for. This is just used to determine whether to simulate noise via the radiometer equation or to just add a bias from the receiver temperature.
- **rng** (*np.random.Generator, optional*) – Random number generator.

Notes

Considering the SNR in autocorrelations is typically very high, we only add a receiver temperature bias to the autocorrelations.

Methods

__call__(lsts, freqs, **kwargs)	Compute the thermal noise.
get_aliases()	Get all the aliases by which this model can be identified.
get_model(mdl)	Get a model with a particular name (including aliases).
get_models([with_aliases])	Get a dictionary of models associated with this component.
resample_Tsky(lsts, freqs[, Tsky_mdl, Tsky, ...])	Evaluate an array of sky temperatures.

hera_sim.noise.ThermalNoise.__call__

`ThermalNoise.__call__`(*lsts: ndarray*, *freqs: ndarray*, ***kwargs*)

Compute the thermal noise.

Parameters

- **lsts** – Local siderial times at which to compute the noise.
- **freqs** – Frequencies at which to compute the noise.

Returns

array – A 2D array shaped `(lsts, freqs)` with the thermal noise. If the provided `antpair` is for an autocorrelation, then only a receiver temperature bias is returned.

Raises

`NotImplementedError` – This method does not yet have support for handling the case when the provided LST array has a phase wrap and a sky temperature interpolation object is intended to be used to simulate the noise.

hera_sim.noise.ThermalNoise.get_aliases

classmethod ThermalNoise.**get_aliases**() → tuple[str]

Get all the aliases by which this model can be identified.

hera_sim.noise.ThermalNoise.get_model

classmethod ThermalNoise.**get_model**(*mdl: str*) → *SimulationComponent*

Get a model with a particular name (including aliases).

hera_sim.noise.ThermalNoise.get_models

classmethod ThermalNoise.**get_models**(*with_aliases=False*) → dict[str, *hera_sim.components.SimulationComponent*]

Get a dictionary of models associated with this component.

hera_sim.noise.ThermalNoise.resample_Tsky

static ThermalNoise.**resample_Tsky**(*lsts*, *freqs*, *Tsky_mdl=None*, *Tsky=180.0*, *mfreq=0.18*, *index=-2.5*)

Evaluate an array of sky temperatures.

Parameters

- **lsts** (*array-like of float*) – LSTs at which to sample the sky tmeperature.
- **freqs** (*array_like of float*) – The frequencies at which to sample the temperature, in GHz.
- **Tsky_mdl** (*callable, optional*) – Callable function of `(lsts, freqs)`. If not given, use a power-law defined by the next three parameters.
- **Tsky** (*float, optional*) – Sky temperature at `mfreq`. Only used if `Tsky_mdl` not given.
- **mfreq** (*float, optional*) – Reference freq for sky temperature. Only used if `Tsky_mdl` not given.
- **index** (*float, optional*) – Spectral index of sky temperature model. Only used if `Tsky_mdl` not given.

Returns

ndarray – The sky temperature as a 2D array, first axis LSTs and second axis freqs.

Attributes

attrs_to_pull	Mapping between parameter names and Simulator attributes
is_multiplicative	Whether this systematic multiplies existing visibilities
is_randomized	Whether this systematic contains a randomized component
return_type	Whether the returned value is per-antenna, per-baseline, or the full array

hera_sim.noise.ThermalNoise.attrs_to_pull

ThermalNoise.**attrs_to_pull**: dict = {'antpair': None, 'autovis': None}

Mapping between parameter names and Simulator attributes

hera_sim.noise.ThermalNoise.is_multiplicative

ThermalNoise.**is_multiplicative**: bool = False

Whether this systematic multiplies existing visibilities

hera_sim.noise.ThermalNoise.is_randomized

ThermalNoise.**is_randomized**: bool = True

Whether this systematic contains a randomized component

hera_sim.noise.ThermalNoise.return_type

ThermalNoise.**return_type**: str | None = 'per_baseline'

Whether the returned value is per-antenna, per-baseline, or the full array

hera_sim.rfi

Models of radio frequency interference.

Classes

DTV([dtv_band, dtv_channel_width, ...])	Generate RFI arising from digitial TV channels.
Impulse([impulse_chance, impulse_strength, rng])	Generate RFI impulses (short time, broad frequency).
RFI(**kwargs)	Base class for RFI models.
RfiStation(f0[, duty_cycle, strength, std, ...])	Generate RFI based on a particular "station".
Scatter([scatter_chance, scatter_strength, ...])	Generate random RFI scattered around the waterfall.
Stations([stations, rng])	A collection of RFI stations.

hera_sim.rfi.DTV

class hera_sim.rfi.**DTV**(*dtv_band=(0.174, 0.214)*, *dtv_channel_width=0.008*, *dtv_chance=0.0001*, *dtv_strength=10.0*, *dtv_std=10.0*, *rng=None*)

Generate RFI arising from digitial TV channels.

Digitial TV is assumed to be reasonably broad-band and scattered in time.

Parameters

- **dtv_band** (*tuple, optional*) – Lower edges of each of the DTV bands.
- **dtv_channel_width** (*float, optional*) – Channel width in GHz.
- **dtv_chance** (*float, optional*) – Chance that any particular time will have DTV.

- **dtv_strength** (*float, optional*) – Mean strength of RFI.
- **dtv_std** (*float, optional*) – Standard deviation of RFI strength.
- **rng** (*np.random.Generator, optional*) – Random number generator.

Methods

`__call__`(lsts, freqs, **kwargs)	Generate the RFI.
`get_aliases`()	Get all the aliases by which this model can be identified.
`get_model`(mdl)	Get a model with a particular name (including aliases).
`get_models`([with_aliases])	Get a dictionary of models associated with this component.

hera_sim.rfi.DTV.__call__

DTV.**__call__**(*lsts*, *freqs*, ***kwargs*)

Generate the RFI.

Parameters

- **lsts** (*array-like*) – LSTs at which to generate the RFI.
- **freqs** (*array-like of float*) – Frequencies in GHz.

Returns

array-like of float – 2D array of RFI magnitudes as a function of LST and frequency.

hera_sim.rfi.DTV.get_aliases

classmethod DTV.**get_aliases**() → tuple[str]

Get all the aliases by which this model can be identified.

hera_sim.rfi.DTV.get_model

classmethod DTV.**get_model**(*mdl: str*) → *SimulationComponent*

Get a model with a particular name (including aliases).

hera_sim.rfi.DTV.get_models

classmethod DTV.**get_models**(*with_aliases=False*) → dict[str, *hera_sim.components.SimulationComponent*]

Get a dictionary of models associated with this component.

Attributes

attrs_to_pull	Mapping between parameter names and Simulator attributes
is_multiplicative	Whether this systematic multiplies existing visibilities
is_randomized	Whether this systematic contains a randomized component
return_type	Whether the returned value is per-antenna, per-baseline, or the full array

hera_sim.rfi.DTV.attrs_to_pull

DTV.**attrs_to_pull**: dict = {}

Mapping between parameter names and Simulator attributes

hera_sim.rfi.DTV.is_multiplicative

DTV.**is_multiplicative**: bool = False

Whether this systematic multiplies existing visibilities

hera_sim.rfi.DTV.is_randomized

DTV.**is_randomized**: bool = True

Whether this systematic contains a randomized component

hera_sim.rfi.DTV.return_type

DTV.**return_type**: str | None = 'per_baseline'

Whether the returned value is per-antenna, per-baseline, or the full array

hera_sim.rfi.Impulse

class hera_sim.rfi.**Impulse**(*impulse_chance=0.001*, *impulse_strength=20.0*, *rng=None*)

Generate RFI impulses (short time, broad frequency).

Parameters

- **impulse_chance** (*float, optional*) – The probability in any given LST that an impulse RFI will occur.
- **impulse_strength** (*float, optional*) – Strength of the impulse. This will not be randomized, though a phase offset as a function of frequency will be applied, and will be random for each impulse.
- **rng** (*np.random.Generator, optional*) – Random number generator.

Methods

__call__(lsts, freqs, **kwargs)	Generate the RFI.
get_aliases()	Get all the aliases by which this model can be identified.
get_model(mdl)	Get a model with a particular name (including aliases).
get_models([with_aliases])	Get a dictionary of models associated with this component.

hera_sim.rfi.Impulse.__call__

Impulse.**__call__**(*lsts*, *freqs*, ***kwargs*)

Generate the RFI.

Parameters

- **lsts** (*array-like*) – LSTs at which to generate the RFI.
- **freqs** (*array-like of float*) – Frequencies in arbitrary units.

Returns

array-like of float – 2D array of RFI magnitudes as a function of LST and frequency.

hera_sim.rfi.Impulse.get_aliases

classmethod Impulse.**get_aliases**() → tuple[str]

Get all the aliases by which this model can be identified.

hera_sim.rfi.Impulse.get_model

classmethod Impulse.**get_model**(*mdl: str*) → *SimulationComponent*

Get a model with a particular name (including aliases).

hera_sim.rfi.Impulse.get_models

classmethod Impulse.**get_models**(*with_aliases=False*) → dict[str, *hera_sim.components.SimulationComponent*]

Get a dictionary of models associated with this component.

Attributes

attrs_to_pull	Mapping between parameter names and Simulator attributes
is_multiplicative	Whether this systematic multiplies existing visibilities
is_randomized	Whether this systematic contains a randomized component
return_type	Whether the returned value is per-antenna, per-baseline, or the full array

hera_sim.rfi.Impulse.attrs_to_pull

Impulse.**attrs_to_pull**: dict = {}

Mapping between parameter names and Simulator attributes

hera_sim.rfi.Impulse.is_multiplicative

Impulse.**is_multiplicative**: bool = False

Whether this systematic multiplies existing visibilities

hera_sim.rfi.Impulse.is_randomized

Impulse.**is_randomized**: bool = True

Whether this systematic contains a randomized component

hera_sim.rfi.Impulse.return_type

Impulse.**return_type**: str | None = 'per_baseline'

Whether the returned value is per-antenna, per-baseline, or the full array

hera_sim.rfi.RFI

class hera_sim.rfi.**RFI**(***kwargs*)

Base class for RFI models.

This is an *abstract* class, and should not be directly instantiated. It represents a "component" – a modular part of a simulation for which several models may be defined. Models for this component may be defined by subclassing this abstract base class and implementing (at least) the *__call__()* method. Some of these are implemented within hera_sim already, but custom models may be implemented outside of hera_sim, and used on equal footing with the the internal models (as long as they subclass this abstract component).

As with all components, all parameters that define the behaviour of the model are accepted at class instantiation. The *__call__()* method actually computes the simulated effect of the component (typically, but not always, a set of visibilities or gains), by *default* using these parameters. However, these parameters can be over-ridden at call-time. Inputs such as the frequencies, times or baselines at which to compute the effect are specific to the call, and do not get passed at instantiation.

Methods

__call__(**kwargs)	Compute the component model.
get_aliases()	Get all the aliases by which this model can be identified.
get_model(mdl)	Get a model with a particular name (including aliases).
get_models([with_aliases])	Get a dictionary of models associated with this component.

hera_sim.rfi.RFI.__call__

abstract RFI.**__call__**(***kwargs*)

Compute the component model.

hera_sim.rfi.RFI.get_aliases

classmethod RFI.**get_aliases**() → tuple[str]

Get all the aliases by which this model can be identified.

hera_sim.rfi.RFI.get_model

classmethod RFI.**get_model**(*mdl: str*) → *SimulationComponent*

Get a model with a particular name (including aliases).

hera_sim.rfi.RFI.get_models

classmethod RFI.**get_models**(*with_aliases=False*) → dict[str, *hera_sim.components.SimulationComponent*]

Get a dictionary of models associated with this component.

Attributes

attrs_to_pull	Mapping between parameter names and Simulator attributes
is_multiplicative	Whether this systematic multiplies existing visibilities
is_randomized	Whether this systematic contains a randomized component
return_type	Whether the returned value is per-antenna, per-baseline, or the full array

hera_sim.rfi.RFI.attrs_to_pull

RFI.**attrs_to_pull**: dict = {}

Mapping between parameter names and Simulator attributes

hera_sim.rfi.RFI.is_multiplicative

RFI.**is_multiplicative**: bool = False

Whether this systematic multiplies existing visibilities

hera_sim.rfi.RFI.is_randomized

RFI.**is_randomized**: bool = False

Whether this systematic contains a randomized component

hera_sim.rfi.RFI.return_type

RFI.**return_type**: str | None = None

Whether the returned value is per-antenna, per-baseline, or the full array

hera_sim.rfi.RfiStation

class hera_sim.rfi.**RfiStation**(*f0: float*, *duty_cycle: float = 1.0*, *strength: float = 100.0*, *std: float = 10.0*, *timescale: float = 100.0*, *rng: Generator | None = None*)

Generate RFI based on a particular "station".

Parameters

- **f0** (*float*) – Frequency that the station transmits (any units are fine).
- **duty_cycle** (*float, optional*) – With `timescale`, controls how long the station is seen as "on". In particular, `duty_cycle` specifies which parts of the station's cycle are considered "on". Can be considered roughly a percentage of on time.
- **strength** (*float, optional*) – Mean magnitude of the transmission.
- **std** (*float, optional*) – Standard deviation of the random RFI magnitude.
- **timescale** (*float, optional*) – Controls the length of a transmision "cycle". Low points in the sin-wave cycle are considered "off" and high points are considered "on" (just how high is controlled by `duty_cycle`). This is the wavelength (in seconds) of that cycle.
- **rng** (*np.random.Generator, optional*) – Random number generator.

Notes

This creates RFI with random magnitude in each time bin based on a normal distribution, with custom strength and variability. RFI is assumed to exist in one frequency channel, with some spillage into an adjacent channel, proportional to the distance to that channel from the station's frequency. It is not assumed to be always on, but turns on for some amount of time at regular intervals.

Methods

__call__(lsts, freqs)	Compute the RFI for this station.

hera_sim.rfi.RfiStation.__call__

RfiStation.**__call__**(*lsts*, *freqs*)

Compute the RFI for this station.

Parameters

- **lsts** (*array-like*) – LSTs at which to generate the RFI.
- **freqs** (*array-like of float*) – Frequencies in units of `f0`.

Returns

array-like – 2D array of RFI magnitudes as a function of LST and frequency.

hera_sim.rfi.Scatter

class hera_sim.rfi.**Scatter**(*scatter_chance=0.0001*, *scatter_strength=10.0*, *scatter_std=10.0*, *rng=None*)

Generate random RFI scattered around the waterfall.

Parameters

- **scatter_chance** (*float, optional*) – Probability that any LST/freq bin will be occupied by RFI.
- **scatter_strength** (*float, optional*) – Mean strength of RFI in any bin (each bin will receive its own random strength).
- **scatter_std** (*float, optional*) – Standard deviation of the RFI strength.
- **rng** (*np.random.Generator, optional*) – Random number generator.

Methods

__call__(lsts, freqs, **kwargs)	Generate the RFI.
get_aliases()	Get all the aliases by which this model can be identified.
get_model(mdl)	Get a model with a particular name (including aliases).
get_models([with_aliases])	Get a dictionary of models associated with this component.

hera_sim.rfi.Scatter.__call__

Scatter.**__call__**(*lsts*, *freqs*, ***kwargs*)

Generate the RFI.

Parameters

- **lsts** (*array-like*) – LSTs at which to generate the RFI.
- **freqs** (*array-like of float*) – Frequencies in arbitrary units.

Returns

array-like of float – 2D array of RFI magnitudes as a function of LST and frequency.

hera_sim.rfi.Scatter.get_aliases

classmethod Scatter.**get_aliases**() → tuple[str]

Get all the aliases by which this model can be identified.

hera_sim.rfi.Scatter.get_model

classmethod Scatter.**get_model**(*mdl: str*) → *SimulationComponent*

Get a model with a particular name (including aliases).

hera_sim.rfi.Scatter.get_models

classmethod Scatter.**get_models**(*with_aliases=False*) → dict[str, *hera_sim.components.SimulationComponent*]

Get a dictionary of models associated with this component.

Attributes

attrs_to_pull	Mapping between parameter names and Simulator attributes
is_multiplicative	Whether this systematic multiplies existing visibilities
is_randomized	Whether this systematic contains a randomized component
return_type	Whether the returned value is per-antenna, per-baseline, or the full array

hera_sim.rfi.Scatter.attrs_to_pull

Scatter.**attrs_to_pull**: dict = {}

Mapping between parameter names and Simulator attributes

hera_sim.rfi.Scatter.is_multiplicative

Scatter.**is_multiplicative**: bool = False

Whether this systematic multiplies existing visibilities

hera_sim.rfi.Scatter.is_randomized

Scatter.**is_randomized**: bool = True

Whether this systematic contains a randomized component

hera_sim.rfi.Scatter.return_type

Scatter.**return_type**: str | None = 'per_baseline'

Whether the returned value is per-antenna, per-baseline, or the full array

hera_sim.rfi.Stations

class hera_sim.rfi.**Stations**(*stations=None*, *rng=None*)

A collection of RFI stations.

Generates RFI from all given stations.

Parameters

- **stations** (list of *RfiStation*) – The list of stations that produce RFI.
- **rng** (*np.random.Generator, optional*) – Random number generator.

Methods

__call__(lsts, freqs, **kwargs)	Generate the RFI from all stations.
get_aliases()	Get all the aliases by which this model can be identified.
get_model(mdl)	Get a model with a particular name (including aliases).
get_models([with_aliases])	Get a dictionary of models associated with this component.

hera_sim.rfi.Stations.__call__

Stations.__call__(*lsts*, *freqs*, ***kwargs*)

Generate the RFI from all stations.

Parameters

- **lsts** (*array-like*) – LSTs at which to generate the RFI.
- **freqs** (*array-like of float*) – Frequencies in units of `f0` for each station.

Returns

array-like of float – 2D array of RFI magnitudes as a function of LST and frequency.

Raises

`TypeError` – If input stations are not of the correct type.

hera_sim.rfi.Stations.get_aliases

classmethod Stations.**get_aliases**() → tuple[str]

Get all the aliases by which this model can be identified.

hera_sim.rfi.Stations.get_model

classmethod Stations.**get_model**(*mdl: str*) → *SimulationComponent*

Get a model with a particular name (including aliases).

hera_sim.rfi.Stations.get_models

classmethod Stations.**get_models**(*with_aliases=False*) → dict[str, *hera_sim.components.SimulationComponent*]

Get a dictionary of models associated with this component.

Attributes

attrs_to_pull	Mapping between parameter names and Simulator attributes
is_multiplicative	Whether this systematic multiplies existing visibilities
is_randomized	Whether this systematic contains a randomized component
return_type	Whether the returned value is per-antenna, per-baseline, or the full array

hera_sim.rfi.Stations.attrs_to_pull

Stations.**attrs_to_pull**: dict = {}

Mapping between parameter names and Simulator attributes

hera_sim.rfi.Stations.is_multiplicative

Stations.**is_multiplicative**: bool = False

Whether this systematic multiplies existing visibilities

hera_sim.rfi.Stations.is_randomized

Stations.**is_randomized**: bool = True

Whether this systematic contains a randomized component

hera_sim.rfi.Stations.return_type

Stations.**return_type**: str | None = 'per_baseline'

Whether the returned value is per-antenna, per-baseline, or the full array

hera_sim.sigchain

Models of signal-chain systematics.

This module defines several models of systematics that arise in the signal chain, for example bandpass gains, reflections and cross-talk.

Functions

apply_gains(vis, gains, bl)	Apply antenna-based gains to a visibility.
gen_bandpass(freqs, ants[, gain_spread, ...])	
gen_delay_phs(freqs, ants[, dly_rng, rng])	
gen_reflection_coefficient(freqs, amp, dly, phs)	Randomly generate reflection coefficients.
vary_gains_in_time(gains, times[, freqs, ...])	Vary gain amplitudes, phases, or delays in time.

hera_sim.sigchain.apply_gains

hera_sim.sigchain.**apply_gains**(*vis: float | ndarray*, *gains: dict[int, float | numpy.ndarray]*, *bl: tuple[int, int]*) → ndarray

Apply antenna-based gains to a visibility.

Parameters

- **vis** – The visibilities of the given baseline as a function of frequency.

- **gains** – Dictionary where keys are antenna numbers and values are arrays of gains as a function of frequency.
- **bl** – 2-tuple of integers specifying the antenna numbers in the particular baseline.

Returns

vis – The visibilities with gains applied.

hera_sim.sigchain.gen_bandpass

hera_sim.sigchain.**gen_bandpass**(*freqs*, *ants*, *gain_spread=0.1*, *bp_poly=None*, *rng=None*)

hera_sim.sigchain.gen_delay_phs

hera_sim.sigchain.**gen_delay_phs**(*freqs*, *ants*, *dly_rng=(-20, 20)*, *rng=None*)

hera_sim.sigchain.gen_reflection_coefficient

hera_sim.sigchain.**gen_reflection_coefficient**(*freqs*, *amp*, *dly*, *phs*, *conj=False*)

Randomly generate reflection coefficients.

Parameters

- **freqs** (*array_like of float*) – Frequencies, units are arbitrary but must be the inverse of `dly`.
- **amp** (*array_like of float*) – Either a scalar amplitude, or 1D with size Nfreqs, or 2D with shape (Ntimes, Nfreqs).
- **dly** (*[type]*) – Either a scalar delay, or 1D with size Nfreqs, or 2D with shape (Ntimes, Nfreqs). Units are inverse of `freqs`.
- **phs** (*[type]*) – Either a scalar phase, or 1D with size Nfreqs, or 2D with shape (Ntimes, Nfreqs). Units radians.
- **conj** (*bool, optional*) – Whether to conjugate the gain.

Returns

array_like – The reflection gains as a 2D array of (Ntimes, Nfreqs).

hera_sim.sigchain.vary_gains_in_time

hera_sim.sigchain.**vary_gains_in_time**(*gains*, *times*, *freqs=None*, *delays=None*, *parameter='amp'*, *variation_ref_time=None*, *variation_timescale=None*, *variation_amp=0.05*, *variation_mode='linear'*, *rng=None*)

Vary gain amplitudes, phases, or delays in time.

Notes

If the gains initially have the form

$$g(\nu) = g_0(\nu)\exp(i2\pi\nu\tau + i\phi)$$

then the output gains have the form

$$g(\nu, t) = g_0(\nu, t)\exp\big(i2\pi\nu\tau(t) + i\phi(t)\big).$$

Parameters

- **gains** (*dict*) – Dictionary mapping antenna numbers to gain spectra/waterfalls.
- **times** (*array-like of float*) – Times at which to simulate time variation. Should be the same length as the data to which the gains will be applied. Should also be in the same units as `variation_ref_time` and `variation_timescale`.
- **freqs** (*array-like of float, optional*) – Frequencies at which the gains are evaluated, in GHz. Only needs to be specified for adding time variation to the delays.
- **delays** (*dict, optional*) – Dictionary mapping antenna numbers to gain delays, in ns.
- **parameter** (*str, optional*) – Which gain parameter to vary; must be one of ("amp", "phs", "dly").
- **variation_ref_time** (*float or array-like of float, optional*) – Reference time(s) used for generating time variation. For linear and sinusoidal variation, this is the time where the gains are equal to their original, time-independent values. Should be in the same units as the `times` array. Default is to use the center of the `times` provided.
- **variation_timescale** (*float or array-like of float, optional*) – Timescale(s) for one cycle of the variation(s), in the same units as the provided `times`. Default is to use the duration of the entire `times` array.
- **variation_amp** (*float or array-like of float, optional*) – Amplitude(s) of the variation(s) introduced. This is *not* the peak-to-peak amplitude! This also does not have exactly the same interpretation for each type of variation mode. For amplitude and delay variation, this represents the amplitude of modulations–so it can be interpreted as a fractional variation. For phase variation, this represents an absolute, time-dependent phase offset to introduce to the gains; however, it is still *not* a peak-to-peak amplitude.
- **variation_mode** (*str or array-like of str, optional*) – Which type(s) of variation to simulate. Supported modes are "linear", "sinusoidal", and "noiselike". Default is "linear". Note that the "linear" mode produces a triangle wave variation with period twice the corresponding timescale; this ensures that the gains vary linearly over the entire set of provided times if the default variation timescale is used.
- **rng** (*np.random.Generator, optional*) – Random number generator.

Returns

time_varied_gains (*dict*) – Dictionary mapping antenna numbers to gain waterfalls.

Classes

Bandpass([gain_spread, dly_rng, bp_poly, ...])	Generate bandpass gains.
CrossCouplingCrosstalk([amp, dly, phs, ...])	Generate cross-coupling xtalk.
CrossCouplingSpectrum([n_copies, amp_range, ...])	Generate a cross-coupling spectrum.
Crosstalk(**kwargs)	Base class for cross-talk models.
Gain(**kwargs)	Base class for systematic gains.
MutualCoupling([uvbeam, reflection, ...])	Simulate mutual coupling according to Josaitis+ 2022.
OverAirCrossCoupling([emitter_pos, ...])	Crosstalk model based on the mechanism described in HERA Memo 104.
ReflectionSpectrum([n_copies, amp_range, ...])	Generate many reflections between a range of delays.
Reflections([amp, dly, phs, conj, ...])	Produce multiplicative reflection gains.
WhiteNoiseCrosstalk([amplitude, rng])	Generate cross-talk that is simply white noise.

hera_sim.sigchain.Bandpass

class hera_sim.sigchain.**Bandpass**(*gain_spread: float | np.ndarray = 0.1*, *dly_rng: tuple = (-20, 20)*, *bp_poly: str | callable | np.ndarray | None = None*, *taper: str | callable | np.ndarray | None = None*, *taper_kwds: dict | None = None*, *rng: np.random.Generator | None = None*)

Generate bandpass gains.

Parameters

- **gain_spread** – Standard deviation of random gains. Default is about 10% variation across antennas.
- **dly_rng** – Lower and upper range of delays which are uniformly sampled, in nanoseconds. Default is -20 ns to +20 ns.
- **bp_poly** – Either an array of polynomial coefficients, a callable object that provides the bandpass amplitude as a function of frequency (in GHz), or a string providing a path to a file that can be read into an interpolation object. By default, the HERA Phase One bandpass is used.
- **taper** – Taper to apply to the simulated gains. Default is to not apply a taper.
- **taper_kwds** – Keyword arguments used in generating the taper.
- **rng** – Random number generator.

Methods

__call__(freqs, ants, **kwargs)	Generate the bandpass.
get_aliases()	Get all the aliases by which this model can be identified.
get_model(mdl)	Get a model with a particular name (including aliases).
get_models([with_aliases])	Get a dictionary of models associated with this component.

hera_sim.sigchain.Bandpass.__call__

Bandpass.**__call__**(*freqs*, *ants*, ***kwargs*)

Generate the bandpass.

Parameters

- **freqs** (*array_like of float*) – Frequencies in GHz.
- **ants** (*array_like of int*) – Antenna numbers for which to produce gains.

Returns

dict – Keys are antenna numbers and values are arrays of bandpass gains as a function of frequency.

hera_sim.sigchain.Bandpass.get_aliases

classmethod Bandpass.**get_aliases**() → tuple[str]

Get all the aliases by which this model can be identified.

hera_sim.sigchain.Bandpass.get_model

classmethod Bandpass.**get_model**(*mdl: str*) → *SimulationComponent*

Get a model with a particular name (including aliases).

hera_sim.sigchain.Bandpass.get_models

classmethod Bandpass.**get_models**(*with_aliases=False*) → dict[str, *hera_sim.components.SimulationComponent*]

Get a dictionary of models associated with this component.

Attributes

attrs_to_pull	Mapping between parameter names and Simulator attributes
is_multiplicative	Whether this systematic multiplies existing visibilities
is_randomized	Whether this systematic contains a randomized component
return_type	Whether the returned value is per-antenna, per-baseline, or the full array

hera_sim.sigchain.Bandpass.attrs_to_pull

Bandpass.**attrs_to_pull**: dict = {'ants': 'antpos'}

Mapping between parameter names and Simulator attributes

hera_sim.sigchain.Bandpass.is_multiplicative

Bandpass.**is_multiplicative**: bool = True

Whether this systematic multiplies existing visibilities

hera_sim.sigchain.Bandpass.is_randomized

Bandpass.**is_randomized**: bool = True

Whether this systematic contains a randomized component

hera_sim.sigchain.Bandpass.return_type

Bandpass.**return_type**: str | None = 'per_antenna'

Whether the returned value is per-antenna, per-baseline, or the full array

hera_sim.sigchain.CrossCouplingCrosstalk

class hera_sim.sigchain.**CrossCouplingCrosstalk**(*amp=None*, *dly=None*, *phs=None*, *conj=False*, *amp_jitter=0*, *dly_jitter=0*, *rng=None*)

Generate cross-coupling xtalk.

Parameters

- **amp** (*float, optional*) – Mean Amplitude of the reflection gains.
- **dly** (*float, optional*) – Mean delay of the reflection gains.
- **phs** (*float, optional*) – Phase of the reflection gains.
- **conj** (*bool, optional*) – Whether to conjugate the gain.
- **amp_jitter** (*float, optional*) – Final amplitudes are multiplied by a normal variable with mean one, and with standard deviation of `amp_jitter`.
- **dly_jitter** (*float, optional*) – Final delays are offset by a normal variable with mean zero and standard deviation `dly_jitter`.
- **rng** (*np.random.Generator, optional*) – Random number generator.

Methods

__call__(freqs, autovis, **kwargs)	Copute the cross-correlations.
gen_reflection_coefficient(freqs, amp, dly, phs)	Randomly generate reflection coefficients.
get_aliases()	Get all the aliases by which this model can be identified.
get_model(mdl)	Get a model with a particular name (including aliases).
get_models([with_aliases])	Get a dictionary of models associated with this component.

hera_sim.sigchain.CrossCouplingCrosstalk.__call__

CrossCouplingCrosstalk.**__call__**(*freqs*, *autovis*, ***kwargs*)

Copute the cross-correlations.

Parameters

- **freqs** (*array_like of float*) – Frequencies in units inverse to `dly`.
- **autovis** (*array_like of float*) – The autocorrelations as a function of frequency.

Returns

array – The cross-coupling contribution to the visibility, same shape as `freqs`.

hera_sim.sigchain.CrossCouplingCrosstalk.gen_reflection_coefficient

static CrossCouplingCrosstalk.**gen_reflection_coefficient**(*freqs*, *amp*, *dly*, *phs*, *conj=False*)

Randomly generate reflection coefficients.

Parameters

- **freqs** (*array_like of float*) – Frequencies, units are arbitrary but must be the inverse of `dly`.
- **amp** (*array_like of float*) – Either a scalar amplitude, or 1D with size Nfreqs, or 2D with shape (Ntimes, Nfreqs).
- **dly** (*[type]*) – Either a scalar delay, or 1D with size Nfreqs, or 2D with shape (Ntimes, Nfreqs). Units are inverse of `freqs`.
- **phs** (*[type]*) – Either a scalar phase, or 1D with size Nfreqs, or 2D with shape (Ntimes, Nfreqs). Units radians.
- **conj** (*bool, optional*) – Whether to conjugate the gain.

Returns

array_like – The reflection gains as a 2D array of (Ntimes, Nfreqs).

hera_sim.sigchain.CrossCouplingCrosstalk.get_aliases

classmethod CrossCouplingCrosstalk.**get_aliases**() → tuple[str]

Get all the aliases by which this model can be identified.

hera_sim.sigchain.CrossCouplingCrosstalk.get_model

classmethod CrossCouplingCrosstalk.**get_model**(*mdl: str*) → *SimulationComponent*

Get a model with a particular name (including aliases).

hera_sim.sigchain.CrossCouplingCrosstalk.get_models

classmethod CrossCouplingCrosstalk.**get_models**(*with_aliases=False*) → dict[str, *hera_sim.components.SimulationComponent*]

Get a dictionary of models associated with this component.

Attributes

attrs_to_pull	Mapping between parameter names and Simulator attributes
is_multiplicative	Whether this systematic multiplies existing visibilities
is_randomized	Whether this systematic contains a randomized component
return_type	Whether the returned value is per-antenna, per-baseline, or the full array

hera_sim.sigchain.CrossCouplingCrosstalk.attrs_to_pull

CrossCouplingCrosstalk.**attrs_to_pull: dict = {'autovis': None}**

Mapping between parameter names and Simulator attributes

hera_sim.sigchain.CrossCouplingCrosstalk.is_multiplicative

CrossCouplingCrosstalk.**is_multiplicative: bool = False**

Whether this systematic multiplies existing visibilities

hera_sim.sigchain.CrossCouplingCrosstalk.is_randomized

CrossCouplingCrosstalk.**is_randomized: bool = True**

Whether this systematic contains a randomized component

hera_sim.sigchain.CrossCouplingCrosstalk.return_type

CrossCouplingCrosstalk.**return_type: str | None = 'per_baseline'**

Whether the returned value is per-antenna, per-baseline, or the full array

hera_sim.sigchain.CrossCouplingSpectrum

class hera_sim.sigchain.**CrossCouplingSpectrum**(*n_copies=10*, *amp_range=(-4, -6)*, *dly_range=(1000, 1200)*, *phs_range=(-3.141592653589793, 3.141592653589793)*, *amp_jitter=0*, *dly_jitter=0*, *amp_logbase=10*, *symmetrize=True*, *rng=None*)

Generate a cross-coupling spectrum.

This generates multiple copies of *CrossCouplingCrosstalk* into the visibilities.

Parameters

- **n_copies** (*int, optional*) – Number of random cross-talk models to add.
- **amp_range** (*tuple, optional*) – Two-tuple of floats specifying the range of amplitudes to be sampled regularly in log-space.
- **dly_range** (*tuple, optional*) – Two-tuple of floats specifying the range of delays to be sampled at regular intervals.
- **phs_range** (*tuple, optional*) – Range of uniformly random phases.
- **amp_jitter** (*int, optional*) – Standard deviation of random jitter to be applied to the regular amplitudes.
- **dly_jitter** (*int, optional*) – Standard deviation of the random jitter to be applied to the regular delays.
- **amp_logbase** (*float, optional*) – Base of the logarithm to use for generating amplitudes.
- **symmetrize** (*bool, optional*) – Whether to also produce statistically equivalent cross-talk at negative delays. Note that while the statistics are equivalent, both amplitudes and delays will be different random realizations.
- **rng** (*np.random.Generator, optional*) – Random number generator.

Notes

The generated amplitudes will be in the range `amp_logbase ** amp_range[0]` to `amp_logbase ** amp_range[1]`.

Methods

__call__(freqs, autovis, **kwargs)	Compute the cross-correlations.
get_aliases()	Get all the aliases by which this model can be identified.
get_model(mdl)	Get a model with a particular name (including aliases).
get_models([with_aliases])	Get a dictionary of models associated with this component.

hera_sim.sigchain.CrossCouplingSpectrum.__call__

CrossCouplingSpectrum.**__call__**(*freqs*, *autovis*, ***kwargs*)

Compute the cross-correlations.

Parameters

- **freqs** (*array_like of float*) – Frequencies in units inverse to `dly`.
- **autovis** (*array_like of float*) – The autocorrelations as a function of frequency.

Returns

array – The cross-coupling contribution to the visibility, same shape as `freqs`.

hera_sim.sigchain.CrossCouplingSpectrum.get_aliases

classmethod CrossCouplingSpectrum.**get_aliases**() → tuple[str]

Get all the aliases by which this model can be identified.

hera_sim.sigchain.CrossCouplingSpectrum.get_model

classmethod CrossCouplingSpectrum.**get_model**(*mdl: str*) → *SimulationComponent*

Get a model with a particular name (including aliases).

hera_sim.sigchain.CrossCouplingSpectrum.get_models

classmethod CrossCouplingSpectrum.**get_models**(*with_aliases=False*) → dict[str, *hera_sim.components.SimulationComponent*]

Get a dictionary of models associated with this component.

Attributes

attrs_to_pull	Mapping between parameter names and Simulator attributes
is_multiplicative	Whether this systematic multiplies existing visibilities
is_randomized	Whether this systematic contains a randomized component
return_type	Whether the returned value is per-antenna, per-baseline, or the full array

hera_sim.sigchain.CrossCouplingSpectrum.attrs_to_pull

CrossCouplingSpectrum.**attrs_to_pull**: dict = {'autovis': None}

Mapping between parameter names and Simulator attributes

hera_sim.sigchain.CrossCouplingSpectrum.is_multiplicative

CrossCouplingSpectrum.**is_multiplicative**: bool = False

Whether this systematic multiplies existing visibilities

hera_sim.sigchain.CrossCouplingSpectrum.is_randomized

CrossCouplingSpectrum.**is_randomized**: bool = True

Whether this systematic contains a randomized component

hera_sim.sigchain.CrossCouplingSpectrum.return_type

CrossCouplingSpectrum.**return_type**: str | None = 'per_baseline'

Whether the returned value is per-antenna, per-baseline, or the full array

hera_sim.sigchain.Crosstalk

class hera_sim.sigchain.**Crosstalk**(***kwargs*)

Base class for cross-talk models.

This is an *abstract* class, and should not be directly instantiated. It represents a "component" – a modular part of a simulation for which several models may be defined. Models for this component may be defined by subclassing this abstract base class and implementing (at least) the *__call__()* method. Some of these are implemented within hera_sim already, but custom models may be implemented outside of hera_sim, and used on equal footing with the the internal models (as long as they subclass this abstract component).

As with all components, all parameters that define the behaviour of the model are accepted at class instantiation. The *__call__()* method actually computes the simulated effect of the component (typically, but not always, a set of visibilities or gains), by *default* using these parameters. However, these parameters can be over-ridden at call-time. Inputs such as the frequencies, times or baselines at which to compute the effect are specific to the call, and do not get passed at instantiation.

Methods

__call__(**kwargs)	Compute the component model.
get_aliases()	Get all the aliases by which this model can be identified.
get_model(mdl)	Get a model with a particular name (including aliases).
get_models([with_aliases])	Get a dictionary of models associated with this component.

hera_sim.sigchain.Crosstalk.__call__

abstract Crosstalk.**__call__**(***kwargs*)

Compute the component model.

hera_sim.sigchain.Crosstalk.get_aliases

classmethod Crosstalk.**get_aliases**() → tuple[str]

Get all the aliases by which this model can be identified.

hera_sim.sigchain.Crosstalk.get_model

classmethod Crosstalk.**get_model**(*mdl: str*) → *SimulationComponent*

Get a model with a particular name (including aliases).

hera_sim.sigchain.Crosstalk.get_models

classmethod Crosstalk.**get_models**(*with_aliases=False*) → dict[str, *hera_sim.components.SimulationComponent*]

Get a dictionary of models associated with this component.

Attributes

attrs_to_pull	Mapping between parameter names and Simulator attributes
is_multiplicative	Whether this systematic multiplies existing visibilities
is_randomized	Whether this systematic contains a randomized component
return_type	Whether the returned value is per-antenna, per-baseline, or the full array

hera_sim.sigchain.Crosstalk.attrs_to_pull

Crosstalk.**attrs_to_pull**: dict = {}

Mapping between parameter names and Simulator attributes

hera_sim.sigchain.Crosstalk.is_multiplicative

Crosstalk.**is_multiplicative**: bool = False

Whether this systematic multiplies existing visibilities

hera_sim.sigchain.Crosstalk.is_randomized

Crosstalk.**is_randomized**: bool = False

Whether this systematic contains a randomized component

hera_sim.sigchain.Crosstalk.return_type

Crosstalk.**return_type**: str | None = None

Whether the returned value is per-antenna, per-baseline, or the full array

hera_sim.sigchain.Gain

class hera_sim.sigchain.**Gain**(***kwargs*)

Base class for systematic gains.

This is an *abstract* class, and should not be directly instantiated. It represents a "component" – a modular part of a simulation for which several models may be defined. Models for this component may be defined by subclassing this abstract base class and implementing (at least) the *__call__()* method. Some of these are implemented within hera_sim already, but custom models may be implemented outside of hera_sim, and used on equal footing with the the internal models (as long as they subclass this abstract component).

As with all components, all parameters that define the behaviour of the model are accepted at class instantiation. The *__call__()* method actually computes the simulated effect of the component (typically, but not always, a set of visibilities or gains), by *default* using these parameters. However, these parameters can be over-ridden at call-time. Inputs such as the frequencies, times or baselines at which to compute the effect are specific to the call, and do not get passed at instantiation.

Methods

__call__(**kwargs)	Compute the component model.
get_aliases()	Get all the aliases by which this model can be identified.
get_model(mdl)	Get a model with a particular name (including aliases).
get_models([with_aliases])	Get a dictionary of models associated with this component.

hera_sim.sigchain.Gain.__call__

abstract Gain.**__call__**(***kwargs*)

Compute the component model.

hera_sim.sigchain.Gain.get_aliases

classmethod Gain.**get_aliases**() → tuple[str]

Get all the aliases by which this model can be identified.

hera_sim.sigchain.Gain.get_model

classmethod Gain.**get_model**(*mdl: str*) → *SimulationComponent*

Get a model with a particular name (including aliases).

hera_sim.sigchain.Gain.get_models

classmethod Gain.**get_models**(*with_aliases=False*) → dict[str, *hera_sim.components.SimulationComponent*]

Get a dictionary of models associated with this component.

Attributes

attrs_to_pull	Mapping between parameter names and Simulator attributes
is_multiplicative	Whether this systematic multiplies existing visibilities
is_randomized	Whether this systematic contains a randomized component
return_type	Whether the returned value is per-antenna, per-baseline, or the full array

hera_sim.sigchain.Gain.attrs_to_pull

Gain.**attrs_to_pull: dict = {}**

Mapping between parameter names and Simulator attributes

hera_sim.sigchain.Gain.is_multiplicative

Gain.**is_multiplicative**: bool = **False**

Whether this systematic multiplies existing visibilities

hera_sim.sigchain.Gain.is_randomized

Gain.**is_randomized**: bool = **False**

Whether this systematic contains a randomized component

hera_sim.sigchain.Gain.return_type

Gain.**return_type**: str | None = **None**

Whether the returned value is per-antenna, per-baseline, or the full array

hera_sim.sigchain.MutualCoupling

class hera_sim.sigchain.**MutualCoupling**(*uvbeam: UVBeam | str | Path | None = None*, *reflection: ndarray | Callable | None = None*, *omega_p: ndarray | Callable | None = None*, *ant_1_array: ndarray | None = None*, *ant_2_array: ndarray | None = None*, *pol_array: ndarray | None = None*, *array_layout: dict | None = None*, *coupling_matrix: ndarray | None = None*, *pixel_interp: str = 'az_za_simple'*, *freq_interp: str = 'cubic'*, *beam_kwargs: dict | None = None*, *use_numba: bool = True*)

Simulate mutual coupling according to Josaitis+ 2022.

This class simulates the "first-order coupling" between visibilities in an array. The model assumes that coupling is induced via re-radiation of incident astrophysical radiation due to an impedance mismatch at the antenna feed, and that the re-radiated signal is in the far-field of every other antenna in the array. Full details can be found here:

MNRAS

arXiv

The essential equations from the paper are Equations 9 and 19. The implementation here effectively calculates Equation 19 for every visibility in the provided data. The original publication contains an error in Equation 9 (the effective height in transmission should have a complex conjugation applied), which we correct for in our implementation. In addition to this, we assume that every antenna feed has the same impedance, reflection coefficient, and effective height. Applying the correct conjugation, and enforcing these assumptions, the first-order correction to the visibility $\mathbf{V}_{ij}$ can be written as:

$$\mathbf{V}_{ij}^{\mathrm{xt}} = \sum_k \Big[(1-\delta_{kj})\mathbf{V}_{ik}^0 \mathbf{X}_{jk}^\dagger + (1-\delta_{ik})\mathbf{X}_{ik}\mathbf{V}_{kj}^0\Big],$$

where the "xt" superscript is shorthand for "crosstalk", the "0" superscript refers to the "zeroth-order" visibilities, δ_{ij} is the Kronecker delta, and $\mathbf{X}_{ij}$ is a "coupling matrix" that describes how radiation emitted from antenna j is received by antenna i. The coupling matrix can be written as

$$\mathbf{X}_{jk} \equiv \frac{i\eta_0}{4\lambda}\frac{\Gamma_k}{R_k}\frac{e^{i2\pi\nu\tau_{jk}}}{b_{jk}}\mathbf{J}_j(\hat{\mathbf{b}}_{jk})\mathbf{J}_k(\hat{\mathbf{b}}_{kj})^\dagger h_0^2,$$

where Γ is the reflection coefficient, R is the real part of the impedance, η_0 is the impedance of free space, λ is the wavelength of the radiation, ν is the frequency of the radiation, $\tau = b/c$ is the delay of the baseline, b is the baseline length, $\hat{\mathbf{b}}_{ij}$ is a unit vector pointing from antenna i to antenna j, $\mathbf{J}$ is the Jones matrix describing the antenna's peak-normalized far-field radiation pattern, and h_0 is the amplitude of the antenna's effective height.

The boldfaced variables without any overhead decorations indicate 2x2 matrices:

$$\mathbf{V} = \begin{pmatrix} V_{XX} & V_{XY} \\ V_{YX} & V_{YY} \end{pmatrix}, \quad \mathbf{J} = \frac{1}{h_0} \begin{pmatrix} h_{X\theta} & h_{X\phi} \\ h_{Y\theta} & h_{Y\phi} \end{pmatrix}$$

The effective height can be rewritten as

$$h_0^2 = \frac{4\lambda^2 R}{\eta_0 \Omega_p}$$

where Ω_p is the beam area (i.e. integral of the peak-normalized power beam). Substituting this in to the previous expression for the coupling coefficient and taking antennas to be identical gives

$$\mathbf{X}_{jk} = \frac{i\Gamma}{\Omega_p} \frac{e^{i2\pi\nu\tau_{jk}}}{b_{jk}/\lambda} \mathbf{J}(\hat{\mathbf{b}}_{jk}) \mathbf{J}(\hat{\mathbf{b}}_{kj})^\dagger.$$

In order to efficiently simulate the mutual coupling, the antenna and polarization axes of the visibilities and coupling matrix are combined into a single "antenna-polarization" axis, and the problem is recast as a simple matrix multiplication.

Parameters

- **uvbeam** – The beam (i.e. Jones matrix) to be used for calculating the coupling matrix. This may either be a `pyuvdata.UVBeam` object, a path to a file that may be read into a `pyuvdata.UVBeam` object, or a string identifying which `pyuvsim.AnalyticBeam` to use. Not required if providing a pre-calculated coupling matrix.
- **reflection** – The reflection coefficient to use for calculating the coupling matrix. Should be either a `np.ndarray` or an interpolation object that gives the reflection coefficient as a function of frequency (in GHz). Not required if providing a pre-calculated coupling matrix.
- **omega_p** – The integral of the peak-normalized power beam as a function of frequency (in GHz). Not required if providing a pre-calculated coupling matrix.
- **ant_1_array** – Array of integers specifying the number of the first antenna in each visibility. Required for calculating the coupling matrix and the coupled visibilities.
- **ant_2_array** – Array of integers specifying the number of the second antenna in each visibility.
- **pol_array** – Array of integers representing polarization numbers, following the convention used for `pyuvdata.UVData` objects. Required for calculating the coupled visibilities.
- **array_layout** – Dictionary mapping antenna numbers to their positions in local East- North-Up coordinates, expressed in meters. Not required if providing a pre-calculated coupling matrix.
- **coupling_matrix** – Matrix describing how radiation is coupled between antennas in the array. Should have shape *(1, n_freqs, 2*n_ants, 2*n_ants)*. The even elements along the "antenna-polarization" axes correspond to the "X" polarization; the odd elements correspond to the "Y" polarization.
- **pixel_interp** – The name of the spatial interpolation method used for the beam. Not required if using an analytic beam or if providing a pre-computed coupling matrix.
- **freq_interp** – The order of the spline to be used for interpolating the beam in frequency. Not required if using an analytic beam or if providing a pre-computed coupling matrix.

- **beam_kwargs** – Additional keywords used for either reading in a beam or creating an analytic beam.
- **use_numba** – Whether to use `numba` for accelerating the simulation. Default is to use `numba` if it is installed.

Methods

__call__(freqs, visibilities, **kwargs)	Calculate the first-order coupled visibilities.
build_coupling_matrix(freqs, array_layout, ...)	Calculate the coupling matrix used for mutual coupling simulation.
get_aliases()	Get all the aliases by which this model can be identified.
get_model(mdl)	Get a model with a particular name (including aliases).
get_models([with_aliases])	Get a dictionary of models associated with this component.

hera_sim.sigchain.MutualCoupling.__call__

`MutualCoupling.__call__`(*freqs: ndarray*, *visibilities: ndarray*, ***kwargs*) → ndarray

Calculate the first-order coupled visibilities.

Parameters

- **freqs** – The observed frequencies, in GHz.
- **visibilities** – The full set of visibilities for the array. Should have shape *(n_bls*n_times, n_freqs, [1,] n_pols)*.
- **kwargs** – Additional parameters to use instead of the current attribute values for the class instance. See the class docstring for details.

Returns

xt_vis – The first-order correction to the visibilities due to mutual coupling between array elements. Has the same shape as the provided visibilities.

Notes

This method is somewhat memory hungry, as it produces two arrays which are each twice as large as the input visibility array in intermediate steps of the calculation.

hera_sim.sigchain.MutualCoupling.build_coupling_matrix

static `MutualCoupling.build_coupling_matrix`(*freqs: ndarray*, *array_layout: dict*, *uvbeam: UVBeam | str*, *reflection: ndarray | Callable | None = None*, *omega_p: ndarray | Callable | None = None*, *pixel_interp: str | None = 'az_za_simple'*, *freq_interp: str | None = 'cubic'*, ***beam_kwargs*) → ndarray

Calculate the coupling matrix used for mutual coupling simulation.

See the *MutualCoupling* class docstring for a description of the coupling matrix.

Parameters

- **freqs** – The observed frequencies, in GHz.
- **array_layout** – Dictionary mapping antenna numbers to their positions in local East-North-Up coordinates, expressed in meters. Not required if providing a pre-calculated coupling matrix.
- **uvbeam** – The beam (i.e. Jones matrix) to be used for calculating the coupling matrix. This may either be a `pyuvdata.UVBeam` object, a path to a file that may be read into a `pyuvdata.UVBeam` object, or a string identifying which `pyuvsim.AnalyticBeam` to use. Not required if providing a pre-calculated coupling matrix.
- **reflection** – The reflection coefficient to use for calculating the coupling matrix. Should be either a `np.ndarray` or an interpolation object that gives the reflection coefficient as a function of frequency (in GHz).
- **omega_p** – The integral of the peak-normalized power beam as a function of frequency (in GHz). If this is not provided, then it will be calculated from the provided beam model.
- **pixel_interp** – The name of the spatial interpolation method used for the beam. Not required if using an analytic beam or if providing a pre-computed coupling matrix.
- **freq_interp** – The order of the spline to be used for interpolating the beam in frequency. Not required if using an analytic beam or if providing a pre-computed coupling matrix.
- **beam_kwargs** – Additional keywords used for either reading in a beam or creating an analytic beam.

hera_sim.sigchain.MutualCoupling.get_aliases

classmethod MutualCoupling.**get_aliases**() → tuple[str]

Get all the aliases by which this model can be identified.

hera_sim.sigchain.MutualCoupling.get_model

classmethod MutualCoupling.**get_model**(*mdl: str*) → *SimulationComponent*

Get a model with a particular name (including aliases).

hera_sim.sigchain.MutualCoupling.get_models

classmethod MutualCoupling.**get_models**(*with_aliases=False*) → dict[str, *hera_sim.components.SimulationComponent*]

Get a dictionary of models associated with this component.

Attributes

attrs_to_pull	Mapping between parameter names and Simulator attributes
is_multiplicative	Whether this systematic multiplies existing visibilities
is_randomized	Whether this systematic contains a randomized component
return_type	Whether the returned value is per-antenna, per-baseline, or the full array

hera_sim.sigchain.MutualCoupling.attrs_to_pull

MutualCoupling.**attrs_to_pull**: dict = {'ant_1_array': 'ant_1_array', 'ant_2_array': 'ant_2_array', 'array_layout': 'antpos', 'pol_array': 'polarization_array', 'visibilities': 'data_array'}

Mapping between parameter names and Simulator attributes

hera_sim.sigchain.MutualCoupling.is_multiplicative

MutualCoupling.**is_multiplicative**: bool = False

Whether this systematic multiplies existing visibilities

hera_sim.sigchain.MutualCoupling.is_randomized

MutualCoupling.**is_randomized**: bool = False

Whether this systematic contains a randomized component

hera_sim.sigchain.MutualCoupling.return_type

MutualCoupling.**return_type**: str | None = 'full_array'

Whether the returned value is per-antenna, per-baseline, or the full array

hera_sim.sigchain.OverAirCrossCoupling

class hera_sim.sigchain.**OverAirCrossCoupling**(*emitter_pos: ndarray | Sequence | None = None*, *cable_delays: dict[int, float] | None = None*, *base_amp: float = 2e-05*, *amp_norm: float = 100*, *amp_slope: float = -1*, *amp_decay_base: float = 10*, *n_copies: int = 10*, *amp_jitter: float = 0*, *dly_jitter: float = 0*, *max_delay: float = 2000*, *amp_decay_fac: float = 0.01*, *rng: Generator | None = None*)

Crosstalk model based on the mechanism described in HERA Memo 104.

This model describes first-order coupling between a visibility V_{ij} and the autocorrelations for each antenna involved. Physically, it is modeled as the signal from one antenna traveling to the receiverator, then being broadcast

to the other antenna. Under this model, the cross-coupling component V_{ij}^{cc} can be described via

$$V_{ij}^{\mathrm{cc}} = \epsilon_{ij}^{*} V_{ii} + \epsilon_{ji} V_{jj},$$

where the reflection coefficient ϵ_{ij} is modeled as

$$\epsilon_{ij} = A_i \exp\big[2\pi i \nu(\tau_{i,\mathrm{cable}} + \tau_{X\to j})\big].$$

Here, X denotes the position of the receiverator (or rather, where the excess signal is radiated from), and the indices i, j refer to antennas. So, $\tau_{i,\mathrm{cable}}$ is the delay from the signal traveling down the cable from antenna i to the receiverator, and $\tau_{X\to j}$ denotes the delay from the signal traveling over-the-air from the receiverator to antenna j. As usual, A_i is the amplitude of the reflection coefficient. Here, the amplitude is described by three free parameters, $a, \vec{r}_X, \beta$:

$$A_i = a|\vec{r}_i - \vec{r}_X|^{\beta}.$$

a is a base amplitude, $\vec{r}_X$ is the receiverator position, and β describes how quickly the amplitude falls off with distance from the receiverator, and is typically taken to be negative. For more details, refer to HERA Memo 104 for more details:

http://reionization.org/manual_uploads/HERA104_Crosstalk_Physical_Model.html

Parameters

- **emitter_pos** – Receiverator position, in meters, in local ENU coordinates.
- **cable_delays** – Mapping from antenna numbers to cable delays, in nanoseconds.
- **base_amp** – Base amplitude of reflection coefficient. If *amp_slope* is set to 0, then this is the amplitude of all of the reflection coefficients.
- **amp_norm** – Distance from the receiverator, in meteres, at which the cross-coupling amplitude is equal to `base_amp`.
- **amp_slope** – Power-law index describing how rapidly the reflection coefficient decays with distance from the receiverator.
- **amp_decay_base** – Logarithmic base to use when generating the additional peaks in the cross-coupling spectrum.
- **n_copies** – Number of peaks in the cross-coupling spectrum at positive and negative delays, separately.
- **amp_jitter** – Fractional jitter to apply to the amplitudes of the peaks in the cross-coupling spectrum.
- **dly_jitter** – Absolute jitter to apply to the delays of the peaks in the cross-coupling spectrum, in nanoseconds.
- **max_delay** – Magnitude of the maximum delay to which the cross-coupling spectrum extends, in nanoseconds.
- **amp_decay_fac** – Ratio of the amplitude of the last peak in the cross-coupling spectrum to the first peak. In other words, how much the cross-coupling spectrum decays over the full range of delays it covers.
- **rng** – Random number generator.

See also:

CrossCouplingSpectrum

Methods

__call__(freqs, antpair, antpos, autovis_i, ...)	Generate a cross-coupling spectrum modeled via HERA Memo 104.
get_aliases()	Get all the aliases by which this model can be identified.
get_model(mdl)	Get a model with a particular name (including aliases).
get_models([with_aliases])	Get a dictionary of models associated with this component.

hera_sim.sigchain.OverAirCrossCoupling.__call__

OverAirCrossCoupling.**__call__**(*freqs: ndarray*, *antpair: tuple[int, int]*, *antpos: dict[int, numpy.ndarray]*, *autovis_i: ndarray*, *autovis_j: ndarray*, ***kwargs*) → ndarray

Generate a cross-coupling spectrum modeled via HERA Memo 104.

Parameters

- **freqs** – Frequencies at which to evaluate the reflection coefficients, in GHz.
- **antpair** – The two antennas involved in forming the visibility.
- **antpos** – Mapping from antenna numbers to positions in meters, in local ENU coordinates.
- **autovis_i** – Autocorrelation for the first antenna in the pair.
- **autovis_j** – Autocorrelation for the second antenna in the pair.

Returns

xtalk_vis – Array with the cross-coupling visibility. Has the same shape as the input autocorrelations. This systematic is not applied to the auto-correlations.

hera_sim.sigchain.OverAirCrossCoupling.get_aliases

classmethod OverAirCrossCoupling.**get_aliases**() → tuple[str]

Get all the aliases by which this model can be identified.

hera_sim.sigchain.OverAirCrossCoupling.get_model

classmethod OverAirCrossCoupling.**get_model**(*mdl: str*) → *SimulationComponent*

Get a model with a particular name (including aliases).

hera_sim.sigchain.OverAirCrossCoupling.get_models

classmethod OverAirCrossCoupling.**get_models**(*with_aliases=False*) → dict[str, *hera_sim.components.SimulationComponent*]

Get a dictionary of models associated with this component.

Attributes

attrs_to_pull	Mapping between parameter names and Simulator attributes
is_multiplicative	Whether this systematic multiplies existing visibilities
is_randomized	Whether this systematic contains a randomized component
return_type	Whether the returned value is per-antenna, per-baseline, or the full array

hera_sim.sigchain.OverAirCrossCoupling.attrs_to_pull

OverAirCrossCoupling.**attrs_to_pull: dict = {'antpair': None, 'autovis_i': None, 'autovis_j': None}**

Mapping between parameter names and Simulator attributes

hera_sim.sigchain.OverAirCrossCoupling.is_multiplicative

OverAirCrossCoupling.**is_multiplicative: bool = False**

Whether this systematic multiplies existing visibilities

hera_sim.sigchain.OverAirCrossCoupling.is_randomized

OverAirCrossCoupling.**is_randomized: bool = True**

Whether this systematic contains a randomized component

hera_sim.sigchain.OverAirCrossCoupling.return_type

OverAirCrossCoupling.**return_type: str | None = 'per_baseline'**

Whether the returned value is per-antenna, per-baseline, or the full array

hera_sim.sigchain.ReflectionSpectrum

class hera_sim.sigchain.**ReflectionSpectrum**(*n_copies: int = 20*, *amp_range: tuple[float, float] = (-3, -4)*, *dly_range: tuple[float, float] = (200, 1000)*, *phs_range: tuple[float, float] = (-3.141592653589793, 3.141592653589793)*, *amp_jitter: float = 0.05*, *dly_jitter: float = 30*, *amp_logbase: float = 10*, *rng: Generator | None = None*)

Generate many reflections between a range of delays.

Amplitudes are distributed on a logarithmic grid, while delays are distributed on a linear grid. Effectively, this gives a reflection spectrum whose amplitude decreases exponentially over the range of delays specified.

Parameters

- **n_copies** – Number of peaks in the reflection spectrum.
- **amp_range** – Max/min of the amplitudes of the reflections in the spectrum. The spectrum amplitudes monotonically decrease (up to jitter).
- **dly_range** – Min/max of the delays at which the reflections are injected, in ns.
- **phs_range** – Bounds of the uniform distribution from which to draw reflection phases.
- **amp_jitter** – Fractional jitter in amplitude across antennas for each of the reflections.
- **dly_jitter** – Absolute jitter in delay across antennas for each of the reflections.
- **amp_logbase** – Base of the logarithm to use for generating reflection amplitudes.
- **rng** – Random number generator.

Notes

The generated amplitudes will be in the range `amp_logbase ** amp_range[0]` to `amp_logbase ** amp_range[1]`.

Methods

__call__(freqs, ants, **kwargs)	Generate a series of reflections.
get_aliases()	Get all the aliases by which this model can be identified.
get_model(mdl)	Get a model with a particular name (including aliases).
get_models([with_aliases])	Get a dictionary of models associated with this component.

hera_sim.sigchain.ReflectionSpectrum.__call__

ReflectionSpectrum.**__call__**(*freqs: ndarray*, *ants: Sequence[int]*, ***kwargs*) → dict[int, numpy.ndarray]

Generate a series of reflections.

Parameters

- **freqs** – Frequencies at which to calculate the reflection coefficients. These should be provided in GHz.
- **ants** – Antenna numbers for which to generate reflections.

Returns

reflection_gains – Reflection gains for each antenna.

hera_sim.sigchain.ReflectionSpectrum.get_aliases

classmethod ReflectionSpectrum.**get_aliases**() → tuple[str]

Get all the aliases by which this model can be identified.

hera_sim.sigchain.ReflectionSpectrum.get_model

classmethod ReflectionSpectrum.**get_model**(*mdl: str*) → *SimulationComponent*

Get a model with a particular name (including aliases).

hera_sim.sigchain.ReflectionSpectrum.get_models

classmethod ReflectionSpectrum.**get_models**(*with_aliases=False*) → dict[str, *hera_sim.components.SimulationComponent*]

Get a dictionary of models associated with this component.

Attributes

attrs_to_pull	Mapping between parameter names and Simulator attributes
is_multiplicative	Whether this systematic multiplies existing visibilities
is_randomized	Whether this systematic contains a randomized component
return_type	Whether the returned value is per-antenna, per-baseline, or the full array

hera_sim.sigchain.ReflectionSpectrum.attrs_to_pull

ReflectionSpectrum.**attrs_to_pull**: dict = {'ants': 'antpos'}

Mapping between parameter names and Simulator attributes

hera_sim.sigchain.ReflectionSpectrum.is_multiplicative

ReflectionSpectrum.**is_multiplicative**: bool = True

Whether this systematic multiplies existing visibilities

hera_sim.sigchain.ReflectionSpectrum.is_randomized

ReflectionSpectrum.**is_randomized**: bool = True

Whether this systematic contains a randomized component

hera_sim.sigchain.ReflectionSpectrum.return_type

ReflectionSpectrum.**return_type**: str | None = 'per_antenna'

Whether the returned value is per-antenna, per-baseline, or the full array

hera_sim.sigchain.Reflections

class hera_sim.sigchain.**Reflections**(*amp=None*, *dly=None*, *phs=None*, *conj=False*, *amp_jitter=0*, *dly_jitter=0*, *rng=None*)

Produce multiplicative reflection gains.

Parameters

- **amp** (*float, optional*) – Mean Amplitude of the reflection gains.
- **dly** (*float, optional*) – Mean delay of the reflection gains.
- **phs** (*float, optional*) – Phase of the reflection gains.
- **conj** (*bool, optional*) – Whether to conjugate the gain.
- **amp_jitter** (*float, optional*) – Final amplitudes are multiplied by a normal variable with mean one, and with standard deviation of `amp_jitter`.
- **dly_jitter** (*float, optional*) – Final delays are offset by a normal variable with mean zero and standard deviation `dly_jitter`.
- **rng** (*np.random.Generator, optional*) – Random number generator.

Methods

__call__(freqs, ants, **kwargs)	Generate the bandpass.
gen_reflection_coefficient(freqs, amp, dly, phs)	Randomly generate reflection coefficients.
get_aliases()	Get all the aliases by which this model can be identified.
get_model(mdl)	Get a model with a particular name (including aliases).
get_models([with_aliases])	Get a dictionary of models associated with this component.

hera_sim.sigchain.Reflections.__call__

Reflections.**__call__**(*freqs*, *ants*, ***kwargs*)

Generate the bandpass.

Parameters

- **freqs** (*array_like of float*) – Frequencies in units inverse to `dly`.
- **ants** (*array_like of int*) – Antenna numbers for which to produce gains.

Returns

dict – Keys are antenna numbers and values are arrays of bandpass gains.

hera_sim.sigchain.Reflections.gen_reflection_coefficient

static Reflections.**gen_reflection_coefficient**(*freqs*, *amp*, *dly*, *phs*, *conj=False*)

Randomly generate reflection coefficients.

Parameters

- **freqs** (*array_like of float*) – Frequencies, units are arbitrary but must be the inverse of `dly`.
- **amp** (*array_like of float*) – Either a scalar amplitude, or 1D with size Nfreqs, or 2D with shape (Ntimes, Nfreqs).
- **dly** (*[type]*) – Either a scalar delay, or 1D with size Nfreqs, or 2D with shape (Ntimes, Nfreqs). Units are inverse of `freqs`.
- **phs** (*[type]*) – Either a scalar phase, or 1D with size Nfreqs, or 2D with shape (Ntimes, Nfreqs). Units radians.
- **conj** (*bool, optional*) – Whether to conjugate the gain.

Returns

array_like – The reflection gains as a 2D array of (Ntimes, Nfreqs).

hera_sim.sigchain.Reflections.get_aliases

classmethod Reflections.**get_aliases**() → tuple[str]

Get all the aliases by which this model can be identified.

hera_sim.sigchain.Reflections.get_model

classmethod Reflections.**get_model**(*mdl: str*) → *SimulationComponent*

Get a model with a particular name (including aliases).

hera_sim.sigchain.Reflections.get_models

classmethod Reflections.**get_models**(*with_aliases=False*) → dict[str, *hera_sim.components.SimulationComponent*]

Get a dictionary of models associated with this component.

Attributes

attrs_to_pull	Mapping between parameter names and Simulator attributes
is_multiplicative	Whether this systematic multiplies existing visibilities
is_randomized	Whether this systematic contains a randomized component
return_type	Whether the returned value is per-antenna, per-baseline, or the full array

hera_sim.sigchain.Reflections.attrs_to_pull

Reflections.**attrs_to_pull: dict = {'ants': 'antpos'}**

Mapping between parameter names and Simulator attributes

hera_sim.sigchain.Reflections.is_multiplicative

Reflections.**is_multiplicative: bool = True**

Whether this systematic multiplies existing visibilities

hera_sim.sigchain.Reflections.is_randomized

Reflections.**is_randomized**: bool = True

Whether this systematic contains a randomized component

hera_sim.sigchain.Reflections.return_type

Reflections.**return_type**: str | None = 'per_antenna'

Whether the returned value is per-antenna, per-baseline, or the full array

hera_sim.sigchain.WhiteNoiseCrosstalk

class hera_sim.sigchain.**WhiteNoiseCrosstalk**(*amplitude=3.0*, *rng=None*)

Generate cross-talk that is simply white noise.

Parameters

- **amplitude** (*float, optional*) – The amplitude of the white noise spectrum (i.e. its standard deviation).
- **rng** (*np.random.Generator, optional*) – Random number generator.

Methods

__call__(freqs, **kwargs)	Compute the cross-correlations.
get_aliases()	Get all the aliases by which this model can be identified.
get_model(mdl)	Get a model with a particular name (including aliases).
get_models([with_aliases])	Get a dictionary of models associated with this component.

hera_sim.sigchain.WhiteNoiseCrosstalk.__call__

WhiteNoiseCrosstalk.**__call__**(*freqs*, ***kwargs*)

Compute the cross-correlations.

Parameters

freqs (*array_like of float*) – Frequencies in units inverse to `dly`.

Returns

array – The cross-coupling contribution to the visibility, same shape as `freqs`.

hera_sim.sigchain.WhiteNoiseCrosstalk.get_aliases

classmethod WhiteNoiseCrosstalk.**get_aliases**() → tuple[str]

Get all the aliases by which this model can be identified.

hera_sim.sigchain.WhiteNoiseCrosstalk.get_model

classmethod WhiteNoiseCrosstalk.**get_model**(*mdl: str*) → *SimulationComponent*

Get a model with a particular name (including aliases).

hera_sim.sigchain.WhiteNoiseCrosstalk.get_models

classmethod WhiteNoiseCrosstalk.**get_models**(*with_aliases=False*) → dict[str, *hera_sim.components.SimulationComponent*]

Get a dictionary of models associated with this component.

Attributes

attrs_to_pull	Mapping between parameter names and Simulator attributes
is_multiplicative	Whether this systematic multiplies existing visibilities
is_randomized	Whether this systematic contains a randomized component
return_type	Whether the returned value is per-antenna, per-baseline, or the full array

hera_sim.sigchain.WhiteNoiseCrosstalk.attrs_to_pull

WhiteNoiseCrosstalk.**attrs_to_pull: dict = {}**

Mapping between parameter names and Simulator attributes

hera_sim.sigchain.WhiteNoiseCrosstalk.is_multiplicative

WhiteNoiseCrosstalk.**is_multiplicative: bool = False**

Whether this systematic multiplies existing visibilities

hera_sim.sigchain.WhiteNoiseCrosstalk.is_randomized

WhiteNoiseCrosstalk.**is_randomized: bool = True**

Whether this systematic contains a randomized component

hera_sim.sigchain.WhiteNoiseCrosstalk.return_type

WhiteNoiseCrosstalk.**return_type: str | None = 'per_baseline'**

Whether the returned value is per-antenna, per-baseline, or the full array

hera_sim.simulate

Module containing a high-level interface for `hera_sim`.

This module defines the *Simulator* class, which provides the user with a high-level interface to all of the features provided by `hera_sim`. For detailed instructions on how to manage a simulation using the *Simulator*, please refer to the tutorials.

Classes

Simulator(*[, data, defaults_config, ...])	Simulate visibilities and/or instrumental effects for an entire array.

hera_sim.simulate.Simulator

class hera_sim.simulate.**Simulator**(*, *data: str | UVData | None = None*, *defaults_config: str | dict | None = None*, *redundancy_tol: float = 1.0*, ***kwargs*)

Simulate visibilities and/or instrumental effects for an entire array.

Parameters

- **data** – `pyuvdata.UVData` object to use for the simulation or path to a UVData-supported file.
- **defaults_config** – Path to defaults configuraiton, seasonal keyword, or configuration dictionary for setting default simulation parameters. See tutorial on setting defaults for further information.
- **redundancy_tol** – Position tolerance for finding redundant groups, in meters. Default is 1 meter.
- **kwargs** – Parameters to use for initializing UVData object if none is provided. If `data` is a file path, then these parameters are used when reading the file. Otherwise, the parameters are used in creating a `UVData` object using *empty_uvdata()*.

Variables

- **data** (`pyuvdata.UVData` instance) – Object containing simulated visibilities and metadata.
- **extras** (*dict*) – Dictionary to use for storing extra parameters.
- **antpos** (*dict*) – Dictionary pairing antenna numbers to ENU positions in meters.

- **lsts** (*np.ndarray of float*) – Observed LSTs in radians.
- **freqs** (*np.ndarray of float*) – Observed frequencies in GHz.
- **times** (*np.ndarray of float*) – Observed times in JD.
- **pols** (*list of str*) – Polarization strings.
- **red_grps** (*list of list of int*) – Redundant baseline groups. Each entry is a list containing the baseline integer for each member of that redundant group.
- **red_vecs** (list of numpy.ndarray of float) – Average of all the baselines for each redundant group.
- **red_lengths** (*list of float*) – Length of each redundant baseline.

Methods

add(component, *[, add_vis, ret_vis, seed, ...])	Simulate an effect then apply and/or return the result.
apply_defaults(config[, refresh])	Apply the provided default configuration.
calculate_filters(*[, delay_filter_kwargs, ...])	Pre-compute fringe-rate and delay filters for the entire array.
chunk_sim_and_save(save_dir[, ref_files, ...])	Chunk a simulation in time and write to disk.
get(component[, key])	Retrieve an effect that was previously simulated.
plot_array()	Generate a plot of the array layout in ENU coordinates.
refresh()	Refresh the object.
run_sim([sim_file])	Run an entire simulation.
write(filename[, save_format])	Write the `data` to disk using a `pyuvdata`-supported filetype.

hera_sim.simulate.Simulator.add

Simulator.**add**(*component: str | type[*hera_sim.components.SimulationComponent*] | SimulationComponent, *, add_vis: bool = True, ret_vis: bool = False, seed: str | int | None = None, vis_filter: Sequence | None = None, component_name: str | None = None, **kwargs*) → ndarray | dict[int, numpy.ndarray] | None

Simulate an effect then apply and/or return the result.

Parameters

- **component** – Effect to be simulated. This can either be an alias of the effect, or the class (or instance thereof) that simulates the effect.
- **add_vis** – Whether to apply the effect to the simulated data. Default is True.
- **ret_vis** – Whether to return the simulated effect. Nothing is returned by default.
- **seed** – How to seed the random number generator. Can either directly provide a seed as an integer, or use one of the supported keywords. See tutorial for using the *Simulator* for supported seeding modes. Default is to use a seed based on the current random state.
- **vis_filter** – Iterable specifying which antennas/polarizations for which the effect should be simulated. See tutorial for using the *Simulator* for details of supported formats and functionality.

- **component_name** – Name to use when recording the parameters used for simulating the effect. Default is to use the name of the class used to simulate the effect.
- ****kwargs** – Optional keyword arguments for the provided `component`.

Returns

effect – The simulated effect; only returned if `ret_vis` is set to `True`. If the simulated effect is multiplicative, then a dictionary mapping antenna numbers to the per-antenna effect (as a `np.ndarray`) is returned. Otherwise, the effect for the entire array is returned with the same structure as the `pyuvdata.UVData.data_array` that the data is stored in.

hera_sim.simulate.Simulator.apply_defaults

`Simulator.`**`apply_defaults`**(*config: str | dict | None*, *refresh: bool = True*)

Apply the provided default configuration.

Equivalent to calling `set()` with the same parameters.

Parameters

- **config** – If given, either a path pointing to a defaults configuration file, a string identifier of a particular config (e.g. 'h1c') or a dictionary of configuration parameters (see `Defaults`).
- **refresh** – Whether to refresh the defaults.

hera_sim.simulate.Simulator.calculate_filters

`Simulator.`**`calculate_filters`**(**, delay_filter_kwargs: dict[str, Union[float, str]] | None = None*, *fringe_filter_kwargs: dict[str, Union[float, str, numpy.ndarray]] | None = None*)

Pre-compute fringe-rate and delay filters for the entire array.

Parameters

- **delay_filter_kwargs** – Extra parameters necessary for generating a delay filter. See `utils.gen_delay_filter()` for details.
- **fringe_filter_kwargs** – Extra parameters necessary for generating a fringe filter. See `utils.gen_fringe_filter()` for details.

hera_sim.simulate.Simulator.chunk_sim_and_save

`Simulator.`**`chunk_sim_and_save`**(*save_dir*, *ref_files=None*, *Nint_per_file=None*, *prefix=None*, *sky_cmp=None*, *state=None*, *filetype='uvh5'*, *clobber=True*)

Chunk a simulation in time and write to disk.

This function is a thin wrapper around *chunk_sim_and_save()*; please see that function's documentation for more information.

hera_sim.simulate.Simulator.get

Simulator.**get**(*component: str | type[hera_sim.components.SimulationComponent] | SimulationComponent*, *key: int | str | tuple[int, int] | tuple[int, int, str] | None = None*) → ndarray | dict[int, numpy.ndarray]

Retrieve an effect that was previously simulated.

Parameters

- **component** – Effect that is to be retrieved. See *add()* for more details.
- **key** –

  Key for retrieving simulated effect. Possible choices are as follows:
  An integer may specify either a single antenna (for per-antenna effects) or be a `pyuvdata`-style baseline integer. A string specifying a polarization can be used to retrieve the effect for every baseline for the specified polarization. A length-2 tuple of integers can be used to retrieve the effect for that baseline for all polarizations. A length-3 tuple specifies a particular baseline and polarization for which to retrieve the effect.

  Not specifying a key results in the effect being returned for all baselines (or antennas, if the effect is per-antenna) and polarizations.

Returns

effect – The simulated effect appropriate for the provided key. Return type depends on the effect being simulated and the provided key. See the tutorial Jupyter notebook for the *Simulator* for example usage.

Notes

This will only produce the correct output if the simulated effect is independent of the data itself. If the simulated effect contains a randomly-generated component, then the random seed must have been set when the effect was initially simulated.

hera_sim.simulate.Simulator.plot_array

Simulator.**plot_array**()

Generate a plot of the array layout in ENU coordinates.

hera_sim.simulate.Simulator.refresh

Simulator.**refresh**()

Refresh the object.

This zeros the data array, resets the history, and clears the instance's `_components` dictionary.

hera_sim.simulate.Simulator.run_sim

`Simulator.`**`run_sim`**(*sim_file=None*, ***sim_params*)

Run an entire simulation.

Parameters

- **sim_file** – Path to a configuration file specifying simulation parameters. Required if `sim_params` is not provided.
- ****sim_params** – Once-nested dictionary mapping simulation components to models, with each model mapping to a dictionary of parameter-value pairs. Required if `sim_file` is not provided.

Returns

components – List of simulation components that were generated with the parameter `ret_vis` set to `True`, returned in the order that they were simulated. This is only returned if there is at least one simulation component with `ret_vis` set to `True` in its configuration file/dictionary.

Examples

Suppose we have the following configuration dictionary:

```
sim_params = {
    "pntsrc_foreground": {"seed": "once", "nsrcs": 500},
    "gains": {"seed": "once", "dly_rng": [-20, 20], "ret_vis": True},
    "reflections": {"seed": "once", "dly_jitter": 10},
}
```

Invoking this method with `**sim_params` as its argument will simulate visibilities appropriate for a sky with 500 point sources, generate bandpass gains for each antenna and apply the effect to the foreground data, then generate cable reflections with a Gaussian jitter in the reflection delays with a standard deviation of 10 ns and apply the effect to the data. The return value will be a list with one entry: a dictionary mapping antenna numbers to their associated bandpass gains.

The same effect can be achieved by writing a YAML file that is loaded into a dictionary formatted as above. See the *Simulator* tutorial for a more in-depth explanation of how to use this method.

hera_sim.simulate.Simulator.write

`Simulator.`**`write`**(*filename*, *save_format='uvh5'*, ***kwargs*)

Write the `data` to disk using a `pyuvdata`-supported filetype.

Deprecated Methods

add_eor(model, **kwargs)	Add an EoR-like model to the visibilities.
add_foregrounds(model, **kwargs)	Add foregrounds to the visibilities.
add_gains(**kwargs)	Apply bandpass gains to the visibilities.
add_noise(model, **kwargs)	Add thermal noise to the visibilities.
add_rfi(model, **kwargs)	Add RFI to the visibilities.
add_sigchain_reflections([ants])	Apply reflections to the visibilities.
add_xtalk([model, bls])	Add crosstalk to the visibilities.

hera_sim.simulate.Simulator.add_eor

Simulator.**add_eor**(*model*, ***kwargs*)

Add an EoR-like model to the visibilities.

Deprecated since version 1.0: This will be removed in 2.0. Use the *add()* method instead.

hera_sim.simulate.Simulator.add_foregrounds

Simulator.**add_foregrounds**(*model*, ***kwargs*)

Add foregrounds to the visibilities.

Deprecated since version 1.0: This will be removed in 2.0. Use the *add()* method instead.

hera_sim.simulate.Simulator.add_gains

Simulator.**add_gains**(***kwargs*)

Apply bandpass gains to the visibilities.

Deprecated since version 1.0: This will be removed in 2.0. Use the *add()* method instead.

hera_sim.simulate.Simulator.add_noise

Simulator.**add_noise**(*model*, ***kwargs*)

Add thermal noise to the visibilities.

Deprecated since version 1.0: This will be removed in 2.0. Use the *add()* method instead.

hera_sim.simulate.Simulator.add_rfi

Simulator.**add_rfi**(*model*, ***kwargs*)

Add RFI to the visibilities.

Deprecated since version 1.0: This will be removed in 2.0. Use the *add()* method instead.

hera_sim.simulate.Simulator.add_sigchain_reflections

Simulator.**add_sigchain_reflections**(*ants=None*, ***kwargs*)

Apply reflections to the visibilities. See *add()* for details.

Deprecated since version 1.0: This will be removed in 2.0. Use the *add()* method instead.

hera_sim.simulate.Simulator.add_xtalk

Simulator.**add_xtalk**(*model='gen_whitenoise_xtalk'*, *bls=None*, ***kwargs*)

Add crosstalk to the visibilities. See *add()* for more details.

Deprecated since version 1.0: This will be removed in 2.0. Use the *add()* method instead.

Attributes

ant_1_array	
ant_2_array	
antenna_numbers	
antpos	Mapping between antenna numbers and ENU positions in meters.
channel_width	Channel width, assuming each channel is the same width.
data_array	Array storing the visibilities.
freqs	Frequencies in GHz.
integration_time	Integration time, assuming it's identical across baselines.
lsts	Observed Local Sidereal Times in radians.
polarization_array	
pols	Array of polarization strings.
times	Simulation times in JD.

hera_sim.simulate.Simulator.ant_1_array

property Simulator.**ant_1_array**

hera_sim.simulate.Simulator.ant_2_array

property Simulator.**ant_2_array**

hera_sim.simulate.Simulator.antenna_numbers

property Simulator.**antenna_numbers**

hera_sim.simulate.Simulator.antpos

property Simulator.**antpos**

Mapping between antenna numbers and ENU positions in meters.

hera_sim.simulate.Simulator.channel_width

Simulator.**channel_width**

Channel width, assuming each channel is the same width.

hera_sim.simulate.Simulator.data_array

property Simulator.**data_array**

Array storing the visibilities.

hera_sim.simulate.Simulator.freqs

property Simulator.**freqs**

Frequencies in GHz.

hera_sim.simulate.Simulator.integration_time

Simulator.**integration_time**

Integration time, assuming it's identical across baselines.

hera_sim.simulate.Simulator.lsts

property Simulator.**lsts**

Observed Local Sidereal Times in radians.

hera_sim.simulate.Simulator.polarization_array

property Simulator.**polarization_array**

hera_sim.simulate.Simulator.pols

property Simulator.**pols**

Array of polarization strings.

hera_sim.simulate.Simulator.times

property Simulator.**times**

Simulation times in JD.

hera_sim.utils

Utility module.

Functions

Jy2T(freqs, omega_p)	Convert Janskys to Kelvin.
calc_max_fringe_rate(fqs, ew_bl_len_ns)	Calculate the max fringe-rate seen by an East-West baseline.
compute_ha(lsts, ra)	Compute hour angle from local sidereal time and right ascension.
find_baseline_orientations(antenna_numbers, ...)	Find the orientation of each redundant baseline group.
gen_delay_filter(freqs, bl_len_ns[, ...])	Generate a delay filter in delay space.
gen_fringe_filter(lsts, freqs, ew_bl_len_ns)	Generate a fringe rate filter in fringe-rate & freq space.
gen_white_noise([size, rng])	Produce complex Gaussian noise with unity variance.
get_bl_len_magnitude(bl_len_ns)	Get the magnitude of the length of the given baseline.
jansky_to_kelvin(freqs, omega_p)	Return Kelvin -> Jy conversion as a function of frequency.
matmul(left, right[, use_numba])	Helper function for matrix multiplies used in mutual coupling sims.
reshape_vis(vis, ant_1_array, ant_2_array, ...)	Reshaping helper for mutual coupling sims.
rough_delay_filter(data[, freqs, bl_len_ns, ...])	A rough low-pass delay filter of data array along last axis.
rough_fringe_filter(data[, lsts, freqs, ...])	A rough fringe rate filter of data along zeroth axis.
tanh_window(x[, x_min, x_max, scale_low, ...])	
wrap2pipi(a)	Wrap values of an array to [-; +] modulo 2.

hera_sim.utils.Jy2T

hera_sim.utils.**Jy2T**(*freqs*, *omega_p*)

Convert Janskys to Kelvin.

Deprecated in v1.0.0. Will be removed in v1.1.0

hera_sim.utils.calc_max_fringe_rate

hera_sim.utils.**calc_max_fringe_rate**(*fqs: ndarray*, *ew_bl_len_ns: float*) → ndarray

Calculate the max fringe-rate seen by an East-West baseline.

Parameters

- **fqs** – Frequency array [GHz] ew_bl_len_ns (float): projected East-West baseline length [ns]
- **ew_bl_len_ns** – The EW baseline length, in nanosec.

Returns

fr_max – Maximum fringe rate [Hz]

hera_sim.utils.compute_ha

hera_sim.utils.**compute_ha**(*lsts: ndarray*, *ra: float*) → ndarray

Compute hour angle from local sidereal time and right ascension.

Parameters

- **lsts** – Local sidereal times of the observation to be generated [radians]. Shape=(NTIMES,)
- **ra** – The right ascension of a point source [radians].

Returns

ha – Hour angle corresponding to the provide ra and times. Shape=(NTIMES,)

hera_sim.utils.find_baseline_orientations

hera_sim.utils.**find_baseline_orientations**(*antenna_numbers: ndarray*, *enu_antpos: ndarray*) → dict[tuple[int, int], float]

Find the orientation of each redundant baseline group.

Parameters

- **antenna_numbers** – Array containing antenna numbers corresponding to the provided antenna positions.
- **enu_antpos** – `(Nants,3)` array containing the antenna positions in a local topocentric frame with basis (east, north, up).

Returns

antpair2angle – Dictionary mapping antenna pairs `(ai,aj)` to baseline orientations. Orientations are defined on [0,2pi).

hera_sim.utils.gen_delay_filter

hera_sim.utils.**gen_delay_filter**(*freqs: ndarray*, *bl_len_ns: float | ndarray | Sequence*, *standoff: float = 0.0*, *delay_filter_type: str | None = 'gauss'*, *min_delay: float | None = None*, *max_delay: float | None = None*, *normalize: float | None = None*) → ndarray

Generate a delay filter in delay space.

Parameters

- **freqs** – Frequency array [GHz]

- **bl_len_ns** – The baseline length in nanosec (i.e. 1e9 * metres / c). If scalar, interpreted as E-W length, if len(2), interpreted as EW and NS length, otherwise the full [EW, NS, Z] length. Unspecified dimensions are assumed to be zero.
- **standoff** – Supra-horizon buffer [nanosec]
- **delay_filter_type** – Options are `['gauss', 'trunc_gauss', 'tophat', 'none']`. This sets the filter profile. `gauss` has a 1-sigma as horizon (+ standoff) divided by four, `trunc_gauss` is same but truncated above 1-sigma. `'none'` means filter is identically one.
- **min_delay** – Minimum absolute delay of filter
- **max_delay** – Maximum absolute delay of filter
- **normalize** – If set, will normalize the filter such that the power of the output matches the power of the input times the normalization factor. If not set, the filter merely has a maximum of unity.

Returns

delay_filter – Delay filter in delay space (1D)

hera_sim.utils.gen_fringe_filter

hera_sim.utils.**gen_fringe_filter**(*lsts: ndarray*, *freqs: ndarray*, *ew_bl_len_ns: float*, *fringe_filter_type: str | None = 'tophat'*, ***filter_kwargs*) → ndarray

Generate a fringe rate filter in fringe-rate & freq space.

Parameters

- **lsts** – lst array [radians]
- **freqs** – Frequency array [GHz]
- **ew_bl_len_ns** – Projected East-West baseline length [nanosec]
- **fringe_filter_type** – Options `['tophat', 'gauss', 'custom', 'none']`
- ****filter_kwargs** – These are specific to each `fringe_filter_type`.

  For `filter_type == 'gauss'`:

  – **fr_width** (float or array): Sets gaussian width in fringe-rate [Hz]

  For `filter_type == 'custom'`:

  – **FR_filter** (ndarray): shape (Nfrates, Nfreqs) with custom filter (must be fftshifted, see below)

  – **FR_frates** (ndarray): array of FR_filter fringe rates [Hz] (must be monotonically increasing)

  – **FR_freqs** (ndarray): array of FR_filter freqs [GHz]

Returns

fringe_filter – 2D array in fringe-rate & freq space

Notes

If `filter_type == 'tophat'` filter is a tophat out to max fringe-rate set by ew_bl_len_ns.

If `filter_type == 'gauss'` filter is a Gaussian centered on max fringe-rate with width set by kwarg fr_width in Hz

If `filter_type == 'custom'` filter is a custom 2D (Nfrates, Nfreqs) filter fed as 'FR_filter' its fringe-rate array is fed as "FR_frates" in Hz, its freq array is fed as "FR_freqs" in GHz. Note that input `FR_filter` must be fft-shifted along axis 0, but output filter is `ifftshift`-ed back along axis 0.

If `filter_type == 'none'` fringe filter is identically one.

hera_sim.utils.gen_white_noise

hera_sim.utils.**gen_white_noise**(*size: int | tuple[int] = 1*, *rng: Generator | None = None*) → ndarray

Produce complex Gaussian noise with unity variance.

Parameters

- **size** – Shape of output array. Can be an integer if a single dimension is required, otherwise a tuple of ints.
- **rng** – Random number generator.

Returns

noise – White noise realization with specified shape.

hera_sim.utils.get_bl_len_magnitude

hera_sim.utils.**get_bl_len_magnitude**(*bl_len_ns: float | ndarray | Sequence*) → float

Get the magnitude of the length of the given baseline.

Parameters

bl_len_ns – The baseline length in nanosec (i.e. 1e9 * metres / c). If scalar, interpreted as E-W length, if len(2), interpreted as EW and NS length, otherwise the full [EW, NS, Z] length. Unspecified dimensions are assumed to be zero.

Returns

mag – The magnitude of the baseline length.

hera_sim.utils.jansky_to_kelvin

hera_sim.utils.**jansky_to_kelvin**(*freqs: ndarray*, *omega_p: Beam | ndarray*) → ndarray

Return Kelvin -> Jy conversion as a function of frequency.

Parameters

- **freqs** – Frequencies for which to calculate the conversion. Units of GHz.
- **omega_p** – Beam area as a function of frequency. Must have the same shape as `freqs` if an ndarray. Otherwise, must be an interpolation object which converts frequencies (in GHz) to beam size.

Returns

Jy_to_K – Array for converting Jy to K, same shape as `freqs`.

hera_sim.utils.matmul

hera_sim.utils.**matmul**(*left: ndarray*, *right: ndarray*, *use_numba: bool = False*) → ndarray

Helper function for matrix multiplies used in mutual coupling sims.

The `MutualCoupling` class performs two matrix multiplications of arrays with shapes `(1, Nfreqs, 2*Nant, 2*Nant)` and `(Ntimes, Nfreqs, 2*Nant, 2*Nant)`. Typically the number of antennas is much less than the number of frequency channels, so the parallelization used by `numpy`'s matrix multiplication routine tends to be sub-optimal. This routine–when used with `numba`–produces a substantial speedup in matrix multiplication for typical HERA-sized problems.

Parameters

- **left, right** – Input arrays to perform matrix multiplication left @ right.
- **use_numba** – Whether to use `numba` to speed up the matrix multiplication.

Returns

prod – Product of the matrix multiplication left @ right.

Notes

hera_sim.utils.reshape_vis

hera_sim.utils.**reshape_vis**(*vis: ndarray*, *ant_1_array: ndarray*, *ant_2_array: ndarray*, *pol_array: ndarray*, *antenna_numbers: ndarray*, *n_times: int*, *n_freqs: int*, *n_ants: int*, *n_pols: int*, *invert: bool = False*, *use_numba: bool = True*) → ndarray

Reshaping helper for mutual coupling sims.

The mutual coupling simulations take as input, and return, a data array with shape `(Nblts, Nfreqs, Npols)`, but perform matrix multiplications on the data array reshaped to `(Ntimes, Nfreqs, 2*Nants, 2*Nants)`. This function performs the reshaping between the matrix multiply shape and the input/output array shapes.

Parameters

- **vis** – Input data array.
- **ant_1_array** – Array specifying the first antenna in each baseline.
- **ant_2_array** – Array specifying the second antenna in each baseline.
- **pol_array** – Array specifying the observed polarizations via polarization numbers.
- **antenna_numbers** – Array specifying all of the antennas to include in the reshaped data.
- **n_times** – Number of integrations in the data.
- **n_freqs** – Number of frequency channels in the data.
- **n_ants** – Number of antennas.
- **n_pols** – Number of polarizations in the data.
- **invert** – Whether to reshape to `pyuvdata.UVData`'s data array shape.
- **use_numba** – Whether to use `numba` to speed up the reshaping.

Returns

reshaped_vis – Input data reshaped to desired shape.

hera_sim.utils.rough_delay_filter

hera_sim.utils.**rough_delay_filter**(*data: ndarray*, *freqs: ndarray | None = None*, *bl_len_ns: ndarray | None = None*, *, *delay_filter: ndarray | None = None*, ***kwargs*) → ndarray

A rough low-pass delay filter of data array along last axis.

Parameters

- **data** – Data to be filtered along last axis
- **freqs** – Frequencies of the filter [GHz]
- **bl_len_ns** – The baseline length (see *gen_delay_filter()*).
- **delay_filter** – The pre-computed filter to use. A filter can be created on-the-fly by passing kwargs.
- ****kwargs** – Passed to *gen_delay_filter()*.

Returns

filt_data – Filtered data array (same shape as `data`).

hera_sim.utils.rough_fringe_filter

hera_sim.utils.**rough_fringe_filter**(*data: ndarray*, *lsts: ndarray | None = None*, *freqs: ndarray | None = None*, *ew_bl_len_ns: float | None = None*, *, *fringe_filter: ndarray | None = None*, ***kwargs*) → ndarray

A rough fringe rate filter of data along zeroth axis.

Parameters

- **data** – data to filter along zeroth axis
- **fringe_filter** – A pre-computed fringe-filter to use. Computed on the fly if not given.
- ****kwargs** – Passed to *gen_fringe_filter()* to compute the fringe filter on the fly (if necessary). If so, at least `lsts`, `freqs`, and `ew_bl_len_ns` are required.

Returns

filt_data – Filtered data (same shape as `data`).

hera_sim.utils.tanh_window

hera_sim.utils.**tanh_window**(*x*, *x_min=None*, *x_max=None*, *scale_low=1*, *scale_high=1*)

hera_sim.utils.wrap2pipi

hera_sim.utils.**wrap2pipi**(*a*)

Wrap values of an array to [-; +] modulo 2.

Parameters

a (*array_like*) – Array of values to be wrapped to [-; +].

Returns

res (*array_like*) – Array of 'a' values wrapped to [-; +].

hera_sim.cli_utils

Useful helper functions and argparsers for running simulations via CLI.

Functions

get_filing_params(config)	Extract filing parameters from a configuration dictionary.
validate_config(config)	Validate the contents of a loaded configuration file.
write_calfits(gains, filename[, sim, freqs, ...])	Write gains to disk as a calfits file.

hera_sim.cli_utils.get_filing_params

hera_sim.cli_utils.**get_filing_params**(*config: dict*)

Extract filing parameters from a configuration dictionary.

Parameters
config – The full configuration dict.

Returns
dict – Filing parameter from the config, with default entries filled in.

Raises
`ValueError` – If `output_format` not in "miriad", "uvfits", or "uvh5".

hera_sim.cli_utils.validate_config

hera_sim.cli_utils.**validate_config**(*config: dict*)

Validate the contents of a loaded configuration file.

Parameters
config – The full configuration dict.

Raises
`ValueError` – If either insufficient information is provided, or the info is not valid.

hera_sim.cli_utils.write_calfits

hera_sim.cli_utils.**write_calfits**(*gains*, *filename*, *sim=None*, *freqs=None*, *times=None*, *x_orientation='north'*, *clobber=False*)

Write gains to disk as a calfits file.

Parameters

- **gains** (*dict*) – Dictionary mapping antenna numbers or (ant, pol) tuples to gains. Gains may either be spectra or waterfalls.
- **filename** (*str*) – Name of file, including desired extension.
- **sim** (`pyuvdata.UVData` instance or *Simulator* instance) – Object containing metadata pertaining to the gains to be saved. Does not need to be provided if both `freqs` and `times` are provided.

- **freqs** (*array-like of float*) – Frequencies corresponding to gains, in Hz. Does not need to be provided if `sim` is provided.
- **times** (*array-like of float*) – Times corresponding to gains, in JD. Does not need to be provided if `sim` is provided.
- **x_orientation** (*str, optional*) – Cardinal direction that the x-direction corresponds to. Defaults to the HERA configuration of north.
- **clobber** (*bool, optional*) – Whether to overwrite existing file in the case of a name conflict. Default is to *not* overwrite conflicting files.

hera_sim.components

A module providing discoverability features for hera_sim.

Functions

component(cls)	Decorator to create a new *SimulationComponent* that tracks its models.
get_all_components([with_aliases])	Get a dictionary of component names mapping to a dictionary of models.
get_all_models([with_aliases])	Get a dictionary of model names mapping to their classes for all possible models.
get_model(mdl[, cmp])	Get a particular model, based on its name.
get_models(cmp[, with_aliases])	Get a dict of model names mapping to model classes for a particular component.
list_all_components([with_aliases])	Lists all discoverable components.

hera_sim.components.component

hera_sim.components.**component**(*cls*)

Decorator to create a new *SimulationComponent* that tracks its models.

hera_sim.components.get_all_components

hera_sim.components.**get_all_components**(*with_aliases=False*) → dict[str, dict[str, *hera_sim.components.SimulationComponent*]]

Get a dictionary of component names mapping to a dictionary of models.

hera_sim.components.get_all_models

hera_sim.components.**get_all_models**(*with_aliases: bool = False*) → dict[str, *hera_sim.components.SimulationComponent*]

Get a dictionary of model names mapping to their classes for all possible models.

See also:

get_models()
Return a similar dictionary but filtered to a single kind of component.

hera_sim.components.get_model

hera_sim.components.**get_model**(*mdl: str*, *cmp: str | None = None*) → type[*hera_sim.components.SimulationComponent*]

Get a particular model, based on its name.

Parameters

- **mdl** – The name (or alias) of the model to get.
- **cmp** – If desired, limit the search to a specific component name. This helps if there are name clashes between models.

Returns
cmp – The *SimulationComponent* corresponding to the desired model.

hera_sim.components.get_models

hera_sim.components.**get_models**(*cmp: str*, *with_aliases: bool = False*) → dict[str, *hera_sim.components.SimulationComponent*]

Get a dict of model names mapping to model classes for a particular component.

hera_sim.components.list_all_components

hera_sim.components.**list_all_components**(*with_aliases: bool = True*) → str

Lists all discoverable components.

Parameters
with_aliases – If True, also include model aliases in the output.

Returns
str – A string summary of the available models.

Classes

SimulationComponent(**kwargs)	Base class for defining simulation component models.

hera_sim.components.SimulationComponent

class hera_sim.components.**SimulationComponent**(***kwargs*)

Base class for defining simulation component models.

This class serves two main purposes:

- Provide a simple interface for discovering simulation component models (see `list_discoverable_components()`).
- Ensure that each subclass can create abstract methods.

The *component()*: class decorator provides a simple way of accomplishing the above, while also providing some useful extra features.

Variables

- **is_multiplicative** (*bool*) – Specifies whether the model `cls` is a multiplicative effect. This parameter lets the *Simulator* class determine how to apply the effect simulated by `cls`. Default setting is False (i.e. the model is assumed to be additive unless specified otherwise).
- **return_type** (*str | None*) – Whether the returned result is per-antenna, per-baseline, or the full data array. This tells the *Simulator* how it should handle the returned result.
- **attrs_to_pull** (*dict*) – Dictionary mapping parameter names to *Simulator* attributes to be retrieved when setting up for simulation.

Methods

__call__(**kwargs)	Compute the component model.
get_aliases()	Get all the aliases by which this model can be identified.
get_model(mdl)	Get a model with a particular name (including aliases).
get_models([with_aliases])	Get a dictionary of models associated with this component.

hera_sim.components.SimulationComponent.__call__

abstract SimulationComponent.**__call__**(***kwargs*)

Compute the component model.

hera_sim.components.SimulationComponent.get_aliases

classmethod SimulationComponent.**get_aliases**() → tuple[str]

Get all the aliases by which this model can be identified.

hera_sim.components.SimulationComponent.get_model

classmethod SimulationComponent.**get_model**(*mdl: str*) → *SimulationComponent*

Get a model with a particular name (including aliases).

hera_sim.components.SimulationComponent.get_models

classmethod SimulationComponent.**get_models**(*with_aliases=False*) → dict[str, *hera_sim.components.SimulationComponent*]

Get a dictionary of models associated with this component.

Attributes

attrs_to_pull	Mapping between parameter names and Simulator attributes
is_multiplicative	Whether this systematic multiplies existing visibilities
is_randomized	Whether this systematic contains a randomized component
return_type	Whether the returned value is per-antenna, per-baseline, or the full array

hera_sim.components.SimulationComponent.attrs_to_pull

SimulationComponent.**attrs_to_pull: dict = {}**

Mapping between parameter names and Simulator attributes

hera_sim.components.SimulationComponent.is_multiplicative

SimulationComponent.**is_multiplicative: bool = False**

Whether this systematic multiplies existing visibilities

hera_sim.components.SimulationComponent.is_randomized

SimulationComponent.**is_randomized**: bool = False

Whether this systematic contains a randomized component

hera_sim.components.SimulationComponent.return_type

SimulationComponent.**return_type**: str | None = None

Whether the returned value is per-antenna, per-baseline, or the full array

5.2.2 Visibility Simulators

Simulation Framework

hera_sim.visibilities.simulators	Module defining a high-level visibility simulator wrapper.

hera_sim.visibilities.simulators

Module defining a high-level visibility simulator wrapper.

Functions

load_simulator_from_yaml(config)	Construct a visibility simulator from a YAML file.

hera_sim.visibilities.simulators.load_simulator_from_yaml

hera_sim.visibilities.simulators.**load_simulator_from_yaml**(*config: Path | str*) → *VisibilitySimulator*

Construct a visibility simulator from a YAML file.

Classes

ModelData(*, uvdata, sky_model[, beam_ids, ...])	An object containing all the information required to perform visibility simulation.
VisibilitySimulation(data_model, simulator)	An object representing a visibility simulation, including data and simulator.
VisibilitySimulator()	Base class for all hera_sim-compatible visibility simulators.

hera_sim.visibilities.simulators.ModelData

class hera_sim.visibilities.simulators.**ModelData**(*, *uvdata: UVData | str | Path*, *sky_model: SkyModel*, *beam_ids: dict[str, int] | Sequence[int] | None = None*, *beams: BeamList | list[Union[pyuvsim.analyticbeam.AnalyticBeam, pyuvdata.uvbeam.uvbeam.UVBeam]] | None = None*, *normalize_beams: bool = False*)

An object containing all the information required to perform visibility simulation.

Parameters

- **uvdata** – A `pyuvdata.UVData` object contain information about the "observation". If a path, must point to a UVData-readable file.
- **sky_model** – A model for the sky to simulate.
- **beams** – UVBeam models for as many antennae as have unique beams. Initialized from *obsparams*, if included. Defaults to a single uniform beam which is applied for every antenna. Each beam is the response of an individual antenna and NOT a per-baseline response. Shape=(N_BEAMS,).
- **beam_ids** – List of integers specifying which beam model each antenna uses (i.e. the index of *beams* which it should refer to). Also accepts a dictionary in the format used by pyuvsim (i.e. `antenna_name: index`), which is converted to such a list. By default, if one beam is given all antennas use the same beam, whereas if a beam is given per antenna, they are used in their given order. Shape=(N_ANTS,).
- **normalize_beams** – Whether to peak-normalize the beams. This removes the bandpass from the beams' data arrays and moves it into their `bandpass_array` attributes.

Notes

Input beam models represent the responses of individual antennas and are NOT the same as per-baseline "primary beams". This interpretation of a "primary beam" would be the product of the responses of two input antenna beams.

Methods

from_config(config_file[, normalize_beams])	Initialize the *ModelData* from a pyuvsim-compatible config.
write_config_file(filename[, direc, ...])	Writes a YAML config file corresponding to the current UVData object.

hera_sim.visibilities.simulators.ModelData.from_config

classmethod ModelData.**from_config**(*config_file: str | Path*, *normalize_beams: bool = False*) → *ModelData*

Initialize the *ModelData* from a pyuvsim-compatible config.

hera_sim.visibilities.simulators.ModelData.write_config_file

ModelData.**write_config_file**(*filename*, *direc='.'*, *beam_filepath=None*, *antenna_layout_path=None*)

Writes a YAML config file corresponding to the current UVData object.

Parameters

- **filename** (*str*) – Filename of the config file.
- **direc** (*str*) – Directory in which to place the config file and its supporting files.
- **beam_filepath** (*str, optional*) – Where to put the beam information. Default is to place it alongside the config file, but with extension '.beams'.
- **antenna_layout_path** (*str, optional*) – Where to put the antenna layout CSV file. Default is alongside the main config file, but appended with '_antenna_layout.csv'.

Attributes

freqs	Frequnecies at which data is defined.
lsts	Local Sidereal Times in radians.
n_beams	Number of beam models used.

hera_sim.visibilities.simulators.ModelData.freqs

ModelData.**freqs**

Frequnecies at which data is defined.

hera_sim.visibilities.simulators.ModelData.lsts

ModelData.**lsts**

Local Sidereal Times in radians.

hera_sim.visibilities.simulators.ModelData.n_beams

ModelData.**n_beams**

Number of beam models used.

hera_sim.visibilities.simulators.VisibilitySimulation

class hera_sim.visibilities.simulators.**VisibilitySimulation**(*data_model:* ModelData, *simulator:* VisibilitySimulator, *n_side:* *int* = *32*)

An object representing a visibility simulation, including data and simulator.

Methods

simulate()	Perform the visibility simulation.

hera_sim.visibilities.simulators.VisibilitySimulation.simulate

VisibilitySimulation.**simulate**()

Perform the visibility simulation.

Attributes

n_side	
uvdata	A simple view into the UVData object in the *data_model*.
data_model	
simulator	

hera_sim.visibilities.simulators.VisibilitySimulation.n_side

VisibilitySimulation.**n_side**: int = 32

hera_sim.visibilities.simulators.VisibilitySimulation.uvdata

property VisibilitySimulation.**uvdata**: UVData

A simple view into the UVData object in the *data_model*.

hera_sim.visibilities.simulators.VisibilitySimulation.data_model

VisibilitySimulation.**data_model**: *ModelData*

hera_sim.visibilities.simulators.VisibilitySimulation.simulator

VisibilitySimulation.**simulator:** *VisibilitySimulator*

hera_sim.visibilities.simulators.VisibilitySimulator

class hera_sim.visibilities.simulators.**VisibilitySimulator**

Base class for all hera_sim-compatible visibility simulators.

To define a new simulator, make a subclass. The subclass should overwrite available class-attributes as necessary, and specify a `__version__` of the simulator code itself.

The *simulate()* abstract method *must* be overwritten in the subclass, to perform the actual simulation. The *validate()* method *may* also be overwritten to validate the given *UVData* input for the particular simulator.

The subclass may define any number of simulator-specific parameters as part of its init method.

Finally, to enable constructing the simulator in command-line applications, a *from_yaml()* method is provided. This will load a YAML file's contents as a dictionary, and then instantiate the subclass with the parameters in that dict. To enable some control over this process, the subclass can overwrite the `_from_yaml_dict()` private method, which takes in the dictionary read from the YAML file, and transforms any necessary parameters before constructing the class. For example, if the class required a set of data from a file, the YAML might contain the filename itself, and in `_from_yaml_dict()`, the file would be read and the data itself passed to the constructor.

Methods

compress_data_model(data_model)	Temporarily delete/remove data from the model to reduce memory usage.
estimate_memory(data_model)	Estimate the memory usage of the simulator in GB.
from_yaml(yaml_config)	Generate the simulator from a YAML file or dictionary.
restore_data_model(data_model)	Restore data from the model removed by *compress_data_model()*.
simulate(data_model)	Simulate the visibilities.
validate(data_model)	Check that the data model complies with the assumptions of the simulator.

hera_sim.visibilities.simulators.VisibilitySimulator.compress_data_model

VisibilitySimulator.**compress_data_model**(*data_model*)

Temporarily delete/remove data from the model to reduce memory usage.

Anything that is removed here should be restored after the simulation.

hera_sim.visibilities.simulators.VisibilitySimulator.estimate_memory

VisibilitySimulator.**estimate_memory**(*data_model:* ModelData) → float

Estimate the memory usage of the simulator in GB.

This is used to estimate the amount of memory needed to run the simulator.

Note: the default method is very much a lower bound – just the size of the output visibilities. Each individual simulator may or may not implement a more accurate estimate.

hera_sim.visibilities.simulators.VisibilitySimulator.from_yaml

classmethod VisibilitySimulator.**from_yaml**(*yaml_config: dict | str | Path*) → *VisibilitySimulator*

Generate the simulator from a YAML file or dictionary.

hera_sim.visibilities.simulators.VisibilitySimulator.restore_data_model

VisibilitySimulator.**restore_data_model**(*data_model*)

Restore data from the model removed by *compress_data_model()*.

hera_sim.visibilities.simulators.VisibilitySimulator.simulate

abstract VisibilitySimulator.**simulate**(*data_model:* ModelData) → ndarray

Simulate the visibilities.

hera_sim.visibilities.simulators.VisibilitySimulator.validate

VisibilitySimulator.**validate**(*data_model:* ModelData)

Check that the data model complies with the assumptions of the simulator.

Attributes

diffuse_ability	Whether this particular simulator has the ability to simulate diffuse maps directly.
point_source_ability	Whether this particular simulator has the ability to simulate point sources directly.

hera_sim.visibilities.simulators.VisibilitySimulator.diffuse_ability

VisibilitySimulator.**diffuse_ability = False**

Whether this particular simulator has the ability to simulate diffuse maps directly.

hera_sim.visibilities.simulators.VisibilitySimulator.point_source_ability

VisibilitySimulator.**point_source_ability = True**

Whether this particular simulator has the ability to simulate point sources directly.

Built-In Simulators

hera_sim.visibilities.pyuvsim_wrapper.UVSim([quiet])	A wrapper around the pyuvsim simulator.

hera_sim.visibilities.pyuvsim_wrapper.UVSim

class hera_sim.visibilities.pyuvsim_wrapper.**UVSim**(*quiet: bool = False*)

A wrapper around the pyuvsim simulator.

Parameters

quiet – If True, don't print anything.

Methods

compress_data_model(data_model)	Temporarily delete/remove data from the model to reduce memory usage.
estimate_memory(data_model)	Estimate the memory usage of the simulator in GB.
from_yaml(yaml_config)	Generate the simulator from a YAML file or dictionary.
restore_data_model(data_model)	Restore data from the model removed by *compress_data_model()*.
simulate(data_model)	Simulate the visibilities.
validate(data_model)	Check that the data model complies with the assumptions of the simulator.

hera_sim.visibilities.pyuvsim_wrapper.UVSim.compress_data_model

UVSim.**compress_data_model**(*data_model*)

Temporarily delete/remove data from the model to reduce memory usage.

Anything that is removed here should be restored after the simulation.

hera_sim.visibilities.pyuvsim_wrapper.UVSim.estimate_memory

UVSim.**estimate_memory**(*data_model:* ModelData) → float

Estimate the memory usage of the simulator in GB.

This is used to estimate the amount of memory needed to run the simulator.

Note: the default method is very much a lower bound – just the size of the output visibilities. Each individual simulator may or may not implement a more accurate estimate.

hera_sim.visibilities.pyuvsim_wrapper.UVSim.from_yaml

classmethod UVSim.**from_yaml**(*yaml_config: dict | str | Path*) → *VisibilitySimulator*

Generate the simulator from a YAML file or dictionary.

hera_sim.visibilities.pyuvsim_wrapper.UVSim.restore_data_model

UVSim.**restore_data_model**(*data_model*)

Restore data from the model removed by *compress_data_model()*.

hera_sim.visibilities.pyuvsim_wrapper.UVSim.simulate

UVSim.**simulate**(*data_model:* ModelData)

Simulate the visibilities.

hera_sim.visibilities.pyuvsim_wrapper.UVSim.validate

UVSim.**validate**(*data_model:* ModelData)

Check that the data model complies with the assumptions of the simulator.

Attributes

diffuse_ability	Whether this particular simulator has the ability to simulate diffuse maps directly.
point_source_ability	Whether this particular simulator has the ability to simulate point sources directly.

hera_sim.visibilities.pyuvsim_wrapper.UVSim.diffuse_ability

UVSim.**diffuse_ability = False**

Whether this particular simulator has the ability to simulate diffuse maps directly.

hera_sim.visibilities.pyuvsim_wrapper.UVSim.point_source_ability

UVSim.**point_source_ability = True**

Whether this particular simulator has the ability to simulate point sources directly.

5.3 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

5.3.1 Bug reports

When reporting a bug please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

5.3.2 Documentation improvements

`hera_sim` could always use more documentation, whether as part of the official `hera_sim` docs or in docstrings.

5.3.3 Feature requests and feedback

The best way to send feedback is to file an issue at https://github.com/HERA-Team/hera_sim/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that code contributions are welcome :)

5.3.4 Development

To set up `hera_sim` for local development:

1. If you are *not* on the HERA-Team collaboration, make a fork of hera_sim (look for the "Fork" button).
2. Clone the repository locally. If you made a fork in step 1:

   ```
   git clone git@github.com:your_name_here/hera_sim.git
   ```

   Otherwise:

```
git clone git@github.com:HERA-Team/hera_sim.git
```

3. Create a branch for local development (you will *not* be able to push to "master"):

```
git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

4. Make a development environment. We highly recommend using conda for this. With conda, just run:

```
conda env create -n hera python=3
pip install -e .[dev]
pre-commit install
```

5. Commit your changes and push your branch to GitHub:

```
git add .
git commit -m "Your detailed description of your changes."
git push origin name-of-your-bugfix-or-feature
```

6. Submit a pull request through the GitHub website.

Pull Request Guidelines

If you need some code review or feedback while you're developing the code just make the pull request.

For merging, you should:

1. Include passing tests (run `pytest`)
2. Update documentation when there's new API, functionality etc.
3. Add a note to `CHANGELOG.rst` about the changes.
4. Add yourself to `AUTHORS.rst`.

5.4 Developing `hera_sim`

All docstrings should be written in Numpy docstring format.

5.5 Authors

- HERA-Team - https://github.com/HERA-Team

5.6 Changelog

5.6.1 dev

Added

- Classes subclassed from `SimulationComponent` now have a `is_randomized` class attribute that informs the `Simulator` of whether it should provide a `BitGenerator` to the class when simulating the component. - Classes which use a random component should now have a `rng` attribute,

  which should be treated in the same manner as other model parameters. In other words, random states are now effectively treated as model parameters.

- New simulator class `FFTVis` that uses the `fftvis` package to simulate visibilities. This is a CPU-based visibility simulator that is faster than `MatVis` for large, compact arrays.

Changed

- All random number generation now uses the new `numpy` API. - Rather than seed the global random state, a new `BitGenerator` is made

  with whatever random seed is desired.

  - The Simulator API has remained virtually unchanged, but the internal logic that handles random state management has received a significant update.

Deprecated

- Support for Python 3.9 has been dropped.

Fixed

- API calls for pyuvdata v2.4.0.

5.6.2 v4.1.0 [2023.06.26]

This release heavily focuses on performance of the visibility simulators, and in particular the `VisCPU` simulator.

Fixed

- Passing `spline_interp_opts` now correctly pipes these options through to the visibility simulators.

Added

- New `_blt_order_kws` class-attribute for `VisibilitySimulator` subclasses, that can be used to create the mock metadata in an order corresponding to that required by the simulator (instead of re-ordering after data creation, which can take some time).
- New optional `compress_data_model()` method on `VisibilitySimulator` subclasses that allows unnecessary metadata in the `UVData` object to be dropped before simulation (can be restored afterwards with the associated `restore_data_model()`). This can reduce peak memory usage.
- New `check_antenna_conjugation` parameter for the `VisCPU` simulator, to allow turning off checks for antenna conjugation, which takes time and is unnecessary for mock datasets.
- Dependency on `hera-cli-utils` which adds options like `--log-level` and `--profile` to `hera-sim-vis.py`.
- Option to use a taper in generating a bandpass.
- `utils.tanh_window` function for generating a two-sided tanh window.
- `interpolators.Reflection` class for building a complex reflection coefficient interpolator from a `npz` archive.
- Reflection coefficient and beam integral `npz` archives for the phase 1 and phase 4 systems (i.e., dipole feed and Vivaldi feed).

Changed

- `run_check_acceptability` is now `False` by default when constructing simulations from obsparams configuration files, to improve performance.
- For `VisCPU` simulator, we no longer copy the whole data array when simulating, but instead just fill the existing one, to save on peak RAM.
- Made `VisCPU._reorder_vis()` much faster (like 99% time reduction).
- The `--compress` option to `hera-sim-vis.py` is no longer a boolean flag but takes a file argument. This file will be written as a cache of the baseline-time indices required to keep when compressing by redundancy.

5.6.3 v4.0.0 [2023.05.22]

Breaking Changes

- Removed the `HealVis` wrapper. Use `pyuvsim` instead.

Changed

- Updated package to always use future array shapes for `pyuvdata` objects (this includes updates to `PolyBeam` and `Simulator` objects amongst others).

5.6.4 v3.1.1 [2023.02.23]

Changed

- Coupling matrix calculation in *MutualCoupling* has been updated to correctly calculate the coupling coefficients from the provided E-field beam.

5.6.5 v3.1.0 [2023.01.17]

Added

- *MutualCoupling* class that simulates the systematic described in Josaitis et al. 2021.
- **New class attributes for the *SimulationComponent* class:**
  - `return_type` specifies what type of return value to expect;
  - `attrs_to_pull` specifies which `Simulator` attributes to use.
- Some helper functions for *MutualCoupling* matrix multiplications.
- More attributes from the underlying `UVData` object exposed to the *Simulator*.

Changed

- `Simulator._update_args` logic has been improved.
- *Simulator* attributes `lsts`, `times`, and `freqs` are no longer cached.

5.6.6 v3.0.0

Removed

- Finally removed ability to set `use_pixel_beams` and `bm_pix` on the VisCPU simulator. This was removed in v1.0.0 of `vis_cpu`.
- Official support for py37.

Internals

- Added isort and pyupgrade pre-commit hooks for cleaner code.

5.6.7 v2.3.4 [2022.06.08]

Added

- `NotImplementedError` raised when trying to simulate noise using an interpolated sky temperature and phase-wrapped LSTs.
- More comparison tests of pyuvsim wrapper.

Fixed

- Inferred integration time in `ThermalNoise` when phase-wrapped LSTs are used.
- Added `**kwargs` to `PolyBeam.interp` method to match UVBeam.
- healvis wrapper properly sets cross-pol visibilities to zero.

Changed

- Temporarily forced all UVData objects in the code to use current array shapes.

5.6.8 v2.3.3 [2022.02.21]

Added

- `adjustment.interpolate_to_reference` now supports interpolating in time when there is a phase wrap in LST.

Changed

- Some logical statements in `adjustment.interpolate_to_reference` were changed to use binary operators on logical arrays instead of e.g. `np.logical_or`.

5.6.9 v2.3.2 [2022.02.18]

Added

- `_extract_kwargs` attribute added to the `SimulationComponent` class. This attribute is used by the `Simulator` to determine which optional parameters should actually be extracted from the data.
- `antpair` optional parameter added to the `ThermalNoise` class. This is used to determine whether to simulate noise via the radiometer equation (as is appropriate for a cross-correlation) or to just add a bias from the receiver temperature (which is our proxy for what should happen to an auto-correlation).

Fixed

- The `Simulator` class now correctly uses the auto-correlations to simulate noise for the cross-correlations.

5.6.10 v2.3.1 [2022.01.19]

Fixed

- Using the `normalize_beams` option is now possible with the `from_config` class method.

5.6.11 v2.3.0 [2022.01.19]

Added

- `normalize_beams` option in `ModelData` class. Setting this parameter to `True` enforces peak-normalization on all of the beams used in the simulation. The default behavior is to not peak-normalize the beams.

5.6.12 v2.2.1 [2022.01.14]

Added

- `OverAirCrossCoupling` now has a parameter `amp_norm`. This lets the user decide at what distance from the receiverator the gain of the emitted signal is equal to the base amplitude.

Fixed

- `OverAirCrossCoupling` now only simulates the systematic for cross-correlations.
- `ReflectionSpectrum` class had its `is_multiplicative` attribute set to True.

5.6.13 v2.2.0 [2022.01.13]

Added

- New `ReflectionSpectrum` class to generate multiple reflections over a specified range of delays/amplitudes.

Fixed

- Corrected some parameter initializations in `sigchain` module.

5.6.14 v2.1.0 [2022.01.12]

Added

- New `OverAirCrossCoupling` class to better model crosstalk in H1C data.

Changed

- Slightly modified `Simulator` logic for automatically choosing parameter values. This extends the number of cases the class can handle, but will be changed in a future update.

5.6.15 v2.0.0 [2021.11.16]

Added

- New VisibilitySimulator interface. See the `<https://hera-sim.readthedocs.io/en/latest/tutorials/visibility_simulator.html> Visibility Simulator Tutorial`_ for details. This is a breaking change for usage of the visibility simulators, and includes more robust handling of polarization, fixed ordering of data when put back into the `UVData` objects, more native support for using `pyradiosky` to define the sky model, and improved support for `vis_cpu`.
- Interface directly to the `pyuvsim` simulation engine.
- Ability to load tutorial data from the installed package.
- New and refactored tests for visibility simulations.

Fixed

- default `feed_array` for `PolyBeam` fixed.

Changed

- Updated tutorial for the visibility simulator interface (see above link).
- `vis_cpu` made an optional extra
- removed the `conversions` module, which is now in the `vis_cpu` package.
- Can now properly use `pyuvdata>=2.2.0`.

5.6.16 v1.1.1 [2021.08.21]

Added

- Add a Zernike polynomial beam model.

5.6.17 v1.1.0 [2021.08.04]

Added

- Enable polarization support for `vis_cpu` (handles polarized primary beams, but only Stokes I sky model so far)
- Add a polarized version of the analytic PolyBeam model.

5.6.18 v1.0.2 [2021.07.01]

Fixed

- Bug in retrieval of unique LSTs by *Simulator* when a blt-order other than time-baseline is used has been fixed. LSTs should now be correctly retrieved.
- *empty_uvdata()* now sets the `phase_type` attribute to "drift".

5.6.19 v1.0.1 [2021.06.30]

Added

Fixed

- Discrepancy in *PointSourceForeground* documentation and actual implementation has been resolved. Simulated foregrounds now look reasonable.

Changed

- The time parameters used for generating an example `Simulator` instance in the tutorial have been updated to match their description.
- *Simulator* tutorial has been changed slightly to account for the foreground fix.

5.6.20 v1.0.0 [2021.06.16]

Added

- **`adjustment` module from HERA Phase 1 Validation work**
  - **`adjust_to_reference()`**
    - High-level interface for making one set of data comply with another set of data. This may involve rephasing or interpolating in time and/or interpolating in frequency. In the case of a mismatch between the two array layouts, this algorithm will select a subset of antennas to provide the greatest number of unique baselines that remain in the downselected array.
  - All other functions in this module exist only to modularize the above function.
- *cli_utils* module providing utility functions for the CLI simulation script.
- ***components* module providing an abstract base class for simulation components.**
  - Any new simulation components should be subclassed from the *SimulationComponent* ABC. New simulation components subclassed appropriately are automatically discoverable by the *Simulator* class. A MWE for subclassing new components is as follows:

    ```
    @component
    class Component:
        pass

    class Model(Component):
        ...
    ```

    The `Component` base class tracks any models subclassed from it and makes it discoverable to the *Simulator*.
- New "season" configuration (called `"debug"`), intended to be used for debugging the *Simulator* when making changes that might not be easily tested.
- ***chunk_sim_and_save()* function from HERA Phase 1 Validation work**
  - This function allows the user to write a `pyuvdata.UVData` object to disk in chunks of some set number of integrations per file (either specified directly, or specified implicitly by providing a list of reference

files). This is very useful for taking a large simulation and writing it to disk in a way that mimics how the correlator writes files to disk.

- Ability to generate noise visibilities based on autocorrelations from the data. This is achieved by providing a value for the `autovis` parameter in the `thermal_noise` function (see *ThermalNoise*).
- The *vary_gains_in_time()* provides an interface for taking a gain spectrum and applying time variation (linear, sinusoidal, or noiselike) to any of the reflection coefficient parameters (amplitude, phase, or delay).
- The *CrossCouplingSpectrum* provides an interface for generating multiple realizations of the cross-coupling systematic spaced logarithmically in amplitude and linearly in delay. This is ported over from the Validation work.

Fixed

- The reionization signal produced by `eor.noiselike_eor` is now guaranteed to be real-valued for autocorrelations (although the statistics of the EoR signal for the autocorrelations still need to be investigated for correctness).

Changed

- **API BREAKING CHANGES**
  - All functions that take frequencies and LSTs as arguments have had their signatures changed to `func(lsts, freqs, *args, **kwargs)`.
  - Functions that employ *rough_fringe_filter()* or *rough_delay_filter()* as part of the visibility calculation now have parameters `delay_filter_kwargs` and/or `fringe_filter_kwargs`, which are dictionaries that are ultimately passed to the filtering functions. `foregrounds.diffuse_foreground` and `eor.noiselike_eor` are both affected by this.
  - Some parameters have been renamed to enable simpler handling of package-wide defaults. Parameters that have been changed are:
    - `filter_type` -> `delay_filter_type` in *gen_delay_filter()*
    - `filter_type` -> `fringe_filter_type` in *gen_fringe_filter()*
    - `chance` -> `impulse_chance` in `rfi_impulse` (see *Impulse*)
    - `strength` -> `impulse_strength` in `rfi_impulse` (see *Impulse*)
    - Similar changes were made in `rfi_dtv` (*DTV*) and `rfi_scatter` (*Scatter*).
  - Any occurrence of the parameter `fqs` has been replaced with `freqs`.
  - The `noise.jy2T` function was moved to *utils* and renamed. See *jansky_to_kelvin()*.
  - The parameter `fq0` has been renamed to `f0` in *RfiStation*.
  - The `_listify` function has been moved from *rfi* to *utils*.
  - `sigchain.HERA_NRAO_BANDPASS` no longer exists in the code, but may be loaded from the file `HERA_H1C_BANDPASS.npy` in the `data` directory.
- **Other Changes**
  - The *Simulator* has undergone many changes that make the class much easier to use, while also providing a handful of extra features. The new *Simulator* provides the following features:
    - A universal *add()* method for applying any of the effects implemented in `hera_sim`, as well as any custom effects defined by the user.
    - A *get()* method that retrieves any previously simulated effect.

    - The option to apply a simulated effect to only a subset of antennas, baselines, and/or polarizations, accessed through using the `vis_filter` parameter.
    - Multiple modes of seeding the random state to achieve a higher degree of realism than previously available.
    - The *calculate_filters()* method pre-calculates the fringe-rate and delay filters for the entire array and caches the result. This provides a marginal-to-modest speedup for small arrays, but can provide a significant speedup for very large arrays. Benchmarking results TBD.
    - An instance of the *Simulator* may be generated with an empty call to the class if any of the season defaults are active (or if the user has provided some other sufficiently complete set of default settings).
    - Some of the methods for interacting with the underlying `pyuvdata.UVData` object have been exposed to the *Simulator* (e.g. `get_data`).
    - An easy reference to the *chunk_sim_and_save()* function.
  - *foregrounds*, *eor*, *noise*, *rfi*, *antpos*, and *sigchain* have been modified to implement the features using callable classes. The old functions still exist for backwards-compatibility, but moving forward any additions to visibility or systematics simulators should be implemented using callable classes and be appropriately subclassed from *SimulationComponent*.
  - *empty_uvdata()* has had almost all of its parameter values set to default as `None`. Additionally, the `n_freq`, `n_times`, `antennas` parameters are being deprecated and will be removed in a future release.
  - *white_noise()* is being deprecated. This function has been moved to the utility module and can be found at *gen_white_noise()*.

5.6.21 v0.4.0 [2021.05.01]

Added

- **New features added to `vis_cpu`**
  - **Analytic beam interpolation**
    - Instead of gridding the beam and interpolating the grid using splines, the beam can be interpolated directly by calling its `interp` method.
    - The user specifies this by passing `use_pixel_beams=False` to `vis_cpu`.
  - **A simple MPI parallelization scheme**
    - Simulation scripts may be run using `mpirun/mpiexec`
    - The user imports `mpi4py` into their script and passes `mpi_comm=MPI.COMM_WORLD` to vis_cpu
  - **New `PolyBeam` and `PerturbedPolyBeam` analytic beams (classes)**
    - Derived from `pyuvsim.Analytic beam`
    - Based on axisymmetric Chebyshev polynomial fits to the Fagnoni beam.
    - PerturbedPolyBeam is capable of expressing a range of non-redundancy effects, including per-beam stretch factors, perturbed sidelobes, and ellipticity/rotation.

5.6.22 v0.3.0 [2019.12.10]

Added

- **New sub-package `simulators`**
  - **`VisibilitySimulators` class**
    - Provides a common interface to interferometric visibility simulators. Users instantiate one of its subclasses and provide input antenna and sky scenarios.
    - `HealVis` subclass
    - Provides an interface to the `healvis` visibility simulator.
  - **`VisCPU` subclass**
    - Provides an interface to the `viscpu` visibility simulator.
  - **`conversions` module**
    - Not intended to be interfaced with by the end user; it provides useful coordinate transformations for `VisibilitySimulators`.

5.6.23 v0.2.0 [2019.11.20]

Added

- **Command-line Interface**
  - Use anywhere with `hera_sim run [options] INPUT`
  - Tutorial available on readthedocs
- **Enhancement of `run_sim` method of `Simulator` class**
  - **Allows for each simulation component to be returned**
    - Components returned as a list of 2-tuples `(model_name, visibility)`
    - Components returned by specifying `ret_vis=True` in their kwargs
- **Option to seed random number generators for various methods**
  - Available via the `Simulator.add_` methods by specifying the kwarg `seed_redundantly=True`
  - Seeds are stored in `Simulator` object, and may be saved as a `npy` file when using the `Simulator.write_data` method
- **New YAML tag `!antpos`**
  - Allows for antenna layouts to be constructed using `hera_sim.antpos` functions by specifying parameters in config file

Fixed

- Changelog formatting for v0.1.0 entry

Changed

- **Implementation of `defaults` module**
  - Allows for semantic organization of config files
  - **Parameters that have the same name take on the same value**
    - e.g. `std` in various `rfi` functions only has one value, even if it's specified multiple times

5.6.24 v0.1.0 [2019.08.28]

Added

- **New module `interpolators`**
  - **Classes intended to be interfaced with by end-users:**
    - **`Tsky`**
      - Provides an interface for generating a sky temperature interpolation object when provided with a `.npz` file and interpolation kwargs.
    - **`Beam`, `Bandpass`**
      - Provides an interface for generating either a `poly1d` or `interp1d` interpolation object when provided with an appropriate datafile.
- **New module `defaults`**
  - Provides an interface which allows the user to dynamically adjust default parameter settings for various `hera_sim` functions.
- **New module `__yaml_constructors`**
  - Not intended to be interfaced with by the end user; this module just provides a location for defining new YAML tags to be used in conjunction with the `defaults` module features and the `Simulator.run_sim` method.
- **New directory `config`**
  - Provides a location to store configuration files.

Fixed

Changed

- HERA-specific variables had their definitions removed from the codebase. Objects storing these variables still exist in the codebase, but their definitions now come from loading in data stored in various new files added to the `data` directory.

5.6.25 v0.0.1

- Initial released version

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

h

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W