hera_sim Documentation

Release 0.0.0

HERA-Team

May 28, 2020

CONTENTS

`hera_sim` is a simple simulator that generates instrumental effects and applies them to visibilities.

CHAPTER

ONE

CONTENTS

1.1 Installation

1.1.1 Requirements

Requires:

- `numpy`
- `scipy`
- `aipy`
- `pyuvdata`

Then, at the command line, navigate to the `hera_sim` repo/directory, and:

```
pip install .
```

If developing, from the top-level directory do:

```
pip install -e .
```

1.2 Tutorials and FAQs

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

1.2.1 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.")
```

(continues on next page)

(continued from previous page)

```
/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)
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)
```

(continues on next page)

(continued from previous page)

```
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:

1.2.2 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).

What follows is a quick tour of the main functionality this provides.

Setup

[1]:
```
%matplotlib inline
import os
import matplotlib.pyplot as plt
import numpy as np
import astropy.units as u

import hera_sim
import uvtools

from hera_sim.simulate import Simulator
from hera_sim.noise import HERA_Tsky_mdl
from hera_sim.data import DATA_PATH
from hera_sim.interpolators import Beam
```

```
/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.")
```

(continues on next page)

(continued from previous page)

```
/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)
```

The Simulator Class

The `Simulator` class holds everything required to generate and do basic analysis on a simulation. It can be instantiated by submitting any one of the following: a filename pointing to an existing simulation in uvfits, uvh5 or miriad format; a `UVData` object, or a set of keyword arguments which are passed to the `hera_sim.io.empty_uvdata` function. Any keyword arguments passed to the `Simulator` initializer are passed directly to the `hera_sim.io.empty_uvdata`, and those are in turn passed directly to the `pyuvsim.simsetup.initialize_uvdata_from_keywords` function, so that setting up a simulation object is relatively straightforward. The default parameter values for the `empty_uvdata` function are all set to `None`, but are given default values to show a suggested minimal set of parameters necessary to initialize a `UVData` object (and, in turn, a `Simulator` object):

```
hera_sim.io.empty_uvdata(
    Ntimes=None,
    start_time=None,
    integration_time=None,
    Nfreqs=None,
    start_freq=None,
    channel_width=None,
    array_layout=None,
    **kwargs,
)
```

If you are curious about the full set of parameters that may be used to initialize a `UVData` object, then please see the documentation for the `pyuvsim.simsetup.initialize_uvdata_from_keywords` function.

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 with an H1C-appropriate integration time of 10.7 seconds, and a 7-element hexagonal array.

[2]:
```
# first, generate the array. this is returned as a dict
# with antenna numbers as keys and ENU positions as values
ants = hera_sim.antpos.hex_array(2, split_core=False, outriggers=0)

# for clarity about some timing parameters
start_time = 2458115.9 # JD
integration_time = 10.7 # seconds
Ntimes = int(30 * u.min.to("s") / integration_time)
```

(continues on next page)

(continued from previous page)

```
sim = Simulator(
    Nfreqs=100,
    start_freq=1e8,
    bandwidth=1e8,
    Ntimes=Ntimes,
    start_time=start_time,
    integration_time=integration_time,
    array_layout=ants
)
```

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

You can retrieve the frequencies (in GHz) with the `freqs` attribute. (`sim.freqs`)
You can retrieve the LSTs (in radians) with the `lsts` attribute. (`sim.lsts`)
You can retrieve the antenna array with the `antpos` attribute. (`sim.antpos`)

[3]:
```
sim.freqs
```

[3]:
```
array([0.1  , 0.101, 0.102, 0.103, 0.104, 0.105, 0.106, 0.107, 0.108,
       0.109, 0.11 , 0.111, 0.112, 0.113, 0.114, 0.115, 0.116, 0.117,
       0.118, 0.119, 0.12 , 0.121, 0.122, 0.123, 0.124, 0.125, 0.126,
       0.127, 0.128, 0.129, 0.13 , 0.131, 0.132, 0.133, 0.134, 0.135,
       0.136, 0.137, 0.138, 0.139, 0.14 , 0.141, 0.142, 0.143, 0.144,
       0.145, 0.146, 0.147, 0.148, 0.149, 0.15 , 0.151, 0.152, 0.153,
       0.154, 0.155, 0.156, 0.157, 0.158, 0.159, 0.16 , 0.161, 0.162,
       0.163, 0.164, 0.165, 0.166, 0.167, 0.168, 0.169, 0.17 , 0.171,
       0.172, 0.173, 0.174, 0.175, 0.176, 0.177, 0.178, 0.179, 0.18 ,
       0.181, 0.182, 0.183, 0.184, 0.185, 0.186, 0.187, 0.188, 0.189,
       0.19 , 0.191, 0.192, 0.193, 0.194, 0.195, 0.196, 0.197, 0.198,
       0.199])
```

[4]:
```
sim.lsts
```

[4]:
```
array([4.58108965, 4.58186991, 4.58265017, 4.58343042, 4.58421068,
       4.58499094, 4.58577119, 4.58655145, 4.58733171, 4.58811196,
       4.58889222, 4.58967247, 4.59045273, 4.59123299, 4.59201324,
       4.5927935 , 4.59357376, 4.59435401, 4.59513427, 4.59591452,
       4.59669478, 4.59747504, 4.59825529, 4.59903555, 4.59981581,
       4.60059606, 4.60137632, 4.60215657, 4.60293683, 4.60371709,
       4.60449734, 4.6052776 , 4.60605786, 4.60683811, 4.60761837,
       4.60839863, 4.60917888, 4.60995914, 4.6107394 , 4.61151965,
       4.61229991, 4.61308017, 4.61386042, 4.61464068, 4.61542093,
       4.61620119, 4.61698145, 4.6177617 , 4.61854196, 4.61932222,
       4.62010247, 4.62088273, 4.62166299, 4.62244324, 4.6232235 ,
       4.62400375, 4.62478401, 4.62556427, 4.62634452, 4.62712478,
       4.62790503, 4.62868529, 4.62946555, 4.6302458 , 4.63102606,
       4.63180632, 4.63258657, 4.63336683, 4.63414709, 4.63492734,
       4.6357076 , 4.63648786, 4.63726811, 4.63804837, 4.63882863,
       4.63960888, 4.64038914, 4.64116939, 4.64194965, 4.64272991,
       4.64351016, 4.64429042, 4.64507068, 4.64585093, 4.64663119,
       4.64741145, 4.6481917 , 4.64897196, 4.64975221, 4.65053247,
       4.65131273, 4.65209298, 4.65287324, 4.65365349, 4.65443375,
       4.65521401, 4.65599426, 4.65677452, 4.65755478, 4.65833503,
```

(continues on next page)

(continued from previous page)

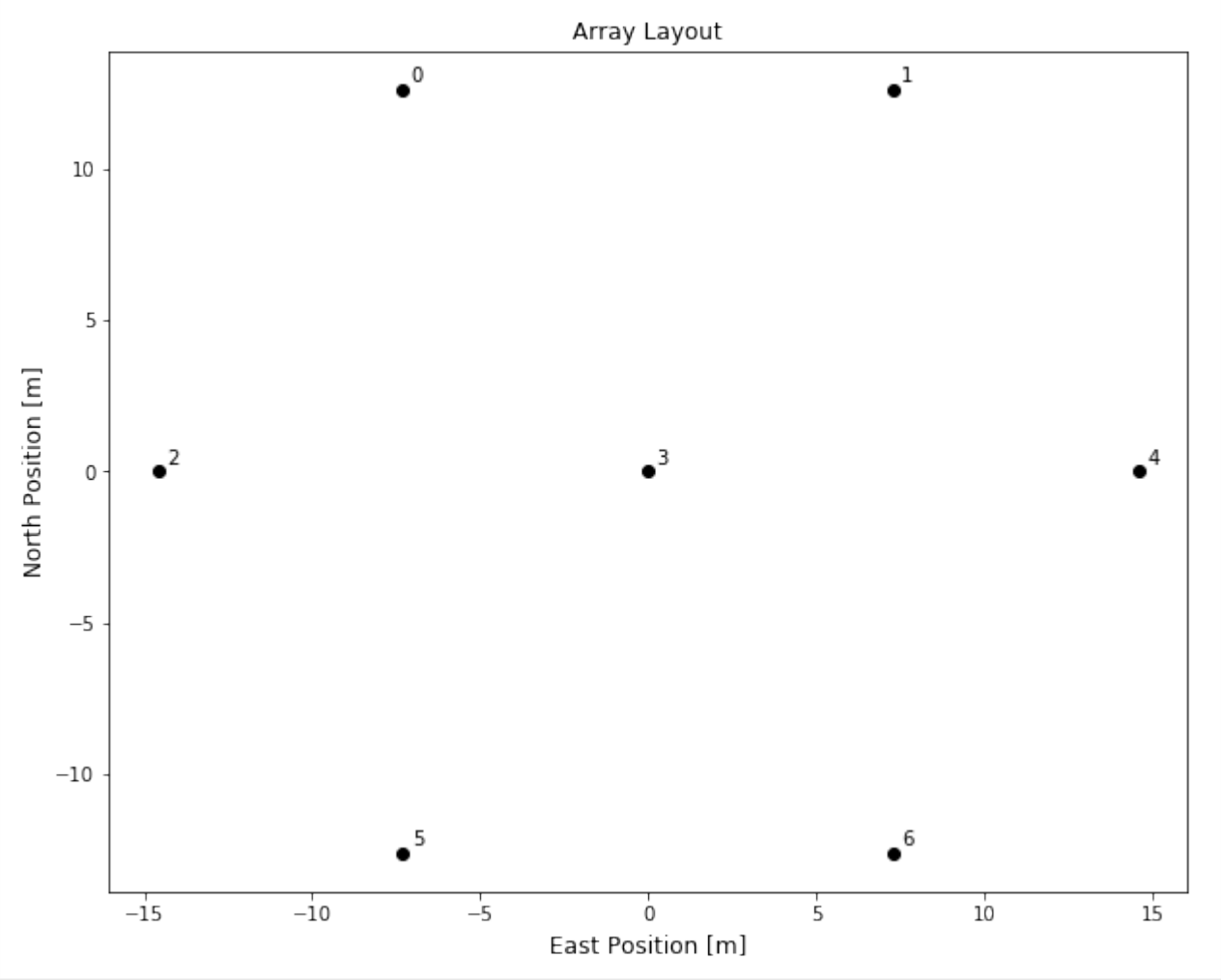
```
       4.65911529, 4.65989555, 4.6606758 , 4.66145606, 4.66223632,
       4.66301657, 4.66379683, 4.66457709, 4.66535734, 4.6661376 ,
       4.66691785, 4.66769811, 4.66847837, 4.66925862, 4.67003888,
       4.67081914, 4.67159939, 4.67237965, 4.67315991, 4.67394016,
       4.67472042, 4.67550068, 4.67628093, 4.67706119, 4.67784144,
       4.6786217 , 4.67940196, 4.68018221, 4.68096247, 4.68174272,
       4.68252298, 4.68330324, 4.68408349, 4.68486375, 4.68564401,
       4.68642426, 4.68720452, 4.68798478, 4.68876503, 4.68954529,
       4.69032555, 4.6911058 , 4.69188606, 4.69266631, 4.69344657,
       4.69422683, 4.69500708, 4.69578734, 4.6965676 , 4.69734785,
       4.69812811, 4.69890837, 4.69968862, 4.70046888, 4.70124914,
       4.70202939, 4.70280965, 4.7035899 , 4.70437016, 4.70515042,
       4.70593067, 4.70671093, 4.70749118, 4.70827144, 4.7090517 ,
       4.70983195, 4.71061221, 4.71139247])
```

[5]:
```
sim.antpos
```

[5]:
```
{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])}
```

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

[6]:
```
fig = sim.plot_array()
plt.show()
```



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

[7]:
```
from pyuvdata import UVData
isinstance(sim.data, UVData)
```

[7]:
```
True
```

We'll use a standard waterfall plot throughout the notebook to show the progress of the simulation:

[8]:
```
def waterfall(
    vis, freqs=sim.freqs * 1e3, lsts=sim.lsts,
    vmax=None, vrange=None, title=None,
):
    """
    A wrapper around the uvtools' waterfall function providing some
    extra labelling and plot adjustment.
    """
    fig, ax = plt.subplots(
        2,1, sharex=True, sharey=True, figsize=(12,10)
    )

    if title is not None:
        ax[0].set_title(title, fontsize=12)
```

(continues on next page)

(continued from previous page)

```
    plt.sca(ax[0])
    uvtools.plot.waterfall(
        vis, mode='log', mx=vmax, drng=vrange,
        extent=(freqs.min(), freqs.max(), lsts.min(), lsts.max())
    )
    plt.colorbar(label=r'log$_{10}$(Vis/Jy)')
    plt.ylabel("LST", fontsize=12)

    plt.sca(ax[1])
    uvtools.plot.waterfall(
        vis,
        mode='phs',
        cmap='twilight',
        extent=(freqs.min(), freqs.max(), lsts.min(), lsts.max())
    )
    plt.colorbar(label='Phase [rad]')
    plt.xlabel("Frequency [MHz]", fontsize=12)
    plt.ylabel("LST", fontsize=12)
```

Adding Effects

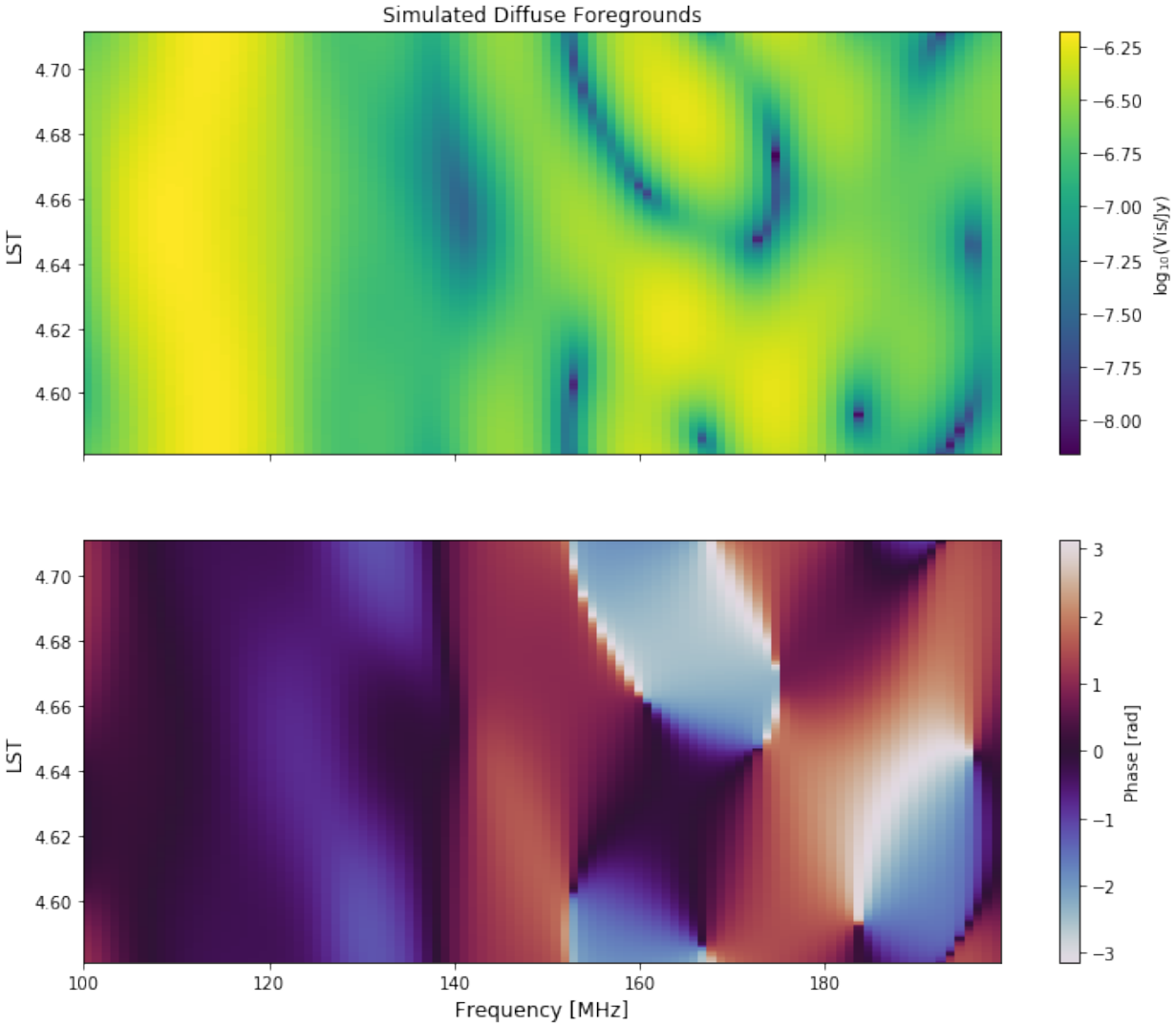
Effects may be added to a simulation by using the `add` method. This method takes one argument and variable keyword arguments: the required argument `component` may be either a string identifying the name of a `hera_sim` class (or an alias thereof, see below), or a callable object that has either `lsts` or `freqs` (or both) as a parameter(s) and returns an object with shape (`lsts.size`, `freqs.size`) or (`freqs.size`,) (granted, the `Simulator` does not do a check on the input parameters or the shape of returned values, but passing a callable object that returns something incorrectly shaped will likely result in an exception being raised). Let's walk through an example.

A Simple Example

Let's begin by simulating diffuse foregrounds. To do this, we'll need to specify a sky temperature model and a beam area interpolator object (or array of values corresponding to a frequency-dependent beam size). Let's use the sky temperature model and beam polynomial fit for H1C:

[9]:
```
from hera_sim.interpolators import Beam
from hera_sim.noise import HERA_Tsky_mdl
from hera_sim.data import DATA_PATH

# HERA_Tsky_mdl is a dictionary with keys 'xx' and 'yy'
Tsky_mdl = HERA_Tsky_mdl['xx']

# there are useful files in the data directory, with this being one
beamfile = os.path.join(DATA_PATH, "HERA_H1C_BEAM_POLY.npy")

# we can make a beam interpolation object from a path to a .npy file
omega_p = Beam(beamfile)

# now to add the foregrounds
sim.add("diffuse_foreground", Tsky_mdl=Tsky_mdl, omega_p=omega_p)
```

[10]:
```
# let's check out the data for the (0,1) baseline
vis = sim.data.get_data((0,1,'xx'))
waterfall(vis, title='Simulated Diffuse Foregrounds')
```



Now, whenever an effect is simulated using the `add` method, that effect (and any non-default optional parameters used) is logged: a note of what was simulated is put into the object's history, and the information is also stored in a hidden attribute `_components`.

[11]:
```
print(sim.data.history)
```

```
hera_sim v1.0.0: Added DiffuseForeground using kwargs:
Tsky_mdl = <hera_sim.interpolators.Tsky object at 0x7fb870d28b50>
omega_p = <hera_sim.interpolators.Beam object at 0x7fb86dbd3390>
```

[12]:
```
sim._components
```

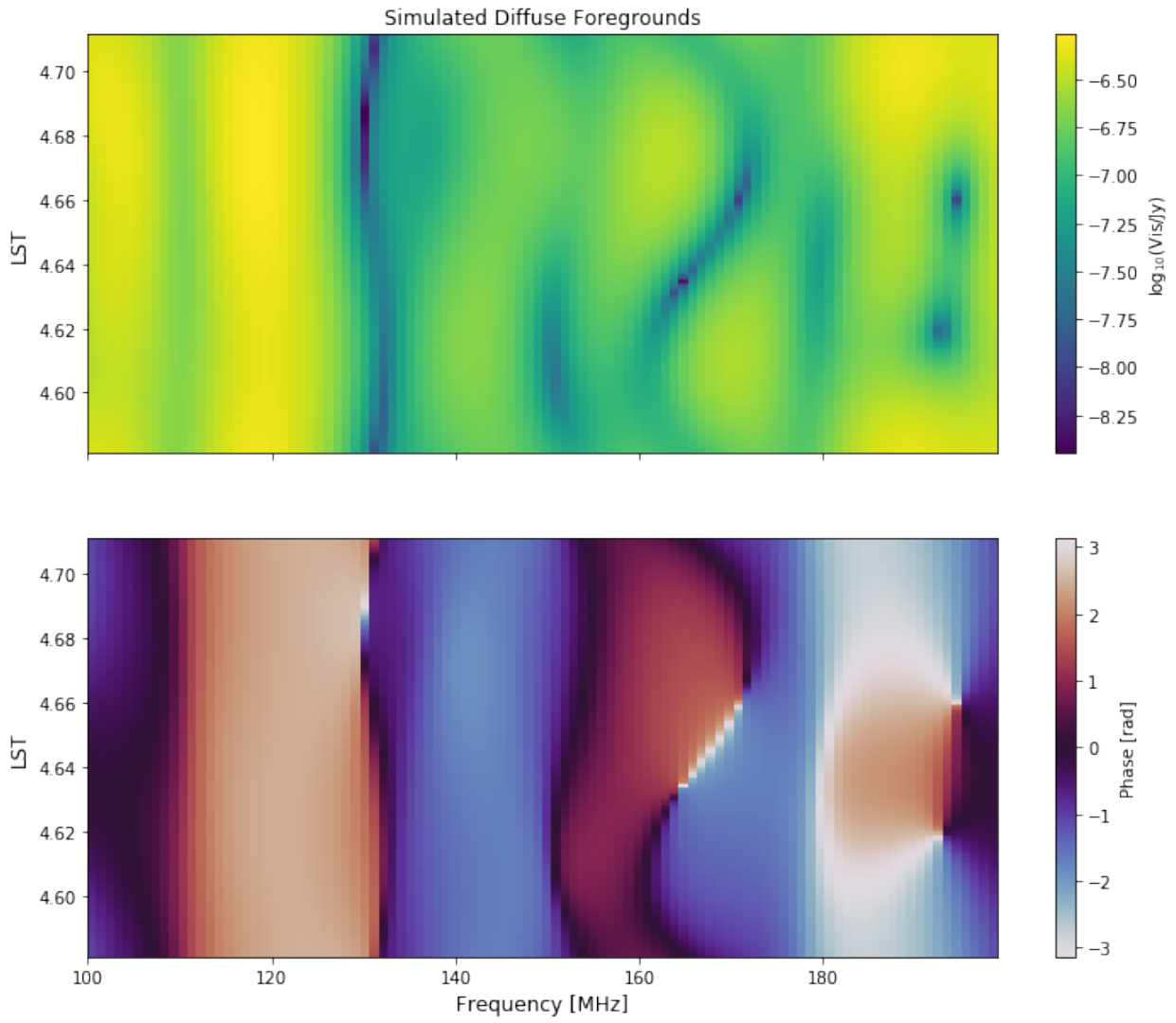
[12]:
```
{'diffuse_foreground': {'Tsky_mdl': <hera_sim.interpolators.Tsky at 0x7fb870d28b50>,
  'omega_p': <hera_sim.interpolators.Beam at 0x7fb86dbd3390>}}
```

Note that the actual visibility simulated is not cached, but rather the parameters necessary for recreating are saved and

can be used to later re-simulate the effect. Now, the diffuse foreground simulator uses a random number generator, so if we want to ensure repeatability, we have to make sure it's seeded in some repeatable way…

[13]:
```
# refresh the simulation
# this zeros the data array, resets the history, and clears the _components dictionary
sim.refresh()

from hera_sim.foregrounds import DiffuseForeground
# now let's seed by redundant group
sim.add(DiffuseForeground, Tsky_mdl=Tsky_mdl, omega_p=omega_p, seed_mode="redundantly
↪")
```

[14]:
```
# and let's take a look
vis = sim.data.get_data((0,1,'xx'))
waterfall(vis, title='Simulated Diffuse Foregrounds')
```



[15]:
```
# as for the history
print(sim.data.history)
```

```
hera_sim v1.0.0: Added DiffuseForeground using kwargs:
Tsky_mdl = <hera_sim.interpolators.Tsky object at 0x7fb870d28b50>
omega_p = <hera_sim.interpolators.Beam object at 0x7fb86dbd3390>
seed_mode = redundantly
```

[16]:
```
# and the _components
sim._components
```

[16]:
```
{hera_sim.foregrounds.DiffuseForeground: {'Tsky_mdl': <hera_sim.interpolators.Tsky at␣
↪0x7fb870d28b50>,
  'omega_p': <hera_sim.interpolators.Beam at 0x7fb86dbd3390>,
  'seed_mode': 'redundantly'}}
```

Note that using the `DiffuseForeground` class to simulate the effect caused the `Simulator` to log the component using the class itself as the key.

[17]:
```
# we can even re-simulate the effect
vis_copy = sim.get(DiffuseForeground, ant1=0, ant2=1, pol='xx')
np.all(np.isclose(vis,vis_copy))
```

[17]:
```
True
```

[18]:
```
# but do baselines within a redundant group agree?
for reds in sim._get_reds():
    red_grp = [sim.data.baseline_to_antnums(red) for red in reds]
    print(list(red_grp))
```

```
[(0, 0), (1, 1), (2, 2), (3, 3), (4, 4), (5, 5), (6, 6)]
[(0, 1), (2, 3), (3, 4), (5, 6)]
[(0, 2), (1, 3), (3, 5), (4, 6)]
[(0, 3), (1, 4), (2, 5), (3, 6)]
[(0, 4), (2, 6)]
[(0, 5), (1, 6)]
[(0, 6)]
[(1, 2), (4, 5)]
[(1, 5)]
[(2, 4)]
```

[19]:
```
np.all(
    np.isclose(sim.data.get_data(0,1), sim.data.get_data(2,3))
)
```

[19]:
```
True
```

A final note on this: when the `seed_mode` parameter is set to `"redundantly"`, a seed is generated for each redundant group, and that seed is used to ensure that each baseline within a redundant group observes the same realization of the effect; the only other mode that's currently supported is `"once"`, which just seeds the random number generator once before any effects are simulated.

Filtering Effects

The `add` method also allows for an optional keyword argument `vis_filter`. This may be an iterable of arbitrary length, or an iterable of keys of various lengths. This is still under-developed and may have a different implementation in the near future, so this section will not go into any detail about how it works.

As a quick example, let's choose our filter so that only the `'xx'` polarization for the baseline `(0,1)` (and its conjugate) is simulated. To check that it works, we'll test that the antennas' autoc

[20]:
```
sim.refresh()
sim.add("noiselike_eor", vis_filter=(0,1,'xx'))
np.all(sim.data.get_data(0,0,'xx') == 0)
```

[20]:
```
True
```

[21]:
```
np.all(sim.data.get_data(0,1,'xx') != 0) and np.all(sim.data.get_data(1,0,'xx') != 0)
```

[21]:
```
True
```

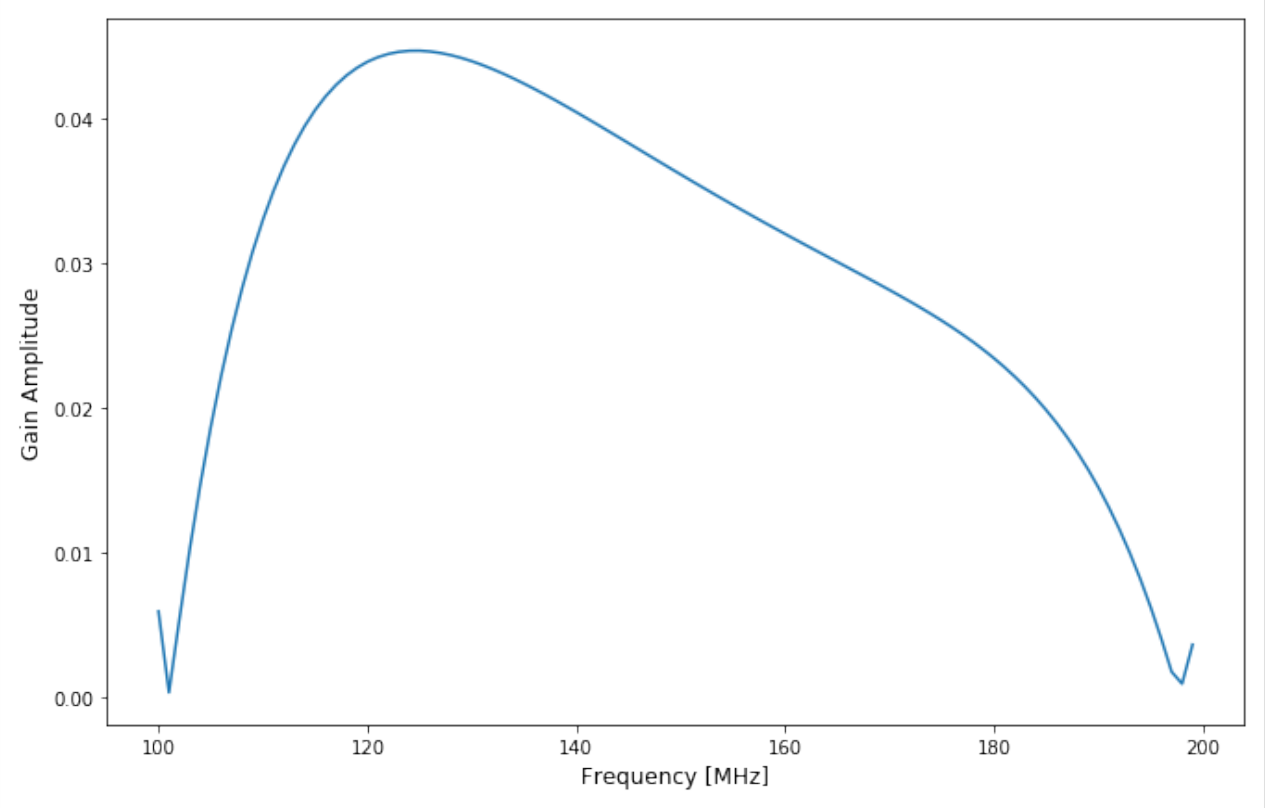
Ordering of Effects

Some of the effects that can be simulated act on visibilities additively, whereas others act multiplicatively. The `Simulator` class keeps track of the order in which simulation components have been added, and issues warnings if the simulation components have been added out of order. In particular, it issues warnings if a multiplicative model is added to an empty set of visibilities or if an absolute visibility is added after a multiplicative effect has been added. Let's check it out.

[22]:
```
# first, let's add a multiplicative effect
gains = sim.add("gains", seed_mode="once", ret_vis=True)
```

Note that you can set the `ret_vis` parameter to `True` in order to have the `add` method return the effect it calculated. In this case, it's a dictionary whose keys are antenna numbers and whose values are complex gains as a function of frequency.

[23]:
```
# take a quick peek at the gains
fig = plt.figure(figsize=(11,7))
ax = fig.add_subplot(111)
ax.plot(sim.freqs * 1e3, np.abs(gains[0]))
ax.set_xlabel("Frequency [MHz]", fontsize=12)
ax.set_ylabel("Gain Amplitude", fontsize=12)
plt.show()
```



Now let's try to add an additive effect, like EoR.

[24]:
```
sim.add("noiselike_eor")
```

```
You are adding visibilities to a data array *after* multiplicative effects have been␣
↪introduced.
```

Voila! The `Simulator` knows that a multiplicative effect (bandpass gains) have already been applied to the data and issues a warning when an additive effect is added afterward. Now let's refresh the simulation and check what happens if we try to add gains immediately after.

[25]:
```
sim.refresh()
```

[26]:
```
# now try adding a multiplicative effect to a fresh simulation
gains = sim.add("gains", seed_mode="once", ret_vis=True)
```

```
You are trying to compute a multiplicative effect, but no visibilities have been␣
↪simulated yet.
```

[27]:
```
# the effect is still simulated, but of course nothing happens
bool(gains) and np.all(sim.data.data_array == 0)
```

[27]:
```
True
```

[28]:
```
# note, however, that this warning is only issued once
sim.add("gains")
```

Using Custom Components

You can also use custom-defined callable objects as simulation components. Here's an example using a function:

[29]:
```
# first, make up a function
def mock_vis(lsts, freqs):
    return np.ones((lsts.size, freqs.size), dtype=np.complex)

# then refresh the sim and add the effect
sim.refresh()
sim.add(mock_vis)
```

```
You are attempting to compute a component but have not specified an ``is_
↪multiplicative`` attribute for the component. The component will be added under the␣
↪assumption that it is *not* multiplicative.
```

[30]:
```
# and now let's see what it's logged as
print(sim.data.history)
```

```
hera_sim v1.0.0: Added mock_vis using kwargs:
```

[31]:
```
# as for the components dictionary
sim._components
```

[31]:
```
{<function __main__.mock_vis(lsts, freqs)>: {}}
```

Note that a warning is raised if a user-defined function (or callable class) does not have an `is_multiplicative` attribute—this will be the case for *all* functions, but class implementations can have such an attribute defined.

Registering Classes

To wrap this section up, let's briefly cover how the `Simulator` class knows what to look for when the `add` method is called. All of the simulation components defined in the `hera_sim` repo are designed as callable classes which inherit from a `registry`. A new `registry` can be created by defining a class and decorating it with the `registry` decorator:

[32]:
```
from hera_sim import registry

@registry
class ExampleRegistry:
    is_multiplicative = False
    pass

class MockVis(ExampleRegistry):
    __aliases__ = ("mock_vis_alias",)
    def __call__(self, lsts, freqs):
        return np.ones((lsts.size, freqs.size), dtype=np.complex)
```

The above cell defines a new `registry` called `ExampleRegistry` and creates a class `MockVis` that is tracked by `ExampleRegistry`. Since the registry has its `is_multiplicative` attribute set to `False`, any class that inherits from it (and does not change the attribute's value) will also be considered an additive model.

[33]:
```
ExampleRegistry._models
```

[33]:
```
{'MockVis': __main__.MockVis}
```

[34]:
```
MockVis.is_multiplicative
```

[34]:
```
False
```

Since the `MockVis` class has `"mock_vis_alias"` as one of the entries in its `__aliases__` attribute, it can be discovered with that string:

[35]:
```
sim.refresh()
sim.add("mock_vis_alias")
```

[36]:
```
np.all(sim.data.data_array == 1)
```

[36]:
```
True
```

[37]:
```
print(sim.data.history)
```

```
hera_sim v1.0.0: Added MockVis using kwargs:


```

[38]:
```
sim._components
```

[38]:
```
{'mock_vis_alias': {}}
```

If you try to add a component that is not discoverable, then an exception will be raised and you'll be shown a list of known aliases:

[39]:
```
# show the message this way to let the rest of the kernel run
try:
    sim.add("not_a_known_model")
except AttributeError as err:
    print(err)
```

```
The component 'not_a_known_model' wasn't found. The following aliases are known:␣
↪cross_coupling_xtalk, thermal_noise, noiselikeeor, gen_gains, gen_reflection_gains,␣
↪impulse, gen_cross_coupling_xtalk, diffuseforeground, rfi_dtv, sigchain_reflections,
↪ rfi_impulse, diffuse_foreground, gen_whitenoise_xtalk, white_noise_xtalk, dtv,␣
↪pointsourceforeground, hexarray, bandpass, gains, stations, rfi_stations,␣
↪lineararray, reflections, mockvis, thermalnoise, whitenoisecrosstalk, scatter, mock_
↪vis_alias, crosscouplingcrosstalk, bandpass_gain, noiselike_eor, pntsrc_foreground,␣
↪rfi_scatter
```

If you would like to see what classes are registered and discoverable by the `Simulator` class, then you can use the `list_discoverable_components` function:

[40]:
```
hera_sim.list_discoverable_components()
```

```
hera_sim.antpos.LinearArray
hera_sim.antpos.HexArray
hera_sim.foregrounds.DiffuseForeground
hera_sim.foregrounds.PointSourceForeground
hera_sim.noise.ThermalNoise
hera_sim.rfi.Stations
hera_sim.rfi.Impulse
hera_sim.rfi.Scatter
hera_sim.rfi.DTV
hera_sim.sigchain.Bandpass
hera_sim.sigchain.Reflections
hera_sim.sigchain.CrossCouplingCrosstalk
```

(continues on next page)

(continued from previous page)

```
hera_sim.sigchain.WhiteNoiseCrosstalk
hera_sim.eor.NoiselikeEoR
__main__.MockVis
```

One final caveat: currently, the `add` method is implemented so that an exception is raised when a user-defined function that does not exist in the global namespace is passed as the `component`. So, if you want to import a simulation component *that is implemented as a function* from some other module, then be sure to import the function into the global namespace.

The `run_sim` Method

The `Simulator` class also features the `run_sim` method, which allows you to run an entire simulation by either specifying a dictionary whose keys are strings that specify simulation components (or acceptable aliases) compatible with the `add` method and whose values are the optional parameter settings for the simulation components.

An Example

Let's keep the example short and sweet: diffuse foregrounds with bandpass gains.

[41]:
```
# first, refresh the simulation to reset the history and _components
sim.refresh()

# now make a dictionary of simulation parameters
sim_params = {
    "diffuse_foreground" : {
        "Tsky_mdl" : Tsky_mdl, "omega_p" : omega_p, "seed_mode" : "redundantly"
    },
    "gains" : {"seed_mode" : "once"}
}

sim.run_sim(**sim_params)
```

[42]:
```
# and let's inspect the history
print(sim.data.history)
```

```
hera_sim v1.0.0: Added DiffuseForeground using kwargs:
Tsky_mdl = <hera_sim.interpolators.Tsky object at 0x7fb870d28b50>
omega_p = <hera_sim.interpolators.Beam object at 0x7fb86dbd3390>
seed_mode = redundantly
hera_sim v1.0.0: Added Bandpass using kwargs:
seed_mode = once
```

We can also use a YAML file to control the simulation. Note the use of the YAML tags `!Tsky` and `!Beam`; these are defined in the `__yaml_constructors` module, and they use the `Tsky` and `Beam` classes, respectively, from the `interpolators` module to construct an interpolation object from the provided `datafile` (and optional `interp_kwargs` dictionary). Note that currently these classes assume that relative paths are intended to be relative to the `hera_sim.data` directory, but this behavior may change in the future.

[43]:
```
# let's first make a configuration file
import tempfile
tempdir = tempfile.mkdtemp()
config = os.path.join(tempdir, "example.yaml")
```

(continues on next page)

(continued from previous page)

```
with open(config, 'w') as cfg:
    cfg.write(
"""
diffuse_foreground:
    Tsky_mdl: !Tsky
        datafile: HERA_Tsky_Reformatted.npz
    omega_p: !Beam
        datafile: HERA_H1C_BEAM_POLY.npy
    seed_mode: redundantly
gains:
    seed_mode: once
"""
)

# now refresh the simulation and run the new simulation
sim.refresh()
sim.run_sim(config)
```

[44]:
```
# again let's look at the history
print(sim.data.history)
```

```
hera_sim v1.0.0: Added DiffuseForeground using kwargs:
Tsky_mdl = <hera_sim.interpolators.Tsky object at 0x7fb8704b5cd0>
omega_p = <hera_sim.interpolators.Beam object at 0x7fb86e8598d0>
seed_mode = redundantly
hera_sim v1.0.0: Added Bandpass using kwargs:
seed_mode = once
```

[45]:
```
# and the components
sim._components
```

[45]:
```
{'diffuse_foreground': {'Tsky_mdl': <hera_sim.interpolators.Tsky at 0x7fb8704b5cd0>,
  'omega_p': <hera_sim.interpolators.Beam at 0x7fb86e8598d0>,
  'seed_mode': 'redundantly'},
 'gains': {'seed_mode': 'once'}}
```

Finally, you can return individual components from a simulation like so:

[46]:
```
# return some things, but not all things
sim_params["diffuse_foreground"]["ret_vis"] = True
sim_params["pntsrc_foreground"] = {"seed_mode" : "redundantly", "ret_vis" : True}

# reorder so that multiplicative effects come last
new_sim_params = {"diffuse_foreground" : {}, "pntsrc_foreground" : {}, "gains" : {}}
new_sim_params.update(sim_params)

# refresh the simulation and run
sim.refresh()
results = dict(sim.run_sim(**new_sim_params))
```

[47]:
```
# tada!
results.keys()
```

[47]:
```
dict_keys(['diffuse_foreground', 'pntsrc_foreground'])
```

[48]:
```
np.all(results['diffuse_foreground'].dtype == np.complex)
```

[48]:
```
True
```

[49]:
```
results['diffuse_foreground'].shape == sim.data.data_array.shape
```

[49]:
```
True
```

[50]:
```
np.all(
    np.isclose(results['diffuse_foreground'], sim.get("diffuse_foreground"))
)
```

[50]:
```
True
```

[51]:
```
sim.refresh()
sim.add("noiselike_eor", vis_filter=(0,1,'xx'))
```

[ ]:
```
```

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

1.2.3 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.
```

(continues on next page)

(continued from previous page)

```
  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()]

    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:
```

(continues on next page)

(continued from previous page)

```
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]
```

[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)))
```

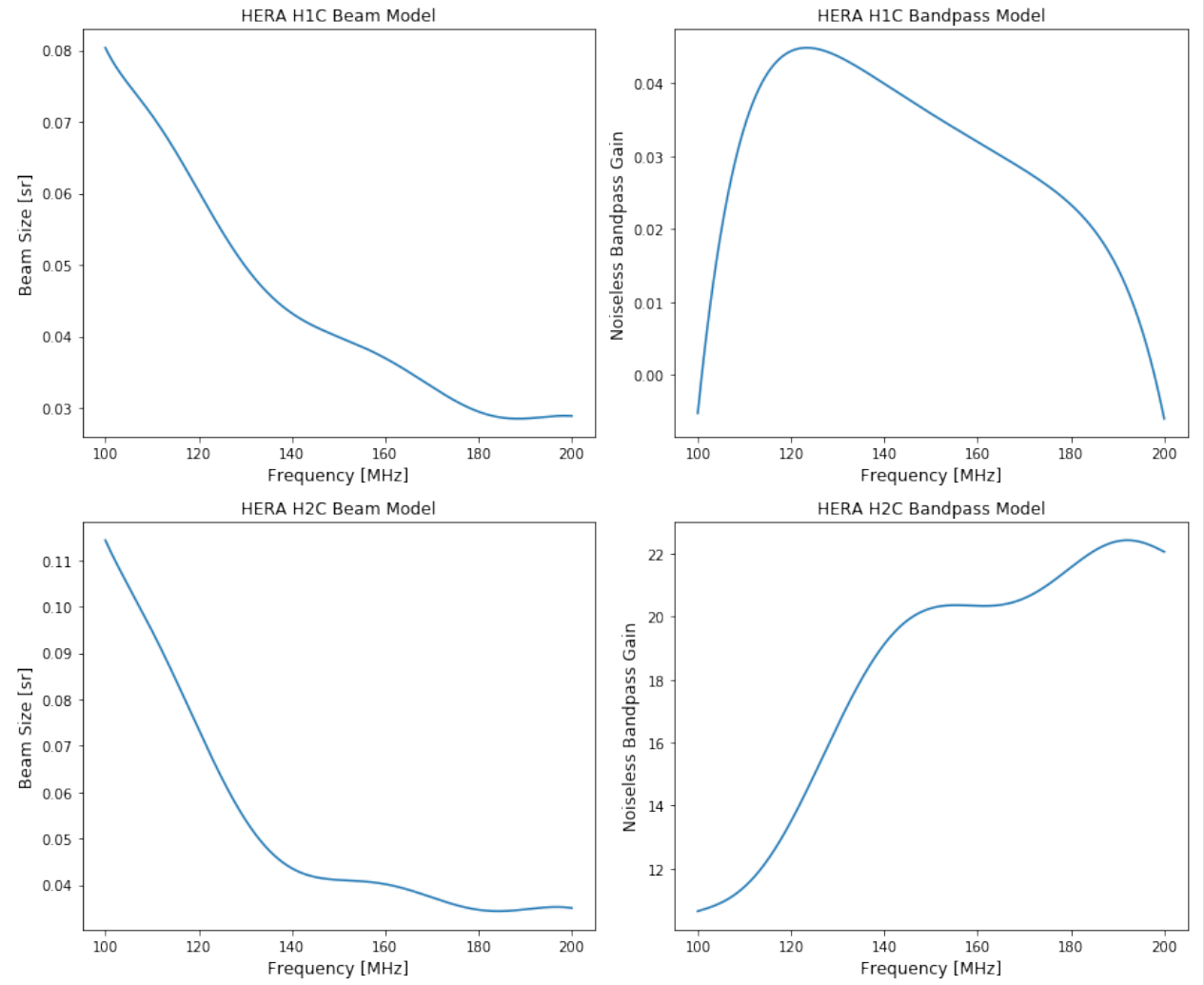
```
H1C noise identical to H2C noise? False
H2C noise identical to its other version? True
Either noise identical to other noise? False
```

[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)
```

(continues on next page)

(continued from previous page)

```
    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))
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},
```

(continues on next page)

(continued from previous page)

```
  '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}},
 '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)
```

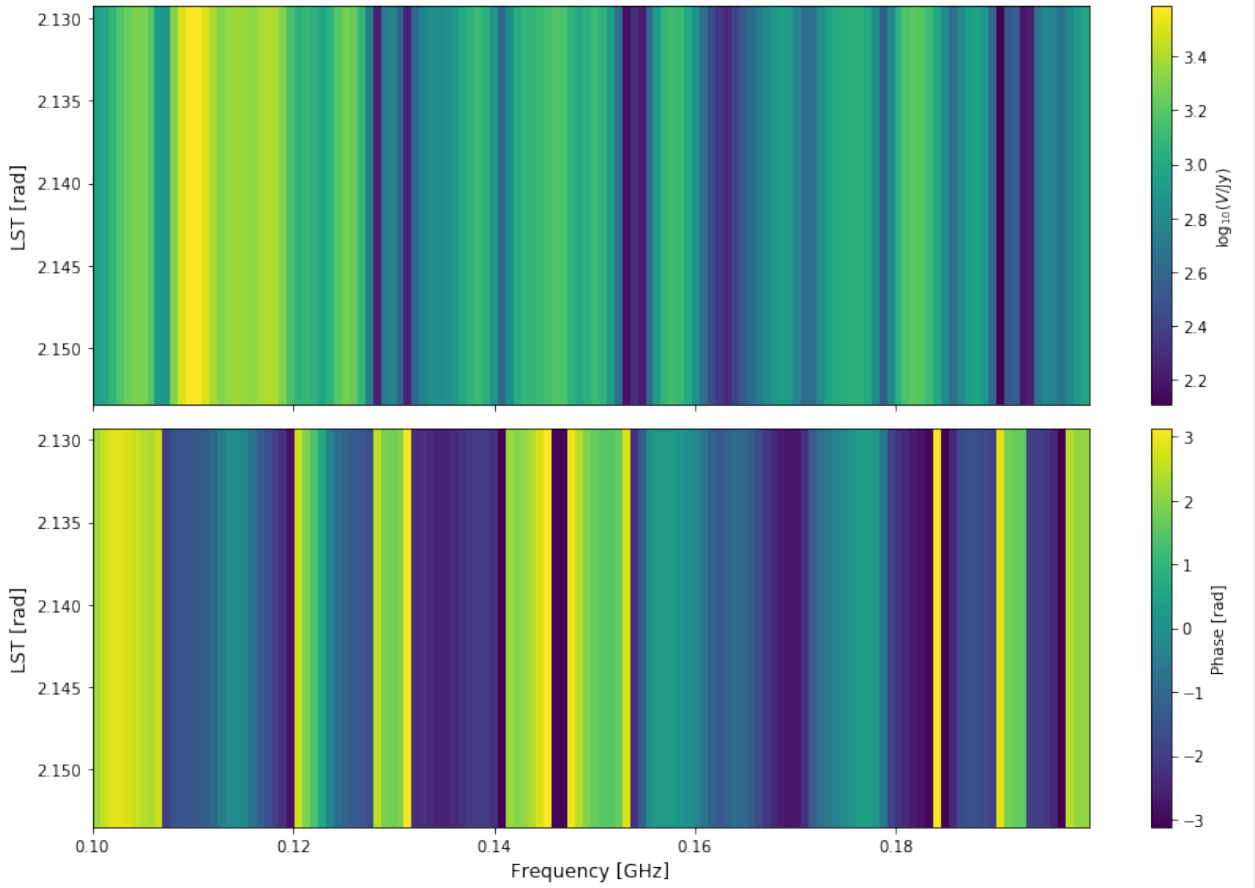
[13]:
```
# parameters for two different simulations
hera_sim.defaults.set('h1c')
sim1params = {'pntsrc_foreground': {},
              'noiselike_eor': {},
```

(continues on next page)

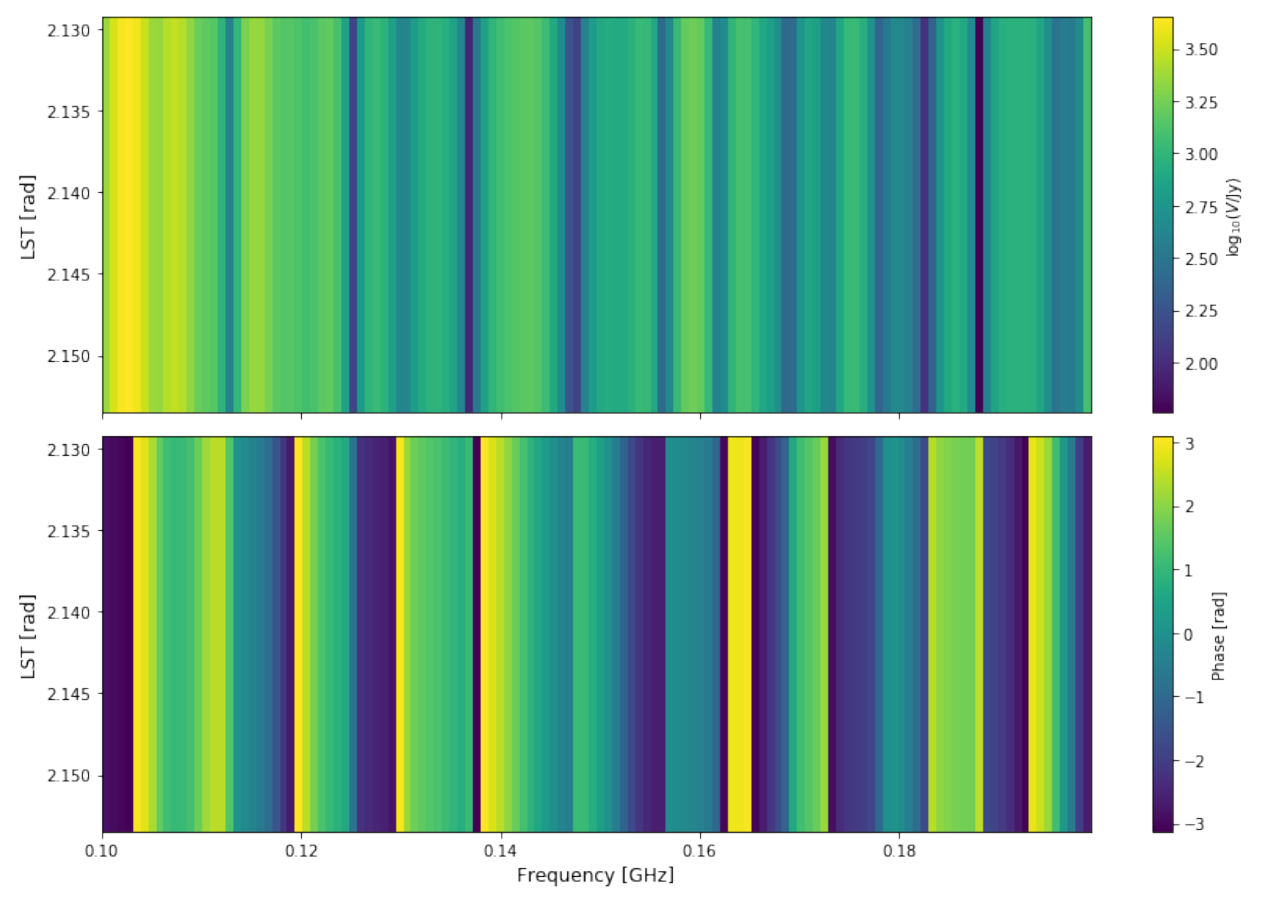
(continued from previous page)

```
                'diffuse_foreground': {}}
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:

1.2.4 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:

```
$ hera_sim run --help
Traceback (most recent call last):
  File "/home/docs/checkouts/readthedocs.org/user_builds/hera-sim/conda/
↪infrastructure/bin/hera_sim", line 5, in <module>
    from hera_sim.cli import main
  File "/home/docs/checkouts/readthedocs.org/user_builds/hera-sim/conda/
↪infrastructure/lib/python3.8/site-packages/hera_sim/__init__.py", line 21, in
↪<module>
    from . import io
  File "/home/docs/checkouts/readthedocs.org/user_builds/hera-sim/conda/
↪infrastructure/lib/python3.8/site-packages/hera_sim/io.py", line 7, in <module>
    from pyuvsim.simsetup import initialize_uvdata_from_keywords
  File "/home/docs/checkouts/readthedocs.org/user_builds/hera-sim/conda/
↪infrastructure/lib/python3.8/site-packages/pyuvsim/__init__.py", line 7, in <module>
```

(continues on next page)

(continued from previous page)

```
    from .uvsim import *  # noqa
  File "/home/docs/checkouts/readthedocs.org/user_builds/hera-sim/conda/
↪infrastructure/lib/python3.8/site-packages/pyuvsim/uvsim.py", line 14, in <module>
    import pyradiosky
ModuleNotFoundError: No module named 'pyradiosky'
```

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'
    21            output_format: 'uvh5'
    22            clobber: True
    23            kwargs:
    24                save_seeds: True # if seeding RNG by redundant group
    25        # freq and time entries currently configured for hera_sim use
    26        freq:
    27            n_freq: 100
    28            channel_width: 122070.3125
    29            start_freq: 46920776.3671875
    30        time:
    31            n_times: 10
    32            integration_time: 8.59
    33            start_time: 2457458.1738949567
    34        telescope:
    35            # generate from an antenna layout csv
    36            # array_layout: 'antenna_layout.csv'
    37            # generate using hera_sim.antpos
    38            array_layout: !antpos
    39                array_type: "hex"
    40                hex_num: 3
    41                sep: 14.6
    42                split_core: False
    43                outriggers: 0
    44            omega_p: !Beam
    45                # non-absolute paths are assumed to be specified relative to the
    46                # hera_sim data path
```

(continues on next page)

(continued from previous page)

```
    47                datafile: HERA_H2C_BEAM_MODEL.npz
    48                interp_kwargs:
    49                    interpolator: interp1d
    50                    fill_value: extrapolate
    51                    # if you want to use a polynomial interpolator instead, then
    52                    # interpolator: poly1d
    53                    # kwargs not accepted for this; see numpy.poly1d␣
↪documentation
    54        defaults:
    55            # This must be a string specifying an absolute path to a default
    56            # configuration file or one of the season default keywords
    57            default_config: 'h2c'
    58        systematics:
    59            rfi:
    60                # see hera_sim.rfi documentation for details on parameter names
    61                rfi_stations:
    62                    stations: !!null
    63                rfi_impulse:
    64                    chance: 0.001
    65                    strength: 20.0
    66                rfi_scatter:
    67                    chance: 0.0001
    68                    strength: 10.0
    69                    std: 10.0
    70                rfi_dtv:
    71                    freq_min: 0.174
    72                    freq_max: 0.214
    73                    width: 0.008
    74                    chance: 0.0001
    75                    strength: 10.0
    76                    strength_std: 10.0
    77            sigchain:
    78                gains:
    79                    gain_spread: 0.1
    80                    dly_rng: [-20, 20]
    81                    bp_poly: HERA_H1C_BANDPASS.npy
    82                sigchain_reflections:
    83                    amp: !!null
    84                    dly: !!null
    85                    phs: !!null
    86            crosstalk:
    87                # only one of the two crosstalk methods should be specified
    88                gen_whitenoise_xtalk:
    89                    amplitude: 3.0
    90                # gen_cross_coupling_xtalk:
    91                    # amp: !!null
    92                    # dly: !!null
    93                    # phs: !!null
    94            noise:
    95                thermal_noise:
    96                    Trx: 0
    97        sky:
    98            Tsky_mdl: !Tsky
    99                # non-absolute paths are assumed to be relative to the hera_sim
   100                # data folder
   101                datafile: HERA_Tsky_Reformatted.npz
   102                # interp kwargs are passed to scipy.interp.RectBivariateSpline
```

(continues on next page)

(continued from previous page)

```
  103                interp_kwargs:
  104                    pol: xx # this is popped when making a Tsky object
  105            eor:
  106                noiselike_eor:
  107                    eor_amp: 0.00001
  108                    min_delay: !!null
  109                    max_delay: !!null
  110                    seed_redundantly: True # so redundant baselines see same sky
  111                    fringe_filter_type: tophat
  112                    fringe_filter_kwargs: {}
  113            foregrounds:
  114                # if using hera_sim.foregrounds
  115                diffuse_foreground:
  116                    seed_redundantly: True # redundant baselines see same sky
  117                    standoff: 0
  118                    delay_filter_type: tophat
  119                    delay_filter_normalize: !!null
  120                    fringe_filter_type: tophat
  121                    fringe_filter_kwargs: {}
  122                pntsrc_foreground:
  123                    seed_redundantly: True
  124                    nsrcs: 1000
  125                    Smin: 0.3
  126                    Smax: 300
  127                    beta: -1.5
  128                    spectral_index_mean: -1.0
  129                    spectral_index_std: 0.5
  130                    reference_freq: 0.5
  131                # Note regarding seed_redundantly:
  132                # This ensures that baselines within a redundant group see the␣
↪same sky;
  133                # however, this does not ensure that the sky is actually␣
↪consistent. So,
  134                # while the data produced can be absolutely calibrated, it␣
↪cannot be
  135                # used to make sensible images (at least, I don't *think* it␣
↪can be).
  136
  137        simulation:
  138            # specify which components to simulate in desired order
  139            # this should be a complete list of the things to include if hera_
↪sim
  140            # is the simulator being used. this will necessarily look different
  141            # if other simulators are used, but that's not implemented yet
  142            #
  143            components: [foregrounds,
  144                         noise,
  145                         eor,
  146                         rfi,
  147                         sigchain, ]
  148            # list particular model components to exclude from simulation
  149            exclude: [sigchain_reflections,
  150                      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: '.'
    outfile_name: 'quick_and_dirty_sim'
    output_format: 'uvh5'
    clobber: True
    kwargs:
        save_seeds: True # if seeding RNG by redundant group
```

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
freq:
    n_freq: 100
    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
```

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
defaults:
    # This must be a string specifying an absolute path to a default
    # configuration file or one of the season default keywords
    default_config: 'h2c'
```

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
systematics:
    rfi:
        # see hera_sim.rfi documentation for details on parameter names
        rfi_stations:
            stations: !!null
        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:
        gains:
            gain_spread: 0.1
            dly_rng: [-20, 20]
            bp_poly: HERA_H1C_BANDPASS.npy
        sigchain_reflections:
            amp: !!null
            dly: !!null
            phs: !!null
    crosstalk:
        # only one of the two crosstalk methods should be specified
        gen_whitenoise_xtalk:
```

(continues on next page)

(continued from previous page)

```
            amplitude: 3.0
        # gen_cross_coupling_xtalk:
            # amp: !!null
            # dly: !!null
            # phs: !!null
    noise:
        thermal_noise:
            Trx: 0
```

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
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:
            eor_amp: 0.00001
            min_delay: !!null
            max_delay: !!null
            seed_redundantly: True # so redundant baselines see same sky
            fringe_filter_type: tophat
            fringe_filter_kwargs: {}
    foregrounds:
        # if using hera_sim.foregrounds
        diffuse_foreground:
            seed_redundantly: True # redundant baselines see same sky
            standoff: 0
            delay_filter_type: tophat
            delay_filter_normalize: !!null
            fringe_filter_type: tophat
            fringe_filter_kwargs: {}
        pntsrc_foreground:
            seed_redundantly: True
            nsrcs: 1000
            Smin: 0.3
            Smax: 300
            beta: -1.5
            spectral_index_mean: -1.0
            spectral_index_std: 0.5
```

(continues on next page)

(continued from previous page)

```
            reference_freq: 0.5
```

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
simulation:
    # specify which components to simulate in desired order
    components: [foregrounds,
                 noise,
                 eor,
                 rfi,
                 sigchain, ]
    # list particular model components to exclude from simulation
    exclude: [sigchain_reflections,
              gen_whitenoise_xtalk,]
```

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.

1.3 API Reference

1.3.1 hera_sim

hera_sim.vis	
hera_sim.antpos	A module for creating antenna array configurations.
hera_sim.defaults	This module is designed to allow for easy interfacing with simulation default parameters in an interactive environment.
hera_sim.eor	
hera_sim.foregrounds	Reimagining of the foregrounds module, using an object-oriented approach.

continues on next page

Table 1 – continued from previous page

hera_sim.interpolators	This module provides interfaces to different interpolation classes.
hera_sim.io	
hera_sim.noise	
hera_sim.rfi	
hera_sim.sigchain	
hera_sim.simulate	
hera_sim.utils	Utility module

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

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.HexArray

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

Build a hexagonal array configuration, nominally matching HERA.

Methods

__init__([sep, split_core, outriggers])	

Parameters

- **sep** (*int, optional*) – The separation between adjacent grid points, in meters.

hera_sim.antpos.HexArray.__init__

HexArray.**__init__**(*sep=14.6*, *split_core=True*, *outriggers=2*)

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.

Attributes

is_multiplicative

hera_sim.antpos.HexArray.is_multiplicative

HexArray.**is_multiplicative = False**

hera_sim.antpos.LinearArray

class hera_sim.antpos.**LinearArray**(*sep=14.6*)
Build a linear (east-west) array configuration.

Methods

__init__([sep])	**Parameters** **sep** (*float, optional*) – The separation between adjacent antennas, in meters.

hera_sim.antpos.LinearArray.__init__

LinearArray.**__init__**(*sep=14.6*)

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

Attributes

is_multiplicative

hera_sim.antpos.LinearArray.is_multiplicative

LinearArray.**is_multiplicative = False**

hera_sim.defaults

This module is designed to allow for easy interfacing with simulation default parameters in an interactive environment.

Classes

Defaults([config])	Class for dynamically changing hera_sim parameter defaults.

hera_sim.defaults.Defaults

class hera_sim.defaults.**Defaults**(*config=None*)

Class for dynamically changing hera_sim parameter defaults.

This class handles the retreival of simulation default parameters from YAML files and the ability to switch the default settings while in an interactive environment. This class is intended to exist as a singleton; as such, an instance is created at the end of this module, and that instance is what is imported in the hera_sim constructor. See below for example usage within hera_sim.

Examples

To set the default parameters to those appropriate for the H2C observing season (and activate the use of those defaults):

hera_sim.defaults.set('h2c')

To set the defaults to a custom set of defaults, you must first create a configuration YAML. Assuming the path to the YAML is stored in the variable *config_path*, these defaults would be set via the following line:

hera_sim.defaults.set(config_path)

To revert back to using defaults defined in function signatures:

hera_sim.defaults.deactivate()

To view what the default value is for a particular parameter, do:

hera_sim.defaults(parameter),

where *parameter* is a string with the name of the parameter as listed in the configuration file. To view the entire set of default parameters, use:

hera_sim.defaults()

Methods

__init__([config])	Load in a configuration and check its formatting.
activate()	Activate the defaults.
apply(func_kwargs, **kwargs)	Just update the kwargs given the function kwargs.
deactivate()	Revert to function defaults.
set(new_config[, refresh])	Set the defaults to those specified in *new_config*.

hera_sim.defaults.Defaults.__init__

Defaults.**__init__**(*config=None*)

Load in a configuration and check its formatting.

Parameters **config** (*str or dict, optional (default 'h1c')*) – May either be an absolute path to a configuration YAML, one of the observing season keywords ('h1c', 'h2c'), or a dictionary with the appropriate format.

Notes

The configuration file may be formatted in practically any way, as long as it is parsable by *pyyaml*. That said, the resulting configuration will *always* take the form {param : value} for every item (param, value) such that *value* is not a dict. A consequence of this is that any parameters whose names are not unique will take on the value specified last in the config. The raw configuration is kept in memory, but it currently is not used for overriding any default values.

Examples

Consider the following contents of a configuration file:

foregrounds:

Tsky_mdl: !Tsky datafile: HERA_Tsky_Reformatted.npz

seed_redundantly: True nsrcs: 500

gains: gain_spread: 0.1 dly_rng: [-10, 10] bp_poly: HERA_H1C_BANDPASS.npy

This would result in the following set of defaults:

{Tsky_mdl: <hera_sim.interpolators.Tsky instance>, seed_redundantly: True, nsrcs: 500, gain_spread: 0.1, dly_rng: [-10,10] bp_poly: HERA_H1C_BANDPASS.npy }

Now consider a different configuration file:

sky:

eor: eor_amp: 0.001

systematics:

rfi:

rfi_stations: stations: !!null

rfi_impulse: chance: 0.01

rfi_scatter: chance: 0.35

crosstalk: amplitude: 1.25

gains: gain_spread: 0.2

noise: Trx: 150

Since the parser recursively unpacks the raw configuration dictionary until no entry is nested, the resulting config is:

{eor_amp: 0.001, stations: None, chance: 0.35, amplitude: 1.25, gain_spread: 0.2, Trx: 150 }

hera_sim.defaults.Defaults.activate

Defaults.**activate**()
Activate the defaults.

hera_sim.defaults.Defaults.apply

Defaults.**apply**(*func_kwargs*, ***kwargs*)
Just update the kwargs given the function kwargs.

hera_sim.defaults.Defaults.deactivate

Defaults.**deactivate**()
Revert to function defaults.

hera_sim.defaults.Defaults.set

Defaults.**set**(*new_config*, *refresh=False*)
Set the defaults to those specified in *new_config*.

Parameters

- **new_config** (*str or dict*) – Absolute path to configuration file or dictionary of configuration parameters formatted in the same way a configuration would be loaded.
- **refresh** (*bool, optional*) – Choose whether to completely overwrite the old config or just add new values to it.

Notes

Calling this method also activates the defaults.

hera_sim.foregrounds

Reimagining of the foregrounds module, using an object-oriented approach.

Classes

DiffuseForeground([Tsky_mdl, omega_p, ...])	
PointSourceForeground([nsrcs, Smin, Smax, ...])	

hera_sim.foregrounds.DiffuseForeground

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

Methods

__init__([Tsky_mdl, omega_p, ...])	

Parameters

- **Tsky_mdl** (*interpolation object*) – Sky temperature model, in units of Kelvin. Must be callable

hera_sim.foregrounds.DiffuseForeground.__init__

DiffuseForeground.**__init__**(*Tsky_mdl=None*, *omega_p=None*, *delay_filter_kwargs=None*, *fringe_filter_kwargs=None*)

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 .. func:: utils.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 .. func:: utils.rough_fringe_filter. Default is to use the following settings:

  fringe_filter_type : tophat

Attributes

is_multiplicative

hera_sim.foregrounds.DiffuseForeground.is_multiplicative

DiffuseForeground.**is_multiplicative = False**

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*)

Methods

__init__([nsrcs, Smin, Smax, beta, ...])	**Parameters** • **nsrcs** (*int, optional*) – Number of sources to place on the sky. Point sources are

hera_sim.foregrounds.PointSourceForeground.__init__

PointSourceForeground.**__init__**(*nsrcs=1000*, *Smin=0.3*, *Smax=300*, *beta=- 1.5*, *spectral_index_mean=- 1*, *spectral_index_std=0.5*, *reference_freq=0.15*)

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`. The default behavior is to use 1000 sources.
- **Smin** (*float, optional*) – Lower bound of the power-law distribution to draw flux-densities from, in units of Jy. Default is 0.3 Jy.
- **Smax** (*float, optional*) – Upper bound of the power-law distribution to draw flux-densities from, in units of Jy. Default is 300 Jy.
- **beta** (*float, optional*) – Power law index for the source counts versus flux-density. Default is -1.5.
- **spectral_index_mean** (*float, optional*) – The mean of the normal distribution to draw source spectral indices from. Default is -1.

- **spectral_index_std** (*float, optional*) – The standard deviation of the normal distribution to draw source spectral indices from. Default is 0.5.
- **reference_freq** (*float, optional*) – Reference frequency used to make the point source flux densities chromatic, in units of GHz. Default is 0.15 GHz.

Attributes

is_multiplicative	

hera_sim.foregrounds.PointSourceForeground.is_multiplicative

PointSourceForeground.**is_multiplicative = False**

hera_sim.interpolators

This module provides interfaces to different interpolation classes.

Classes

Bandpass(datafile, **interp_kwargs)	Bandpass interpolation object; subclass of *FreqInterpolator*.
Beam(datafile, **interp_kwargs)	Beam interpolation object; subclass of *FreqInterpolator*.
FreqInterpolator(datafile, **interp_kwargs)	Frequency interpolator; subclass of *Interpolator*.
Interpolator(datafile, **interp_kwargs)	Base interpolator class
Tsky(datafile, **interp_kwargs)	Sky temperature interpolator; subclass of *Interpolator*.

hera_sim.interpolators.Bandpass

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

Bandpass interpolation object; subclass of *FreqInterpolator*.

Methods

__init__(datafile, **interp_kwargs)	Extend the *FreqInterpolator* constructor.

hera_sim.interpolators.Bandpass.__init__

Bandpass.**__init__**(*datafile*, ***interp_kwargs*)
Extend the *FreqInterpolator* constructor.

Parameters

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

hera_sim.interpolators.Beam

class hera_sim.interpolators.**Beam**(*datafile*, ***interp_kwargs*)
Beam interpolation object; subclass of *FreqInterpolator*.

Methods

__init__(datafile, **interp_kwargs)	Extend the *FreqInterpolator* constructor.

hera_sim.interpolators.Beam.__init__

Beam.**__init__**(*datafile*, ***interp_kwargs*)
Extend the *FreqInterpolator* constructor.

Parameters

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

hera_sim.interpolators.FreqInterpolator

class hera_sim.interpolators.**FreqInterpolator**(*datafile*, ***interp_kwargs*)
Frequency interpolator; subclass of *Interpolator*.

Methods

__init__(datafile, **interp_kwargs)	Extend the *Interpolator* constructor.

hera_sim.interpolators.FreqInterpolator.__init__

FreqInterpolator.**__init__**(*datafile*, ***interp_kwargs*)
Extend the *Interpolator* constructor.

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.

hera_sim.interpolators.Interpolator

class hera_sim.interpolators.**Interpolator**(*datafile*, ***interp_kwargs*)
Base interpolator class

Methods

__init__(datafile, **interp_kwargs)	Initialize an *Interpolator* object with necessary attributes.

hera_sim.interpolators.Interpolator.__init__

Interpolator.**__init__**(*datafile*, ***interp_kwargs*)
Initialize an *Interpolator* object with necessary attributes.

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.

hera_sim.interpolators.Tsky

class hera_sim.interpolators.**Tsky**(*datafile*, ***interp_kwargs*)
Sky temperature interpolator; subclass of *Interpolator*.

Methods

__init__(datafile, **interp_kwargs)	Extend the *Interpolator* constructor.

hera_sim.interpolators.Tsky.__init__

Tsky.**__init__**(*datafile*, ***interp_kwargs*)
Extend the *Interpolator* constructor.

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).

Attributes

freqs
lsts
meta
tsky

hera_sim.interpolators.Tsky.freqs

property Tsky.**freqs**

hera_sim.interpolators.Tsky.lsts

property Tsky.**lsts**

hera_sim.interpolators.Tsky.meta

property Tsky.**meta**

hera_sim.interpolators.Tsky.tsky

property Tsky.**tsky**

hera_sim.utils

Utility module

Functions

Jy2T(freqs, omega_p)	Return Kelvin -> Jy conversion as a function of frequency.
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.
gen_delay_filter(fqs, bl_len_ns[, standoff, ...])	Generate a delay filter in delay space.
gen_fringe_filter(lsts, fqs, ew_bl_len_ns[, ...])	Generate a fringe rate filter in fringe-rate & freq space.
gen_white_noise([size])	Produce complex Gaussian noise with unity variance.
get_bl_len_magnitude(bl_len_ns)	Get the magnitude of the length of the given baseline.
rough_delay_filter(data, fqs, bl_len_ns[, ...])	A rough low-pass delay filter of data array along last axis.
rough_fringe_filter(data, lsts, fqs, ...[, ...])	A rough fringe rate filter of data along zeroth axis.

hera_sim.utils.Jy2T

hera_sim.utils.**Jy2T**(*freqs*, *omega_p*)

Return Kelvin -> Jy conversion as a function of frequency.

Parameters

- **freqs** (*ndarray*) – Frequencies for which to calculate the conversion. Units of Hz.
- **omega_p** (*ndarray or interpolators.Beam*) – 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 Hz) to beam size.

Returns **Jy_to_K** (*ndarray*) – Array for converting Jy to K, same shape as `freqs`.

hera_sim.utils.calc_max_fringe_rate

hera_sim.utils.**calc_max_fringe_rate**(*fqs*, *ew_bl_len_ns*)

Calculate the max fringe-rate seen by an East-West baseline.

Parameters

- **fqs** (*ndarray*) – frequency array [GHz]
- **ew_bl_len_ns** (*float*) – projected East-West baseline length [ns]

Returns *fr_max (float)* – fringe rate [Hz]

hera_sim.utils.compute_ha

hera_sim.utils.**compute_ha**(*lsts*, *ra*)

Compute hour angle from local sidereal time and right ascension.

Arg:

lsts: array-like, shape=(NTIMES,), radians local sidereal times of the observation to be generated.

ra: float, radians the right ascension of a point source.

Returns

ha –

array-like, shape=(NTIMES,) hour angle corresponding to the provide ra and times"'

hera_sim.utils.gen_delay_filter

hera_sim.utils.**gen_delay_filter**(*fqs*, *bl_len_ns*, *standoff=0.0*, *filter_type='gauss'*, *min_delay=None*, *max_delay=None*, *normalize=None*)

Generate a delay filter in delay space.

Parameters

- **fqs** (*ndarray*) – frequency array [GHz]
- **bl_len_ns** (*float or array*) – total baseline length or baseline vector in [ns]
- **standoff** (*float*) – supra-horizon buffer [nanosec]

- **filter_type** (*str*) – options=['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** (*float*) – minimum absolute delay of filter
- **max_delay** (*float*) – maximum absolute delay of filter
- **normalize** – float, optional 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 (ndarray)* – delay filter in delay space

hera_sim.utils.gen_fringe_filter

hera_sim.utils.**gen_fringe_filter**(*lsts*, *fqs*, *ew_bl_len_ns*, *filter_type='tophat'*, ***filter_kwargs*)

Generate a fringe rate filter in fringe-rate & freq space.

Parameters

- **lsts** (*ndarray*) – lst array [radians]
- **fqs** (*ndarray*) – frequency array [GHz]
- **ew_bl_len_ns** (*float*) – projected East-West baseline length [nanosec]
- **filter_type** (*str*) – options=['tophat', 'gauss', 'custom', 'none']
- **filter_kwargs** – kwargs for different filter types filter_type == 'gauss'

  fr_width (float or array): Sets gaussian width in fringe-rate [Hz]

  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 (ndarray)* – 2D ndarray 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 frate 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 fftshifted along axis 0, but output filter is ifftshifted 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=1*)

Produce complex Gaussian noise with unity variance.

Parameters **size** (*int or tuple, optional*) – Shape of output array.

Returns **noise** (*ndarray*) – White noise realization with specified shape.

hera_sim.utils.get_bl_len_magnitude

hera_sim.utils.**get_bl_len_magnitude**(*bl_len_ns*)

Get the magnitude of the length of the given baseline.

Parameters **bl_len_ns** (*scalar or array_like*) – 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 *float* – The magnitude of the baseline length.

hera_sim.utils.rough_delay_filter

hera_sim.utils.**rough_delay_filter**(*data*, *fqs*, *bl_len_ns*, *standoff=0.0*, *delay_filter_type='gauss'*, *min_delay=None*, *max_delay=None*, *normalize=None*)

A rough low-pass delay filter of data array along last axis.

Parameters

- **data** (*ndarray*) – data to be filtered along last axis
- **fqs** (*ndarray*) – frequency array [GHz]
- **bl_len_ns** (*float or array*) – total baseline length or baseline vector [nanosec]
- **standoff** (*float*) – supra-horizon buffer [nanosec]
- **filter_type** (*str*) – options=['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** (*float*) – minimum absolute delay of filter
- **max_delay** (*float*) – maximum absolute delay of filter
- **normalize** – float, optional 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 *filt_data (ndarray)* – filtered data array

hera_sim.utils.rough_fringe_filter

hera_sim.utils.**rough_fringe_filter**(*data*, *lsts*, *fqs*, *ew_bl_len_ns*, *fringe_filter_type='tophat'*, ***filter_kwargs*)

A rough fringe rate filter of data along zeroth axis.

Parameters

- **data** (*ndarray*) – data to filter along zeroth axis
- **lsts** (*ndarray*) – LST array [radians]
- **fqs** (*ndarray*) – frequency array [GHz]
- **ew_bl_len_ns** (*float*) – projected East-West baseline length [nanosec]
- **filter_type** (*str*) – options=['tophat', 'gauss', 'custom', 'none']
- **filter_kwargs** – kwargs for different filter types filter_type == 'gauss'

  fr_width (float or array): Sets gaussian width in fringe-rate [Hz]

  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 *filt_data (ndarray)* – filtered data

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 frate array is fed as "FR_frates" in Hz, its freq array is fed as "FR_freqs" in GHz

If filter_type == 'none': fringe filter is identically one.

1.4 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

1.4.1 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.

1.4.2 Documentation improvements

`hera_sim` could always use more documentation, whether as part of the official `hera_sim` docs or in docstrings.

1.4.3 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 :)

1.4.4 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 -f travis-environment.yml
   ```

4. When you're done making changes, run all the checks, doc builder and spell checker with tox one command:

   ```
   tox
   ```

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 `tox`)[1].
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`.

1.5 Developing `hera_sim`

`hera_sim` broadly follows the best-practices laid out in XXX.

Todo: where is that best-practices doc?

All docstrings should be written in Google docstring format.

1.6 AUTHORS

- HERA-Team - https://github.com/HERA-Team

1.7 Changelog

1.7.1 v1.0.0 [???]

Added

Fixed

Changed

- All functions that take frequencies and LSTs as arguments have had their signatures changed to func(lsts, freqs, *args, **kwargs)
- Changes to handling of functions which employ a fringe or delay filtering step with variable keywords for the filters used. Filter keywords are now specified with individual dictionaries called `delay_filter_kwargs` or `fringe_filter_kwargs`, depending on the filter used.
  - **Functions affected by this change are as follows:**
    * `diffuse_foreground`
    * `noiselike_eor`

[1] If you don't have all the necessary python versions available locally you can rely on Travis - it will run the tests for each change you add in the pull request.
It will be slower though ...

- Changes to parameters that shared the same name but represented conceptually different objects in various functions. Functions affected by this are as follows:
  - `utils.rough_delay_filter`
  - `utils.rough_fringe_filter`
  - Most RFI functions
- The `io.empty_uvdata` function had its default keyword values set to `None`. The keywords accepted by this function have also been changed to match their names in `pyuvsim.simsetup.initialize_uvdata_from_keywords`
- Changes to parameters in most RFI models. Optional parameters that are common to many models (but should not share the same name), such as `std` or `chance`, have been prefixed with the model name and an underscore (e.g. `dtv_chance`). Various other parameters have been renamed for consistency/clarity. Note that the `freq_min` and `freq_max` parameters for `rfi_dtv` have been replaced by the single parameter `dtv_band`, which is a tuple denoting the edges of the DTV band in GHz.

1.7.2 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`.

1.7.3 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

1.7.4 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.

1.7.5 v0.0.1

- Initial released version

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

h

INDEX

Symbols

A

B

C

D

F

G

H

I

J

L

M

P

R

S

T