hera_sim Documentation

HERA-Team

Jun 11, 2019

Contents

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

CHAPTER 1

Contents

1.1 Installation

1.1.1 Requirements

Requires:

- *numpy*
- *scipy*
- *aipy*
- *hera_cal* (which requires *h5py*)
- *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`.

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

[5]:
```
from hera_sim import foregrounds, noise, sigchain, rfi
```

[6]:
```
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 = 30.
```

Foregrounds

Diffuse Foregrounds

[7]:
```
Tsky_mdl = noise.HERA_Tsky_mdl['xx']
vis_fg_diffuse = foregrounds.diffuse_foreground(Tsky_mdl, lsts, fqs, bl_len_ns)
```

[8]:
```
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

[9]:
```
vis_fg_pntsrc = foregrounds.pntsrc_foreground(lsts, fqs, bl_len_ns, nsrcs=200)
```

[10]:
```
plt.figure()
plt.subplot(211); uvtools.plot.waterfall(vis_fg_pntsrc, mode='log', mx=MX, drng=DRNG);
↪ plt.colorbar()#; plt.ylim(0,4000)
plt.subplot(212); uvtools.plot.waterfall(vis_fg_pntsrc, mode='phs'); plt.colorbar();␣
↪plt.ylim(0,4000)
plt.show()
```

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

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

Diffuse and Point-Source Foregrounds

[11]:
```
vis_fg = vis_fg_diffuse + vis_fg_pntsrc
```

[12]:
```
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

[13]:
```
tsky = noise.resample_Tsky(fqs,lsts,Tsky_mdl=noise.HERA_Tsky_mdl['xx'])
t_rx = 150.
nos_jy = noise.sky_noise_jy(tsky + t_rx, fqs, lsts)
```

[14]:
```
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>
```

[16]:
```
vis_fg_nos = vis_fg + nos_jy
```

[17]:
```
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

[18]:
```
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
```

[19]:
```
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>
```

```
/home/steven/miniconda3/envs/hera_sim/lib/python2.7/site-packages/uvtools/plot.py:13:␣
↪RuntimeWarning: divide by zero encountered in log10
  data = np.log10(data)
```

[21]:
```
vis_fg_nos_rfi = vis_fg_nos + rfi_all
```

[22]:
```
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

[23]:
```
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>
```

[24]:
```
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

[25]:
```
xtalk = sigchain.gen_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>
```

1.3 API Reference

1.3.1 hera_sim

hera_sim.vis	
hera_sim.antpos	A module defining routines for creating antenna array configurations.
hera_sim.eor	A module containing functions for generating EoR-like signals.
hera_sim.foregrounds	A module with functions for generating foregrounds signals.
hera_sim.io	A module containing routines for interfacing data produced by *hera_sim* with other codes, especially UV-Data.
hera_sim.noise	A module for generating realistic HERA noise.
hera_sim.rfi	A module for generating realistic HERA RFI.
hera_sim.sigchain	A module for modeling HERA signal chains.
hera_sim.utils	Utility module

hera_sim.vis

Functions

aa_to_eq2tops(aa, jds)	
hmap_to_I(h)	
hmap_to_bm_cube(hmaps[, beam_px])	Convert healpix map to beam map cube.
hmap_to_crd_eq(h)	
sim_red_data(reds[, gains, shape, gain_scatter])	Simulate noise-free random but redundant (up to differing gains) visibilities.
vis_cpu(antpos, freq, eq2tops, crd_eq, ...)	Calculate visibility from an input intensity map and beam model.

hera_sim.vis.aa_to_eq2tops

hera_sim.vis.**aa_to_eq2tops**(*aa*, *jds*)

hera_sim.vis.hmap_to_I

hera_sim.vis.**hmap_to_I**(*h*)

hera_sim.vis.hmap_to_bm_cube

hera_sim.vis.**hmap_to_bm_cube**(*hmaps*, *beam_px=63*)
Convert healpix map to beam map cube.

Parameters

- **hmaps** (*list of 3D arrays*) – healpix maps for each antenna.
- **beam_px** (*int*) – number of pixels on a size for the beam map cube.

Returns *ndarray, shape[nant, beam_px, beam_px]* – the beam map cube.

hera_sim.vis.hmap_to_crd_eq

hera_sim.vis.**hmap_to_crd_eq**(*h*)

hera_sim.vis.sim_red_data

hera_sim.vis.**sim_red_data**(*reds*, *gains=None*, *shape=(10, 10)*, *gain_scatter=0.1*)
Simulate noise-free random but redundant (up to differing gains) visibilities.

Parameters

- **reds** (*list of list of tuples*) – list of lists of baseline-pol tuples where each sublist has only redundant pairs
- **gains** (*dict*) – pre-specify base gains to then scatter on top of in the {(index,antpol): ndarray} format. Default gives all ones.
- **shape** (*tuple*) – (Ntimes, Nfreqs).
- **gain_scatter** (*float*) – relative amplitude of per-antenna complex gain scatter

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

hera_sim.vis.vis_cpu

hera_sim.vis.**vis_cpu**(*antpos*, *freq*, *eq2tops*, *crd_eq*, *I_sky*, *bm_cube*, *real_dtype=<class 'numpy.float32'>*, *complex_dtype=<class 'numpy.complex64'>*)
Calculate visibility from an input intensity map and beam model.

Parameters

- **antpos (array_like, shape** – (NANT, 3)): antenna position array.
- **freq** (*float*) – frequency to evaluate the visibilities at [GHz].
- **eq2tops (array_like, shape** – (NTIMES, 3, 3)): Set of 3x3 transformation matrices converting equatorial coordinates to topocentric at each hour angle (and declination) in the dataset.
- **crd_eq (array_like, shape** – (3, NPIX)): equatorial coordinates of Healpix pixels.
- **I_sky (array_like, shape** – (NPIX,)): intensity distribution on the sky, stored as array of Healpix pixels.
- **bm_cube (array_like, shape** – (NANT, BM_PIX, BM_PIX)): beam maps for each antenna.

- **real_dtype, complex_dtype** (*dtype, optional*) – data type to use for real and complex-valued arrays.

Returns *array_like, shape(NTIMES, NANTS, NANTS)* – visibilities

hera_sim.antpos

A module defining routines for creating antenna array configurations. Input arguments for each function are arbitrary, but the return value is always a dictionary with keys representing antenna numbers, and values giving the 3D position of each antenna.

Functions

hex_array(hex_num[, sep, split_core, outriggers])	Build a hexagonal array configuration, nominally matching HERA's ideal configuration.
linear_array(nants[, sep])	Build a linear (east-west) array configuration.

hera_sim.antpos.hex_array

hera_sim.antpos.**hex_array**(*hex_num*, *sep=14.6*, *split_core=True*, *outriggers=2*)

Build a hexagonal array configuration, nominally matching HERA's ideal configuration.

Parameters

- **hex_num** (*int*) – the hexagon (radial) number of the core configuration. Number of core antennas returned is 3N^2 - 3N + 1.
- **sep** (*float*) – the separation between hexagonal grid points (meters).
- **split_core** (*bool*) – fractures the hexagonal core into tridrents that subdivide a hexagonal grid. Loses N antennas, so the number of core antennas returned is 3N^2 - 4N + 1.
- **outriggers** (*int*) – adds R extra rings of outriggers around the core that tile with the core to produce a fully-sampled UV plane. The first ring corresponds to the exterior of a hexNum=3 hexagon. Adds 3R^2 + 9R antennas.

Returns

dict –

a dictionary of antenna numbers and positions. Positions are x,y,z in topocentric coordinates, in meters.

hera_sim.antpos.linear_array

hera_sim.antpos.**linear_array**(*nants*, *sep=14.6*)

Build a linear (east-west) array configuration.

Parameters

- **nants** (*int*) – the number of antennas in the configuration.
- **sep** (*float*) – the separation between linearly spaced antennas (meters).

Returns

dict –

A dictionary of antenna numbers and positions. Positions are x,y,z in topocentric coordinates, in meters.

hera_sim.eor

A module containing functions for generating EoR-like signals.

Each model may take arbitrary parameters, but must return a 2D complex array containing the visibilities at the requested baseline, for the requested lsts and frequencies.

Functions

noiselike_eor(lsts, fqs, bl_vec[, eor_amp, ...])	Generate a noise-like, fringe-filtered EoR visibility.

hera_sim.eor.noiselike_eor

hera_sim.eor.**noiselike_eor**(*lsts*, *fqs*, *bl_vec*, *eor_amp=1e-05*, *min_delay=None*, *max_delay=None*, *fringe_filter_type='tophat'*, ***fringe_filter_kwargs*)

Generate a noise-like, fringe-filtered EoR visibility.

Parameters

- **lsts** (*ndarray*) – LSTs [radians]
- **fqs** (*ndarray*) – frequencies [GHz]
- **bl_vec** (*ndarray*) – East-North-Up (i.e. Topocentric) baseline vector in nanoseconds [East, North, Up]
- **eor_amp** (*float*) – amplitude of EoR signal [arbitrary units]
- **min_delay** (*float*) – minimum delay of signal to keep in nanosec (i.e. filter out below this delay)
- **max_delay** (*float*) – maximum delay of signal to keep in nanosec (i.e. filter out above this delay)
- **fringe_filter_type** (*str*) – type of fringe-rate filter, see utils.gen_fringe_filter()
- **fringe_filter_kwargs** – kwargs given fringe_filter_type, see utils.gen_fringe_filter()

Returns *vis (ndarray)* – simulated complex visibility

Notes

Based on the order of operations (delay filter then fringe-rate filter), modes outside of min and max delay will contain some spillover power due to the frequency-dependent nature of the fringe-rate filter.

hera_sim.foregrounds

A module with functions for generating foregrounds signals.

Each function may take arbitrary parameters, but should return a 2D array of visibilities for the requested baseline at the requested lsts and frequencies.

Functions

diffuse_foreground(lsts, fqs, bl_vec[, ...])	Produce a (NTIMES,NFREQS) mock-up of what diffuse foregrounds could look like on a baseline of a provided geometric length.
pntsrc_foreground(lsts, fqs, bl_vec[, ...])	Produce a (NTIMES,NFREQS) mock-up of what point-source foregrounds could look like on a baseline of a provided geometric length.

hera_sim.foregrounds.diffuse_foreground

hera_sim.foregrounds.**diffuse_foreground**(*lsts*, *fqs*, *bl_vec*, *Tsky_mdl=None*, *omega_p=None*, *standoff=0.0*, *delay_filter_type='tophat'*, *delay_filter_normalize=None*, *fringe_filter_type='tophat'*, ***fringe_filter_kwargs*)

Produce a (NTIMES,NFREQS) mock-up of what diffuse foregrounds could look like on a baseline of a provided geometric length.

Parameters

- **lsts** (*array-like*) – shape=(NTIMES,), radians local sidereal times of the observation to be generated.
- **fqs** (*array-like*) – shape=(NFREQS,), GHz the spectral frequencies of the observation to be generated.
- **bl_vec** (*array-like*) – shape=(3,), nanosec East-North-Up (i.e. Topocentric) baseline vector [East, North, Up]
- **Tsky_mdl** (*callable*) – interpolation object an interpolation object that returns the sky temperature as a function of (lst, freqs). Called as Tsky_mdl(lsts, fqs).
- **omega_p** (*array-like*) – shape=(NFREQS,) steradians Sky-integral of beam power. Default is to use noise.HERA_BEAM_POLY polynomial fit.
- **standoff** (*float*) – baseline horizon buffer [ns] for modeling suprahorizon emission
- **delay_filter_type** (*str*) – type of delay filter to use, see utils.gen_delay_filter
- **delay_filter_normalize** (*float*) – delay filter normalization, see utils.gen_delay_filter
- **fringe_filter_type** (*str*) – type of fringe-rate filter, see utils.gen_fringe_filter
- **fringe_filter_kwargs** – kwargs given fringe_filter_type, see utils.gen_fringe_filter

Returns

mdl (array-like) –

shape=(NTIMES,NFREQS) mock diffuse foreground visibility spectra vs. time

hera_sim.foregrounds.pntsrc_foreground

hera_sim.foregrounds.**pntsrc_foreground**(*lsts*, *fqs*, *bl_vec*, *nsrcs=1000*, *Smin=0.3*, *Smax=300*, *beta=-1.5*, *spectral_index_mean=-1*, *spectral_index_std=0.5*, *reference_freq=0.15*)

Produce a (NTIMES,NFREQS) mock-up of what point-source foregrounds could look like on a baseline of a

provided geometric length. Results have phase coherence within an observation but not between repeated calls of this function (i.e. no phase coherence between baselines). Beam width is currently hardcoded for HERA.

Parameters

- **lsts** (*array-like*) – shape=(NTIMES,), radians local sidereal times of the observation to be generated.
- **fqs** (*array-like*) – shape=(NFREQS,), GHz the spectral frequencies of the observation to be generated.
- **bl_vec** (*array-like*) – shape=(3,), nanosec East-North-Up (i.e. Topocentric) baseline vector [East, North, Up]
- **nsrcs** (*float*) – default=1000. the number of mock sources to put on the sky, drawn from a power-law of flux-densities with an index of beta. between Smin and Smax
- **Smin** (*float*) – [Jy], default=0.3 the minimum flux density to sample
- **Smax** (*float*) – [Jy], default=300 the maximum flux density to sample
- **beta** (*float*) – default=-1.5 the power-law index of point-source counts versus flux density
- **spectral_index_mean** (*float*) – default=-1 the mean spectral index of point sources drawn
- **spectral_index_std** (*float*) – default=0.5 the standard deviation of the spectral index of point sources drawn
- **reference_freq** (*float*) – [GHz], default=0.15 the frequency from which spectral indices extrapolate

Returns

vis (array-like) –

shape=(NTIMES,NFREQS) mock point-source foreground visibility spectra vs. time

hera_sim.io

A module containing routines for interfacing data produced by *hera_sim* with other codes, especially UVData.

Functions

empty_uvdata(nfreq, ntimes, ants[, ...])	Create an empty UVData object with valid metadata and zeroed data arrays with the correct dimensions.

hera_sim.io.empty_uvdata

hera_sim.io.**empty_uvdata**(*nfreq, ntimes, ants, antpairs=None, pols=['xx'], time_per_integ=10.7, min_freq=0.1, channel_bw=9.765625e-05, instrument='hera_sim', telescope_location=[5109342.82705015, 2005241.83929272, -3239939.40461961], telescope_lat_lon_alt=(-0.53619179912885, 0.3739944696510935, 1073.0000000074506), object_name='sim_data', start_jd=2458119.5, vis_units='uncalib'*)

Create an empty UVData object with valid metadata and zeroed data arrays with the correct dimensions.

Parameters

- **nfreq** (*int*) – number of frequency channels.

- **ntimes** (*int*) – number of LST bins.
- **ant** (*dict*) – antenna positions. The key should be an integer antenna ID, and the value should be a tuple of (x, y, z) positions in the units required by UVData.antenna_positions (meters, position relative to telescope_location). Example:

```
ants = {0 : (20., 20., 0.)}
```

- **antpairs** (*list of len-2 tuples*) – List of baselines as antenna pair tuples, e.g. `bls = [(1, 2), (3,4)]`. All antennas must be in the ants dict.
- **pols** (*list of str, optional*) – polarization strings.
- **time_per_integ** (*float, optional*) – Time per integration.
- **min_freq** (*float, optional*) – minimum frequency of the frequency array [GHz]
- **channel_bw** (*float, optional*) – frequency channel bandwidth [GHz].
- **instrument** (*str, optional*) – name of the instrument.
- **telescope_location** (*list of float, optional*) – location of the telescope, in default UVData coordinate system. Expects a list of length 3.
- **telescope_lat_lon_alt** (*tuple of float, optional*) – Latitude, longitude, and altitude of telescope, corresponding to the coordinates in telescope_location. Default: HERA_LAT_LON_ALT.
- **object_name** (*str, optional*) – name of UVData object
- **start_jd** (*float, optional*) – Julian date of the first time sample in the dataset.
- **vis_units** (*str, optional*) – assumed units of the visibility data.

Returns `pyuvdata.UVData` – A new UVData object containing valid metadata and blank (zeroed) arrays.

hera_sim.noise

A module for generating realistic HERA noise.

Functions

bm_poly_to_omega_p(fqs[, bm_poly])	Convert polynomial coefficients to beam area.
jy2T(fqs, omega_p)	Return [mK] / [Jy] for a beam size vs.
resample_Tsky(fqs, lsts[, Tsky_mdl, Tsky, ...])	Re-sample a model of the sky temperature at particular freqs and lsts.
sky_noise_jy(Tsky, fqs, lsts, omega_p[, B, ...])	Generate Gaussian noise (in Jy units) corresponding to a sky temperature model integrated for the specified integration time and bandwidth.
thermal_noise(fqs, lsts[, Tsky_mdl, Trx, ...])	Create thermal noise visibilities.
white_noise([size])	Produce complex Gaussian white noise with a variance of unity.

hera_sim.noise.bm_poly_to_omega_p

hera_sim.noise.**bm_poly_to_omega_p**(*fqs*, *bm_poly=array([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])*)

Convert polynomial coefficients to beam area.

Parameters

- **fqs** (*array-like*) – shape=(NFREQS,), GHz frequency array
- **bm_poly** (*polynomial*) – default=HERA_BEAM_POLY a polynomial fit to sky-integral, solid-angle beam size of observation as a function of frequency.

Returns

omega_p –

(array-like): shape=(NFREQS,), steradian sky-integral of peak-normalized beam power

hera_sim.noise.jy2T

hera_sim.noise.**jy2T**(*fqs*, *omega_p*)

Return [mK] / [Jy] for a beam size vs. frequency.

Arg:

fqs (array-like): shape=(NFREQS,), GHz the spectral frequencies of the observation to be generated.

omega_p (array-like): shape=(NFREQS,) steradians Sky-integral of beam power.

Returns

jy_to_mK (array-like) –

shape=(NFREQS,) a frequency-dependent scalar converting Jy to mK for the provided beam size.''

hera_sim.noise.resample_Tsky

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

Re-sample a model of the sky temperature at particular freqs and lsts.

Parameters

- **fqs** (*array-like*) – shape=(NFREQS,), GHz the spectral frequencies of the observation to be generated.
- **lsts** (*array-like*) – shape=(NTIMES,), radians local sidereal times of the observation to be generated.
- **Tsky_mdl** (*callable*) – interpolation object, default=None if provided, an interpolation object that returns the sky temperature as a function of (lst, freqs). Called as Tsky(lsts,fqs).
- **Tsky** (*float*) – Kelvin if Tsky_mdl not provided, an isotropic sky temperature corresponding to the provided mfreq.
- **mfreq** (*float*) – GHz the spectral frequency, in GHz, at which Tsky is specified
- **index** (*float*) – default=-2.5 the spectral index used to extrapolate Tsky to other frequencies

Returns

tsky (array-like) –

shape=(NTIMES,NFREQS) sky temperature vs. time and frequency

hera_sim.noise.sky_noise_jy

hera_sim.noise.**sky_noise_jy**(*Tsky*, *fqs*, *lsts*, *omega_p*, *B=None*, *inttime=10.7*)

Generate Gaussian noise (in Jy units) corresponding to a sky temperature model integrated for the specified integration time and bandwidth.

Parameters

- **Tsky** (*array-like*) – shape=(NTIMES,NFREQS), K the sky temperature at each time/frequency observation
- **fqs** (*array-like*) – shape=(NFREQS,), GHz the spectral frequencies of the observation
- **lsts** (*array-like*) – shape=(NTIMES,), radians local sidereal times of the observation
- **omega_p** (*array-like*) – shape=(NFREQS,) steradians Sky-integral of beam power.
- **B** (*float*) – default=None, GHz the channel width used to integrate noise. If not provided, defaults to the delta between fqs,
- **inttime** (*float*) – default=10.7, seconds the time used to integrate noise. If not provided, defaults to delta between lsts.

Returns

noise (array-like) –

shape=(NTIMES,NFREQS) complex Gaussian noise vs. time and frequency

hera_sim.noise.thermal_noise

hera_sim.noise.**thermal_noise**(*fqs*, *lsts*, *Tsky_mdl=None*, *Trx=0*, *omega_p=None*, *inttime=10.7*, ***kwargs*)

Create thermal noise visibilities.

Parameters

- **fqs** (*1d array*) – frequencies, in GHz.
- **lsts** (*1d array*) – times, in rad.
- **Tsky_mdl** (*callable, optional*) – a callable model, with signature `Tsky_mdl(lsts, fqs)`, which returns a 2D array of global beam-averaged sky temperatures (in K) as a function of LST and frequency.
- **Trx** (*float, optional*) – receiver temperature, in K.
- **omega_p** (*array-like*) – shape=(NFREQS,) steradians Sky-integral of beam power. Default is to use noise.HERA_BEAM_POLY to create omega_p.
- **inttime** (*float, optional*) – the integration time, in sec.
- ****kwargs** – passed to *resample_Tsky()*.

Returns *2d array size(lsts, fqs)* – the thermal visibilities [Jy].

hera_sim.noise.white_noise

hera_sim.noise.**white_noise**(*size=1*)

Produce complex Gaussian white noise with a variance of unity.

Parameters **size** (*int or tuple, optional*) – shape of output samples.

Returns

noise (ndarray) –

shape=size random white noise realization

hera_sim.rfi

A module for generating realistic HERA RFI.

Functions

rfi_dtv(fqs, lsts[, rfi, freq_min, ...])	Generate an (NTIMES, NFREQS) waterfall containing Digital TV RFI.
rfi_impulse(fqs, lsts[, rfi, chance, strength])	Generate an (NTIMES,NFREQS) waterfall containing RFI impulses that are localized in time but span the frequency band.
rfi_scatter(fqs, lsts[, rfi, chance, ...])	Generate an (NTIMES,NFREQS) waterfall containing RFI impulses that are localized in time but span the frequency band.
rfi_stations(fqs, lsts[, stations, rfi])	Generate an (NTIMES,NFREQS) waterfall containing RFI stations that are localized in frequency.

hera_sim.rfi.rfi_dtv

hera_sim.rfi.**rfi_dtv**(*fqs*, *lsts*, *rfi=None*, *freq_min=0.174*, *freq_max=0.214*, *width=0.008*, *chance=0.0001*, *strength=10*, *strength_std=10*)

Generate an (NTIMES, NFREQS) waterfall containing Digital TV RFI.

DTV RFI is expected to be of uniform bandwidth (eg. 8MHz), in contiguous bands, in a nominal frequency range. Furthermore, it is expected to be short-duration, and so is implemented as randomly affecting discrete LSTS.

There may be evidence that close times are correlated in having DTV RFI, and this is *not currently implemented*.

Parameters

- **fqs** (*array-like*) – shape=(NFREQS,), GHz the spectral frequencies of the waterfall to be generated.
- **lsts** (*array-like*) – shape=(NTIMES,), radians local sidereal times of the waterfall to be generated.
- **rfi** (*array-like*) – shape=(NTIMES,NFREQS), default=None an array to which the RFI will be added. If None, a new array is generated.
- **freq_min, freq_max** (*float*) – the min and max frequencies of the full DTV band [GHz]
- **width** (*float*) – Width of individual DTV bands [GHz]

- **chance** (*float*) – default=0.0001 the probability that a time/freq bin will be assigned an RFI impulse
- **strength** (*float*) – Jy, default=10 the average amplitude of the spike generated in each time/freq bin
- **strength_std** (*float*) – Jy, default = 10 the standard deviation of the amplitudes drawn for each time/freq bin

Returns

rfi (array-like) –

shape=(NTIMES,NFREQS) a waterfall containing RFI

hera_sim.rfi.rfi_impulse

hera_sim.rfi.**rfi_impulse**(*fqs*, *lsts*, *rfi=None*, *chance=0.001*, *strength=20.0*)

Generate an (NTIMES,NFREQS) waterfall containing RFI impulses that are localized in time but span the frequency band.

Parameters

- **fqs** (*array-like*) – shape=(NFREQS,), GHz the spectral frequencies of the waterfall to be generated.
- **lsts** (*array-like*) – shape=(NTIMES,), radians local sidereal times of the waterfall to be generated.
- **rfi** (*array-like*) – shape=(NTIMES,NFREQS), default=None an array to which the RFI will be added. If None, a new array is generated.
- **chance** (*float*) – the probability that a time bin will be assigned an RFI impulse
- **strength** (*float*) – Jy the strength of the impulse generated in each time/freq bin

Returns

rfi (array-like) –

shape=(NTIMES,NFREQS) a waterfall containing RFI' '

hera_sim.rfi.rfi_scatter

hera_sim.rfi.**rfi_scatter**(*fqs*, *lsts*, *rfi=None*, *chance=0.0001*, *strength=10*, *std=10*)

Generate an (NTIMES,NFREQS) waterfall containing RFI impulses that are localized in time but span the frequency band.

Parameters

- **fqs** (*array-like*) – shape=(NFREQS,), GHz the spectral frequencies of the waterfall to be generated.
- **lsts** (*array-like*) – shape=(NTIMES,), radians local sidereal times of the waterfall to be generated.
- **rfi** (*array-like*) – shape=(NTIMES,NFREQS), default=None an array to which the RFI will be added. If None, a new array is generated.
- **chance** (*float*) – default=0.0001 the probability that a time/freq bin will be assigned an RFI impulse

- **strength** (*float*) – Jy, default=10 the average amplitude of the spike generated in each time/freq bin
- **std** (*float*) – Jy, default = 10 the standard deviation of the amplitudes drawn for each time/freq bin

Returns

rfi (array-like) –

shape=(NTIMES,NFREQS) a waterfall containing RFI

hera_sim.rfi.rfi_stations

hera_sim.rfi.**rfi_stations**(*fqs, lsts, stations=[(0.1007, 1.0, 100000.0, 10.0, 100.0), (0.1016, 1.0, 100000.0, 10.0, 100.0), (0.1024, 1.0, 100000.0, 10.0, 100.0), (0.1028, 1.0, 3000.0, 1.0, 100.0), (0.1043, 1.0, 100000.0, 10.0, 100.0), (0.105, 1.0, 10000.0, 3.0, 100.0), (0.1052, 1.0, 100000.0, 10.0, 100.0), (0.1061, 1.0, 100000.0, 10.0, 100.0), (0.1064, 1.0, 10000.0, 3.0, 100.0), (0.1371, 0.2, 100000.0, 3.0, 600.0), (0.1372, 0.2, 100000.0, 3.0, 600.0), (0.1373, 0.2, 100000.0, 3.0, 600.0), (0.1374, 0.2, 100000.0, 3.0, 600.0), (0.1375, 0.2, 100000.0, 3.0, 600.0), (0.1831, 1.0, 100000.0, 30.0, 1000), (0.1891, 1.0, 2000.0, 1.0, 1000), (0.1911, 1.0, 100000.0, 30.0, 1000), (0.1972, 1.0, 100000.0, 30.0, 1000)], rfi=None*)

Generate an (NTIMES,NFREQS) waterfall containing RFI stations that are localized in frequency.

Parameters

- **lsts** (*array-like*) – shape=(NTIMES,), radians local sidereal times of the waterfall to be generated.
- **fqs** (*array-like*) – shape=(NFREQS,), GHz the spectral frequencies of the waterfall to be generated.
- **stations** (*iterable*) – list of 5-tuples, default=HERA_RFI_STATIONS a list of (FREQ, DUTY_CYCLE, STRENGTH, STD, TIMESCALE) tuples for RfiStations that will be injected into waterfall. Instead of a tuple, an instance of *RfiStation* may be given.
- **rfi** (*array-like*) – shape=(NTIMES,NFREQS), default=None an array to which the RFI will be added. If None, a new array is generated.

Returns

rfi (array-like) –

shape=(NTIMES,NFREQS) a waterfall containing RFI

Classes

RfiStation(fq0[, duty_cycle, strength, std, ...])	Class for representing an RFI transmitter.

hera_sim.rfi.RfiStation

class hera_sim.rfi.**RfiStation**(*fq0*, *duty_cycle=1.0*, *strength=100.0*, *std=10.0*, *timescale=100.0*)

Class for representing an RFI transmitter.

Parameters

- **fq0** (*float*) – GHz center frequency of the RFI transmitter
- **duty_cycle** (*float*) – default=1. fraction of times that RFI transmitter is on
- **strength** (*float*) – Jy, default=100 amplitude of RFI transmitter
- **std** (*float*) – default=10. standard deviation of transmission amplitude
- **timescale** (*scalar*) – seconds, default=100. timescale for sinusoidal variation in transmission amplitude'‘'

Methods

__init__(fq0[, duty_cycle, strength, std, ...])	Initialize self.
gen_rfi(fqs, lsts[, rfi])	Generate an (NTIMES,NFREQS) waterfall containing RFI.

hera_sim.rfi.RfiStation.__init__

RfiStation.**__init__**(*fq0*, *duty_cycle=1.0*, *strength=100.0*, *std=10.0*, *timescale=100.0*)
Initialize self. See help(type(self)) for accurate signature.

hera_sim.rfi.RfiStation.gen_rfi

RfiStation.**gen_rfi**(*fqs*, *lsts*, *rfi=None*)
Generate an (NTIMES,NFREQS) waterfall containing RFI.

Parameters

- **lsts** (*array-like*) – shape=(NTIMES,), radians local sidereal times of the waterfall to be generated.
- **fqs** (*array-like*) – shape=(NFREQS,), GHz the spectral frequencies of the waterfall to be generated.
- **rfi** (*array-like*) – shape=(NTIMES,NFREQS), default=None an array to which the RFI will be added. If None, a new array is generated.

Returns

rfi (array-like) –

shape=(NTIMES,NFREQS) a waterfall containing RFI

hera_sim.sigchain

A module for modeling HERA signal chains.

Functions

apply_gains(vis, gains, bl)	Apply to a (NTIMES,NFREQS) visibility waterfall the bandpass functions for its constituent antennas.

Continued on next page

Table 11 – continued from previous page

apply_xtalk(vis, xtalk)	Apply to a (NTIMES,NFREQS) visibility waterfall a crosstalk signal
gen_bandpass(fqs, ants[, gain_spread])	Produce a set of mock bandpass gains with variation based around the HERA_NRAO_BANDPASS model.
gen_cross_coupling_xtalk(fqs, autovis[, ...])	Generate a cross coupling systematic (e.g.
gen_delay_phs(fqs, ants[, dly_rng])	Produce a set of mock complex phasors corresponding to cables delays.
gen_gains(fqs, ants[, gain_spread, dly_rng])	Produce a set of mock bandpasses perturbed around a HERA_NRAO_BANDPASS model and complex phasors corresponding to cables delays.
gen_reflection_coefficient(fqs, amp, dly, phs)	Generate a reflection coefficient.
gen_reflection_gains(fqs, ants[, amp, dly, ...])	Generate a signal chain reflection as an antenna gain.
gen_whitenoise_xtalk(fqs[, amplitude])	Generate a white-noise cross-talk model for specified bls.

hera_sim.sigchain.apply_gains

hera_sim.sigchain.**apply_gains**(*vis*, *gains*, *bl*)

Apply to a (NTIMES,NFREQS) visibility waterfall the bandpass functions for its constituent antennas.

Parameters

- **vis** (*array-like*) – shape=(NTIMES,NFREQS) the visibility waterfall to which gains will be applied
- **gains** (*dictionary*) – a dictionary of antenna numbers as keys and complex gain ndarrays as values (e.g. output of *gen_gains()*) with shape as either (NTIMES,NFREQS) or (NFREQS,)
- **bl** (*2-tuple*) – a (i, j) tuple representing the baseline corresponding to this visibility. g_i * g_j.conj() will be multiplied into vis.

Returns

vis (array-like) –

shape=(NTIMES,NFREQS) the visibility waterfall with gains applied, unless antennas in bl don't exist in gains, then input vis is returned

hera_sim.sigchain.apply_xtalk

hera_sim.sigchain.**apply_xtalk**(*vis*, *xtalk*)

Apply to a (NTIMES,NFREQS) visibility waterfall a crosstalk signal

Parameters

- **vis** (*array-like*) – shape=(NTIMES,NFREQS) the visibility waterfall to which gains will be applied
- **xtalk** (*array-like*) – shape=(NTIMES,NFREQS) or (NFREQS,) the crosstalk signal to be applied.

Returns

vis (array-like) –

shape=(NTIMES,NFREQS) the visibility waterfall with crosstalk injected

hera_sim.sigchain.gen_bandpass

hera_sim.sigchain.**gen_bandpass**(*fqs*, *ants*, *gain_spread=0.1*)

Produce a set of mock bandpass gains with variation based around the HERA_NRAO_BANDPASS model.

Parameters

- **fqs** (*array-like*) – shape=(NFREQS,), GHz the spectral frequencies of the bandpasses
- **ants** (*iterable*) – the indices/names of the antennas
- **gain_spread** (*float*) – default=0.1 the fractional variation in gain harmonics

Returns *g (dictionary)* – a dictionary of ant:bandpass pairs where keys are elements of ants and bandpasses are complex arrays with shape (NFREQS,)

See also:

gen_gains(): uses this function to generate full gains.

hera_sim.sigchain.gen_cross_coupling_xtalk

hera_sim.sigchain.**gen_cross_coupling_xtalk**(*fqs*, *autovis*, *amp=None*, *dly=None*, *phs=None*, *conj=False*)

Generate a cross coupling systematic (e.g. crosstalk).

A cross coupling systematic is the auto-correlation visibility multiplied by a coupling coefficient. If V_{11} is the auto-correlation visibility of antenna 1, and ϵ_{12} is the coupling coefficient, then cross correlation visibility takes the form

$$V_{12} = v_1 v_2^* + V_{11}\epsilon_{12}^*$$

where ϵ_{12} is modeled as a reflection coefficient constructed as

$Args : fqs(1Dndarray) : frequencies[GHz]autovis(2Dndarray) : auto - correlationvisibilityndarrayofshape(Ntim$

Returns *2D ndarray* – xtalk model of shape (Ntimes, Nfreqs)

hera_sim.sigchain.gen_delay_phs

hera_sim.sigchain.**gen_delay_phs**(*fqs*, *ants*, *dly_rng=(-20, 20)*)

Produce a set of mock complex phasors corresponding to cables delays.

Parameters

- **fqs** (*array-like*) – shape=(NFREQS,), GHz the spectral frequencies of the bandpasses
- **ants** (*iterable*) – the indices/names of the antennas
- **dly_range** (*2-tuple*) – ns the range of the delay

Returns *g (dictionary)* – a dictionary of ant:exp(2pi*i*tau*fqs) pairs where keys are elements of ants and values are complex arrays with shape (NFREQS,)

See also:

gen_gains(): uses this function to generate full gains.

hera_sim.sigchain.gen_gains

hera_sim.sigchain.**gen_gains**(*fqs*, *ants*, *gain_spread=0.1*, *dly_rng=(-20, 20)*)

Produce a set of mock bandpasses perturbed around a HERA_NRAO_BANDPASS model and complex phasors corresponding to cables delays.

Parameters

- **fqs** (*array-like*) – shape=(NFREQS,), GHz the spectral frequencies of the bandpasses
- **ants** (*iterable*) – the indices/names of the antennas
- **gain_spread** (*float*) – default=0.1 the fractional variation in gain harmonics
- **dly_range** (*2-tuple*) – ns the range of the delay

Returns *g (dictionary)* – a dictionary of ant:bandpass * exp(2pi*i*tau*fqs) pairs where keys are elements of ants and bandpasses are complex arrays with shape (NFREQS,)

See also:

apply_gains(): apply gains from this function to a visibility

hera_sim.sigchain.gen_reflection_coefficient

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

Generate a reflection coefficient.

The reflection coefficient is described as

$$\epsilon = A * \exp(2i\pi\tau\nu + i\phi)$$

Parameters

- **fqs** (*1D ndarray*) – frequencies [GHz]
- **amp** (*float or ndarray*) – reflection amplitude
- **dly** (*float or ndarray*) – reflection delay [nanosec]
- **phs** (*float or ndarray*) – reflection phase [radian]
- **conj** (*bool, optional*) – if True, conjugate the reflection coefficient

Returns *complex ndarray* – complex reflection gain

Notes

Reflection terms can be fed as floats, in which case output coefficient is a 1D array of shape (Nfreqs,) or they can be fed as (Ntimes, 1) ndarrays, in which case output coefficient is a 2D narray of shape (Ntimes, Nfreqs)

hera_sim.sigchain.gen_reflection_gains

hera_sim.sigchain.**gen_reflection_gains**(*fqs*, *ants*, *amp=None*, *dly=None*, *phs=None*, *conj=False*)

Generate a signal chain reflection as an antenna gain.

A signal chain reflection is a copy of an antenna voltage stream at a boosted delay, and can be incorporated via a gain term

$$g_1 = (1 + \epsilon_{11})$$

where ϵ_{11} is antenna 1's reflection coefficient which can be constructed as

$$\epsilon_{11} = A_{11} * \exp(2i\pi\tau_{11}\nu + i\phi_{11})$$

Parameters

- **fqs** (*1D ndarray*) – frequencies [GHz]
- **ants** (*list of integers*) – antenna numbers
- **amp** (*list, optional*) – antenna reflection amplitudes for each antenna. Default is 1.0
- **dly** (*list, optional*) – antenna reflection delays [nanosec]. Default is 0.0
- **phs** (*lists, optional*) – antenna reflection phases [radian]. Default is 0.0
- **conj** (*bool, optional*) – if True, conjugate the reflection coefficients

Returns *dictionary* – keys are antenna numbers and values are complex reflection gains

Notes

Reflection terms for each antenna can be fed as a list of floats, in which case the output coefficients are 1D arrays of shape (Nfreqs,) or they can be fed as a list of ndarrays of shape (Ntimes, 1), in which case output coefficients are 2D narrays of shape (Ntimes, Nfreqs)

hera_sim.sigchain.gen_whitenoise_xtalk

hera_sim.sigchain.**gen_whitenoise_xtalk** (*fqs*, *amplitude=3.0*)

Generate a white-noise cross-talk model for specified bls.

Parameters

- **fqs** (*ndarray*) – frequencies of observation [GHz]
- **amplitude** (*float*) – amplitude of cross-talk in visibility units

Returns *1D ndarray* – xtalk model across frequencies

See also:

apply_xtalk(): apply the output of this function to a visibility.

hera_sim.utils

Utility module

Functions

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.
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.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.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*, *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*, *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] 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 ...

1.7 Changelog

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

h

Index

Symbols

A

B

C

D

E

G

H

J

L

N

P

R

S

T

V

W