

PyREx Documentation

Release 1.9.0

Ben Hokanson-Fasig

Aug 27, 2019

CONTENTS

1	About PyREx	1
1.1	Installation	1
1.2	Quick Code Example	1
1.3	Units	2
2	How to Use PyREx	3
2.1	Working with Signal Objects	3
2.2	Antenna Class and Subclasses	12
2.3	AntennaSystem and Detector Classes	18
2.4	Ice and Earth Models	23
2.5	Ray Tracing	24
2.6	Particle Generation	26
2.7	Full Simulation	27
2.8	Data File I/O	29
2.9	More Examples	32
3	Custom Sub-Package	33
3.1	Build Your Own Custom Module	34
4	Example Code	38
4.1	Plot Detector Geometry	38
4.2	Askaryan Frequency Content	40
4.3	Calculate Effective Area	41
4.4	Examine a Single Event	43
5	Contributing to PyREx	46
5.1	Branching Model	46
5.2	Contributing via Pull Request	47
5.3	Contributing with Direct Access	47
5.4	Releasing a New Version	48
6	PyREx API	49
6.1	PyREx Package Imports	49
6.2	Individual Module APIs	68
6.3	Included Custom Sub-Packages	120
7	Version History	182
7.1	Version 1.9.0	182
7.2	Version 1.8.2	183
7.3	Version 1.8.1	183
7.4	Version 1.8.0	184

7.5	Version 1.7.0	185
7.6	Version 1.6.0	185
7.7	Version 1.5.0	186
7.8	Version 1.4.2	187
7.9	Version 1.4.1	187
7.10	Version 1.4.0	187
7.11	Version 1.3.1	187
7.12	Version 1.3.0	188
7.13	Version 1.2.1	188
7.14	Version 1.2.0	188
7.15	Version 1.1.2	189
7.16	Version 1.1.1	189
7.17	Version 1.1.0	189
7.18	Version 1.0.3	190
7.19	Version 1.0.2	190
7.20	Version 1.0.1	190
7.21	Version 1.0.0	191
7.22	Version 0.0.0	191
8	GitHub README	192
8.1	PyREx - (Python package for Radio Experiments)	192
	Bibliography	194
	Python Module Index	196
	Index	197

ABOUT PYREX

PyREx (**P**ython package for **R**adio **E**xperiments) is a Python package designed to simulate the measurement of Askaryan pulses via in-ice radio antenna arrays. The code was written for the ARA collaboration with considerations for future radio arrays. As such, the package is designed to be highly modular so that it can easily be used for other radio projects (e.g. ARIANNA, RNO, and IceCube Gen2).

1.1 Installation

The easiest way to get the PyREx package is using `pip` as follows:

```
pip install git+https://github.com/bhokansonfasig/pyrex#egg=pyrex
```

PyREx requires python version 3.6+ as well as numpy version 1.13+, scipy version 0.19+, and h5py version 2.7+, which should be automatically installed when installing via `pip`.

Alternatively, you can download the code from <https://github.com/bhokansonfasig/pyrex/> and then either include the `pyrex` directory (the one containing the python modules) in your `PYTHON_PATH`, or just copy the `pyrex` directory into your working directory. PyREx is not currently available on PyPI, so a simple `pip install pyrex` will not have the intended effect.

1.2 Quick Code Example

The most basic simulation can be produced as follows:

First, import the package:

```
import pyrex
```

Then, create a particle generator object that will produce random particles in a cylinder with radius and depth of 1 km and with a fixed energy of 100 PeV:

```
particle_generator = pyrex.CylindricalGenerator(dr=1000, dz=1000, energy=1e8)
```

An array of antennas that represent the detector is also needed. The base `Antenna` class provides a basic antenna with a flat frequency response and no trigger condition. Here we make a single vertical “string” of four antennas with no noise:

```
antenna_array = []
for z in [-100, -150, -200, -250]:
    antenna_array.append(
```

(continues on next page)

(continued from previous page)

```
pyrex.Antenna(position=(0,0,z), noisy=False)
)
```

Finally, we want to pass these into the *EventKernel* and produce an event:

```
kernel = pyrex.EventKernel(generator=particle_generator,
                           antennas=antenna_array)
kernel.event()
```

Now the signals received by each antenna can be accessed by their *waveforms* parameter:

```
import matplotlib.pyplot as plt
for ant in kernel.ant_array:
    for wave in ant.waveforms:
        plt.figure()
        plt.plot(wave.times, wave.values)
        plt.show()
```

1.3 Units

For ease of use, PyREx tries to use consistent units in all classes and functions. The units used are mostly SI with a few exceptions listed in bold below:

Metric	Unit
time	seconds (s)
frequency	hertz (Hz)
distance	meters (m)
density	grams per cubic centimeter (g/cm³)
material thickness	grams per square centimeter (g/cm²)
temperature	kelvin (K)
energy	gigaelectronvolts (GeV)
resistance	ohms
voltage	volts (V)
electric field	volts per meter (V/m)

HOW TO USE PYREX

This section describes in detail how to use a majority of the functions and classes included in the base PyREx package, along with short example code segments. The code in each section is designed to run sequentially, and the code examples all assume these imports:

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.fftpack
import scipy.signal
import pyrex
```

All of the following examples can also be found (and easily run) in the Code Examples python notebook found in the examples directory.

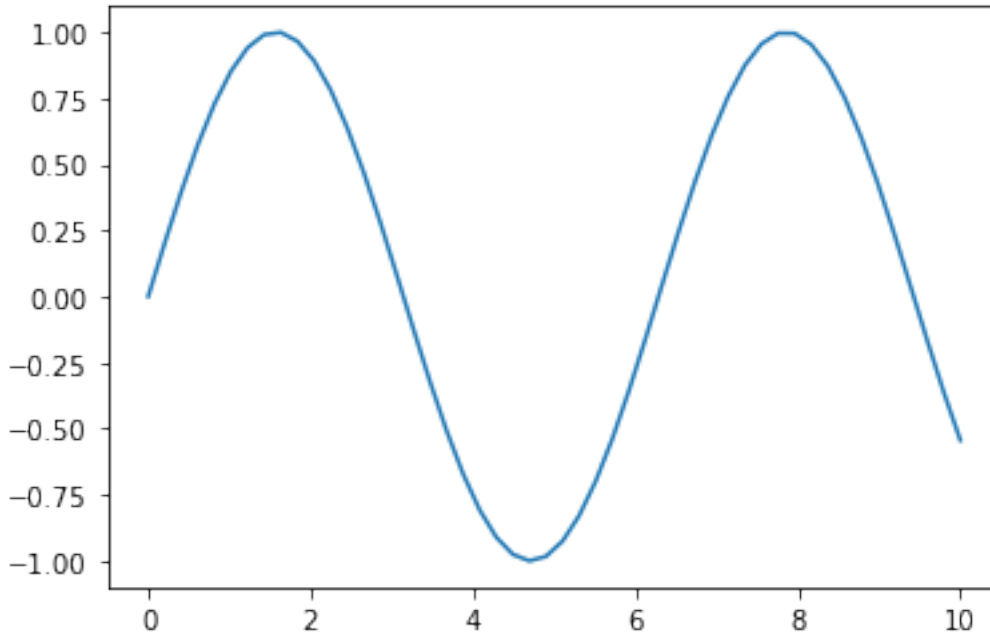
2.1 Working with Signal Objects

The base *Signal* class consists of an array of times and an array of corresponding signal values, and is instantiated with these two arrays. The `times` array is assumed to be in units of seconds, but there are no general units for the `values` array. It is worth noting that the *Signal* object stores shallow copies of the passed arrays, so changing the original arrays will not affect the *Signal* object.

```
time_array = np.linspace(0, 10)
value_array = np.sin(time_array)
my_signal = pyrex.Signal(times=time_array, values=value_array)
```

Plotting the *Signal* object is as simple as plotting the times vs the values:

```
plt.plot(my_signal.times, my_signal.values)
plt.show()
```

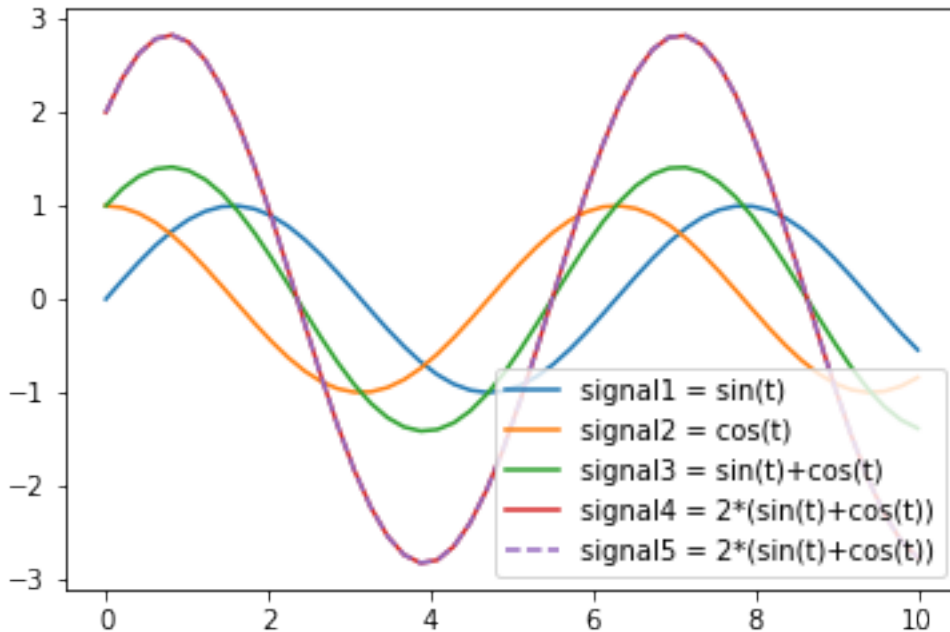


While there are no specified units for `Signal.values`, there is the option to specify the `value_type` of the values. This is done using the `Signal.Type` enum. By default, a `Signal` object has `value_type=Type.unknown`. However, if the signal represents a voltage, electric field, or power; `value_type` can be set to `Signal.Type.voltage`, `Signal.Type.field`, or `Signal.Type.power` respectively:

```
my_voltage_signal = pyrex.Signal(times=time_array, values=value_array,
                                  value_type=pyrex.Signal.Type.voltage)
```

`Signal` objects can be added as long as they have the same time array and `value_type`. `Signal` objects can also be multiplied by numeric types, which will multiply the values attribute of the signal.

```
time_array = np.linspace(0, 10)
values1 = np.sin(time_array)
values2 = np.cos(time_array)
signal1 = pyrex.Signal(time_array, values1)
plt.plot(signal1.times, signal1.values,
         label="signal1 = sin(t)")
signal2 = pyrex.Signal(time_array, values2)
plt.plot(signal2.times, signal2.values,
         label="signal2 = cos(t)")
signal3 = signal1 + signal2
plt.plot(signal3.times, signal3.values,
         label="signal3 = sin(t)+cos(t)")
signal4 = 2 * signal3
plt.plot(signal4.times, signal4.values,
         label="signal4 = 2*(sin(t)+cos(t))")
all_signals = [signal1, signal2, signal3]
signal5 = sum(all_signals)
plt.plot(signal5.times, signal5.values, '--',
         label="signal5 = 2*(sin(t)+cos(t))")
plt.legend()
plt.show()
```

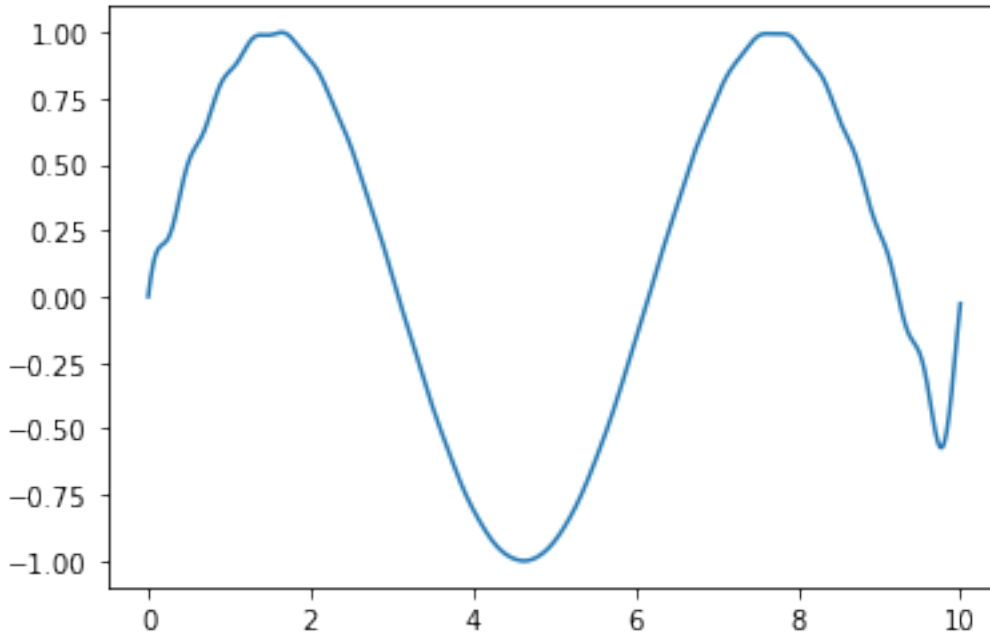


The `Signal` class provides many convenience attributes for dealing with signals:

```
my_signal.dt == my_signal.times[1] - my_signal.times[0]
my_signal.spectrum == scipy.fftpack.fft(my_signal.values)
my_signal.frequencies == scipy.fftpack.fftfreq(n=len(my_signal.values),
                                              d=my_signal.dt)
my_signal.envelope == np.abs(scipy.signal.hilbert(my_signal.values))
```

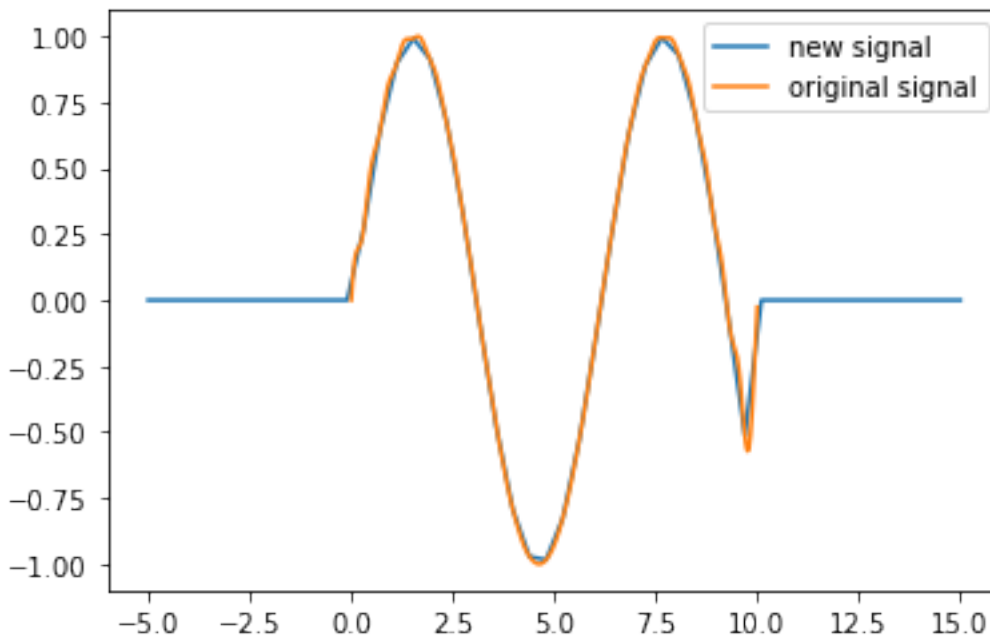
The `Signal` class also provides functions for manipulating the signal. The `Signal.resample()` method will resample the times and values arrays to the given number of points (with the same endpoints):

```
my_signal.resample(1001)
len(my_signal.times) == len(my_signal.values) == 1001
my_signal.times[0] == 0
my_signal.times[-1] == 10
plt.plot(my_signal.times, my_signal.values)
plt.show()
```

The `Signal.with_times()` method will interpolate/extrapolate the signal's values onto a new times array:

```
new_times = np.linspace(-5, 15)
new_signal = my_signal.with_times(new_times)
plt.plot(new_signal.times, new_signal.values, label="new signal")
plt.plot(my_signal.times, my_signal.values, label="original signal")
plt.legend()
plt.show()
```



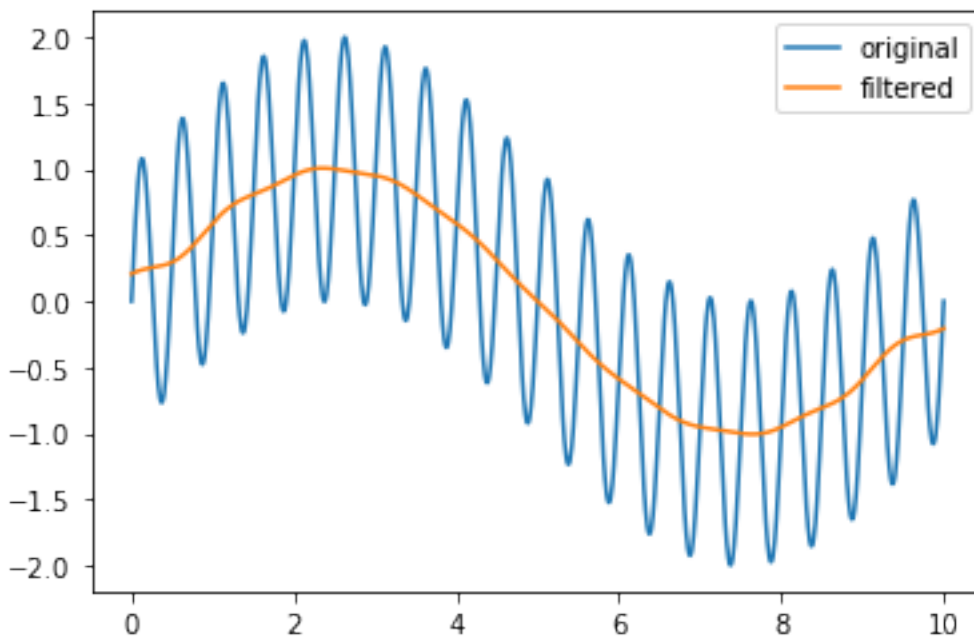
The `Signal.filter_frequencies()` method will apply a frequency-domain filter to the values array based on the passed frequency response function. In cases where the filter is designed for only positive frequencies (as below) the filtered frequency may exhibit strange behavior, including potentially having an imaginary part. To resolve that

issue, pass `force_real=True` to the `Signal.filter_frequencies()` method which will extrapolate the given filter to negative frequencies and ensure a real-valued filtered signal.

```
def lowpass_filter(frequency):
    if frequency < 1:
        return 1
    else:
        return 0

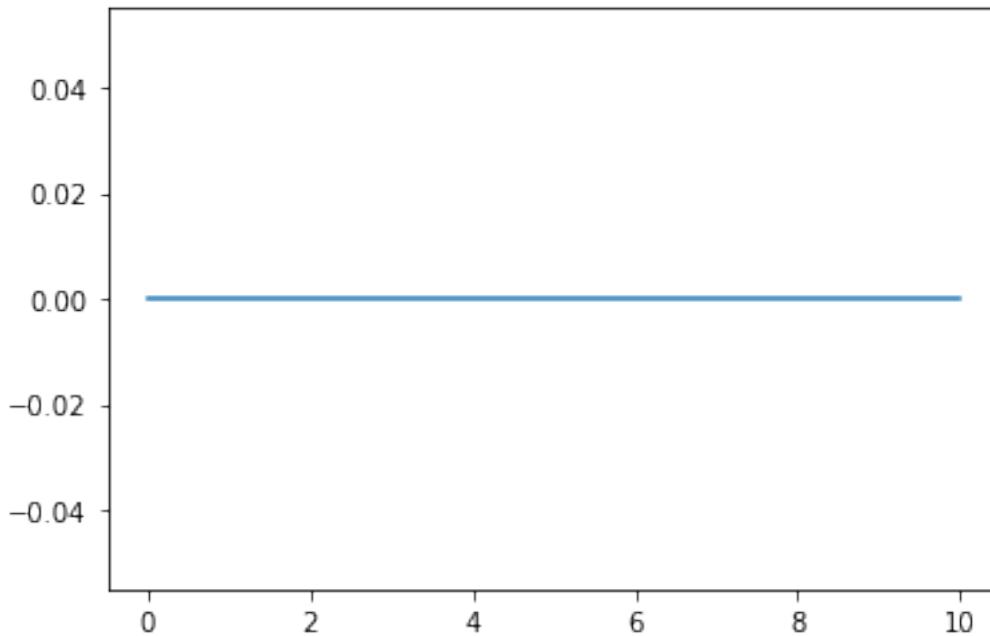
time_array = np.linspace(0, 10, 1001)
value_array = np.sin(0.1*2*np.pi*time_array) + np.sin(2*2*np.pi*time_array)
my_signal = pyrex.Signal(times=time_array, values=value_array)

plt.plot(my_signal.times, my_signal.values, label="original")
my_signal.filter_frequencies(lowpass_filter, force_real=True)
plt.plot(my_signal.times, my_signal.values, label="filtered")
plt.legend()
plt.show()
```



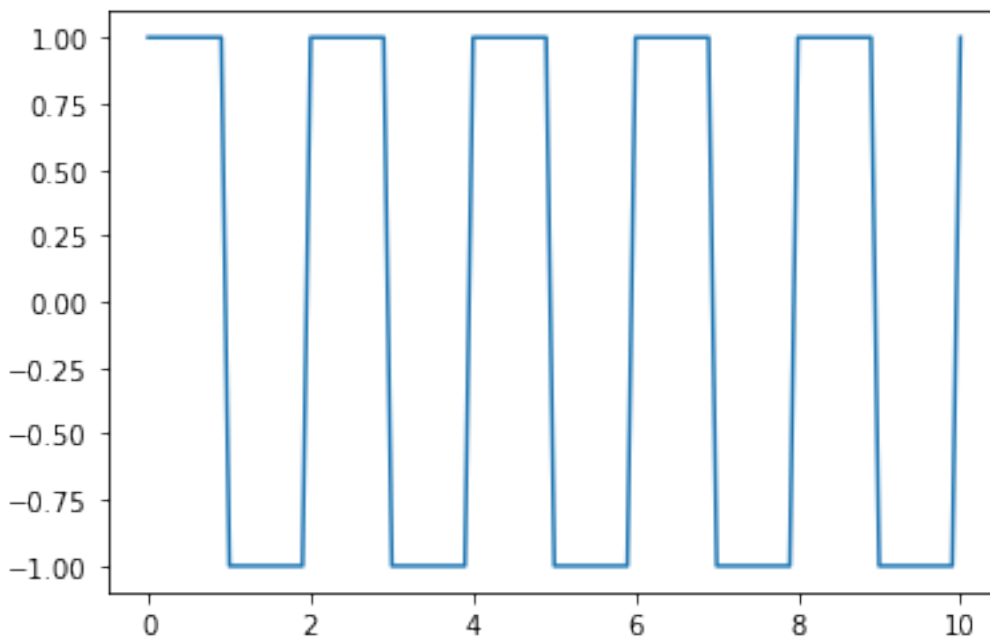
A number of classes which inherit from the `Signal` class are included in PyREx: `EmptySignal`, `FunctionSignal`, `AskaryanSignal`, and `ThermalNoise`. `EmptySignal` is simply a signal whose values are all zero:

```
time_array = np.linspace(0,10)
empty = pyrex.EmptySignal(times=time_array)
plt.plot(empty.times, empty.values)
plt.show()
```



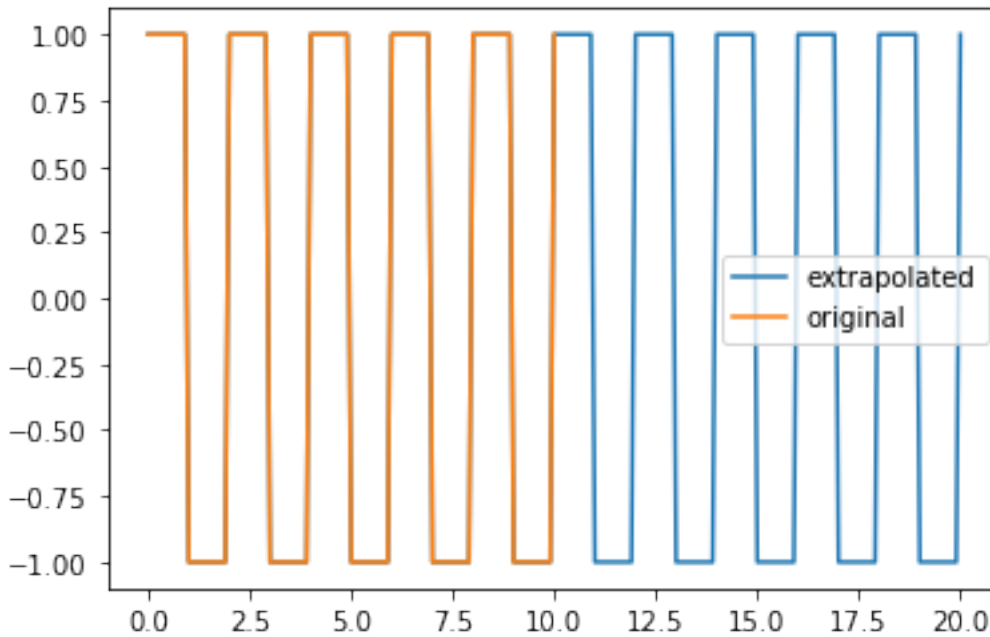
FunctionSignal takes a function of time and creates a signal based on that function:

```
time_array = np.linspace(0, 10, num=101)
def square_wave(time):
    if int(time)%2==0:
        return 1
    else:
        return -1
square_signal = pyrex.FunctionSignal(times=time_array, function=square_wave)
plt.plot(square_signal.times, square_signal.values)
plt.show()
```



Additionally, *FunctionSignal* leverages its knowledge of the function to more accurately interpolate and extrapolate values for the `Signal.with_times()` method:

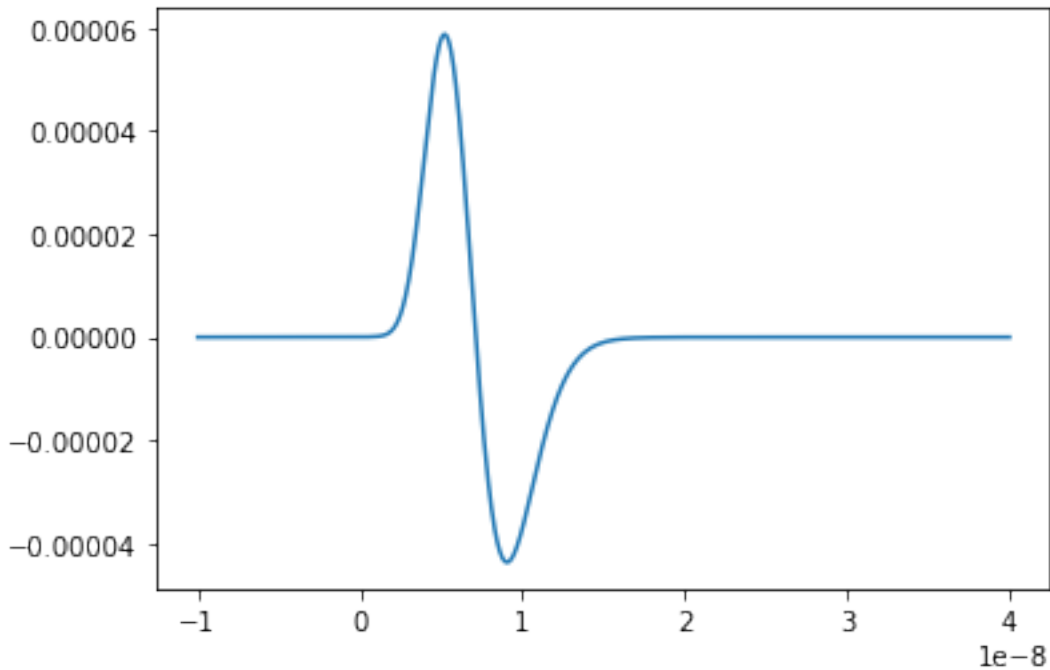
```
new_times = np.linspace(0, 20, num=201)
long_square_signal = square_signal.with_times(new_times)
plt.plot(long_square_signal.times, long_square_signal.values, label="extrapolated")
plt.plot(square_signal.times, square_signal.values, label="original")
plt.legend()
plt.show()
```



AskaryanSignal produces an Askaryan pulse (in V/m) on a time array resulting from a given neutrino observed at a given angle from the shower axis and at a given distance from the shower vertex. For more about using the *Particle* class, see *Particle Generation*.

```
time_array = np.linspace(-10e-9, 40e-9, 1001)
neutrino_energy = 1e8 # GeV
neutrino = pyrex.Particle("nu_e", vertex=(0, 0, -1000), direction=(0, 0, -1),
                          energy=neutrino_energy)
neutrino.interaction.em_frac = 1
neutrino.interaction.had_frac = 0
observation_angle = 45 * np.pi/180 # radians
observation_distance = 2000 # meters
askaryan = pyrex.AskaryanSignal(times=time_array, particle=neutrino,
                                viewing_angle=observation_angle,
                                viewing_distance=observation_distance)

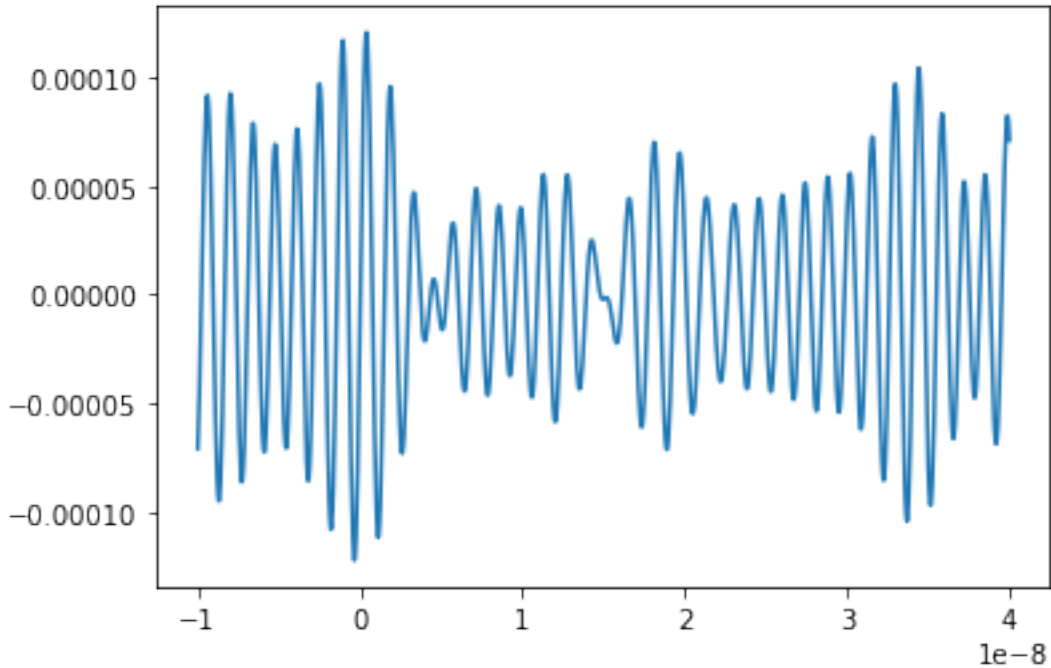
print(askaryan.value_type)
plt.plot(askaryan.times, askaryan.values)
plt.show()
```



`ThermalNoise` produces Rayleigh noise (in V) at a given temperature and resistance which has been passed through a bandpass filter of the given frequency range:

```
time_array = np.linspace(-10e-9, 40e-9, 1001)
noise_temp = 300 # K
system_resistance = 1000 # ohm
frequency_range = (550e6, 750e6) # Hz
noise = pyrex.ThermalNoise(times=time_array, temperature=noise_temp,
                           resistance=system_resistance,
                           f_band=frequency_range)

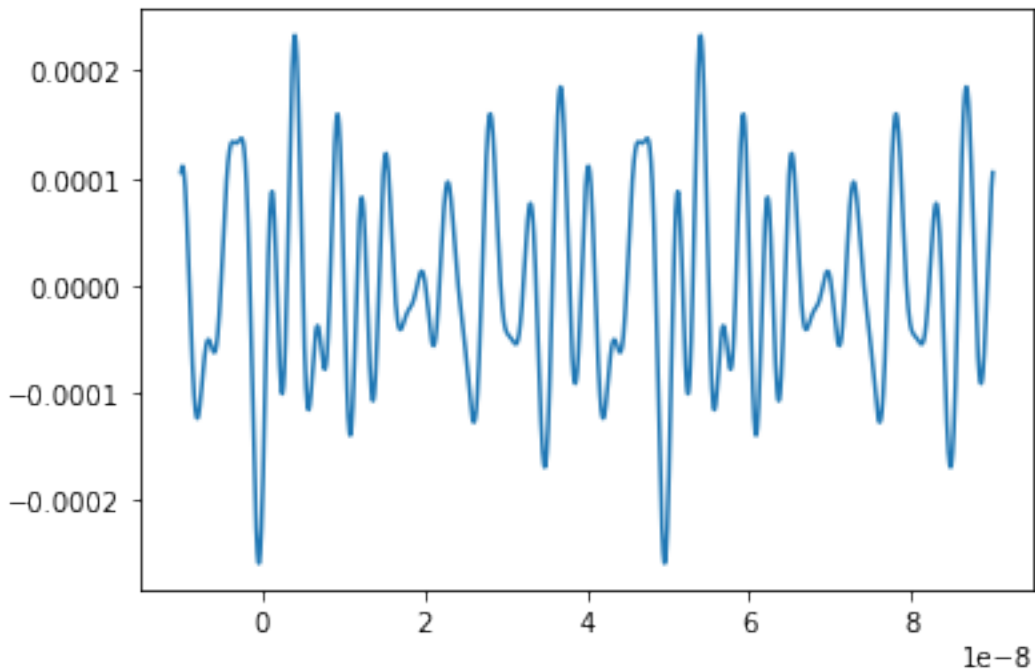
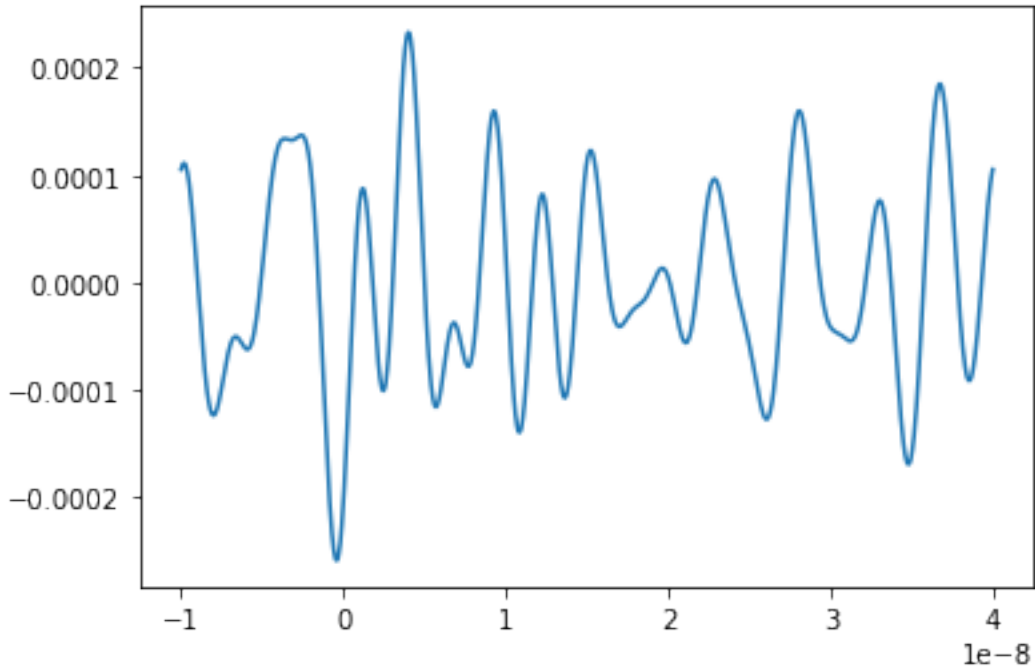
print(noise.value_type)
plt.plot(noise.times, noise.values)
plt.show()
```



Note that since `ThermalNoise` inherits from `FunctionSignal`, it can be extrapolated nicely to new times. It may be highly periodic outside of its original time range however, unless a larger number of frequencies is requested on initialization.

```
short_noise = pyrex.ThermalNoise(times=time_array, temperature=noise_temp,
                                resistance=system_resistance,
                                f_band=(100e6, 400e6))
long_noise = short_noise.with_times(np.linspace(-10e-9, 90e-9, 2001))

plt.plot(short_noise.times, short_noise.values)
plt.show()
plt.plot(long_noise.times, long_noise.values)
plt.show()
```



2.2 Antenna Class and Subclasses

The base [Antenna](#) class provided by PyREx is designed to be subclassed in order to match the needs of each project. At its core, an [Antenna](#) object is initialized with a position, a temperature, and a frequency range, as well as optionally a resistance (for noise calculations) and a boolean dictating whether or not noise should be added to the antenna's signals (note that if noise is to be added, a resistance must be specified).

```
# Please note that some values are unrealistic for demonstration purposes
position = (0, 0, -100) # m
temperature = 300 # K
resistance = 1e17 # ohm
frequency_range = (0, 5) # Hz
basic_antenna = pyrex.Antenna(position=position, temperature=temperature,
                              resistance=resistance,
                              freq_range=frequency_range)
noiseless_antenna = pyrex.Antenna(position=position, noisy=False)
```

The basic properties of an *Antenna* object are `is_hit` and `waveforms`. The `is_hit` property specifies whether or not the antenna has been triggered by an event. `waveforms` is a list of all the waveforms which have triggered the antenna. The antenna also defines a *signals* attribute, which is a list of all signals the antenna has received, and `all_waveforms` which is a list of all waveforms (signal plus noise) the antenna has received including those which didn't trigger. Finally, the antenna has an `is_hit_mc` property which is similar to `is_hit`, but does not count triggers where noise alone would have triggered the antenna.

```
basic_antenna.is_hit == False
basic_antenna.waveforms == []
```

The *Antenna* class contains two attributes and three methods which represent characteristics of the antenna as they relate to signal processing. The attributes are `efficiency` and `antenna_factor`, and the methods are `Antenna.frequency_response()`, `Antenna.directional_gain()`, and `Antenna.polarization_gain()`. The attributes are to be set and the methods overwritten in order to customize the way the antenna responds to incoming signals. `efficiency` is simply a scalar which multiplies the signal the antenna receives (default value is 1). `antenna_factor` is a factor used in converting received electric fields into voltages (`antenna_factor = E / V`; default value is 1). `Antenna.frequency_response()` takes a frequency or list of frequencies (in Hz) and returns the frequency response of the antenna at each frequency given (default always returns 1). `Antenna.directional_gain()` takes angles `theta` and `phi` in the antenna's coordinates and returns the antenna's gain for a signal coming from that direction (default always returns 1). `Antenna.directional_gain()` is dependent on the antenna's orientation, which is defined by its `z_axis` and `x_axis` attributes. To change the antenna's orientation, use the `Antenna.set_orientation()` method which takes `z_axis` and `x_axis` arguments. Finally, `Antenna.polarization_gain()` takes a polarization vector and returns the antenna's gain for a signal with that polarization (default always returns 1).

```
basic_antenna.efficiency == 1
basic_antenna.antenna_factor == 1
freqs = [1, 2, 3, 4, 5]
basic_antenna.frequency_response(freqs) == [1, 1, 1, 1, 1]
basic_antenna.directional_gain(theta=np.pi/2, phi=0) == 1
basic_antenna.polarization_gain([0,0,1]) == 1
```

The *Antenna* class defines an `Antenna.trigger()` method which is also expected to be overwritten. `Antenna.trigger()` takes a *Signal* object as an argument and returns a boolean of whether or not the antenna would trigger on that signal (default always returns `True`).

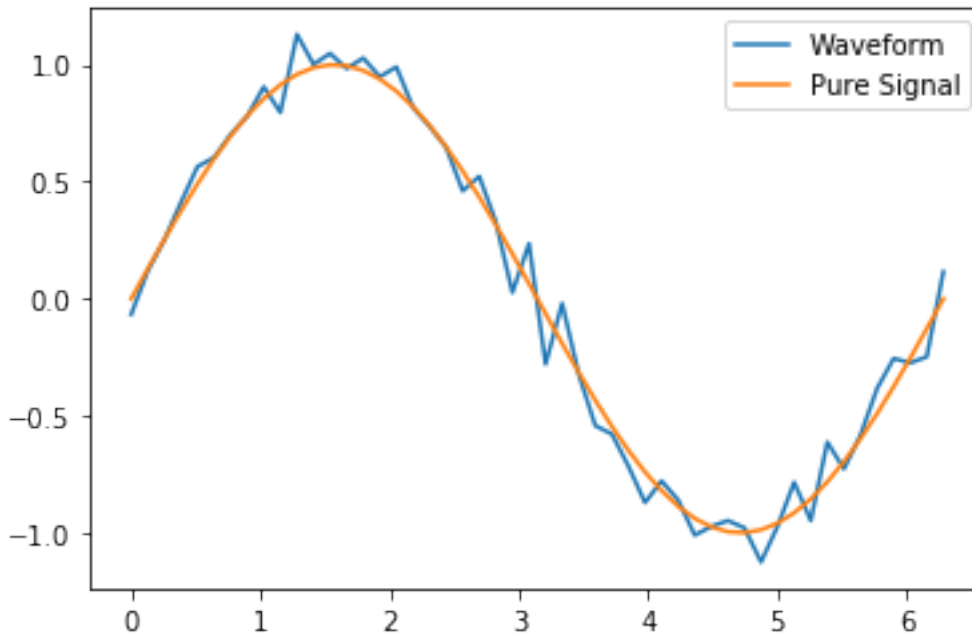
```
basic_antenna.trigger(pyrex.Signal([0],[0])) == True
```

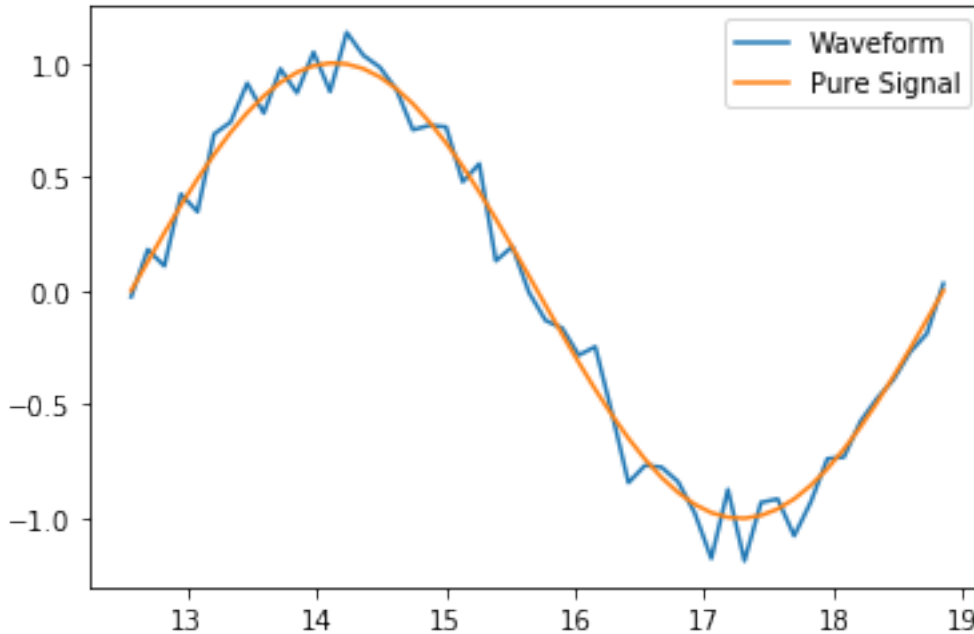
The *Antenna* class also defines an `Antenna.receive()` method which takes a *Signal* object and processes the signal according to the antenna's attributes (`efficiency`, `antenna_factor`, `response`, `directional_gain`, and `polarization_gain` as described above). To use the `Antenna.receive()` method, simply pass it the *Signal* object the antenna sees, and the *Antenna* class will handle the rest. You can also optionally specify the direction of travel of the signal (used in the `Antenna.directional_gain()` calculation) and the polarization direction of the signal (used in the `Antenna.polarization_gain()` calculation). If either of these is unspecified, the corresponding gain will simply be set to 1.


```

incoming_signal_1 = pyrex.FunctionSignal(np.linspace(0,2*np.pi), np.sin,
                                         value_type=pyrex.Signal.Type.voltage)
incoming_signal_2 = pyrex.FunctionSignal(np.linspace(4*np.pi,6*np.pi), np.sin,
                                         value_type=pyrex.Signal.Type.voltage)
basic_antenna.receive(incoming_signal_1)
basic_antenna.receive(incoming_signal_2, direction=[0,0,1], polarization=[1,0,0])
basic_antenna.is_hit == True
for waveform, pure_signal in zip(basic_antenna.waveforms, basic_antenna.signals):
    plt.figure()
    plt.plot(waveform.times, waveform.values, label="Waveform")
    plt.plot(pure_signal.times, pure_signal.values, label="Pure Signal")
    plt.legend()
    plt.show()

```

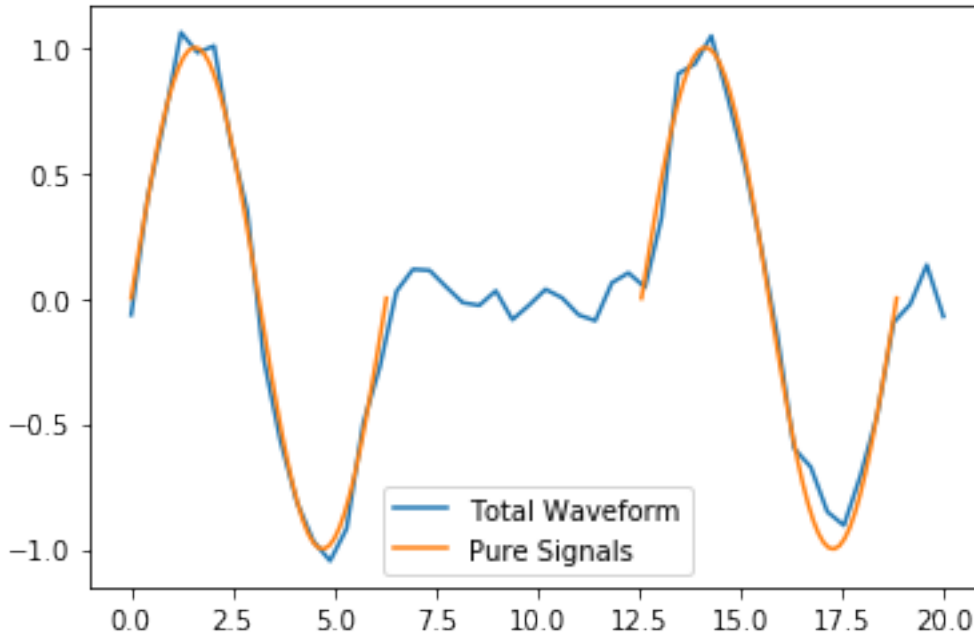




Beyond `Antenna.waveforms`, the `Antenna` object also provides methods for checking the waveform and trigger status for arbitrary times: `Antenna.full_waveform()` and `Antenna.is_hit_during()`. Both of these methods take a time array as an argument and return either the waveform `Signal` object for those times or whether said waveform triggered the antenna, respectively.

```
total_waveform = basic_antenna.full_waveform(np.linspace(0,20))
plt.plot(total_waveform.times, total_waveform.values, label="Total Waveform")
plt.plot(incoming_signal_1.times, incoming_signal_1.values, label="Pure Signals")
plt.plot(incoming_signal_2.times, incoming_signal_2.values, color="C1")
plt.legend()
plt.show()

basic_antenna.is_hit_during(np.linspace(0, 200e-9)) == True
```

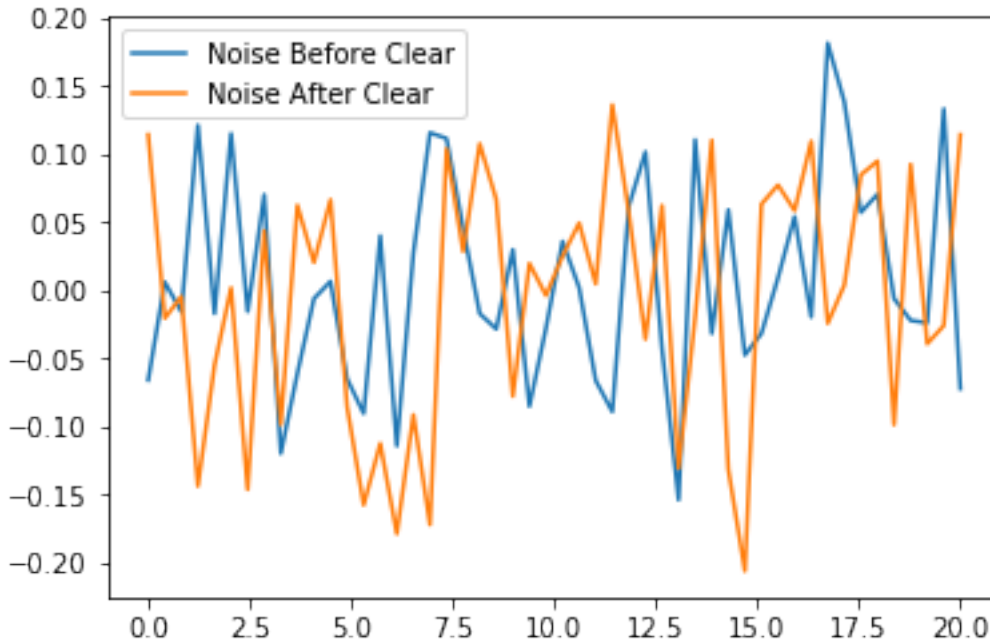


Finally, the `Antenna` class defines an `Antenna.clear()` method which will reset the antenna to a state of having received no signals:

```
basic_antenna.clear()
basic_antenna.is_hit == False
len(basic_antenna.waveforms) == 0
```

The `Antenna.clear()` method can also optionally reset the source of noise waveforms by passing `reset_noise=True` so that if the same signals are given after the antenna is cleared, the noise waveforms will be different:

```
noise_before = basic_antenna.make_noise(np.linspace(0, 20))
plt.plot(noise_before.times, noise_before.values, label="Noise Before Clear")
basic_antenna.clear(reset_noise=True)
noise_after = basic_antenna.make_noise(np.linspace(0, 20))
plt.plot(noise_after.times, noise_after.values, label="Noise After Clear")
plt.legend()
plt.show()
```



To create a custom antenna, simply inherit from the `Antenna` class:

```
class NoiselessThresholdAntenna(pyrex.Antenna):
    def __init__(self, position, threshold):
        super().__init__(position=position, noisy=False)
        self.threshold = threshold

    def trigger(self, signal):
        if max(np.abs(signal.values)) > self.threshold:
            return True
        else:
            return False
```

Our custom `NoiselessThresholdAntenna` should only trigger when the amplitude of a signal exceeds its threshold value:

```
my_antenna = NoiselessThresholdAntenna(position=(0, 0, 0), threshold=2)

incoming_signal = pyrex.FunctionSignal(np.linspace(0,10), np.sin,
                                       value_type=pyrex.Signal.Type.voltage)

my_antenna.receive(incoming_signal)
my_antenna.is_hit == False
len(my_antenna.waveforms) == 0
len(my_antenna.all_waveforms) == 1

incoming_signal = pyrex.Signal(incoming_signal.times,
                               5*incoming_signal.values,
                               incoming_signal.value_type)

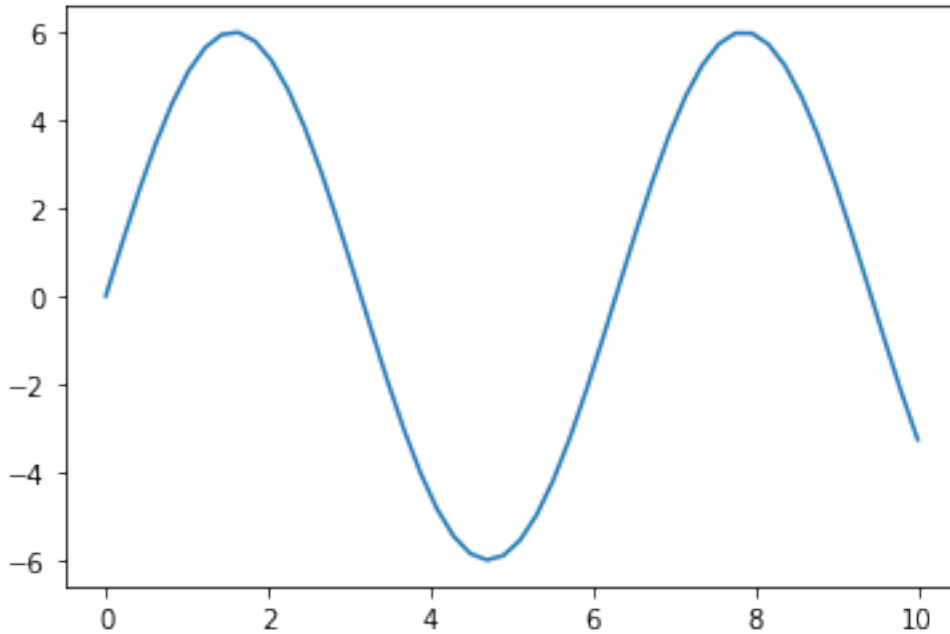
my_antenna.receive(incoming_signal)
my_antenna.is_hit == True
len(my_antenna.waveforms) == 1
len(my_antenna.all_waveforms) == 2

for wave in my_antenna.waveforms:
```

(continues on next page)

(continued from previous page)

```
plt.figure()
plt.plot(wave.times, wave.values)
plt.show()
```



For more on customizing PyREx, see the *Custom Sub-Package* section.

PyREx also defines *DipoleAntenna*, a subclass of *Antenna* which provides a basic threshold trigger, a basic bandpass filter frequency response, a sine-function directional gain, and a typical dot-product polarization effect. A *DipoleAntenna* object can be created as follows:

```
antenna_identifier = "antenna 1"
position = (0, 0, -100)
center_frequency = 250e6 # Hz
bandwidth = 300e6 # Hz
resistance = 100 # ohm
antenna_length = 3e8/center_frequency/2 # m
polarization_direction = (0, 0, 1)
trigger_threshold = 1e-5 # V
dipole = pyrex.DipoleAntenna(name=antenna_identifier, position=position,
                             center_frequency=center_frequency,
                             bandwidth=bandwidth, resistance=resistance,
                             effective_height=antenna_length,
                             orientation=polarization_direction,
                             trigger_threshold=trigger_threshold)
```

2.3 AntennaSystem and Detector Classes

The *AntennaSystem* class is designed to bridge the gap between the basic antenna classes and realistic antenna systems including front-end processing of the antenna's signals. It is designed to be subclassed, but by default it takes as an argument the *Antenna* class or subclass it is extending, or an object of that class. It provides an interface nearly identical to that of the *Antenna* class, but where an *AntennaSystem.front_end()* method (which by default

does nothing) is applied to the extended antenna's signals.

To extend an *Antenna* class or subclass into a full antenna system, inherit from the *AntennaSystem* class and define the *AntennaSystem.front_end()* method. If the front end of the antenna system requires some time to equilibrate to noise signals, that can be specified in the *AntennaSystem.lead_in_time* attribute, adding that amount of time before any waveforms to be processed. A different trigger also optionally can be defined for the antenna system (by default it uses the antenna's trigger):

```
class PowerAntennaSystem(pyrex.AntennaSystem):
    """Antenna system whose signals and waveforms are powers instead of
    voltages."""
    def __init__(self, position, temperature, resistance, frequency_range):
        super().__init__(pyrex.Antenna)
        # The setup_antenna method simply passes all arguments on to the
        # antenna class passed to super.__init__() and stores the resulting
        # antenna to self.antenna
        self.setup_antenna(position=position, temperature=temperature,
                           resistance=resistance,
                           freq_range=frequency_range)

    def front_end(self, signal):
        return pyrex.Signal(signal.times, signal.values**2,
                             value_type=pyrex.Signal.Type.power)
```

Objects of this class can then, for the most part, be interacted with as though they were regular antenna objects:

```
position = (0, 0, -100) # m
temperature = 300 # K
resistance = 1e17 # ohm
frequency_range = (0, 5) # Hz

basic_antenna_system = PowerAntennaSystem(position=position,
                                           temperature=temperature,
                                           resistance=resistance,
                                           frequency_range=frequency_range)

basic_antenna_system.trigger(pyrex.Signal([0],[0])) == True

incoming_signal_1 = pyrex.FunctionSignal(np.linspace(0,2*np.pi), np.sin,
                                         value_type=pyrex.Signal.Type.voltage)
incoming_signal_2 = pyrex.FunctionSignal(np.linspace(4*np.pi,6*np.pi), np.sin,
                                         value_type=pyrex.Signal.Type.voltage)
basic_antenna_system.receive(incoming_signal_1)
basic_antenna_system.receive(incoming_signal_2, direction=[0,0,1],
                             polarization=[1,0,0])
basic_antenna_system.is_hit == True
for waveform, pure_signal in zip(basic_antenna_system.waveforms,
                                basic_antenna_system.signals):
    plt.figure()
    plt.plot(waveform.times, waveform.values, label="Waveform")
    plt.plot(pure_signal.times, pure_signal.values, label="Pure Signal")
    plt.legend()
    plt.show()

total_waveform = basic_antenna_system.full_waveform(np.linspace(0,20))
plt.plot(total_waveform.times, total_waveform.values, label="Total Waveform")
plt.plot(incoming_signal_1.times, incoming_signal_1.values, label="Pure Signals")
plt.plot(incoming_signal_2.times, incoming_signal_2.values, color="C1")
```

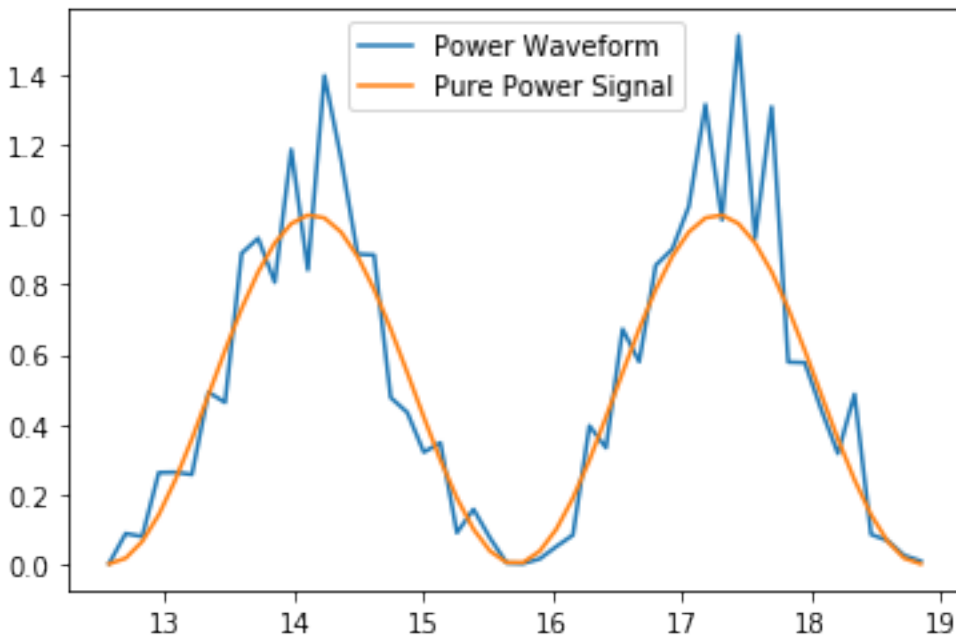
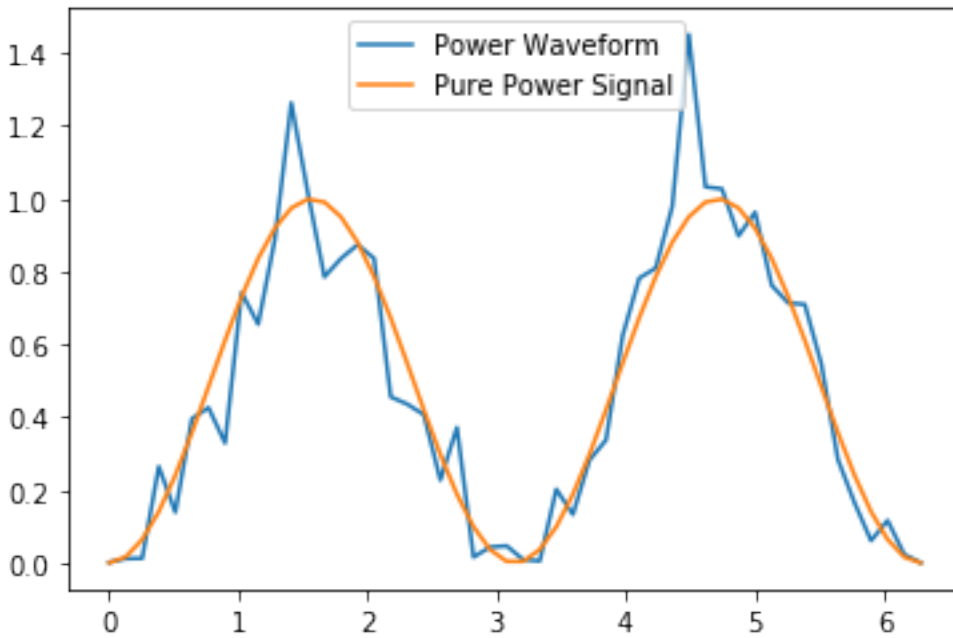
(continues on next page)

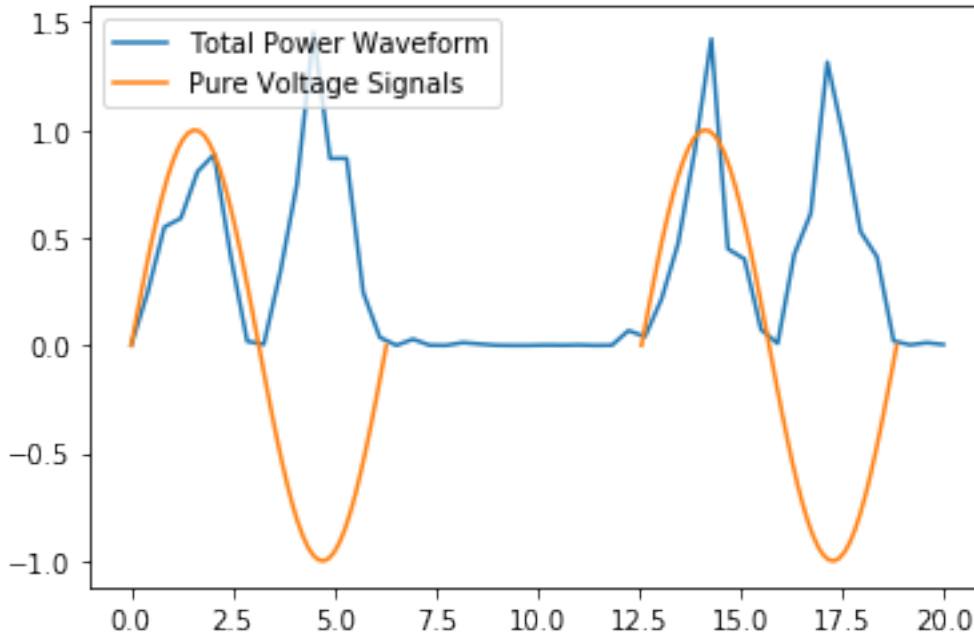
(continued from previous page)

```
plt.legend()
plt.show()

basic_antenna_system.is_hit_during(np.linspace(0, 200e-9)) == True

basic_antenna_system.clear()
basic_antenna_system.is_hit == False
len(basic_antenna_system.waveforms) == 0
```





The `Detector` class is another convenience class meant to be subclassed. It is useful for automatically generating many antennas (as would be used in a detector). Subclasses must define a `Detector.set_positions()` method to assign vector positions to the `self.antenna_positions` attribute. By default `Detector.set_positions()` will raise a `NotImplementedError`. Additionally subclasses may extend the default `Detector.build_antennas()` method which by default simply builds antennas of a passed antenna class using any keyword arguments passed to the method. In addition to simply generating many antennas at desired positions, another convenience of the `Detector` class is that once the `Detector.build_antennas()` method is run, it can be iterated directly as though the object were a list of the antennas it generated. And finally, the `Detector.triggered()` method will check whether any of the antennas have been triggered, and can be overridden in subclasses to define a more complicated detector trigger. An example of subclassing the `Detector` class is shown below:

```
class AntennaGrid(pyrex.Detector):
    """A detector composed of a plane of antennas in a rectangular grid layout
    some distance below the ice."""
    def set_positions(self, number, separation=10, depth=-50):
        self.antenna_positions = []
        n_x = int(np.sqrt(number))
        n_y = int(number/n_x)
        dx = separation
        dy = separation
        for i in range(n_x):
            x = -dx*n_x/2 + dx/2 + dx*i
            for j in range(n_y):
                y = -dy*n_y/2 + dy/2 + dy*j
                self.antenna_positions.append((x, y, depth))

grid_detector = AntennaGrid(9)

# Build the antennas
temperature = 300 # K
resistance = 1e17 # ohm
frequency_range = (0, 5) # Hz
grid_detector.build_antennas(pyrex.Antenna, temperature=temperature,
```

(continues on next page)

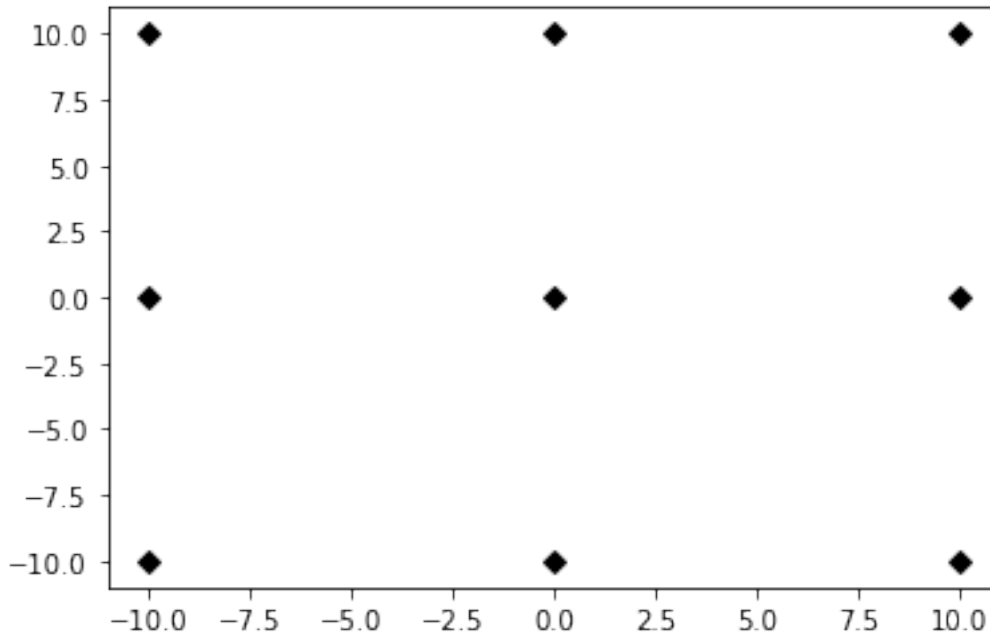
(continued from previous page)

```

                                resistance=resistance,
                                freq_range=frequency_range)

plt.figure(figsize=(6,6))
for antenna in grid_detector:
    x = antenna.position[0]
    y = antenna.position[1]
    plt.plot(x, y, "kD")
plt.ylim(plt.xlim())
plt.show()

```



Due to the parallels between [Antenna](#) and [AntennaSystem](#), an antenna system may also be used in the custom detector class. Note however, that the antenna positions must be accessed as `antenna.antenna.position` since we didn't define a position attribute for the `PowerAntennaSystem`:

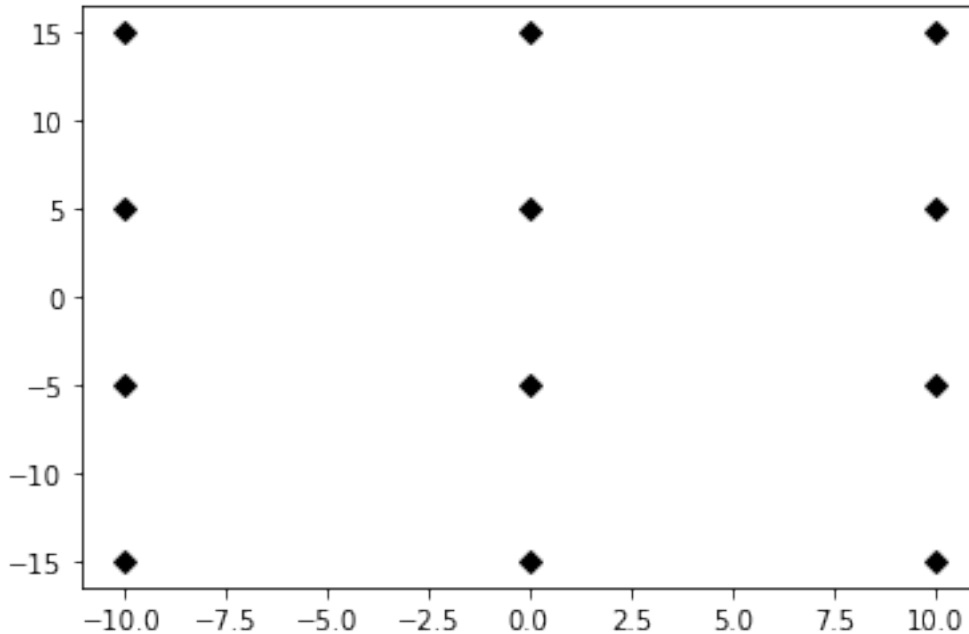
```

grid_detector = AntennaGrid(12)

# Build the antennas
temperature = 300 # K
resistance = 1e17 # ohm
frequency_range = (0, 5) # Hz
grid_detector.build_antennas(PowerAntennaSystem, temperature=temperature,
                             resistance=resistance,
                             frequency_range=frequency_range)

for antenna in grid_detector:
    x = antenna.antenna.position[0]
    y = antenna.antenna.position[1]
    plt.plot(x, y, "kD")
plt.show()

```



For convenience, objects derived from the `Detector` class can be added into a `CombinedDetector` object, which behaves similarly. The `CombinedDetector.build_antennas()` method should work seamlessly if the sub-detectors have the same `build_antennas()` method, otherwise it will do its best to dispatch keyword arguments between the sub-detectors. Similarly the `CombinedDetector.triggered()` method will return `True` if either sub-detector was triggered, with arguments to the method dispatched to the proper sub-triggers.

2.4 Ice and Earth Models

PyREx provides an ice model object `ice`, which is an instance of whichever ice model class is preferred (currently `pyrex.ice_model.AntarcticIce`). The `ice` object provides methods for calculating characteristics of the ice at different depths and frequencies outlined below:

```
depth = -1000 # m
pyrex.ice.temperature(depth)
pyrex.ice.index(depth)
pyrex.ice.gradient(depth)
frequency = 1e8 # Hz
pyrex.ice.attenuation_length(depth, frequency)
```

PyREx also provides two functions related to its earth model: `prem_density()` and `slant_depth()`. `prem_density()` calculates the density in grams per cubic centimeter of the earth at a given radius:

```
radius = 6360000 # m
pyrex.prem_density(radius)
```

`slant_depth()` calculates the material thickness in grams per square centimeter of a chord cutting through the earth at a given nadir angle, starting from a given depth:

```
nadir_angle = 60 * np.pi/180 # radians
depth = 1000 # m
pyrex.slant_depth(nadir_angle, depth)
```

2.5 Ray Tracing

PyREx provides ray tracing in the *RayTracer* and *RayTracePath* classes. *RayTracer* takes a launch point and receiving point as arguments (and optionally an ice model and z-step), and will solve for the paths between the points (as *RayTracePath* objects).

```
start = (0, 0, -250) # m
finish = (750, 0, -100) # m
my_ray_tracer = pyrex.RayTracer(from_point=start, to_point=finish)
```

The two most useful properties of *RayTracer* are *exists* and *solutions*. The *exists* property is a boolean value of whether or not path solutions exist between the launch and receiving points. *solutions* is the list of (zero or two) *RayTracePath* objects which exist between the launch and receiving points. There are many other properties available in *RayTracer*, outlined in the *PyREx API* section, which are mostly used internally and maybe not interesting otherwise.

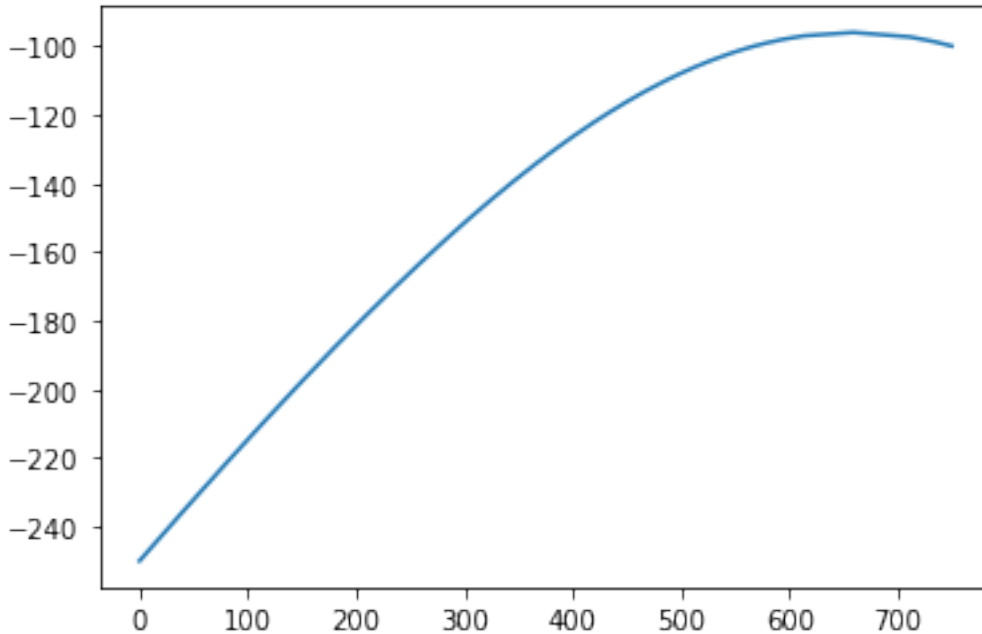
```
my_ray_tracer.exists
my_ray_tracer.solutions
```

The *RayTracePath* class contains the attributes of the paths between points. The most useful properties of *RayTracePath* are *tof*, *path_length*, *emitted_direction*, and *received_direction*. These properties provide the time of flight, path length, and direction of rays at the launch and receiving points respectively.

```
my_path = my_ray_tracer.solutions[0]
my_path.tof
my_path.path_length
my_path.emitted_direction
my_path.received_direction
```

RayTracePath also provides a *RayTracePath.attenuation()* method which gives the attenuation of the signal at a given frequency (or frequencies), and a *RayTracePath.coordinates* property which gives the x, y, and z coordinates of the path (useful mostly for plotting, and are not guaranteed to be accurate for other purposes).

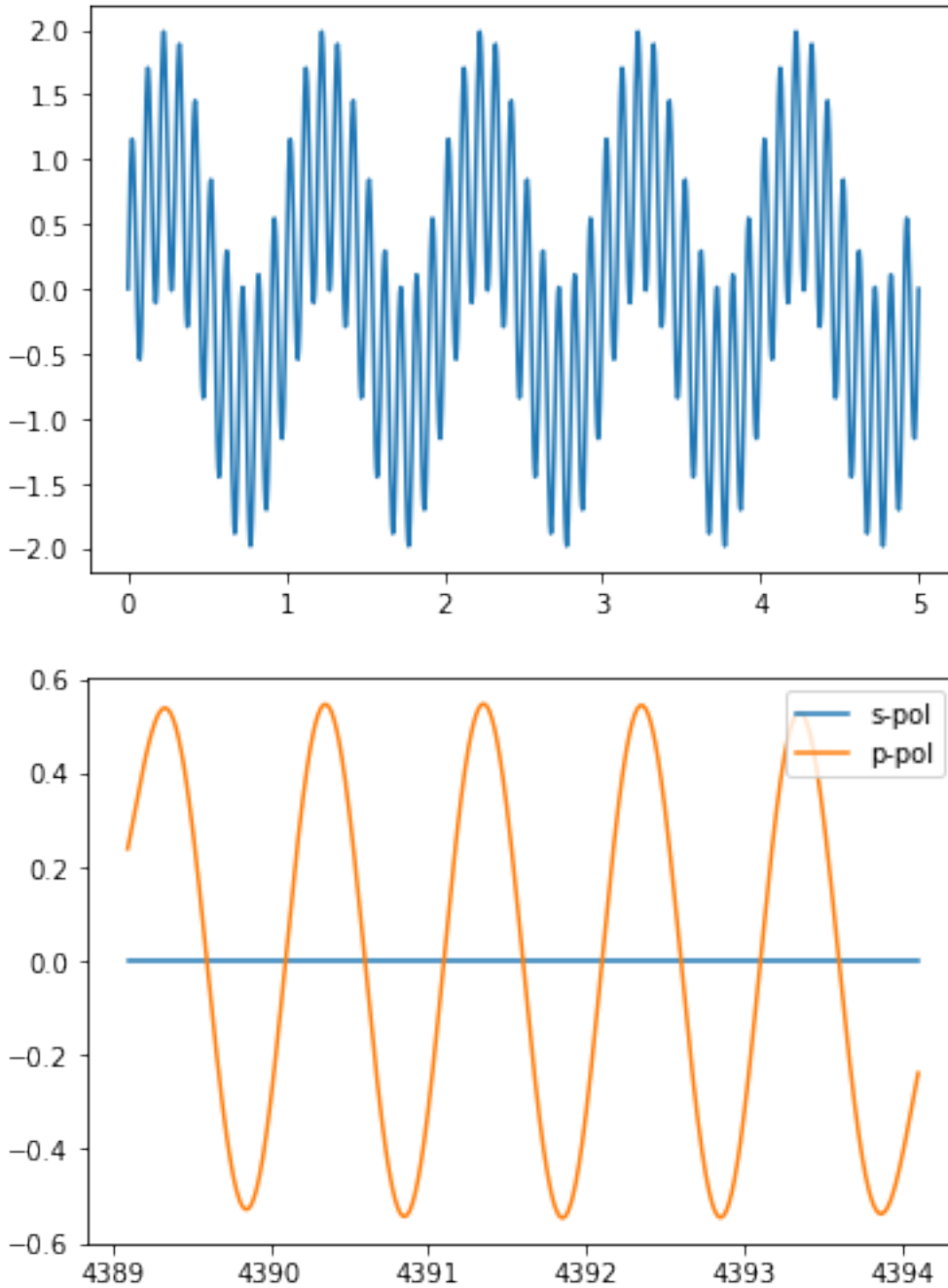
```
frequency = 100e6 # Hz
my_path.attenuation(frequency)
my_path.attenuation(np.linspace(1e8, 1e9, 11))
plt.plot(my_path.coordinates[0], my_path.coordinates[2])
plt.show()
```



Finally, `RayTracePath.propagate()` propagates a *Signal* object from the launch point to the receiving point of the path by applying the frequency-dependent attenuation from `RayTracePath.attenuation()`, and shifting the signal times by `RayTracePath.tof`. Note that it does not apply a $1/R$ effect based on the path length. If needed, this effect should be added in manually. `RayTracePath.propagate()` returns the *Signal* objects and polarization vectors of the s-polarized and p-polarized portions of the signal.

```
time_array = np.linspace(0, 5e-9, 1001)
launch_signal = (
    pyrex.FunctionSignal(time_array, lambda t: np.sin(1e9*2*np.pi*t))
    + pyrex.FunctionSignal(time_array, lambda t: np.sin(1e10*2*np.pi*t))
)
plt.plot(launch_signal.times*1e9, launch_signal.values)
plt.show()
# Polarize perpendicular to the path in the x-z plane
launch_pol = np.cross(my_path.emitted_direction, (0, 1, 0))
print(launch_pol)

rec_signals, rec_pols = my_path.propagate(launch_signal, polarization=launch_pol)
plt.plot(rec_signals[0].times*1e9, rec_signals[0].values, label="s-pol signal")
plt.plot(rec_signals[1].times*1e9, rec_signals[1].values, label="p-pol signal")
plt.legend()
plt.show()
print(rec_pols)
```



2.6 Particle Generation

PyREx includes the `Particle` class as a container for information about neutrinos which are generated to produce Askaryan pulses. A `Particle` contains an id, a vertex, a direction, an energy, an interaction, and a weight:

```
particle_type = pyrex.Particle.Type.electron_neutrino
initial_position = (0,0,0) # m
```

(continues on next page)

(continued from previous page)

```
direction_vector = (0,0,-1)
particle_energy = 1e8 # GeV
particle = pyrex.Particle(particle_id=particle_type, vertex=initial_position,
                        direction=direction_vector, energy=particle_energy)
```

The interaction attribute is an instance of an Interaction class (*NeutrinoInteraction* by default) which is a model for how the neutrino interacts in the ice. It has a kind denoting whether the interaction will be charged-current or neutral-current, an inelasticity, `em_frac` and `had_frac` describing the resulting particle shower(s), and `cross_section` and `interaction_length` in the ice at the energy of the parent *Particle* object:

```
type(particle.interaction)
particle.interaction.kind
particle.interaction.inelasticity
particle.interaction.em_frac
particle.interaction.had_frac
particle.interaction.cross_section
particle.interaction.interaction_length
```

PyREx also includes a number of classes for generating random neutrinos in various ice volumes. The *CylindricalGenerator* and *RectangularGenerator* classes generate neutrinos uniformly in cylindrical or rectangular volumes respectively. These generator classes take as arguments the necessary dimensions and an energy (which can be a scalar value or a function returning scalar values). Additional arguments include whether to reject events shadowed by the Earth, as well as a desired flavor ratio:

```
volume_radius = 1000 # m
volume_depth = 500 # m
flavor_ratio = (1, 1, 1) # even distribution of neutrino flavors
my_generator = pyrex.CylindricalGenerator(dr=volume_radius,
                                          dz=volume_depth,
                                          energy=particle_energy,
                                          shadow=False,
                                          flavor_ratio=flavor_ratio)
my_generator.create_event()
```

The `create_event()` method of the generator returns an *Event* object, which contains a tree of *Particle* objects representing the event. Currently this tree will only contain a single neutrino, but could be expanded in the future in order to describe more exotic events. The neutrino is available as the only element in the list `Event.roots`. It can also be accessed by iterating the *Event* object.

Lastly, PyREx includes *ListGenerator* and *FileGenerator* classes which can be used to reproduce pre-generated events from either a list or from simulation output files, respectively. For example, to continuously re-throw our *Particle* object from above:

```
repetitive_generator = pyrex.ListGenerator([pyrex.Event(particle)])
repetitive_generator.create_event()
repetitive_generator.create_event()
```

2.7 Full Simulation

PyREx provides the *EventKernel* class to control a basic simulation including the creation of neutrinos and their respective signals, the propagation of their pulses to the antennas, and the triggering of the antennas. The *EventKernel* is designed to be modular and can use a specific ice model, ray tracer, and signal times as specified in optional arguments (the defaults are explicitly specified below):

```

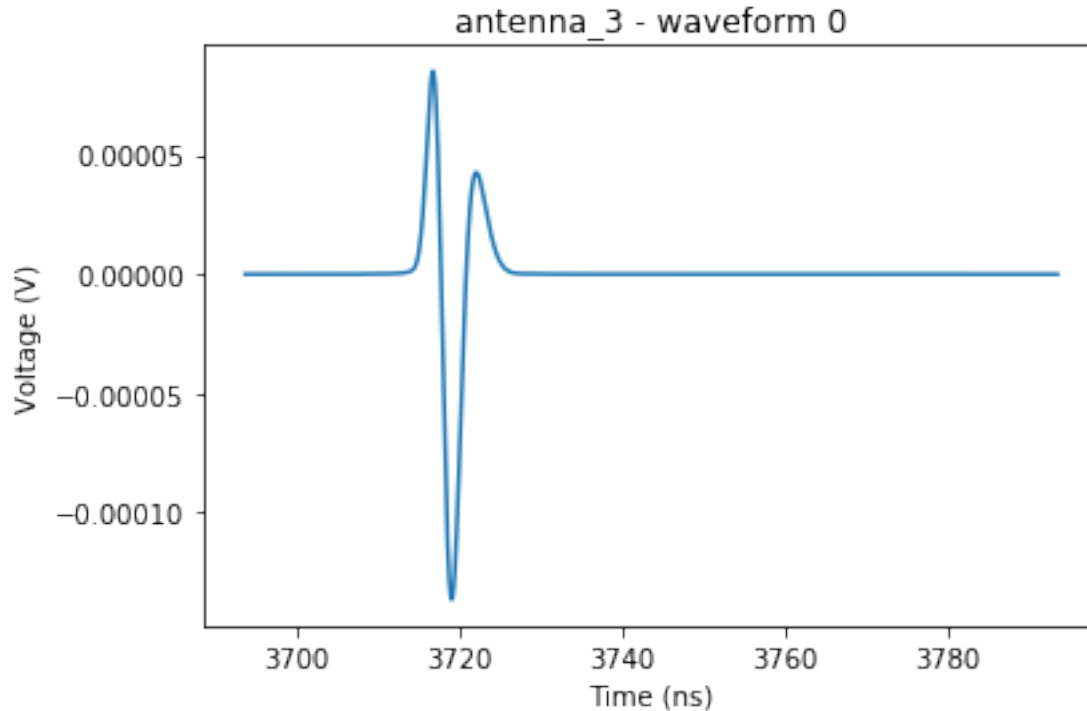
particle_generator = pyrex.CylindricalGenerator(dr=1000, dz=1000, energy=1e8)
detector = []
for i, z in enumerate([-100, -150, -200, -250]):
    detector.append(
        pyrex.DipoleAntenna(name="antenna_"+str(i), position=(0, 0, z),
                           center_frequency=250e6, bandwidth=300e6,
                           resistance=0, effective_height=0.6,
                           trigger_threshold=1e-4, noisy=False)
    )
kernel = pyrex.EventKernel(generator=particle_generator,
                           antennas=detector,
                           ice_model=pyrex.ice,
                           ray_tracer=pyrex.RayTracer,
                           signal_times=np.linspace(-20e-9, 80e-9, 2000,
                                                    endpoint=False))

triggered = False
while not triggered:
    for antenna in detector:
        antenna.clear()
    event = kernel.event()
    for antenna in detector:
        if antenna.is_hit:
            triggered = True
            break

particle = event.roots[0]
print("Particle type: ", particle.id)
print("Shower vertex: ", particle.vertex)
print("Shower axis: ", particle.direction)
print("Particle energy: ", particle.energy)
print("Interaction type:", particle.interaction.kind)
print("Electromagnetic shower fraction:", particle.interaction.em_frac)
print("Hadronic shower fraction: ", particle.interaction.had_frac)
print("Event weight:", particle.weight)

for antenna in detector:
    for i, wave in enumerate(antenna.waveforms):
        plt.plot(wave.times * 1e9, wave.values)
        plt.xlabel("Time (ns)")
        plt.ylabel("Voltage (V)")
        plt.title(antenna.name + " - waveform "+str(i))

```



2.8 Data File I/O

The `File` class controls the reading and writing of data files for simulation. At the most basic it takes a filename and mode in which to open the file, and if the file type is supported the object will be the appropriate file handler. Like python's `open()` function, the `File` class works as a context manager and should preferably be used in `with` statements. Currently the only data file type supported by PyREx is HDF5. Depending on whether an HDF5 file is being read or written there are additional keyword arguments that may be provided to `File`. HDF5 files support the following modes: 'r' for read-only, 'w' for write (overwrites existing file), 'a'/'r+' for append (doesn't overwrite existing file), and 'x' for write (fails if file exists already).

If writing an HDF5 file, the optional arguments specify which event data to write. The available write options are `write_particles`, `write_triggers`, `write_antenna_triggers`, `write_rays`, `write_noise`, and `write_waveforms`. Most of these are self-explanatory, but `write_antenna_triggers` will write triggering information for each antenna in the detector and `write_noise` will write the frequency data required to replicate noise waveforms. The last optional argument is `require_trigger` which specifies which data should only be written when the detector is triggered. If a boolean value, requires trigger or not for all data with the exception of particle and trigger data, which is always written. If a list of strings, the listed data will require triggers and any other data will always be written.

The most straightforward way to write data files is to pass a `File` object to the `EventKernel` object handling the simulation. In such a case, a global trigger condition should be passed to the `EventKernel` as well, either as a function which acts on a detector object, or as the "global" key in a dictionary of functions representing various trigger conditions:

```
particle_generator = pyrex.CylindricalGenerator(dr=1000, dz=1000, energy=1e8)
detector = []
for i, z in enumerate([-100, -150, -200, -250]):
    detector.append(
        pyrex.DipoleAntenna(name="antenna_"+str(i), position=(0, 0, z),
```

(continues on next page)

(continued from previous page)

```

        center_frequency=250e6, bandwidth=300e6,
        resistance=0, effective_height=0.6,
        trigger_threshold=1e-4, noisy=False)
    )

def global_trigger_condition(det):
    for ant in det:
        if ant.is_hit:
            return True
    return False

def even_antenna_trigger(det):
    for i, ant in enumerate(det):
        if i%2==0 and ant.is_hit:
            return True
    return False

trigger_conditions = {
    "global": global_trigger_condition,
    "evens": even_antenna_trigger,
    "ant1": lambda det: det[1].is_hit
}

with pyrex.File('my_data_file.h5', 'x') as f:
    kernel = pyrex.EventKernel(generator=particle_generator,
                               antennas=detector,
                               event_writer=f,
                               triggers=trigger_conditions)

    for _ in range(10):
        for antenna in detector:
            antenna.clear()
        event, triggered = kernel.event()

```

If you want to manually write the data file, then the `File.set_detector()` and `File.add()` methods are necessary. `File.set_detector()` associates the given antennas with the file object (and writes their data) and `File.add()` adds the data from the given event to the file. Here we also manually open and close the file object with `File.open()` and `File.close()`, and add some metadata to the file with `File.add_file_metadata()`:

```

f = pyrex.File('my_data_file_2.h5', 'w')
f.open()

f.add_file_metadata({"write_style": "manual", "number_of_events": 10})

f.set_detector(detector)

kernel = pyrex.EventKernel(generator=particle_generator,
                           antennas=detector)

for _ in range(10):
    for antenna in detector:
        antenna.clear()
    event = kernel.event()
    triggered = False
    for antenna in detector:
        if antenna.is_hit:

```

(continues on next page)

(continued from previous page)

```

        triggered = True
        break
    f.add(event, triggered=triggered)

f.close()

```

The `File` objects also support writing miscellaneous analysis data to the file. `File.create_analysis_dataset()` creates and returns a basic HDF5 dataset. `File.create_analysis_metadataset()` creates a joined set of tables for string and float data, which can be written to with `File.add_analysis_metadata()`. And finally, `File.add_analysis_indices()` allows for linking event indices to specific rows of analysis data.

```

with pyrex.File('my_data_file.h5', 'a') as f:
    f.create_analysis_metadataset("effective_volume")
    gen_vol = (np.pi*1000**2)*1000
    # Just set an arbitrary number of triggers for now. We'll get into reading
    # files in the examples below.
    n_triggers = 5
    data = {
        "generation_volume": gen_vol,
        "veff": n_triggers/10*gen_vol,
        "error": np.sqrt(n_triggers)/10*gen_vol,
        "unit": "m^3"
    }
    f.add_analysis_metadata("effective_volume", data)

    other = f.create_analysis_dataset("meaningless_data",
                                     data=np.ones((20, 5)))
    other.attrs['rows_per_event'] = 2
    for i in range(10):
        f.add_analysis_indices("meaningless_data", global_index=i,
                              start_index=2*i, length=2)

```

If reading an HDF5 file, the `slice_range` argument specifies the size of event slices to load into memory at once when iterating over events. In general, increasing the `slice_range` will improve the speed of iteration at the cost of greater memory consumption.

```

with pyrex.File('my_data_file.h5', 'r', slice_range=100) as f:
    pass

```

When reading HDF5 files, there are a number of methods and attributes available to access the data. With the `File` object alone, `File.file_metadata` contains a dictionary of the file's metadata and `File.antenna_info` contains a list of dictionaries with data for each antenna in the detector the file was run with. If waveform data is available, `File.get_waveforms()` can be used to get all waveforms in the file or a specific subset based on `event_id`, `antenna_id`, and `waveform_type` arguments. Finally, direct access to the contents of the HDF5 file is supported through either the proper paths or nicknames.

```

with pyrex.File('my_data_file.h5', 'r') as f:
    print(f.file_metadata)
    print(f.antenna_info[0])

    # No waveform data was stored above, so these will fail if run
    # All waveforms:
    # wfs = f.get_waveforms()
    # Waveforms from event 0
    # wfs = f.get_waveforms(event_id=0)

```

(continues on next page)

(continued from previous page)

```

# Waveforms in antenna 1 from all events
# wfs = f.get_waveforms(antenna_id=1)
# Direct waveform in antenna 4 from event 5
# wf = f.get_waveforms(event_id=5, antenna_id=4, waveform_type=0)

# Using full file path
triggers = f['data/triggers']
# Using dataset nickname
particle_string_metadata = f['particles_meta_str']
# Using analysis dataset nickname
other = f['meaningless_data']

```

HDF5 files opened in read-only mode can also be iterated over, which allows access to the data for each event in turn. When iterating, the event objects have the following methods for accessing data. `get_particle_info()` and `get_rays_info()` return a list of dictionaries or attribute values for the event's particles or rays, respectively. The `is_neutrino`, `is_nubar`, and `flavor` attributes also contain the associated basic information about the base particle of the event. `get_waveforms()` returns the waveforms for the event, or a specific subset based on `antenna_id` and `waveform_type` (as above). The `triggered` attribute contains whether the event triggered the detector and the `get_triggered_components()` method returns a list of the trigger conditions of the detector which were met (as specified when writing the file). And finally, if noise data is recorded for the event it is contained in the `noise_bases` attribute. Iteration of the HDF5 files supports slicing as long as the step size is positive-valued, and individual events can also be reached by indexing the *File* object.

```

with pyrex.File('my_data_file.h5', 'r') as f:
    for event in f:
        print(event.is_neutrino, event.is_nubar, event.flavor)
        print(event.triggered, event.get_triggered_components())

    for event in f[1:5:2]:
        print(event.get_particle_info('particle_name'),
              event.get_particle_info('vertex'))
        print(np.degrees(event.get_rays_info('receiving_angle')))

    print(f[3].get_rays_info('tof'))

    # No waveform data was stored above, so this will fail if run
    # wfs = f[3].get_waveforms(antenna_id=2)

```

2.9 More Examples

For more code examples, see the *Example Code* section and the python notebooks in the examples directory.

CUSTOM SUB-PACKAGE

While the PyREx package provides a basis for simulation, the real benefits come in customizing the analysis for different purposes. To this end the custom sub-package allows for plug-in style modules to be distributed for different collaborations.

By default PyREx comes with a few custom modules included, listed below. More information about each of these modules can be found in their respective API sections:

- *Askaryan Radio Array* (*pyrex.custom.ara*)
- *Antarctic Ross Ice-Shelf Antenna Neutrino Array* (*pyrex.custom.arianna*)
- *IceCube Radio Extension* (*pyrex.custom.irex*)
- *Layered Ice Models* (*pyrex.custom.layered_ice*)

Other institutions and research groups are encouraged to create their own custom modules to integrate with PyREx. These modules have full access to PyREx as if they were a native part of the package. When PyREx is loaded it automatically scans for these custom modules in certain parts of the filesystem and includes any modules that it can find. The first place searched is the `custom` directory in the PyREx package itself. Next, if a `.pyrex-custom` directory exists in the user's home directory (note the leading `.`), its subdirectories are searched for `custom` directories and any modules in these directories are included. Finally, if a `pyrex-custom` directory exists in the current working directory (this time without the leading `.`), its subdirectories are similarly scanned for modules inside `custom` directories. Note that if any name-clashing occurs, the first result found takes precedence (without warning). Additionally, none of these `custom` directories should contain an `__init__.py` file, or else the plug-in system may not work (For more information on the implementation, see PEP 420 and/or David Beazley's 2015 PyCon talk on Modules and Packages at <https://youtu.be/0oTh1CXRaQ0?t=1h25m45s>).

As an example, in the following filesystem layout (which is not meant to reflect the actual current modules available to PyREx) the available custom modules are `pyrex.custom.pyspice`, `pyrex.custom.irex`, `pyrex.custom.ara`, `pyrex.custom.arianna`, and `pyrex.custom.my_analysis`. Additionally note that the name clash for the ARA module will result in the module included in PyREx being loaded and the ARA module in `.pyrex-custom` will be ignored.

```

/path/to/site-packages/pyrex/
|-- __init__.py
|-- signals.py
|-- antenna.py
|-- ...
|-- custom/
|   |-- pypspice.py
|   |-- irex/
|       |-- __init__.py
|       |-- antenna.py
|       |-- ...
|   |-- ara/
|       |-- __init__.py
|       |-- antenna.py
|       |-- ...

/path/to/home_dir/.pyrex-custom/
|-- ara/
|   |-- custom/
|       |-- ara/
|           |-- __init__.py
|           |-- antenna.py
|           |-- ...
|-- arianna/
|   |-- custom/
|       |-- arianna/
|           |-- __init__.py
|           |-- antenna.py
|           |-- ...

/path/to/cwd/pyrex-custom/
|-- my_analysis_module/
|   |-- custom/
|       |-- my_analysis.py

```

3.1 Build Your Own Custom Module

In the course of using PyREx you may wish to change some behavior of parts of the code. Due to the modularity of the code, many behaviors should be customizable by substituting in your own classes inheriting from those already in PyREx. By adding these classes to your own custom module, your code can behave as though it was a native part of the PyREx package. Below the classes which can be easily substituted with your own version are listed, and descriptions of the behavior expected of the classes is outlined.

3.1.1 Askaryan Signal

The *AskaryanSignal* class is responsible for storing the time-domain signal of the Askaryan signal produced by a particle shower. The `__init__()` method of an *AskaryanSignal*-like class must accept the arguments listed below:

Attribute	Description
times	A list-type (usually a numpy array) of time values at which to calculate the amplitude of the Askaryan pulse.
particle	A Particle object representing the neutrino that causes the event. Should have an energy, vertex, id, and an interaction with an em_frac and had_frac.
viewing_angle	The viewing angle in radians measured from the shower axis.
viewing_distance	The distance of the observation point from the shower vertex.
ice	The ice model to be used for describing the medium's index of refraction.
t0	The starting time of the Askaryan pulse / showers (default 0).

The `__init__()` method should result in a *Signal* object with values being a numpy array of amplitudes corresponding to the given times and should have a proper `value_type`. Additionally, all methods of the *Signal* class should be implemented (typically by just inheriting from *Signal*).

3.1.2 Antenna / Antenna System

The *Antenna* class is primarily responsible for receiving and triggering on *Signal* objects. The `__init__()` method of an *Antenna*-like class must accept a `position` argument, and any other arguments may be specified as desired. The `__init__()` method should set the `position` attribute to the given argument. If not inheriting from *Antenna*, the following methods and attributes must be implemented and may require the `__init__()` method to set some other attributes. *AntennaSystem*-like classes must expose the same required methods and attributes as *Antenna*-like classes, typically by passing calls down to an underlying *Antenna*-like object and applying some extra processing.

The `signals` attribute should contain a list of all pure *Signal* objects that the antenna has seen. This is different from the `all_waveforms` attribute, which should contain a list of all waveform (pure signal + noise) *Signal* objects the antenna has seen. Yet again different from the `waveforms` attribute, which should contain only those waveforms which have triggered the antenna.

If using the default `all_waveforms` and `waveforms`, a `_noises` attribute and `_triggers` attribute must be initialized to empty lists in `__init__()`. Additionally a `make_noise()` method must be defined which takes a times array and returns a *Signal* object with noise amplitudes in the `values` attribute. If using the default `make_noise()` method, a `_noise_master` attribute must be set in `__init__()` to either `None` or a *Signal* object that can generate noise waveforms (setting `_noise_master` to `None` and handling noise generation with the attributes `freq_range` and `noise_rms`, or `temperature` and `resistance`, is recommended).

A `full_waveform()` method is required which will take a times array and return a *Signal* object of the waveform the antenna sees at those times. If using the default `full_waveform()`, a `noisy` attribute is required which contains a boolean value of whether or not the antenna includes noise in its waveforms. If `noisy` is `True` then a `make_noise()` method is also required, as described in the previous paragraph.

An `is_hit` attribute is required which will be a boolean of whether or not the antenna has been triggered by any waveforms. Similarly an `is_hit_during()` method is required which will take a times array and return a boolean of whether the antenna is triggered during those times.

The `trigger()` method of the antenna should take a *Signal* object and return a boolean of whether or not that signal would trigger the antenna.

The `clear()` method should reset the antenna to a state of having received no signals (i.e. the state just after initialization), and should accept a boolean for `reset_noise` which will force the noise waveforms to be recalculated. If using the default `clear()` method, the `_noises` and `_triggers` attributes must be lists.

A `receive()` method is required which will take a *Signal* object as `signal`, a 3-vector (list) as `direction`, and a 3-vector (list) as `polarization`. This function doesn't return anything, but instead processes the input signal and stores it to the `signals` list (and anything else needed for the antenna to have officially received the signal). This

is the final required method, but if using the default `receive()` method, an `antenna_factor` attribute is needed to define the conversion from electric field to voltage and an `efficiency` attribute is required, along with four more methods which must be defined:

The `_convert_to_antenna_coordinates()` method should take a point in cartesian coordinates and return the `r`, `theta`, and `phi` values of that point relative to the antenna. The `directional_gain()` method should take `theta` and `phi` in radians and return a (complex) gain based on the directional response of the antenna. Similarly the `polarization_gain()` method should take a polarization 3-vector (list) of an incoming signal and return a (complex) gain based on the polarization response of the antenna. Finally, the `response()` method should take a list of frequencies and return the (complex) gains of the frequency response of the antenna. This assumes that the directional and frequency responses are separable. If this is not the case then the gains may be better handled with a custom `receive()` method.

3.1.3 Detector

The preferred method of creating your own detector class is to inherit from the `Detector` class and then implement the `set_positions()` method, the `triggered()` method, and potentially the `build_antennas()` method. However the only requirement of a `Detector`-like object is that iterating over it will visit each antenna exactly once. This means that a simple list of antennas is an acceptable rudimentary detector. The advantages of using the `Detector` class are easy breaking into subsets (a detector could be made up of stations, which in turn are made up of strings) and the simpler `triggered()` method for trigger checks.

3.1.4 Ice Model

Ice model classes are responsible for describing the properties of the ice as functions of depth and frequency. While not explicitly required, all ice model classes in PyREx are defined only with static and class methods, so no `__init__()` method is actually necessary. The necessary methods, however, are as follows:

The `index()` method should take a depth (or numpy array of depths) and return the corresponding index of refraction. Conversely, the `depth_with_index()` method should take an index of refraction (or numpy array of indices) and return the corresponding depths. In the case of degeneracy here (for example with uniform ice), the recommended behavior is to return the shallowest depth with the given index, though PyREx's behavior in cases of non-monotonic index functions is not well defined.

The `temperature()` method should take a depth (or numpy array of depths) and return the corresponding ice temperature in Kelvin.

Finally, the `attenuation_length()` function should take a depth (or numpy array of depths) and a frequency (or numpy array of frequencies) and return the corresponding attenuation length. In the case of one scalar and one array argument, a simple 1D array should be returned. In the case of both arguments being arrays, the return value should be a 2D array where each row represents different frequencies at a single depth and each column represents different depths at a single frequency.

3.1.5 Ray Tracer / Ray Trace Path

The `RayTracer` and `RayTracePath` classes are responsible for handling ray tracing through the ice between shower vertices and antenna positions. The `RayTracer` class finds the paths between the two points and the `RayTracePath` calculates values along the path. Due to the potential for high calculation costs, the PyREx `RayTracer` and `RayTracePath` classes inherit from a `LazyMutableClass` which allows the use of a `lazy_property()` decorator to cache results of attribute calculations. It is recommended that any other ray tracing classes consider doing this as well.

The `__init__()` method of a `RayTracer`-like class should take as arguments a 3-vector (list) `from_point`, a 3-vector (list) `to_point`, and an IceModel-like `ice_model`. The only required features of the class are a boolean

attribute `exists` recording whether or not paths exist between the given points, and an iterable attribute `solutions` which iterates over *RayTracePath*-like objects between the points.

A *RayTracePath*-like class will be initialized by a corresponding *RayTracer*-like object, so there are no requirements on its `__init__()` method. The path must have `emitted_direction` and `received_direction` attributes which are numpy arrays of the cartesian direction the ray is pointing at the `from_point` and `to_point` of the ray tracer, respectively. The path must also have attributes for the `path_length` and `tof` (time of flight) along the path.

The path class must have a `propagate()` method which takes a *Signal* object as its argument and propagates that signal by applying any attenuation and time of flight. This method does not have a return value. Additionally, note that any 1/R factor that the signal could have is not applied in this method, but externally by dividing the signal values by the `path_length`. If using the default `propagate()` method, an `attenuation()` method is required which takes an array of frequencies `f` and returns the attenuation factors for a signal along the path at those frequencies.

Finally, though not required it is recommended that the path have a `coordinates` attribute which is a list of lists of the x, y, and z coordinates along the path (with some reasonable step size). This method is used for plotting purposes and does not need to have the accuracy necessary for calculations.

3.1.6 Interaction Model

The interaction model used for *Particle* interactions in ice handles the cross sections and interaction lengths of neutrinos, as well as the ratios of their interaction types and the resulting shower fractions. An interaction class should inherit from *Interaction* (preferably keeping its `__init__()` method) and should implement the following methods:

The `cross_section` property method should return the neutrino cross section for the *Interaction*.*particle* parent, specific to the *Interaction.kind*. Similarly the `total_cross_section` property method should return the neutrino cross section for the *Interaction.particle* parent, but this should be the total cross section for both charged-current and neutral-current interactions. The `interaction_length` and `total_interaction_length` properties will convert these cross sections to interaction lengths automatically.

The `choose_interaction()` method should return a value from *Interaction.Type* representing the interaction type based on a random choice. Similarly the `choose_inelasticity()` method should return an inelasticity value based on a random choice, and the `choose_shower_fractions()` method return calculate electromagnetic and hadronic fractions based on the `inelasticity` attribute storing the inelasticity value from `choose_inelasticity()`. The `choose_shower_fractions()` can be either chosen based on random processes like secondary generation or deterministic.

3.1.7 Particle Generator

The particle generator classes are quite flexible. The only requirement is that they possess an `create_event()` method which returns a *Event* object consisting of at least one *Particle*. The *BaseGenerator* class provides a solid foundation for basic uniform generators in a volume, requiring only implementation of the `get_vertex()` and `get_exit_points()` methods for the specific volume at a minimum.

EXAMPLE CODE

This section includes a number of more complete code examples for performing various tasks with PyREx. Each example includes a description of what it does, comments throughout describing the process, and a reference to the corresponding example script or notebook which can be run independent of one another. The examples are organized roughly from more basic to more complex.

4.1 Plot Detector Geometry

In this example we will make a few simple plots of the geometry of a detector object, handy for presentations or for visualizing your work. This code can be run from the Plot Detector notebook in the examples directory.

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import pyrex
import pyrex.custom.irex as irex

# First we need to initialize the detector object and build its antennas.
# For this example we'll just use a basic station geometry. Since we won't be
# throwing any particles at it, the arguments of the antennas are largely
# unimportant, but we will set up the antennas to alternately be oriented
# vertically or horizontally.
detector = irex.StationGrid(stations=4, station_type=irex.RegularStation,
                           antennas_per_string=4, antenna_separation=10)
def alternating_orientation(index, antenna):
    if index%2==0:
        return ((0,0,1), (1,0,0))
    else:
        return ((1,0,0), (0,0,1))
detector.build_antennas(trigger_threshold=0,
                       orientation_scheme=alternating_orientation)

# Let's also define a function which will highlight certain antennas in red.
# This one will highlight all antennas which are oriented horizontally.
def highlight(antenna_system):
    # Since the antennas in our detector are technically AntennaSystems,
    # to access the orientation we need to get the antenna object
    # which is a member of the AntennaSystem
    return np.dot(antenna_system.antenna.z_axis, (0,0,1)) == 0

# For our first plot, let's make a 3-D image of the whole detector.
fig = plt.figure()
```

(continues on next page)

(continued from previous page)

```

ax = fig.add_subplot(111, projection='3d')

# Plot the antennas which satisfy the highlight condition in red
xs = [ant.position[0] for ant in detector if highlight(ant)]
ys = [ant.position[1] for ant in detector if highlight(ant)]
zs = [ant.position[2] for ant in detector if highlight(ant)]
ax.scatter(xs, ys, zs, c="r")

# Plot the other antennas in black
xs = [ant.position[0] for ant in detector if not highlight(ant)]
ys = [ant.position[1] for ant in detector if not highlight(ant)]
zs = [ant.position[2] for ant in detector if not highlight(ant)]
ax.scatter(xs, ys, zs, c="k")

plt.show()

# Now let's plot the detector in a couple different 2-D angles.
# First, a top-down view of the entire detector.
plt.figure(figsize=(5, 5))

xs = [ant.position[0] for ant in detector if highlight(ant)]
ys = [ant.position[1] for ant in detector if highlight(ant)]
plt.scatter(xs, ys, c="r")

xs = [ant.position[0] for ant in detector if not highlight(ant)]
ys = [ant.position[1] for ant in detector if not highlight(ant)]
plt.scatter(xs, ys, c="k")

plt.title("Detector Geometry (Top View)")
plt.xlabel("x-position")
plt.ylabel("y-position")
plt.show()

# Next, let's take an x-z view of a single station. Let's also add in some
# string graphics by drawing lines from bottom antennas to the top of the ice.
plt.figure(figsize=(5, 5))

for station in detector.subsets:
    for string in station.subsets:
        lowest_antenna = sorted(string.subsets,
                                key=lambda ant: ant.position[2])[0]
        plt.plot([lowest_antenna.position[0], lowest_antenna.position[0]],
                  [lowest_antenna.position[2], 0], c="k", lw=1, zorder=-1)

xs = [ant.position[0] for ant in detector if highlight(ant)]
zs = [ant.position[2] for ant in detector if highlight(ant)]
plt.scatter(xs, zs, c="r", label="Horizontal")

xs = [ant.position[0] for ant in detector if not highlight(ant)]
zs = [ant.position[2] for ant in detector if not highlight(ant)]
plt.scatter(xs, zs, c="k", label="Vertical")

plt.xlim(200, 300)
plt.title("Single-Station Geometry (Side View)")
plt.xlabel("x-position")
plt.ylabel("z-position")
plt.legend()

```

(continues on next page)

(continued from previous page)

```
plt.show()
```

4.2 Askaryan Frequency Content

In this example we explore how the frequency spectrum of an Askaryan pulse changes as a function of the off-cone angle (i.e. the angular distance between the Cherenkov angle and the observation angle). This code can be run from the Frequency Content notebook in the examples directory.

```
import numpy as np
import matplotlib.pyplot as plt
import pyrex

# First, set up a neutrino source and find the index of refraction at its depth.
# Then use that index of refraction to calculate the Cherenkov angle.
source = pyrex.Particle("nu_e", vertex=(0, 0, -1000), direction=(0, 0, -1),
                        energy=1e8)
n = pyrex.IceModel.index(source.vertex[2])
ch_angle = np.arccos(1/n)

# Now, for a range of dthetas, generate an Askaryan pulse dtheta away from the
# Cherenkov angle and plot its frequency spectrum.
for dtheta in np.radians(np.logspace(-1, 1, 5)):
    n_pts = 10001
    pulse = pyrex.AskaryanSignal(times=np.linspace(-20e-9, 80e-9, n_pts),
                                particle=source,
                                viewing_angle=ch_angle-dtheta,
                                viewing_distance=1000)
    plt.plot(pulse.frequencies[:int(n_pts/2)] * 1e-6, # Convert from Hz to MHz
             np.abs(pulse.spectrum[:int(n_pts/2)]))
    plt.title("Frequency Spectrum of Askaryan Pulse\n"+
              str(round(np.degrees(dtheta),2))+" Degrees Off-Cone")
    plt.xlabel("Frequency (MHz)")
    plt.xlim(0, 3000)
    plt.show()

# Actually, we probably really want to see the frequency content after the
# signal has propagated through the ice a bit. So first set up the ray tracer
# from our neutrino source to some other point where our antenna might be
# (and make sure a path between those two points exists).
rt = pyrex.RayTracer(from_point=source.vertex, to_point=(500, 0, -100))
if not rt.exists:
    raise ValueError("Path to antenna doesn't exist!")

# Finally, plot the signal spectrum as it appears at the antenna position by
# propagating it along the (first solution) path.
path = rt.solutions[0]
for dtheta in np.radians(np.logspace(-1, 1, 5)):
    n_pts = 2048
    pulse = pyrex.AskaryanSignal(times=np.linspace(-20e-9, 80e-9, n_pts),
                                particle=source,
                                viewing_angle=ch_angle-dtheta,
                                viewing_distance=path.path_length)

    path.propagate(pulse)
    plt.plot(pulse.frequencies[:int(n_pts/2)] * 1e-6, # Convert from Hz to MHz
```

(continues on next page)

(continued from previous page)

```

        np.abs(pulse.spectrum)[:int(n_pts/2)])
    plt.title("Frequency Spectrum of Askaryan Pulse\n"+
             str(round(np.degrees(dtheta),2))+" Degrees Off-Cone")
    plt.xlabel("Frequency (MHz)")
    plt.xlim(0, 3000)
    plt.show()

# You may notice the sharp cutoff in the frequency spectrum above 1 GHz.
# This is due to the ice model, which defines the attenuation length in a
# piecewise manner for frequencies above or below 1 GHz.

```

4.3 Calculate Effective Area

In this example we will calculate the effective area of a detector over a range of energies. This code can be run from the Effective Area notebook in the examples directory.

Warning: In order to finish reasonably quickly, the number of events thrown in this example is low. This means that there are likely not enough events to accurately represent the effective area of the detector. For an accurate measurement, the number of events must be increased, but this will need much more time to run in that case.

```

import numpy as np
import matplotlib.pyplot as plt
import pyrex
import pyrex.custom.ara as ara

# First let's set the number of events that we will be throwing at each energy,
# and the energies we will be using. As stated in the warning, the number of
# events is set low to speed up the example, but that means the results are
# likely inaccurate. The energies are high to increase the chance of triggering.
n_events = 100
energies = [1e9, 2e9, 5e9, 1e10] # GeV

# Next, set up the detector to be measured. Here we use a single standard
# ARA station.
detector = ara.HexagonalGrid(station_type=ara.RegularStation,
                             stations=1)
detector.build_antennas(power_threshold=-6.15)

# Now set up a neutrino generator for each energy. We'll use unrealistically
# small volumes to increase the chance of triggering.
generators = [pyrex.CylindricalGenerator(dr=1000, dz=1000, energy=energy)
              for energy in energies]

# And then set up the event kernels for each energy. Let's use the ArasimIce
# class as our ice model since it calculates attenuations faster at the loss
# of some accuracy.
kernels = [pyrex.EventKernel(generator=gen, antennas=detector,
                             ice_model=pyrex.ice_model.ArasimIce())
           for gen in generators]

# Now run each kernel and record the number of events from each that triggered
# the detector. In this case we'll set our trigger condition to 3/8 antennas

```

(continues on next page)

(continued from previous page)

```

# triggering in a single polarization.
triggers = np.zeros(len(energies))
for i, kernel in enumerate(kernels):
    print("Running energy", energies[i])
    for j in range(n_events):
        print(j, "..", sep="", end="")
        detector.clear(reset_noise=True)
        particle = kernel.event()
        triggered = detector.triggered(station_requirement=1,
                                      polarized_antenna_requirement=3)

        if triggered:
            triggers[i] += 1
            print("y", end=" ")
        else:
            print("n", end=" ")

        if j%10==9:
            print(flush=True)
    print(triggers[i], "events triggered at", energies[i]/1e6, "PeV")
print("Done")

# Now that we have the trigger counts for each energy, we can calculate the
# effective volumes by scaling the trigger probability by the generation volume.
# Errors are calculated assuming poisson counting statistics.
generation_volumes = np.ones(4)*(np.pi*1000**2)*1000
effective_volumes = triggers / n_events * generation_volumes
volume_errors = np.sqrt(triggers) / n_events * generation_volumes

plt.errorbar(energies, effective_volumes, yerr=volume_errors,
             marker="o", markersize=5, linestyle=":", capsize=5)
ax = plt.gca()
ax.set_xscale("log")
ax.set_yscale("log")
plt.title("Detector Effective Volume")
plt.xlabel("Neutrino Energy (GeV)")
plt.ylabel("Effective Volume (km^3)")
plt.show()

# Then from the effective volumes, we can calculate the effective areas.
# The effective area is the probability interaction in the ice volume times the
# effective volume. The probability of interaction in the ice volume is given by
# the interaction cross section times the density of the ice. Calculate the
# cross section as an average of the neutrino and antineutrino cross sections.
cross_sections = np.zeros(len(energies))
for i, energy in enumerate(energies):
    nu = pyrex.Particle(particle_id="nu_e", vertex=(0, 0, 0),
                       direction=(0, 0, 1), energy=energy)
    nu_bar = pyrex.Particle(particle_id="nu_e_bar", vertex=(0, 0, 0),
                           direction=(0, 0, 1), energy=energy)
    cross_sections[i] = (nu.interaction.total_cross_section +
                        nu_bar.interaction.total_cross_section) / 2
ice_density = 0.92 # g/cm^3
ice_density *= 1e15 # converted to g/km^3 = nucleons/km^3
effective_areas = 6.022e23 * ice_density * cross_sections * effective_volumes
effective_areas *= 1e-4 # converted from cm^2 to m^2
area_errors = 6.022e23 * ice_density * cross_sections * volume_errors * 1e-4

```

(continues on next page)

(continued from previous page)

```
plt.errorbar(energies, effective_areas, yerr=area_errors,
             marker="o", markersize=5, linestyle=":", capsize=5)
ax = plt.gca()
ax.set_xscale("log")
ax.set_yscale("log")
plt.title("Detector Effective Area")
plt.xlabel("Neutrino Energy (GeV)")
plt.ylabel("Effective Area (m^2)")
plt.show()
```

4.4 Examine a Single Event

In this example we will generate a single event with a given vertex, direction, and energy, and then we'll examine the event by plotting the waveforms. This is typically useful for auditing events from a larger simulation. This code can be run from the Examine Event notebook in the examples directory.

```
import numpy as np
import matplotlib.pyplot as plt
import pyrex
import pyrex.custom.ara as ara

# First let's rebuild our detector that was used in the simulation.
det = ara.HexagonalGrid(station_type=ara.RegularStation,
                        stations=1, lowest_antenna=-100)
det.build_antennas(power_threshold=-6.15)

# Then let's plot a couple views of it just to be sure everything looks right.
fig, ax = plt.subplots(1, 2, figsize=(12, 5))
ax[0].scatter([ant.position[0] for ant in det],
              [ant.position[1] for ant in det],
              c='k')
ax[0].set_title("Detector Top View")
ax[0].set_xlabel("x-position")
ax[0].set_ylabel("y-position")
ax[1].scatter([ant.position[0] for ant in det],
              [ant.position[2] for ant in det],
              c='k')
ax[1].set_title("Detector Side View")
ax[1].set_xlabel("x-position")
ax[1].set_ylabel("z-position")
plt.show()

# Now set up a particle generator that will just throw the one event we're
# interested in, and create an event kernel with our detector and our generator.
p = pyrex.Particle(particle_id=pyrex.Particle.Type.electron_neutrino,
                  vertex=[1002.65674195, -421.95118348, -586.0953201],
                  direction=[-0.90615395, -0.41800062, -0.06450191],
                  energy=1e9)
p.interaction.kind = p.interaction.Type.charged_current
p.interaction.em_frac = 1
p.interaction.had_frac = 0
gen = pyrex.ListGenerator(pyrex.Event(p))
kern = pyrex.EventKernel(antennas=det, generator=gen)
```

(continues on next page)

(continued from previous page)

```

# Then make sure our detector is cleared out and throw the event!
# reset_noise will make sure we get new noise waveforms every time.
det.clear(reset_noise=True)
kern.event()

# Now let's take a look at the waveforms of the event. Since each event has a
# first and second ray, plot their waveforms side-by-side for each antenna.
for i, ant in enumerate(det):
    fig, ax = plt.subplots(1, 2, figsize=(12, 3))
    for j, wave in enumerate(ant.all_waveforms):
        ax[j].plot(wave.times*1e9, wave.values)
        ax[j].set_xlabel("Time (ns)")
        ax[j].set_ylabel("Amplitude (V)")
        ax[j].set_title("First Ray" if j%2==0 else "Second Ray")
    fig.suptitle("String "+str(int(i/4))+" "+ant.name)
    plt.show()

# From the plots it looks like the first ray is the one that triggered the
# detector. Let's calculate a signal-to-noise ratio of the first-ray waveform
# for each antenna.
print("Signal-to-noise ratios:")
for i, ant in enumerate(det):
    wave = ant.all_waveforms[0]
    signal_pp = np.max(wave.values) - np.min(wave.values)
    noise = ant.front_end(ant.antenna.make_noise(wave.times))
    noise_rms = np.sqrt(np.mean(noise.values**2))
    print(" String "+str(int(i/4))+" "+ant.name+":", signal_pp/(2*noise_rms))

# Let's also take a look at the trigger condition, which passes the waveform
# through a tunnel diode. Again we can plot the tunnel diode's integrated
# waveform for each ray side-by-side. The red lines indicate the trigger level.
# If the integrated waveform goes beyond those lines the antenna is triggered.
for i, ant in enumerate(det):
    fig, ax = plt.subplots(1, 2, figsize=(12, 3))
    for j, wave in enumerate(ant.all_waveforms):
        triggered = ant.trigger(wave)
        trigger_wave = ant.tunnel_diode(wave)
        # The first time ant.trigger is run for an antenna, the power mean and
        # rms are calculated which will determine the trigger condition.
        low_trigger = (ant._power_mean -
                       ant._power_rms*np.abs(ant.power_threshold))
        high_trigger = (ant._power_mean +
                       ant._power_rms*np.abs(ant.power_threshold))
        ax[j].plot(trigger_wave.times*1e9, trigger_wave.values)
        ax[j].axhline(low_trigger, color='r')
        ax[j].axhline(high_trigger, color='r')
        ax[j].set_title("Triggered" if triggered else "Missed")
        ax[j].set_xlabel("Time (ns)")
        ax[j].set_ylabel("Integrated Power (V^2)")
    fig.suptitle("String "+str(int(i/4))+" "+ant.name)
    plt.show()

# Finally, let's look at the relative trigger times to make sure they look
# reasonable. We could get the true relative trigger times from the waveforms
# by just taking the differences of their first times, but instead let's
# pretend we're doing an analysis and just use the times of the maxima.
trig_times = []

```

(continues on next page)

(continued from previous page)

```
for ant in det:
    wave = ant.all_waveforms[0]
    trig_times.append(wave.times[np.argmax(np.abs(wave.values))])

# Then we can plot the progression of the event by coloring the antennas where
# red is the earliest time and blue/purple is the latest time.
fig, ax = plt.subplots(3, 1, figsize=(5, 16))
ax[0].scatter([ant.position[0] for ant in det],
              [ant.position[1] for ant in det],
              c=trig_times, cmap='rainbow_r')
ax[0].set_title("Detector Top View")
ax[0].set_xlabel("x-position")
ax[0].set_ylabel("y-position")
ax[1].scatter([ant.position[0] for ant in det],
              [ant.position[2] for ant in det],
              c=trig_times, cmap='rainbow_r')
ax[1].set_title("Detector Side View")
ax[1].set_xlabel("x-position")
ax[1].set_ylabel("z-position")
ax[2].scatter([ant.position[1] for ant in det],
              [ant.position[2] for ant in det],
              c=trig_times, cmap='rainbow_r')
ax[2].set_title("Detector Side View 2")
ax[2].set_xlabel("y-position")
ax[2].set_ylabel("z-position")
plt.show()
```


CONTRIBUTING TO PYREX

PyREx is currently being maintained by [Ben Hokanson-Fasig](#). Any direct contributions to the code base should be made through GitHub as described in the following sections, and will be reviewed by the maintainer or another approved reviewer. Note that contributions are also possible less formally through the creation of custom plug-ins, as described in [Custom Sub-Package](#).

5.1 Branching Model

PyREx code contributions should follow a specific git branching model sometimes referred to as the [Gitflow Workflow](#). In this model the `master` branch is reserved for release versions of the code, and most development takes place in feature branches which merge back to the `develop` branch.

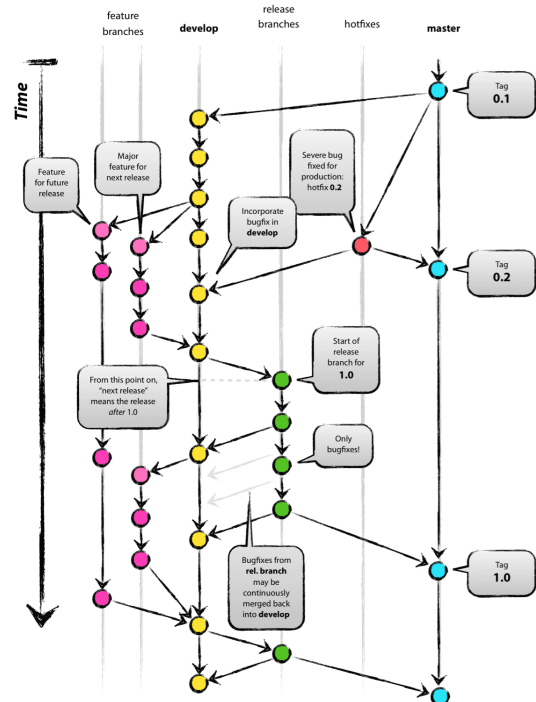
The basic steps to add a feature are as follows:

1. From the `develop` branch, create a new feature branch.
2. In your feature branch, write the code.
3. Merge the feature branch back into the `develop` branch.
4. Delete the feature branch.

Then when it comes time for the next release, the maintainer will:

1. Create a release branch from the `develop` branch.
2. Document the changes for the new version.
3. Make any bug fixes necessary.
4. Merge the release branch into the `master` branch.
5. Tag the release with the version number.
6. Merge the release branch back into the `develop` branch.
7. Delete the release branch.

In order to make these processes easier, two shell scripts `feature.sh` and `release.sh` were created to automate the steps of the above processes respectively. The use of these scripts is defined in the following sections.



5.2 Contributing via Pull Request

The preferred method of contributing code to PyREx is to submit a pull request on GitHub. The general process for doing this is as follows:

First, if you haven't already you will need to fork the repository so that you have a copy of the code in which you can make your changes. This can be done by visiting <https://github.com/bhokansonfasig/pyrex/> and clicking the Fork button in the upper-right.

Next you likely want to clone the repository onto your computer to edit the code. To do this, visit your fork on GitHub and click the Clone or download button and in your terminal run the git clone command with the copied link.

```
git clone https://github.com/YOUR-USERNAME/NAME-OF-FORKED-REPO
```

If you want your local clone to stay synced with the main PyREx repository, then you can set up an [upstream remote](#).

Now before changing the code, you need to create a feature branch in which you can work. To do this, use the `feature.sh` script with the new action:

```
./feature.sh new feature-branch-name
```

This will create a new branch for you with the name you give it, and it will push the branch to GitHub. The name you use for your feature branch (in place of `feature-branch-name` above) should be a relatively short name, all lowercase with hyphens between words, and descriptive of the feature you are adding. If you would prefer that the branch not be pushed to GitHub immediately, you can use the `private` action in place of `new` in the command above.

Now that you have a feature branch set up, you can write the code for the new feature in this branch. Once you've implemented (and tested!) the feature and you're ready for it to be added to PyREx, submit a pull request to the PyREx repository. To do this, go back to <https://github.com/bhokansonfasig/pyrex/> and click the New pull request button. On the Compare changes page, click compare across forks. The base fork should be the main PyREx repository, the base branch should be `develop`, the head fork should be your fork of PyREx, and the compare branch should be your newly finished feature branch. Then after adding a title and description of your new feature, click Create pull request.

The last step is for the maintainer and other reviewers to review your code and either suggest changes or accept the pull request, at which point your code will be integrated for the next PyREx release!

5.3 Contributing with Direct Access

If you have direct access to the PyREx repository on GitHub, you can make changes without the need for a pull request. In this case the first step is to create a new feature branch with `feature.sh` as described above:

```
./feature.sh new feature-branch-name
```

Now in the feature branch, write and test your new code. Once that's finished you can merge the feature branch back using the `merge` action of `feature.sh`:

```
./feature.sh merge feature-branch-name
```

Note that (as long as the merge is successful) this also deletes the feature branch locally and on GitHub.

5.4 Releasing a New Version

If you are the maintainer of the code base (or were appointed by the maintainer to handle releases), then you will be responsible for creating and merging release branches to the `master` branch. This process is streamlined using the `release.sh` script. When it's time for a new release of the code, start by using the script to create a new release branch:

```
./release.sh new X.Y.Z
```

This creates a new branch named `release-X.Y.Z` where `X.Y.Z` is the release version number. Note that version numbers should follow [Semantic Versioning](#), and if alpha, beta, release candidate, or other pre-release versions are necessary, lowercase letters may be added to the end of the version number. Additionally if creating a hotfix branch rather than a proper release, that can be specified at the end of the `release.sh` call:

```
./release.sh new X.Y.Z hotfix
```

Once the new release branch is created, the first commit to the branch should consist only of a change to the version number in the code so that it matches the release version number. This commit should have the message “Bumped version number to X.Y.Z”.

The next step is to document all changes in the new release in the version history documentation. To help with this, `release.sh` prints out a list of all the commits since the last release. If you need to see this list again, you can use

```
git log master..release-X.Y.Z --oneline --no-merges
```

Once the documentation is up to date with all the changes (including updating any places in the usage or the examples which may have become outdated), do some bug testing and be sure that all code tests are passing. Then when you're sure the release is ready you can merge the release branch into the `master` and `develop` branches with

```
./release.sh merge X.Y.Z
```

This script will handle tagging the release and will delete the local release branch. If the release branch ended up pushed to GitHub at some point, it will need to be deleted there either through their interface or using

```
git push -d origin release-X.Y.Z
```

PYREX API

The API documentation here is split into three sections. First, the *PyREx Package Imports* section documents all classes and functions that are imported by PyREx under a `from pyrex import *` command. Next, the *Individual Module APIs* section is a full documentation of all the modules which make up the base PyREx package. And finally, the *Included Custom Sub-Packages* section documents the custom subpackages included with PyREx by default.

6.1 PyREx Package Imports

<i>Signal</i>	Base class for time-domain signals.
<i>EmptySignal</i>	Class for signal with zero amplitude (all values = 0).
<i>FunctionSignal</i>	Class for signals generated by a function.
<i>AskaryanSignal</i>	alias of <i>pyrex.askaryan.ARVZAskaryanSignal</i>
<i>ThermalNoise</i>	Class for thermal Rayleigh noise signals.
<i>Antenna</i>	Base class for antennas.
<i>DipoleAntenna</i>	Class for half-wave dipole antennas.
<i>AntennaSystem</i>	Base class for antenna system with front-end processing.
<i>Detector</i>	Base class for detectors for easily building up sets of antennas.
<i>ice</i>	Class describing the ice at the south pole.
<i>prem_density</i>	Calculates the Earth's density at a given radius.
<i>slant_depth</i>	Calculates the material thickness of a chord cutting through Earth.
<i>NeutrinoInteraction</i>	alias of <i>pyrex.particle.CTWInteraction</i>
<i>Particle</i>	Class for storing particle attributes.
<i>Event</i>	Class for storing a tree of <i>Particle</i> objects representing an event.
<i>CylindricalGenerator</i>	Class to generate neutrino vertices in a cylindrical ice volume.
<i>RectangularGenerator</i>	Class to generate neutrino vertices in a rectangular ice volume.
<i>ListGenerator</i>	Class to generate neutrino events from a list.
<i>FileGenerator</i>	Class to generate neutrino events from simulation file(s).
<i>RayTracer</i>	alias of <i>pyrex.ray_tracing.SpecializedRayTracer</i>

Continued on next page

Table 1 – continued from previous page

<i>RayTracePath</i>	alias of <i>pyrex.ray_tracing.SpecializedRayTracePath</i>
<i>EventKernel</i>	High-level kernel for controlling event simulation.
<i>File</i>	Class for reading or writing data files.

6.1.1 pyrex.Signal

class `pyrex.Signal` (*times*, *values*, *value_type=None*)

Base class for time-domain signals.

Stores the time-domain information for signal values. Supports adding between signals with the same times array and value type.

Parameters

times [array_like] 1D array of times (s) for which the signal is defined.

values [array_like] 1D array of values of the signal corresponding to the given *times*. Will be resized to the size of *times* by zero-padding or truncating as necessary.

value_type [optional] Type of signal, representing the units of the values. Values should be from the `Signal.Type` enum, but integer or string values may work if carefully chosen. `Signal.Type.undefined` by default.

Attributes

times, **values** [ndarray] 1D arrays of times (s) and corresponding values which define the signal.

value_type [Signal.Type] Type of signal, representing the units of the values.

Type [Enum] Enum containing possible types (units) for signal values.

dt The time spacing of the *times* array, or `None` if invalid.

frequencies The FFT frequencies of the signal.

spectrum The FFT complex spectrum values of the signal.

envelope The envelope of the signal by Hilbert transform.

Methods

Type	Enum containing possible types (units) for signal values.
<code>filter_frequencies(freq_response[, force_real])</code>	Apply the given frequency response function to the signal, in-place.
<code>resample(n)</code>	Resamples the signal into n points in the same time range, in-place.
<code>with_times(new_times)</code>	Returns a representation of this signal over a different times array.

6.1.2 pyrex.EmptySignal

class `pyrex.EmptySignal` (*times*, *value_type=None*)

Class for signal with zero amplitude (all values = 0).

Parameters

times [array_like] 1D array of times (s) for which the signal is defined.

value_type [optional] Type of signal, representing the units of the values. Must be from the `Signal.Type` Enum.

See also:

Signal Base class for time-domain signals.

Attributes

times, values [ndarray] 1D arrays of times (s) and corresponding values which define the signal.

value_type [Signal.Type] Type of signal, representing the units of the values.

Type [Enum] Enum containing possible types (units) for signal values.

dt The time spacing of the *times* array, or `None` if invalid.

frequencies The FFT frequencies of the signal.

spectrum The FFT complex spectrum values of the signal.

envelope The envelope of the signal by Hilbert transform.

Methods

Type	Enum containing possible types (units) for signal values.
filter_frequencies(freq_response[, force_real])	Apply the given frequency response function to the signal, in-place.
resample(n)	Resamples the signal into n points in the same time range, in-place.
with_times(new_times)	Returns a representation of this signal over a different times array.

6.1.3 pyrex.FunctionSignal

class `pyrex.FunctionSignal` (*times, function, value_type=None*)

Class for signals generated by a function.

Parameters

times [array_like] 1D array of times (s) for which the signal is defined.

function [function] Function which evaluates the corresponding value(s) for a given time or array of times.

value_type [optional] Type of signal, representing the units of the values. Must be from the `Signal.Type` Enum.

See also:

Signal Base class for time-domain signals.

EmptySignal Class for signal with zero amplitude.

Attributes

times, values [ndarray] 1D arrays of times (s) and corresponding values which define the signal.

value_type [Signal.Type] Type of signal, representing the units of the values.

Type [Enum] Enum containing possible types (units) for signal values.

function [function] Function to evaluate the signal values at given time(s).

dt The time spacing of the *times* array, or *None* if invalid.

frequencies The FFT frequencies of the signal.

spectrum The FFT complex spectrum values of the signal.

envelope The envelope of the signal by Hilbert transform.

Methods

Type	Enum containing possible types (units) for signal values.
filter_frequencies(freq_response[, force_real])	Apply the given frequency response function to the signal, in-place.
resample(n)	Resamples the signal into n points in the same time range, in-place.
with_times(new_times)	Returns a representation of this signal over a different times array.

6.1.4 pyrex.AskaryanSignal

`pyrex.AskaryanSignal`

alias of `pyrex.askaryan.ARVZAskaryanSignal`

6.1.5 pyrex.ThermalNoise

class `pyrex.ThermalNoise` (*times*, *f_band*, *f_amplitude=None*, *rms_voltage=None*, *temperature=None*, *resistance=None*, *n_freqs=0*)

Class for thermal Rayleigh noise signals.

The Rayleigh thermal noise is calculated in a given frequency band with flat or otherwise specified amplitude and random phase at some number of frequencies. Values are scaled to a provided or calculated RMS voltage.

Parameters

times [array_like] 1D array of times (s) for which the signal is defined.

f_band [array_like] Array of two elements denoting the frequency band (Hz) of the noise. The first element should be smaller than the second.

f_amplitude [float or function, optional] The frequency-domain amplitude of the noise. If *float*, then all frequencies will have the same amplitude. If *function*, then the function is evaluated at each frequency to determine its amplitude. By default, uses Rayleigh-distributed amplitudes.

rms_voltage [float, optional] The RMS voltage (V) of the noise. If specified, this value will be used instead of the RMS voltage calculated from the values of *temperature* and *resistance*.

temperature [float, optional] The thermal noise temperature (K). Used in combination with the value of *resistance* to calculate the RMS voltage of the noise.

resistance [float, optional] The resistance (ohm) for the noise. Used in combination with the value of *temperature* to calculate the RMS voltage of the noise.

n_freqs [int, optional] The number of frequencies within the frequency band to use to calculate the noise signal. By default determines the number of frequencies based on the FFT bin size of *times*.

Raises

ValueError If the RMS voltage cannot be calculated (i.e. *rms_voltage* or both *temperature* and *resistance* are `None`).

Warning: Since this class inherits from `FunctionSignal`, its `with_times` method will properly extrapolate noise outside of the provided times. Be warned however that outside of the original signal times the noise signal will be highly periodic. Since the default number of frequencies used is based on the FFT bin size of *times*, the period of the noise signal is actually the length of *times*. As a result if you are planning on extrapolating the noise signal, increasing the number of frequencies used is strongly recommended.

See also:

`FunctionSignal` Class for signals generated by a function.

Notes

Calculation of the noise signal is based on the Rayleigh noise model used by ANITA [1]. Modifications have been made to the default to make the frequency-domain amplitudes Rayleigh-distributed, under the suggestion that this makes for more realistic noise traces.

References

[1]

Attributes

times, values [ndarray] 1D arrays of times (s) and corresponding values which define the signal.

value_type [Signal.Type.voltage] Type of signal, representing the units of the values.

Type [Enum] Enum containing possible types (units) for signal values.

function [function] Function to evaluate the signal values at given time(s).

f_min [float] Minimum frequency of the noise frequency band.

f_max [float] Maximum frequency of the noise frequency band.

freqs, amps, phases [ndarray] The frequencies used to define the noise signal and their corresponding amplitudes and phases.

rms [float] The RMS value of the noise signal.

dt The time spacing of the *times* array, or `None` if invalid.

frequencies The FFT frequencies of the signal.

spectrum The FFT complex spectrum values of the signal.

envelope The envelope of the signal by Hilbert transform.

Methods

Type	Enum containing possible types (units) for signal values.
filter_frequencies(freq_response[, force_real])	Apply the given frequency response function to the signal, in-place.
resample(n)	Resamples the signal into n points in the same time range, in-place.
with_times(new_times)	Returns a representation of this signal over a different times array.

6.1.6 pyrex.Antenna

class `pyrex.Antenna` (*position*, *z_axis*=(0, 0, 1), *x_axis*=(1, 0, 0), *antenna_factor*=1, *efficiency*=1, *noisy*=True, *unique_noise_waveforms*=10, *freq_range*=None, *temperature*=None, *resistance*=None, *noise_rms*=None)

Base class for antennas.

Stores the attributes of an antenna as well as handling receiving, processing, and storing signals and adding noise.

Parameters

- position** [array_like] Vector position of the antenna.
- z_axis** [array_like, optional] Vector direction of the z-axis of the antenna.
- x_axis** [array_like, optional] Vector direction of the x-axis of the antenna.
- antenna_factor** [float, optional] Antenna factor used for converting electric field values to voltages.
- efficiency** [float, optional] Antenna efficiency applied to incoming signal values.
- noisy** [boolean, optional] Whether or not the antenna should add noise to incoming signals.
- unique_noise_waveforms** [int, optional] The number of expected noise waveforms needed for each received signal to have its own noise.
- freq_range** [array_like, optional] The frequency band in which the antenna operates (used for noise production).
- temperature** [float, optional] The noise temperature (K) of the antenna. Used in combination with *resistance* to calculate the RMS voltage of the antenna noise.
- resistance** [float, optional] The noise resistance (ohm) of the antenna. Used in combination with *temperature* to calculate the RMS voltage of the antenna noise.
- noise_rms** [float, optional] The RMS voltage (V) of the antenna noise. If specified, this value will be used instead of the RMS voltage calculated from the values of *temperature* and *resistance*.

Attributes

- position** [array_like] Vector position of the antenna.
- z_axis** [ndarray] Vector direction of the z-axis of the antenna.
- x_axis** [ndarray] Vector direction of the x-axis of the antenna.
- antenna_factor** [float] Antenna factor used for converting electric field values to voltages.

efficiency [float] Antenna efficiency applied to incoming signal values.

noisy [boolean] Whether or not the antenna should add noise to incoming signals.

unique_noises [int] The number of expected noise waveforms needed for each received signal to have its own noise.

freq_range [array_like] The frequency band in which the antenna operates (used for noise production).

temperature [float or None] The noise temperature (K) of the antenna. Used in combination with *resistance* to calculate the RMS voltage of the antenna noise.

resistance [float or None] The noise resistance (ohm) of the antenna. Used in combination with *temperature* to calculate the RMS voltage of the antenna noise.

noise_rms [float or None] The RMS voltage (v) of the antenna noise. If not `None`, this value will be used instead of the RMS voltage calculated from the values of *temperature* and *resistance*.

signals [list of Signal] The signals which have been received by the antenna.

is_hit Boolean of whether the antenna has been triggered.

is_hit_mc_truth Boolean of whether the antenna has been triggered by signal.

waveforms Signal + noise (if *noisy*) for each triggered antenna hit.

all_waveforms Signal + noise (if *noisy*) for all antenna hits.

Methods

<code>apply_response(signal[, direction, ...])</code>	Process the complete antenna response for an incoming signal.
<code>clear([reset_noise])</code>	Reset the antenna to an empty state.
<code>directional_gain(theta, phi)</code>	Calculate the (complex) directional gain of the antenna.
<code>frequency_response(frequencies)</code>	Calculate the (complex) frequency response of the antenna.
<code>full_waveform(times)</code>	Signal + noise (if <i>noisy</i>) for the given times.
<code>is_hit_during(times)</code>	Check if the antenna is triggered in a time range.
<code>make_noise(times)</code>	Creates a noise signal over the given times.
<code>polarization_gain(polarization)</code>	Calculate the (complex) polarization gain of the antenna.
<code>receive(signal[, direction, polarization, ...])</code>	Process and store one or more incoming (polarized) signals.
<code>set_orientation([z_axis, x_axis])</code>	Sets the orientation of the antenna.
<code>trigger(signal)</code>	Check if the antenna triggers on a given signal.

6.1.7 pyrex.DipoleAntenna

class `pyrex.DipoleAntenna` (*name, position, center_frequency, bandwidth, resistance, orientation=(0, 0, 1), trigger_threshold=0, effective_height=None, noisy=True, unique_noise_waveforms=10*)

Class for half-wave dipole antennas.

Stores the attributes of an antenna as well as handling receiving, processing, and storing signals and adding noise. Uses a first-order butterworth filter for the frequency response. Includes a simple threshold trigger.

Parameters

- name** [str] Name of the antenna.
- position** [array_like] Vector position of the antenna.
- center_frequency** [float] Tuned frequency (Hz) of the dipole.
- bandwidth** [float] Bandwidth (Hz) of the antenna.
- resistance** [float] The noise resistance (ohm) of the antenna. Used to calculate the RMS voltage of the antenna noise.
- orientation** [array_like, optional] Vector direction of the z-axis of the antenna.
- trigger_threshold** [float, optional] Voltage threshold (V) above which signals will trigger.
- effective_height** [float, optional] Effective length (m) of the antenna. By default calculated by the tuned *center_frequency* of the dipole.
- noisy** [boolean, optional] Whether or not the antenna should add noise to incoming signals.
- unique_noise_waveforms** [int, optional] The number of expected noise waveforms needed for each received signal to have its own noise.

See also:

[*Antenna*](#) Base class for antennas.

Attributes

- name** [str] Name of the antenna.
- position** [array_like] Vector position of the antenna.
- z_axis** [ndarray] Vector direction of the z-axis of the antenna.
- x_axis** [ndarray] Vector direction of the x-axis of the antenna.
- antenna_factor** [float] Antenna factor used for converting electric field values to voltages.
- efficiency** [float] Antenna efficiency applied to incoming signal values.
- threshold** [float, optional] Voltage threshold (V) above which signals will trigger.
- effective_height** [float, optional] Effective length of the antenna. By default calculated by the tuned *center_frequency* of the dipole.
- filter_coeffs** [tuple of ndarray] Coefficients of the transfer function of the butterworth bandpass filter to be used for frequency response.
- noisy** [boolean] Whether or not the antenna should add noise to incoming signals.
- unique_noises** [int] The number of expected noise waveforms needed for each received signal to have its own noise.
- freq_range** [array_like] The frequency band in which the antenna operates (used for noise production).
- temperature** [float or None] The noise temperature (K) of the antenna. Used in combination with *resistance* to calculate the RMS voltage of the antenna noise.
- resistance** [float or None] The noise resistance (ohm) of the antenna. Used in combination with *temperature* to calculate the RMS voltage of the antenna noise.

noise_rms [float or None] The RMS voltage (V) of the antenna noise. If not `None`, this value will be used instead of the RMS voltage calculated from the values of *temperature* and *resistance*.

signals [list of Signal] The signals which have been received by the antenna.

is_hit Boolean of whether the antenna has been triggered.

is_hit_mc_truth Boolean of whether the antenna has been triggered by signal.

waveforms Signal + noise (if `noisy`) for each triggered antenna hit.

all_waveforms Signal + noise (if `noisy`) for all antenna hits.

Methods

<code>apply_response(signal[, direction, ...])</code>	Process the complete antenna response for an incoming signal.
<code>clear([reset_noise])</code>	Reset the antenna to an empty state.
<code>directional_gain(theta, phi)</code>	Calculate the (complex) directional gain of the antenna.
<code>frequency_response(frequencies)</code>	Calculate the (complex) frequency response of the antenna.
<code>full_waveform(times)</code>	Signal + noise (if <code>noisy</code>) for the given times.
<code>is_hit_during(times)</code>	Check if the antenna is triggered in a time range.
<code>make_noise(times)</code>	Creates a noise signal over the given times.
<code>polarization_gain(polarization)</code>	Calculate the (complex) polarization gain of the antenna.
<code>receive(signal[, direction, polarization, ...])</code>	Process and store one or more incoming (polarized) signals.
<code>set_orientation([z_axis, x_axis])</code>	Sets the orientation of the antenna.
<code>trigger(signal)</code>	Check if the antenna triggers on a given signal.

6.1.8 pyrex.AntennaSystem

class `pyrex.AntennaSystem` (*antenna*)

Base class for antenna system with front-end processing.

Behaves similarly to an antenna by passing some functionality downward to an antenna class, but additionally applies some front-end processing (e.g. an electronics chain) to the signals received.

Parameters

antenna [Antenna] Antenna class or subclass to be extended with a front end. Can also accept an Antenna object directly.

See also:

[`pyrex.Antenna`](#) Base class for antennas.

Attributes

antenna [Antenna] Antenna object extended by the front end.

lead_in_time [float] Lead-in time (s) required for the front end to equilibrate. Automatically added in before calculation of signals and waveforms.

is_hit Boolean of whether the antenna system has been triggered.

is_hit_mc_truth Boolean of whether the antenna has been triggered by signal.

signals The signals received by the antenna with front-end processing.

waveforms The antenna system signal + noise for each triggered hit.

all_waveforms The antenna system signal + noise for all hits.

Methods

<code>apply_response(signal[, direction, ...])</code>	Process the complete antenna response for an incoming signal.
<code>clear([reset_noise])</code>	Reset the antenna system to an empty state.
<code>front_end(signal)</code>	Apply front-end processes to a signal and return the output.
<code>full_waveform(times)</code>	Signal + noise for the antenna system for the given times.
<code>is_hit_during(times)</code>	Check if the antenna system is triggered in a time range.
<code>make_noise(times)</code>	Creates a noise signal over the given times.
<code>receive(signal[, direction, polarization, ...])</code>	Process and store one or more incoming (polarized) signals.
<code>set_orientation([z_axis, x_axis])</code>	Sets the orientation of the antenna system.
<code>setup_antenna(*args, **kwargs)</code>	Setup the antenna by passing along its init arguments.
<code>trigger(signal)</code>	Check if the antenna system triggers on a given signal.

6.1.9 pyrex.Detector

class `pyrex.Detector(*args, **kwargs)`

Base class for detectors for easily building up sets of antennas.

Designed for automatically generating antenna positions based on geometry parameters, then building all the antennas with some properties. Any parameters to the `__init__` method are automatically passed on to the `set_positions` method. Once the antennas have been built, the object can be directly iterated over to iterate over the antennas (as if the object were just a list of the antennas).

Raises

ValueError If `test_antenna_positions` is `True` and an antenna is found to be above the ice surface.

See also:

`pyrex.Antenna` Base class for antennas.

`AntennaSystem` Base class for antenna system with front-end processing.

Notes

When this class is subclassed, the `__init__` method will mirror the signature of the `set_positions` method so that parameters can be easily discovered.

The class is designed to be flexible in what defines a “detector”. This should allow for easier modularization by defining detectors whose subsets are detectors themselves, and so on. For example, a string of antennas could be set up as a subclass of *Detector* which sets up some antennas in a vertical line. Then a station could be set up as a subclass of *Detector* which sets up multiple instances of the string class at different positions. Then a final overarching detector class can subclass *Detector* and set up multiple instances of the station class at different positions. In this example the `subsets` of the overarching detector class would be the station objects, the `subsets` of the station objects would be the string objects, and the `subsets` of the string objects would finally be the antenna objects. But the way the iteration of the *Detector* class is built, iterating over that overarching detector class would iterate directly over each antenna in each string in each station as a simple 1D list.

Attributes

- antenna_positions** [list] List (potentially with sub-lists) of the positions of the antennas generated by the `set_positions` method.
- subsets** [list] List of the antenna or detector objects which make up the detector.
- test_antenna_positions** [boolean] Class attribute for whether or not an error should be raised if antenna positions are found above the surface of the ice (where simulation behavior is ill-defined). Defaults to `True`.

Methods

<code>build_antennas(*args, **kwargs)</code>	Creates antenna objects at the set antenna positions.
<code>clear([reset_noise])</code>	Reset the detector to an empty state.
<code>set_positions(*args, **kwargs)</code>	Sets the positions of antennas in the detector.
<code>triggered(*args[, require_mc_truth])</code>	Check if the detector is triggered based on its current state.

6.1.10 pyrex.ice

`pyrex.ice = <pyrex.ice_model.AntarcticIce object>`

Class describing the ice at the south pole.

For convenience, consists of static methods and class methods, so creating an instance of the class may not be necessary. In all methods, the depth `z` should be given as a negative value if it is below the surface of the ice.

Parameters

- n0** [float, optional] Asymptotic index of refraction of the deep ice.
- k** [float, optional] Multiplicative factor for the index of refraction parameterization.
- a** [float, optional] Exponential factor for the index of refraction parameterization with units of 1/m.
- valid_range** [array_like of float, optional] Range of depths over which the uniform index of refraction applies. Assumed to have two elements where the first value is lower (deeper, more negative) than the second.
- index_above** [float or None, optional] Index of refraction above the ice region. If *None*, uses the same index of refraction as the top of the ice.
- index_below** [float or None, optional] Index of refraction below the ice region. If *None*, uses the same index of refraction as the bottom of the ice.

Notes

Parameterizations mostly based on South Pole ice characteristics outlined by Matt Newcomb [1].

References

[1]

Attributes

n0, k, a [float] Parameters of the index of refraction of the ice.

valid_range [tuple] Range of depths over which the ice model is valid. Consists of two elements where the first value is lower (deeper, more negative) than the second.

index_above

index_below

6.1.11 pyrex.prem_density

`pyrex.prem_density(r)`

Calculates the Earth's density at a given radius.

Density from the Preliminary reference Earth Model (PREM). Supports passing an array of radii or a single radius.

Parameters

r [array_like] Radius (m) at which to calculate density.

Returns

array_like Density (g/cm³) of the Earth at the given radii.

Notes

The density calculation is based on the Preliminary reference Earth Model [1].

References

[1]

6.1.12 pyrex.slant_depth

`pyrex.slant_depth(angle, depth, step=500)`

Calculates the material thickness of a chord cutting through Earth.

Integrates the Earth's density along the chord. Uses the PREM model for density.

Parameters

angle [float] Nadir angle (radians) of the chord's direction.

depth [float] (Positive-valued) depth (m) of the chord endpoint.

step [float, optional] Step size (m) for the integration.

Returns

float Material thickness (g/cm²) along the chord starting from *depth* and passing through the Earth at *angle*.

See also:

prem_density Calculates the Earth's density at a given radius.

6.1.13 pyrex.NeutrinoInteraction

`pyrex.NeutrinoInteraction`

alias of `pyrex.particle.CTWInteraction`

6.1.14 pyrex.Particle

class `pyrex.Particle`(*particle_id*, *vertex*, *direction*, *energy*, *interaction_model*=<class 'pyrex.particle.CTWInteraction'>, *interaction_type*=None, *weight*=None)

Class for storing particle attributes.

Parameters

particle_id Identification value of the particle type. Values should be from the `Particle.Type` enum, but integer or string values may work if carefully chosen. `Particle.Type` undefined by default.

vertex [array_like] Vector position (m) of the particle.

direction [array_like] Vector direction of the particle's velocity.

energy [float] Energy (GeV) of the particle.

interaction_model [optional] Class to use to describe interactions of the particle. Should inherit from (or behave like) the base `Interaction` class.

interaction_type [optional] Value of the interaction type. Values should be from the `Interaction.Type` enum, but integer or string values may work if carefully chosen. By default, the *interaction_model* will choose an interaction type.

weight [float, optional] Total Monte Carlo weight of the particle. The calculation of this weight depends on the particle generation method, but this value should be the total weight representing the probability of this particle's event occurring.

See also:

Interaction Base class for describing neutrino interaction attributes.

Attributes

id [`Particle.Type`] Identification value of the particle type.

vertex [array_like] Vector position (m) of the particle.

direction [array_like] (Unit) vector direction of the particle's velocity.

energy [float] Energy (GeV) of the particle.

interaction [`Interaction`] Instance of the *interaction_model* class to be used for calculations related to interactions of the particle.

weight [float] Total Monte Carlo weight of the particle

survival_weight [float] Monte Carlo weight of the particle surviving to its vertex. Represents the probability that the particle does not interact along its path through the Earth.

interaction_weight [float] Monte Carlo weight of the particle interacting at its vertex. Represents the probability that the particle interacts specifically at its given vertex.

Methods

Type	Enum containing possible particle types.
------	--

6.1.15 pyrex.Event

class `pyrex.Event` (*roots*)

Class for storing a tree of *Particle* objects representing an event.

The event may be comprised of any number of root *Particle* objects specified at initialization. Each *Particle* in the tree may have any number of child *Particle* objects. Iterating the tree will return all *Particle* objects, but in no guaranteed order.

Parameters

roots [Particle or list of Particle] Root *Particle* objects for the event tree.

See also:

Particle Class for storing particle attributes.

Attributes

roots [Particle or list of Particle] Root *Particle* objects for the event tree.

Methods

<code>add_children(parent, children)</code>	Add the given <i>children</i> to the <i>parent Particle</i> object.
<code>get_children(parent)</code>	Get the children of the given <i>parent Particle</i> object.
<code>get_from_level(level)</code>	Get all <i>Particle</i> objects some <i>level</i> deep into the event tree.
<code>get_parent(child)</code>	Get the parent of the given <i>child Particle</i> object.

6.1.16 pyrex.CylindricalGenerator

class `pyrex.CylindricalGenerator` (*dr, dz, energy, shadow=False, flavor_ratio=(1, 1, 1), interaction_model=<class 'pyrex.particle.CTWInteraction'>*)

Class to generate neutrino vertices in a cylindrical ice volume.

Generates neutrinos in a cylinder with given radius and height.

Parameters

dr [float] Radius of the ice volume. Neutrinos generated within (0, *dr*).

dz [float] Height of the ice volume in the z-direction. Neutrinos generated within (-*dz*, 0).

energy [float or function] Energy (GeV) of the neutrinos. If `float`, all neutrinos have the same constant energy. If `function`, neutrinos are generated with the energy returned by successive function calls.

shadow [bool, optional] Whether Earth shadowing effects should be used to reject events. If `True` then neutrinos which don't survive transit through the Earth will be skipped when creating events. If `False` then all events are allowed and assigned a weight to scale their probability of occurrence.

flavor_ratio [array_like, optional] Flavor ratio of neutrinos to be generated. Of the form [electron, muon, tau] neutrino fractions.

interaction_model [optional] Class to use to describe interactions of the generated particles. Should inherit from (or behave like) the base `Interaction` class.

See also:

`pyrex.particle.Interaction` Base class for describing neutrino interaction attributes.

Attributes

count [int] Number of neutrinos produced by the generator, including those not returned due to Earth shadowing or other effects.

dr [float] Radius of the ice volume. Neutrinos generated within $(0, dr)$.

dz [float] Height of the ice volume in the z-direction. Neutrinos generated within $(-dz, 0)$.

get_energy [function] Function returning energy (GeV) of the neutrinos by successive function calls.

shadow [bool] Whether Earth shadowing effects will be used to reject events.

ratio [ndarray] (Normalized) flavor ratio of neutrinos to be generated. Of the form [electron, muon, tau] neutrino fractions.

interaction_model [Interaction] Class to use to describe interactions of the generated particles.

Methods

<code>create_event()</code>	Generate a neutrino event in the ice volume.
<code>get_direction()</code>	Get the direction of the next particle to be generated.
<code>get_exit_points(particle)</code>	Get the intersections of the particle path with the ice volume edges.
<code>get_particle_type()</code>	Get the particle type of the next particle to be generated.
<code>get_vertex()</code>	Get the vertex of the next particle to be generated.
<code>get_weights(particle)</code>	Get the weighting factors to be applied to the particle.

6.1.17 pyrex.RectangularGenerator

class `pyrex.RectangularGenerator` (`dx`, `dy`, `dz`, `energy`, `shadow=False`, `flavor_ratio=(1, 1, 1)`, `interaction_model=<class 'pyrex.particle.CTWInteraction'>`)

Class to generate neutrino vertices in a rectangular ice volume.

Generates neutrinos in a box with given width, length, and height.

Parameters

- dx** [float] Width of the ice volume in the x-direction. Neutrinos generated within $(-dx / 2, dx / 2)$.
- dy** [float] Length of the ice volume in the y-direction. Neutrinos generated within $(-dy / 2, dy / 2)$.
- dz** [float] Height of the ice volume in the z-direction. Neutrinos generated within $(-dz, 0)$.
- energy** [float or function] Energy (GeV) of the neutrinos. If `float`, all neutrinos have the same constant energy. If `function`, neutrinos are generated with the energy returned by successive function calls.
- shadow** [bool, optional] Whether Earth shadowing effects should be used to reject events. If `True` then neutrinos which don't survive transit through the Earth will be skipped when creating events. If `False` then all events are allowed and assigned a weight to scale their probability of occurrence.
- flavor_ratio** [array_like, optional] Flavor ratio of neutrinos to be generated. Of the form [electron, muon, tau] neutrino fractions.
- interaction_model** [optional] Class to use to describe interactions of the generated particles. Should inherit from (or behave like) the base `Interaction` class.

See also:

[`pyrex.particle.Interaction`](#) Base class for describing neutrino interaction attributes.

Attributes

- count** [int] Number of neutrinos produced by the generator, including those not returned due to Earth shadowing or other effects.
- dx** [float] Width of the ice volume in the x-direction. Neutrinos generated within $(-dx / 2, dx / 2)$.
- dy** [float] Length of the ice volume in the y-direction. Neutrinos generated within $(-dy / 2, dy / 2)$.
- dz** [float] Height of the ice volume in the z-direction. Neutrinos generated within $(-dz, 0)$.
- get_energy** [function] Function returning energy (GeV) of the neutrinos by successive function calls.
- shadow** [bool] Whether Earth shadowing effects will be used to reject events.
- ratio** [ndarray] (Normalized) flavor ratio of neutrinos to be generated. Of the form [electron, muon, tau] neutrino fractions.
- interaction_model** [Interaction] Class to use to describe interactions of the generated particles.

Methods

<code>create_event()</code>	Generate a neutrino event in the ice volume.
<code>get_direction()</code>	Get the direction of the next particle to be generated.
<code>get_exit_points(particle)</code>	Get the intersections of the particle path with the ice volume edges.
<code>get_particle_type()</code>	Get the particle type of the next particle to be generated.
<code>get_vertex()</code>	Get the vertex of the next particle to be generated.

Continued on next page

Table 13 – continued from previous page

<code>get_weights(particle)</code>	Get the weighting factors to be applied to the particle.
------------------------------------	--

6.1.18 pyrex.ListGenerator

class `pyrex.ListGenerator` (*events*, *loop=True*)

Class to generate neutrino events from a list.

Generates events by simply pulling them from a list of *Event* objects. By default returns to the start of the list once the end is reached, but can optionally fail after reaching the list's end.

Parameters

events [Event, or list of Event] List of *Event* objects to draw from. If only a single *Event* object is given, creates a list of that event alone.

loop [boolean, optional] Whether or not to return to the start of the list after throwing the last *Event*. If `False`, raises an error if trying to throw after the last *Event*.

See also:

pyrex.Event Class for storing a tree of *Particle* objects representing an event.

pyrex.Particle Class for storing particle attributes.

Attributes

count [int] Number of neutrinos produced by the generator.

events [list of Event] List to draw *Event* objects from, sequentially.

loop [boolean] Whether or not to loop through the list more than once.

Methods

<code>create_event()</code>	Generate a neutrino event.
-----------------------------	----------------------------

6.1.19 pyrex.FileGenerator

class `pyrex.FileGenerator` (*files*, *slice_range=100*, *interaction_model=<class 'pyrex.particle.CTWInteraction'>*)

Class to generate neutrino events from simulation file(s).

Generates neutrinos by pulling their attributes from a (list of) simulation output file(s). Designed to make reproducing simulations easier.

Parameters

files [str or list of str] List of file names containing neutrino event information. If only a single file name is provided, creates a list with that file alone.

slice_range [int, optional] Number of events to load into memory at a time from the files. Increasing this value should result in an improvement in speed, while decreasing this value should result in an improvement in memory consumption.

interaction_model [optional] Class used to describe the interactions of the stored particles.

Warning: This generator only supports *Event* objects containing a single level of *Particle* objects. Any dependencies among *Particle* objects will be ignored and they will all appear in the root level.

See also:

pyrex.particle.Interaction Base class for describing neutrino interaction attributes.

pyrex.Event Class for storing a tree of *Particle* objects representing an event.

pyrex.Particle Class for storing particle attributes.

Attributes

count [int] Number of neutrinos produced by the generator.

files [list of str] List of file names containing neutrino information.

Methods

<code>create_event()</code>	Generate a neutrino.
-----------------------------	----------------------

6.1.20 pyrex.RayTracer

pyrex.RayTracer
alias of *pyrex.ray_tracing.SpecializedRayTracer*

6.1.21 pyrex.RayTracePath

pyrex.RayTracePath
alias of *pyrex.ray_tracing.SpecializedRayTracePath*

6.1.22 pyrex.EventKernel

class pyrex.EventKernel (*generator*, *antennas*, *ice_model*=<pyrex.ice_model.AntarcticIce object>, *ray_tracer*=<class 'pyrex.ray_tracing.SpecializedRayTracer'>, *signal_model*=<class 'pyrex.askaryan.ARVZAskaryanSignal'>, *signal_times*=array([-2.000e-08, -1.995e-08, -1.990e-08, ..., 7.985e-08, 7.990e-08, 7.995e-08]), *event_writer*=None, *triggers*=None)

High-level kernel for controlling event simulation.

The kernel is responsible for handling the classes and objects which control the major simulation steps: particle creation, signal production, signal propagation, and antenna response. The modular kernel structure allows for easy switching of the classes or objects which handle any of the simulation steps.

Parameters

generator A particle generator to create neutrino events.

antennas An iterable object consisting of antenna objects which can receive and store signals.

ice_model [optional] An ice model describing the ice surrounding the *antennas*.

ray_tracer [optional] A ray tracer capable of propagating signals from the neutrino vertex to the antenna positions.

- signal_model** [optional] A signal class which generates signals based on the particle.
- signal_times** [array_like, optional] The array of times over which the neutrino signal should be generated.
- event_writer** [File, optional] A file object to be used for writing data output.
- triggers** [function or dict, optional] A function or dictionary with function values representing trigger conditions of the detector. If a dictionary, must have a “global” key with its value representing the global detector trigger.

See also:

- pyrex.Event** Class for storing a tree of *Particle* objects representing an event.
- pyrex.Particle** Class for storing particle attributes.
- pyrex.ice_model.AntarcticIce** Class describing the ice at the south pole.
- pyrex.RayTracer** Class for calculating the ray-trace solutions between points.
- pyrex.AskaryanSignal** Class for generating Askaryan signals according to ARVZ parameterization.
- pyrex.File** Class for reading or writing data files.

Notes

The kernel is designed to be modular so individual parts of the simulation chain can be exchanged. In order to interchange the pieces, their classes require the following at a minimum:

The particle generator *generator* must have a `create_event` method which takes no arguments and returns a *Event* object consisting of *Particle* objects with `vertex`, `direction`, `energy`, and `weight` attributes.

The antenna iterable *antennas* must yield each antenna object once when iterating directly over *antennas*. Each antenna object must have a `position` attribute and a `receive` method which takes a signal object as its first argument, and `ndarray` objects as `direction` and `polarization` keyword arguments.

The *ice_model* must have an `index` method returning the index of refraction given a (negative-valued) depth, and it must support anything required of it by the *ray_tracer*.

The *ray_tracer* must be initialized with the particle vertex and an antenna position as its first two arguments, and the *ice_model* of the kernel as the `ice_model` keyword argument. The ray tracer must also have `exists` and `solutions` attributes, the first of which denotes whether any paths exist between the given points and the second of which is an iterable revealing each path between the points. These paths must have `emitted_direction`, `received_direction`, and `path_length` attributes, as well as a `propagate` method which takes a signal object and applies the propagation effects of the path in-place to that object.

The *signal_model* must be initialized with the *signal_times* array, a *Particle* object from the *Event*, the `viewing_angle` and `viewing_distance` according to the *ray_tracer*, and the *ice_model*. The object created should be a *Signal* object with `times` and `values` attributes representing the time-domain Askaryan signal produced by the *Particle*.

Attributes

- gen** The particle generator responsible for particle creation.
- antennas** The iterable of antennas responsible for handling applying their response and storing the resulting signals.
- ice** The ice model describing the ice containing the *antennas*.
- ray_tracer** The ray tracer responsible for signal propagation through the *ice*.

signal_model The signal class to use to generate signals based on the particle.

signal_times The array of times over which the neutrino signal should be generated.

writer The file object to be used for writing data output.

triggers The trigger condition(s) of the detector.

Methods

<code>event()</code>	Create a neutrino event and run it through the simulation chain.
----------------------	--

6.1.23 pyrex.File

class `pyrex.File`

Class for reading or writing data files.

Works as a context manager and allows for reading or writing simulation/real data or analysis-level data to the given file. Chooses the appropriate class for handling the given file type.

Parameters

filename [str] File name to open in the given write mode.

mode [str, optional] Mode with which to open the file.

****kwargs** Keyword arguments passed on to the appropriate file handler.

See also:

`pyrex.io.HDF5Reader` Class for reading data from an hdf5 file.

`pyrex.io.HDF5Writer` Class for writing data to an hdf5 file.

Attributes

readers [dict] Dictionary with file extensions as keys and values with the corresponding classes used to handle reading of those file types.

writers [dict] Dictionary with file extensions as keys and values with the corresponding classes used to handle writing of those file types.

6.2 Individual Module APIs

6.2.1 Helper Functions (`pyrex.internal_functions`)

Helper functions and classes for use in PyREx modules.

This module is intended as a container for functions, typically used in more than one PyREx module, which are not physics-motivated and are instead used mainly to clean up code. Functions and classes in this module may also be computer-science-motivated structures that python doesn't include naturally.

<code>normalize</code>	Normalize the given vector.
<code>get_from_enum</code>	Find the enum value given some representation of it.

Continued on next page

Table 17 – continued from previous page

<code>flatten</code>	Flattens an iterator to iterate over all elements individually.
<code>complex_interp</code>	Perform interpolation on complex values.
<code>complex_bilinear_interp</code>	Perform bilinear interpolation on complex values.
<code>mirror_func</code>	Mirror the attributes of one function onto another.
<code>lazy_property</code>	Decorator that makes a property lazily evaluated.
<code>LazyMutableClass</code>	Class with lazy properties which may depend on other class attributes.

pyrex.internal_functions.normalize

`pyrex.internal_functions.normalize(vector)`

Normalize the given vector.

Parameters

vector [array_like]

Returns

ndarray Normalized form of *vector*.

Examples

```
>>> normalize([5,0,0])
array([1., 0., 0.]
```

```
>>> v = np.array([1,0,1])
>>> normalize(v)
array([0.70710678, 0.          , 0.70710678])
```

pyrex.internal_functions.get_from_enum

`pyrex.internal_functions.get_from_enum(value, enum)`

Find the enum value given some representation of it.

Transforms the given *value* into the corresponding value from the *enum* by checking the type of *value* given.

Parameters

value Representation of the desired *enum* value. If already a member of *enum*, no change. If *str*, assumed to be a name in the *enum*. Otherwise, assumed to be a value type of the *enum*.

enum [Enum] Python Enum to compare names values with.

Returns

Enum value Value in the *enum* represented by the given *value*.

Examples


```
>>> from enum import Enum
>>> class Color(Enum):
...     red = 1
...     green = 2
...     blue = 3
>>> get_from_enum(Color.red, Color)
<Color.red: 1>
>>> get_from_enum("green", Color)
<Color.green: 2>
>>> get_from_enum(3, Color)
<Color.blue: 3>
```

pyrex.internal_functions.flatten

`pyrex.internal_functions.flatten(iterator, dont_flatten=())`

Flattens an iterator to iterate over all elements individually.

Flattens all iterable elements in the given iterator recursively and yields the resulting flat iterator. Can optionally not flatten certain classes. Will not flatten strings or bytes to avoid recursion errors.

Parameters

iterator [iterable object] Iterable object to flatten.

dont_flatten [tuple_like, optional] Tuple (or similar) of classes which should not be flattened.

Yields

element [any] Each element of *iterator* with sub-iterators expanded out.

Notes

Since `str` and `bytes` objects are always considered iterable despite their length, these objects will not be flattened and will remain intact.

If a class is asked not to be flattened, any sub-iterators contained in an iterator of that class will not be flattened either (see examples).

Examples

```
>>> list(flatten([1, 2, (3, 'four', [5, 6], 7), [8, 9]]))
[1, 2, 3, 'four', 5, 6, 7, 8, 9]
```

```
>>> list(flatten([1, 2, (3, 'four', [5, 6], 7), [8, 9]], dont_flatten=(tuple,)))
[1, 2, (3, 'four', [5, 6], 7), 8, 9]
```

```
>>> list(flatten([1, 2, [3, 'four', (5, 6), 7], [8, 9]], dont_flatten=(tuple,)))
[1, 2, 3, 'four', (5, 6), 7, 8, 9]
```

pyrex.internal_functions.complex_interp

`pyrex.internal_functions.complex_interp(x, xp, fp, method='cartesian', outer=None)`

Perform interpolation on complex values.

Calculates the linear interpolation of an array of complex values *fp*. Interpolates in real and imaginary parts for the Cartesian method, and interpolates in gain and (unwrapped) phase for the Euler method.

Parameters

- x** [array_like] The x-coordinates at which to evaluate the interpolated values.
- xp** [array_like] The x-coordinates of the data points, must be increasing.
- fp** [array_like] The (complex-valued) y-coordinates of the data points, same length as *xp*.
- method** [{ 'cartesian', 'euler' }, optional] The interpolation method to use between data points. 'Cartesian' will interpolate linearly in the real/imaginary plane. 'Euler' will interpolate linearly in the gain and (unwrapped) phase values.
- outer** [None or float, optional] The value to use for values of *x* outside of the range of *xp*. In the 'Cartesian' method it will be applied as a complex value. In the 'Euler' method it will be applied to the gain, but phases will always retain their value at the closest edge of *xp*.

Returns

- y** [ndarray] The (complex-valued) interpolated values, same shape as *x*.

Raises

- ValueError** If an unspecified interpolation method is given. Or if *xp* and *fp* have different length. Or if *xp* and *fp* are not 1-D sequences.

pyrex.internal_functions.complex_bilinear_interp

`pyrex.internal_functions.complex_bilinear_interp(x, y, xp, yp, fp, method='cartesian', unwrap_axis=0)`

Perform bilinear interpolation on complex values.

Calculates the bilinear interpolation of an array of complex values *fp*. Interpolates in real and imaginary parts for the Cartesian method, and interpolates in gain and (unwrapped) phase for the Euler method.

Parameters

- x** [float] The x-coordinate at which to evaluate the interpolated values.
- y** [float] The y-coordinate at which to evaluate the interpolated values.
- xp** [array_like] The x-coordinates of the data points, must be increasing.
- yp** [array_like] The y-coordinates of the data points, must be increasing.
- fp** [array_like] The (complex-valued) z-coordinates of the data points, with its second dimension the same length as *xp* and its third dimension the same length as *yp*.
- method** [{ 'cartesian', 'euler' }, optional] The interpolation method to use between data points. 'Cartesian' will interpolate linearly in the real/imaginary plane. 'Euler' will interpolate linearly in the gain and (unwrapped) phase values.
- unwrap_axis** [int, optional] When used with the 'euler' method, controls which axis of *fp* the phases are unwrapped around.

Returns

- z** [ndarray] The (complex-valued) interpolated values.

Raises

- ValueError** If an unspecified interpolation method is given. Or if *x* or *y* lies outside of the range of *xp* or *yp*. Or if the shape of *fp* is incompatible with *xp* and *yp*.

pyrex.internal_functions.mirror_func

`pyrex.internal_functions.mirror_func(match_func, run_func, self=None)`

Mirror the attributes of one function onto another.

Creates a function which operates like one function, but has all the attributes of another. Works for functions or class methods.

Parameters

match_func [function] Function with the attributes to be mirrored.

run_func [function] Function with the desired behavior.

self [object or None, optional] If None, `run_func` called as a regular function, otherwise `run_func` is called as a class method (with `self` as its first argument).

Returns

function Function with the behavior of `run_func`, but the attributes of `match_func`.

Examples

```
>>> from inspect import signature
>>> def descriptive_add(a, b):
...     """Function with a descriptive docstring."""
...     pass
>>> def add_implementation(x, y):
...     # Actually adds, but no docs or anything
...     return x+y
>>> my_add = mirror_func(descriptive_add, add_implementation)
>>> my_add(2, 3)
5
>>> my_add.__doc__
'Function with a descriptive docstring.'
>>> signature(my_add)
<Signature (a, b)>
```

```
>>> from inspect import signature
>>> class A:
...     def __init__(self, value):
...         self.value = value
...     def mult(self, factor, power=1):
...         """Multiplies value by factor and raises to power."""
...         return (self.value*factor)**power
>>> class B(A):
...     def __init__(self, value):
...         self.value = value
...         # Make the mult method look the same as for A, but with
...         # different behavior
...         self.mult = mirror_func(A.mult, B.different_mult, self=self)
...     def different_mult(self, *args, **kwargs):
...         """Different implementation of mult."""
...         return (self.value*int(args[0]))**kwargs['power']
>>> b = B(5)
>>> b.mult(2.5, power=2)
100
>>> b.mult.__doc__
```

(continues on next page)

(continued from previous page)

```
'Multiplies by factor and raises to power.'
>>> signature(b.mult)
<Signature (self, factor, power=1)>
```

pyrex.internal_functions.lazy_property

`pyrex.internal_functions.lazy_property` (*fn*)

Decorator that makes a property lazily evaluated.

Acts like the standard python `property` decorator, but the first time the decorated property is accessed an attribute with the property's name prefixed by `'_lazy_'` will be created and the value of the property will be stored. Upon further access of the property, the stored value will be returned instead of recalculating it.

Parameters

fn [function] Function returning class property which is to be decorated.

Returns

function Lazy-evaluation property function.

See also:

LazyMutableClass Class for lazy properties dependent on attributes.

Notes

Using the `lazy_property` decorator instead of the simple python `property` decorator increases the time for property access (after the initial calculation) from ~0.5 microseconds to ~5 microseconds, so `lazy_property` is only recommended for use on properties with calculation times >5 microseconds which are likely to be accessed more than once.

Examples

```
>>> from time import sleep
>>> class A:
...     def __init__(self, value):
...         self.value = value
...     @lazy_property
...     def twice(self):
...         sleep(5)
...         return self.value*2
>>> a = A(1)
>>> "_lazy_twice" in a.__dict__
False
>>> a.twice
2
>>> "_lazy_twice" in a.__dict__
True
>>> a.twice
2
```

pyrex.internal_functions.LazyMutableClass

class `pyrex.internal_functions.LazyMutableClass` (*static_attributes=None*)

Class with lazy properties which may depend on other class attributes.

This class is intended as a base class for any class which desires lazy properties which depend on other attributes and thus may need to be recalculated when the class attributes change. Any lazy properties in this class will be lazily evaluated as usual until one of the given static attributes changes, at which point all lazy properties will be cleared and will be recalculated on their next call. By default the static attributes of the class will be set to all attributes present at the time of the `LazyMutableClass.__init__` call.

Parameters

static_attributes [None or sequence of str, optional] Set of attribute names on which the lazy properties depend. If `None` then it will contain all members of `__dict__` at the time of the call.

See also:

[*lazy_property*](#) Decorator for lazily-evaluated properties.

Examples

```

>>> from time import sleep
>>> class A(LazyMutableClass):
...     def __init__(self, value):
...         self.value = value
...         super().__init__()
...     @lazy_property
...     def twice(self):
...         sleep(5)
...         return self.value*2
>>> a = A(1)
>>> "_lazy_twice" in a.__dict__
False
>>> a.twice
2
>>> "_lazy_twice" in a.__dict__
True
>>> a.twice
2
>>> a.value = 5
>>> "_lazy_twice" in a.__dict__
False
>>> a.twice
10
>>> "_lazy_twice" in a.__dict__
True
>>> a.twice
10

```

6.2.2 Signal Processing (`pyrex.signals`)

Module containing classes for digital signal processing.

All classes in this module hold time-domain information about some signals, and have methods for manipulating this data as it relates to digital signal processing and general physics.

<i>Signal</i>	Base class for time-domain signals.
<i>EmptySignal</i>	Class for signal with zero amplitude (all values = 0).
<i>FunctionSignal</i>	Class for signals generated by a function.
<i>GaussianNoise</i>	Class for gaussian noise signals with standard deviation sigma.
<i>ThermalNoise</i>	Class for thermal Rayleigh noise signals.

pyrex.signals.Signal

class pyrex.signals.**Signal** (*times*, *values*, *value_type=None*)

Base class for time-domain signals.

Stores the time-domain information for signal values. Supports adding between signals with the same times array and value type.

Parameters

times [array_like] 1D array of times (s) for which the signal is defined.

values [array_like] 1D array of values of the signal corresponding to the given *times*. Will be resized to the size of *times* by zero-padding or truncating as necessary.

value_type [optional] Type of signal, representing the units of the values. Values should be from the `Signal.Type` enum, but integer or string values may work if carefully chosen. `Signal.Type.undefined` by default.

Attributes

times, values [ndarray] 1D arrays of times (s) and corresponding values which define the signal.

value_type [`Signal.Type`] Type of signal, representing the units of the values.

Type [Enum] Enum containing possible types (units) for signal values.

dt The time spacing of the *times* array, or `None` if invalid.

frequencies The FFT frequencies of the signal.

spectrum The FFT complex spectrum values of the signal.

envelope The envelope of the signal by Hilbert transform.

Methods

Type	Enum containing possible types (units) for signal values.
filter_frequencies(freq_response[, force_real])	Apply the given frequency response function to the signal, in-place.
resample(n)	Resamples the signal into n points in the same time range, in-place.
with_times(new_times)	Returns a representation of this signal over a different times array.

pyrex.signals.EmptySignal

class pyrex.signals.**EmptySignal** (*times*, *value_type=None*)

Class for signal with zero amplitude (all values = 0).

Parameters

times [array_like] 1D array of times (s) for which the signal is defined.

value_type [optional] Type of signal, representing the units of the values. Must be from the `Signal.Type` Enum.

See also:

Signal Base class for time-domain signals.

Attributes

times, values [ndarray] 1D arrays of times (s) and corresponding values which define the signal.

value_type [Signal.Type] Type of signal, representing the units of the values.

Type [Enum] Enum containing possible types (units) for signal values.

dt The time spacing of the *times* array, or `None` if invalid.

frequencies The FFT frequencies of the signal.

spectrum The FFT complex spectrum values of the signal.

envelope The envelope of the signal by Hilbert transform.

Methods

Type	Enum containing possible types (units) for signal values.
filter_frequencies(freq_response[, force_real])	Apply the given frequency response function to the signal, in-place.
resample(n)	Resamples the signal into n points in the same time range, in-place.
with_times(new_times)	Returns a representation of this signal over a different times array.

pyrex.signals.FunctionSignal

class pyrex.signals.**FunctionSignal** (*times*, *function*, *value_type=None*)

Class for signals generated by a function.

Parameters

times [array_like] 1D array of times (s) for which the signal is defined.

function [function] Function which evaluates the corresponding value(s) for a given time or array of times.

value_type [optional] Type of signal, representing the units of the values. Must be from the `Signal.Type` Enum.

See also:

Signal Base class for time-domain signals.

EmptySignal Class for signal with zero amplitude.

Attributes

times, values [ndarray] 1D arrays of times (s) and corresponding values which define the signal.

value_type [Signal.Type] Type of signal, representing the units of the values.

Type [Enum] Enum containing possible types (units) for signal values.

function [function] Function to evaluate the signal values at given time(s).

dt The time spacing of the *times* array, or *None* if invalid.

frequencies The FFT frequencies of the signal.

spectrum The FFT complex spectrum values of the signal.

envelope The envelope of the signal by Hilbert transform.

Methods

Type	Enum containing possible types (units) for signal values.
filter_frequencies(freq_response[, force_real])	Apply the given frequency response function to the signal, in-place.
resample(n)	Resamples the signal into n points in the same time range, in-place.
with_times(new_times)	Returns a representation of this signal over a different times array.

pyrex.signals.GaussianNoise

class pyrex.signals.**GaussianNoise** (*times, sigma*)

Class for gaussian noise signals with standard deviation sigma.

Calculates each time value independently from a normal distribution.

Parameters

times [array_like] 1D array of times (s) for which the signal is defined.

values [array_like] 1D array of values of the signal corresponding to the given *times*. Will be resized to the size of *times* by zero-padding or truncating.

value_type Type of signal, representing the units of the values. Must be from the *Signal.Type* Enum.

See also:

Signal Base class for time-domain signals.

Attributes

times, values [ndarray] 1D arrays of times (s) and corresponding values which define the signal.

value_type [Signal.Type.voltage] Type of signal, representing the units of the values.

Type [Enum] Enum containing possible types (units) for signal values.

dt The time spacing of the *times* array, or `None` if invalid.

frequencies The FFT frequencies of the signal.

spectrum The FFT complex spectrum values of the signal.

envelope The envelope of the signal by Hilbert transform.

Methods

Type	Enum containing possible types (units) for signal values.
<code>filter_frequencies(freq_response[, force_real])</code>	Apply the given frequency response function to the signal, in-place.
<code>resample(n)</code>	Resamples the signal into <i>n</i> points in the same time range, in-place.
<code>with_times(new_times)</code>	Returns a representation of this signal over a different times array.

pyrex.signals.ThermalNoise

class `pyrex.signals.ThermalNoise` (*times*, *f_band*, *f_amplitude*=*None*, *rms_voltage*=*None*, *temperature*=*None*, *resistance*=*None*, *n_freqs*=0)

Class for thermal Rayleigh noise signals.

The Rayleigh thermal noise is calculated in a given frequency band with flat or otherwise specified amplitude and random phase at some number of frequencies. Values are scaled to a provided or calculated RMS voltage.

Parameters

times [array_like] 1D array of times (s) for which the signal is defined.

f_band [array_like] Array of two elements denoting the frequency band (Hz) of the noise. The first element should be smaller than the second.

f_amplitude [float or function, optional] The frequency-domain amplitude of the noise. If *float*, then all frequencies will have the same amplitude. If *function*, then the function is evaluated at each frequency to determine its amplitude. By default, uses Rayleigh-distributed amplitudes.

rms_voltage [float, optional] The RMS voltage (V) of the noise. If specified, this value will be used instead of the RMS voltage calculated from the values of *temperature* and *resistance*.

temperature [float, optional] The thermal noise temperature (K). Used in combination with the value of *resistance* to calculate the RMS voltage of the noise.

resistance [float, optional] The resistance (ohm) for the noise. Used in combination with the value of *temperature* to calculate the RMS voltage of the noise.

n_freqs [int, optional] The number of frequencies within the frequency band to use to calculate the noise signal. By default determines the number of frequencies based on the FFT bin size of *times*.

Raises

ValueError If the RMS voltage cannot be calculated (i.e. *rms_voltage* or both *temperature* and *resistance* are *None*).

Warning: Since this class inherits from `FunctionSignal`, its `with_times` method will properly extrapolate noise outside of the provided times. Be warned however that outside of the original signal times the noise signal will be highly periodic. Since the default number of frequencies used is based on the FFT bin size of `times`, the period of the noise signal is actually the length of `times`. As a result if you are planning on extrapolating the noise signal, increasing the number of frequencies used is strongly recommended.

See also:

FunctionSignal Class for signals generated by a function.

Notes

Calculation of the noise signal is based on the Rayleigh noise model used by ANITA [1]. Modifications have been made to the default to make the frequency-domain amplitudes Rayleigh-distributed, under the suggestion that this makes for more realistic noise traces.

References

[1]

Attributes

- times, values** [ndarray] 1D arrays of times (s) and corresponding values which define the signal.
- value_type** [Signal.Type.voltage] Type of signal, representing the units of the values.
- Type** [Enum] Enum containing possible types (units) for signal values.
- function** [function] Function to evaluate the signal values at given time(s).
- f_min** [float] Minimum frequency of the noise frequency band.
- f_max** [float] Maximum frequency of the noise frequency band.
- freqs, amps, phases** [ndarray] The frequencies used to define the noise signal and their corresponding amplitudes and phases.
- rms** [float] The RMS value of the noise signal.
- dt** The time spacing of the `times` array, or `None` if invalid.
- frequencies** The FFT frequencies of the signal.
- spectrum** The FFT complex spectrum values of the signal.
- envelope** The envelope of the signal by Hilbert transform.

Methods

Type	Enum containing possible types (units) for signal values.
<code>filter_frequencies(freq_response[, force_real])</code>	Apply the given frequency response function to the signal, in-place.
<code>resample(n)</code>	Resamples the signal into n points in the same time range, in-place.

Continued on next page

Table 23 – continued from previous page

<code>with_times(new_times)</code>	Returns a representation of this signal over a different times array.
------------------------------------	---

6.2.3 Askaryan Models (`pyrex.askaryan`)

Module containing Askaryan model signals classes.

Contains various Askaryan models based in both the frequency and time domains. All classes inherit from the `Signal` class, meaning they must all be converted into the time domain at the end of initialization.

<code>ZHSAskaryanSignal</code>	Class for generating Askaryan signals according to ZHS parameterization.
<code>AVZAskaryanSignal</code>	Class for generating Askaryan signals according to ARVZ parameterization.
<code>ARVZAskaryanSignal</code>	Class for generating Askaryan signals according to ARVZ parameterization.
<code>AskaryanSignal</code>	alias of <code>pyrex.askaryan.ARVZAskaryanSignal</code>

`pyrex.askaryan.ZHSAskaryanSignal`

class `pyrex.askaryan.ZHSAskaryanSignal` (*times*, *particle*, *viewing_angle*, *viewing_distance*=1, *ice_model*=<`pyrex.ice_model.AntarcticIce` object>, *t0*=0)

Class for generating Askaryan signals according to ZHS parameterization.

Stores the time-domain information for an Askaryan electric field (V/m) produced by the electromagnetic shower initiated by a neutrino.

Parameters

times [array_like] 1D array of times (s) for which the signal is defined.

particle [Particle] `Particle` object responsible for the shower which produces the Askaryan signal. Should have an energy in GeV, vertex in m, and id, plus an interaction with an `em_frac` and `had_frac`.

viewing_angle [float] Observation angle (radians) measured relative to the shower axis.

viewing_distance [float, optional] Distance (m) between the shower vertex and the observation point (along the ray path).

ice_model [optional] The ice model to be used for describing the index of refraction of the medium.

t0 [float, optional] Pulse offset time (s), i.e. time at which the shower takes place.

Raises

ValueError If the *particle* object is not a neutrino or antineutrino with a charged-current or neutral-current interaction.

See also:

Signal Base class for time-domain signals.

`pyrex.Particle` Class for storing particle attributes.

Notes

Calculates the Askaryan signal based on the ZHS parameterization [1]. Uses equations 20 and 21 to calculate the electric field close to the Cherenkov angle.

References

[1]

Attributes

- times, values** [ndarray] 1D arrays of times (s) and corresponding values which define the signal.
- value_type** [Signal.Type.field] Type of signal, representing the units of the values.
- Type** [Enum] Enum containing possible types (units) for signal values.
- energy** [float] Energy (GeV) of the electromagnetic shower producing the pulse.
- vector_potential**
- dt** The time spacing of the *times* array, or None if invalid.
- frequencies** The FFT frequencies of the signal.
- spectrum** The FFT complex spectrum values of the signal.
- envelope** The envelope of the signal by Hilbert transform.

Methods

Type	Enum containing possible types (units) for signal values.
filter_frequencies(freq_response[, force_real])	Apply the given frequency response function to the signal, in-place.
resample(n)	Resamples the signal into n points in the same time range, in-place.
with_times(new_times)	Returns a representation of this signal over a different times array.

pyrex.askaryan.AVZAskaryanSignal

class pyrex.askaryan.AVZAskaryanSignal (*times, particle, viewing_angle, viewing_distance=1, ice_model=<pyrex.ice_model.AntarcticIce object>, t0=0*)

Class for generating Askaryan signals according to ARVZ parameterization.

Stores the time-domain information for an Askaryan electric field (V/m) produced by the electromagnetic and hadronic showers initiated by a neutrino.

Parameters

- times** [array_like] 1D array of times (s) for which the signal is defined.
- particle** [Particle] Particle object responsible for the showers which produce the Askaryan signal. Should have an energy in GeV, vertex in m, and id, plus an interaction with an `em_frac` and `had_frac`.
- viewing_angle** [float] Observation angle (radians) measured relative to the shower axis.

viewing_distance [float, optional] Distance (m) between the shower vertex and the observation point (along the ray path).

ice_model [optional] The ice model to be used for describing the index of refraction of the medium.

t0 [float, optional] Pulse offset time (s), i.e. time at which the showers take place.

Raises

ValueError If the *particle* object is not a neutrino or antineutrino with a charged-current or neutral-current interaction.

See also:

Signal Base class for time-domain signals.

`pyrex.Particle` Class for storing particle attributes.

Notes

Calculates the Askaryan signal based on the AVZ parameterization [1]. Matches the NuRadioMC implementation named ‘Alvarez2000’, including the LPM effect correction added based on an earlier paper by Alvarez-Muniz and Zas [2].

References

[1], [2]

Attributes

times, values [ndarray] 1D arrays of times (s) and corresponding values which define the signal.

value_type [Signal.Type.field] Type of signal, representing the units of the values.

Type [Enum] Enum containing possible types (units) for signal values.

em_energy [float] Energy (GeV) of the electromagnetic shower producing the pulse.

had_energy [float] Energy (GeV) of the hadronic shower producing the pulse.

dt The time spacing of the *times* array, or *None* if invalid.

frequencies The FFT frequencies of the signal.

spectrum The FFT complex spectrum values of the signal.

envelope The envelope of the signal by Hilbert transform.

Methods

Type	Enum containing possible types (units) for signal values.
filter_frequencies(freq_response[, force_real])	Apply the given frequency response function to the signal, in-place.
resample(n)	Resamples the signal into n points in the same time range, in-place.

Continued on next page

Table 26 – continued from previous page

<code>with_times(new_times)</code>	Returns a representation of this signal over a different times array.
------------------------------------	---

pyrex.askaryan.ARVZAskaryanSignal

class `pyrex.askaryan.ARVZAskaryanSignal` (*times*, *particle*, *viewing_angle*, *viewing_distance=1*, *ice_model=<pyrex.ice_model.AntarcticIce object>*, *t0=0*)

Class for generating Askaryan signals according to ARVZ parameterization.

Stores the time-domain information for an Askaryan electric field (V/m) produced by the electromagnetic and hadronic showers initiated by a neutrino.

Parameters

times [array_like] 1D array of times (s) for which the signal is defined.

particle [Particle] `Particle` object responsible for the showers which produce the Askaryan signal. Should have an `energy` in GeV, `vertex` in m, and `id`, plus an interaction with an `em_frac` and `had_frac`.

viewing_angle [float] Observation angle (radians) measured relative to the shower axis.

viewing_distance [float, optional] Distance (m) between the shower vertex and the observation point (along the ray path).

ice_model [optional] The ice model to be used for describing the index of refraction of the medium.

t0 [float, optional] Pulse offset time (s), i.e. time at which the showers take place.

Raises

ValueError If the *particle* object is not a neutrino or antineutrino with a charged-current or neutral-current interaction.

See also:

Signal Base class for time-domain signals.

pyrex.Particle Class for storing particle attributes.

Notes

Calculates the Askaryan signal based on the ARVZ parameterization [1]. Uses a Greisen model for the electromagnetic shower profile [2], [3] and a Gaisser-Hillas model for the hadronic shower profile [4], [5]. Calculates the electric field from the vector potential using the convolution method outlined in section 4 of the ARVZ paper, which results in the most efficient calculation of the parameterization.

References

[1], [2], [3], [4], [5]

Attributes

times, values [ndarray] 1D arrays of times (s) and corresponding values which define the signal.

value_type [Signal.Type.field] Type of signal, representing the units of the values.

Type [Enum] Enum containing possible types (units) for signal values.

em_energy [float] Energy (GeV) of the electromagnetic shower producing the pulse.

had_energy [float] Energy (GeV) of the hadronic shower producing the pulse.

vector_potential The vector potential of the signal.

dt The time spacing of the *times* array, or *None* if invalid.

frequencies The FFT frequencies of the signal.

spectrum The FFT complex spectrum values of the signal.

envelope The envelope of the signal by Hilbert transform.

Methods

<code>RAC(time, energy)</code>	Calculates $R*A_C$ at the given time and energy.
<code>Type</code>	Enum containing possible types (units) for signal values.
<code>em_shower_profile(z, energy[, density, ...])</code>	Calculates the electromagnetic shower longitudinal charge profile.
<code>filter_frequencies(freq_response[, force_real])</code>	Apply the given frequency response function to the signal, in-place.
<code>had_shower_profile(z, energy[, density, ...])</code>	Calculates the hadronic shower longitudinal charge profile.
<code>max_length(energy[, density, crit_energy, ...])</code>	Calculates the depth of a particle shower maximum.
<code>resample(n)</code>	Resamples the signal into <i>n</i> points in the same time range, in-place.
<code>shower_signal(times, energy, ...)</code>	Calculate the signal values for some shower type.
<code>with_times(new_times)</code>	Returns a representation of this signal over a different times array.

pyrex.askaryan.AskaryanSignal

`pyrex.askaryan.AskaryanSignal`
 alias of `pyrex.askaryan.ARVZAskaryanSignal`

6.2.4 Antennas (pyrex. antenna)

Module containing antenna classes responsible of receiving signals.

These classes are intended to model the properties of antennas including how signals are received as well as the production of noise. A number of attributes like directional gain, frequency response, and antenna factor may be necessary to calculate how signals are manipulated upon reception by an antenna.

<i>Antenna</i>	Base class for antennas.
<i>DipoleAntenna</i>	Class for half-wave dipole antennas.

pyrex.antenna.Antenna

```
class pyrex.antenna.Antenna (position, z_axis=(0, 0, 1), x_axis=(1, 0, 0), antenna_factor=1,
                             efficiency=1,      noisy=True,      unique_noise_waveforms=10,
                             freq_range=None,    temperature=None,    resistance=None,
                             noise_rms=None)
```

Base class for antennas.

Stores the attributes of an antenna as well as handling receiving, processing, and storing signals and adding noise.

Parameters

- position** [array_like] Vector position of the antenna.
- z_axis** [array_like, optional] Vector direction of the z-axis of the antenna.
- x_axis** [array_like, optional] Vector direction of the x-axis of the antenna.
- antenna_factor** [float, optional] Antenna factor used for converting electric field values to voltages.
- efficiency** [float, optional] Antenna efficiency applied to incoming signal values.
- noisy** [boolean, optional] Whether or not the antenna should add noise to incoming signals.
- unique_noise_waveforms** [int, optional] The number of expected noise waveforms needed for each received signal to have its own noise.
- freq_range** [array_like, optional] The frequency band in which the antenna operates (used for noise production).
- temperature** [float, optional] The noise temperature (K) of the antenna. Used in combination with *resistance* to calculate the RMS voltage of the antenna noise.
- resistance** [float, optional] The noise resistance (ohm) of the antenna. Used in combination with *temperature* to calculate the RMS voltage of the antenna noise.
- noise_rms** [float, optional] The RMS voltage (V) of the antenna noise. If specified, this value will be used instead of the RMS voltage calculated from the values of *temperature* and *resistance*.

Attributes

- position** [array_like] Vector position of the antenna.
- z_axis** [ndarray] Vector direction of the z-axis of the antenna.
- x_axis** [ndarray] Vector direction of the x-axis of the antenna.
- antenna_factor** [float] Antenna factor used for converting electric field values to voltages.
- efficiency** [float] Antenna efficiency applied to incoming signal values.
- noisy** [boolean] Whether or not the antenna should add noise to incoming signals.
- unique_noises** [int] The number of expected noise waveforms needed for each received signal to have its own noise.
- freq_range** [array_like] The frequency band in which the antenna operates (used for noise production).
- temperature** [float or None] The noise temperature (K) of the antenna. Used in combination with *resistance* to calculate the RMS voltage of the antenna noise.

resistance [float or None] The noise resistance (ohm) of the antenna. Used in combination with *temperature* to calculate the RMS voltage of the antenna noise.

noise_rms [float or None] The RMS voltage (v) of the antenna noise. If not `None`, this value will be used instead of the RMS voltage calculated from the values of *temperature* and *resistance*.

signals [list of Signal] The signals which have been received by the antenna.

is_hit Boolean of whether the antenna has been triggered.

is_hit_mc_truth Boolean of whether the antenna has been triggered by signal.

waveforms Signal + noise (if `noisy`) for each triggered antenna hit.

all_waveforms Signal + noise (if `noisy`) for all antenna hits.

Methods

<code>apply_response(signal[, direction, ...])</code>	Process the complete antenna response for an incoming signal.
<code>clear([reset_noise])</code>	Reset the antenna to an empty state.
<code>directional_gain(theta, phi)</code>	Calculate the (complex) directional gain of the antenna.
<code>frequency_response(frequencies)</code>	Calculate the (complex) frequency response of the antenna.
<code>full_waveform(times)</code>	Signal + noise (if <code>noisy</code>) for the given times.
<code>is_hit_during(times)</code>	Check if the antenna is triggered in a time range.
<code>make_noise(times)</code>	Creates a noise signal over the given times.
<code>polarization_gain(polarization)</code>	Calculate the (complex) polarization gain of the antenna.
<code>receive(signal[, direction, polarization, ...])</code>	Process and store one or more incoming (polarized) signals.
<code>set_orientation([z_axis, x_axis])</code>	Sets the orientation of the antenna.
<code>trigger(signal)</code>	Check if the antenna triggers on a given signal.

pyrex.antenna.DipoleAntenna

```
class pyrex.antenna.DipoleAntenna (name, position, center_frequency, band-
                                width, resistance, orientation=(0, 0, 1), trig-
                                ger_threshold=0, effective_height=None, noisy=True,
                                unique_noise_waveforms=10)
```

Class for half-wave dipole antennas.

Stores the attributes of an antenna as well as handling receiving, processing, and storing signals and adding noise. Uses a first-order butterworth filter for the frequency response. Includes a simple threshold trigger.

Parameters

name [str] Name of the antenna.

position [array_like] Vector position of the antenna.

center_frequency [float] Tuned frequency (Hz) of the dipole.

bandwidth [float] Bandwidth (Hz) of the antenna.

resistance [float] The noise resistance (ohm) of the antenna. Used to calculate the RMS voltage of the antenna noise.

orientation [array_like, optional] Vector direction of the z-axis of the antenna.

trigger_threshold [float, optional] Voltage threshold (V) above which signals will trigger.

effective_height [float, optional] Effective length (m) of the antenna. By default calculated by the tuned *center_frequency* of the dipole.

noisy [boolean, optional] Whether or not the antenna should add noise to incoming signals.

unique_noise_waveforms [int, optional] The number of expected noise waveforms needed for each received signal to have its own noise.

See also:

[*Antenna*](#) Base class for antennas.

Attributes

name [str] Name of the antenna.

position [array_like] Vector position of the antenna.

z_axis [ndarray] Vector direction of the z-axis of the antenna.

x_axis [ndarray] Vector direction of the x-axis of the antenna.

antenna_factor [float] Antenna factor used for converting electric field values to voltages.

efficiency [float] Antenna efficiency applied to incoming signal values.

threshold [float, optional] Voltage threshold (V) above which signals will trigger.

effective_height [float, optional] Effective length of the antenna. By default calculated by the tuned *center_frequency* of the dipole.

filter_coeffs [tuple of ndarray] Coefficients of the transfer function of the butterworth bandpass filter to be used for frequency response.

noisy [boolean] Whether or not the antenna should add noise to incoming signals.

unique_noises [int] The number of expected noise waveforms needed for each received signal to have its own noise.

freq_range [array_like] The frequency band in which the antenna operates (used for noise production).

temperature [float or None] The noise temperature (K) of the antenna. Used in combination with *resistance* to calculate the RMS voltage of the antenna noise.

resistance [float or None] The noise resistance (ohm) of the antenna. Used in combination with *temperature* to calculate the RMS voltage of the antenna noise.

noise_rms [float or None] The RMS voltage (V) of the antenna noise. If not `None`, this value will be used instead of the RMS voltage calculated from the values of *temperature* and *resistance*.

signals [list of Signal] The signals which have been received by the antenna.

is_hit Boolean of whether the antenna has been triggered.

is_hit_mc_truth Boolean of whether the antenna has been triggered by signal.

waveforms Signal + noise (if *noisy*) for each triggered antenna hit.

all_waveforms Signal + noise (if *noisy*) for all antenna hits.

Methods

<code>apply_response(signal[, direction, ...])</code>	Process the complete antenna response for an incoming signal.
<code>clear([reset_noise])</code>	Reset the antenna to an empty state.
<code>directional_gain(theta, phi)</code>	Calculate the (complex) directional gain of the antenna.
<code>frequency_response(frequencies)</code>	Calculate the (complex) frequency response of the antenna.
<code>full_waveform(times)</code>	Signal + noise (if <code>noisy</code>) for the given times.
<code>is_hit_during(times)</code>	Check if the antenna is triggered in a time range.
<code>make_noise(times)</code>	Creates a noise signal over the given times.
<code>polarization_gain(polarization)</code>	Calculate the (complex) polarization gain of the antenna.
<code>receive(signal[, direction, polarization, ...])</code>	Process and store one or more incoming (polarized) signals.
<code>set_orientation([z_axis, x_axis])</code>	Sets the orientation of the antenna.
<code>trigger(signal)</code>	Check if the antenna triggers on a given signal.

6.2.5 High-level Detectors (`pyrex.detector`)

Module containing higher-level detector-related classes.

The classes in this module are responsible for higher-level operations of the antennas and detectors than in the antenna module. This includes functions like front-end electronics chains and trigger systems.

<i><code>AntennaSystem</code></i>	Base class for antenna system with front-end processing.
<i><code>Detector</code></i>	Base class for detectors for easily building up sets of antennas.
<i><code>CombinedDetector</code></i>	Class for detectors which have been added together.

`pyrex.detector.AntennaSystem`

class `pyrex.detector.AntennaSystem` (*antenna*)

Base class for antenna system with front-end processing.

Behaves similarly to an antenna by passing some functionality downward to an antenna class, but additionally applies some front-end processing (e.g. an electronics chain) to the signals received.

Parameters

antenna [Antenna] Antenna class or subclass to be extended with a front end. Can also accept an `Antenna` object directly.

See also:

`pyrex.Antenna` Base class for antennas.

Attributes

antenna [Antenna] Antenna object extended by the front end.

lead_in_time [float] Lead-in time (s) required for the front end to equilibrate. Automatically added in before calculation of signals and waveforms.

is_hit Boolean of whether the antenna system has been triggered.

is_hit_mc_truth Boolean of whether the antenna has been triggered by signal.

signals The signals received by the antenna with front-end processing.

waveforms The antenna system signal + noise for each triggered hit.

all_waveforms The antenna system signal + noise for all hits.

Methods

<code>apply_response(signal[, direction, ...])</code>	Process the complete antenna response for an incoming signal.
<code>clear([reset_noise])</code>	Reset the antenna system to an empty state.
<code>front_end(signal)</code>	Apply front-end processes to a signal and return the output.
<code>full_waveform(times)</code>	Signal + noise for the antenna system for the given times.
<code>is_hit_during(times)</code>	Check if the antenna system is triggered in a time range.
<code>make_noise(times)</code>	Creates a noise signal over the given times.
<code>receive(signal[, direction, polarization, ...])</code>	Process and store one or more incoming (polarized) signals.
<code>set_orientation([z_axis, x_axis])</code>	Sets the orientation of the antenna system.
<code>setup_antenna(*args, **kwargs)</code>	Setup the antenna by passing along its init arguments.
<code>trigger(signal)</code>	Check if the antenna system triggers on a given signal.

pyrex.detector.Detector

class `pyrex.detector.Detector(*args, **kwargs)`

Base class for detectors for easily building up sets of antennas.

Designed for automatically generating antenna positions based on geometry parameters, then building all the antennas with some properties. Any parameters to the `__init__` method are automatically passed on to the `set_positions` method. Once the antennas have been built, the object can be directly iterated over to iterate over the antennas (as if the object were just a list of the antennas).

Raises

ValueError If `test_antenna_positions` is True and an antenna is found to be above the ice surface.

See also:

[`pyrex.Antenna`](#) Base class for antennas.

[`AntennaSystem`](#) Base class for antenna system with front-end processing.

Notes

When this class is subclassed, the `__init__` method will mirror the signature of the `set_positions` method so that parameters can be easily discovered.

The class is designed to be flexible in what defines a “detector”. This should allow for easier modularization by defining detectors whose subsets are detectors themselves, and so on. For example, a string of antennas could be set up as a subclass of `Detector` which sets up some antennas in a vertical line. Then a station could be set up as a subclass of `Detector` which sets up multiple instances of the string class at different positions. Then a final overarching detector class can subclass `Detector` and set up multiple instances of the station class at different positions. In this example the `subsets` of the overarching detector class would be the station objects, the `subsets` of the station objects would be the string objects, and the `subsets` of the string objects would finally be the antenna objects. But the way the iteration of the `Detector` class is built, iterating over that overarching detector class would iterate directly over each antenna in each string in each station as a simple 1D list.

Attributes

- antenna_positions** [list] List (potentially with sub-lists) of the positions of the antennas generated by the `set_positions` method.
- subsets** [list] List of the antenna or detector objects which make up the detector.
- test_antenna_positions** [boolean] Class attribute for whether or not an error should be raised if antenna positions are found above the surface of the ice (where simulation behavior is ill-defined). Defaults to `True`.

Methods

<code>build_antennas(*args, **kwargs)</code>	Creates antenna objects at the set antenna positions.
<code>clear([reset_noise])</code>	Reset the detector to an empty state.
<code>set_positions(*args, **kwargs)</code>	Sets the positions of antennas in the detector.
<code>triggered(*args[, require_mc_truth])</code>	Check if the detector is triggered based on its current state.

`pyrex.detector.CombinedDetector`

class `pyrex.detector.CombinedDetector` (**detectors*)

Class for detectors which have been added together.

Designed to allow addition of `Detector` and `Antenna`-like objects which can still build all antennas and trigger by smartly passing down keyword arguments to the subsets. Maintains all other properties of the `Detector` objects.

Raises

- ValueError** If `test_antenna_positions` is `True` and an antenna is found to be above the ice surface.

See also:

`pyrex.Antenna` Base class for antennas.

`AntennaSystem` Base class for antenna system with front-end processing.

`Detector` Base class for detectors for easily building up sets of antennas.

Attributes

antenna_positions [list] List of the positions of the antennas in the detector.

subsets [list] List of the antenna or detector objects which make up the detector.

test_antenna_positions [boolean] Class attribute for whether or not an error should be raised if antenna positions are found above the surface of the ice (where simulation behavior is ill-defined). Defaults to `True`.

Methods

<code>build_antennas(*args, **kwargs)</code>	Creates antenna objects at the set antenna positions.
<code>clear([reset_noise])</code>	Reset the detector to an empty state.
<code>set_positions(*args, **kwargs)</code>	Sets the positions of antennas in the detector.
<code>triggered(*args[, require_mc_truth])</code>	Check if the detector is triggered based on its current state.

6.2.6 Earth Model (`pyrex.earth_model`)

Module containing earth model functions.

The earth model uses the Preliminary Earth Model (PREM) for density as a function of radius and a simple integrator for calculation of the slant depth along a straight path through the Earth.

<code>prem_density</code>	Calculates the Earth's density at a given radius.
<code>slant_depth</code>	Calculates the material thickness of a chord cutting through Earth.

`pyrex.earth_model.prem_density`

`pyrex.earth_model.prem_density(r)`

Calculates the Earth's density at a given radius.

Density from the Preliminary reference Earth Model (PREM). Supports passing an array of radii or a single radius.

Parameters

r [array_like] Radius (m) at which to calculate density.

Returns

array_like Density (g/cm³) of the Earth at the given radii.

Notes

The density calculation is based on the Preliminary reference Earth Model [1].

References

[1]

pyrex.earth_model.slant_depth

`pyrex.earth_model.slant_depth` (*angle*, *depth*, *step*=500)

Calculates the material thickness of a chord cutting through Earth.

Integrates the Earth's density along the chord. Uses the PREM model for density.

Parameters

angle [float] Nadir angle (radians) of the chord's direction.

depth [float] (Positive-valued) depth (m) of the chord endpoint.

step [float, optional] Step size (m) for the integration.

Returns

float Material thickness (g/cm²) along the chord starting from *depth* and passing through the Earth at *angle*.

See also:

[`prem_density`](#) Calculates the Earth's density at a given radius.

6.2.7 Ice Models (`pyrex.ice_model`)

Module containing ice model classes.

Contains models for ice at the South Pole and at Summit Station in Greenland, plus a model for ice with uniform index of refraction.

<code>AntarcticIce</code>	Class describing the ice at the south pole.
<code>UniformIce</code>	Class describing ice with a uniform index of refraction.
<code>ArasimIce</code>	Class describing the ice at the south pole.
<code>GreenlandIce</code>	Class describing the ice at Summit Station in Greenland.

pyrex.ice_model.AntarcticIce

class `pyrex.ice_model.AntarcticIce` (*n0*=1.78, *k*=0.43, *a*=0.0132, *valid_range*=(-2850, 0), *index_above*=1, *index_below*=None)

Class describing the ice at the south pole.

For convenience, consists of static methods and class methods, so creating an instance of the class may not be necessary. In all methods, the depth *z* should be given as a negative value if it is below the surface of the ice.

Parameters

n0 [float, optional] Asymptotic index of refraction of the deep ice.

k [float, optional] Multiplicative factor for the index of refraction parameterization.

a [float, optional] Exponential factor for the index of refraction parameterization with units of 1/m.

valid_range [array_like of float, optional] Range of depths over which the uniform index of refraction applies. Assumed to have two elements where the first value is lower (deeper, more negative) than the second.

index_above [float or None, optional] Index of refraction above the ice region. If *None*, uses the same index of refraction as the top of the ice.

index_below [float or None, optional] Index of refraction below the ice region. If *None*, uses the same index of refraction as the bottom of the ice.

Notes

Parameterizations mostly based on South Pole ice characteristics outlined by Matt Newcomb [1].

References

[1]

Attributes

n0, k, a [float] Parameters of the index of refraction of the ice.

valid_range [tuple] Range of depths over which the ice model is valid. Consists of two elements where the first value is lower (deeper, more negative) than the second.

index_above The index of refraction above the ice’s valid range.

index_below The index of refraction below the ice’s valid range.

Methods

<code>attenuation_length(z, f)</code>	Calculates attenuation lengths for given depths and frequencies.
<code>contains(point)</code>	Determines if the given point is within the ice’s valid range.
<code>depth_with_index(n)</code>	Calculates the corresponding depth for a given index of refraction.
<code>gradient(z)</code>	Calculates the gradient of the index of refraction at a given depth.
<code>index(z)</code>	Calculates the index of refraction of the ice at a given depth.
<code>temperature(z)</code>	Calculates the temperature of the ice at a given depth.

pyrex.ice_model.UniformIce

class `pyrex.ice_model.UniformIce` (*index*, *valid_range*=(-2850, 0), *index_above*=1, *index_below*=None)

Class describing ice with a uniform index of refraction.

In all methods, the depth *z* should be given as a negative value if it is below the surface of the ice.

Parameters

index [float] Index of refraction of the ice.

valid_range [array_like of float] Range of depths over which the uniform index of refraction applies. Assumed to have two elements where the first value is lower (deeper, more negative) than the second.

index_above [float or None, optional] Index of refraction above the ice region. If *None*, uses the same index of refraction.

index_below [float or None, optional] Index of refraction below the ice region. If *None*, uses

the same index of refraction.

Attributes

n [float] Index of refraction of the ice.

valid_range [tuple] Range of depths over which the ice model is valid. Consists of two elements where the first value is lower (deeper, more negative) than the second.

index_above The index of refraction above the ice's valid range.

index_below The index of refraction below the ice's valid range.

Methods

<code>attenuation_length(z, f)</code>	Calculates attenuation lengths for given depths and frequencies.
<code>contains(point)</code>	Determines if the given point is within the ice's valid range.
<code>depth_with_index(n)</code>	Calculates the corresponding depth for a given index of refraction.
<code>gradient(z)</code>	Calculates the gradient of the index of refraction at a given depth.
<code>index(z)</code>	Calculates the index of refraction of the ice at a given depth.
<code>temperature(z)</code>	Calculates the temperature of the ice at a given depth.

`pyrex.ice_model.ArasimIce`

class `pyrex.ice_model.ArasimIce` (*n0=1.78, k=0.43, a=0.0132, valid_range=(-2850, 0), index_above=1, index_below=None*)

Class describing the ice at the south pole.

Designed to match ice model used in AraSim. For convenience, consists of static methods and class methods, so creating an instance of the class may not be necessary. In all methods, the depth *z* should be given as a negative value if it is below the surface of the ice.

Parameters

n0 [float, optional] Asymptotic index of refraction of the deep ice.

k [float, optional] Multiplicative factor for the index of refraction parameterization.

a [float, optional] Exponential factor for the index of refraction parameterization with units of 1/m.

valid_range [array_like of float, optional] Range of depths over which the uniform index of refraction applies. Assumed to have two elements where the first value is lower (deeper, more negative) than the second.

index_above [float or None, optional] Index of refraction above the ice region. If *None*, uses the same index of refraction as the top of the ice.

index_below [float or None, optional] Index of refraction below the ice region. If *None*, uses the same index of refraction as the bottom of the ice.

Attributes

n0, k, a [float] Parameters of the index of refraction of the ice.

valid_range [tuple] Range of depths over which the ice model is valid. Consists of two elements where the first value is lower (deeper, more negative) than the second.

index_above The index of refraction above the ice's valid range.

index_below The index of refraction below the ice's valid range.

Methods

<code>attenuation_length(z, f)</code>	Calculates attenuation lengths for given depths and frequencies.
<code>contains(point)</code>	Determines if the given point is within the ice's valid range.
<code>depth_with_index(n)</code>	Calculates the corresponding depth for a given index of refraction.
<code>gradient(z)</code>	Calculates the gradient of the index of refraction at a given depth.
<code>index(z)</code>	Calculates the index of refraction of the ice at a given depth.
<code>temperature(z)</code>	Calculates the temperature of the ice at a given depth.

pyrex.ice_model.GreenlandIce

class `pyrex.ice_model.GreenlandIce` (*n0=1.775, k=0.448, a=0.0247, valid_range=(-3000, 0), index_above=1, index_below=None*)

Class describing the ice at Summit Station in Greenland.

For convenience, consists of static methods and class methods, so creating an instance of the class may not be necessary. In all methods, the depth *z* should be given as a negative value if it is below the surface of the ice.

Parameters

n0 [float, optional] Asymptotic index of refraction of the deep ice.

k [float, optional] Multiplicative factor for the index of refraction parameterization.

a [float, optional] Exponential factor for the index of refraction parameterization with units of 1/m.

valid_range [array_like of float, optional] Range of depths over which the uniform index of refraction applies. Assumed to have two elements where the first value is lower (deeper, more negative) than the second.

index_above [float or None, optional] Index of refraction above the ice region. If *None*, uses the same index of refraction as the top of the ice.

index_below [float or None, optional] Index of refraction below the ice region. If *None*, uses the same index of refraction as the bottom of the ice.

Notes

Index of refraction parameterization based on a slightly altered version of the density parameterization at Summit Station [1]. The altered version ignores the break at small depths in order to have a uniform index of refraction parameterization matching the form of the Antarctic index. The temperature and attenuation length parameterizations are also based on parameterizations defined for Summit Station [2].

References

[1], [2]

Attributes

n0, k, a [float] Parameters of the index of refraction of the ice.

valid_range [tuple] Range of depths over which the ice model is valid. Consists of two elements where the first value is lower (deeper, more negative) than the second.

index_above The index of refraction above the ice's valid range.

index_below The index of refraction below the ice's valid range.

Methods

<code>attenuation_length(z, f)</code>	Calculates attenuation lengths for given depths and frequencies.
<code>contains(point)</code>	Determines if the given point is within the ice's valid range.
<code>depth_with_index(n)</code>	Calculates the corresponding depth for a given index of refraction.
<code>gradient(z)</code>	Calculates the gradient of the index of refraction at a given depth.
<code>index(z)</code>	Calculates the index of refraction of the ice at a given depth.
<code>temperature(z)</code>	Calculates the temperature of the ice at a given depth.

6.2.8 Ray Tracers (`pyrex.ray_tracing`)

Module containing classes for ray tracing through the ice.

Ray tracer classes correspond to ray trace path classes, where the ray tracer is responsible for calculating the existence and launch angle of paths between points, and the ray tracer path objects are responsible for returning information about propagation along their respective path.

<i>BasicRayTracePath</i>	Class for representing a single ray-trace solution between points.
<i>SpecializedRayTracePath</i>	Class for representing a single ray-trace solution between points.
<i>BasicRayTracer</i>	Class for calculating the ray-trace solutions between points.
<i>SpecializedRayTracer</i>	Class for calculating the ray-trace solutions between points.
<i>UniformRayTracePath</i>	Class for representing a single ray solution in uniform ice.
<i>UniformRayTracer</i>	Class for calculating ray solutions in uniform ice.
<i>RayTracer</i>	alias of <code>pyrex.ray_tracing.SpecializedRayTracer</code>
<i>RayTracePath</i>	alias of <code>pyrex.ray_tracing.SpecializedRayTracePath</code>

pyrex.ray_tracing.BasicRayTracePath

class `pyrex.ray_tracing.BasicRayTracePath` (*parent_tracer, launch_angle, direct*)

Class for representing a single ray-trace solution between points.

Stores parameters of the ray path with calculations performed by integrating z-steps of size `dz`. Most properties are lazily evaluated to save on computation time. If any attributes of the class instance are changed, the lazily-evaluated properties will be cleared.

Parameters

parent_tracer [BasicRayTracer] Ray tracer for which this path is a solution.

launch_angle [float] Launch angle (radians) of the ray path.

direct [boolean] Whether the ray path is direct. If `True` this means the path does not “turn over”. If `False` then the path does “turn over” by either reflection or refraction after reaching some maximum depth.

See also:

[`pyrex.internal_functions.LazyMutableClass`](#) Class with lazy properties which may depend on other class attributes.

[`BasicRayTracer`](#) Class for calculating the ray-trace solutions between points.

Notes

Even more attributes than those listed are available for the class, but are mainly for internal use. These attributes can be found by exploring the source code.

Attributes

from_point [ndarray] The starting point of the ray path.

to_point [ndarray] The ending point of the ray path.

theta0 [float] The launch angle of the ray path at *from_point*.

ice The ice model used for the ray tracer.

dz [float] The z-step (m) to be used for integration of the ray path attributes.

direct [boolean] Whether the ray path is direct. If `True` this means the path does not “turn over”. If `False` then the path does “turn over” by either reflection or refraction after reaching some maximum depth.

emitted_direction

received_direction

path_length

tof

coordinates

Methods

<code>attenuation(f)</code>	Calculate the attenuation factor for signal frequencies.
<code>propagate([signal, polarization])</code>	Propagate the signal with optional polarization along the ray path.
<code>theta(z)</code>	Polar angle of the ray at the given depths.
<code>z_integral(integrand)</code>	Calculate the numerical integral of the given integrand.

pyrex.ray_tracing.SpecializedRayTracePath

class `pyrex.ray_tracing.SpecializedRayTracePath` (*parent_tracer, launch_angle, direct*)

Class for representing a single ray-trace solution between points.

Stores parameters of the ray path with calculations performed analytically (with the exception of attenuation). These calculations require the index of refraction of the ice to be of the form $n(z)=n_0-k*\exp(a*z)$. However this restriction allows for most of the integrations to be performed analytically. The attenuation is the only attribute which is still calculated by numerical integration with z-steps of size dz . Most properties are lazily evaluated to save on computation time. If any attributes of the class instance are changed, the lazily-evaluated properties will be cleared.

Parameters

parent_tracer [SpecializedRayTracer] Ray tracer for which this path is a solution.

launch_angle [float] Launch angle (radians) of the ray path.

direct [boolean] Whether the ray path is direct. If `True` this means the path does not “turn over”. If `False` then the path does “turn over” by either reflection or refraction after reaching some maximum depth.

See also:

`pyrex.internal_functions.LazyMutableClass` Class with lazy properties which may depend on other class attributes.

`SpecializedRayTracer` Class for calculating the ray-trace solutions between points.

Notes

Even more attributes than those listed are available for the class, but are mainly for internal use. These attributes can be found by exploring the source code.

The requirement that the ice model go as $n(z)=n_0-k*\exp(a*z)$ is implemented by requiring the ice model to inherit from `AntarcticIce`. Obviously this is not fool-proof, but likely the ray tracing will obviously fail if the index follows a very different functional form.

Attributes

from_point [ndarray] The starting point of the ray path.

to_point [ndarray] The ending point of the ray path.

theta0 [float] The launch angle of the ray path at *from_point*.

ice The ice model used for the ray tracer.

dz [float] The z-step (m) to be used for integration of the ray path attributes.

direct [boolean] Whether the ray path is direct. If `True` this means the path does not “turn over”. If `False` then the path does “turn over” by either reflection or refraction after reaching some maximum depth.

uniformity_factor [float] Factor (<1) of the base index of refraction (n_0 in the ice model) beyond which calculations start to break down numerically.

beta_tolerance [float] β value (near 0) below which calculations start to break down numerically.

emitted_direction

received_direction

path_length

tof

coordinates

Methods

<code>attenuation(f)</code>	Calculate the attenuation factor for signal frequencies.
<code>propagate([signal, polarization])</code>	Propagate the signal with optional polarization along the ray path.
<code>theta(z)</code>	Polar angle of the ray at the given depths.
<code>z_integral(integrand[, numerical, x_func])</code>	Calculate the integral of the given integrand.

`pyrex.ray_tracing.BasicRayTracer`

class `pyrex.ray_tracing.BasicRayTracer` (*from_point*, *to_point*,
ice_model=`<pyrex.ice_model.AntarcticIce object>`,
dz=1)

Class for calculating the ray-trace solutions between points.

Calculations performed by integrating z-steps of size dz . Most properties are lazily evaluated to save on computation time. If any attributes of the class instance are changed, the lazily-evaluated properties will be cleared.

Parameters

from_point [array_like] Vector starting point of the ray path.

to_point [array_like] Vector ending point of the ray path.

ice_model [optional] The ice model used for the ray tracer.

dz [float, optional] The z-step (m) to be used for integration of the ray path attributes.

See also:

`pyrex.internal_functions.LazyMutableClass` Class with lazy properties which may depend on other class attributes.

`BasicRayTracePath` Class for representing a single ray-trace solution between points.

Notes

Even more attributes than those listed are available for the class, but are mainly for internal use. These attributes can be found by exploring the source code.

Attributes

from_point [ndarray] The starting point of the ray path.

to_point [ndarray] The ending point of the ray path.

ice The ice model used for the ray tracer.

dz [float] The z-step (m) to be used for integration of the ray path attributes.

solution_class Class for representing a single ray-trace solution between points.

exists

expected_solutions

solutions

Methods

<code>angle_search(true_r, r_function, min_angle, ...)</code>	Calculates the angle where <i>r_function</i> (angle) == <i>true_r</i> .
<code>solution_class</code>	alias of <i>BasicRayTracePath</i>

pyrex.ray_tracing.SpecializedRayTracer

class `pyrex.ray_tracing.SpecializedRayTracer` (*from_point*, *to_point*,
ice_model=<pyrex.ice_model.AntarcticIce
object>, *dz*=1)

Class for calculating the ray-trace solutions between points.

Calculations in this class require the index of refraction of the ice to be of the form $n(z)=n_0-k*\exp(a*z)$. However this restriction allows for most of the integrations to be performed analytically. Most properties are lazily evaluated to save on computation time. If any attributes of the class instance are changed, the lazily-evaluated properties will be cleared.

Parameters

from_point [array_like] Vector starting point of the ray path.

to_point [array_like] Vector ending point of the ray path.

ice_model [optional] The ice model used for the ray tracer.

dz [float, optional] The z-step (m) to be used for integration of the ray path attributes.

See also:

pyrex.internal_functions.LazyMutableClass Class with lazy properties which may depend on other class attributes.

SpecializedRayTracePath Class for representing a single ray-trace solution between points.

Notes

Even more attributes than those listed are available for the class, but are mainly for internal use. These attributes can be found by exploring the source code.

The requirement that the ice model go as $n(z)=n_0-k*\exp(a*z)$ is implemented by requiring the ice model to inherit from `AntarcticIce`. Obviously this is not fool-proof, but likely the ray tracing will obviously fail if the index follows a very different functional form.

Attributes

from_point [ndarray] The starting point of the ray path.

to_point [ndarray] The ending point of the ray path.

ice The ice model used for the ray tracer.

dz [float] The z-step (m) to be used for integration of the ray path attributes.

solution_class Class for representing a single ray-trace solution between points.

exists

expected_solutions

solutions

Methods

<code>angle_search(true_r, r_function, min_angle, ...)</code>	Calculates the angle where <i>r_function</i> (angle) == <i>true_r</i> .
<code>solution_class</code>	alias of <i>SpecializedRayTracePath</i>

pyrex.ray_tracing.UniformRayTracePath

class `pyrex.ray_tracing.UniformRayTracePath` (*parent_tracer, launch_angle, reflections*)

Class for representing a single ray solution in uniform ice.

Stores parameters of the ray path through uniform ice. Most properties are lazily evaluated to save on computation time. If any attributes of the class instance are changed, the lazily-evaluated properties will be cleared.

Parameters

parent_tracer [UniformRayTracer] Ray tracer for which this path is a solution.

launch_angle [float] Launch angle (radians) of the ray path.

reflections [int] Number of reflections made by the ray path at boundaries of the ice.

See also:

`pyrex.internal_functions.LazyMutableClass` Class with lazy properties which may depend on other class attributes.

`UniformRayTracer` Class for calculating ray solutions in uniform ice.

Notes

Even more attributes than those listed are available for the class, but are mainly for internal use. These attributes can be found by exploring the source code.

Attributes

from_point [ndarray] The starting point of the ray path.

to_point [ndarray] The ending point of the ray path.

theta0 [float] The launch angle of the ray path at *from_point*.

ice The ice model used for the ray tracer.

direct [boolean] Whether the ray path is direct (does not reflect).

emitted_direction

received_direction

path_length

tof

coordinates

Methods

<code>attenuation(f[, dz])</code>	Calculate the attenuation factor for signal frequencies.
<code>propagate([signal, polarization])</code>	Propagate the signal with optional polarization along the ray path.

pyrex.ray_tracing.UniformRayTracer

class `pyrex.ray_tracing.UniformRayTracer` (*from_point, to_point, ice_model*)

Class for calculating ray solutions in uniform ice.

Calculations performed using straight-line paths. Most properties are lazily evaluated to save on computation time. If any attributes of the class instance are changed, the lazily-evaluated properties will be cleared.

Parameters

from_point [array_like] Vector starting point of the ray path.

to_point [array_like] Vector ending point of the ray path.

ice_model The ice model used for the ray tracer.

See also:

`pyrex.internal_functions.LazyMutableClass` Class with lazy properties which may depend on other class attributes.

`UniformRayTracePath` Class for representing a single ray solution in uniform ice.

Notes

Even more attributes than those listed are available for the class, but are mainly for internal use. These attributes can be found by exploring the source code.

Attributes

from_point [ndarray] The starting point of the ray path.

to_point [ndarray] The ending point of the ray path.

ice The ice model used for the ray tracer.

solution_class Class for representing a single ray solution in uniform ice.

exists

expected_solutions

solutions

Methods

<code>solution_class</code>	alias of <i>UniformRayTracePath</i>
-----------------------------	-------------------------------------

pyrex.ray_tracing.RayTracer

`pyrex.ray_tracing.RayTracer`
alias of *pyrex.ray_tracing.SpecializedRayTracer*

pyrex.ray_tracing.RayTracePath

`pyrex.ray_tracing.RayTracePath`
alias of *pyrex.ray_tracing.SpecializedRayTracePath*

6.2.9 Particles and Interaction Models (pyrex.particle)

Module for particles (neutrinos) and neutrino interactions in the ice.

Included in the module are the *Particle* class for storing particle/shower attributes and some *Interaction* classes which store models describing neutrino interactions.

<i>Event</i>	Class for storing a tree of <i>Particle</i> objects representing an event.
<i>Particle</i>	Class for storing particle attributes.
<i>Interaction</i>	Base class for describing neutrino interaction attributes.
<i>GQRSInteraction</i>	Class for describing neutrino interaction attributes.
<i>CTWInteraction</i>	Class for describing neutrino interaction attributes.
<i>NeutrinoInteraction</i>	alias of <i>pyrex.particle.CTWInteraction</i>

pyrex.particle.Event

class `pyrex.particle.Event` (*roots*)
Class for storing a tree of *Particle* objects representing an event.

The event may be comprised of any number of root *Particle* objects specified at initialization. Each *Particle* in the tree may have any number of child *Particle* objects. Iterating the tree will return all *Particle* objects, but in no guaranteed order.

Parameters

roots [Particle or list of Particle] Root *Particle* objects for the event tree.

See also:

Particle Class for storing particle attributes.

Attributes

roots [Particle or list of Particle] Root *Particle* objects for the event tree.

Methods

<code>add_children(parent, children)</code>	Add the given <i>children</i> to the <i>parent Particle</i> object.
<code>get_children(parent)</code>	Get the children of the given <i>parent Particle</i> object.
<code>get_from_level(level)</code>	Get all <i>Particle</i> objects some <i>level</i> deep into the event tree.
<code>get_parent(child)</code>	Get the parent of the given <i>child Particle</i> object.

pyrex.particle.Particle

```
class pyrex.particle.Particle(particle_id, vertex, direction, energy, interaction_model=<class
                             'pyrex.particle.CTWInteraction'>, interaction_type=None,
                             weight=None)
```

Class for storing particle attributes.

Parameters

particle_id Identification value of the particle type. Values should be from the `Particle.Type` enum, but integer or string values may work if carefully chosen. `Particle.Type.undefined` by default.

vertex [array_like] Vector position (m) of the particle.

direction [array_like] Vector direction of the particle's velocity.

energy [float] Energy (GeV) of the particle.

interaction_model [optional] Class to use to describe interactions of the particle. Should inherit from (or behave like) the base `Interaction` class.

interaction_type [optional] Value of the interaction type. Values should be from the `Interaction.Type` enum, but integer or string values may work if carefully chosen. By default, the *interaction_model* will choose an interaction type.

weight [float, optional] Total Monte Carlo weight of the particle. The calculation of this weight depends on the particle generation method, but this value should be the total weight representing the probability of this particle's event occurring.

See also:

Interaction Base class for describing neutrino interaction attributes.

Attributes

id [Particle.Type] Identification value of the particle type.

vertex [array_like] Vector position (m) of the particle.

direction [array_like] (Unit) vector direction of the particle's velocity.

energy [float] Energy (GeV) of the particle.

interaction [Interaction] Instance of the *interaction_model* class to be used for calculations related to interactions of the particle.

weight [float] Total Monte Carlo weight of the particle

survival_weight [float] Monte Carlo weight of the particle surviving to its vertex. Represents the probability that the particle does not interact along its path through the Earth.

interaction_weight [float] Monte Carlo weight of the particle interacting at its vertex. Represents the probability that the the particle interacts specifically at its given vertex.

Methods

Type	Enum containing possible particle types.
------	--

pyrex.particle.Interaction

class `pyrex.particle.Interaction` (*particle*, *kind=None*)

Base class for describing neutrino interaction attributes.

Defaults to values which will result in zero probability of interaction.

Parameters

particle [Particle] `Particle` object for which the interaction is defined.

kind [optional] Value of the interaction type. Values should be from the `Interaction.Type` enum, but integer or string values may work if carefully chosen. By default will be chosen by the `choose_interaction` method.

See also:

Particle Class for storing particle attributes.

Attributes

particle [Particle] `Particle` object for which the interaction is defined.

kind [Interaction.Type] Value of the interaction type.

inelasticity [float] Inelasticity value from `choose_inelasticity` distribution for the interaction.

em_frac [float] Fraction of *particle* energy deposited into an electromagnetic shower.

had_frac [float] Fraction of *particle* energy deposited into a hadronic shower.

total_cross_section The total neutrino cross section (cm²) of the *particle* type.

total_interaction_length The neutrino interaction length (cmwe) of the *particle* type.

cross_section The neutrino cross section (cm²) of the *particle* interaction.

interaction_length The neutrino interaction length (cmwe) of the *particle* interaction.

Methods

Type	Enum containing possible interaction types.
choose_inelasticity()	Choose an inelasticity for the <code>particle</code> attribute's shower.
choose_interaction()	Choose an interaction type for the <code>particle</code> attribute.
choose_shower_fractions()	Choose the electromagnetic and hadronic shower fractions.

pyrex.particle.GQRSInteraction

class pyrex.particle.GQRSInteraction (*particle*, *kind=None*)

Class for describing neutrino interaction attributes.

Calculates values related to the interaction(s) of a given *particle*. Values based on GQRS 1998.

Parameters

particle [Particle] `Particle` object for which the interaction is defined.

kind [optional] Value of the interaction type. Values should be from the `Interaction.Type` enum, but integer or string values may work if carefully chosen. By default will be chosen by the `choose_interaction` method.

See also:

Interaction Base class for describing neutrino interaction attributes.

Particle Class for storing particle attributes.

Notes

Neutrino intractions based on the GQRS Ultrahigh-Energy Neutrino Interactions paper [1].

References

[1]

Attributes

particle [Particle] `Particle` object for which the interaction is defined.

kind [Interaction.Type] Value of the interaction type.

inelasticity [float] Inelasticity value from `choose_inelasticity` distribution for the interaction.

em_frac [float] Fraction of *particle* energy deposited into an electromagnetic shower.

had_frac [float] Fraction of *particle* energy deposited into a hadronic shower.

total_cross_section The total neutrino cross section (cm²) of the *particle* type.

total_interaction_length The neutrino interaction length (cmwe) of the *particle* type.

cross_section The neutrino cross section (cm²) of the *particle* interaction.

interaction_length The neutrino interaction length (cmwe) of the `particle` interaction.

Methods

Type	Enum containing possible interaction types.
<code>choose_inelasticity()</code>	Choose an inelasticity for the <code>particle</code> attribute's shower.
<code>choose_interaction()</code>	Choose an interaction type for the <code>particle</code> attribute.
<code>choose_shower_fractions()</code>	Choose the electromagnetic and hadronic shower fractions.

pyrex.particle.CTWInteraction

class `pyrex.particle.CTWInteraction` (*particle*, *kind=None*)

Class for describing neutrino interaction attributes.

Calculates values related to the interaction(s) of a given *particle*. Values based on CTW 2011.

Parameters

particle [Particle] `Particle` object for which the interaction is defined.

kind [optional] Value of the interaction type. Values should be from the `Interaction`. Type enum, but integer or string values may work if carefully chosen. By default will be chosen by the `choose_interaction` method.

See also:

Interaction Base class for describing neutrino interaction attributes.

Particle Class for storing particle attributes.

Notes

Neutrino intractions based on the CTW High Energy Neutrino-Nucleon Cross Sections paper [1]. Secondary generation method to determine shower fractions was pulled from AraSim, which is unchanged from icemc.

References

[1]

Attributes

particle [Particle] `Particle` object for which the interaction is defined.

kind [Interaction.Type] Value of the interaction type.

inelasticity [float] Inelasticity value from `choose_inelasticity` distribution for the interaction.

em_frac [float] Fraction of *particle* energy deposited into an electromagnetic shower.

had_frac [float] Fraction of *particle* energy deposited into a hadronic shower.

total_cross_section The total neutrino cross section (cm²) of the `particle` type.

total_interaction_length The neutrino interaction length (cmwe) of the `particle` type.

cross_section The neutrino cross section (cm²) of the `particle` interaction.

interaction_length The neutrino interaction length (cmwe) of the `particle` interaction.

Methods

Type	Enum containing possible interaction types.
choose_inelasticity()	Choose an inelasticity for the <code>particle</code> attribute's shower.
choose_interaction()	Choose an interaction type for the <code>particle</code> attribute.
choose_shower_fractions()	Choose the electromagnetic and hadronic shower fractions.

pyrex.particle.NeutrinoInteraction

`pyrex.particle.NeutrinoInteraction`
alias of `pyrex.particle.CTWInteraction`

6.2.10 Event Generators (pyrex.generation)

Module for particle (neutrino) generators.

Generators are responsible for the input of events into the simulation.

<i>BaseGenerator</i>	Base class for neutrino generators.
<i>CylindricalGenerator</i>	Class to generate neutrino vertices in a cylindrical ice volume.
<i>RectangularGenerator</i>	Class to generate neutrino vertices in a rectangular ice volume.
<i>ShadowGenerator</i>	Class to generate neutrino vertices with Earth shadowing.
<i>ListGenerator</i>	Class to generate neutrino events from a list.
<i>NumpyFileGenerator</i>	Class to generate neutrino events from numpy file(s).
<i>FileGenerator</i>	Class to generate neutrino events from simulation file(s).

pyrex.generation.BaseGenerator

class `pyrex.generation.BaseGenerator` (*energy*, *shadow=False*, *flavor_ratio=(1, 1, 1)*, *interaction_model=<class 'pyrex.particle.CTWInteraction'>*)

Base class for neutrino generators.

Provides methods for generating neutrino attributes except for neutrino vertex, which should be provided by child classes to generate neutrinos in specific volumes.

Parameters

energy [float or function] Energy (GeV) of the neutrinos. If `float`, all neutrinos have the

same constant energy. If `function`, neutrinos are generated with the energy returned by successive function calls.

shadow [bool, optional] Whether Earth shadowing effects should be used to reject events. If `True` then neutrinos which don't survive transit through the Earth will be skipped when creating events. If `False` then all events are allowed and assigned a weight to scale their probability of occurrence.

flavor_ratio [array_like, optional] Flavor ratio of neutrinos to be generated. Of the form [electron, muon, tau] neutrino fractions.

interaction_model [optional] Class to use to describe interactions of the generated particles. Should inherit from (or behave like) the base `Interaction` class.

See also:

`pyrex.particle.Interaction` Base class for describing neutrino interaction attributes.

Attributes

count [int] Number of neutrinos produced by the generator, including those not returned due to Earth shadowing or other effects.

get_energy [function] Function returning energy (GeV) of the neutrinos by successive function calls.

shadow [bool] Whether Earth shadowing effects will be used to reject events.

ratio [ndarray] (Normalized) flavor ratio of neutrinos to be generated. Of the form [electron, muon, tau] neutrino fractions.

interaction_model [Interaction] Class to use to describe interactions of the generated particles.

Methods

<code>create_event()</code>	Generate a neutrino event in the ice volume.
<code>get_direction()</code>	Get the direction of the next particle to be generated.
<code>get_exit_points(particle)</code>	Get the intersections of the particle path with the ice volume edges.
<code>get_particle_type()</code>	Get the particle type of the next particle to be generated.
<code>get_vertex()</code>	Get the vertex of the next particle to be generated.
<code>get_weights(particle)</code>	Get the weighting factors to be applied to the particle.

pyrex.generation.CylindricalGenerator

```
class pyrex.generation.CylindricalGenerator(dr, dz, energy, shadow=False,
                                           flavor_ratio=(1, 1, 1), in-
                                           teraction_model=<class
                                           'pyrex.particle.CTWInteraction'>)
```

Class to generate neutrino vertices in a cylindrical ice volume.

Generates neutrinos in a cylinder with given radius and height.

Parameters

dr [float] Radius of the ice volume. Neutrinos generated within $(0, dr)$.

dz [float] Height of the ice volume in the z-direction. Neutrinos generated within $(-dz, 0)$.

energy [float or function] Energy (GeV) of the neutrinos. If `float`, all neutrinos have the same constant energy. If `function`, neutrinos are generated with the energy returned by successive function calls.

shadow [bool, optional] Whether Earth shadowing effects should be used to reject events. If `True` then neutrinos which don't survive transit through the Earth will be skipped when creating events. If `False` then all events are allowed and assigned a weight to scale their probability of occurrence.

flavor_ratio [array_like, optional] Flavor ratio of neutrinos to be generated. Of the form [electron, muon, tau] neutrino fractions.

interaction_model [optional] Class to use to describe interactions of the generated particles. Should inherit from (or behave like) the base `Interaction` class.

See also:

`pyrex.particle.Interaction` Base class for describing neutrino interaction attributes.

Attributes

count [int] Number of neutrinos produced by the generator, including those not returned due to Earth shadowing or other effects.

dr [float] Radius of the ice volume. Neutrinos generated within $(0, dr)$.

dz [float] Height of the ice volume in the z-direction. Neutrinos generated within $(-dz, 0)$.

get_energy [function] Function returning energy (GeV) of the neutrinos by successive function calls.

shadow [bool] Whether Earth shadowing effects will be used to reject events.

ratio [ndarray] (Normalized) flavor ratio of neutrinos to be generated. Of the form [electron, muon, tau] neutrino fractions.

interaction_model [Interaction] Class to use to describe interactions of the generated particles.

Methods

<code>create_event()</code>	Generate a neutrino event in the ice volume.
<code>get_direction()</code>	Get the direction of the next particle to be generated.
<code>get_exit_points(particle)</code>	Get the intersections of the particle path with the ice volume edges.
<code>get_particle_type()</code>	Get the particle type of the next particle to be generated.
<code>get_vertex()</code>	Get the vertex of the next particle to be generated.
<code>get_weights(particle)</code>	Get the weighting factors to be applied to the particle.

pyrex.generation.RectangularGenerator

```
class pyrex.generation.RectangularGenerator(dx, dy, dz, energy, shadow=False,
                                           flavor_ratio=(1, 1, 1), interaction_model=<class
                                           'pyrex.particle.CTWInteraction'>)
```

Class to generate neutrino vertices in a rectangular ice volume.

Generates neutrinos in a box with given width, length, and height.

Parameters

- dx** [float] Width of the ice volume in the x-direction. Neutrinos generated within $(-dx / 2, dx / 2)$.
- dy** [float] Length of the ice volume in the y-direction. Neutrinos generated within $(-dy / 2, dy / 2)$.
- dz** [float] Height of the ice volume in the z-direction. Neutrinos generated within $(-dz, 0)$.
- energy** [float or function] Energy (GeV) of the neutrinos. If `float`, all neutrinos have the same constant energy. If `function`, neutrinos are generated with the energy returned by successive function calls.
- shadow** [bool, optional] Whether Earth shadowing effects should be used to reject events. If `True` then neutrinos which don't survive transit through the Earth will be skipped when creating events. If `False` then all events are allowed and assigned a weight to scale their probability of occurrence.
- flavor_ratio** [array_like, optional] Flavor ratio of neutrinos to be generated. Of the form [electron, muon, tau] neutrino fractions.
- interaction_model** [optional] Class to use to describe interactions of the generated particles. Should inherit from (or behave like) the base `Interaction` class.

See also:

[*pyrex.particle.Interaction*](#) Base class for describing neutrino interaction attributes.

Attributes

- count** [int] Number of neutrinos produced by the generator, including those not returned due to Earth shadowing or other effects.
- dx** [float] Width of the ice volume in the x-direction. Neutrinos generated within $(-dx / 2, dx / 2)$.
- dy** [float] Length of the ice volume in the y-direction. Neutrinos generated within $(-dy / 2, dy / 2)$.
- dz** [float] Height of the ice volume in the z-direction. Neutrinos generated within $(-dz, 0)$.
- get_energy** [function] Function returning energy (GeV) of the neutrinos by successive function calls.
- shadow** [bool] Whether Earth shadowing effects will be used to reject events.
- ratio** [ndarray] (Normalized) flavor ratio of neutrinos to be generated. Of the form [electron, muon, tau] neutrino fractions.
- interaction_model** [Interaction] Class to use to describe interactions of the generated particles.

Methods

<code>create_event()</code>	Generate a neutrino event in the ice volume.
<code>get_direction()</code>	Get the direction of the next particle to be generated.
<code>get_exit_points(particle)</code>	Get the intersections of the particle path with the ice volume edges.
<code>get_particle_type()</code>	Get the particle type of the next particle to be generated.
<code>get_vertex()</code>	Get the vertex of the next particle to be generated.
<code>get_weights(particle)</code>	Get the weighting factors to be applied to the particle.

pyrex.generation.ShadowGenerator

```
class pyrex.generation.ShadowGenerator(dx, dy, dz, energy, flavor_ratio=(1,
                                                                    1,
                                                                    1),
                                     interaction_model=<class
                                     'pyrex.particle.CTWInteraction'>)
```

Class to generate neutrino vertices with Earth shadowing.

Deprecated since version 1.8.2: *ShadowGenerator* has been replaced by *RectangularGenerator*. The same shadowing behavior can be achieved by providing the *shadow* argument.

Generates neutrinos in a box with given width, length, and height. Accounts for Earth shadowing by comparing the neutrino interaction length to the material thickness of the Earth along the neutrino path, and rejecting particles which would interact before reaching the vertex. Note the subtle difference in x and y ranges compared to the z range.

Parameters

- dx** [float] Width of the ice volume in the x-direction. Neutrinos generated within $(-dx / 2, dx / 2)$.
- dy** [float] Length of the ice volume in the y-direction. Neutrinos generated within $(-dy / 2, dy / 2)$.
- dz** [float] Height of the ice volume in the z-direction. Neutrinos generated within $(-dz, 0)$.
- energy** [float or function] Energy (GeV) of the neutrinos. If `float`, all neutrinos have the same constant energy. If `function`, neutrinos are generated with the energy returned by successive function calls.
- flavor_ratio** [array_like, optional] Flavor ratio of neutrinos to be generated. Of the form [electron, muon, tau] neutrino fractions.
- interaction_model** [optional] Class to use to describe interactions of the generated particles. Should inherit from (or behave like) the base `Interaction` class.

See also:

pyrex.particle.Interaction Base class for describing neutrino interaction attributes.

Attributes

- count** [int] Number of neutrinos produced by the generator, including those not returned due to Earth shadowing or other effects.
- dx** [float] Width of the ice volume in the x-direction. Neutrinos generated within $(-dx / 2, dx / 2)$.

- dy** [float] Length of the ice volume in the y-direction. Neutrinos generated within $(-dy / 2, dy / 2)$.
- dz** [float] Height of the ice volume in the z-direction. Neutrinos generated within $(-dz, 0)$.
- get_energy** [function] Function returning energy (GeV) of the neutrinos by successive function calls.
- shadow** [bool] Whether Earth shadowing effects will be used to reject events.
- ratio** [ndarray] (Normalized) flavor ratio of neutrinos to be generated. Of the form [electron, muon, tau] neutrino fractions.
- interaction_model** [Interaction] Class to use to describe interactions of the generated particles.

Methods

<code>create_event()</code>	Generate a neutrino event in the ice volume.
<code>get_direction()</code>	Get the direction of the next particle to be generated.
<code>get_exit_points(particle)</code>	Get the intersections of the particle path with the ice volume edges.
<code>get_particle_type()</code>	Get the particle type of the next particle to be generated.
<code>get_vertex()</code>	Get the vertex of the next particle to be generated.
<code>get_weights(particle)</code>	Get the weighting factors to be applied to the particle.

pyrex.generation.ListGenerator

class `pyrex.generation.ListGenerator` (*events*, *loop=True*)

Class to generate neutrino events from a list.

Generates events by simply pulling them from a list of `Event` objects. By default returns to the start of the list once the end is reached, but can optionally fail after reaching the list's end.

Parameters

- events** [Event, or list of Event] List of `Event` objects to draw from. If only a single `Event` object is given, creates a list of that event alone.
- loop** [boolean, optional] Whether or not to return to the start of the list after throwing the last `Event`. If `False`, raises an error if trying to throw after the last `Event`.

See also:

[`pyrex.Event`](#) Class for storing a tree of `Particle` objects representing an event.

[`pyrex.Particle`](#) Class for storing particle attributes.

Attributes

- count** [int] Number of neutrinos produced by the generator.
- events** [list of Event] List to draw `Event` objects from, sequentially.
- loop** [boolean] Whether or not to loop through the list more than once.

Methods

<code>create_event()</code>	Generate a neutrino event.
-----------------------------	----------------------------

`pyrex.generation.NumpyFileGenerator`

class `pyrex.generation.NumpyFileGenerator` (*files*, *interaction_model=<class 'pyrex.particle.CTWInteraction'>*)

Class to generate neutrino events from numpy file(s).

Generates neutrinos by pulling their attributes from a (list of) .npz file(s). Each file must have four to six arrays, containing the id values, vertices, directions, energies, and optional interaction types and weights respectively so the first particle will have properties given by the first elements of these arrays. Tries to smartly figure out which array is which based on their names, but if the arrays are unnamed, assumes they are in the order used above.

Parameters

files [str or list of str] List of file names containing neutrino information. If only a single file name is provided, creates a list with that file alone.

interaction_model [optional] Class used to describe the interactions of the stored particles.

Warning: This generator only supports `Event` objects containing a single `Particle` object. There is currently no way to read from files where an `Event` contains multiple `Particle` objects with some dependencies.

See also:

`pyrex.particle.Interaction` Base class for describing neutrino interaction attributes.

`pyrex.Event` Class for storing a tree of `Particle` objects representing an event.

`pyrex.Particle` Class for storing particle attributes.

Attributes

files [list of str] List of file names containing neutrino information.

ids [ndarray] Array of particle id values from the current file.

vertices [ndarray] Array of neutrino vertices from the current file.

directions [ndarray] Array of neutrino directions from the current file.

energies [ndarray] Array of neutrino energies from the current file.

interactions [ndarray] Array of interaction types from the current file.

weights [ndarray] Array of neutrino weights from the current file.

Methods

<code>create_event()</code>	Generate a neutrino.
-----------------------------	----------------------

pyrex.generation.FileGenerator

```
class pyrex.generation.FileGenerator (files, slice_range=100, interaction_model=<class 'pyrex.particle.CTWInteraction'>)
```

Class to generate neutrino events from simulation file(s).

Generates neutrinos by pulling their attributes from a (list of) simulation output file(s). Designed to make reproducing simulations easier.

Parameters

files [str or list of str] List of file names containing neutrino event information. If only a single file name is provided, creates a list with that file alone.

slice_range [int, optional] Number of events to load into memory at a time from the files. Increasing this value should result in an improvement in speed, while decreasing this value should result in an improvement in memory consumption.

interaction_model [optional] Class used to describe the interactions of the stored particles.

Warning: This generator only supports `Event` objects containing a single level of `Particle` objects. Any dependencies among `Particle` objects will be ignored and they will all appear in the root level.

See also:

pyrex.particle.Interaction Base class for describing neutrino interaction attributes.

pyrex.Event Class for storing a tree of `Particle` objects representing an event.

pyrex.Particle Class for storing particle attributes.

Attributes

count [int] Number of neutrinos produced by the generator.

files [list of str] List of file names containing neutrino information.

Methods

<code>create_event()</code>	Generate a neutrino.
-----------------------------	----------------------

6.2.11 Simulation Kernel (`pyrex.kernel`)

Module for the simulation kernel.

The simulation kernel is responsible for running through the simulation chain by controlling classes and objects which will independently produce neutrinos, create corresponding signals, propagate the signals to antennas, and handle antenna processing of the signals.

<i>EventKernel</i>	High-level kernel for controlling event simulation.
--------------------	---

pyrex.kernel.EventKernel

```
class pyrex.kernel.EventKernel (generator, antennas, ice_model=<pyrex.ice_model.AntarcticIce
object>, ray_tracer=<class 'pyrex.ray_tracing.SpecializedRayTracer'>,
signal_model=<class 'pyrex.askaryan.ARVZAskaryanSignal'>,
signal_times=array([-2.000e-08, -1.995e-08, -1.990e-08,
..., 7.985e-08, 7.990e-08, 7.995e-08]), event_writer=None,
triggers=None)
```

High-level kernel for controlling event simulation.

The kernel is responsible for handling the classes and objects which control the major simulation steps: particle creation, signal production, signal propagation, and antenna response. The modular kernel structure allows for easy switching of the classes or objects which handle any of the simulation steps.

Parameters

- generator** A particle generator to create neutrino events.
- antennas** An iterable object consisting of antenna objects which can receive and store signals.
- ice_model** [optional] An ice model describing the ice surrounding the *antennas*.
- ray_tracer** [optional] A ray tracer capable of propagating signals from the neutrino vertex to the antenna positions.
- signal_model** [optional] A signal class which generates signals based on the particle.
- signal_times** [array_like, optional] The array of times over which the neutrino signal should be generated.
- event_writer** [File, optional] A file object to be used for writing data output.
- triggers** [function or dict, optional] A function or dictionary with function values representing trigger conditions of the detector. If a dictionary, must have a “global” key with its value representing the global detector trigger.

See also:

- pyrex.Event** Class for storing a tree of *Particle* objects representing an event.
- pyrex.Particle** Class for storing particle attributes.
- pyrex.ice_model.AntarcticIce** Class describing the ice at the south pole.
- pyrex.RayTracer** Class for calculating the ray-trace solutions between points.
- pyrex.AskaryanSignal** Class for generating Askaryan signals according to ARVZ parameterization.
- pyrex.File** Class for reading or writing data files.

Notes

The kernel is designed to be modular so individual parts of the simulation chain can be exchanged. In order to interchange the pieces, their classes require the following at a minimum:

The particle generator *generator* must have a `create_event` method which takes no arguments and returns a *Event* object consisting of *Particle* objects with `vertex`, `direction`, `energy`, and `weight` attributes.

The antenna iterable *antennas* must yield each antenna object once when iterating directly over *antennas*. Each antenna object must have a `position` attribute and a `receive` method which takes a signal object as its first argument, and `ndarray` objects as `direction` and `polarization` keyword arguments.

The *ice_model* must have an `index` method returning the index of refraction given a (negative-valued) depth, and it must support anything required of it by the *ray_tracer*.

The *ray_tracer* must be initialized with the particle vertex and an antenna position as its first two arguments, and the *ice_model* of the kernel as the `ice_model` keyword argument. The ray tracer must also have `exists` and `solutions` attributes, the first of which denotes whether any paths exist between the given points and the second of which is an iterable revealing each path between the points. These paths must have `emitted_direction`, `received_direction`, and `path_length` attributes, as well as a `propagate` method which takes a signal object and applies the propagation effects of the path in-place to that object.

The *signal_model* must be initialized with the *signal_times* array, a *Particle* object from the *Event*, the *viewing_angle* and *viewing_distance* according to the *ray_tracer*, and the *ice_model*. The object created should be a *Signal* object with `times` and `values` attributes representing the time-domain Askaryan signal produced by the *Particle*.

Attributes

- gen** The particle generator responsible for particle creation.
- antennas** The iterable of antennas responsible for handling applying their response and storing the resulting signals.
- ice** The ice model describing the ice containing the *antennas*.
- ray_tracer** The ray tracer responsible for signal propagation through the *ice*.
- signal_model** The signal class to use to generate signals based on the particle.
- signal_times** The array of times over which the neutrino signal should be generated.
- writer** The file object to be used for writing data output.
- triggers** The trigger condition(s) of the detector.

Methods

<code>event()</code>	Create a neutrino event and run it through the simulation chain.
----------------------	--

6.2.12 Data File I/O (`pyrex.io`)

Module containing classes for reading and writing data files.

Includes reader and writer for hdf5 data files, as well as base reader and writer classes which can be extended to read and write other file formats.

<i>File</i>	Class for reading or writing data files.
<i>HDF5Reader</i>	Class for reading data from an hdf5 file.
<i>HDF5Writer</i>	Class for writing data to an hdf5 file.
<i>EventIterator</i>	Class for iterating over event data from an hdf5 file.

`pyrex.io.File`

class `pyrex.io.File`

Class for reading or writing data files.

Works as a context manager and allows for reading or writing simulation/real data or analysis-level data to the

given file. Chooses the appropriate class for handling the given file type.

Parameters

filename [str] File name to open in the given write mode.

mode [str, optional] Mode with which to open the file.

****kwargs** Keyword arguments passed on to the appropriate file handler.

See also:

[`pyrex.io.HDF5Reader`](#) Class for reading data from an hdf5 file.

[`pyrex.io.HDF5Writer`](#) Class for writing data to an hdf5 file.

Attributes

readers [dict] Dictionary with file extensions as keys and values with the corresponding classes used to handle reading of those file types.

writers [dict] Dictionary with file extensions as keys and values with the corresponding classes used to handle writing of those file types.

`pyrex.io.HDF5Reader`

class `pyrex.io.HDF5Reader` (*filename, slice_range=10*)

Class for reading data from an hdf5 file.

Works as a context manager and allows for reading simulation/real data or analysis-level data from the given file.

Parameters

filename [str] File name to open in read mode.

slice_range [int, optional] Number of events to include in each slice when iterating the file. Increasing this value should result in an improvement in speed, while decreasing this value should result in an improvement in memory consumption.

See also:

[`pyrex.io.EventIterator`](#) Class for iterating over event data from an hdf5 file.

Attributes

filename [str] Name of the file (to be) opened.

is_open Boolean of whether the file is currently open.

antenna_info Antenna data from the file.

file_metadata Metadata from the file's metadata datasets.

Methods

<code>close()</code>	Close the hdf5 file.
<code>get_waveforms([event_id, antenna_id, ...])</code>	Get waveform data from the file.
<code>open()</code>	Open the hdf5 file for reading.

pyrex.io.HDF5Writer

```
class pyrex.io.HDF5Writer(filename, mode='x', write_particles=True, write_triggers=True,
                           write_antenna_triggers=False, write_rays=True, write_noise=False,
                           write_waveforms=False, require_trigger=True)
```

Class for writing data to an hdf5 file.

Works as a context manager and allows for writing simulation/real data or analysis-level data to the given file.

Parameters

filename [str] File name to open in the given write mode.

mode [str, optional] Mode with which to open the file. 'w' writes to the file, overwriting any existing data. 'x' writes to the file, failing if the file exists. 'a' appends to the file, creating if the file doesn't exist. 'r+' appends to the file, failing if the file doesn't exist.

write_particles [bool, optional] Whether to write particle metadata when adding event data to the file.

write_waveforms [bool, optional] Whether to write waveform data when adding event data to the file.

write_triggers [bool, optional] Whether to write trigger data when adding event data to the file.

require_trigger [bool or list of str, optional] Whether to write data only when the detector is triggered. If `False` all data will be written for every event. If `True` most data will be written only on a detector trigger (particle metadata and trigger data will still be written for every event). If a list of strings is provided, then the specified data types will be written only on a detector trigger and all other data types will be written for every event.

Other Parameters

write_antenna_triggers [bool, optional] Whether to write trigger data of individual antennas when adding event data to the file.

write_rays [bool, optional] Whether to write ray metadata when adding event data to the file.

write_noise [bool, optional] Whether to write noise-generation metadata when adding event data to the file.

Attributes

filename [str] Name of the file (to be) opened.

is_open Boolean of whether the file is currently open.

has_detector Boolean of whether the file has a linked detector.

Methods

<code>add(event[, triggered, ray_paths, ...])</code>	Add the data from an event to the file.
<code>add_analysis_indices(name, global_index, ...)</code>	Write the given start and length indices to the event indices dataset.
<code>add_analysis_metadata(name, metadata[, index])</code>	Writes the given <i>metadata</i> to the analysis metadata group at <i>name</i> .
<code>add_file_metadata(metadata)</code>	Writes the given <i>metadata</i> to the general file metadata group.
<code>close()</code>	Close the hdf5 file.

Continued on next page

Table 66 – continued from previous page

<code>create_analysis_dataset(name, *args, **kwargs)</code>	Create the given analysis dataset in the file.
<code>create_analysis_group(name, *args, **kwargs)</code>	Create the given analysis group in the file.
<code>create_analysis_metadataset(name, *args, ...)</code>	Create the given analysis metadata group in the file.
<code>open()</code>	Open the hdf5 file for writing.
<code>set_detector(detector)</code>	Link the given <i>detector</i> to the file.

pyrex.io.EventIterator

class `pyrex.io.EventIterator` (*hdf5_file*, *slice_range=10*, *start_event=None*, *stop_event=None*, *step=None*)

Class for iterating over event data from an hdf5 file.

Stores data for a chunk of events according to the *slice_range*. Increasing the *slice_range* should result in an improvement in speed, while decreasing it should result in an improvement in memory consumption.

Parameters

hdf5_file Open file object to be read from.

slice_range [int, optional] Number of events to include in each slice when iterating the file.

start_event [int, optional] Event index of the file from which to start iteration.

stop_event [int, optional] Event index of the file at which to stop iteration (this event will not be included in the iteration).

step [int, optional] Step size of the iterator. Must be greater than zero.

Attributes

triggered Whether or not the event triggered the detector.

noise_bases Noise bases for each antenna of the event for reproducing noise.

is_neutrino Whether the event's initial particle is a neutrino.

flavor Flavor of the event's initial particle.

is_nubar Whether the event's initial particle is an anti-neutrino.

Methods

<code>get_particle_info([attribute])</code>	Get particle data for the event.
<code>get_rays_info([attribute])</code>	Get ray path data for the event.
<code>get_triggered_components([ray])</code>	Get the components of the detector triggered by the event.
<code>get_waveforms([antenna_id, waveform_type])</code>	Get waveform data for the event.

6.3 Included Custom Sub-Packages

6.3.1 Askaryan Radio Array (`pyrex.custom.ara`)

The ARA module contains classes for antennas and detectors as found or proposed for the ARA project.

The *HpolAntenna* and *VpolAntenna* classes are models of ARA Hpol and Vpol antennas using data lifted from AraSim. They use the antenna directional gains in `data/ARA_dipoletest1_output_MY_fixed.txt` and `data/ARA_bicone6in_output_MY_fixed.txt` respectively, and the electronics gains in `data/ARA_Electronics_TotalGain_TwoFilters.txt`. The trigger condition of these antennas is based on a comparison of the maximum value of the tunnel-diode-convolved waveforms with the rms value of a tunnel-diode-convolved noise waveform.

The *ARAStrng* class creates a string of alternating *HpolAntenna* and *VpolAntenna* objects, as in a typical ARA station. The *PhasedArrayString* class implements a more densely-packed string of antennas which trigger based on a threshold trigger on the best beam-formed combination of the antenna waveforms. The *RegularStation* class creates a station at the given position with 4 (or another given number) strings spaced evenly around the station center. The *AlbrechtStation* class (proposed by Albrecht Karle) creates two phased array strings at the station center, one of *VpolAntenna* objects and the other of *HpolAntenna* objects, as well as 3 (or another given number) outrigger strings spaced evenly around the station center. The *HexagonalGrid* class creates a hexagonal grid of stations, spiralling outward from the center.

Default Package Imports

<i>HpolAntenna</i>	ARA Hpol (“quad-slot”) antenna system with front-end processing.
<i>VpolAntenna</i>	ARA Vpol (“bicone” or “birdcage”) antenna system with front-end processing.
<i>ARAStrng</i>	String of ARA Hpol and Vpol antennas.
<i>PhasedArrayString</i>	Phased string of closely-packed antennas.
<i>RegularStation</i>	Station geometry with strings evenly spaced radially around the center.
<i>AlbrechtStation</i>	Station geometry with center phased string and some outrigger strings.
<i>HexagonalGrid</i>	Hexagonal grid of stations or strings.

pyrex.custom.ara.HpolAntenna

```
class pyrex.custom.ara.HpolAntenna (name,      position,      power_threshold,      amplifica-
                                     tion=1,      amplifier_clipping=1,      noisy=True,
                                     unique_noise_waveforms=10)
```

ARA Hpol (“quad-slot”) antenna system with front-end processing.

Applies as the front end a filter representing the full ARA electronics chain (including amplification) and signal clipping. Additionally provides a method for passing a signal through the tunnel diode.

Parameters

name [str] Name of the antenna.

position [array_like] Vector position of the antenna.

power_threshold [float] Power threshold for trigger condition. Antenna triggers if a signal passed through the tunnel diode exceeds this threshold times the noise RMS of the tunnel diode.

amplification [float, optional] Amplification to be applied to the signal pre-clipping. Note that the usual ARA electronics amplification is already applied without this.

amplifier_clipping [float, optional] Voltage (V) above which the amplified signal is clipped (in positive and negative values).

noisy [boolean, optional] Whether or not the antenna should add noise to incoming signals.

unique_noise_waveforms [int, optional] The number of expected noise waveforms needed for each received signal to have its own noise.

See also:

ARAAntennaSystem Antenna system extending base ARA antenna with front-end processing.

Attributes

antenna [Antenna] Antenna object extended by the front end.

name [str] Name of the antenna.

position [array_like] Vector position of the antenna.

power_threshold [float] Power threshold for trigger condition. Antenna triggers if a signal passed through the tunnel diode exceeds this threshold times the noise RMS of the tunnel diode.

amplification [float] Amplification to be applied to the signal pre-clipping. Note that the usual ARA electronics amplification is already applied without this.

amplifier_clipping [float] Voltage (V) above which the amplified signal is clipped (in positive and negative values).

is_hit Boolean of whether the antenna system has been triggered.

is_hit_mc_truth Boolean of whether the antenna has been triggered by signal.

signals The signals received by the antenna with front-end processing.

waveforms The antenna system signal + noise for each triggered hit.

all_waveforms The antenna system signal + noise for all hits.

Methods

<code>apply_response(signal[, direction, ...])</code>	Process the complete antenna response for an incoming signal.
<code>clear([reset_noise])</code>	Reset the antenna system to an empty state.
<code>front_end(signal)</code>	Apply front-end processes to a signal and return the output.
<code>full_waveform(times)</code>	Signal + noise for the antenna system for the given times.
<code>interpolate_filter(frequencies)</code>	Generate interpolated filter values for given frequencies.
<code>is_hit_during(times)</code>	Check if the antenna system is triggered in a time range.
<code>make_noise(times)</code>	Creates a noise signal over the given times.
<code>receive(signal[, direction, polarization, ...])</code>	Process and store one or more incoming (polarized) signals.
<code>set_orientation([z_axis, x_axis])</code>	Sets the orientation of the antenna system.
<code>setup_antenna(response_data[, ...])</code>	Setup the antenna by passing along its init arguments.
<code>trigger(signal)</code>	Check if the antenna system triggers on a given signal.

Continued on next page

Table 69 – continued from previous page

tunnel_diode(signal)	Calculate a signal as processed by the tunnel diode.
----------------------	--

pyrex.custom.ara.VpolAntenna

```
class pyrex.custom.ara.VpolAntenna (name, position, power_threshold, amplifica-
                                tion=1, amplifier_clipping=1, noisy=True,
                                unique_noise_waveforms=10)
```

ARA Vpol (“bicone” or “birdcage”) antenna system with front-end processing.

Applies as the front end a filter representing the full ARA electronics chain (including amplification) and signal clipping. Additionally provides a method for passing a signal through the tunnel diode.

Parameters

name [str] Name of the antenna.

position [array_like] Vector position of the antenna.

power_threshold [float] Power threshold for trigger condition. Antenna triggers if a signal passed through the tunnel diode exceeds this threshold times the noise RMS of the tunnel diode.

amplification [float, optional] Amplification to be applied to the signal pre-clipping. Note that the usual ARA electronics amplification is already applied without this.

amplifier_clipping [float, optional] Voltage (V) above which the amplified signal is clipped (in positive and negative values).

noisy [boolean, optional] Whether or not the antenna should add noise to incoming signals.

unique_noise_waveforms [int, optional] The number of expected noise waveforms needed for each received signal to have its own noise.

See also:

ARAAntennaSystem Antenna system extending base ARA antenna with front-end processing.

Attributes

antenna [Antenna] Antenna object extended by the front end.

name [str] Name of the antenna.

position [array_like] Vector position of the antenna.

power_threshold [float] Power threshold for trigger condition. Antenna triggers if a signal passed through the tunnel diode exceeds this threshold times the noise RMS of the tunnel diode.

amplification [float] Amplification to be applied to the signal pre-clipping. Note that the usual ARA electronics amplification is already applied without this.

amplifier_clipping [float] Voltage (V) above which the amplified signal is clipped (in positive and negative values).

is_hit Boolean of whether the antenna system has been triggered.

is_hit_mc_truth Boolean of whether the antenna has been triggered by signal.

signals The signals received by the antenna with front-end processing.

waveforms The antenna system signal + noise for each triggered hit.

all_waveforms The antenna system signal + noise for all hits.

Methods

<code>apply_response(signal[, direction, ...])</code>	Process the complete antenna response for an incoming signal.
<code>clear([reset_noise])</code>	Reset the antenna system to an empty state.
<code>front_end(signal)</code>	Apply front-end processes to a signal and return the output.
<code>full_waveform(times)</code>	Signal + noise for the antenna system for the given times.
<code>interpolate_filter(frequencies)</code>	Generate interpolated filter values for given frequencies.
<code>is_hit_during(times)</code>	Check if the antenna system is triggered in a time range.
<code>make_noise(times)</code>	Creates a noise signal over the given times.
<code>receive(signal[, direction, polarization, ...])</code>	Process and store one or more incoming (polarized) signals.
<code>set_orientation([z_axis, x_axis])</code>	Sets the orientation of the antenna system.
<code>setup_antenna(response_data[, ...])</code>	Setup the antenna by passing along its init arguments.
<code>trigger(signal)</code>	Check if the antenna system triggers on a given signal.
<code>tunnel_diode(signal)</code>	Calculate a signal as processed by the tunnel diode.

pyrex.custom.ara.ARAString

```
class pyrex.custom.ara.ARAString(x, y, antennas_per_string=4, antenna_separation=10,
                                lowest_antenna=-200)
```

String of ARA Hpol and Vpol antennas.

Sets the positions of antennas on string based on the parameters. Once the antennas have been built with `build_antennas`, the object can be directly iterated over to iterate over the antennas (as if the object were just a list of the antennas).

Parameters

x [float] Cartesian x-position (m) of the string.

y [float] Cartesian y-position (m) of the string.

antennas_per_string [float, optional] Total number of antennas to be placed on the string.

antenna_separation [float or list of float, optional] The vertical separation (m) of antennas on the string. If `float`, all antennas are separated by the same constant value. If `list`, the separations in the list are the separations of neighboring antennas starting from the lowest up to the highest.

lowest_antenna [float, optional] The Cartesian z-position (m) of the lowest antenna on the string.

Raises

ValueError If `test_antenna_positions` is `True` and an antenna is found to be above the ice surface.

See also:

`pyrex.custom.ara.HpolAntenna` ARA Hpol (“quad-slot”) antenna system with front-end processing.

`pyrex.custom.ara.VpolAntenna` ARA Vpol (“bicone” or “birdcage”) antenna system with front-end processing.

Notes

This class is designed to be the lowest subset level of a detector. It can (and should) be used for the subsets of some other `Detector` subclass to build up a full detector. Then when its “parent” is iterated, the instances of this class will be iterated as though they were all part of one flat list.

Attributes

antenna_positions [list] List (potentially with sub-lists) of the positions of the antennas generated by the `set_positions` method.

subsets [list] List of the antenna or detector objects which make up the detector.

test_antenna_positions [boolean] Class attribute for whether or not an error should be raised if antenna positions are found above the surface of the ice (where simulation behavior is ill-defined). Defaults to `True`.

Methods

<code>build_antennas(power_threshold[, ...])</code>	Creates antenna objects at the set antenna positions.
<code>clear([reset_noise])</code>	Reset the detector to an empty state.
<code>set_positions(x, y[, antennas_per_string, ...])</code>	Generates antenna positions along the string.
<code>triggered([antenna_requirement, ...])</code>	Check if the string is triggered based on its current state.

`pyrex.custom.ara.PhasedArrayString`

```
class pyrex.custom.ara.PhasedArrayString(x,      y,      antennas_per_string=10,  an-
                                     tenna_separation=1,      lowest_antenna=-
                                     100,      antenna_type=<class
                                     'pyrex.custom.ara.antenna.VpolAntenna'>)
```

Phased string of closely-packed antennas.

Sets the positions of antennas on string based on the parameters. Once the antennas have been built with `build_antennas`, the object can be directly iterated over to iterate over the antennas (as if the object were just a list of the antennas).

Parameters

x [float] Cartesian x-position (m) of the string.

y [float] Cartesian y-position (m) of the string.

antennas_per_string [float, optional] Total number of antennas to be placed on the string.

antenna_separation [float or list of float, optional] The vertical separation (m) of antennas on the string. If `float`, all antennas are separated by the same constant value. If `list`, the separations in the list are the separations of neighboring antennas starting from the lowest up to the highest.

lowest_antenna [float, optional] The Cartesian z-position (m) of the lowest antenna on the string.

antenna_type [optional] The class to be used to create the antenna objects.

Raises

ValueError If `test_antenna_positions` is `True` and an antenna is found to be above the ice surface.

See also:

`pyrex.custom.ara.HpolAntenna` ARA Hpol (“quad-slot”) antenna system with front-end processing.

`pyrex.custom.ara.VpolAntenna` ARA Vpol (“bicone” or “birdcage”) antenna system with front-end processing.

Notes

This class is designed to be the lowest subset level of a detector. It can (and should) be used for the subsets of some other `Detector` subclass to build up a full detector. Then when its “parent” is iterated, the instances of this class will be iterated as though they were all part of one flat list.

Attributes

antenna_type The class to be used to create the antenna objects.

antenna_positions [list] List (potentially with sub-lists) of the positions of the antennas generated by the `set_positions` method.

subsets [list] List of the antenna or detector objects which make up the detector.

test_antenna_positions [boolean] Class attribute for whether or not an error should be raised if antenna positions are found above the surface of the ice (where simulation behavior is ill-defined). Defaults to `True`.

Methods

<code>build_antennas(power_threshold[, ...])</code>	Creates antenna objects at the set antenna positions.
<code>clear([reset_noise])</code>	Reset the detector to an empty state.
<code>set_positions(x, y[, antennas_per_string, ...])</code>	Generates antenna positions along the string.
<code>triggered(beam_threshold[, delays, angles, ...])</code>	Check if the string is triggered based on its current state.

pyrex.custom.ara.RegularStation

```
class pyrex.custom.ara.RegularStation(x, y, strings_per_station=4, station_diameter=20, string_type=<class 'pyrex.custom.ara.detector.ARAStrng'>,
**string_kwargs)
```

Station geometry with strings evenly spaced radially around the center.

Sets the positions of strings around the station based on the parameters. Supports any string type and passes extra keyword arguments on to the string class. Once the antennas have been built with `build_antennas`, the object can be directly iterated over to iterate over the antennas (as if the object were just a list of the antennas).

Parameters

x [float] Cartesian x-position (m) of the station.

y [float] Cartesian y-position (m) of the station.

strings_per_station [float, optional] Number of strings to be placed evenly around the station.

station_diameter [float, optional] Diameter (m) of the circle around which strings are placed.

string_type [optional] Class to be used for creating string objects for *subsets*.

****string_kwargs** Keyword arguments to be passed on to the `__init__` methods of the *string_type* class.

Raises

ValueError If `test_antenna_positions` is `True` and an antenna is found to be above the ice surface.

See also:

`pyrex.custom.ara.HpolAntenna` ARA Hpol (“quad-slot”) antenna system with front-end processing.

`pyrex.custom.ara.VpolAntenna` ARA Vpol (“bicone” or “birdcage”) antenna system with front-end processing.

`ARAStrng` String of ARA Hpol and Vpol antennas.

Notes

This class is designed to have string-like objects (which are subclasses of `Detector`) as its *subsets*. Then whenever an object of this class is iterated, all the antennas of its strings will be yielded as in a 1D list.

Attributes

antenna_positions [list] List (potentially with sub-lists) of the positions of the antennas generated by the `set_positions` method.

subsets [list] List of the antenna or detector objects which make up the detector.

test_antenna_positions [boolean] Class attribute for whether or not an error should be raised if antenna positions are found above the surface of the ice (where simulation behavior is ill-defined). Defaults to `True`.

Methods

<code>build_antennas(*args, **kwargs)</code>	Creates antenna objects at the set antenna positions.
<code>clear([reset_noise])</code>	Reset the detector to an empty state.
<code>set_positions(x, y[, strings_per_station, ...])</code>	Generates antenna positions around the station.
<code>triggered([polarized_antenna_requirement, ...])</code>	Check if the station is triggered based on its current state.

pyrex.custom.ara.AlbrechtStation

```
class pyrex.custom.ara.AlbrechtStation(x, y, station_diameter=40,
                                     hpol_phased_antennas=10,
                                     vpol_phased_antennas=10,
                                     hpol_phased_separation=1,
                                     vpol_phased_separation=1, hpol_phased_lowest=-49,
                                     vpol_phased_lowest=-69, outrigger_strings_per_station=3,
                                     outrigger_string_type=<class
                                     'pyrex.custom.ara.detector.ARAString'>, **outrigger_string_kwargs)
```

Station geometry with center phased string and some outrigger strings.

Station geometry proposed by Albrecht with a phased array string of each polarization at the station center, plus a number of outrigger strings evenly spaced radially around the station center. Sets the positions of strings around the station based on the parameters. Supports any string type and passes extra keyword arguments on to the string class. Once the antennas have been built with `build_antennas`, the object can be directly iterated over to iterate over the antennas (as if the object were just a list of the antennas).

Parameters

x [float] Cartesian x-position (m) of the station.

y [float] Cartesian y-position (m) of the station.

station_diameter [float, optional] Diameter (m) of the circle around which outrigger strings are placed.

hpol_phased_antennas [float, optional] Number of Hpol phased antennas for the center string.

vpol_phased_antennas [float, optional] Number of Vpol phased antennas for the center string.

hpol_phased_separation [float or list of float, optional] Antenna separation (m) for the phased Hpol antennas.

vpol_phased_separation [float or list of float, optional] Antenna separation (m) for the phased Vpol antennas.

hpol_phased_lowest [float, optional] Cartesian z-position (m) of the lowest phased Hpol antenna.

vpol_phased_lowest [float, optional] Cartesian z-position (m) of the lowest phased Vpol antenna.

outrigger_strings_per_station [float, optional] Number of outrigger strings to be placed evenly around the station.

outrigger_string_type [optional] Class to be used for creating outrigger string objects for *subsets*.

****outrigger_string_kwargs** Keyword arguments to be passed on to the `__init__` methods of the *outrigger_string_type* class. The default values for `antennas_per_string`, `antenna_separation`, and `lowest_antenna` are altered for this station geometry.

Raises

ValueError If `test_antenna_positions` is `True` and an antenna is found to be above the ice surface.

See also:

`pyrex.custom.ara.HpolAntenna` ARA Hpol (“quad-slot”) antenna system with front-end processing.

`pyrex.custom.ara.VpolAntenna` ARA Vpol (“bicone” or “birdcage”) antenna system with front-end processing.

`ARAStrng` String of ARA Hpol and Vpol antennas.

`PhasedArrayString` Phased string of closely-packed antennas.

Notes

This class is designed to have string-like objects (which are subclasses of `Detector`) as its *subsets*. Then whenever an object of this class is iterated, all the antennas of its strings will be yielded as in a 1D list.

Attributes

`antenna_positions` [list] List (potentially with sub-lists) of the positions of the antennas generated by the `set_positions` method.

`subsets` [list] List of the antenna or detector objects which make up the detector.

`test_antenna_positions` [boolean] Class attribute for whether or not an error should be raised if antenna positions are found above the surface of the ice (where simulation behavior is ill-defined). Defaults to `True`.

Methods

<code>build_antennas(*args, **kwargs)</code>	Creates antenna objects at the set antenna positions.
<code>clear([reset_noise])</code>	Reset the detector to an empty state.
<code>set_positions(x, y[, station_diameter, ...])</code>	Generates antenna positions around the station.
<code>triggered(beam_threshold[, ...])</code>	Check if the station is triggered based on its current state.

`pyrex.custom.ara.HexagonalGrid`

```
class pyrex.custom.ara.HexagonalGrid (stations=1,      station_separation=2000,      sta-
                                     tion_type=<class 'pyrex.custom.ara.detector.RegularStation'>,
                                     **station_kwargs)
```

Hexagonal grid of stations or strings.

Sets the positions of stations by spiralling outward in a hexagonal grid. Supports any station type (including string types) and passes extra keyword arguments on to the station class. Once the antennas have been built with `build_antennas`, the object can be directly iterated over to iterate over the antennas (as if the object were just a list of the antennas).

Parameters

`stations` [float, optional] Number of stations to be placed.

`station_separation` [float, optional] Distance (m) between adjacent stations.

`station_type` [optional] Class to be used for creating station objects for *subsets*.

`station_kwargs`** Keyword arguments to be passed on to the `__init__` methods of the *station_type* class.

Raises

ValueError If `test_antenna_positions` is `True` and an antenna is found to be above the ice surface.

See also:

`pyrex.custom.ara.HpolAntenna` ARA Hpol (“quad-slot”) antenna system with front-end processing.

`pyrex.custom.ara.VpolAntenna` ARA Vpol (“bicone” or “birdcage”) antenna system with front-end processing.

`ARAStrng` String of ARA Hpol and Vpol antennas.

`RegularStation` Station geometry with strings evenly spaced radially around the center.

Notes

This class is designed to have station-like or string-like objects (which are subclasses of `Detector`) as its *subsets*. Then whenever an object of this class is iterated, all the antennas of its strings will be yielded as in a 1D list.

Attributes

`antenna_positions` [list] List (potentially with sub-lists) of the positions of the antennas generated by the `set_positions` method.

`subsets` [list] List of the antenna or detector objects which make up the detector.

`test_antenna_positions` [boolean] Class attribute for whether or not an error should be raised if antenna positions are found above the surface of the ice (where simulation behavior is ill-defined). Defaults to `True`.

Methods

<code>build_antennas(*args, **kwargs)</code>	Creates antenna objects at the set antenna positions.
<code>clear([reset_noise])</code>	Reset the detector to an empty state.
<code>set_positions([stations, ...])</code>	Generates antenna positions around the station.
<code>triggered([station_requirement, ...])</code>	Check if the detector is triggered based on its current state.

Individual Module APIs

Custom Antennas (`pyrex.custom.ara.antenna`)

Module containing customized antenna classes for ARA.

Many of the methods here mirror methods used in the antennas in AraSim, to ensure that AraSim results can be matched.

<code>_read_arasim_antenna_data</code>	Gather antenna directionality data from an AraSim-formatted data file.
<code>_read_filter_data</code>	Gather frequency-dependent filtering data from a data file.
<code>ARAAntenna</code>	Antenna class to be used for ARA antennas.

Continued on next page

Table 76 – continued from previous page

<i>ARAAntennaSystem</i>	Antenna system extending base ARA antenna with front-end processing.
<i>HpolAntenna</i>	ARA Hpol (“quad-slot”) antenna system with front-end processing.
<i>VpolAntenna</i>	ARA Vpol (“bicone” or “birdcage”) antenna system with front-end processing.

pyrex.custom.ara.antenna._read_arasim_antenna_data

`pyrex.custom.ara.antenna._read_arasim_antenna_data(filename)`

Gather antenna directionality data from an AraSim-formatted data file.

The data file should have columns for theta, phi, dB gain, non-dB gain, and phase (in degrees). This should be divided into sections for each frequency with a header line “freq : X MHz”, optionally followed by a second line “SWR : Y”.

Parameters

filename [str] Name of the data file.

Returns

response [ndarray] 3-D array of complex-valued voltage gains as a function of frequency along axis 0, zenith along axis 1, and azimuth along axis 2.

frequencies [ndarray] Frequencies (Hz) corresponding to axis 0 of *response*.

thetas [ndarray] Zenith angles (degrees) corresponding to axis 1 of *response*.

phis [ndarray] Azimuth angles (degrees) corresponding to axis 2 of *response*.

pyrex.custom.ara.antenna._read_filter_data

`pyrex.custom.ara.antenna._read_filter_data(filename)`

Gather frequency-dependent filtering data from a data file.

The data file should have columns for frequency, non-dB gain, and phase (in radians).

Parameters

filename [str] Name of the data file.

Returns

gains [ndarray] Complex-valued voltage gains as a function of frequency.

frequencies [ndarray] Frequencies (Hz) corresponding to the values of *gains*.

pyrex.custom.ara.antenna.ARAAntenna

```
class pyrex.custom.ara.antenna.ARAAntenna(response_data, position, center_frequency,
                                         bandwidth, resistance, orientation=(0,
                                         0, 1), efficiency=1, noisy=True,
                                         unique_noise_waveforms=10)
```

Antenna class to be used for ARA antennas.

Stores the attributes of an antenna as well as handling receiving, processing, and storing signals and adding noise. Antenna response based on provided models.

Parameters

- response_data** [tuple of array_like] Tuple containing the response data for the antenna along the theta and phi polarization directions. The first and second elements should contain 3-D arrays of the antenna response model in the theta and phi polarizations, respectively, as a function of frequency (axis 0), zenith (axis 1), and azimuth (axis 2). The remaining elements should be the values of the frequency, zenith, and azimuth axes, respectively.
- position** [array_like] Vector position of the antenna.
- center_frequency** [float] Frequency (Hz) at the center of the antenna's frequency range.
- bandwidth** [float] Bandwidth (Hz) of the antenna.
- resistance** [float] The noise resistance (ohm) of the antenna. Used to calculate the RMS voltage of the antenna noise.
- orientation** [array_like, optional] Vector direction of the z-axis of the antenna.
- efficiency** [float, optional] Antenna efficiency applied to incoming signal values.
- noisy** [boolean, optional] Whether or not the antenna should add noise to incoming signals.
- unique_noise_waveforms** [int, optional] The number of expected noise waveforms needed for each received signal to have its own noise.

See also:

[*pyrex.Antenna*](#) Base class for antennas.

Attributes

- position** [array_like] Vector position of the antenna.
- z_axis** [ndarray] Vector direction of the z-axis of the antenna.
- x_axis** [ndarray] Vector direction of the x-axis of the antenna.
- antenna_factor** [float] Antenna factor used for converting fields to voltages.
- efficiency** [float] Antenna efficiency applied to incoming signal values.
- noisy** [boolean] Whether or not the antenna should add noise to incoming signals.
- unique_noises** [int] The number of expected noise waveforms needed for each received signal to have its own noise.
- freq_range** [array_like] The frequency band in which the antenna operates (used for noise production).
- temperature** [float or None] The noise temperature (K) of the antenna. Used in combination with *resistance* to calculate the RMS voltage of the antenna noise.
- resistance** [float or None] The noise resistance (ohm) of the antenna. Used in combination with *temperature* to calculate the RMS voltage of the antenna noise.
- noise_rms** [float or None] The RMS voltage (v) of the antenna noise. If not *None*, this value will be used instead of the RMS voltage calculated from the values of *temperature* and *resistance*.
- signals** [list of Signal] The signals which have been received by the antenna.
- is_hit** Boolean of whether the antenna has been triggered.

is_hit_mc_truth Boolean of whether the antenna has been triggered by signal.

waveforms Signal + noise (if `noisy`) for each triggered antenna hit.

all_waveforms Signal + noise (if `noisy`) for all antenna hits.

Methods

<code>apply_response(signal[, direction, ...])</code>	Process the complete antenna response for an incoming signal.
<code>clear([reset_noise])</code>	Reset the antenna to an empty state.
<code>directional_gain(theta, phi)</code>	Calculate the (complex) directional gain of the antenna.
<code>directional_response(theta, phi, polarization)</code>	Generate the (complex) frequency-dependent directional response.
<code>frequency_response(frequencies)</code>	Calculate the (complex) frequency response of the antenna.
<code>full_waveform(times)</code>	Signal + noise (if <code>noisy</code>) for the given times.
<code>is_hit_during(times)</code>	Check if the antenna is triggered in a time range.
<code>make_noise(times)</code>	Creates a noise signal over the given times.
<code>polarization_gain(polarization)</code>	Calculate the (complex) polarization gain of the antenna.
<code>receive(signal[, direction, polarization, ...])</code>	Process and store one or more incoming (polarized) signals.
<code>set_orientation([z_axis, x_axis])</code>	Sets the orientation of the antenna.
<code>trigger(signal)</code>	Check if the antenna triggers on a given signal.

pyrex.custom.ara.antenna.ARAAntennaSystem

```
class pyrex.custom.ara.antenna.ARAAntennaSystem(response_data, name, position,
                                                power_threshold, orientation=(0,
                                                0, 1), amplification=1, ampli-
                                                fier_clipping=1, noisy=True,
                                                unique_noise_waveforms=10,
                                                **kwargs)
```

Antenna system extending base ARA antenna with front-end processing.

Applies as the front end a filter representing the full ARA electronics chain (including amplification) and signal clipping. Additionally provides a method for passing a signal through the tunnel diode.

Parameters

response_data [tuple of array_like] Tuple containing the response data for the antenna along the theta and phi polarization directions. The first and second elements should contain 3-D arrays of the antenna response model in the theta and phi polarizations, respectively, as a function of frequency (axis 0), zenith (axis 1), and azimuth (axis 2). The remaining elements should be the values of the frequency, zenith, and azimuth axes, respectively.

name [str] Name of the antenna.

position [array_like] Vector position of the antenna.

power_threshold [float] Power threshold for trigger condition. Antenna triggers if a signal passed through the tunnel diode exceeds this threshold times the noise RMS of the tunnel diode.

- orientation** [array_like, optional] Vector direction of the z-axis of the antenna.
- amplification** [float, optional] Amplification to be applied to the signal pre-clipping. Note that the usual ARA electronics amplification is already applied without this.
- amplifier_clipping** [float, optional] Voltage (V) above which the amplified signal is clipped (in positive and negative values).
- noisy** [boolean, optional] Whether or not the antenna should add noise to incoming signals.
- unique_noise_waveforms** [int, optional] The number of expected noise waveforms needed for each received signal to have its own noise.

See also:

pyrex.AntennaSystem Base class for antenna system with front-end processing.

ARAAntenna Antenna class to be used for ARA antennas.

Attributes

- antenna** [Antenna] *Antenna* object extended by the front end.
- name** [str] Name of the antenna.
- position** [array_like] Vector position of the antenna.
- power_threshold** [float] Power threshold for trigger condition. Antenna triggers if a signal passed through the tunnel diode exceeds this threshold times the noise RMS of the tunnel diode.
- amplification** [float] Amplification to be applied to the signal pre-clipping. Note that the usual ARA electronics amplification is already applied without this.
- amplifier_clipping** [float] Voltage (V) above which the amplified signal is clipped (in positive and negative values).
- lead_in_time** [float] Lead-in time (s) required for the front end to equilibrate. Automatically added in before calculation of signals and waveforms.
- is_hit** Boolean of whether the antenna system has been triggered.
- is_hit_mc_truth** Boolean of whether the antenna has been triggered by signal.
- signals** The signals received by the antenna with front-end processing.
- waveforms** The antenna system signal + noise for each triggered hit.
- all_waveforms** The antenna system signal + noise for all hits.

Methods

<code>apply_response(signal[, direction, ...])</code>	Process the complete antenna response for an incoming signal.
<code>clear([reset_noise])</code>	Reset the antenna system to an empty state.
<code>front_end(signal)</code>	Apply front-end processes to a signal and return the output.
<code>full_waveform(times)</code>	Signal + noise for the antenna system for the given times.

Continued on next page

Table 78 – continued from previous page

<code>interpolate_filter(frequencies)</code>	Generate interpolated filter values for given frequencies.
<code>is_hit_during(times)</code>	Check if the antenna system is triggered in a time range.
<code>make_noise(times)</code>	Creates a noise signal over the given times.
<code>receive(signal[, direction, polarization, ...])</code>	Process and store one or more incoming (polarized) signals.
<code>set_orientation([z_axis, x_axis])</code>	Sets the orientation of the antenna system.
<code>setup_antenna(response_data[, ...])</code>	Setup the antenna by passing along its init arguments.
<code>trigger(signal)</code>	Check if the antenna system triggers on a given signal.
<code>tunnel_diode(signal)</code>	Calculate a signal as processed by the tunnel diode.

pyrex.custom.ara.antenna.HpolAntenna

class `pyrex.custom.ara.antenna.HpolAntenna` (*name*, *position*, *power_threshold*, *amplification*=1, *amplifier_clipping*=1, *noisy*=True, *unique_noise_waveforms*=10)

ARA Hpol (“quad-slot”) antenna system with front-end processing.

Applies as the front end a filter representing the full ARA electronics chain (including amplification) and signal clipping. Additionally provides a method for passing a signal through the tunnel diode.

Parameters

name [str] Name of the antenna.

position [array_like] Vector position of the antenna.

power_threshold [float] Power threshold for trigger condition. Antenna triggers if a signal passed through the tunnel diode exceeds this threshold times the noise RMS of the tunnel diode.

amplification [float, optional] Amplification to be applied to the signal pre-clipping. Note that the usual ARA electronics amplification is already applied without this.

amplifier_clipping [float, optional] Voltage (V) above which the amplified signal is clipped (in positive and negative values).

noisy [boolean, optional] Whether or not the antenna should add noise to incoming signals.

unique_noise_waveforms [int, optional] The number of expected noise waveforms needed for each received signal to have its own noise.

See also:

ARAAntennaSystem Antenna system extending base ARA antenna with front-end processing.

Attributes

antenna [Antenna] `Antenna` object extended by the front end.

name [str] Name of the antenna.

position [array_like] Vector position of the antenna.

power_threshold [float] Power threshold for trigger condition. Antenna triggers if a signal passed through the tunnel diode exceeds this threshold times the noise RMS of the tunnel diode.

amplification [float] Amplification to be applied to the signal pre-clipping. Note that the usual ARA electronics amplification is already applied without this.

amplifier_clipping [float] Voltage (V) above which the amplified signal is clipped (in positive and negative values).

is_hit Boolean of whether the antenna system has been triggered.

is_hit_mc_truth Boolean of whether the antenna has been triggered by signal.

signals The signals received by the antenna with front-end processing.

waveforms The antenna system signal + noise for each triggered hit.

all_waveforms The antenna system signal + noise for all hits.

Methods

<code>apply_response(signal[, direction, ...])</code>	Process the complete antenna response for an incoming signal.
<code>clear([reset_noise])</code>	Reset the antenna system to an empty state.
<code>front_end(signal)</code>	Apply front-end processes to a signal and return the output.
<code>full_waveform(times)</code>	Signal + noise for the antenna system for the given times.
<code>interpolate_filter(frequencies)</code>	Generate interpolated filter values for given frequencies.
<code>is_hit_during(times)</code>	Check if the antenna system is triggered in a time range.
<code>make_noise(times)</code>	Creates a noise signal over the given times.
<code>receive(signal[, direction, polarization, ...])</code>	Process and store one or more incoming (polarized) signals.
<code>set_orientation([z_axis, x_axis])</code>	Sets the orientation of the antenna system.
<code>setup_antenna(response_data[, ...])</code>	Setup the antenna by passing along its init arguments.
<code>trigger(signal)</code>	Check if the antenna system triggers on a given signal.
<code>tunnel_diode(signal)</code>	Calculate a signal as processed by the tunnel diode.

pyrex.custom.ara.antenna.VpolAntenna

class `pyrex.custom.ara.antenna.VpolAntenna` (*name*, *position*, *power_threshold*, *amplification*=1, *amplifier_clipping*=1, *noisy*=True, *unique_noise_waveforms*=10)

ARA Vpol (“bicone” or “birdcage”) antenna system with front-end processing.

Applies as the front end a filter representing the full ARA electronics chain (including amplification) and signal clipping. Additionally provides a method for passing a signal through the tunnel diode.

Parameters

name [str] Name of the antenna.

position [array_like] Vector position of the antenna.

power_threshold [float] Power threshold for trigger condition. Antenna triggers if a signal passed through the tunnel diode exceeds this threshold times the noise RMS of the tunnel diode.

amplification [float, optional] Amplification to be applied to the signal pre-clipping. Note that the usual ARA electronics amplification is already applied without this.

amplifier_clipping [float, optional] Voltage (V) above which the amplified signal is clipped (in positive and negative values).

noisy [boolean, optional] Whether or not the antenna should add noise to incoming signals.

unique_noise_waveforms [int, optional] The number of expected noise waveforms needed for each received signal to have its own noise.

See also:

ARAAntennaSystem Antenna system extending base ARA antenna with front-end processing.

Attributes

antenna [Antenna] Antenna object extended by the front end.

name [str] Name of the antenna.

position [array_like] Vector position of the antenna.

power_threshold [float] Power threshold for trigger condition. Antenna triggers if a signal passed through the tunnel diode exceeds this threshold times the noise RMS of the tunnel diode.

amplification [float] Amplification to be applied to the signal pre-clipping. Note that the usual ARA electronics amplification is already applied without this.

amplifier_clipping [float] Voltage (V) above which the amplified signal is clipped (in positive and negative values).

is_hit Boolean of whether the antenna system has been triggered.

is_hit_mc_truth Boolean of whether the antenna has been triggered by signal.

signals The signals received by the antenna with front-end processing.

waveforms The antenna system signal + noise for each triggered hit.

all_waveforms The antenna system signal + noise for all hits.

Methods

<code>apply_response(signal[, direction, ...])</code>	Process the complete antenna response for an incoming signal.
<code>clear([reset_noise])</code>	Reset the antenna system to an empty state.
<code>front_end(signal)</code>	Apply front-end processes to a signal and return the output.
<code>full_waveform(times)</code>	Signal + noise for the antenna system for the given times.
<code>interpolate_filter(frequencies)</code>	Generate interpolated filter values for given frequencies.

Continued on next page

Table 80 – continued from previous page

<code>is_hit_during(times)</code>	Check if the antenna system is triggered in a time range.
<code>make_noise(times)</code>	Creates a noise signal over the given times.
<code>receive(signal[, direction, polarization, ...])</code>	Process and store one or more incoming (polarized) signals.
<code>set_orientation([z_axis, x_axis])</code>	Sets the orientation of the antenna system.
<code>setup_antenna(response_data[, ...])</code>	Setup the antenna by passing along its init arguments.
<code>trigger(signal)</code>	Check if the antenna system triggers on a given signal.
<code>tunnel_diode(signal)</code>	Calculate a signal as processed by the tunnel diode.

Custom Detectors (`pyrex.custom.ara.detector`)

Module containing customized detector geometry classes for ARA.

Designed to be flexible such that stations can be built up from any string types and the detector grid can be made up of stations or strings.

<code>convert_hex_coords</code>	Convert from hexagonal coordinates to Cartesian.
<code>ARAString</code>	String of ARA Hpol and Vpol antennas.
<code>PhasedArrayString</code>	Phased string of closely-packed antennas.
<code>RegularStation</code>	Station geometry with strings evenly spaced radially around the center.
<code>AlbrechtStation</code>	Station geometry with center phased string and some outrigger strings.
<code>HexagonalGrid</code>	Hexagonal grid of stations or strings.

`pyrex.custom.ara.detector.convert_hex_coords`

`pyrex.custom.ara.detector.convert_hex_coords(hex_coords, unit=1)`

Convert from hexagonal coordinates to Cartesian.

Parameters

hex_coords [array_like] Array with two elements representing the hexagonal coordinate.

unit [float, optional] Optional unit used to multiply the Cartesian coordinates.

Returns

x [float] Cartesian x-position with the unit correction.

y [float] Cartesian y-position with the unit correction.

Notes

Hexagonal coordinate system defined along non-perpendicular axes where the first axis is 30 degrees from the Cartesian x-axis and the second axis is parallel to the Cartesian y-axis. The conversion equations are therefore $x=h_0-h_1/2$ and $y=h_1*\sqrt{3}/2$.

pyrex.custom.ara.detector.ARAString

```
class pyrex.custom.ara.detector.ARAString(x, y, antennas_per_string=4, antenna_separation=10, lowest_antenna=-200)
```

String of ARA Hpol and Vpol antennas.

Sets the positions of antennas on string based on the parameters. Once the antennas have been built with `build_antennas`, the object can be directly iterated over to iterate over the antennas (as if the object were just a list of the antennas).

Parameters

x [float] Cartesian x-position (m) of the string.

y [float] Cartesian y-position (m) of the string.

antennas_per_string [float, optional] Total number of antennas to be placed on the string.

antenna_separation [float or list of float, optional] The vertical separation (m) of antennas on the string. If `float`, all antennas are separated by the same constant value. If `list`, the separations in the list are the separations of neighboring antennas starting from the lowest up to the highest.

lowest_antenna [float, optional] The Cartesian z-position (m) of the lowest antenna on the string.

Raises

ValueError If `test_antenna_positions` is `True` and an antenna is found to be above the ice surface.

See also:

`pyrex.custom.ara.HpolAntenna` ARA Hpol (“quad-slot”) antenna system with front-end processing.

`pyrex.custom.ara.VpolAntenna` ARA Vpol (“bicone” or “birdcage”) antenna system with front-end processing.

Notes

This class is designed to be the lowest subset level of a detector. It can (and should) be used for the subsets of some other `Detector` subclass to build up a full detector. Then when its “parent” is iterated, the instances of this class will be iterated as though they were all part of one flat list.

Attributes

antenna_positions [list] List (potentially with sub-lists) of the positions of the antennas generated by the `set_positions` method.

subsets [list] List of the antenna or detector objects which make up the detector.

test_antenna_positions [boolean] Class attribute for whether or not an error should be raised if antenna positions are found above the surface of the ice (where simulation behavior is ill-defined). Defaults to `True`.

Methods

<code>build_antennas(power_threshold[, ...])</code>	Creates antenna objects at the set antenna positions.
---	---

Continued on next page

Table 82 – continued from previous page

<code>clear([reset_noise])</code>	Reset the detector to an empty state.
<code>set_positions(x, y[, antennas_per_string, ...])</code>	Generates antenna positions along the string.
<code>triggered([antenna_requirement, ...])</code>	Check if the string is triggered based on its current state.

pyrex.custom.ara.detector.PhasedArrayString

```
class pyrex.custom.ara.detector.PhasedArrayString(x, y, antennas_per_string=10,
                                                    antenna_separation=1,
                                                    lowest_antenna=-100,
                                                    antenna_type=<class
                                                    'pyrex.custom.ara.antenna.VpolAntenna'>)
```

Phased string of closely-packed antennas.

Sets the positions of antennas on string based on the parameters. Once the antennas have been built with `build_antennas`, the object can be directly iterated over to iterate over the antennas (as if the object were just a list of the antennas).

Parameters

x [float] Cartesian x-position (m) of the string.

y [float] Cartesian y-position (m) of the string.

antennas_per_string [float, optional] Total number of antennas to be placed on the string.

antenna_separation [float or list of float, optional] The vertical separation (m) of antennas on the string. If `float`, all antennas are separated by the same constant value. If `list`, the separations in the list are the separations of neighboring antennas starting from the lowest up to the highest.

lowest_antenna [float, optional] The Cartesian z-position (m) of the lowest antenna on the string.

antenna_type [optional] The class to be used to create the antenna objects.

Raises

ValueError If `test_antenna_positions` is `True` and an antenna is found to be above the ice surface.

See also:

[`pyrex.custom.ara.HpolAntenna`](#) ARA Hpol (“quad-slot”) antenna system with front-end processing.

[`pyrex.custom.ara.VpolAntenna`](#) ARA Vpol (“bicone” or “birdcage”) antenna system with front-end processing.

Notes

This class is designed to be the lowest subset level of a detector. It can (and should) be used for the subsets of some other `Detector` subclass to build up a full detector. Then when its “parent” is iterated, the instances of this class will be iterated as though they were all part of one flat list.

Attributes

antenna_type The class to be used to create the antenna objects.

antenna_positions [list] List (potentially with sub-lists) of the positions of the antennas generated by the `set_positions` method.

subsets [list] List of the antenna or detector objects which make up the detector.

test_antenna_positions [boolean] Class attribute for whether or not an error should be raised if antenna positions are found above the surface of the ice (where simulation behavior is ill-defined). Defaults to `True`.

Methods

<code>build_antennas(power_threshold[, ...])</code>	Creates antenna objects at the set antenna positions.
<code>clear([reset_noise])</code>	Reset the detector to an empty state.
<code>set_positions(x, y[, antennas_per_string, ...])</code>	Generates antenna positions along the string.
<code>triggered(beam_threshold[, delays, angles, ...])</code>	Check if the string is triggered based on its current state.

`pyrex.custom.ara.detector.RegularStation`

```
class pyrex.custom.ara.detector.RegularStation(x, y, strings_per_station=4, station_diameter=20, string_type=<class 'pyrex.custom.ara.detector.ARAString'>, **string_kwargs)
```

Station geometry with strings evenly spaced radially around the center.

Sets the positions of strings around the station based on the parameters. Supports any string type and passes extra keyword arguments on to the string class. Once the antennas have been built with `build_antennas`, the object can be directly iterated over to iterate over the antennas (as if the object were just a list of the antennas).

Parameters

x [float] Cartesian x-position (m) of the station.

y [float] Cartesian y-position (m) of the station.

strings_per_station [float, optional] Number of strings to be placed evenly around the station.

station_diameter [float, optional] Diameter (m) of the circle around which strings are placed.

string_type [optional] Class to be used for creating string objects for *subsets*.

****string_kwargs** Keyword arguments to be passed on to the `__init__` methods of the *string_type* class.

Raises

ValueError If `test_antenna_positions` is `True` and an antenna is found to be above the ice surface.

See also:

[`pyrex.custom.ara.HpolAntenna`](#) ARA Hpol (“quad-slot”) antenna system with front-end processing.

[`pyrex.custom.ara.VpolAntenna`](#) ARA Vpol (“bicone” or “birdcage”) antenna system with front-end processing.

[`ARAString`](#) String of ARA Hpol and Vpol antennas.

Notes

This class is designed to have string-like objects (which are subclasses of `Detector`) as its *subsets*. Then whenever an object of this class is iterated, all the antennas of its strings will be yielded as in a 1D list.

Attributes

- antenna_positions** [list] List (potentially with sub-lists) of the positions of the antennas generated by the `set_positions` method.
- subsets** [list] List of the antenna or detector objects which make up the detector.
- test_antenna_positions** [boolean] Class attribute for whether or not an error should be raised if antenna positions are found above the surface of the ice (where simulation behavior is ill-defined). Defaults to `True`.

Methods

<code>build_antennas(*args, **kwargs)</code>	Creates antenna objects at the set antenna positions.
<code>clear([reset_noise])</code>	Reset the detector to an empty state.
<code>set_positions(x, y[, strings_per_station, ...])</code>	Generates antenna positions around the station.
<code>triggered([polarized_antenna_requirement, ...])</code>	Check if the station is triggered based on its current state.

pyrex.custom.ara.detector.AlbrechtStation

```
class pyrex.custom.ara.detector.AlbrechtStation(x, y, station_diameter=40,
                                                hpol_phased_antennas=10,
                                                vpol_phased_antennas=10,
                                                hpol_phased_separation=1,
                                                vpol_phased_separation=1,
                                                hpol_phased_lowest=-49,
                                                vpol_phased_lowest=-69,
                                                outrigger_strings_per_station=3,
                                                outrigger_string_type=<class
                                                'pyrex.custom.ara.detector.ARAString'>,
                                                **outrigger_string_kwargs)
```

Station geometry with center phased string and some outrigger strings.

Station geometry proposed by Albrecht with a phased array string of each polarization at the station center, plus a number of outrigger strings evenly spaced radially around the station center. Sets the positions of strings around the station based on the parameters. Supports any string type and passes extra keyword arguments on to the string class. Once the antennas have been built with `build_antennas`, the object can be directly iterated over to iterate over the antennas (as if the object were just a list of the antennas).

Parameters

- x** [float] Cartesian x-position (m) of the station.
- y** [float] Cartesian y-position (m) of the station.
- station_diameter** [float, optional] Diameter (m) of the circle around which outrigger strings are placed.
- hpol_phased_antennas** [float, optional] Number of Hpol phased antennas for the center string.
- vpol_phased_antennas** [float, optional] Number of Vpol phased antennas for the center string.

hpol_phased_separation [float or list of float, optional] Antenna separation (m) for the phased Hpol antennas.

vpol_phased_separation [float or list of float, optional] Antenna separation (m) for the phased Vpol antennas.

hpol_phased_lowest [float, optional] Cartesian z-position (m) of the lowest phased Hpol antenna.

vpol_phased_lowest [float, optional] Cartesian z-position (m) of the lowest phased Vpol antenna.

outrigger_strings_per_station [float, optional] Number of outrigger strings to be placed evenly around the station.

outrigger_string_type [optional] Class to be used for creating outrigger string objects for *subsets*.

****outrigger_string_kwargs** Keyword arguments to be passed on to the `__init__` methods of the *outrigger_string_type* class. The default values for `antennas_per_string`, `antenna_separation`, and `lowest_antenna` are altered for this station geometry.

Raises

ValueError If `test_antenna_positions` is `True` and an antenna is found to be above the ice surface.

See also:

pyrex.custom.ara.HpolAntenna ARA Hpol (“quad-slot”) antenna system with front-end processing.

pyrex.custom.ara.VpolAntenna ARA Vpol (“bicone” or “birdcage”) antenna system with front-end processing.

ARAStrng String of ARA Hpol and Vpol antennas.

PhasedArrayString Phased string of closely-packed antennas.

Notes

This class is designed to have string-like objects (which are subclasses of `Detector`) as its *subsets*. Then whenever an object of this class is iterated, all the antennas of its strings will be yielded as in a 1D list.

Attributes

antenna_positions [list] List (potentially with sub-lists) of the positions of the antennas generated by the `set_positions` method.

subsets [list] List of the antenna or detector objects which make up the detector.

test_antenna_positions [boolean] Class attribute for whether or not an error should be raised if antenna positions are found above the surface of the ice (where simulation behavior is ill-defined). Defaults to `True`.

Methods

<code>build_antennas(*args, **kwargs)</code>	Creates antenna objects at the set antenna positions.
<code>clear([reset_noise])</code>	Reset the detector to an empty state.
<code>set_positions(x, y[, station_diameter, ...])</code>	Generates antenna positions around the station.

Continued on next page

Table 85 – continued from previous page

<code>triggered(beam_threshold[, ...])</code>	Check if the station is triggered based on its current state.
---	---

pyrex.custom.ara.detector.HexagonalGrid

```
class pyrex.custom.ara.detector.HexagonalGrid(stations=1,      station_separation=2000,
                                              station_type=<class
                                              'pyrex.custom.ara.detector.RegularStation'>,
                                              **station_kwargs)
```

Hexagonal grid of stations or strings.

Sets the positions of stations by spiralling outward in a hexagonal grid. Supports any station type (including string types) and passes extra keyword arguments on to the station class. Once the antennas have been built with `build_antennas`, the object can be directly iterated over to iterate over the antennas (as if the object were just a list of the antennas).

Parameters

stations [float, optional] Number of stations to be placed.

station_separation [float, optional] Distance (m) between adjacent stations.

station_type [optional] Class to be used for creating station objects for *subsets*.

****station_kwargs** Keyword arguments to be passed on to the `__init__` methods of the *station_type* class.

Raises

ValueError If `test_antenna_positions` is `True` and an antenna is found to be above the ice surface.

See also:

[*pyrex.custom.ara.HpolAntenna*](#) ARA Hpol (“quad-slot”) antenna system with front-end processing.

[*pyrex.custom.ara.VpolAntenna*](#) ARA Vpol (“bicone” or “birdcage”) antenna system with front-end processing.

[*ARAStrng*](#) String of ARA Hpol and Vpol antennas.

[*RegularStation*](#) Station geometry with strings evenly spaced radially around the center.

Notes

This class is designed to have station-like or string-like objects (which are subclasses of `Detector`) as its *subsets*. Then whenever an object of this class is iterated, all the antennas of its strings will be yielded as in a 1D list.

Attributes

antenna_positions [list] List (potentially with sub-lists) of the positions of the antennas generated by the `set_positions` method.

subsets [list] List of the antenna or detector objects which make up the detector.

test_antenna_positions [boolean] Class attribute for whether or not an error should be raised if antenna positions are found above the surface of the ice (where simulation behavior is ill-defined). Defaults to `True`.

Methods

<code>build_antennas(*args, **kwargs)</code>	Creates antenna objects at the set antenna positions.
<code>clear([reset_noise])</code>	Reset the detector to an empty state.
<code>set_positions([stations, ...])</code>	Generates antenna positions around the station.
<code>triggered([station_requirement, ...])</code>	Check if the detector is triggered based on its current state.

6.3.2 Antarctic Ross Ice-Shelf Antenna Neutrino Array (`pyrex.custom.arianna`)

The ARIANNA module contains classes for antennas as found in the ARIANNA project.

The `LPDA` class is the model of the ARIANNA LPDA antenna based on data from NuRadioReco. It uses directional/polarization gain from `data/createLPDA_100MHz_InfFirn.adl` and `data/createLPDA_100MHz_InfFirn.ral`, and amplification gain from `data/amp_300_gain.csv` and `data/amp_300_phase.csv`. The trigger condition of the antenna requires the signal to reach above and below some threshold values within a trigger window.

Default Package Imports

<code>LPDA</code>	ARIANNA LPDA antenna system.
-------------------	------------------------------

`pyrex.custom.arianna.LPDA`

```
class pyrex.custom.arianna.LPDA(name, position, threshold, trigger_window=5e-09, z_axis=(0, 0, 1), x_axis=(1, 0, 0), amplification=1, amplifier_clipping=1, noisy=True, unique_noise_waveforms=10)
```

ARIANNA LPDA antenna system.

Applies as the front end a filter representing the ARIANNA amplifier and signal clipping.

Parameters

name [str] Name of the antenna.

position [array_like] Vector position of the antenna.

threshold [float] Voltage sigma threshold for the trigger condition.

trigger_window [float] Time window (ns) for the trigger condition.

z_axis [array_like, optional] Vector direction of the z-axis of the antenna.

x_axis [array_like, optional] Vector direction of the x-axis of the antenna.

amplification [float, optional] Amplification to be applied to the signal pre-clipping.

amplifier_clipping [float, optional] Voltage (V) above which the amplified signal is clipped (in positive and negative values).

noisy [boolean, optional] Whether or not the antenna should add noise to incoming signals.

unique_noise_waveforms [int, optional] The number of expected noise waveforms needed for each received signal to have its own noise.

See also:

ARIANNAAntennaSystem Antenna system extending base ARIANNA antenna with front-end processing.

ARIANNAAntenna Antenna class to be used for ARIANNA antennas.

Attributes

antenna [Antenna] Antenna object extended by the front end.

name [str] Name of the antenna.

position [array_like] Vector position of the antenna.

threshold [float] Voltage sigma threshold for the trigger condition.

trigger_window [float] Time window (ns) for the trigger condition.

amplification [float] Amplification to be applied to the signal pre-clipping. Note that the usual ARA electronics amplification is already applied without this.

amplifier_clipping [float] Voltage (V) above which the amplified signal is clipped (in positive and negative values).

is_hit Boolean of whether the antenna system has been triggered.

is_hit_mc_truth Boolean of whether the antenna has been triggered by signal.

signals The signals received by the antenna with front-end processing.

waveforms The antenna system signal + noise for each triggered hit.

all_waveforms The antenna system signal + noise for all hits.

Methods

<code>apply_response(signal[, direction, ...])</code>	Process the complete antenna response for an incoming signal.
<code>clear([reset_noise])</code>	Reset the antenna system to an empty state.
<code>front_end(signal)</code>	Apply front-end processes to a signal and return the output.
<code>full_waveform(times)</code>	Signal + noise for the antenna system for the given times.
<code>interpolate_filter(frequencies)</code>	Generate interpolated filter values for given frequencies.
<code>is_hit_during(times)</code>	Check if the antenna system is triggered in a time range.
<code>make_noise(times)</code>	Creates a noise signal over the given times.
<code>receive(signal[, direction, polarization, ...])</code>	Process and store one or more incoming (polarized) signals.
<code>set_orientation([z_axis, x_axis])</code>	Sets the orientation of the antenna system.
<code>setup_antenna([center_frequency, bandwidth, ...])</code>	Setup the antenna by passing along its init arguments.
<code>trigger(signal)</code>	Check if the antenna system triggers on a given signal.

Individual Module APIs

Custom Antennas (`pyrex.custom.arianna.antenna`)

Module containing customized antenna classes for ARIANNA.

Based primarily on the LPDA implementation (and data) in NuRadioReco.

<code>_read_response_data</code>	Gather antenna effective height data from a set of WIPLD data files.
<code>_read_amplifier_data</code>	Gather frequency-dependent amplifier data from data files.
<code>ARIANNAAntenna</code>	Antenna class to be used for ARIANNA antennas.
<code>ARIANNAAntennaSystem</code>	Antenna system extending base ARIANNA antenna with front-end processing.
<code>LPDA</code>	ARIANNA LPDA antenna system.

`pyrex.custom.arianna.antenna._read_response_data`

`pyrex.custom.arianna.antenna._read_response_data(filename)`

Gather antenna effective height data from a set of WIPLD data files.

Data files should exist with names *filename.ra1* and *filename.ad1*. The *.ad1* file should contain frequencies in the first column, real and imaginary parts of the impedance in the sixth and seventh columns, and S-parameter data in the eighth and ninth columns. The *.ra1* file should contain phi and theta in the first two columns, the real and imaginary parts of the phi field in the next two columns, and the real and imaginary parts of the theta field in the next two columns. This should be divided into sections for each frequency with a header line ” > Gen. no. 1 X GHz 73 91 Gain” where “X” is the frequency in GHz.

Parameters

filename [str] Name of the data files without extension. Extensions *.ra1* and *.ad1* will be added automatically.

Returns

theta_response [ndarray] 3-D array of complex-valued effective heights in the theta polarization as a function of frequency along axis 0, zenith along axis 1, and azimuth along axis 2.

phi_response [ndarray] 3-D array of complex-valued effective heights in the phi polarization as a function of frequency along axis 0, zenith along axis 1, and azimuth along axis 2.

frequencies [ndarray] Frequencies (Hz) corresponding to axis 0 of *theta_response* and *phi_response*.

thetas [ndarray] Zenith angles (degrees) corresponding to axis 1 of *theta_response* and *phi_response*.

phis [ndarray] Azimuth angles (degrees) corresponding to axis 2 of *theta_response* and *phi_response*.

Raises

ValueError If the frequency values of the *.ra1* and *.ad1* files don’t match.

pyrex.custom.arianna.antenna._read_amplifier_data

```
pyrex.custom.arianna.antenna._read_amplifier_data(gain_filename, phase_filename,
                                                  gain_offset=0)
```

Gather frequency-dependent amplifier data from data files.

Each data file should have columns for frequency, gain or phase data, and a third empty column. The gain should be in dB and the phase should be in degrees.

Parameters

gain_filename [str] Name of the data file containing gains (in dB).

phase_filename [str] Name of the data file containing phases (in degrees).

gain_offset [float, optional] Offset to apply to the gain values (in dB).

Returns

gains [ndarray] Complex-valued voltage gains as a function of frequency.

frequencies [ndarray] Frequencies (Hz) corresponding to the values of *gains*.

pyrex.custom.arianna.antenna.ARIANNAAntenna

```
class pyrex.custom.arianna.antenna.ARIANNAAntenna(response_data, position, center_frequency, bandwidth, resistance, z_axis=(0, 0, 1), x_axis=(1, 0, 0), efficiency=1, noisy=True, unique_noise_waveforms=10)
```

Antenna class to be used for ARIANNA antennas.

Stores the attributes of an antenna as well as handling receiving, processing, and storing signals and adding noise. Antenna response based on provided models.

Parameters

response_data [tuple of array_like] Tuple containing the response data for the antenna along the theta and phi polarization directions. The first and second elements should contain 3-D arrays of the antenna response model in the theta and phi polarizations, respectively, as a function of frequency (axis 0), zenith (axis 1), and azimuth (axis 2). The remaining elements should be the values of the frequency, zenith, and azimuth axes, respectively.

position [array_like] Vector position of the antenna.

center_frequency [float] Frequency (Hz) at the center of the antenna's frequency range.

bandwidth [float] Bandwidth (Hz) of the antenna.

resistance [float] The noise resistance (ohm) of the antenna. Used to calculate the RMS voltage of the antenna noise.

z_axis [array_like, optional] Vector direction of the z-axis of the antenna.

x_axis [array_like, optional] Vector direction of the x-axis of the antenna.

efficiency [float, optional] Antenna efficiency applied to incoming signal values.

noisy [boolean, optional] Whether or not the antenna should add noise to incoming signals.

unique_noise_waveforms [int, optional] The number of expected noise waveforms needed for each received signal to have its own noise.

See also:

`pyrex.Antenna` Base class for antennas.

Attributes

- position** [array_like] Vector position of the antenna.
- z_axis** [ndarray] Vector direction of the z-axis of the antenna.
- x_axis** [ndarray] Vector direction of the x-axis of the antenna.
- antenna_factor** [float] Antenna factor used for converting fields to voltages.
- efficiency** [float] Antenna efficiency applied to incoming signal values.
- noisy** [boolean] Whether or not the antenna should add noise to incoming signals.
- unique_noises** [int] The number of expected noise waveforms needed for each received signal to have its own noise.
- freq_range** [array_like] The frequency band in which the antenna operates (used for noise production).
- temperature** [float or None] The noise temperature (K) of the antenna. Used in combination with *resistance* to calculate the RMS voltage of the antenna noise.
- resistance** [float or None] The noise resistance (ohm) of the antenna. Used in combination with *temperature* to calculate the RMS voltage of the antenna noise.
- noise_rms** [float or None] The RMS voltage (v) of the antenna noise. If not `None`, this value will be used instead of the RMS voltage calculated from the values of *temperature* and *resistance*.
- signals** [list of Signal] The signals which have been received by the antenna.
- is_hit** Boolean of whether the antenna has been triggered.
- is_hit_mc_truth** Boolean of whether the antenna has been triggered by signal.
- waveforms** Signal + noise (if *noisy*) for each triggered antenna hit.
- all_waveforms** Signal + noise (if *noisy*) for all antenna hits.

Methods

<code>apply_response(signal[, direction, ...])</code>	Process the complete antenna response for an incoming signal.
<code>clear([reset_noise])</code>	Reset the antenna to an empty state.
<code>directional_gain(theta, phi)</code>	Calculate the (complex) directional gain of the antenna.
<code>directional_response(theta, phi, polarization)</code>	Generate the (complex) frequency-dependent directional response.
<code>frequency_response(frequencies)</code>	Calculate the (complex) frequency response of the antenna.
<code>full_waveform(times)</code>	Signal + noise (if <i>noisy</i>) for the given times.
<code>is_hit_during(times)</code>	Check if the antenna is triggered in a time range.
<code>make_noise(times)</code>	Creates a noise signal over the given times.

Continued on next page

Table 90 – continued from previous page

<code>polarization_gain(polarization)</code>	Calculate the (complex) polarization gain of the antenna.
<code>receive(signal[, direction, polarization, ...])</code>	Process and store one or more incoming (polarized) signals.
<code>set_orientation([z_axis, x_axis])</code>	Sets the orientation of the antenna.
<code>trigger(signal)</code>	Check if the antenna triggers on a given signal.

pyrex.custom.arianna.antenna.ARIANNAAntennaSystem

```
class pyrex.custom.arianna.antenna.ARIANNAAntennaSystem(response_data, name,
                                                         position, threshold,
                                                         trigger_window=5e-
                                                         09, z_axis=(0, 0, 1),
                                                         x_axis=(1, 0, 0), am-
                                                         plification=1, am-
                                                         plifier_clipping=1,
                                                         noisy=True,
                                                         unique_noise_waveforms=10,
                                                         **kwargs)
```

Antenna system extending base ARIANNA antenna with front-end processing.

Applies as the front end a filter representing the ARIANNA amplifier and signal clipping.

Parameters

response_data [tuple of array_like] Tuple containing the response data for the antenna along the theta and phi polarization directions. The first and second elements should contain 3-D arrays of the antenna response model in the theta and phi polarizations, respectively, as a function of frequency (axis 0), zenith (axis 1), and azimuth (axis 2). The remaining elements should be the values of the frequency, zenith, and azimuth axes, respectively.

name [str] Name of the antenna.

position [array_like] Vector position of the antenna.

threshold [float] Voltage sigma threshold for the trigger condition.

trigger_window [float] Time window (ns) for the trigger condition.

z_axis [array_like, optional] Vector direction of the z-axis of the antenna.

x_axis [array_like, optional] Vector direction of the x-axis of the antenna.

amplification [float, optional] Amplification to be applied to the signal pre-clipping.

amplifier_clipping [float, optional] Voltage (V) above which the amplified signal is clipped (in positive and negative values).

noisy [boolean, optional] Whether or not the antenna should add noise to incoming signals.

unique_noise_waveforms [int, optional] The number of expected noise waveforms needed for each received signal to have its own noise.

See also:

[`pyrex.AntennaSystem`](#) Base class for antenna system with front-end processing.

[`ARIANNAAntenna`](#) Antenna class to be used for ARIANNA antennas.

Attributes

antenna [Antenna] Antenna object extended by the front end.

name [str] Name of the antenna.

position [array_like] Vector position of the antenna.

threshold [float] Voltage sigma threshold for the trigger condition.

trigger_window [float] Time window (s) for the trigger condition.

amplification [float] Amplification to be applied to the signal pre-clipping. Note that the usual ARA electronics amplification is already applied without this.

amplifier_clipping [float] Voltage (V) above which the amplified signal is clipped (in positive and negative values).

lead_in_time [float] Lead-in time (s) required for the front end to equilibrate. Automatically added in before calculation of signals and waveforms.

is_hit Boolean of whether the antenna system has been triggered.

is_hit_mc_truth Boolean of whether the antenna has been triggered by signal.

signals The signals received by the antenna with front-end processing.

waveforms The antenna system signal + noise for each triggered hit.

all_waveforms The antenna system signal + noise for all hits.

Methods

<code>apply_response(signal[, direction, ...])</code>	Process the complete antenna response for an incoming signal.
<code>clear([reset_noise])</code>	Reset the antenna system to an empty state.
<code>front_end(signal)</code>	Apply front-end processes to a signal and return the output.
<code>full_waveform(times)</code>	Signal + noise for the antenna system for the given times.
<code>interpolate_filter(frequencies)</code>	Generate interpolated filter values for given frequencies.
<code>is_hit_during(times)</code>	Check if the antenna system is triggered in a time range.
<code>make_noise(times)</code>	Creates a noise signal over the given times.
<code>receive(signal[, direction, polarization, ...])</code>	Process and store one or more incoming (polarized) signals.
<code>set_orientation([z_axis, x_axis])</code>	Sets the orientation of the antenna system.
<code>setup_antenna([center_frequency, bandwidth, ...])</code>	Setup the antenna by passing along its init arguments.
<code>trigger(signal)</code>	Check if the antenna system triggers on a given signal.

pyrex.custom.arianna.antenna.LPDA

class `pyrex.custom.arianna.antenna.LPDA` (*name, position, threshold, trigger_window=5e-09, z_axis=(0, 0, 1), x_axis=(1, 0, 0), amplification=1, amplifier_clipping=1, noisy=True, unique_noise_waveforms=10*)

ARIANNA LPDA antenna system.

Applies as the front end a filter representing the ARIANNA amplifier and signal clipping.

Parameters

- name** [str] Name of the antenna.
- position** [array_like] Vector position of the antenna.
- threshold** [float] Voltage sigma threshold for the trigger condition.
- trigger_window** [float] Time window (ns) for the trigger condition.
- z_axis** [array_like, optional] Vector direction of the z-axis of the antenna.
- x_axis** [array_like, optional] Vector direction of the x-axis of the antenna.
- amplification** [float, optional] Amplification to be applied to the signal pre-clipping.
- amplifier_clipping** [float, optional] Voltage (V) above which the amplified signal is clipped (in positive and negative values).
- noisy** [boolean, optional] Whether or not the antenna should add noise to incoming signals.
- unique_noise_waveforms** [int, optional] The number of expected noise waveforms needed for each received signal to have its own noise.

See also:

[**ARIANNAAntennaSystem**](#) Antenna system extending base ARIANNA antenna with front-end processing.

[**ARIANNAAntenna**](#) Antenna class to be used for ARIANNA antennas.

Attributes

- antenna** [Antenna] *Antenna* object extended by the front end.
- name** [str] Name of the antenna.
- position** [array_like] Vector position of the antenna.
- threshold** [float] Voltage sigma threshold for the trigger condition.
- trigger_window** [float] Time window (ns) for the trigger condition.
- amplification** [float] Amplification to be applied to the signal pre-clipping. Note that the usual ARA electronics amplification is already applied without this.
- amplifier_clipping** [float] Voltage (V) above which the amplified signal is clipped (in positive and negative values).
- is_hit** Boolean of whether the antenna system has been triggered.
- is_hit_mc_truth** Boolean of whether the antenna has been triggered by signal.
- signals** The signals received by the antenna with front-end processing.
- waveforms** The antenna system signal + noise for each triggered hit.
- all_waveforms** The antenna system signal + noise for all hits.

Methods

<code>apply_response(signal[, direction, ...])</code>	Process the complete antenna response for an incoming signal.
---	---

Continued on next page

Table 92 – continued from previous page

<code>clear([reset_noise])</code>	Reset the antenna system to an empty state.
<code>front_end(signal)</code>	Apply front-end processes to a signal and return the output.
<code>full_waveform(times)</code>	Signal + noise for the antenna system for the given times.
<code>interpolate_filter(frequencies)</code>	Generate interpolated filter values for given frequencies.
<code>is_hit_during(times)</code>	Check if the antenna system is triggered in a time range.
<code>make_noise(times)</code>	Creates a noise signal over the given times.
<code>receive(signal[, direction, polarization, ...])</code>	Process and store one or more incoming (polarized) signals.
<code>set_orientation([z_axis, x_axis])</code>	Sets the orientation of the antenna system.
<code>setup_antenna([center_frequency, bandwidth, ...])</code>	Setup the antenna by passing along its init arguments.
<code>trigger(signal)</code>	Check if the antenna system triggers on a given signal.

6.3.3 IceCube Radio Extension (`pyrex.custom.irex`)

The IREX module contains classes for antennas and detectors which use waveform envelopes rather than raw waveforms. The detectors provided allow for testing of grid and station geometries.

The `EvelopeHpol` and `EvelopeVpol` classes wrap models of ARA Hpol and Vpol antennas with an additional front-end which uses a diode-bridge circuit to create waveform envelopes. The trigger condition for these antennas is a simple threshold trigger on the envelopes.

The `IREXString` class creates a string of `EvelopeVpol` antennas at a given position. The `RegularStation` class creates a station at a given position with 4 (or another given number) strings spaced evenly around the station center. The `CoxeterStation` class creates a station at a given position similar to the `RegularStation`, but with one string at the station center and the rest spaced evenly around the center. The `StationGrid` class creates a rectangular grid of stations (or strings, as specified by the station type). The dimensions of the grid in stations is N_x by N_y where N is the total number of stations, $N_x = \text{floor}(\sqrt{N})$, and $N_y = \text{floor}(N/N_x)$.

Default Package Imports

<code>EnvelopeHpol</code>	ARA Hpol (“quad-slot”) antenna system with front-end processing.
<code>EnvelopeVpol</code>	ARA Vpol (“bicone” or “birdcage”) antenna system with front-end processing.
<code>IREXString</code>	String of IREX Vpol antennas.
<code>RegularStation</code>	Station geometry with strings evenly spaced radially around the center.
<code>CoxeterStation</code>	Station geometry with center string and the rest evenly spaced radially.
<code>StationGrid</code>	Rectangular grid of stations or strings.

pyrex.custom.irex.EnvelopeHpol

```
class pyrex.custom.irex.EnvelopeHpol (name, position, trigger_threshold,
                                     time_over_threshold=0, orientation=(0, 0, 1),
                                     amplification=1, amplifier_clipping=1, envelope_amplification=1,
                                     envelope_method='analytic', noisy=True, unique_noise_waveforms=10)
```

ARA Hpol (“quad-slot”) antenna system with front-end processing.

Consists of an ARA Hpol antenna with typical responses, front-end electronics, and amplifier clipping, but with an additional amplification and envelope circuit applied after all other front-end processing.

Parameters

- name** [str] Name of the antenna.
- position** [array_like] Vector position of the antenna.
- trigger_threshold** [float] Threshold (V) for trigger condition. Antenna triggers if the voltage value of the waveform exceeds this value.
- time_over_threshold** [float, optional] Time (s) that the voltage waveform must exceed *trigger_threshold* for the antenna to trigger.
- orientation** [array_like, optional] Vector direction of the z-axis of the antenna.
- amplification** [float, optional] Amplification to be applied to the signal pre-clipping. Note that the usual ARA electronics amplification is already applied without this.
- amplifier_clipping** [float, optional] Voltage (V) above which the amplified signal is clipped (in positive and negative values).
- envelope_amplification** [float, optional] Amplification to be applied to the signal after the typical ARA front end, before the envelope circuit.
- envelope_method** [(('hilbert', 'analytic', 'spice') + ('basic', 'biased', 'doubler', 'bridge', 'log amp')), optional] String describing the circuit (and calculation method) to be used for envelope calculation. If the string contains “hilbert”, the hilbert envelope is used. If the string contains “analytic”, an analytic form is used to calculate the circuit output. If the string contains “spice”, *ngspice* is used to calculate the circuit output. The default value “analytic” uses an analytic diode bridge circuit.
- noisy** [boolean, optional] Whether or not the antenna should add noise to incoming signals.
- unique_noise_waveforms** [int, optional] The number of expected noise waveforms needed for each received signal to have its own noise.

Attributes

- antenna** [Antenna] *Antenna* object extended by the front end.
- name** [str] Name of the antenna.
- position** [array_like] Vector position of the antenna.
- trigger_threshold** [float] Threshold (V) for trigger condition. Antenna triggers if the voltage value of the waveform exceeds this value.
- time_over_threshold** [float] Time (s) that the voltage waveform must exceed *trigger_threshold* for the antenna to trigger.
- envelope_amplification** [float] Amplification to be applied to the signal pre-clipping. Note that the usual ARA electronics amplification is already applied without this.

envelope_method [str] String describing the circuit (and calculation method) to be used for envelope calculation.

is_hit Boolean of whether the antenna system has been triggered.

is_hit_mc_truth Boolean of whether the antenna has been triggered by signal.

signals The signals received by the antenna with front-end processing.

waveforms The antenna system signal + noise for each triggered hit.

all_waveforms The antenna system signal + noise for all hits.

Methods

<code>apply_response(signal[, direction, ...])</code>	Process the complete antenna response for an incoming signal.
<code>clear([reset_noise])</code>	Reset the antenna system to an empty state.
<code>envelopeless_front_end(signal)</code>	Apply front-end processes to a signal and return the output.
<code>front_end(signal)</code>	Apply front-end processes to a signal and return the output.
<code>full_waveform(times)</code>	Signal + noise for the antenna system for the given times.
<code>interpolate_filter(frequencies)</code>	Generate interpolated filter values for given frequencies.
<code>is_hit_during(times)</code>	Check if the antenna system is triggered in a time range.
<code>make_envelope(signal)</code>	Return the signal envelope based on the antenna's <code>envelope_method</code> .
<code>make_noise(times)</code>	Creates a noise signal over the given times.
<code>receive(signal[, direction, polarization, ...])</code>	Process and store one or more incoming (polarized) signals.
<code>set_orientation([z_axis, x_axis])</code>	Sets the orientation of the antenna system.
<code>setup_antenna(response_data[, ...])</code>	Setup the antenna by passing along its init arguments.
<code>trigger(signal)</code>	Check if the antenna triggers on a given signal.
<code>tunnel_diode(signal)</code>	Calculate a signal as processed by the tunnel diode.

pyrex.custom.irex.EnvelopeVpol

```
class pyrex.custom.irex.EnvelopeVpol (name, position, trigger_threshold,
                                     time_over_threshold=0, orientation=(0, 0, 1),
                                     amplification=1, amplifier_clipping=1, envelope_amplification=1,
                                     envelope_method='analytic',
                                     noisy=True, unique_noise_waveforms=10)
```

ARA Vpol (“bicone” or “birdcage”) antenna system with front-end processing.

Consists of an ARA Vpol antenna with typical responses, front-end electronics, and amplifier clipping, but with an additional amplification and envelope circuit applied after all other front-end processing.

Parameters

name [str] Name of the antenna.

position [array_like] Vector position of the antenna.

trigger_threshold [float] Threshold (V) for trigger condition. Antenna triggers if the voltage value of the waveform exceeds this value.

time_over_threshold [float, optional] Time (s) that the voltage waveform must exceed *trigger_threshold* for the antenna to trigger.

orientation [array_like, optional] Vector direction of the z-axis of the antenna.

amplification [float, optional] Amplification to be applied to the signal pre-clipping. Note that the usual ARA electronics amplification is already applied without this.

amplifier_clipping [float, optional] Voltage (V) above which the amplified signal is clipped (in positive and negative values).

envelope_amplification [float, optional] Amplification to be applied to the signal after the typical ARA front end, before the envelope circuit.

envelope_method [{"('hilbert', 'analytic', 'spice') + ('basic', 'biased', 'doubler', 'bridge', 'log amp')}, optional] String describing the circuit (and calculation method) to be used for envelope calculation. If the string contains "hilbert", the hilbert envelope is used. If the string contains "analytic", an analytic form is used to calculate the circuit output. If the string contains "spice", *ngspice* is used to calculate the circuit output. The default value "analytic" uses an analytic diode bridge circuit.

noisy [boolean, optional] Whether or not the antenna should add noise to incoming signals.

unique_noise_waveforms [int, optional] The number of expected noise waveforms needed for each received signal to have its own noise.

Attributes

antenna [Antenna] *Antenna* object extended by the front end.

name [str] Name of the antenna.

position [array_like] Vector position of the antenna.

trigger_threshold [float] Threshold (V) for trigger condition. Antenna triggers if the voltage value of the waveform exceeds this value.

time_over_threshold [float] Time (s) that the voltage waveform must exceed *trigger_threshold* for the antenna to trigger.

envelope_amplification [float] Amplification to be applied to the signal pre-clipping. Note that the usual ARA electronics amplification is already applied without this.

envelope_method [str] String describing the circuit (and calculation method) to be used for envelope calculation.

is_hit Boolean of whether the antenna system has been triggered.

is_hit_mc_truth Boolean of whether the antenna has been triggered by signal.

signals The signals received by the antenna with front-end processing.

waveforms The antenna system signal + noise for each triggered hit.

all_waveforms The antenna system signal + noise for all hits.

Methods

<code>apply_response(signal[, direction, ...])</code>	Process the complete antenna response for an incoming signal.
<code>clear([reset_noise])</code>	Reset the antenna system to an empty state.
<code>envelopeless_front_end(signal)</code>	Apply front-end processes to a signal and return the output.
<code>front_end(signal)</code>	Apply front-end processes to a signal and return the output.
<code>full_waveform(times)</code>	Signal + noise for the antenna system for the given times.
<code>interpolate_filter(frequencies)</code>	Generate interpolated filter values for given frequencies.
<code>is_hit_during(times)</code>	Check if the antenna system is triggered in a time range.
<code>make_envelope(signal)</code>	Return the signal envelope based on the antenna's <code>envelope_method</code> .
<code>make_noise(times)</code>	Creates a noise signal over the given times.
<code>receive(signal[, direction, polarization, ...])</code>	Process and store one or more incoming (polarized) signals.
<code>set_orientation([z_axis, x_axis])</code>	Sets the orientation of the antenna system.
<code>setup_antenna(response_data[, ...])</code>	Setup the antenna by passing along its init arguments.
<code>trigger(signal)</code>	Check if the antenna triggers on a given signal.
<code>tunnel_diode(signal)</code>	Calculate a signal as processed by the tunnel diode.

pyrex.custom.irex.IREXString

class `pyrex.custom.irex.IREXString`(*x*, *y*, *antennas_per_string*=2, *antenna_separation*=50, *lowest_antenna*=-100)

String of IREX Vpol antennas.

Sets the positions of antennas on string based on the parameters. Once the antennas have been built with `build_antennas`, the object can be directly iterated over to iterate over the antennas (as if the object were just a list of the antennas).

Parameters

x [float] Cartesian x-position (m) of the string.

y [float] Cartesian y-position (m) of the string.

antennas_per_string [float, optional] Total number of antennas to be placed on the string.

antenna_separation [float or list of float, optional] The vertical separation (m) of antennas on the string. If `float`, all antennas are separated by the same constant value. If `list`, the separations in the list are the separations of neighboring antennas starting from the lowest up to the highest.

lowest_antenna [float, optional] The Cartesian z-position (m) of the lowest antenna on the string.

Raises

ValueError If `test_antenna_positions` is `True` and an antenna is found to be above the ice surface.

See also:

`pyrex.custom.irex.EnvelopeHpol` ARA Hpol (“quad-slot”) antenna system with front-end processing.

`pyrex.custom.irex.EnvelopeVpol` ARA Vpol (“bicone” or “birdcage”) antenna system with front-end processing.

Notes

This class is designed to be the lowest subset level of a detector. It can (and should) be used for the subsets of some other `Detector` subclass to build up a full detector. Then when its “parent” is iterated, the instances of this class will be iterated as though they were all part of one flat list.

Attributes

antenna_positions [list] List (potentially with sub-lists) of the positions of the antennas generated by the `set_positions` method.

subsets [list] List of the antenna or detector objects which make up the detector.

test_antenna_positions [boolean] Class attribute for whether or not an error should be raised if antenna positions are found above the surface of the ice (where simulation behavior is ill-defined). Defaults to `True`.

Methods

<code>build_antennas(trigger_threshold[, ...])</code>	Creates antenna objects at the set antenna positions.
<code>clear([reset_noise])</code>	Reset the detector to an empty state.
<code>set_positions(x, y[, antennas_per_string, ...])</code>	Generates antenna positions along the string.
<code>triggered([antenna_requirement, ...])</code>	Check if the string is triggered based on its current state.

`pyrex.custom.irex.RegularStation`

```
class pyrex.custom.irex.RegularStation(x, y, strings_per_station=4, station_diameter=50, string_type=<class 'pyrex.custom.irex.detector.IREXString'>, **string_kwargs)
```

Station geometry with strings evenly spaced radially around the center.

Sets the positions of strings around the station based on the parameters. Supports any string type and passes extra keyword arguments on to the string class. Once the antennas have been built with `build_antennas`, the object can be directly iterated over to iterate over the antennas (as if the object were just a list of the antennas).

Parameters

x [float] Cartesian x-position (m) of the station.

y [float] Cartesian y-position (m) of the station.

strings_per_station [float, optional] Number of strings to be placed evenly around the station.

station_diameter [float, optional] Diameter (m) of the circle around which strings are placed.

string_type [optional] Class to be used for creating string objects for *subsets*.

****string_kwargs** Keyword arguments to be passed on to the `__init__` methods of the *string_type* class.

Raises

ValueError If `test_antenna_positions` is `True` and an antenna is found to be above the ice surface.

See also:

`pyrex.custom.irex.EnvelopeHpol` ARA Hpol (“quad-slot”) antenna system with front-end processing.

`pyrex.custom.irex.EnvelopeVpol` ARA Vpol (“bicone” or “birdcage”) antenna system with front-end processing.

`IREXString` String of IREX Vpol antennas.

Notes

This class is designed to have string-like objects (which are subclasses of `Detector`) as its *subsets*. Then whenever an object of this class is iterated, all the antennas of its strings will be yielded as in a 1D list.

Attributes

antenna_positions [list] List (potentially with sub-lists) of the positions of the antennas generated by the `set_positions` method.

subsets [list] List of the antenna or detector objects which make up the detector.

test_antenna_positions [boolean] Class attribute for whether or not an error should be raised if antenna positions are found above the surface of the ice (where simulation behavior is ill-defined). Defaults to `True`.

Methods

<code>build_antennas(*args, **kwargs)</code>	Creates antenna objects at the set antenna positions.
<code>clear([reset_noise])</code>	Reset the detector to an empty state.
<code>set_positions(x, y[, strings_per_station, ...])</code>	Generates antenna positions around the station.
<code>triggered([antenna_requirement, ...])</code>	Check if the station is triggered based on its current state.

pyrex.custom.irex.CoxeterStation

```
class pyrex.custom.irex.CoxeterStation(x, y, strings_per_station=4, station_diameter=50, string_type=<class 'pyrex.custom.irex.detector.IREXString'>, **string_kwargs)
```

Station geometry with center string and the rest evenly spaced radially.

Sets the positions of strings around the station based on the parameters. Supports any string type and passes extra keyword arguments on to the string class. Once the antennas have been built with `build_antennas`, the object can be directly iterated over to iterate over the antennas (as if the object were just a list of the antennas).

Parameters

x [float] Cartesian x-position (m) of the station.

y [float] Cartesian y-position (m) of the station.

strings_per_station [float, optional] Number of strings to be placed around the station. Note that the first string is always placed at the center and the rest of the strings are placed evenly around that center string.

station_diameter [float, optional] Diameter (m) of the circle around which strings are placed.

string_type [optional] Class to be used for creating string objects for *subsets*.

****string_kwargs** Keyword arguments to be passed on to the `__init__` methods of the *string_type* class.

Raises

ValueError If `test_antenna_positions` is `True` and an antenna is found to be above the ice surface.

See also:

pyrex.custom.irex.EnvelopeHpol ARA Hpol (“quad-slot”) antenna system with front-end processing.

pyrex.custom.irex.EnvelopeVpol ARA Vpol (“bicone” or “birdcage”) antenna system with front-end processing.

IREXString String of IREX Vpol antennas.

Notes

This class is designed to have string-like objects (which are subclasses of `Detector`) as its *subsets*. Then whenever an object of this class is iterated, all the antennas of its strings will be yielded as in a 1D list.

Attributes

antenna_positions [list] List (potentially with sub-lists) of the positions of the antennas generated by the `set_positions` method.

subsets [list] List of the antenna or detector objects which make up the detector.

test_antenna_positions [boolean] Class attribute for whether or not an error should be raised if antenna positions are found above the surface of the ice (where simulation behavior is ill-defined). Defaults to `True`.

Methods

<code>build_antennas(*args, **kwargs)</code>	Creates antenna objects at the set antenna positions.
<code>clear([reset_noise])</code>	Reset the detector to an empty state.
<code>set_positions(x, y[, strings_per_station, ...])</code>	Generates antenna positions around the station.
<code>triggered([antenna_requirement, ...])</code>	Check if the station is triggered based on its current state.

pyrex.custom.irex.StationGrid

```
class pyrex.custom.irex.StationGrid(stations=1,          station_separation=500,          sta-
                                tion_type=<class 'pyrex.custom.irex.detector.IREXString'>,
                                **station_kwargs)
```

Rectangular grid of stations or strings.

Sets the positions of stations in a square layout if possible, otherwise in a rectangular layout (drops any extra

stations). Supports any station type (including string types) and passes extra keyword arguments on to the station class. Once the antennas have been built with `build_antennas`, the object can be directly iterated over to iterate over the antennas (as if the object were just a list of the antennas).

Parameters

- stations** [float, optional] Number of stations to be placed.
- station_separation** [float, optional] Distance (m) between adjacent stations.
- station_type** [optional] Class to be used for creating station objects for *subsets*.
- **station_kwargs** Keyword arguments to be passed on to the `__init__` methods of the *station_type* class.

Raises

- ValueError** If `test_antenna_positions` is `True` and an antenna is found to be above the ice surface.

Warning: If the number of *stations* provided does not divide nicely into a rectangle, extra stations may be dropped without warning. For example, if *stations* is 5, then a 2x2 grid will be created and the last station will be silently dropped.

See also:

IREXString String of IREX Vpol antennas.

RegularStation Station geometry with strings evenly spaced radially around the center.

CoxeterStation Station geometry with center string and the rest evenly spaced radially.

Notes

This class is designed to have station-like or string-like objects (which are subclasses of `Detector`) as its *subsets*. Then whenever an object of this class is iterated, all the antennas of its strings will be yielded as in a 1D list.

Attributes

- antenna_positions** [list] List (potentially with sub-lists) of the positions of the antennas generated by the `set_positions` method.
- subsets** [list] List of the antenna or detector objects which make up the detector.
- test_antenna_positions** [boolean] Class attribute for whether or not an error should be raised if antenna positions are found above the surface of the ice (where simulation behavior is ill-defined). Defaults to `True`.

Methods

<code>build_antennas(*args, **kwargs)</code>	Creates antenna objects at the set antenna positions.
<code>clear([reset_noise])</code>	Reset the detector to an empty state.
<code>set_positions([stations, ...])</code>	Generates antenna positions around the station.
<code>triggered([station_requirement, ...])</code>	Check if the detector is triggered based on its current state.

Individual Module APIs

Custom Front-ends (`pyrex.custom.irex.frontends`)

Module containing IREX front-end circuit models.

Contains wrappers for PySpice circuits as well as analytical forms for some envelope circuits.

<code>basic_envelope_model</code>	Model of a basic diode-capacitor-resistor envelope circuit.
<code>bridge_rectifier_envelope_model</code>	Model of a diode bridge rectifier envelope circuit.

`pyrex.custom.irex.frontends.basic_envelope_model`

`pyrex.custom.irex.frontends.basic_envelope_model` (*signal*, *cap*= $2e-11$, *res*=500)

Model of a basic diode-capacitor-resistor envelope circuit.

Passes the input signal through a basic envelope circuit consisting of a diode, a capacitor, and a resistor. The diode used is modeled after an HSMS 2852 diode.

Parameters

- signal** [Signal] Signal object used as input to the circuit.
- cap** [float, optional] Capacitance (F) of the circuit's capacitor C1.
- res** [float, optional] Resistance (ohm) of the circuit's resistor R1.

Returns

Signal Output of the envelope circuit for the given input.

Notes

Ascii depiction of the basic envelope circuit:

```

Vin---D1---+---+---out
           |   |
          C1  R1
           |   |
           +---+
           |
          gnd

```

`pyrex.custom.irex.frontends.bridge_rectifier_envelope_model`

`pyrex.custom.irex.frontends.bridge_rectifier_envelope_model` (*signal*, *cap*= $2e-11$, *res*=500)

Model of a diode bridge rectifier envelope circuit.

Passes the input signal through a diode bridge rectifier envelope circuit consisting of four diodes in a diode bridge, a capacitor, and a resistor. The diode used is modeled after an HSMS 2852 diode.

Parameters

- signal** [Signal] Signal object used as input to the circuit.

cap [float, optional] Capacitance (F) of the circuit’s capacitor C1.

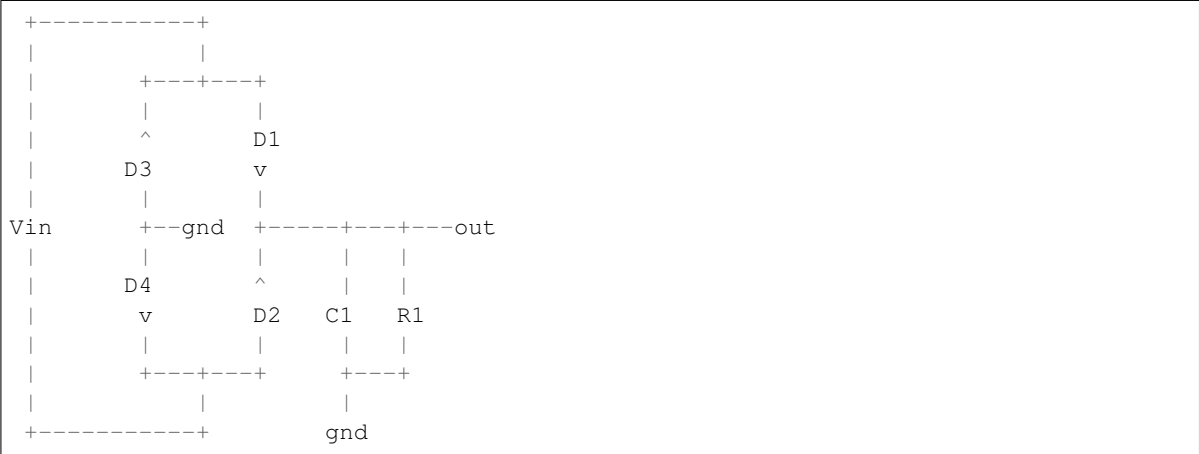
res [float, optional] Resistance (ohm) of the circuit’s resistor R1.

Returns

Signal Output of the envelope circuit for the given input.

Notes

Ascii depiction of the diode bridge rectifier envelope circuit:



Custom Antennas (pyrex.custom.irex.antenna)

Module containing customized antenna classes for IREX.

The IREX antennas are based around existing ARA antennas with an extra envelope circuit applied in the front-end, designed to reduce power consumption and the amount of digitized information.

<i>DipoleTester</i>	Dipole antenna for IREX testing.
<i>EnvelopeSystem</i>	Antenna system extending ARA antennas with an envelope circuit.
<i>EnvelopeHpol</i>	ARA Hpol (“quad-slot”) antenna system with front-end processing.
<i>EnvelopeVpol</i>	ARA Vpol (“bicone” or “birdcage”) antenna system with front-end processing.

pyrex.custom.irex.antenna.DipoleTester

class pyrex.custom.irex.antenna.**DipoleTester** (*position*, *center_frequency*, *bandwidth*, *resistance*, *orientation*=(0, 0, 1), *effective_height*=None, *noisy*=True, *unique_noise_waveforms*=10)

Dipole antenna for IREX testing.

Stores the attributes of an antenna as well as handling receiving, processing, and storing signals and adding noise. Uses a first-order butterworth filter for the frequency response.

Parameters

position [array_like] Vector position of the antenna.

center_frequency [float] Tuned frequency (Hz) of the dipole.

bandwidth [float] Bandwidth (Hz) of the antenna.

resistance [float] The noise resistance (ohm) of the antenna. Used to calculate the RMS voltage of the antenna noise.

orientation [array_like, optional] Vector direction of the z-axis of the antenna.

effective_height [float, optional] Effective length (m) of the antenna. By default calculated by the tuned *center_frequency* of the dipole.

noisy [boolean, optional] Whether or not the antenna should add noise to incoming signals.

unique_noise_waveforms [int, optional] The number of expected noise waveforms needed for each received signal to have its own noise.

Attributes

name [str] Name of the antenna.

position [array_like] Vector position of the antenna.

z_axis [ndarray] Vector direction of the z-axis of the antenna.

x_axis [ndarray] Vector direction of the x-axis of the antenna.

antenna_factor [float] Antenna factor used for converting fields to voltages.

efficiency [float] Antenna efficiency applied to incoming signal values.

threshold [float, optional] Voltage threshold (V) above which signals will trigger.

effective_height [float, optional] Effective length of the antenna. By default calculated by the tuned *center_frequency* of the dipole.

filter_coeffs [tuple of ndarray] Coefficients of transfer function for butterworth bandpass filter to be used for frequency response.

noisy [boolean] Whether or not the antenna should add noise to incoming signals.

unique_noises [int] The number of expected noise waveforms needed for each received signal to have its own noise.

freq_range [array_like] The frequency band in which the antenna operates (used for noise production).

temperature [float or None] The noise temperature (K) of the antenna. Used in combination with *resistance* to calculate the RMS voltage of the antenna noise.

resistance [float or None] The noise resistance (ohm) of the antenna. Used in combination with *temperature* to calculate the RMS voltage of the antenna noise.

noise_rms [float or None] The RMS voltage (V) of the antenna noise. If not *None*, this value will be used instead of the RMS voltage calculated from the values of *temperature* and *resistance*.

signals [list of Signal] The signals which have been received by the antenna.

is_hit Boolean of whether the antenna has been triggered.

is_hit_mc_truth Boolean of whether the antenna has been triggered by signal.

waveforms Signal + noise (if *noisy*) for each triggered antenna hit.

all_waveforms Signal + noise (if *noisy*) for all antenna hits.

Methods

<code>apply_response(signal[, direction, ...])</code>	Process the complete antenna response for an incoming signal.
<code>clear([reset_noise])</code>	Reset the antenna to an empty state.
<code>directional_gain(theta, phi)</code>	Calculate the (complex) directional gain of the antenna.
<code>frequency_response(frequencies)</code>	Calculate the (complex) frequency response of the antenna.
<code>full_waveform(times)</code>	Signal + noise (if <code>noisy</code>) for the given times.
<code>is_hit_during(times)</code>	Check if the antenna is triggered in a time range.
<code>make_noise(times)</code>	Creates a noise signal over the given times.
<code>polarization_gain(polarization)</code>	Calculate the (complex) polarization gain of the antenna.
<code>receive(signal[, direction, polarization, ...])</code>	Process and store one or more incoming (polarized) signals.
<code>set_orientation([z_axis, x_axis])</code>	Sets the orientation of the antenna.
<code>trigger(signal)</code>	Check if the antenna triggers on a given signal.

pyrex.custom.irex.antenna.EnvelopeSystem

```
class pyrex.custom.irex.antenna.EnvelopeSystem(response_data, name, position, trigger_threshold, time_over_threshold=0, orientation=(0, 0, 1), amplification=1, amplifier_clipping=1, envelope_amplification=1, envelope_method='analytic', noisy=True, unique_noise_waveforms=10)
```

Antenna system extending ARA antennas with an envelope circuit.

Consists of an ARA antenna with typical responses, front-end electronics, and amplifier clipping, but with an additional amplification and envelope circuit applied after all other front-end processing.

Parameters

response_data [tuple of array_like] Tuple containing the response data for the antenna along the theta and phi polarization directions. The first and second elements should contain 3-D arrays of the antenna response model in the theta and phi polarizations, respectively, as a function of frequency (axis 0), zenith (axis 1), and azimuth (axis 2). The remaining elements should be the values of the frequency, zenith, and azimuth axes, respectively.

name [str] Name of the antenna.

position [array_like] Vector position of the antenna.

trigger_threshold [float] Threshold (V) for trigger condition. Antenna triggers if the voltage value of the waveform exceeds this value.

time_over_threshold [float, optional] Time (s) that the voltage waveform must exceed *trigger_threshold* for the antenna to trigger.

orientation [array_like, optional] Vector direction of the z-axis of the antenna.

amplification [float, optional] Amplification to be applied to the signal pre-clipping. Note that the usual ARA electronics amplification is already applied without this.

amplifier_clipping [float, optional] Voltage (V) above which the amplified signal is clipped (in positive and negative values).

envelope_amplification [float, optional] Amplification to be applied to the signal after the typical ARA front end, before the envelope circuit.

envelope_method [{"('hilbert', 'analytic', 'spice') + ('basic', 'biased', 'doubler', 'bridge', 'log amp')}, optional] String describing the circuit (and calculation method) to be used for envelope calculation. If the string contains "hilbert", the hilbert envelope is used. If the string contains "analytic", an analytic form is used to calculate the circuit output. If the string contains "spice", `ngspice` is used to calculate the circuit output. The default value "analytic" uses an analytic diode bridge circuit.

noisy [boolean, optional] Whether or not the antenna should add noise to incoming signals.

unique_noise_waveforms [int, optional] The number of expected noise waveforms needed for each received signal to have its own noise.

See also:

`pyrex.custom.ara.antenna.ARAAntennaSystem` Antenna system extending base ARA antenna with front-end processing.

`pyrex.custom.ara.antenna.ARAAntenna` Antenna class to be used for ARA antennas.

Attributes

antenna [Antenna] *Antenna* object extended by the front end.

name [str] Name of the antenna.

position [array_like] Vector position of the antenna.

trigger_threshold [float] Threshold (V) for trigger condition. Antenna triggers if the voltage value of the waveform exceeds this value.

time_over_threshold [float] Time (s) that the voltage waveform must exceed *trigger_threshold* for the antenna to trigger.

envelope_amplification [float] Amplification to be applied to the signal pre-clipping. Note that the usual ARA electronics amplification is already applied without this.

envelope_method [str] String describing the circuit (and calculation method) to be used for envelope calculation.

lead_in_time [float] Lead-in time (s) required for the front end to equilibrate. Automatically added in before calculation of signals and waveforms.

is_hit Boolean of whether the antenna system has been triggered.

is_hit_mc_truth Boolean of whether the antenna has been triggered by signal.

signals The signals received by the antenna with front-end processing.

waveforms The antenna system signal + noise for each triggered hit.

all_waveforms The antenna system signal + noise for all hits.

Methods

<code>apply_response(signal[, direction, ...])</code>	Process the complete antenna response for an incoming signal.
<code>clear([reset_noise])</code>	Reset the antenna system to an empty state.
<code>envelopeless_front_end(signal)</code>	Apply front-end processes to a signal and return the output.
<code>front_end(signal)</code>	Apply front-end processes to a signal and return the output.
<code>full_waveform(times)</code>	Signal + noise for the antenna system for the given times.
<code>interpolate_filter(frequencies)</code>	Generate interpolated filter values for given frequencies.
<code>is_hit_during(times)</code>	Check if the antenna system is triggered in a time range.
<code>make_envelope(signal)</code>	Return the signal envelope based on the antenna's <code>envelope_method</code> .
<code>make_noise(times)</code>	Creates a noise signal over the given times.
<code>receive(signal[, direction, polarization, ...])</code>	Process and store one or more incoming (polarized) signals.
<code>set_orientation([z_axis, x_axis])</code>	Sets the orientation of the antenna system.
<code>setup_antenna(response_data[, ...])</code>	Setup the antenna by passing along its init arguments.
<code>trigger(signal)</code>	Check if the antenna triggers on a given signal.
<code>tunnel_diode(signal)</code>	Calculate a signal as processed by the tunnel diode.

pyrex.custom.irex.antenna.EnvelopeHpol

```
class pyrex.custom.irex.antenna.EnvelopeHpol (name, position, trigger_threshold,
                                             time_over_threshold=0, orientation
                                             =(0, 0, 1), amplification=1, amplifier_clipping=1,
                                             envelope_amplification=1,
                                             envelope_method='analytic', noisy=True,
                                             unique_noise_waveforms=10)
```

ARA Hpol (“quad-slot”) antenna system with front-end processing.

Consists of an ARA Hpol antenna with typical responses, front-end electronics, and amplifier clipping, but with an additional amplification and envelope circuit applied after all other front-end processing.

Parameters

name [str] Name of the antenna.

position [array_like] Vector position of the antenna.

trigger_threshold [float] Threshold (V) for trigger condition. Antenna triggers if the voltage value of the waveform exceeds this value.

time_over_threshold [float, optional] Time (s) that the voltage waveform must exceed *trigger_threshold* for the antenna to trigger.

orientation [array_like, optional] Vector direction of the z-axis of the antenna.

amplification [float, optional] Amplification to be applied to the signal pre-clipping. Note that the usual ARA electronics amplification is already applied without this.

amplifier_clipping [float, optional] Voltage (V) above which the amplified signal is clipped (in positive and negative values).

envelope_amplification [float, optional] Amplification to be applied to the signal after the typical ARA front end, before the envelope circuit.

envelope_method [{"hilbert", "analytic", "spice"} + ("basic", "biased", "doubler", "bridge", "log amp")], optional] String describing the circuit (and calculation method) to be used for envelope calculation. If the string contains "hilbert", the hilbert envelope is used. If the string contains "analytic", an analytic form is used to calculate the circuit output. If the string contains "spice", `ngspice` is used to calculate the circuit output. The default value "analytic" uses an analytic diode bridge circuit.

noisy [boolean, optional] Whether or not the antenna should add noise to incoming signals.

unique_noise_waveforms [int, optional] The number of expected noise waveforms needed for each received signal to have its own noise.

Attributes

antenna [Antenna] *Antenna* object extended by the front end.

name [str] Name of the antenna.

position [array_like] Vector position of the antenna.

trigger_threshold [float] Threshold (V) for trigger condition. Antenna triggers if the voltage value of the waveform exceeds this value.

time_over_threshold [float] Time (s) that the voltage waveform must exceed *trigger_threshold* for the antenna to trigger.

envelope_amplification [float] Amplification to be applied to the signal pre-clipping. Note that the usual ARA electronics amplification is already applied without this.

envelope_method [str] String describing the circuit (and calculation method) to be used for envelope calculation.

is_hit Boolean of whether the antenna system has been triggered.

is_hit_mc_truth Boolean of whether the antenna has been triggered by signal.

signals The signals received by the antenna with front-end processing.

waveforms The antenna system signal + noise for each triggered hit.

all_waveforms The antenna system signal + noise for all hits.

Methods

<code>apply_response(signal[, direction, ...])</code>	Process the complete antenna response for an incoming signal.
<code>clear([reset_noise])</code>	Reset the antenna system to an empty state.
<code>envelopeless_front_end(signal)</code>	Apply front-end processes to a signal and return the output.
<code>front_end(signal)</code>	Apply front-end processes to a signal and return the output.
<code>full_waveform(times)</code>	Signal + noise for the antenna system for the given times.
<code>interpolate_filter(frequencies)</code>	Generate interpolated filter values for given frequencies.

Continued on next page

Table 104 – continued from previous page

<code>is_hit_during(times)</code>	Check if the antenna system is triggered in a time range.
<code>make_envelope(signal)</code>	Return the signal envelope based on the antenna's <code>envelope_method</code> .
<code>make_noise(times)</code>	Creates a noise signal over the given times.
<code>receive(signal[, direction, polarization, ...])</code>	Process and store one or more incoming (polarized) signals.
<code>set_orientation([z_axis, x_axis])</code>	Sets the orientation of the antenna system.
<code>setup_antenna(response_data[, ...])</code>	Setup the antenna by passing along its init arguments.
<code>trigger(signal)</code>	Check if the antenna triggers on a given signal.
<code>tunnel_diode(signal)</code>	Calculate a signal as processed by the tunnel diode.

pyrex.custom.irex.antenna.EnvelopeVpol

```
class pyrex.custom.irex.antenna.EnvelopeVpol (name, position, trigger_threshold,
                                             time_over_threshold=0, orientation=
                                             (0, 0, 1), amplification=1, amplifier_clipping=1,
                                             envelope_amplification=1,
                                             envelope_method='analytic', noisy=True,
                                             unique_noise_waveforms=10)
```

ARA Vpol (“bicone” or “birdcage”) antenna system with front-end processing.

Consists of an ARA Vpol antenna with typical responses, front-end electronics, and amplifier clipping, but with an additional amplification and envelope circuit applied after all other front-end processing.

Parameters

name [str] Name of the antenna.

position [array_like] Vector position of the antenna.

trigger_threshold [float] Threshold (V) for trigger condition. Antenna triggers if the voltage value of the waveform exceeds this value.

time_over_threshold [float, optional] Time (s) that the voltage waveform must exceed *trigger_threshold* for the antenna to trigger.

orientation [array_like, optional] Vector direction of the z-axis of the antenna.

amplification [float, optional] Amplification to be applied to the signal pre-clipping. Note that the usual ARA electronics amplification is already applied without this.

amplifier_clipping [float, optional] Voltage (V) above which the amplified signal is clipped (in positive and negative values).

envelope_amplification [float, optional] Amplification to be applied to the signal after the typical ARA front end, before the envelope circuit.

envelope_method [(('hilbert', 'analytic', 'spice') + ('basic', 'biased', 'doubler', 'bridge', 'log amp')), optional] String describing the circuit (and calculation method) to be used for envelope calculation. If the string contains “hilbert”, the hilbert envelope is used. If the string contains “analytic”, an analytic form is used to calculate the circuit output. If the string contains “spice”, `ngspice` is used to calculate the circuit output. The default value “analytic” uses an analytic diode bridge circuit.

noisy [boolean, optional] Whether or not the antenna should add noise to incoming signals.

unique_noise_waveforms [int, optional] The number of expected noise waveforms needed for each received signal to have its own noise.

Attributes

antenna [Antenna] Antenna object extended by the front end.

name [str] Name of the antenna.

position [array_like] Vector position of the antenna.

trigger_threshold [float] Threshold (V) for trigger condition. Antenna triggers if the voltage value of the waveform exceeds this value.

time_over_threshold [float] Time (s) that the voltage waveform must exceed *trigger_threshold* for the antenna to trigger.

envelope_amplification [float] Amplification to be applied to the signal pre-clipping. Note that the usual ARA electronics amplification is already applied without this.

envelope_method [str] String describing the circuit (and calculation method) to be used for envelope calculation.

is_hit Boolean of whether the antenna system has been triggered.

is_hit_mc_truth Boolean of whether the antenna has been triggered by signal.

signals The signals received by the antenna with front-end processing.

waveforms The antenna system signal + noise for each triggered hit.

all_waveforms The antenna system signal + noise for all hits.

Methods

<code>apply_response(signal[, direction, ...])</code>	Process the complete antenna response for an incoming signal.
<code>clear([reset_noise])</code>	Reset the antenna system to an empty state.
<code>envelopeless_front_end(signal)</code>	Apply front-end processes to a signal and return the output.
<code>front_end(signal)</code>	Apply front-end processes to a signal and return the output.
<code>full_waveform(times)</code>	Signal + noise for the antenna system for the given times.
<code>interpolate_filter(frequencies)</code>	Generate interpolated filter values for given frequencies.
<code>is_hit_during(times)</code>	Check if the antenna system is triggered in a time range.
<code>make_envelope(signal)</code>	Return the signal envelope based on the antenna's <code>envelope_method</code> .
<code>make_noise(times)</code>	Creates a noise signal over the given times.
<code>receive(signal[, direction, polarization, ...])</code>	Process and store one or more incoming (polarized) signals.
<code>set_orientation([z_axis, x_axis])</code>	Sets the orientation of the antenna system.
<code>setup_antenna(response_data[, ...])</code>	Setup the antenna by passing along its init arguments.
<code>trigger(signal)</code>	Check if the antenna triggers on a given signal.
<code>tunnel_diode(signal)</code>	Calculate a signal as processed by the tunnel diode.

Custom Detectors (`pyrex.custom.irex.detector`)

Module containing customized detector geometry classes for IREX.

Designed to be flexible such that stations can be built up from any string types and the detector grid can be made up of stations or strings.

<i>IREXString</i>	String of IREX Vpol antennas.
<i>RegularStation</i>	Station geometry with strings evenly spaced radially around the center.
<i>CoxeterStation</i>	Station geometry with center string and the rest evenly spaced radially.
<i>StationGrid</i>	Rectangular grid of stations or strings.

`pyrex.custom.irex.detector.IREXString`

```
class pyrex.custom.irex.detector.IREXString(x, y, antennas_per_string=2, antenna_separation=50, lowest_antenna=-100)
```

String of IREX Vpol antennas.

Sets the positions of antennas on string based on the parameters. Once the antennas have been built with `build_antennas`, the object can be directly iterated over to iterate over the antennas (as if the object were just a list of the antennas).

Parameters

x [float] Cartesian x-position (m) of the string.

y [float] Cartesian y-position (m) of the string.

antennas_per_string [float, optional] Total number of antennas to be placed on the string.

antenna_separation [float or list of float, optional] The vertical separation (m) of antennas on the string. If `float`, all antennas are separated by the same constant value. If `list`, the separations in the list are the separations of neighboring antennas starting from the lowest up to the highest.

lowest_antenna [float, optional] The Cartesian z-position (m) of the lowest antenna on the string.

Raises

ValueError If `test_antenna_positions` is `True` and an antenna is found to be above the ice surface.

See also:

`pyrex.custom.irex.EnvelopeHpol` ARA Hpol (“quad-slot”) antenna system with front-end processing.

`pyrex.custom.irex.EnvelopeVpol` ARA Vpol (“bicone” or “birdcage”) antenna system with front-end processing.

Notes

This class is designed to be the lowest subset level of a detector. It can (and should) be used for the subsets of some other `Detector` subclass to build up a full detector. Then when its “parent” is iterated, the instances of this class will be iterated as though they were all part of one flat list.

Attributes

- antenna_positions** [list] List (potentially with sub-lists) of the positions of the antennas generated by the `set_positions` method.
- subsets** [list] List of the antenna or detector objects which make up the detector.
- test_antenna_positions** [boolean] Class attribute for whether or not an error should be raised if antenna positions are found above the surface of the ice (where simulation behavior is ill-defined). Defaults to `True`.

Methods

<code>build_antennas(trigger_threshold[, ...])</code>	Creates antenna objects at the set antenna positions.
<code>clear([reset_noise])</code>	Reset the detector to an empty state.
<code>set_positions(x, y[, antennas_per_string, ...])</code>	Generates antenna positions along the string.
<code>triggered([antenna_requirement, ...])</code>	Check if the string is triggered based on its current state.

`pyrex.custom.irex.detector.RegularStation`

```
class pyrex.custom.irex.detector.RegularStation(x, y, strings_per_station=4, station_diameter=50, string_type=<class 'pyrex.custom.irex.detector.IREXString'>, **string_kwargs)
```

Station geometry with strings evenly spaced radially around the center.

Sets the positions of strings around the station based on the parameters. Supports any string type and passes extra keyword arguments on to the string class. Once the antennas have been built with `build_antennas`, the object can be directly iterated over to iterate over the antennas (as if the object were just a list of the antennas).

Parameters

- x** [float] Cartesian x-position (m) of the station.
- y** [float] Cartesian y-position (m) of the station.
- strings_per_station** [float, optional] Number of strings to be placed evenly around the station.
- station_diameter** [float, optional] Diameter (m) of the circle around which strings are placed.
- string_type** [optional] Class to be used for creating string objects for *subsets*.
- **string_kwargs** Keyword arguments to be passed on to the `__init__` methods of the *string_type* class.

Raises

- ValueError** If `test_antenna_positions` is `True` and an antenna is found to be above the ice surface.

See also:

`pyrex.custom.irex.EnvelopeHpol` ARA Hpol (“quad-slot”) antenna system with front-end processing.

`pyrex.custom.irex.EnvelopeVpol` ARA Vpol (“bicone” or “birdcage”) antenna system with front-end processing.

`IREXString` String of IREX Vpol antennas.

Notes

This class is designed to have string-like objects (which are subclasses of `Detector`) as its *subsets*. Then whenever an object of this class is iterated, all the antennas of its strings will be yielded as in a 1D list.

Attributes

`antenna_positions` [list] List (potentially with sub-lists) of the positions of the antennas generated by the `set_positions` method.

`subsets` [list] List of the antenna or detector objects which make up the detector.

`test_antenna_positions` [boolean] Class attribute for whether or not an error should be raised if antenna positions are found above the surface of the ice (where simulation behavior is ill-defined). Defaults to `True`.

Methods

<code>build_antennas(*args, **kwargs)</code>	Creates antenna objects at the set antenna positions.
<code>clear([reset_noise])</code>	Reset the detector to an empty state.
<code>set_positions(x, y[, strings_per_station, ...])</code>	Generates antenna positions around the station.
<code>triggered([antenna_requirement, ...])</code>	Check if the station is triggered based on its current state.

`pyrex.custom.irex.detector.CoxeterStation`

```
class pyrex.custom.irex.detector.CoxeterStation(x, y, strings_per_station=4, station_diameter=50, string_type=<class 'pyrex.custom.irex.detector.IREXString'>, **string_kwargs)
```

Station geometry with center string and the rest evenly spaced radially.

Sets the positions of strings around the station based on the parameters. Supports any string type and passes extra keyword arguments on to the string class. Once the antennas have been built with `build_antennas`, the object can be directly iterated over to iterate over the antennas (as if the object were just a list of the antennas).

Parameters

`x` [float] Cartesian x-position (m) of the station.

`y` [float] Cartesian y-position (m) of the station.

`strings_per_station` [float, optional] Number of strings to be placed around the station. Note that the first string is always placed at the center and the rest of the strings are placed evenly around that center string.

`station_diameter` [float, optional] Diameter (m) of the circle around which strings are placed.

`string_type` [optional] Class to be used for creating string objects for *subsets*.

****string_kwargs** Keyword arguments to be passed on to the `__init__` methods of the *string_type* class.

Raises

ValueError If `test_antenna_positions` is `True` and an antenna is found to be above the ice surface.

See also:

pyrex.custom.irex.EnvelopeHpol ARA Hpol (“quad-slot”) antenna system with front-end processing.

pyrex.custom.irex.EnvelopeVpol ARA Vpol (“bicone” or “birdcage”) antenna system with front-end processing.

IREXString String of IREX Vpol antennas.

Notes

This class is designed to have string-like objects (which are subclasses of *Detector*) as its *subsets*. Then whenever an object of this class is iterated, all the antennas of its strings will be yielded as in a 1D list.

Attributes

antenna_positions [list] List (potentially with sub-lists) of the positions of the antennas generated by the `set_positions` method.

subsets [list] List of the antenna or detector objects which make up the detector.

test_antenna_positions [boolean] Class attribute for whether or not an error should be raised if antenna positions are found above the surface of the ice (where simulation behavior is ill-defined). Defaults to `True`.

Methods

<code>build_antennas(*args, **kwargs)</code>	Creates antenna objects at the set antenna positions.
<code>clear([reset_noise])</code>	Reset the detector to an empty state.
<code>set_positions(x, y[, strings_per_station, ...])</code>	Generates antenna positions around the station.
<code>triggered([antenna_requirement, ...])</code>	Check if the station is triggered based on its current state.

pyrex.custom.irex.detector.StationGrid

```
class pyrex.custom.irex.detector.StationGrid (stations=1,          station_separation=500,
                                             station_type=<class
                                             'pyrex.custom.irex.detector.IREXString'>,
                                             **station_kwargs)
```

Rectangular grid of stations or strings.

Sets the positions of stations in a square layout if possible, otherwise in a rectangular layout (drops any extra stations). Supports any station type (including string types) and passes extra keyword arguments on to the station class. Once the antennas have been built with `build_antennas`, the object can be directly iterated over to iterate over the antennas (as if the object were just a list of the antennas).

Parameters

stations [float, optional] Number of stations to be placed.

station_separation [float, optional] Distance (m) between adjacent stations.

station_type [optional] Class to be used for creating station objects for *subsets*.

****station_kwargs** Keyword arguments to be passed on to the `__init__` methods of the *station_type* class.

Raises

ValueError If `test_antenna_positions` is `True` and an antenna is found to be above the ice surface.

Warning: If the number of *stations* provided does not divide nicely into a rectangle, extra stations may be dropped without warning. For example, if *stations* is 5, then a 2x2 grid will be created and the last station will be silently dropped.

See also:

IREXString String of IREX Vpol antennas.

RegularStation Station geometry with strings evenly spaced radially around the center.

CoxeterStation Station geometry with center string and the rest evenly spaced radially.

Notes

This class is designed to have station-like or string-like objects (which are subclasses of `Detector`) as its *subsets*. Then whenever an object of this class is iterated, all the antennas of its strings will be yielded as in a 1D list.

Attributes

antenna_positions [list] List (potentially with sub-lists) of the positions of the antennas generated by the `set_positions` method.

subsets [list] List of the antenna or detector objects which make up the detector.

test_antenna_positions [boolean] Class attribute for whether or not an error should be raised if antenna positions are found above the surface of the ice (where simulation behavior is ill-defined). Defaults to `True`.

Methods

<code>build_antennas(*args, **kwargs)</code>	Creates antenna objects at the set antenna positions.
<code>clear([reset_noise])</code>	Reset the detector to an empty state.
<code>set_positions([stations, ...])</code>	Generates antenna positions around the station.
<code>triggered([station_requirement, ...])</code>	Check if the detector is triggered based on its current state.

6.3.4 Layered Ice Models (`pyrex.custom.layered_ice`)

The layered ice module contains custom ice models and ray tracers needed for handling stratified ice layers.

The *LayeredIce* class is used to define ice models for individual layers of ice, as well as the depth ranges over which

these ice models are to be applied. The `LayeredRayTracer` class then takes two endpoints and a `LayeredIce` instance and returns all valid `LayeredRayTracePath` objects between those two endpoints. A maximum number of reflections is allowed between the layer boundaries, as specified by `LayeredRayTracer.max_reflections`.

Default Package Imports

<code>LayeredIce</code>	Class describing ice divided into multiple layers.
<code>LayeredRayTracer</code>	Class for calculating the ray solutions in layered ice.
<code>LayeredRayTracePath</code>	Class for representing a single ray solution in layered ice.

`pyrex.custom.layered_ice.LayeredIce`

class `pyrex.custom.layered_ice.LayeredIce` (*layers*, *index_above=1*, *index_below=None*)

Class describing ice divided into multiple layers.

Supports building layers made of any typical ice model. Most methods dispatch to the matching methods of the layers and combine the results appropriately. In all methods, the depth *z* should be given as a negative value if it is below the surface of the ice.

Parameters

layers [array_like] Array containing the ice models for each layer of the ice. Each ice model should have an `index` method for calculating the index of refraction at a depth as well as a `range` attribute describing the depth range of the ice layer.

index_above [float or None, optional] Index of refraction above the uppermost ice layer. If *None*, uses the index of refraction at the uppermost boundary.

index_below [float or None, optional] Index of refraction below the lowermost ice layer. If *None*, uses the index of refraction at the lowermost boundary.

Attributes

layers [list] List of ice models for each layer of the ice.

index_above [float or None, optional] The index of refraction above the ice's valid range.

index_below [float or None, optional] The index of refraction below the ice's valid range.

Methods

<code>contains(point)</code>	Determines if the given point is within the ice's valid range.
<code>depth_with_index(n)</code>	Calculates the corresponding depth for a given index of refraction.
<code>index(z)</code>	Calculates the index of refraction of the ice at a given depth.
<code>layer_at_depth(z)</code>	Determines the layer of ice at a given depth.

`pyrex.custom.layered_ice.LayeredRayTracer`

class `pyrex.custom.layered_ice.LayeredRayTracer` (*from_point*, *to_point*, *ice_model*)

Class for calculating the ray solutions in layered ice.

Calculates paths among the ice layers up to a maximum number of allowed reflections. Most properties are lazily evaluated to save on computation time. If any attributes of the class instance are changed, the lazily-evaluated properties will be cleared.

Parameters

from_point [array_like] Vector starting point of the ray path.

to_point [array_like] Vector ending point of the ray path.

ice_model The ice model used for the ray tracer.

See also:

`pyrex.internal_functions.LazyMutableClass` Class with lazy properties which may depend on other class attributes.

`LayeredRayTracePath` Class for representing a single ray solution in layered ice.

Notes

Even more attributes than those listed are available for the class, but are mainly for internal use. These attributes can be found by exploring the source code.

Attributes

from_point [ndarray] The starting point of the ray path.

to_point [ndarray] The ending point of the ray path.

ice The ice model used for the ray tracer.

max_reflections [int] The maximum number of reflections allowed in a solution path.

solution_class Class for representing a single ray solution in layered ice.

exists

solutions

Methods

<code>solution_class</code>	alias of <code>LayeredRayTracePath</code>
-----------------------------	---

<code>solution_sorting</code>	
-------------------------------	--

pyrex.custom.layered_ice.LayeredRayTracePath

class `pyrex.custom.layered_ice.LayeredRayTracePath` (*parent_tracer, paths*)

Class for representing a single ray solution in layered ice.

Stores parameters of the ray path along the layers of ice traversed. Most methods dispatch to the matching methods of the layers and combine the results appropriately. Most properties are lazily evaluated to save on computation time. If any attributes of the class instance are changed, the lazily-evaluated properties will be cleared.

Parameters

parent_tracer [LayeredRayTracer] Ray tracer for which this path is a solution.

paths [array_like] Array of path objects corresponding to the ice layers in the order of traversal.

See also:

pyrex.internal_functions.LazyMutableClass Class with lazy properties which may depend on other class attributes.

LayeredRayTracer Class for calculating the ray solutions in layered ice.

Notes

Even more attributes than those listed are available for the class, but are mainly for internal use. These attributes can be found by exploring the source code.

Attributes

from_point [ndarray] The starting point of the ray path.

to_point [ndarray] The ending point of the ray path.

ice The ice model used for the ray tracer.

emitted_direction

received_direction

path_length

tof

coordinates

Methods

<code>attenuation(f, *args, **kwargs)</code>	Calculate the attenuation factor for signal frequencies.
<code>propagate([signal, polarization])</code>	Propagate the signal with optional polarization along the ray path.

Individual Module APIs

Custom Ice Models (`pyrex.custom.layered_ice.ice_model`)

Module containing ice model classes.

Includes ice models for layered ice.

<i>LayeredIce</i>	Class describing ice divided into multiple layers.
-------------------	--

`pyrex.custom.layered_ice.ice_model.LayeredIce`

class `pyrex.custom.layered_ice.ice_model.LayeredIce` (*layers*, *index_above=1*, *index_below=None*)

Class describing ice divided into multiple layers.

Supports building layers made of any typical ice model. Most methods dispatch to the matching methods of the layers and combine the results appropriately. In all methods, the depth z should be given as a negative value if it is below the surface of the ice.

Parameters

layers [array_like] Array containing the ice models for each layer of the ice. Each ice model should have an `index` method for calculating the index of refraction at a depth as well as a `range` attribute describing the depth range of the ice layer.

index_above [float or None, optional] Index of refraction above the uppermost ice layer. If *None*, uses the index of refraction at the uppermost boundary.

index_below [float or None, optional] Index of refraction below the lowermost ice layer. If *None*, uses the index of refraction at the lowermost boundary.

Attributes

layers [list] List of ice models for each layer of the ice.

index_above [float or None, optional] The index of refraction above the ice's valid range.

index_below [float or None, optional] The index of refraction below the ice's valid range.

Methods

<code>contains(point)</code>	Determines if the given point is within the ice's valid range.
<code>depth_with_index(n)</code>	Calculates the corresponding depth for a given index of refraction.
<code>index(z)</code>	Calculates the index of refraction of the ice at a given depth.
<code>layer_at_depth(z)</code>	Determines the layer of ice at a given depth.

Custom Ray Tracing (`pyrex.custom.layered_ice.ray_tracing`)

Module containing classes for ray tracing through layered ice.

Supports layers of ice where the index is monotonic within the layer's valid range.

<code>LayeredRayTracePath</code>	Class for representing a single ray solution in layered ice.
<code>LayeredRayTracer</code>	Class for calculating the ray solutions in layered ice.

`pyrex.custom.layered_ice.ray_tracing.LayeredRayTracePath`

class `pyrex.custom.layered_ice.ray_tracing.LayeredRayTracePath` (*parent_tracer*, *paths*)

Class for representing a single ray solution in layered ice.

Stores parameters of the ray path along the layers of ice traversed. Most methods dispatch to the matching methods of the layers and combine the results appropriately. Most properties are lazily evaluated to save on computation time. If any attributes of the class instance are changed, the lazily-evaluated properties will be cleared.

Parameters

parent_tracer [LayeredRayTracer] Ray tracer for which this path is a solution.

paths [array_like] Array of path objects corresponding to the ice layers in the order of traversal.

See also:

`pyrex.internal_functions.LazyMutableClass` Class with lazy properties which may depend on other class attributes.

`LayeredRayTracer` Class for calculating the ray solutions in layered ice.

Notes

Even more attributes than those listed are available for the class, but are mainly for internal use. These attributes can be found by exploring the source code.

Attributes

from_point [ndarray] The starting point of the ray path.

to_point [ndarray] The ending point of the ray path.

ice The ice model used for the ray tracer.

emitted_direction

received_direction

path_length

tof

coordinates

Methods

<code>attenuation(f, *args, **kwargs)</code>	Calculate the attenuation factor for signal frequencies.
<code>propagate([signal, polarization])</code>	Propagate the signal with optional polarization along the ray path.

`pyrex.custom.layered_ice.ray_tracing.LayeredRayTracer`

class `pyrex.custom.layered_ice.ray_tracing.LayeredRayTracer` (*from_point, to_point, ice_model*)

Class for calculating the ray solutions in layered ice.

Calculates paths among the ice layers up to a maximum number of allowed reflections. Most properties are lazily evaluated to save on computation time. If any attributes of the class instance are changed, the lazily-evaluated properties will be cleared.

Parameters

from_point [array_like] Vector starting point of the ray path.

to_point [array_like] Vector ending point of the ray path.

ice_model The ice model used for the ray tracer.

See also:

`pyrex.internal_functions.LazyMutableClass` Class with lazy properties which may depend on other class attributes.

`LayeredRayTracePath` Class for representing a single ray solution in layered ice.

Notes

Even more attributes than those listed are available for the class, but are mainly for internal use. These attributes can be found by exploring the source code.

Attributes

`from_point` [ndarray] The starting point of the ray path.

`to_point` [ndarray] The ending point of the ray path.

`ice` The ice model used for the ray tracer.

`max_reflections` [int] The maximum number of reflections allowed in a solution path.

`solution_class` Class for representing a single ray solution in layered ice.

`exists`

`solutions`

Methods

<code>solution_class</code>	alias of <code>LayeredRayTracePath</code>
-----------------------------	---

<code>solution_sorting</code>	
-------------------------------	--

VERSION HISTORY

7.1 Version 1.9.0

New Features

- Added a `GreenlandIce` ice model based on measurements at Summit Station.
- Added the `AVZAskaryanSignal` Askaryan model from J. Alvarez-Muniz et al (2000) for comparison across simulation packages.
- Added `UniformIce` ice model with a constant index of refraction, along with the corresponding ray tracing with `UniformRayTracer` and `UniformRayTracePath`.
- Added `pyrex.custom.layered_ice` for ice models with indices of refraction defined in stratified layers in the z-direction and the corresponding ray tracing.
- Added `complex_interp()` and `complex_bilinear_interp()` functions for interpolating arrays of complex values using Cartesian or Euler methods.

Changes

- `IceModel` has been deprecated in favor of the `ice` object, an object of the preferred ice model class (currently `AntarcticIce`).
- Removed the `NewcombIce` ice model.
- Handling of s-polarized and p-polarized signals by the `RayTracer` have now been split. This resulted in different behavior for the `RayTracePath.propagate()` and `Antenna.receive()` methods, as well as a new method `Antenna.apply_response()`.
- Renamed `Antenna.response()` method to `Antenna.frequency_response()`.
- ARA antennas have been updated to use the latest XFDTD simulations from the Chiba group.
- The behavior of the `CylindricalShadowGenerator` and `RectangularShadowGenerator` classes have been moved into the `CylindricalGenerator` and `RectangularGenerator` classes, respectively, when the `shadow` argument is set to `True`.

Bug Fixes

- The handling of phase shifts between s-polarized and p-polarized signals during total internal reflection is now properly implemented.
- Fixed a bug in the antenna response of ARA antennas which resulted in an extra factor of $\sin(\theta)$ for `VpolAntenna` objects and had an unexplored effect on `HpolAntenna` objects.

- Fixed an off-by-one error in the final event count given by *ListGenerator* objects.
- Fixed a bug in the Earth shadowing effect produced by *CylindricalGenerator* and *RectangularGenerator* classes.

7.2 Version 1.8.2

New Features

- Added *CylindricalGenerator*, *RectangularGenerator*, *CylindricalShadowGenerator*, and *RectangularShadowGenerator* classes to provide options for generation volumes and how to account for shadowing by the Earth. *RectangularShadowGenerator* has the same behavior as the existing *ShadowGenerator*, so *ShadowGenerator* is being deprecated.
- Added ability to add *Detector* (and *Antenna* or *AntennaSystem*) objects into a *CombinedDetector* for ease of use.
- Added ability to multiply (and divide) *Signal* objects by numeric types.
- Added support for total events thrown in simulation, accessed by the *File.total_events_thrown* attribute of file readers.

Changes

- Separated *Particle.weight* into *Particle.survival_weight* and *Particle.interaction_weight*. Now *Particle.weight* serves as a convenience attribute which gives the product of the two weights.
- Changed *FileGenerator* to read from simulation output files rather than numpy files. For the time being numpy files can be read with *NumpyFileGenerator*.
- All generator classes now have a *count* attribute for keeping track of the total number of events thrown.

Bug Fixes

- Fixed error in *ARAAntenna* signal amplitudes introduced in version 1.8.1.
- Fixed minor bugs in *File* interfaces.

Performance Improvements

- Changed *AskaryanSignal* charge profile and RAC calculations to accept numpy arrays. Should result in marginal improvements in signal calculation speed.

7.3 Version 1.8.1

New Features

- Added ability to write (and subsequently read) simulation data files using *File* objects.
- File I/O supports HDF5 files, but should be considered to be in a public-beta state until the release of version 1.9.0.

- `EventKernel` now accepts `event_writer` and `triggers` arguments for writing simulation data to output files.

Changes

- `ThermalNoise` now uses Rayleigh-distributed amplitudes in frequency space by default.
- Handling of signal polarizations has been more closely integrated with the ray tracer; `RayTracer.propagate()` now propagating the polarization vector as well as the signal.
- 3 dB splitter effect moved from `ARAAntenna.response()` to `ARAAntennaSystem.front_end()` for a more logical separation of antenna and front-end.
- Adjusted default noise rms of `ARIANNAAntennaSystem` to the expected value in ARIANNA.

Bug Fixes

- Corrected signal polarization calculation.
- Fixed calculation of fresnel factors in surface reflection.
- Fixed bug in antenna gains of asymmetric antennas for theta angles near 180 degrees.
- Corrected effective height of antennas modeled by WIPL-D (i.e. LPDA).

7.4 Version 1.8.0

New Features

- Added model of the ARIANNA LPDA based primarily on the implementation in `NuRadioReco`.
- Added `Antenna.is_hit_mc` and `AntennaSystem.is_hit_mc` which test noise-only triggers to determine whether a triggered antenna as truly triggered by signal or not.
- Added `require_mc_truth` argument to `Detector.triggered()` to toggle whether a true Monte Carlo signal trigger (described above with `Antenna.is_hit_mc`) is required for a detector trigger.
- Added `AntennaSystem.lead_in_time` which allows front-end systems time to equilibrate before waveforms are recorded.

Changes

- `Antenna.waveforms` and `Antenna.all_waveforms` now include all relevant signals in the waveform during that time, similar to `Antenna.full_waveform()`.
- `ARAAntenna.interpolate_filter()` moved to `ARAAntennaSystem.interpolate_filter()`, since this better matches the logical location of the front-end electronics.

Bug Fixes

- Fixed error in calculation of ARA Hpol polarization gain.
- Corrected amplification of `ARAAntennaSystem` (previously was silently ignored).
- Corrected tunnel diode and other triggers to use standard deviation from mean rather than rms.

- Fixed accidental duplication of antennas when `Detector.build_antennas()` is called more than once.
- Fixed numerical issue when checking that antenna axes are perpendicular.

7.5 Version 1.7.0

New Features

- Moved `pyrex.custom.ara` module into main PyREx package instead of being a plug-in.
- All docstrings now follow numpy docstring style.
- Added particle types and interaction information to *Particle* class.
- Added Interaction classes *GQRSInteraction* and *CTWInteraction* for defining different neutrino interaction models. Preferred model (*CTWInteraction*) aliased to *NeutrinoInteraction*.
- Added `ShadowGenerator.get_vertex()`, `ShadowGenerator.get_direction()`, `ShadowGenerator.get_particle_type()`, `ShadowGenerator.get_exit_points()`, and `ShadowGenerator.get_weight()` methods for generating neutrinos more modularly.
- Added *Event* class for holding a tree of *Particle* objects. *Event* objects are now returned by generators and the *EventKernel*.
- Added *ZHAskaryanSignal* class for the Zas, Halzen, Stanev parameterization of Askaryan pulses. Mostly for comparison purposes.

Changes

- `ShadowGenerator.create_particle()` changed to `ShadowGenerator.create_event()` and now returns an *Event* object.
- Generator classes moved to `pyrex.generation` module.
- `Signal.ValueTypes` changed to `Signal.Type` to match `Particle.Type` and `Interaction.Type`.
- `FastAskaryanSignal` changed to `ARVZAskaryanSignal`. This class is still the preferred parameterization aliased to *AskaryanSignal*.
- Arguments of *AskaryanSignal* changed to take a *Particle* object rather than taking its parameters individually.
- Removed unused `SlowAskaryanSignal`.
- Now that *AskaryanSignal* can handle different particle and shower types, secondary particle generation was added to determine shower fractions: `NeutrinoInteraction.em_frac` and `NeutrinoInteraction.had_frac`.
- Changed IREX envelope antennas to be an envelope front-end on top of an ARA antenna. Results in `IREXAntennaSystem` becoming `EnvelopeHpol` and `EnvelopeVpol`.

7.6 Version 1.6.0

New Features

- `EventKernel` can now take arguments to specify the ray tracer to be used and the times array to be used in signal generation.
- Added shell scripts to more easily work with git branching model.

Changes

- `ShadowGenerator` `energy_generator` argument changed to `energy` and can now take a function or a scalar value, in which case all particles will have that scalar value for their energy.
- `EventKernel` now uses `pyrex.IceModel` as its ice model by default.
- `Antenna.receive()` method (and `receive()` method of all inheriting antennas) now uses `direction` argument instead of `origin` argument to calculate directional gain.
- `Antenna.clear()` and `Detector.clear()` functions can now optionally reset the noise calculation by using the `reset_noise` argument.
- `Antenna` classes can now set the `unique_noise_waveforms` argument to specify the expected number of unique noise waveforms needed.
- `ArasimIce.attenuation_length()` changed to more closely match `AraSim`.
- `IceModel` reverted to `AntarcticIce` with new index of refraction coefficients matching those of `ArasimIce`.
- `prem_density()` can now be calculated for an array of radii.

Performance Improvements

- Improved performance of `slant_depth()` calculation.
- Improved performance of `IceModel.attenuation_length()` calculation.
- Using the `Antenna` `unique_noise_waveforms` argument can improve noise waveform calculation speed (previously assumed 100 unique waveforms were necessary).

Bug Fixes

- Fixed received direction bug in `EventKernel`, which had still been assuming a straight-ray path.
- Lists in function keyword arguments were changed to tuples to prevent unexpected mutability issues.
- Fixed potential errors in `BasicRayTracer` and `BasicRayTracePath`.

7.7 Version 1.5.0

Changes

- Changed structure of `Detector` class so a detector can be built up from strings to stations to the full detector.
- `Detector.antennas` attribute changed to `Detector.subsets`, which contains the pieces which make up the detector (e.g. antennas on a string, strings in a station).
- Iterating the `Detector` class directly retains its effect of iterating each antenna in the detector directly.

New Features

- Added :meth:`Detector.triggered` and `Detector.clear()` methods.
- Added two new neutrino generators *ListGenerator* and *FileGenerator* designed to pull pre-generated *Particle* objects.

Bug Fixes

- Preserve `value_type` of *Signal* objects passed to `IREXAntennaSystem.front_end()`.

7.8 Version 1.4.2

Performance Improvements

- Improved performance of *FastAskaryanSignal* by reducing the size of the convolution.

Changes

- Adjusted time step of signals generated by kernel slightly (2000 steps instead of 2048).

7.9 Version 1.4.1

Changes

- Improved ray tracing and defaulted to the almost completely analytical *SpecializedRayTracer* and *SpecializedRayTracePath* classes as *RayTracer* and *RayTracePath*.
- Added ray tracer into *EventKernel* to replace *PathFinder* completely.

7.10 Version 1.4.0

New Features

- Implemented full ray tracing in the *RayTracer* and *RayTracePath* classes.

7.11 Version 1.3.1

New Features

- Added diode bridge rectifier envelope circuit analytic model to `irex.frontends` and made it the default analytic envelope model in *IREXAntennaSystem*.
- Added `allow_reflection` attribute to *EventKernel* class to determine whether *ReflectedPathFinder* solutions should be allowed.

Changes

- Changed neutrino interaction model to include all neutrino and anti-neutrino interactions rather than only charged-current neutrino (relevant for `ShadowGenerator` class).

7.12 Version 1.3.0

New Features

- Added and implemented `ReflectedPathFinder` class for rays which undergo total internal reflection and subsequently reach an antenna.

Changes

- Change `AskaryanSignal` angle to always be positive and remove < 90 degree restriction (Alvarez-Muniz, Romero-Wolf, & Zas paper suggests the algorithm should work for all angles).

Performance Improvements

- Improve performance of ice index calculated at many depths.

7.13 Version 1.2.1

New Features

- Added `Antenna.set_orientation()` method for setting the `z_axis` and `x_axis` attributes appropriately.

Bug Fixes

- Fixed bug where `Antenna._convert_to_antenna_coordinates()` function was returning coordinates relative to (0,0,0) rather than the antenna's position.

7.14 Version 1.2.0

Changes

- Changed `custom` module to a package containing `irex` module.
- `custom` package leverages “Implicit Namespace Package” structure to allow plug-in style additions to the package in either the user's `~/pyrex-custom/` directory or the `./pyrex-custom` directory.

7.15 Version 1.1.2

New Features

- Added `Signal.with_times()` method for interpolation/extrapolation of signals to different times.
- Added `Antenna.full_waveform()` and `Antenna.is_hit_during()` methods for calculation of waveform over arbitrary time array and whether said waveform triggers the antenna, respectively.
- Added `IREXAntenna.front_end_processing()` method for processing envelope, amplifying signal, and downsampling result (downsampling currently inactive).

Changes

- Change `Antenna.make_noise()` to use a single master noise object and use `ThermalNoise.with_times()` to calculate noise at different times.
 - To ensure noise is not obviously periodic (for <100 signals), uses 100 times the recommended number of frequencies, which results in longer computation time for noise waveforms.

7.16 Version 1.1.1

Changes

- Moved `ValueTypes` inside `Signal` class. Now access as `Signal.ValueTypes.voltage`, etc.
- Changed signal envelope calculation in custom `IREXAntenna` from hilbert transform to a basic model. Spice model also available, but slower.

7.17 Version 1.1.0

New Features

- Added `Antenna.directional_gain()` and `Antenna.polarization_gain()` methods to base `Antenna`.
 - `Antenna.receive()` method should no longer be overwritten in most cases.
 - `Antenna` now has orientation defined by `z_axis` and `x_axis`.
 - `antenna_factor` and `efficiency` attributes added to `Antenna` for more flexibility.
- Added `value_type` attribute to `Signal` class and derived classes.
 - Current value types are `ValueTypes.undefined`, `ValueTypes.voltage`, `ValueTypes.field`, and `ValueTypes.power`.
 - `Signal` objects now must have the same `value_type` to be added (though those with `ValueTypes.undefined` can be coerced).

Changes

- Made units consistent across PyREx.
- Added ability to define *Antenna* noise by RMS voltage rather than temperature and resistance if desired.
- Allow *DipoleAntenna* to guess at `effective_height` if not specified.

Performance Improvements

- Increase speed of `IceModel.__atten_coeffs()` method, resulting in increased speed of attenuation length calculations.

7.18 Version 1.0.3

New Features

- Added `custom` module to contain classes and functions specific to the IREX project.

7.19 Version 1.0.2

New Features

- Added `Antenna.make_noise()` method so custom antennas can use their own noise functions.

Changes

- Allow passing of numpy arrays of depths and frequencies into most `IceModel` methods.
 - `IceModel.gradient()` must still be calculated at individual depths.
- Added ability to specify RMS voltage of *ThermalNoise* without providing temperature and resistance.
- Removed (deprecated) `Antenna.isHit()`.

Performance Improvements

- Allowing for `IceModel` to calculate many attenuation lengths at once improves speed of `PathFinder.propagate()`.
- Improved speed of `PathFinder.time_of_flight()` and `PathFinder.attenuation()` (and improved accuracy to boot).

7.20 Version 1.0.1

Changes

- Changed *Antenna* to not require a temperature and frequency range if no noise is produced.

Bug Fixes

- Fixed bugs in *AskaryanSignal* that caused the convolution to fail.
- Fixed bugs resulting from converting `IceModel.temperature()` from Celsius to Kelvin.

7.21 Version 1.0.0

- Created PyREx package based on original notebook.
- Added all signal classes to produce full-waveform Askaryan pulses and thermal noise.
- Changed *Antenna* class to *DipoleAntenna* to allow *Antenna* to be a base class.
- Changed `Antenna.isHit()` method to `Antenna.is_hit` property.
- Introduced `IceModel` alias for `AntarcticIce` (or any future preferred ice model).
- Moved `AntarcticIce.attenuationLengthMN()` to its own `NewcombIce` class inheriting from `AntarcticIce`.
- Added `PathFinder.propagate()` to propagate a *Signal* object in a customizable way.
- Changed naming conventions to be more consistent, verbose, and “pythonic”:
 - `AntarcticIce.attenuationLength()` becomes `AntarcticIce.attenuation_length()`.
 - In `pyrex.earth_model`, `RE` becomes `EARTH_RADIUS`.
 - In `pyrex.particle`, `neutrino_interaction` becomes *NeutrinoInteraction*.
 - In `pyrex.particle`, `NA` becomes `AVOGADRO_NUMBER`.
 - `particle` class becomes *Particle* namedtuple.
 - `Particle.vtx` becomes `Particle.vertex`.
 - `Particle.dir` becomes `Particle.direction`.
 - `Particle.E` becomes `Particle.energy`.
 - In `pyrex.particle`, `next_direction()` becomes `random_direction()`.
 - `shadow_generator` becomes `ShadowGenerator`.
 - `PathFinder.exists()` method becomes `PathFinder.exists` property.
 - `PathFinder.getEmittedRay()` method becomes `PathFinder.emitted_ray` property.
 - `PathFinder.getPathLength()` method becomes `PathFinder.path_length` property.
 - `PathFinder.propagateRay()` split into `PathFinder.time_of_flight()` (with corresponding `PathFinder.tof` property) and `PathFinder.attenuation()`.

7.22 Version 0.0.0

Original PyREx python notebook written by Kael Hanson:

<https://gist.github.com/physkael/898a64e6fbf5f0917584c6d31edf7940>

GITHUB README

8.1 PyREx - (Python package for Radio Experiments)

PyREx (**P**ython **p**ackage for **R**adio **E**xperiments) is a Python package designed to simulate the measurement of Askaryan pulses via in-ice radio antenna arrays. The code was written for the ARA collaboration with considerations for future radio arrays. As such, the package is designed to be highly modular so that it can easily be used for other radio projects (e.g. ARIANNA, RNO, and IceCube Gen2).

8.1.1 Useful Links

- Source (GitHub): <https://github.com/bhokansonfasig/pyrex/>
- Documentation: <https://bhokansonfasig.github.io/pyrex/>
- Release notes: <https://bhokansonfasig.github.io/pyrex/build/html/versions.html>

8.1.2 Getting Started

Requirements

PyREx requires python version 3.6+ as well as numpy version 1.13+, scipy version 0.19+, and h5py version 2.7+. After installing python from <https://www.python.org/downloads/>, the required packages can be installed with `pip` as follows, or they will be installed automatically by simply installing `pyrex` as specified in the next section.

```
pip install numpy>=1.13
pip install scipy>=0.19
pip install h5py>=2.7
```

Installing

The easiest way to get the PyREx package is using `pip` as follows:

```
pip install git+https://github.com/bhokansonfasig/pyrex#egg=pyrex
```

Note that since PyREx is not currently available on PyPI, a simple `pip install pyrex` will not have the intended effect.

Optional Dependencies

The following packages are not required for running PyREx by default, but may be useful or required for running some specific parts of the code:

matplotlib

Recommended version: 2.1+

Used for creating plots in example code and auxiliary scripts.

sphinx and numpydoc

Recommended version: 2.1+, 0.9+

Required for building documentation. Only needed by documentation maintainers and release handlers.

PySpice

Recommended version: 1.1

Used by IREX sub-package for some complex front-end circuits. Not needed for default front-ends.

8.1.3 Examples

For examples of how to use PyREx, see the [usage page](#) and the [examples page](#) in the documentation, or the python notebooks in the [examples](#) directory.

8.1.4 Contributing

Contributions to the code base are mostly handled through pull requests on GitHub. Before contributing, please read the [contribution page](#) in the documentation for more information.

8.1.5 Authors

- Ben Hokanson-Fasig

8.1.6 License

[MIT License](#)

Copyright (c) 2019 Ben Hokanson-Fasig

BIBLIOGRAPHY

- [1] A. Connolly et al, ANITA Note #76, “Thermal Noise Studies: Toward A Time-Domain Model of the ANITA Trigger.” https://www.phys.hawaii.edu/eelog/anita_notes/060228_110754/noise_simulation.ps
- [1] M. Newcomb (2013) <http://icecube.wisc.edu/~araproject/radio/>
- [1] A. Dziewonski & D. Anderson, “Preliminary reference Earth model.” *Physics of the Earth and Planetary Interiors* **25**, 297–356 (1981). DOI:10.1016/0031-9201(81)90046-7
- [1] A. Connolly et al, ANITA Note #76, “Thermal Noise Studies: Toward A Time-Domain Model of the ANITA Trigger.” https://www.phys.hawaii.edu/eelog/anita_notes/060228_110754/noise_simulation.ps
- [1] E. Zas, F. Halzen, T. Stanev, “Electromagnetic pulses from high-energy showers: implications for neutrino detection”, *Physical Review D* **45**, 362-376 (1992). DOI:10.1103/PhysRevD.45.362
- [1] J. Alvarez-Muniz et al, “Calculation Methods for Radio Pulses from High Energy Showers.” *Physical Review D* **62**, 063001 (2000). [arXiv:astro-ph/0003315](https://arxiv.org/abs/astro-ph/0003315) DOI:10.1103/PhysRevD.62.063001
- [2] J. Alvarez-Muniz & E. Zas, “The LPM effect for EeV hadronic showers in ice: implications for radio detection of neutrinos.” *Physics Letters B* **434**, 396-406 (1998). [arXiv:astro-ph/9806098](https://arxiv.org/abs/astro-ph/9806098) DOI:10.1016/S0370-2693(98)00905-8
- [1] J. Alvarez-Muniz et al, “Practical and accurate calculations of Askaryan radiation.” *Physical Review D* **84**, 103003 (2011). [arXiv:1106.6283](https://arxiv.org/abs/1106.6283) DOI:10.1103/PhysRevD.84.103003
- [2] K. Greisen, “The Extensive Air Showers.” *Prog. in Cosmic Ray Phys.* **III**, 1 (1956).
- [3] K.D. de Vries et al, “On the feasibility of RADAR detection of high-energy neutrino-induced showers in ice.” *Astropart. Phys.* **60**, 25-31 (2015). [arXiv:1312.4331](https://arxiv.org/abs/1312.4331) DOI:10.1016/j.astropartphys.2014.05.009
- [4] T.K. Gaisser & A.M. Hillas “Reliability of the Method of Constant Intensity Cuts for Reconstructing the Average Development of Vertical Showers.” *ICRC proceedings*, 353 (1977).
- [5] J. Alvarez-Muniz & E. Zas, “EeV Hadronic Showers in Ice: The LPM effect.” *ICRC proceedings*, 17-25 (1999). [arXiv:astro-ph/9906347](https://arxiv.org/abs/astro-ph/9906347)
- [1] A. Dziewonski & D. Anderson, “Preliminary reference Earth model.” *Physics of the Earth and Planetary Interiors* **25**, 297–356 (1981). DOI:10.1016/0031-9201(81)90046-7
- [1] M. Newcomb (2013) <http://icecube.wisc.edu/~araproject/radio/>
- [1] C. Deaconu et al, “Measurements and modeling of near-surface radio propagation in glacial ice and implications for neutrino experiments.” *Physical Review D* **98**, 043010 (2018). [arXiv:1805.12576](https://arxiv.org/abs/1805.12576) DOI:10.1103/PhysRevD.98.043010
- [2] J. Avva et al, “An in Situ Measurement of the Radio-Frequency Attenuation in Ice at Summit Station, Greenland.” *Journal of Glaciology* **61**, no. 229, 1005-1011 (2015). DOI:10.3189/2015JoG15J057
- [1] R. Gandhi et al, “Ultrahigh-Energy Neutrino Interactions.” *Physical Review D* **58**, 093009 (1998). DOI:10.1103/PhysRevD.58.093009

- [1] A. Connolly et al, “Calculation of High Energy Neutrino-Nucleon Cross Sections and Uncertainties Using the MSTW Parton Distribution Functions and Implications for Future Experiments.” *Physical Review D* **83**, 113009 (2011). [arXiv:1102.0691](#) DOI:10.1103/PhysRevD.83.113009

PYTHON MODULE INDEX

p

- `pyrex.antenna`, 84
- `pyrex.askaryan`, 80
- `pyrex.custom.ara.antenna`, 130
- `pyrex.custom.ara.detector`, 138
- `pyrex.custom.arianna.antenna`, 147
- `pyrex.custom.irex.antenna`, 163
- `pyrex.custom.irex.detector`, 171
- `pyrex.custom.irex.frontends`, 162
- `pyrex.custom.layered_ice.ice_model`, 178
- `pyrex.custom.layered_ice.ray_tracing`, 179
- `pyrex.detector`, 88
- `pyrex.earth_model`, 91
- `pyrex.generation`, 108
- `pyrex.ice_model`, 92
- `pyrex.internal_functions`, 68
- `pyrex.io`, 117
- `pyrex.kernel`, 115
- `pyrex.particle`, 103
- `pyrex.ray_tracing`, 96
- `pyrex.signals`, 74

Symbols

`_read_amplifier_data()` (in module *pyrex.custom.arianna.antenna*), 148
`_read_arasim_antenna_data()` (in module *pyrex.custom.ara.antenna*), 131
`_read_filter_data()` (in module *pyrex.custom.ara.antenna*), 131
`_read_response_data()` (in module *pyrex.custom.arianna.antenna*), 147

A

AlbrechtStation (class in *pyrex.custom.ara*), 128
AlbrechtStation (class in *pyrex.custom.ara.detector*), 142
AntarcticIce (class in *pyrex.ice_model*), 92
Antenna (class in *pyrex*), 54
Antenna (class in *pyrex.antenna*), 85
AntennaSystem (class in *pyrex*), 57
AntennaSystem (class in *pyrex.detector*), 88
ARAAntenna (class in *pyrex.custom.ara.antenna*), 131
ARAAntennaSystem (class in *pyrex.custom.ara.antenna*), 133
ArasimIce (class in *pyrex.ice_model*), 94
ARAStrng (class in *pyrex.custom.ara*), 124
ARAStrng (class in *pyrex.custom.ara.detector*), 139
ARIANNAAntenna (class in *pyrex.custom.arianna.antenna*), 148
ARIANNAAntennaSystem (class in *pyrex.custom.arianna.antenna*), 150
ARVZaskaryanSignal (class in *pyrex.askaryan*), 83
AskaryanSignal (in module *pyrex*), 52
AskaryanSignal (in module *pyrex.askaryan*), 84
AVZaskaryanSignal (class in *pyrex.askaryan*), 81

B

BaseGenerator (class in *pyrex.generation*), 108
`basic_envelope_model()` (in module *pyrex.custom.irex.frontends*), 162
BasicRayTracePath (class in *pyrex.ray_tracing*), 97
BasicRayTracer (class in *pyrex.ray_tracing*), 99
`bridge_rectifier_envelope_model()` (in module *pyrex.custom.irex.frontends*), 162

C

CombinedDetector (class in *pyrex.detector*), 90
`complex_bilinear_interp()` (in module *pyrex.internal_functions*), 71
`complex_interp()` (in module *pyrex.internal_functions*), 70
`convert_hex_coords()` (in module *pyrex.custom.ara.detector*), 138
CoxeterStation (class in *pyrex.custom.irex*), 159
CoxeterStation (class in *pyrex.custom.irex.detector*), 173
CTWInteraction (class in *pyrex.particle*), 107
CylindricalGenerator (class in *pyrex*), 62
CylindricalGenerator (class in *pyrex.generation*), 109

D

Detector (class in *pyrex*), 58
Detector (class in *pyrex.detector*), 89
DipoleAntenna (class in *pyrex*), 55
DipoleAntenna (class in *pyrex.antenna*), 86
DipoleTester (class in *pyrex.custom.irex.antenna*), 163

E

EmptySignal (class in *pyrex*), 50
EmptySignal (class in *pyrex.signals*), 76
EnvelopeHpol (class in *pyrex.custom.irex*), 154
EnvelopeHpol (class in *pyrex.custom.irex.antenna*), 167
EnvelopeSystem (class in *pyrex.custom.irex.antenna*), 165
EnvelopeVpol (class in *pyrex.custom.irex*), 155
EnvelopeVpol (class in *pyrex.custom.irex.antenna*), 169
Event (class in *pyrex*), 62
Event (class in *pyrex.particle*), 103
EventIterator (class in *pyrex.io*), 120
EventKernel (class in *pyrex*), 66
EventKernel (class in *pyrex.kernel*), 116

F

File (class in pyrex), 68
 File (class in pyrex.io), 117
 FileGenerator (class in pyrex), 65
 FileGenerator (class in pyrex.generation), 115
 flatten() (in module pyrex.internal_functions), 70
 FunctionSignal (class in pyrex), 51
 FunctionSignal (class in pyrex.signals), 76

G

GaussianNoise (class in pyrex.signals), 77
 get_from_enum() (in module pyrex.internal_functions), 69
 GQRSInteraction (class in pyrex.particle), 106
 GreenlandIce (class in pyrex.ice_model), 95

H

HDF5Reader (class in pyrex.io), 118
 HDF5Writer (class in pyrex.io), 119
 HexagonalGrid (class in pyrex.custom.ara), 129
 HexagonalGrid (class in pyrex.custom.ara.detector), 144
 HpolAntenna (class in pyrex.custom.ara), 121
 HpolAntenna (class in pyrex.custom.ara.antenna), 135

I

ice (in module pyrex), 59
 Interaction (class in pyrex.particle), 105
 IREXString (class in pyrex.custom.irex), 157
 IREXString (class in pyrex.custom.irex.detector), 171

L

LayeredIce (class in pyrex.custom.layered_ice), 176
 LayeredIce (class in pyrex.custom.layered_ice.ice_model), 178
 LayeredRayTracePath (class in pyrex.custom.layered_ice), 177
 LayeredRayTracePath (class in pyrex.custom.layered_ice.ray_tracing), 179
 LayeredRayTracer (class in pyrex.custom.layered_ice), 176
 LayeredRayTracer (class in pyrex.custom.layered_ice.ray_tracing), 180
 lazy_property() (in module pyrex.internal_functions), 73
 LazyMutableClass (class in pyrex.internal_functions), 74
 ListGenerator (class in pyrex), 65
 ListGenerator (class in pyrex.generation), 113
 LPDA (class in pyrex.custom.arianna), 145
 LPDA (class in pyrex.custom.arianna.antenna), 151

M

mirror_func() (in module pyrex.internal_functions), 72

N

NeutrinoInteraction (in module pyrex), 61
 NeutrinoInteraction (in module pyrex.particle), 108
 normalize() (in module pyrex.internal_functions), 69
 NumpyFileGenerator (class in pyrex.generation), 114

P

Particle (class in pyrex), 61
 Particle (class in pyrex.particle), 104
 PhasedArrayString (class in pyrex.custom.ara), 125
 PhasedArrayString (class in pyrex.custom.ara.detector), 140
 prem_density() (in module pyrex), 60
 prem_density() (in module pyrex.earth_model), 91
 pyrex.antenna (module), 84
 pyrex.askaryan (module), 80
 pyrex.custom.ara.antenna (module), 130
 pyrex.custom.ara.detector (module), 138
 pyrex.custom.arianna.antenna (module), 147
 pyrex.custom.irex.antenna (module), 163
 pyrex.custom.irex.detector (module), 171
 pyrex.custom.irex.frontends (module), 162
 pyrex.custom.layered_ice.ice_model (module), 178
 pyrex.custom.layered_ice.ray_tracing (module), 179
 pyrex.detector (module), 88
 pyrex.earth_model (module), 91
 pyrex.generation (module), 108
 pyrex.ice_model (module), 92
 pyrex.internal_functions (module), 68
 pyrex.io (module), 117
 pyrex.kernel (module), 115
 pyrex.particle (module), 103
 pyrex.ray_tracing (module), 96
 pyrex.signals (module), 74

R

RayTracePath (in module pyrex), 66
 RayTracePath (in module pyrex.ray_tracing), 103
 RayTracer (in module pyrex), 66
 RayTracer (in module pyrex.ray_tracing), 103
 RectangularGenerator (class in pyrex), 63
 RectangularGenerator (class in pyrex.generation), 111
 RegularStation (class in pyrex.custom.ara), 126

RegularStation (class in *pyrex.custom.ara.detector*), 141
 RegularStation (class in *pyrex.custom.irex*), 158
 RegularStation (class in *pyrex.custom.irex.detector*), 172

S

ShadowGenerator (class in *pyrex.generation*), 112
 Signal (class in *pyrex*), 50
 Signal (class in *pyrex.signals*), 75
 slant_depth() (in module *pyrex*), 60
 slant_depth() (in module *pyrex.earth_model*), 92
 SpecializedRayTracePath (class in *pyrex.ray_tracing*), 98
 SpecializedRayTracer (class in *pyrex.ray_tracing*), 100
 StationGrid (class in *pyrex.custom.irex*), 160
 StationGrid (class in *pyrex.custom.irex.detector*), 174

T

ThermalNoise (class in *pyrex*), 52
 ThermalNoise (class in *pyrex.signals*), 78

U

UniformIce (class in *pyrex.ice_model*), 93
 UniformRayTracePath (class in *pyrex.ray_tracing*), 101
 UniformRayTracer (class in *pyrex.ray_tracing*), 102

V

VpolAntenna (class in *pyrex.custom.ara*), 123
 VpolAntenna (class in *pyrex.custom.ara.antenna*), 136

Z

ZHSAskaryanSignal (class in *pyrex.askaryan*), 80