PyREx Documentation

Release 1.6.0

Ben Hokanson-Fasig

Jul 02, 2018

CONTENTS

CHAPTER ONE

ABOUT PYREX

PyREx (**Py**thon package for an IceCube **R**adio **Ex**tension) is, as its name suggests, a Python package designed to simulate the measurement of Askaryan pulses via a radio antenna array around the IceCube South Pole Neutrino Observatory. The code is designed to be modular so that it can also be applied to other askaryan radio antennas (e.g. the ARA and ARIANA collaborations).

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+ and scipy version 0.19+, 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 pryex
```

Then, create a particle generator object that will produce random particles in a cube of 1 km on each side with a fixed energy of 100 PeV:

```
particle_generator = pyrex.ShadowGenerator(dx=1000, dy=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(
```

```
        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/cm^3)**
material thickness	**grams per square centimeter (g/cm^2)**
temperature	kelvin (K)
energy	**gigaelectronvolts (GeV)**
resistance	ohms (Ω)
voltage	volts (V)
electric field	volts per meter (V/m)

CHAPTER

TWO

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.signal
import scipy.fftpack
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.ValueTypes` enum. By default, a *Signal* object has `value_type=ValueTypes.unknown`. However, if the signal represents a voltage, electric field, or power; `value_type` can be set to `Signal.ValueTypes.voltage`, `Signal.ValueTypes.field`, or `Signal.ValueTypes.power` respectively:

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

Signal objects can be added as long as they have the same time array and `value_type`. *Signal* objects also support the python `sum()` function:

```
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)")
all_signals = [signal1, signal2, signal3]
signal4 = sum(all_signals)
plt.plot(signal4.times, signal4.values, label="signal4 = 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 freqeuncies (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 due to a neutrino of given energy observed at a given angle from the shower axis:

```
time_array = np.linspace(-10e-9, 40e-9, 1001)
neutrino_energy = 1e8 # GeV
observation_angle = 45 * np.pi/180 # radians
askaryan = pyrex.AskaryanSignal(times=time_array, energy=neutrino_energy,
                                theta=observation_angle)
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.

```
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.response()`, `Antenna.directional_gain()`, and `Antenna.polarization_gain()`. The attributes are to be set and the methods overwritten in order to custmoize 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.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.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.ValueTypes.voltage)
incoming_signal_2 = pyrex.FunctionSignal(np.linspace(4*np.pi,6*np.pi), np.sin,
                                         value_type=pyrex.Signal.ValueTypes.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.ValueTypes.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:
    plt.figure()
    plt.plot(wave.times, wave.values)
    plt.show()
```

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

PyREx defines `DipoleAntenna`, a subclass of `Antenna` which provides a basic threshold trigger, a basic band-pass 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 a `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. A different trigger 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.ValueTypes.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.ValueTypes.voltage)
incoming_signal_2 = pyrex.FunctionSignal(np.linspace(4*np.pi,6*np.pi), np.sin,
                                         value_type=pyrex.Signal.ValueTypes.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")
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. 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,
                             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()
```

2.4 Ice and Earth Models

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

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

PyREx also provides two functions realted 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 = (100, 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 = 500e6 # Hz
my_path.attenuation(100e6)
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.

```
time_array = np.linspace(0, 5e-9, 1001)
my_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(my_signal.times, my_signal.values)
plt.show()

my_path.propagate(my_signal)
my_signal.values /= my_path.path_length
plt.plot(my_signal.times, my_signal.values)
plt.show()
```

2.6 Particle Generation

PyREx includes the *Particle* class as a container for information about neutrinos which are generated to produce Askaryan pulses. *Particle* contains three attributes: `vertex`, `direction`, and `energy`:

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

PyREx also includes a *ShadowGenerator* class for generating random neutrinos, taking into account Earth shadowing. The neutrinos are generated in a box of given size, and with a given energy (which can be a scalar value or a function returning scalar values):

```
box_width = 1000 # m
box_depth = 500 # m
my_generator = pyrex.ShadowGenerator(dx=box_width, dy=box_width,
                                     dz=box_depth,
                                     energy=particle_energy)
my_generator.create_particle()
```

Lastly, PyREx includes `ListGenerator` and `FileGenerator` classes which can be used to reproduce pre-generated particles from either a list or from numpy files, respectively.

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.ShadowGenerator(dx=1000, dy=1000, dz=500,
                                           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=0, noisy=False)
```

```
    )
kernel = pyrex.EventKernel(generator=particle_generator,
                           antennas=detector,
                           ice_model=pyrex.IceModel,
                           ray_tracer=pyrex.RayTracer,
                           signal_times=np.linspace(-20e-9, 80e-9, 2000,
                                                    endpoint=False))

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

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 More Examples

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

CHAPTER THREE

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 custom modules for IREX (IceCube Radio Extension) and ARA (Askaryan Radio Array) accessible at `pyrex.custom.irex` and `pyrex.custom.ara`, respectively. More information about these modules can be found in their respective sections below.

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 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/
|   |-- pyspice.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/
|   |   |-- __init__.py
|   |   |-- antenna.py
|   |   |-- ...

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

3.1 IREX Custom Module

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 Nx by Ny where N is the total number of stations, Nx=floor(sqrt(N)), and Ny=floor(N/Nx).

3.2 ARA Custom Module

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 `ARAString` class creates a string of alternating `HpolAntenna` and `VpolAntenna` ojbects, 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.

3.3 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 behavoirs 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.

3.3.1 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.
`energy`	The energy of the particle shower.
`theta`	The viewing angle in radians measured from the shower axis.
`n`	The index of refraction of the ice at the shower vertex (default 1.78)
`t0`	The starting time of the Askaryan pulse (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`).

3.3.2 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.

3.3.3 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.

3.3.4 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 othe 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.

3.3.5 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* classess 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) alon 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 singal 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.

3.3.6 Particle Generator

The particle generator classes are quite flexible. The only requirement is that they possess a `create_particle()` method which returns a *Particle* object.

CHAPTER FOUR

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 proccess, and a reference to the corresponding example script or notebook which can be run independent of one another. The examples are orgainzed 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 alternatingly 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()
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()
plt.show()
```

4.2 Askaryan Frequency Content

In this example we explore how the freqeuncy 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 the depth of the neutrino source and find the index of refraction
# at that depth.
# Then use that index of refraction to calculate the Cherenkov angle.
depth = -1000
n = pyrex.IceModel.index(depth)
ch_angle = np.arcsin(np.sqrt(1 - 1/n**2))

# Now, for a range of dthetas, generate an Askaryan pulse dtheta away from the
# Chereknov 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),
                                 energy=1e8, theta=ch_angle-dtheta, n=n)
    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=(0, 0, depth), 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),
                                 energy=1e8, theta=ch_angle-dtheta, n=n)
    path.propagate(pulse)
    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()

# 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.
n_events = 10
energies = [1e8, 1e9, 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. Let's scale the generation
# volume by energy so that we're not wasting too much time generating neutrinos
# that will surely never trigger.
dimensions = [2500, 5000, 10000]
generators = [pyrex.ShadowGenerator(dx=2*dim, dy=2*dim, dz=2800, energy=energy)
              for energy, dim in zip(energies, dimensions)]

# 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=pryex.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
# 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("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.array([(2*dim)*(2*dim)*2800 for dim in dimensions])
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("Shower Energy (GeV)")
plt.ylabel("Effective Volume (km^3)")
plt.show()

# Then from the effecitve volumes, we can calculate the effective areas.
# First we need to account for the fact that our energy is the shower energy
# and convert to the neutrino energy. Then the effective area is the probability
# of 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. Since the neutrino type is not specified in the
# simulation, calculate the cross section as a weighted average of neutrino
# cross sections.
nu_energies = 9/5*np.array(energies)
ice_density = 0.92 # g/cm^3
ice_density *= 1e15 # converted to g/km^3 = nucleons/km^3
cross_sections = (pyrex.particle.CC_NU.cross_section(nu_energies) +
                  3*pyrex.particle.NC_NU.cross_section(nu_energies) +
                  pyrex.particle.CC_NUBAR.cross_section(nu_energies) +
                  3*pyrex.particle.NC_NUBAR.cross_section(nu_energies)) / 8
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

plt.errorbar(nu_energies, effective_areas, 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(vertex=[1002.65674195, -421.95118348, -586.0953201],
                   direction=[-0.90615395, -0.41800062, -0.06450191],
                   energy=1e9)
gen = pyrex.ListGenerator(p)
kern = pyrex.EventKernel(antennas=det, generator=gen)

# 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 = []
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()
```

CHAPTER

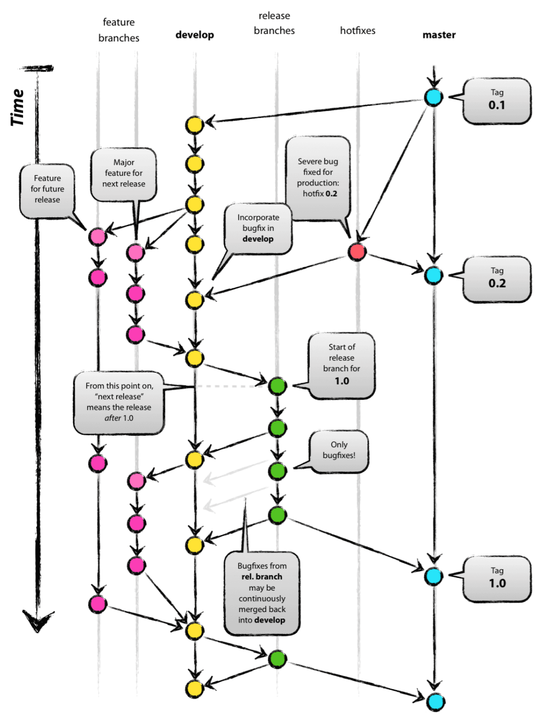
FIVE

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. One 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
```

CHAPTER

SIX

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

`Signal`(times, values[, value_type])	Base class for time-domain signals.
`EmptySignal`(times[, value_type])	Class for signal with zero amplitude (all values = 0).
`FunctionSignal`(times, function[, value_type])	Class for signals generated by a function.
`AskaryanSignal`	alias of `FastAskaryanSignal`
`ThermalNoise`(times, f_band[, f_amplitude, . . .])	Class for thermal Rayleigh noise signals.
`Antenna`(position[, z_axis, x_axis, . . .])	Base class for antennas.
`DipoleAntenna`(name, position, . . . [, . . .])	Class for half-wave dipole antennas.
`AntennaSystem`(antenna)	Base class for antenna system with front-end processing.
`Detector`(*args, **kwargs)	Base class for detectors for easily building up sets of antennas.
`IceModel`	alias of `AntarcticIce`
`prem_density`(r)	Calculates the Earth's density at a given radius.
`slant_depth`(angle, depth[, step])	Calculates the material thickness of a chord cutting through Earth.
`Particle`(vertex, direction, energy)	Class for storing particle attributes.
`ShadowGenerator`(dx, dy, dz, energy)	Class to generate neutrino vertices with Earth shadowing.
`RayTracer`	alias of `SpecializedRayTracer`
`RayTracePath`	alias of `SpecializedRayTracePath`
`EventKernel`(generator, antennas[, . . .])	High-level kernel for controlling event simulation.

6.1.1 pyrex.Signal

class `pyrex.`**`Signal`**(*times*, *values*, *value_type=<ValueTypes.undefined: 0>*)

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 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

Type of signal, representing the units of the values. Must be from the `Signal.ValueTypes` enum.

Attributes

`ValueTypes`	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.

times, values	(ndarray) 1D arrays of times and corresponding values which define the signal.
value_type	Type of signal, representing the units of the values.

Methods

`ValueTypes`	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.

6.1.2 pyrex.EmptySignal

class `pyrex.`**`EmptySignal`**(*times*, *value_type=<ValueTypes.undefined: 0>*)

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

Parameters **times** : array_like

1D array of times for which the signal is defined.

value_type

Type of signal, representing the units of the values. Must be from the `Signal.ValueTypes` Enum.

See also:

[`Signal`] Base class for time-domain signals.

Attributes

ValueTypes	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.

times, values	(ndarray) 1D arrays of times and corresponding values which define the signal.
value_type	Type of signal, representing the units of the values.

Methods

ValueTypes	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=<ValueTypes.undefined: 0>*)

Class for signals generated by a function.

Parameters **times** : array_like

1D array of times 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

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

See also:

Signal Base class for time-domain signals.

EmptySignal Class for signal with zero amplitude.

Attributes

`ValueTypes`	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.

times, values	(ndarray) 1D arrays of times and corresponding values which define the signal.
value_type	Type of signal, representing the units of the values.
function	(function) Function to evaluate the signal values at given time(s).

Methods

`ValueTypes`	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 `FastAskaryanSignal`

6.1.5 pyrex.ThermalNoise

`class` `pyrex.`**`ThermalNoise`**(*times*, *f_band*, *f_amplitude=1*, *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 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.

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 *[R3131]*.

References

[R3131]

Attributes

`ValueTypes`	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.

times, values	(ndarray) 1D arrays of times and corresponding values which define the signal.
value_type	Type of signal, representing the units of the 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.

Methods

`ValueTypes`	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

`is_hit`	Boolean of whether the antenna has been triggered.
`waveforms`	Signal + noise (if `noisy`) for each triggered antenna hit.
`all_waveforms`	Signal + noise (if `noisy`) for all antenna hits.

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.

Methods

`clear`([reset_noise])	Reset the antenna to an empty state.
`directional_gain`(theta, phi)	Calculate the (complex) directional gain of the antenna.
`full_waveform`(times)	Signal + noise (if `noisy`) for the given times.
`is_hit_during`(times)	Check if the antenna is triggered in a time range.

Continued on next page

Table 6.11 – continued from previous page

`make_noise`(times)	Creates a noise signal over the given times.
`polarization_gain`(polarization)	Calculate the (complex) polarization gain of the antenna.
`receive`(signal[, direction, polarization, . . .])	Process and store an incoming signal.
`response`(frequencies)	Calculate the (complex) frequency response of the antenna.
`set_orientation`([z_axis, x_axis])	Sets the orientation of the antenna.
`trigger`(signal)	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

`is_hit`	Boolean of whether the antenna has been triggered.
`waveforms`	Signal + noise (if `noisy`) for each triggered antenna hit.
`all_waveforms`	Signal + noise (if `noisy`) for all antenna hits.

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.

Methods

`clear`([reset_noise])	Reset the antenna to an empty state.
`directional_gain`(theta, phi)	Calculate the (complex) directional gain of the antenna.
`full_waveform`(times)	Signal + noise (if `noisy`) for the given times.
`is_hit_during`(times)	Check if the antenna is triggered in a time range.
`make_noise`(times)	Creates a noise signal over the given times.
`polarization_gain`(polarization)	Calculate the (complex) polarization gain of the antenna.
`receive`(signal[, direction, polarization, . . .])	Process and store an incoming signal.
`response`(frequencies)	Calculate the (complex) frequency response of the antenna.

Continued on next page

Table 6.13 – continued from previous page

`set_orientation`([z_axis, x_axis])	Sets the orientation of the antenna.
`trigger`(signal)	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

`is_hit`	Boolean of whether the antenna system has been triggered.
`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.

antenna	(Antenna) `Antenna` object extended by the front end.

Methods

`clear`([reset_noise])	Reset the antenna system to an empty state.
`front_end`(signal)	Apply front-end processes to a signal and return the output.
`full_waveform`(times)	Signal + noise (if `noisy`) for the given times.
`is_hit_during`(times)	Check if the antenna system is triggered in a time range.
`receive`(signal[, direction, polarization, ...])	Process and store an incoming signal.
`setup_antenna`(*args, **kwargs)	Setup the antenna by passing along its init arguments.
`trigger`(signal)	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 mutliple 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

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

6.1.10 pyrex.IceModel

pyrex.**IceModel**

alias of AntarcticIce

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/cm^3) of the Earth at the given radii.

Notes

The density calculation is based on the Preliminary reference Earth Model *[R2121]*.

References

[R2121]

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/cm^2) 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.Particle

`class pyrex.Particle(vertex, direction, energy)`

Class for storing particle attributes.

Parameters **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.

Attributes

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.

6.1.14 pyrex.ShadowGenerator

class pyrex.**ShadowGenerator**(*dx*, *dy*, *dz*, *energy*)

Class to generate neutrino vertices with Earth shadowing.

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.

See also:

`pyrex.slant_depth` Calculates the material thickness of a chord cutting through Earth.

Attributes

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_generator	(function) Function returning energy (GeV) of the neutrinos by successive function calls.
count	(int) Number of neutrinos produced by the generator.

Methods

`create_particle()`	Generate a neutrino.

6.1.15 pyrex.RayTracer

`pyrex.`**`RayTracer`**
alias of `SpecializedRayTracer`

6.1.16 pyrex.RayTracePath

`pyrex.`**`RayTracePath`**
alias of `SpecializedRayTracePath`

6.1.17 pyrex.EventKernel

`class` `pyrex.`**`EventKernel`**(*generator*, *antennas*, *ice_model=<class 'pyrex.ice_model.AntarcticIce'>*, *ray_tracer=<class 'pyrex.ray_tracing.SpecializedRayTracer'>*, *signal_times=array([-2.000e-08, -1.995e-08, -1.990e-08, ..., 7.985e-08, 7.990e-08, 7.995e-08])*)

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_times : array_like, optional

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

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_particle` method which takes no arguments and returns a *Particle* object with `vertex`, `direction`, and `energy` 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 revelaing 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.

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

Methods

`event()`	Create a neutrino event and run it through the simulation chain.

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.

normalize(vector)	Normalize the given vector.
flatten(iterator[, dont_flatten])	Flattens an iterator to iterate over all elements individually.
mirror_func(match_func, run_func[, self])	Mirror the attributes of one function onto another.
lazy_property(fn)	Decorator that makes a property lazily evaluated.
LazyMutableClass([static_attributes])	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.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.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__
'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

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.

`Signal`(times, values[, value_type])	Base class for time-domain signals.
`EmptySignal`(times[, value_type])	Class for signal with zero amplitude (all values = 0).
`FunctionSignal`(times, function[, value_type])	Class for signals generated by a function.
`SlowAskaryanSignal`(times, energy, theta[, n, t0])	Class for generating Askaryan signals according to ARWZ parametrization.
`FastAskaryanSignal`(times, energy, theta[, n, t0])	Class for generating Askaryan signals according to ARWZ parametrization.
`AskaryanSignal`	alias of `FastAskaryanSignal`
`GaussianNoise`(times, sigma)	Class for gaussian noise signals with standard deviation sigma.
`ThermalNoise`(times, f_band[, f_amplitude, …])	Class for thermal Rayleigh noise signals.

pyrex.signals.Signal

class pyrex.signals.**Signal**(*times*, *values*, *value_type=<ValueTypes.undefined: 0>*)

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 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

Type of signal, representing the units of the values. Must be from the `Signal.ValueTypes` enum.

Attributes

`ValueTypes`	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.

times, values	(ndarray) 1D arrays of times and corresponding values which define the signal.
value_type	Type of signal, representing the units of the values.

Methods

`ValueTypes`	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=<ValueTypes.undefined: 0>*)

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

Parameters **times** : array_like

1D array of times for which the signal is defined.

value_type

Type of signal, representing the units of the values. Must be from the `Signal.ValueTypes` Enum.

See also:

`Signal` Base class for time-domain signals.

Attributes

`ValueTypes`	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.

times, values	(ndarray) 1D arrays of times and corresponding values which define the signal.
value_type	Type of signal, representing the units of the values.

Methods

`ValueTypes`	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=<ValueTypes.undefined: 0>*)
Class for signals generated by a function.

Parameters **times** : array_like

1D array of times 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

Type of signal, representing the units of the values. Must be from the `Signal.ValueTypes` Enum.

See also:

`Signal` Base class for time-domain signals.

`EmptySignal` Class for signal with zero amplitude.

Attributes

`ValueTypes`	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.

times, values	(ndarray) 1D arrays of times and corresponding values which define the signal.
value_type	Type of signal, representing the units of the values.
function	(function) Function to evaluate the signal values at given time(s).

Methods

`ValueTypes`	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.SlowAskaryanSignal

class `pyrex.signals.`**`SlowAskaryanSignal`**(*times*, *energy*, *theta*, *n=1.78*, *t0=0*)

Class for generating Askaryan signals according to ARWZ parametrization.

Stores the time-domain information for an Askaryan electric field (V/m). The amplitude of the pulse is calculated at 1 meter from the shower, so a 1/R effect (where R is the distance from source to observer) must be applied to produce the proper signal amplitude at the observation point.

Parameters **times** : array_like

1D array of times for which the signal is defined.

energy : float

Energy (GeV) of the particle shower producing the pulse.

theta : float

Observation angle (radians) measured relative to the shower axis.

n : float, optional

Index of refraction at the location of the shower.

t0 : float, optional

Pulse offset time (s), i.e. time at which the shower takes place.

Warns **Raises warning that this class is essentially deprecated and**

FastAskaryanSignal should be used instead.

> **Warning:** This class is essentially deprecated. `FastAskaryanSignal` is faster and more accurate. This class is kept for reference only and shouldn't be used.

See also:

`Signal` Base class for time-domain signals.

`FastAskaryanSignal` Faster class for ARWZ Askaryan parametrization.

Notes

Calculates the Askaryan signal based on the ARWZ (Alvarez-Muniz, Romero-Wolf, Zas) parametrization *[R5960]*. Uses a simplified Heitler model for the shower profile *[R6060]*. Calculates the vector potential directly, and then numerically differentiates to get the electric field. This method is considerably slower (~1000x) than the convolution method in `FastAskaryanSignal`, and seems to be backwards in time as well. The class is kept for reference only and shouldn't be used.

References

[R5960], *[R6060]*

Attributes

`ValueTypes`	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.

times, values	(ndarray) 1D arrays of times and corresponding values which define the signal.
value_type	(Signal.ValueTypes.field) Type of signal, representing the units of the values.
energy	(float) Energy (GeV) of the particle shower producing the pulse.
vector_potential	(ndarray) 1D array of vector potential (V*s/m) of the pulse

Methods

`RAC`(time)	Calculates R*A_C at the given time.
`ValueTypes`	Enum containing possible types (units) for signal values.
`charge_profile`(z[, density, crit_energy, ...])	Calculates the EM shower longitudinal charge profile.
`filter_frequencies`(freq_response[, force_real])	Apply the given frequency response function to the signal, in-place.
`max_length`([density, crit_energy, rad_length])	Calculates the maximum length (or shower depth) of an EM shower.
`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.FastAskaryanSignal

class `pyrex.signals.`**`FastAskaryanSignal`**(*times*, *energy*, *theta*, *n=1.78*, *t0=0*)

Class for generating Askaryan signals according to ARWZ parametrization.

Stores the time-domain information for an Askaryan electric field (V/m). The amplitude of the pulse is calculated at 1 meter from the shower, so a 1/R effect (where R is the distance from source to observer) must be applied to produce the proper signal amplitude at the observation point.

Parameters **times** : array_like

1D array of times for which the signal is defined.

energy : float

Energy (GeV) of the particle shower producing the pulse.

theta : float

Observation angle (radians) measured relative to the shower axis.

n : float, optional

Index of refraction at the location of the shower.

t0 : float, optional

Pulse offset time (s), i.e. time at which the shower takes place.

See also:

`Signal` Base class for time-domain signals.

Notes

Calculates the Askaryan signal based on the ARWZ (Alvarez-Muniz, Romero-Wolf, Zas) parametrization *[R4748]*. Uses a Heitler model for the shower profile *[R4848]*. Calculates the electric field directly using the convolution method outlined in the ARWZ paper, which results in the most efficient calculation of the parametrization.

References

[R4748], *[R4848]*

Attributes

`ValueTypes`	Enum containing possible types (units) for signal values.
`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.

times, values	(ndarray) 1D arrays of times and corresponding values which define the signal.
value_type	(Signal.ValueTypes.field) Type of signal, representing the units of the values.
energy	(float) Energy (GeV) of the particle shower producing the pulse.

Methods

`RAC`(time)	Calculates R*A_C at the given time.
`ValueTypes`	Enum containing possible types (units) for signal values.

Continued on next page

Table 6.30 – continued from previous page

`charge_profile`(z[, density, crit_energy, ...])	Calculates the EM shower longitudinal charge profile.
`filter_frequencies`(freq_response[, force_real])	Apply the given frequency response function to the signal, in-place.
`max_length`([density, crit_energy, rad_length])	Calculates the maximum length (or shower depth) of an EM shower.
`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.AskaryanSignal

`pyrex.signals.`**`AskaryanSignal`**

alias of *`FastAskaryanSignal`*

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 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.ValueTypes` Enum.

See also:

`Signal` Base class for time-domain signals.

Attributes

`ValueTypes`	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.

times, values	(ndarray) 1D arrays of times and corresponding values which define the signal.
value_type	Type of signal, representing the units of the values.

Methods

`ValueTypes`	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.ThermalNoise

class `pyrex.signals.`**`ThermalNoise`**(*times*, *f_band*, *f_amplitude=1*, *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 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.

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 *[R7171]*.

References

[R7171]

Attributes

`ValueTypes`	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.

times, values	(ndarray) 1D arrays of times and corresponding values which define the signal.
value_type	Type of signal, representing the units of the 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.

Methods

`ValueTypes`	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 6.34 – continued from previous page

`with_times`(new_times)	Returns a representation of this signal over a different times array.

6.2.3 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.

`Antenna`(position[, z_axis, x_axis, . . .])	Base class for antennas.
`DipoleAntenna`(name, position, . . . [, . . .])	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

`is_hit`	Boolean of whether the antenna has been triggered.
`waveforms`	Signal + noise (if `noisy`) for each triggered antenna hit.
`all_waveforms`	Signal + noise (if `noisy`) for all antenna hits.

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.

Methods

`clear`([reset_noise])	Reset the antenna to an empty state.
`directional_gain`(theta, phi)	Calculate the (complex) directional gain of the antenna.
`full_waveform`(times)	Signal + noise (if `noisy`) for the given times.
`is_hit_during`(times)	Check if the antenna is triggered in a time range.
`make_noise`(times)	Creates a noise signal over the given times.
`polarization_gain`(polarization)	Calculate the (complex) polarization gain of the antenna.
`receive`(signal[, direction, polarization, ...])	Process and store an incoming signal.

Continued on next page

Table 6.37 – continued from previous page

`response`(frequencies)	Calculate the (complex) frequency response of the antenna.
`set_orientation`([z_axis, x_axis])	Sets the orientation of the antenna.
`trigger`(signal)	Check if the antenna triggers on a given signal.

pyrex.antenna.DipoleAntenna

class pyrex.antenna.**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

`is_hit`	Boolean of whether the antenna has been triggered.
`waveforms`	Signal + noise (if `noisy`) for each triggered antenna hit.
`all_waveforms`	Signal + noise (if `noisy`) for all antenna hits.

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.

Methods

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

6.2.4 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.

`AntennaSystem`(antenna)	Base class for antenna system with front-end processing.
`Detector`(*args, **kwargs)	Base class for detectors for easily building up sets of antennas.

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

`is_hit`	Boolean of whether the antenna system has been triggered.
`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.

antenna	(Antenna) `Antenna` object extended by the front end.

Methods

`clear`([reset_noise])	Reset the antenna system to an empty state.
`front_end`(signal)	Apply front-end processes to a signal and return the output.
`full_waveform`(times)	Signal + noise (if `noisy`) for the given times.
`is_hit_during`(times)	Check if the antenna system is triggered in a time range.
`receive`(signal[, direction, polarization, ...])	Process and store an incoming signal.
`setup_antenna`(*args, **kwargs)	Setup the antenna by passing along its init arguments.
`trigger`(signal)	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 mutliple 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

`build_antennas`(*args, **kwargs)	Creates antenna objects at the set antenna positions.
`clear`([reset_noise])	Reset the detector to an empty state.
`set_positions`(*args, **kwargs)	Sets the positions of antennas in the detector.

Continued on next page

Table 6.43 – continued from previous page

triggered(*args, **kwargs)	Check if the detector is triggered based on its current state.

6.2.5 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.

prem_density(r)	Calculates the Earth's density at a given radius.
slant_depth(angle, depth[, step])	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/cm^3) of the Earth at the given radii.

Notes

The density calculation is based on the Preliminary reference Earth Model *[R1313]*.

References

[R1313]

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/cm^2) 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.

6.2.6 Ice Models (`pyrex.ice_model`)

Module containing ice model classes.

The ice model classes contain static and class methods for convenience, and parameters of the ice model are set as class attributes.

`AntarcticIce`	Class describing the ice at the south pole.
`NewcombIce`	Class describing the ice at the south pole.
`ArasimIce`	Class describing the ice at the south pole.
`IceModel`	alias of `AntarcticIce`

pyrex.ice_model.AntarcticIce

class `pyrex.ice_model.`**`AntarcticIce`**

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.

Notes

Mostly based on ice characteristics outlined by Matt Newcomb.

Attributes

k, a, n0	(float) Parameters of the index of refraction of the ice.
thickness	(float) Thickness of the ice sheet.

Methods

`attenuation_length`(z, f)	Calculates attenuation lengths for given depths and frequencies.
`depth_with_index`(n)	Calculates the corresponding depth for a given index of refraction.
`gradient`(z)	Calculates the gradient of the index of refraction at a given depth.

Continued on next page

Table 6.46 – continued from previous page

`index(z)`	Calculates the index of refraction of the ice at a given depth.
`temperature(z)`	Calculates the temperature of the ice at a given depth.

pyrex.ice_model.NewcombIce

class `pyrex.ice_model.`**`NewcombIce`**

Class describing the ice at the south pole.

Uses an attenuation length based on Matt Newcomb's fit. 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.

> **Warning:** The `attenuation_length` method if this class does not currently work properly. This class should not be used until it is fixed.

Notes

Mostly based on ice characteristics outlined by Matt Newcomb.

Attributes

k, a, n0	(float) Parameters of the index of refraction of the ice.
thickness	(float) Thickness of the ice sheet.

Methods

`attenuation_length(z, f)`	Calculates attenuation lengths for given depths and frequencies.
`depth_with_index(n)`	Calculates the corresponding depth for a given index of refraction.
`gradient(z)`	Calculates the gradient of the index of refraction at a given depth.
`index(z)`	Calculates the index of refraction of the ice at a given depth.
`temperature(z)`	Calculates the temperature of the ice at a given depth.

pyrex.ice_model.ArasimIce

class `pyrex.ice_model.`**`ArasimIce`**

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.

Attributes

k, a, n0	(float) Parameters of the index of refraction of the ice.
thickness	(float) Thickness of the ice sheet.
atten_depths, atten_lengths	(list) Depths and corresponding attenuation lengths to be interpolated in the `attenuation_length` calculation.

Methods

`attenuation_length`(z, f)	Calculates attenuation lengths for given depths and frequencies.
`depth_with_index`(n)	Calculates the corresponding depth for a given index of refraction.
`gradient`(z)	Calculates the gradient of the index of refraction at a given depth.
`index`(z)	Calculates the index of refraction of the ice at a given depth.
`temperature`(z)	Calculates the temperature of the ice at a given depth.

pyrex.ice_model.IceModel

`pyrex.ice_model.`**`IceModel`**

alias of *`AntarcticIce`*

6.2.7 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 existance and launch angle of paths between points, and the ray tracer path objects are responsible for returning information about propagation along their respective path.

`BasicRayTracePath`(parent_tracer, ...)	Class for representing a single ray-trace solution between points.
`SpecializedRayTracePath`(parent_tracer, ...)	Class for representing a single ray-trace solution between points.
`BasicRayTracer`(from_point, to_point[, ...])	Class for calculating the ray-trace solutions between points.
`SpecializedRayTracer`(from_point, to_point[, ...])	Class for calculating the ray-trace solutions between points.
`RayTracer`	alias of *`SpecializedRayTracer`*
`RayTracePath`	alias of *`SpecializedRayTracePath`*
`PathFinder`(ice_model, from_point, to_point)	Class for pseudo ray tracing.
`ReflectedPathFinder`(ice_model, from_point, ...)	Class for pseudo ray tracing of a reflected ray.

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

`attenuation`(f)	Calculate the attenuation factor for signal frequencies.
`propagate`(signal)	Propagate the signal along the ray path, in-place.

Continued on next page

Table 6.50 – continued from previous page

`theta`(z)	Polar angle of the ray at the given depths.
`z_integral`(integrand)	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)=n0-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)=n0-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 (n0 in the ice model) beyond which calculations start to break down numerically.
beta_tolerance	(float) `beta` value (near 0) below which calculations start to break down numerically.
emitted_direction	
received_direction	
path_length	
tof	
coordinates	

Methods

`attenuation`(f)	Calculate the attenuation factor for signal frequencies.
`propagate`(signal)	Propagate the signal along the ray path, in-place.
`theta`(z)	Polar angle of the ray at the given depths.
`z_integral`(integrand[, numerical, x_func])	Calculate the integral of the given integrand.

pyrex.ray_tracing.BasicRayTracer

class pyrex.ray_tracing.**BasicRayTracer**(*from_point*, *to_point*, *ice_model=<class 'pyrex.ice_model.AntarcticIce'>*, *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

The ice model used for the ray tracer.

dz : float

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

`solution_class`	alias of `BasicRayTracePath`

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.
exists	
expected_solutions	
solutions	

Methods

`angle_search`(true_r, r_function, min_angle, ...)	Calculates the angle where *r_function* (angle) == *true_r*.
`solution_class`	alias of `BasicRayTracePath`

pyrex.ray_tracing.SpecializedRayTracer

class pyrex.ray_tracing.**SpecializedRayTracer**(*from_point*, *to_point*, *ice_model=<class 'pyrex.ice_model.AntarcticIce'>*, *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)=n0-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

The ice model used for the ray tracer.

dz : float

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)=n0-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

`solution_class`	alias of *SpecializedRayTracePath*

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.
exists	
expected_solutions	
solutions	

Methods

`angle_search`(true_r, r_function, min_angle, . . .)	Calculates the angle where *r_function* (angle) == *true_r*.
`solution_class`	alias of *SpecializedRayTracePath*

pyrex.ray_tracing.RayTracer

`pyrex.ray_tracing.`**`RayTracer`**

alias of *SpecializedRayTracer*

pyrex.ray_tracing.RayTracePath

`pyrex.ray_tracing.`**`RayTracePath`**

alias of *SpecializedRayTracePath*

pyrex.ray_tracing.PathFinder

`class` `pyrex.ray_tracing.`**`PathFinder`**(*ice_model*, *from_point*, *to_point*)

Class for pseudo ray tracing. Just uses straight-line paths.

Parameters ice_model

The ice model used for the ray tracer.

from_point : array_like

Vector starting point of the ray path.

to_point : array_like

Vector ending point of the ray path.

Attributes

`exists`	Boolean of whether the path exists between the endpoints.
`emitted_ray`	Direction in which the ray is emitted.
`received_ray`	Direction from which the ray is received.
`path_length`	Length (m) of the path.
`tof`	Time of flight (s) for a particle along the path.

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.

Methods

`attenuation`(f[, n_steps])	Calculate the attenuation factor for signal frequencies.
`propagate`(signal)	Propagate the signal along the ray path, in-place.
`time_of_flight`([n_steps])	Time of flight (s) for a particle along the path.

pyrex.ray_tracing.ReflectedPathFinder

class pyrex.ray_tracing.**ReflectedPathFinder**(*ice_model*, *from_point*, *to_point*, *reflection_depth=0*)

Class for pseudo ray tracing of a reflected ray. Uses straight-line paths.

Parameters ice_model

The ice model used for the ray tracer.

from_point : array_like

Vector starting point of the ray path.

to_point : array_like

Vector ending point of the ray path.

reflection_depth : float, optional

(Negative-valued) depth (m) at which the ray reflects.

Attributes

`exists`	Boolean of whether the path exists between the endpoints.
`emitted_ray`	Direction in which the ray is emitted.
`received_ray`	Direction from which the ray is received.
`path_length`	Length of the path (m).
`tof`	Time of flight (s) for a particle along the path.

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.
bounce_point	(ndarray) The point at which the ray path is reflected.
path_1	(PathFinder) The path from *from_point* to *bounce_point*.
path_2	(PathFinder) The path from *bounce_point* to *to_point*.

Methods

`attenuation`(f[, n_steps])	Calculate the attenuation factor for signal frequencies.
`get_bounce_point`([reflection_depth])	Calculates the point at which the ray is reflected.
`propagate`(signal)	Propagate the signal along the ray path, in-place.
`time_of_flight`([n_steps])	Time of flight (s) for a particle along the path.

6.2.8 Particles and Generators (`pyrex.particle`)

Module for particles (neutrinos) and neutrino interactions in the ice.

Included in the module are Particle and NeutrinoInteraction classes, as well as different particle generators.

`NeutrinoInteraction`(c, p)	Class for describing neutrino interaction attributes.
`CC_NU`	Class for describing neutrino interaction attributes.
`NC_NU`	Class for describing neutrino interaction attributes.
`CC_NUBAR`	Class for describing neutrino interaction attributes.
`NC_NUBAR`	Class for describing neutrino interaction attributes.
`Particle`(vertex, direction, energy)	Class for storing particle attributes.
`random_direction`()	Generate an arbitrary cartesian unit vector.
`ShadowGenerator`(dx, dy, dz, energy)	Class to generate neutrino vertices with Earth shadowing.
`ListGenerator`(particles[, loop])	Class to generate neutrino vertices from a list.
`FileGenerator`(files)	Class to generate neutrino vertices from numpy file(s).

pyrex.particle.NeutrinoInteraction

class `pyrex.particle.`**`NeutrinoInteraction`**(*c*, *p*)

Class for describing neutrino interaction attributes.

Stores parameters used to describe cross section and interaction length of a specific neutrino interaction.

Parameters **c** : float

Cross section energy coefficient.

p : float

Cross section energy exponent.

Notes

Neutrino intractions based on the GQRS Ultrahigh-Energy Neutrino Interactions Paper *[R1919]*.

References

[R1919]

Attributes

times, values	(ndarray) 1D arrays of times and corresponding values which define the signal.
value_type	Type of signal, representing the units of the values.
ValueTypes	(Enum) Different value types available for *value_type* of signal objects.
dt	
frequencies	
spectrum	
envelope	

Methods

`cross_section`(E)	Calculate the neutrino cross section at a given energy.
`interaction_length`(E)	Calculate the neutrino interaction length at a given energy.

pyrex.particle.CC_NU

`pyrex.particle.`**`CC_NU = <pyrex.particle.NeutrinoInteraction object>`**

Class for describing neutrino interaction attributes.

Stores parameters used to describe cross section and interaction length of a specific neutrino interaction.

Parameters **c** : float

Cross section energy coefficient.

p : float

Cross section energy exponent.

Notes

Neutrino intractions based on the GQRS Ultrahigh-Energy Neutrino Interactions Paper *[R15]*.

References

[R15]

Attributes

times, values	(ndarray) 1D arrays of times and corresponding values which define the signal.
value_type	Type of signal, representing the units of the values.
ValueTypes	(Enum) Different value types available for *value_type* of signal objects.
dt	
frequencies	
spectrum	
envelope	

pyrex.particle.NC_NU

`pyrex.particle.NC_NU = <pyrex.particle.NeutrinoInteraction object>`

Class for describing neutrino interaction attributes.

Stores parameters used to describe cross section and interaction length of a specific neutrino interaction.

Parameters **c** : float

Cross section energy coefficient.

p : float

Cross section energy exponent.

Notes

Neutrino intractions based on the GQRS Ultrahigh-Energy Neutrino Interactions Paper *[R17]*.

References

[R17]

Attributes

times, values	(ndarray) 1D arrays of times and corresponding values which define the signal.
value_type	Type of signal, representing the units of the values.
ValueTypes	(Enum) Different value types available for *value_type* of signal objects.
dt	
frequencies	
spectrum	
envelope	

pyrex.particle.CC_NUBAR

pyrex.particle.**CC_NUBAR = <pyrex.particle.NeutrinoInteraction object>**

Class for describing neutrino interaction attributes.

Stores parameters used to describe cross section and interaction length of a specific neutrino interaction.

Parameters **c** : float

Cross section energy coefficient.

p : float

Cross section energy exponent.

Notes

Neutrino intractions based on the GQRS Ultrahigh-Energy Neutrino Interactions Paper *[R16]*.

References

[R16]

Attributes

times, values	(ndarray) 1D arrays of times and corresponding values which define the signal.
value_type	Type of signal, representing the units of the values.
ValueTypes	(Enum) Different value types available for *value_type* of signal objects.
dt	
frequencies	
spectrum	
envelope	

pyrex.particle.NC_NUBAR

pyrex.particle.**NC_NUBAR = <pyrex.particle.NeutrinoInteraction object>**

Class for describing neutrino interaction attributes.

Stores parameters used to describe cross section and interaction length of a specific neutrino interaction.

Parameters **c** : float

Cross section energy coefficient.

p : float

Cross section energy exponent.

Notes

Neutrino intractions based on the GQRS Ultrahigh-Energy Neutrino Interactions Paper *[R18]*.

References

[R18]

Attributes

times, values	(ndarray) 1D arrays of times and corresponding values which define the signal.
value_type	Type of signal, representing the units of the values.
ValueTypes	(Enum) Different value types available for *value_type* of signal objects.
dt	
frequencies	
spectrum	
envelope	

pyrex.particle.Particle

class `pyrex.particle.`**`Particle`**(*vertex*, *direction*, *energy*)

Class for storing particle attributes.

Parameters **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.

Attributes

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.

pyrex.particle.random_direction

`pyrex.particle.`**`random_direction`**()

Generate an arbitrary cartesian unit vector.

Returns array_like

(Unit) vector with a uniformly distributed random direction.

Notes

Generates random vector direction by pulling from uniform distributions for -1<cos(theta)<1 and 0<phi<2*pi.

pyrex.particle.ShadowGenerator

class pyrex.particle.**ShadowGenerator**(*dx*, *dy*, *dz*, *energy*)

Class to generate neutrino vertices with Earth shadowing.

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.

See also:

`pyrex.slant_depth` Calculates the material thickness of a chord cutting through Earth.

Attributes

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_generator	(function) Function returning energy (GeV) of the neutrinos by successive function calls.
count	(int) Number of neutrinos produced by the generator.

Methods

`create_particle()`	Generate a neutrino.

pyrex.particle.ListGenerator

class pyrex.particle.**ListGenerator**(*particles*, *loop=True*)

Class to generate neutrino vertices from a list.

Generates neutrinos by simply pulling them from a list of `Particle` 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 **particles** : Particle or list of Particle

List of `Particle` objects to draw from. If only a single `Particle` object is given, creates a list of that particle alone.

loop : boolean, optional

Whether or not to return to the start of the list after throwing the last `Particle`. If `False`, raises an error if trying to throw after the last `Particle`.

Attributes

particles	(list of Particle) List to draw `Particle` objects from, sequentially.
loop	(boolean) Whether or not to loop through the list more than once.

Methods

`create_particle()`	Generate a neutrino.

pyrex.particle.FileGenerator

class `pyrex.particle.`**`FileGenerator`**(*files*)

Class to generate neutrino vertices from numpy file(s).

Generates neutrinos by pulling their vertex, direction, and energy from a (list of) .npz file(s). Each file must have three arrays, containing the vertices, directions, and energies respectively so the first particle will have properties given by the first elements of these three 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.

Attributes

files	(list of str) List of file names containing neutrino information.
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.

Methods

`create_particle()`	Generate a neutrino.

6.2.9 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.

`EventKernel`(generator, antennas[, ...])	High-level kernel for controlling event simulation.

pyrex.kernel.EventKernel

class pyrex.kernel.**EventKernel**(*generator*, *antennas*, *ice_model=<class 'pyrex.ice_model.AntarcticIce'>*, *ray_tracer=<class 'pyrex.ray_tracing.SpecializedRayTracer'>*, *signal_times=array([-2.000e-08, -1.995e-08, -1.990e-08, ..., 7.985e-08, 7.990e-08, 7.995e-08])*)

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_times : array_like, optional

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

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_particle` method which takes no arguments and returns a *Particle* object with `vertex`, `direction`, and `energy` 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 revelaing 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.

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

Methods

event()	Create a neutrino event and run it through the simulation chain.

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 *ARAString* class creates a string of alternating *HpolAntenna* and *VpolAntenna* ojbects, 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

HpolAntenna(name, position, power_threshold)	ARA Hpol ("quad-slot") antenna system with front-end processing.
VpolAntenna(name, position, power_threshold)	ARA Vpol ("bicone" or "birdcage") antenna system with front-end processing.
ARAString(x, y[, antennas_per_string, . . .])	String of ARA Hpol and Vpol antennas.

Continued on next page

Table 6.67 – continued from previous page

PhasedArrayString(x, y[, …])	Phased string of closely-packed antennas.
RegularStation(x, y[, strings_per_station, …])	Station geometry with strings evenly spaced radially around the center.
AlbrechtStation(x, y[, station_diameter, …])	Station geometry with center phased string and some outrigger strings.
HexagonalGrid([stations, …])	Hexagonal grid of stations or strings.

pyrex.custom.ara.HpolAntenna

class pyrex.custom.ara.**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.

ARAAntenna Antenna class to be used for ARA antennas.

Attributes

`is_hit`	Boolean of whether the antenna system has been triggered.
`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.

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).

Methods

`clear`([reset_noise])	Reset the antenna system to an empty state.
`front_end`(signal)	Apply front-end processes to a signal and return the output.
`full_waveform`(times)	Signal + noise (if `noisy`) for the given times.
`is_hit_during`(times)	Check if the antenna system is triggered in a time range.
`receive`(signal[, direction, polarization, ...])	Process and store an incoming signal.
`setup_antenna`([center_frequency, bandwidth, ...])	Setup the antenna by passing along its init arguments.
`trigger`(signal)	Check if the antenna system triggers on a given signal.
`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*, *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.

ARAAntenna Antenna class to be used for ARA antennas.

Attributes

`is_hit`	Boolean of whether the antenna system has been triggered.
`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.

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).

Methods

`clear`([reset_noise])	Reset the antenna system to an empty state.
`front_end`(signal)	Apply front-end processes to a signal and return the output.
`full_waveform`(times)	Signal + noise (if `noisy`) for the given times.
`is_hit_during`(times)	Check if the antenna system is triggered in a time range.
`receive`(signal[, direction, polarization, ...])	Process and store an incoming signal.

Continued on next page

Table 6.71 – continued from previous page

`setup_antenna`([center_frequency, bandwidth, ...])	Setup the antenna by passing along its init arguments.
`trigger`(signal)	Check if the antenna system triggers on a given signal.
`tunnel_diode`(signal)	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

`build_antennas`(power_threshold[, …])	Creates antenna objects at the set antenna positions.
`clear`([reset_noise])	Reset the detector to an empty state.
`set_positions`(x, y[, antennas_per_string, …])	Generates antenna positions along the string.
`triggered`([antenna_requirement])	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*, *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

`build_antennas`(power_threshold[, . . .])	Creates antenna objects at the set antenna positions.
`clear`([reset_noise])	Reset the detector to an empty state.
`set_positions`(x, y[, antennas_per_string, . . .])	Generates antenna positions along the string.
`triggered`(beam_threshold[, delays, angles])	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.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 whenver 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

`build_antennas`(*args, **kwargs)	Creates antenna objects at the set antenna positions.
`clear`([reset_noise])	Reset the detector to an empty state.
`set_positions`(x, y[, strings_per_station, . . .])	Generates antenna positions around the station.
`triggered`([polarized_antenna_requirement])	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.

ARAString 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 whenver 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

`build_antennas`(*args, **kwargs)	Creates antenna objects at the set antenna positions.
`clear`([reset_noise])	Reset the detector to an empty state.
`set_positions`(x, y[, station_diameter, …])	Generates antenna positions around the station.
`triggered`(beam_threshold[, …])	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*, *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.

`ARAString` 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 whenver 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

`build_antennas`(*args, **kwargs)	Creates antenna objects at the set antenna positions.
`clear`([reset_noise])	Reset the detector to an empty state.

Continued on next page

Table 6.76 – continued from previous page

`set_positions`([stations, . . .])	Generates antenna positions around the station.
`triggered`([station_requirement])	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.

`_read_directionality_data`(filename)	Gather antenna directionality data from a data file.
`_read_filter_data`(filename)	Gather frequency-dependent filtering data from a data file.
`ARAAntenna`(position, center_frequency, . . .)	Antenna class to be used for ARA antennas.
`ARAAntennaSystem`(name, position, power_threshold)	Antenna system extending base ARA antenna with front-end processing.
`HpolAntenna`(name, position, power_threshold)	ARA Hpol ("quad-slot") antenna system with front-end processing.
`VpolAntenna`(name, position, power_threshold)	ARA Vpol ("bicone" or "birdcage") antenna system with front-end processing.

pyrex.custom.ara.antenna._read_directionality_data

`pyrex.custom.ara.antenna.`**`_read_directionality_data`**(*filename*)

Gather antenna directionality data from a 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 "trans : Y".

Parameters **filename** : str

Name of the data file.

Returns dict

Dictionary containing the data with keys (freq, theta, phi) and values (gain, phase).

set

Set of unique frequencies appearing in the data keys.

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 dict

Dictionary containing the data with keys (freq) and values (gain, phase).

pyrex.custom.ara.antenna.ARAAntenna

class pyrex.custom.ara.antenna.**ARAAntenna**(*position*, *center_frequency*, *bandwidth*, *resistance*, *orientation=(0, 0, 1)*, *efficiency=1*, *noisy=True*, *unique_noise_waveforms=10*, *directionality_data=None*, *directionality_freqs=None*)

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.

Parameters **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.

directionality_data : None or dict, optional

Dictionary containing data on the directionality of the antenna. If `None`, behavior is undefined.

directionality_freqs : None or set, optional

Set of frequencies in the directionality data `dict` keys. If `None`, calculated automatically from *directionality_data*.

See also:

`pyrex.Antenna` Base class for antennas.

Attributes

`is_hit`	Boolean of whether the antenna has been triggered.
`waveforms`	Signal + noise (if `noisy`) for each triggered antenna hit.
`all_waveforms`	Signal + noise (if `noisy`) for all antenna hits.

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.

Methods

`clear`([reset_noise])	Reset the antenna to an empty state.
`directional_gain`(theta, phi)	Calculate the (complex) directional gain of the antenna.
`full_waveform`(times)	Signal + noise (if `noisy`) for the given times.
`generate_directionality_gains`(theta, phi)	Generate the (complex) frequency-dependent directional gains.
`interpolate_filter`(frequencies)	Generate interpolated filter values for given frequencies.
`is_hit_during`(times)	Check if the antenna is triggered in a time range.
`make_noise`(times)	Creates a noise signal over the given times.
`polarization_gain`(polarization)	Calculate the (complex) polarization gain of the antenna.
`receive`(signal[, direction, polarization, . . .])	Process and store an incoming signal.
`response`(frequencies)	Calculate the (complex) frequency response of the antenna.
`set_orientation`([z_axis, x_axis])	Sets the orientation of the antenna.
`trigger`(signal)	Check if the antenna triggers on a given signal.

pyrex.custom.ara.antenna.ARAAntennaSystem

class pyrex.custom.ara.antenna.**ARAAntennaSystem**(*name*, *position*, *power_threshold*, *directionality_data=None*, *directionality_freqs=None*, *orientation=(0, 0, 1)*, *amplification=1*, *amplifier_clipping=1*, *noisy=True*, *unique_noise_waveforms=10*)

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 **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.

directionality_data : None or dict, optional

Dictionary containing data on the directionality of the antenna. If `None`, behavior is undefined.

directionality_freqs : None or set, optional

Set of frequencies in the directionality data `dict` keys. If `None`, calculated automatically from *directionality_data*.

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

`is_hit`	Boolean of whether the antenna system has been triggered.
`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.

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).

Methods

`clear`([reset_noise])	Reset the antenna system to an empty state.
`front_end`(signal)	Apply front-end processes to a signal and return the output.
`full_waveform`(times)	Signal + noise (if `noisy`) for the given times.
`is_hit_during`(times)	Check if the antenna system is triggered in a time range.
`receive`(signal[, direction, polarization, ...])	Process and store an incoming signal.
`setup_antenna`([center_frequency, bandwidth, ...])	Setup the antenna by passing along its init arguments.
`trigger`(signal)	Check if the antenna system triggers on a given signal.
`tunnel_diode`(signal)	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.

ARAAntenna Antenna class to be used for ARA antennas.

Attributes

`is_hit`	Boolean of whether the antenna system has been triggered.
`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.

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).

Methods

`clear`([reset_noise])	Reset the antenna system to an empty state.
`front_end`(signal)	Apply front-end processes to a signal and return the output.
`full_waveform`(times)	Signal + noise (if `noisy`) for the given times.

Continued on next page

Table 6.83 – continued from previous page

`is_hit_during`(times)	Check if the antenna system is triggered in a time range.
`receive`(signal[, direction, polarization, . . .])	Process and store an incoming signal.
`setup_antenna`([center_frequency, bandwidth, . . .])	Setup the antenna by passing along its init arguments.
`trigger`(signal)	Check if the antenna system triggers on a given signal.
`tunnel_diode`(signal)	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.

`ARAAntenna` Antenna class to be used for ARA antennas.

Attributes

`is_hit`	Boolean of whether the antenna system has been triggered.
`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.

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).

Methods

`clear`([reset_noise])	Reset the antenna system to an empty state.
`front_end`(signal)	Apply front-end processes to a signal and return the output.
`full_waveform`(times)	Signal + noise (if `noisy`) for the given times.
`is_hit_during`(times)	Check if the antenna system is triggered in a time range.
`receive`(signal[, direction, polarization, …])	Process and store an incoming signal.
`setup_antenna`([center_frequency, bandwidth, …])	Setup the antenna by passing along its init arguments.
`trigger`(signal)	Check if the antenna system triggers on a given signal.
`tunnel_diode`(signal)	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.

`convert_hex_coords`(hex_coords[, unit])	Convert from hexagonal coordinates to Cartesian.
`ARAString`(x, y[, antennas_per_string, …])	String of ARA Hpol and Vpol antennas.
`PhasedArrayString`(x, y[, …])	Phased string of closely-packed antennas.
`RegularStation`(x, y[, strings_per_station, …])	Station geometry with strings evenly spaced radially around the center.
`AlbrechtStation`(x, y[, station_diameter, …])	Station geometry with center phased string and some outrigger strings.
`HexagonalGrid`([stations, …])	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

`build_antennas`(power_threshold[, …])	Creates antenna objects at the set antenna positions.
`clear`([reset_noise])	Reset the detector to an empty state.
`set_positions`(x, y[, antennas_per_string, …])	Generates antenna positions along the string.
`triggered`([antenna_requirement])	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

`build_antennas`(power_threshold[, . . .])	Creates antenna objects at the set antenna positions.
`clear`([reset_noise])	Reset the detector to an empty state.
`set_positions`(x, y[, antennas_per_string, . . .])	Generates antenna positions along the string.
`triggered`(beam_threshold[, delays, angles])	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 whenver 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

`build_antennas`(*args, **kwargs)	Creates antenna objects at the set antenna positions.
`clear`([reset_noise])	Reset the detector to an empty state.
`set_positions`(x, y[, strings_per_station, . . .])	Generates antenna positions around the station.
`triggered`([polarized_antenna_requirement])	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.

[`ARAString`] 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 whenver 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

`build_antennas`(*args, **kwargs)	Creates antenna objects at the set antenna positions.
`clear`([reset_noise])	Reset the detector to an empty state.
`set_positions`(x, y[, station_diameter, . . .])	Generates antenna positions around the station.
`triggered`(beam_threshold[, . . .])	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.

ARAString 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 whenver 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

`build_antennas`(*args, **kwargs)	Creates antenna objects at the set antenna positions.
`clear`([reset_noise])	Reset the detector to an empty state.
`set_positions`([stations, …])	Generates antenna positions around the station.
`triggered`([station_requirement])	Check if the detector is triggered based on its current state.

6.3.2 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 Nx by Ny where N is the total number of stations, Nx=floor(sqrt(N)), and Ny=floor(N/Nx).

Default Package Imports

EnvelopeHpol(name, position, trigger_threshold)	ARA Hpol ("quad-slot") antenna system with front-end processing.
EnvelopeVpol(name, position, trigger_threshold)	ARA Vpol ("bicone" or "birdcage") antenna system with front-end processing.
IREXString(x, y[, antennas_per_string, ...])	String of IREX Vpol antennas.
RegularStation(x, y[, strings_per_station, ...])	Station geometry with strings evenly spaced radially around the center.
CoxeterStation(x, y[, strings_per_station, ...])	Station geometry with center string and the rest evenly spaced radially.
StationGrid([stations, station_separation, ...])	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 uesd. 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

`is_hit`	Boolean of whether the antenna system has been triggered.
`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.

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.

Methods

`clear`([reset_noise])	Reset the antenna system to an empty state.
`envelopeless_front_end`(signal)	Apply front-end processes to a signal and return the output.
`front_end`(signal)	Apply front-end processes to a signal and return the output.
`full_waveform`(times)	Signal + noise (if noisy) for the given times.
`is_hit_during`(times)	Check if the antenna system is triggered in a time range.
`make_envelope`(signal)	Return the signal envelope based on the antenna’s `envelope_method`.

Continued on next page

Table 6.94 – continued from previous page

`receive`(signal[, direction, polarization, . . .])	Process and store an incoming signal.
`setup_antenna`([center_frequency, bandwidth, . . .])	Setup the antenna by passing along its init arguments.
`trigger`(signal)	Check if the antenna triggers on a given signal.
`tunnel_diode`(signal)	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 uesd. 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

`is_hit`	Boolean of whether the antenna system has been triggered.
`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.

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.

Methods

`clear`([reset_noise])	Reset the antenna system to an empty state.
`envelopeless_front_end`(signal)	Apply front-end processes to a signal and return the output.
`front_end`(signal)	Apply front-end processes to a signal and return the output.
`full_waveform`(times)	Signal + noise (if noisy) for the given times.
`is_hit_during`(times)	Check if the antenna system is triggered in a time range.
`make_envelope`(signal)	Return the signal envelope based on the antenna's `envelope_method`.
`receive`(signal[, direction, polarization, ...])	Process and store an incoming signal.
`setup_antenna`([center_frequency, bandwidth, ...])	Setup the antenna by passing along its init arguments.
`trigger`(signal)	Check if the antenna triggers on a given signal.
`tunnel_diode`(signal)	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

`build_antennas`(trigger_threshold[, ...])	Creates antenna objects at the set antenna positions.
`clear`([reset_noise])	Reset the detector to an empty state.
`set_positions`(x, y[, antennas_per_string, ...])	Generates antenna positions along the string.
`triggered`([antenna_requirement])	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 whenver 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

`build_antennas`(*args, **kwargs)	Creates antenna objects at the set antenna positions.
`clear`([reset_noise])	Reset the detector to an empty state.
`set_positions`(x, y[, strings_per_station, . . .])	Generates antenna positions around the station.
`triggered`([antenna_requirement, . . .])	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 whenver 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

`build_antennas`(*args, **kwargs)	Creates antenna objects at the set antenna positions.
`clear`([reset_noise])	Reset the detector to an empty state.
`set_positions`(x, y[, strings_per_station, ...])	Generates antenna positions around the station.
`triggered`([antenna_requirement, ...])	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, 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 whenver 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

`build_antennas`(*args, **kwargs)	Creates antenna objects at the set antenna positions.

Continued on next page

Table 6.100 – continued from previous page

`clear`([reset_noise])	Reset the detector to an empty state.
`set_positions`([stations, . . .])	Generates antenna positions around the station.
`triggered`([station_requirement])	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.

`basic_envelope_model`(signal[, cap, res])	Model of a basic diode-capacitor-resistor envelope circuit.
`bridge_rectifier_envelope_model`(signal[, . . .])	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:

```
 +-----------+
 |           |
 |       +---+---+
 |       |       |
 |       ^       D1
 |      D3       v
 |       |       |
Vin      +--gnd  +-----+---+---out
 |       |       |     |   |
 |      D4       ^     |   |
 |       v       D2    C1  R1
 |       |       |     |   |
 |       +---+---+     +---+
 |           |         |
 +-----------+        gnd
```

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.

DipoleTester(position, center_frequency, . . .)	Dipole antenna for IREX testing.
EnvelopeSystem(name, position, trigger_threshold)	Antenna system extending ARA antennas with an envelope circuit.

Continued on next page

Table 6.102 – continued from previous page

EnvelopeHpol(name, position, trigger_threshold)	ARA Hpol ("quad-slot") antenna system with front-end processing.
EnvelopeVpol(name, position, trigger_threshold)	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

`is_hit`	Boolean of whether the antenna has been triggered.
`waveforms`	Signal + noise (if `noisy`) for each triggered antenna hit.
`all_waveforms`	Signal + noise (if `noisy`) for all antenna hits.

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.

Methods

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

pyrex.custom.irex.antenna.EnvelopeSystem

class pyrex.custom.irex.antenna.**EnvelopeSystem**(*name*, *position*, *trigger_threshold*, *time_over_threshold=0*, *directionality_data=None*, *directionality_freqs=None*, *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 **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.

directionality_data : None or dict, optional

Dictionary containing data on the directionality of the antenna. If `None`, behavior is undefined.

directionality_freqs : None or set, optional

Set of frequencies in the directionality data `dict` keys. If `None`, calculated automatically from *directionality_data*.

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 uesd. 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

`is_hit`	Boolean of whether the antenna system has been triggered.
`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.

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.

Methods

`clear`([reset_noise])	Reset the antenna system to an empty state.
`envelopeless_front_end`(signal)	Apply front-end processes to a signal and return the output.
`front_end`(signal)	Apply front-end processes to a signal and return the output.
`full_waveform`(times)	Signal + noise (if noisy) for the given times.

Continued on next page

Table 6.106 – continued from previous page

`is_hit_during`(times)	Check if the antenna system is triggered in a time range.
`make_envelope`(signal)	Return the signal envelope based on the antenna's `envelope_method`.
`receive`(signal[, direction, polarization, . . .])	Process and store an incoming signal.
`setup_antenna`([center_frequency, bandwidth, . . .])	Setup the antenna by passing along its init arguments.
`trigger`(signal)	Check if the antenna triggers on a given signal.
`tunnel_diode`(signal)	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 uesd. 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

`is_hit`	Boolean of whether the antenna system has been triggered.
`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.

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.

Methods

`clear`([reset_noise])	Reset the antenna system to an empty state.
`envelopeless_front_end`(signal)	Apply front-end processes to a signal and return the output.
`front_end`(signal)	Apply front-end processes to a signal and return the output.
`full_waveform`(times)	Signal + noise (if noisy) for the given times.
`is_hit_during`(times)	Check if the antenna system is triggered in a time range.
`make_envelope`(signal)	Return the signal envelope based on the antenna's `envelope_method`.
`receive`(signal[, direction, polarization, …])	Process and store an incoming signal.
`setup_antenna`([center_frequency, bandwidth, …])	Setup the antenna by passing along its init arguments.

Continued on next page

Table 6.108 – continued from previous page

`trigger`(signal)	Check if the antenna triggers on a given signal.
`tunnel_diode`(signal)	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 uesd. 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

`is_hit`	Boolean of whether the antenna system has been triggered.
`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.

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.

Methods

`clear`([reset_noise])	Reset the antenna system to an empty state.
`envelopeless_front_end`(signal)	Apply front-end processes to a signal and return the output.
`front_end`(signal)	Apply front-end processes to a signal and return the output.
`full_waveform`(times)	Signal + noise (if noisy) for the given times.
`is_hit_during`(times)	Check if the antenna system is triggered in a time range.
`make_envelope`(signal)	Return the signal envelope based on the antenna's `envelope_method`.
`receive`(signal[, direction, polarization, . . .])	Process and store an incoming signal.
`setup_antenna`([center_frequency, bandwidth, . . .])	Setup the antenna by passing along its init arguments.
`trigger`(signal)	Check if the antenna triggers on a given signal.
`tunnel_diode`(signal)	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.

`IREXString`(x, y[, antennas_per_string, ...])	String of IREX Vpol antennas.
`RegularStation`(x, y[, strings_per_station, ...])	Station geometry with strings evenly spaced radially around the center.
`CoxeterStation`(x, y[, strings_per_station, ...])	Station geometry with center string and the rest evenly spaced radially.
`StationGrid`([stations, station_separation, ...])	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

`build_antennas`(trigger_threshold[, . . .])	Creates antenna objects at the set antenna positions.
`clear`([reset_noise])	Reset the detector to an empty state.
`set_positions`(x, y[, antennas_per_string, . . .])	Generates antenna positions along the string.
`triggered`([antenna_requirement])	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

`build_antennas`(*args, **kwargs)	Creates antenna objects at the set antenna positions.
`clear`([reset_noise])	Reset the detector to an empty state.
`set_positions`(x, y[, strings_per_station, ...])	Generates antenna positions around the station.
`triggered`([antenna_requirement, ...])	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

`build_antennas`(*args, **kwargs)	Creates antenna objects at the set antenna positions.

Continued on next page

Table 6.114 – continued from previous page

`clear`([reset_noise])	Reset the detector to an empty state.
`set_positions`(x, y[, strings_per_station, ...])	Generates antenna positions around the station.
`triggered`([antenna_requirement, ...])	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

`build_antennas`(*args, **kwargs)	Creates antenna objects at the set antenna positions.
`clear`([reset_noise])	Reset the detector to an empty state.
`set_positions`([stations, . . .])	Generates antenna positions around the station.
`triggered`([station_requirement])	Check if the detector is triggered based on its current state.

CHAPTER

SEVEN

VERSION HISTORY

7.1 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`.

7.2 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()`.

7.3 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).

7.4 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.

7.5 Version 1.4.0

New Features

- Implemented full ray tracing in the *RayTracer* and *RayTracePath* classes.

7.6 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 **:classs:'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).

7.7 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 `AksaryanSignal` 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.

7.8 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.

7.9 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.

7.10 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.

7.11 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.

7.12 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.

7.13 Version 1.0.3

New Features

- Added custom module to contain classes and functions specific to the IREX project.

7.14 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).

7.15 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.

7.16 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()`.

7.17 Version 0.0.0

Original PyREx python notebook written by Kael Hanson:

https://gist.github.com/physkael/898a64e6fbf5f0917584c6d31edf7940

CHAPTER

EIGHT

GITHUB README

8.1 PyREx - (Python package for an IceCube Radio Extension)

PyREx (**Py**thon package for an IceCube **R**adio **Ex**tension) is, as its name suggests, a Python package designed to simulate the measurement of Askaryan pulses via a radio antenna array around the IceCube South Pole Neutrino Observatory. The code is designed to be modular so that it can also be applied to other askaryan radio antennas (e.g. the ARA and ARIANA collaborations).

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+ and scipy version 0.19+. After installing python from https://www.python.org/downloads/, numpy and scipy can be installed with `pip` as follows, or by simply installing pyrex as specified in the next section.

```
pip install numpy>=1.13
pip install scipy>=0.19
```

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.

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. Before contributing, for more information please read the contribution page in the documentation.

8.1.5 Authors

- Ben Hokanson-Fasig

8.1.6 License

BIBLIOGRAPHY

[R3131] A. Connolly et al, ANITA Note #76, “Thermal Noise Studies: Toward A Time-Domain Model of the ANITA Trigger.” https://www.phys.hawaii.edu/elog/anita_notes/060228_110754/noise_simulation.ps

[R2121] Dziewonski, Adam M.; Anderson, Don L. (June 1981), “Preliminary reference Earth model.” Physics of the Earth and Planetary Interiors. **25** (4), 297–356 (1981).

[R5960] Jaime Alvarez-Muñiz et al, “Practical and accurate calculations of Askaryan radiation.” Physical Review D **84**, 103003 (2011).

[R6060] 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).

[R4748] Jaime Alvarez-Muñiz et al, “Practical and accurate calculations of Askaryan radiation.” Physical Review D **84**, 103003 (2011).

[R4848] 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).

[R7171] A. Connolly et al, ANITA Note #76, “Thermal Noise Studies: Toward A Time-Domain Model of the ANITA Trigger.” https://www.phys.hawaii.edu/elog/anita_notes/060228_110754/noise_simulation.ps

[R1313] Dziewonski, Adam M.; Anderson, Don L. (June 1981), “Preliminary reference Earth model.” Physics of the Earth and Planetary Interiors. **25** (4), 297–356 (1981).

[R1919] R. Gandhi et al, “Ultrahigh-Energy Neutrino Interactions.” Physical Review D **58**, 093009 (1998).

[R15] R. Gandhi et al, “Ultrahigh-Energy Neutrino Interactions.” Physical Review D **58**, 093009 (1998).

[R17] R. Gandhi et al, “Ultrahigh-Energy Neutrino Interactions.” Physical Review D **58**, 093009 (1998).

[R16] R. Gandhi et al, “Ultrahigh-Energy Neutrino Interactions.” Physical Review D **58**, 093009 (1998).

[R18] R. Gandhi et al, “Ultrahigh-Energy Neutrino Interactions.” Physical Review D **58**, 093009 (1998).

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

P

R

S

T

V