PyREx Documentation

Release 1.6.0

Ben Hokanson-Fasig

Jun 16, 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_generator=lambda: 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,
                           ice_model=pyrex.IceModel, 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.fftpack
import pyrex
```

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

2.1 Working with Signal Objects

The base `Signal` class is simply an array of times and an array of 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 a `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 electric 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 `resample` function 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 `with_times` function 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 `filter_frequencies` function 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 have strange behavior including having an imaginary part. To resolve that issue, pass `force_real=True` to the `filter_frequencies` function 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)
my_signal.filter_frequencies(lowpass_filter, force_real=True)
plt.plot(my_signal.times, my_signal.values)
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 `with_times` function:

```
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="new signal")
plt.plot(square_signal.times, square_signal.values, label="original signal")
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 large 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 inherited from 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 in order to simplify demonstration
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`. `is_hit` 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 `signals`, 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 `response`, `directional_gain`, and `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`). `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`). `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). 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 set_orientation method which takes z_axis and x_axis arguments. Finally, 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 a trigger method which is also expected to be overwritten. 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 a 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 receive function, 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 directional_gain calculation) and the polarization direction of the signal (used in 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: full_waveform and is_hit_during. Both of these methods take a time array as an argument and return the waveform Signal object for those times and 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 a 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 `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 is created as follows:

```python
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 `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, subclass the `AntennaSystem` class and define the `front_end` method. Optionally a trigger can be defined for the antenna system (by default it uses the antenna's trigger):

```python
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:

```python
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 to build a detector). Subclasses must define a `set_positions` method to assign vector positions to the self.antenna_positions attribute. By default `set_positions` will raise a `NotImplementedError`. Additionally subclasses may extend the default `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 `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 the preferred (currently just the basic `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 Particle Generation

PyREx includes `Particle` 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 some 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.

2.6 Ray Tracing

PyREx provides ray tracing in the `RayTracer` and `RayTracerPath` 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 `RayTracerPath` 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 `RayTracer.exists` and `RayTracer.solutions`. `RayTracer.exists` is a boolean value of whether or not path solutions exist between the launch and receiving points. `RayTracer.solutions` is the list of (zero or two) `RayTracerPath` 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 RayTracerPath class contains the attributes of the paths between points. The most useful properties of RayTracerPath are RayTracerPath.tof, RayTracerPath.path_length, RayTracerPath.emitted_direction, and RayTracerPath.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 the RayTracePath.attenuation() method which gives the attenuation of the signal at a given frequency (or frequencies), and the RayTracePath.coordinates property which gives the x, y, and z coordinates of the path (useful mostly for plotting, and are not garuanteed 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 by applying the frequency-dependent attenuation of 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()
```

2.7 Full Simulation

PyREx provides the EventKernel class to control a basic simulation including the creation of neutrinos, 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 a custom module for IREX (IceCube Radio Extension) accessible at `pyrex.custom.irex`. This module includes a more thorough `IREXAntennaSystem` class inheriting from the `AntennaSystem` class which adds a front-end for amplifying the signal, processing signal envelopes, and downsampling the result. It also includes an `IREXDetector` class designed to easily produce different geometries of `IREXAntennaSystem` objects.

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 available custom modules are `pyrex.custom.pyspice`, `pyrex.custom.irex`, `pyrex.custom.ara`, `pyrex.custom.ariana`, and `pyrex.custom.my_analysis`:

```
/path/to/site-packages/pyrex/
|-- __init__.py
|-- signals.py
|-- antenna.py
|-- ...
|-- custom/
|   |-- pyspice.py
|   |-- irex/
|   |   |-- __init__.py
|   |   |-- antenna.py
|   |   |-- ...

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

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

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

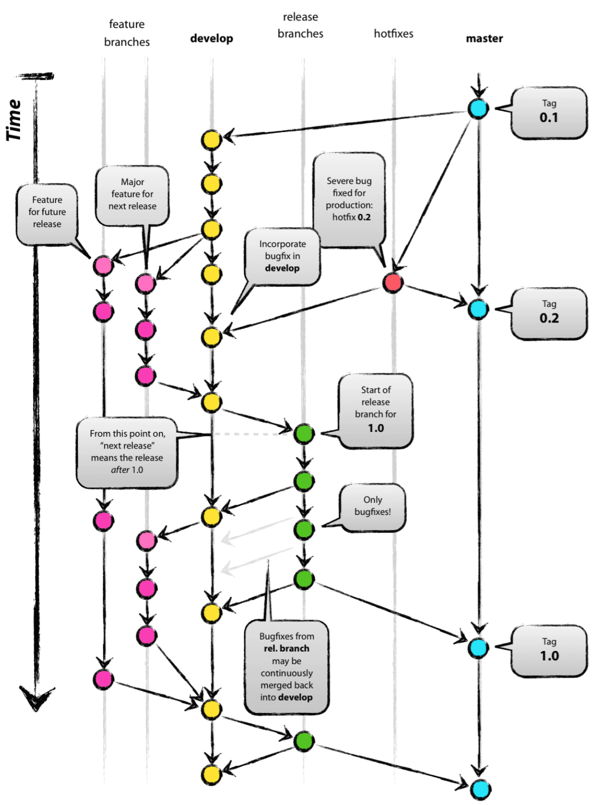
FIVE

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 branch for the feature.
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 *Package contents* section documents all classes and functions that are imported by PyREx under a `from pyrex import *` command. Next, the *Submodules* section is a full documentation of all the modules which make up the base PyREx package. And finally, the *PyREx Custom Subpackage* section documents the custom subpackages contained in PyREx by default.

6.1 Package contents

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

Base class for signals. Takes arrays of times and values (values array forced to size of times array by zero padding or slicing). Supports adding between signals with the same time values, resampling the signal, and calculating the signal's envelope.

class ValueTypes

Enum containing possible types (units) for signal values.

undefined = 0

voltage = 1

field = 2

power = 3

dt

Returns the spacing of the time array, or None if invalid.

envelope

Calculates envelope of the signal by Hilbert transform.

resample(*n*)

Resamples the signal into n points in the same time range.

with_times(*new_times*)

Returns a signal object representing this signal with a different times array. Uses numpy.iterp on values by default.

spectrum

Returns the FFT spectrum of the signal.

frequencies

Returns the FFT frequencies of the signal.

filter_frequencies(*freq_response*, *force_real=False*)

Applies the given frequency response function to the signal. Optionally can attempt to force real results manually if the filter is only specified in positive frequencies.

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

Class for signal with no amplitude (all values = 0)

with_times(*new_times*)
Returns a signal object representing this signal with a different times array. Returns EmptySignal for new times.

class pyrex.**FunctionSignal**(*times*, *function*, *value_type=<ValueTypes.undefined: 0>*)
Bases: `pyrex.signals.Signal`

Class for signals generated by a function

with_times(*new_times*)
Returns a signal object representing this signal with a different times array. Leverages knowledge of the function to properly interpolate and extrapolate.

pyrex.**AskaryanSignal**
alias of `FastAskaryanSignal`

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

Askaryan pulse binned to times from a particle shower with given energy (GeV) observed at angle theta (radians) from the shower axis. Optional parameters are the index of refraction n, and pulse offset to start time t0 (s). Returned signal values are electric fields (V/m).

Note that the amplitude of the pulse goes as 1/R, where R is the distance from source to observer. R is assumed to be 1 meter so that dividing by a different value produces the proper result.

vector_potential
Recover the vector_potential from the electric field. Mostly just for testing purposes.

RAC(*time*)
Calculates R * vector potential (A) at the Cherenkov angle in Vs at the given time (s).

charge_profile(*z*, *density=0.92*, *crit_energy=0.0786*, *rad_length=36.08*)
Calculates the longitudinal charge profile in the EM shower at distance z (m) with parameters for the density (g/cm^3), critical energy (GeV), and electron radiation length (g/cm^2) in ice.

max_length(*density=0.92*, *crit_energy=0.0786*, *rad_length=36.08*)
Calculates the maximum length (m) of an EM shower with parameters for the density (g/cm^3), critical energy (GeV), and electron radiation length (g/cm^2) in ice.

class pyrex.**ThermalNoise**(*times*, *f_band*, *f_amplitude=1*, *rms_voltage=None*, *temperature=None*, *resistance=None*, *n_freqs=0*)
Bases: `pyrex.signals.FunctionSignal`

Thermal Rayleigh noise in the frequency band f_band=[f_min,f_max] (Hz) at a given temperature (K) and resistance (ohms) or with a given RMS voltage (V). Optional parameters are f_amplitude (default 1) which can be a number or a function designating the amplitudes at each frequency, and n_freqs which is the number of frequencies to use (in f_band) for the calculation (default is based on the FFT bin size of the given times array). Returned signal values are voltages (V).

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 an antenna with a given position (m), temperature (K), allowable frequency range (Hz), total resistance (ohm) used for Johnson noise, and whether or not to include noise in the antenna's waveforms. Defines default trigger, frequency response, and signal reception functions that can be overwritten in base classes to customize the antenna.

set_orientation(*z_axis=(0, 0, 1)*, *x_axis=(1, 0, 0)*)

is_hit
Test for whether the antenna has been triggered.

is_hit_during(*times*)
Test for whether the antenna has been triggered during the given times array.

clear(*reset_noise=False*)
Reset the antenna to a state of having received no signals. Can optionally reset noise, which will reset the noise waveform so that a new signal arriving at the same time does not have the same noise.

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

all_waveforms
Signal + noise (if noisy) at all antenna hits, even those that didn't trigger.

full_waveform(*times*)
Signal + noise (if noisy) for the given times array.

make_noise(*times*)
Returns the noise signal generated by the antenna over the given array of times. Used to add noise to signal for production of the antenna's waveforms.

trigger(*signal*)
Function to determine whether or not the antenna is triggered by the given Signal object.

directional_gain(*theta*, *phi*)
Function to calculate the directive electric field gain of the antenna at given angles theta (polar) and phi (azimuthal) relative to the antenna's orientation.

polarization_gain(*polarization*)
Function to calculate the electric field gain due to polarization for a given polarization direction.

response(*frequencies*)
Function to return the frequency response of the antenna at the given frequencies (Hz). This function should return the response as imaginary numbers of the form A*exp(i*phi), where A is the amplitude response and phi is the phase shift.

receive(*signal*, *direction=None*, *polarization=None*, *force_real=False*)
Process incoming signal according to the filter function and store it to the signals list. Optionally applies directional gain if direction is specified, applies polarization gain if polarization is specified, and forces any frequency response filters to return real signals if specified. Subclasses may extend this fuction, but should likely end with super().receive(signal).

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*)
Bases: `pyrex.antenna.Antenna`

Antenna with a given name, position (m), center frequency (Hz), bandwidth (Hz), resistance (ohm), effective height (m), polarization direction, and trigger threshold (V).

trigger(*signal*)
Trigger on the signal if the maximum signal value is above the given threshold.

response(*frequencies*)
Butterworth filter response for the antenna's frequency range.

directional_gain(*theta*, *phi*)
Power gain of dipole antenna goes as sin(theta)^2, so electric field gain goes as sin(theta).

polarization_gain(*polarization*)
Polarization gain is simply the dot product of the polarization with the antenna's z-axis.

class pyrex.**AntennaSystem**(*antenna*)
Base class for an antenna system consisting of an antenna and some front-end processes.

setup_antenna(**args*, ***kwargs*)
Setup the antenna by passing along its init arguments. This function can be overwritten if desired, just make sure to assign the self.antenna attribute in the function.

front_end(*signal*)
This function should take the signal passed (from the antenna) and return the resulting signal after all processing by the antenna system's front-end. By default it just returns the given signal.

is_hit

is_hit_during(*times*)

signals

waveforms

all_waveforms

full_waveform(*times*)

receive(*signal*, *direction=None*, *polarization=None*, *force_real=False*)

clear(*reset_noise=False*)
Reset the antenna system to a state of having received no signals. Can optionally reset noise, which will reset the noise waveform so that a new signal arriving at the same time does not have the same noise.

trigger(*signal*)
Antenna system trigger. Should return True or False for whether the passed signal triggers the antenna system. By default just matches the antenna's trigger.

class pyrex.**Detector**(**args*, ***kwargs*)
Class for automatically generating antenna positions based on geometry criteria. The set_positions method creates a list of antenna positions and the build_antennas method is responsible for actually placing antennas at the generated positions. Once antennas are placed, the class can be directly iterated over to iterate over the antennas (as if it were just a list of antennas itself).

test_antenna_positions = True

set_positions(**args*, ***kwargs*)
Not implemented. Should generate positions for the antennas based on the given arguments and assign those positions to the antenna_positions attribute.

build_antennas(**args*, ***kwargs*)
Sets up antenna objects at the positions stored in the class. By default takes an antenna class and passes a position to the 'position' argument, followed by any other arguments to be passed to this class.

triggered(**args*, ***kwargs*)
Test for whether the detector is triggered based on the current state of the antennas.

clear(*reset_noise=False*)
Convenience method for clearing all antennas in the detector. Can optionally reset noise, which will reset the noise waveforms so that new signals arriving at the same time do not have the same noise.

pyrex.**IceModel**
alias of `AntarcticIce`

class pyrex.ice_model.**ArasimIce**
Bases: *pyrex.ice_model.AntarcticIce*

Class containing characteristics of ice at the south pole. In all cases, depth z is given with negative values in the ice and positive values above the ice. Ice model index is the same as used in the ARA collaboration's AraSim package.

k = 0.43

a = 0.0132

n0 = 1.78

atten_depths = [72.7412, 76.5697, 80.3982, 91.8836, 95.7121, 107.198, 118.683, 133.997

atten_lengths = [1994.67, 1952, 1896, 1842.67, 1797.33, 1733.33, 1680, 1632, 1586.67,

classmethod attenuation_length(*z*, *f*)
: Returns the attenuation length at depth z (m) and frequency f (Hz). Attenuation length not actually frequency dependent; according to AraSim always uses the 300 MHz value. Supports passing a numpy array of depths and/or frequencies. If both are passed as arrays, a 2-D array is returned where each row is a single depth and each column is a single frequency.

pyrex.**prem_density**(*r*)
: Returns the earth's density (g/cm^3) for a given radius r (m). Calculated by the Preliminary Earth Model (PREM). Supports passing a list of radii.

pyrex.**slant_depth**(*angle*, *depth*, *step=500*)
: Returns the material thickness (g/cm^2) for a chord cutting through earth at Nadir angle and starting at (positive-valued) depth (m). Can optionally specify the step size (m).

class pyrex.**Particle**(*vertex*, *direction*, *energy*)
: Class for storing particle attributes. Consists of a 3-D vertex (m), 3-D direction vector (automatically normalized), and an energy (GeV).

class pyrex.**ShadowGenerator**(*dx*, *dy*, *dz*, *energy*)
: Class to generate UHE neutrino vertices in (relatively) shallow detectors. Takes into accout Earth shadowing (sort of). energy should be either an energy in GeV or a function that returns an energy in GeV. Note that the x and y ranges in which particles are created are (-dx/2, dx/2) and (-dy/2, dy/2) while the z range is (-dz, 0).

create_particle()
: Creates a particle with random vertex in cube with a random direction.

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

class pyrex.ray_tracing.**SpecializedRayTracer**(*from_point*, *to_point*, *ice_model=<class 'pyrex.ice_model.AntarcticIce'>*, *dz=1*)
: Bases: *pyrex.ray_tracing.BasicRayTracer*

Ray tracer specifically for ice model with index of refraction n(z) = n0 - k*exp(a*z). Calculations performed using true integral evaluation. Ice model must use methods inherited from pyrex.AntarcticIce

solution_class
: alias of *SpecializedRayTracePath*

valid_ice_model

z_uniform

direct_r_max

peak_angle

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

class pyrex.ray_tracing.**SpecializedRayTracePath** (*parent_tracer*, *launch_angle*, *direct*)
Bases: *pyrex.ray_tracing.BasicRayTracePath*

Class for storing a single ray-trace solution betwen points, specifically for ice model with index of refraction n(z) = n0 - k*exp(a*z). Calculations performed using true integral evaluation (except attenuation). Ice model must use methods inherited from pyrex.AntarcticIce

uniformity_factor = 0.99999

beta_tolerance = 0.005

valid_ice_model

z_uniform

z_integral (*integrand*, *numerical=False*, *x_func=<function SpecializedRayTracePath.<lambda>>*)
Function for integrating a given integrand along the depths of the path.

path_length

tof

attenuation (*f*)
Returns the attenuation factor for a signal of frequency f (Hz) traveling along the path. Supports passing a list of frequencies.

coordinates

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])*)
Kernel for generation of events with a given particle generator, list of antennas, and optionally a non-default ice_model.

event ()
Generate particle, propagate signal through ice to antennas, process signal at antennas, and return the original particle.

6.2 Submodules

6.2.1 pyrex.signals module

Module containing classes for digital signal processing

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

Base class for signals. Takes arrays of times and values (values array forced to size of times array by zero padding or slicing). Supports adding between signals with the same time values, resampling the signal, and calculating the signal's envelope.

class ValueTypes
Bases: enum.Enum

Enum containing possible types (units) for signal values.

undefined = 0

voltage = 1

field = 2

power = 3

dt
Returns the spacing of the time array, or None if invalid.

envelope
Calculates envelope of the signal by Hilbert transform.

resample (*n*)
Resamples the signal into n points in the same time range.

with_times (*new_times*)
Returns a signal object representing this signal with a different times array. Uses numpy.iterp on values by default.

spectrum
Returns the FFT spectrum of the signal.

frequencies
Returns the FFT frequencies of the signal.

filter_frequencies (*freq_response*, *force_real=False*)
Applies the given frequency response function to the signal. Optionally can attempt to force real results manually if the filter is only specified in positive frequencies.

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

Class for signal with no amplitude (all values = 0)

with_times (*new_times*)
Returns a signal object representing this signal with a different times array. Returns EmptySignal for new times.

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

Class for signals generated by a function

with_times (*new_times*)
Returns a signal object representing this signal with a different times array. Leverages knowledge of the function to properly interpolate and extrapolate.

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

Askaryan pulse binned to times from a particle shower with given energy (GeV) observed at angle theta (radians) from the shower axis. Optional parameters are the index of refraction n, and pulse offset to start time t0 (s). Returned signal values are electric fields (V/m).

Note that the amplitude of the pulse goes as 1/R, where R is the distance from source to observer. R is assumed to be 1 meter so that dividing by a different value produces the proper result.

RAC (*time*)
Calculates R * vector potential at the Cherenkov angle in Vs at the given time (s).

charge_profile (*z*, *density=0.92*, *crit_energy=0.0786*, *rad_length=36.08*)
Calculates the longitudinal charge profile in the EM shower at distance z (m) with parameters for the density (g/cm^3), critical energy (GeV), and electron radiation length (g/cm^2) in ice.

max_length (*density=0.92*, *crit_energy=0.0786*, *rad_length=36.08*)
Calculates the maximum length (m) of an EM shower with parameters for the density (g/cm^3), critical energy (GeV), and electron radiation length (g/cm^2) in ice.

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

Bases: *`pyrex.signals.Signal`*

Askaryan pulse binned to times from a particle shower with given energy (GeV) observed at angle theta (radians) from the shower axis. Optional parameters are the index of refraction n, and pulse offset to start time t0 (s). Returned signal values are electric fields (V/m).

Note that the amplitude of the pulse goes as 1/R, where R is the distance from source to observer. R is assumed to be 1 meter so that dividing by a different value produces the proper result.

`vector_potential`

Recover the vector_potential from the electric field. Mostly just for testing purposes.

`RAC`(*time*)

Calculates R * vector potential (A) at the Cherenkov angle in Vs at the given time (s).

`charge_profile`(*z*, *density=0.92*, *crit_energy=0.0786*, *rad_length=36.08*)

Calculates the longitudinal charge profile in the EM shower at distance z (m) with parameters for the density (g/cm^3), critical energy (GeV), and electron radiation length (g/cm^2) in ice.

`max_length`(*density=0.92*, *crit_energy=0.0786*, *rad_length=36.08*)

Calculates the maximum length (m) of an EM shower with parameters for the density (g/cm^3), critical energy (GeV), and electron radiation length (g/cm^2) in ice.

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

alias of *`FastAskaryanSignal`*

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

Bases: *`pyrex.signals.Signal`*

Gaussian noise signal with standard deviation sigma

class `pyrex.signals.`**`ThermalNoise`**(*times*, *f_band*, *f_amplitude=1*, *rms_voltage=None*, *temperature=None*, *resistance=None*, *n_freqs=0*)

Bases: *`pyrex.signals.FunctionSignal`*

Thermal Rayleigh noise in the frequency band f_band=[f_min,f_max] (Hz) at a given temperature (K) and resistance (ohms) or with a given RMS voltage (V). Optional parameters are f_amplitude (default 1) which can be a number or a function designating the amplitudes at each frequency, and n_freqs which is the number of frequencies to use (in f_band) for the calculation (default is based on the FFT bin size of the given times array). Returned signal values are voltages (V).

6.2.2 pyrex.antenna module

Module containing antenna class capable of receiving signals

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*)

Bases: `object`

Base class for an antenna with a given position (m), temperature (K), allowable frequency range (Hz), total resistance (ohm) used for Johnson noise, and whether or not to include noise in the antenna's waveforms. Defines default trigger, frequency response, and signal reception functions that can be overwritten in base classes to customize the antenna.

`set_orientation`(*z_axis=(0, 0, 1)*, *x_axis=(1, 0, 0)*)

`is_hit`

Test for whether the antenna has been triggered.

is_hit_during (*times*)
Test for whether the antenna has been triggered during the given times array.

clear (*reset_noise=False*)
Reset the antenna to a state of having received no signals. Can optionally reset noise, which will reset the noise waveform so that a new signal arriving at the same time does not have the same noise.

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

all_waveforms
Signal + noise (if noisy) at all antenna hits, even those that didn't trigger.

full_waveform (*times*)
Signal + noise (if noisy) for the given times array.

make_noise (*times*)
Returns the noise signal generated by the antenna over the given array of times. Used to add noise to signal for production of the antenna's waveforms.

trigger (*signal*)
Function to determine whether or not the antenna is triggered by the given Signal object.

directional_gain (*theta*, *phi*)
Function to calculate the directive electric field gain of the antenna at given angles theta (polar) and phi (azimuthal) relative to the antenna's orientation.

polarization_gain (*polarization*)
Function to calculate the electric field gain due to polarization for a given polarization direction.

response (*frequencies*)
Function to return the frequency response of the antenna at the given frequencies (Hz). This function should return the response as imaginary numbers of the form A*exp(i*phi), where A is the amplitude response and phi is the phase shift.

receive (*signal*, *direction=None*, *polarization=None*, *force_real=False*)
Process incoming signal according to the filter function and store it to the signals list. Optionally applies directional gain if direction is specified, applies polarization gain if polarization is specified, and forces any frequency response filters to return real signals if specified. Subclasses may extend this fuction, but should likely end with super().receive(signal).

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*)

Bases: `pyrex.antenna.Antenna`

Antenna with a given name, position (m), center frequency (Hz), bandwidth (Hz), resistance (ohm), effective height (m), polarization direction, and trigger threshold (V).

trigger (*signal*)
Trigger on the signal if the maximum signal value is above the given threshold.

response (*frequencies*)
Butterworth filter response for the antenna's frequency range.

directional_gain (*theta*, *phi*)
Power gain of dipole antenna goes as sin(theta)^2, so electric field gain goes as sin(theta).

polarization_gain (*polarization*)
Polarization gain is simply the dot product of the polarization with the antenna's z-axis.

6.2.3 pyrex.detector module

Module containing higher-level AntennaSystem and Detector classes

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

Bases: `object`

Base class for an antenna system consisting of an antenna and some front-end processes.

setup_antenna(**args*, ***kwargs*)

Setup the antenna by passing along its init arguments. This function can be overwritten if desired, just make sure to assign the self.antenna attribute in the function.

front_end(*signal*)

This function should take the signal passed (from the antenna) and return the resulting signal after all processing by the antenna system's front-end. By default it just returns the given signal.

is_hit

is_hit_during(*times*)

signals

waveforms

all_waveforms

full_waveform(*times*)

receive(*signal*, *direction=None*, *polarization=None*, *force_real=False*)

clear(*reset_noise=False*)

Reset the antenna system to a state of having received no signals. Can optionally reset noise, which will reset the noise waveform so that a new signal arriving at the same time does not have the same noise.

trigger(*signal*)

Antenna system trigger. Should return True or False for whether the passed signal triggers the antenna system. By default just matches the antenna's trigger.

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

Bases: `object`

Class for automatically generating antenna positions based on geometry criteria. The set_positions method creates a list of antenna positions and the build_antennas method is responsible for actually placing antennas at the generated positions. Once antennas are placed, the class can be directly iterated over to iterate over the antennas (as if it were just a list of antennas itself).

test_antenna_positions = True

set_positions(**args*, ***kwargs*)

Not implemented. Should generate positions for the antennas based on the given arguments and assign those positions to the antenna_positions attribute.

build_antennas(**args*, ***kwargs*)

Sets up antenna objects at the positions stored in the class. By default takes an antenna class and passes a position to the 'position' argument, followed by any other arguments to be passed to this class.

triggered(**args*, ***kwargs*)

Test for whether the detector is triggered based on the current state of the antennas.

clear(*reset_noise=False*)

Convenience method for clearing all antennas in the detector. Can optionally reset noise, which will reset the noise waveforms so that new signals arriving at the same time do not have the same noise.

6.2.4 pyrex.ice_model module

Module containing ice models. Ice model classes contains static and class methods for convenience. IceModel class is set to the preferred ice model.

class pyrex.ice_model.**AntarcticIce**

Bases: object

Class containing characteristics of ice at the south pole. In all cases, depth z is given with negative values in the ice and positive values above the ice. Index of refraction goes as n(z)=n0-k*exp(az).

k = 0.43

a = 0.0132

n0 = 1.78

thickness = 2850

classmethod gradient(*z*)

Returns the gradient of the index of refraction at depth z (m).

classmethod index(*z*)

Returns the medium's index of refraction, n, at depth z (m). Supports passing a numpy array of depths.

classmethod depth_with_index(*n*)

Returns the depth z (m) at which the medium has the given index of refraction (inverse of index function, assumes index function is monotonic so only one solution exists). Supports passing a numpy array of indices.

static temperature(*z*)

Returns the temperature (K) of the ice at depth z (m). Supports passing a numpy array of depths.

classmethod attenuation_length(*z, f*)

Returns the attenuation length at depth z (m) and frequency f (Hz). Supports passing a numpy array of depths and/or frequencies. If both are passed as arrays, a 2-D array is returned where each row is a single depth and each column is a single frequency.

class pyrex.ice_model.**NewcombIce**

Bases: *pyrex.ice_model.AntarcticIce*

Class inheriting from AntarcticIce, with new attenuation_length function based on Matt Newcomb's fit (DOESN'T CURRENTLY WORK).

k = 0.438

a = 0.0132

n0 = 1.758

classmethod attenuation_length(*z, f*)

Returns the attenuation length at depth z (m) and frequency f (MHz) by Matt Newcomb's fit (DOESN'T CURRENTLY WORK - USE BOGORODSKY).

class pyrex.ice_model.**ArasimIce**

Bases: *pyrex.ice_model.AntarcticIce*

Class containing characteristics of ice at the south pole. In all cases, depth z is given with negative values in the ice and positive values above the ice. Ice model index is the same as used in the ARA collaboration's AraSim package.

k = 0.43

a = 0.0132

n0 = 1.78

atten_depths = [72.7412, 76.5697, 80.3982, 91.8836, 95.7121, 107.198, 118.683, 133.997

atten_lengths = [1994.67, 1952, 1896, 1842.67, 1797.33, 1733.33, 1680, 1632, 1586.67,

classmethod attenuation_length(*z*, *f*)

Returns the attenuation length at depth z (m) and frequency f (Hz). Attenuation length not actually frequency dependent; according to AraSim always uses the 300 MHz value. Supports passing a numpy array of depths and/or frequencies. If both are passed as arrays, a 2-D array is returned where each row is a single depth and each column is a single frequency.

pyrex.ice_model.**IceModel**

alias of *AntarcticIce*

6.2.5 pyrex.earth_model module

Module containing earth model. Uses PREM for density as a function of radius and a simple integrator for calculation of the slant depth as a function of nadir angle.

pyrex.earth_model.**prem_density**(*r*)

Returns the earth's density (g/cm^3) for a given radius r (m). Calculated by the Preliminary Earth Model (PREM). Supports passing a list of radii.

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

Returns the material thickness (g/cm^2) for a chord cutting through earth at Nadir angle and starting at (positive-valued) depth (m). Can optionally specify the step size (m).

6.2.6 pyrex.particle module

Module for particles (namely neutrinos) and neutrino interactions in the ice. Interactions include Earth shadowing (absorption) effect.

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

Bases: object

Class for neutrino interaction attributes.

cross_section(*E*)

Return the cross section (cm^2) at a given energy E (GeV).

interaction_length(*E*)

Return the interaction length (cm) in water equivalent at a given energy E (GeV).

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

Bases: object

Class for storing particle attributes. Consists of a 3-D vertex (m), 3-D direction vector (automatically normalized), and an energy (GeV).

pyrex.particle.**random_direction**()

Generate an arbitrary 3D unit vector.

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

Bases: object

Class to generate UHE neutrino vertices in (relatively) shallow detectors. Takes into accout Earth shadowing (sort of). energy should be either an energy in GeV or a function that returns an energy in GeV. Note that the x and y ranges in which particles are created are (-dx/2, dx/2) and (-dy/2, dy/2) while the z range is (-dz, 0).

create_particle()
Creates a particle with random vertex in cube with a random direction.

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

Class to generate 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.

create_particle()
Pulls next particle from the list.

class pyrex.particle.**FileGenerator**(*files*)
Bases: object

Class to generate 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.

create_particle()
Pulls the next particle from the file(s).

6.2.7 pyrex.ray_tracing module

Module containing class for ray tracing through the ice.

class pyrex.ray_tracing.**BasicRayTracePath**(*parent_tracer*, *launch_angle*, *direct*)
Bases: *pyrex.internal_functions.LazyMutableClass*

Class for storing a single ray-trace solution betwen points. Calculations preformed by integrating z-steps of size dz. Most properties lazily evaluated to save on re-computation time.

z_turn_proximity
Parameter for how closely path approaches z_turn. Necessary to avoid diverging integrals.

z0
Depth of the launching point.

z1
Depth of the receiving point.

n0

rho

phi

beta

z_turn

emitted_direction

received_direction

theta(*z*)
Polar angle of the ray at given depth or array of depths.

z_integral(*integrand*)
Returns the integral of the integrand (a function of z) along the path.

path_length

tof

fresnel

attenuation (*f*)
Returns the attenuation factor for a signal of frequency f (Hz) traveling along the path. Supports passing a list of frequencies.

propagate (*signal*)
Applies attenuation to the signal along the path.

coordinates

class pyrex.ray_tracing.**SpecializedRayTracePath** (*parent_tracer*, *launch_angle*, *direct*)
Bases: *pyrex.ray_tracing.BasicRayTracePath*

Class for storing a single ray-trace solution betwen points, specifically for ice model with index of refraction n(z) = n0 - k*exp(a*z). Calculations performed using true integral evaluation (except attenuation). Ice model must use methods inherited from pyrex.AntarcticIce

uniformity_factor = 0.99999

beta_tolerance = 0.005

valid_ice_model

z_uniform

z_integral (*integrand*, *numerical=False*, *x_func=<function SpecializedRayTracePath.<lambda>>*)
Function for integrating a given integrand along the depths of the path.

path_length

tof

attenuation (*f*)
Returns the attenuation factor for a signal of frequency f (Hz) traveling along the path. Supports passing a list of frequencies.

coordinates

class pyrex.ray_tracing.**BasicRayTracer** (*from_point*, *to_point*, *ice_model=<class 'pyrex.ice_model.AntarcticIce'>*, *dz=1*)
Bases: *pyrex.internal_functions.LazyMutableClass*

Class for proper ray tracing. Calculations performed by integrating z-steps with size dz. Most properties lazily evaluated to save on re-computation time.

solution_class
alias of *BasicRayTracePath*

z_turn_proximity
Parameter for how closely path approaches z_turn. Necessary to avoid diverging integrals.

z0
Depth of lower point. Ray tracing performed as if launching from lower point to higher point.

z1
Depth of higher point. Ray tracing performed as if launching from lower point to higher point.

n0

rho

max_angle

peak_angle

direct_r_max

indirect_r_max

exists

expected_solutions

solutions

direct_angle

indirect_angle_1

indirect_angle_2

static angle_search(*true_r*, *r_function*, *min_angle*, *max_angle*, *tolerance=1e-12*, *max_iterations=100*)
Root-finding algorithm.

class pyrex.ray_tracing.**SpecializedRayTracer**(*from_point*, *to_point*, *ice_model=<class 'pyrex.ice_model.AntarcticIce'>*, *dz=1*)
Bases: *pyrex.ray_tracing.BasicRayTracer*

Ray tracer specifically for ice model with index of refraction n(z) = n0 - k*exp(a*z). Calculations performed using true integral evaluation. Ice model must use methods inherited from pyrex.AntarcticIce

solution_class
alias of *SpecializedRayTracePath*

valid_ice_model

z_uniform

direct_r_max

peak_angle

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

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

class pyrex.ray_tracing.**PathFinder**(*ice_model*, *from_point*, *to_point*)
Bases: object

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

exists
Boolean of whether path exists based on basic total internal reflection calculation.

emitted_ray
Direction in which ray is emitted.

received_ray
Direction from which ray is received.

path_length
Length of the path (m).

tof
Time of flight (s) for a particle along the path. Calculated using default values of self.time_of_flight()

time_of_flight(*n_steps=100*)
Time of flight (s) for a particle along the path.

attenuation (*f*, *n_steps=100*)
Returns the attenuation factor for a signal of frequency f (Hz) traveling along the path. Supports passing a list of frequencies.

propagate (*signal*)
Applies attenuation to the signal along the path.

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

Class for pseudo ray tracing of ray reflected off ice surface. Just uses straight-line paths.

get_bounce_point (*reflection_depth=0*)
Calculation of point at which signal is reflected by the ice surface (z=0).

exists
Boolean of whether path exists based on whether its sub-paths exist and whether it could reflect off the ice surface.

emitted_ray
Direction in which ray is emitted.

received_ray
Direction from which ray is received.

path_length
Length of the path (m).

tof
Time of flight (s) for a particle along the path. Calculated using default values of self.time_of_flight()

time_of_flight (*n_steps=100*)
Time of flight (s) for a particle along the path.

attenuation (*f*, *n_steps=100*)
Returns the attenuation factor for a signal of frequency f (Hz) traveling along the path. Supports passing a list of frequencies.

propagate (*signal*)
Applies attenuation to the signal along the path.

6.2.8 pyrex.kernel module

Module for the simulation kernel. Includes neutrino generation, ray tracking (no raytracing yet), and hit generation.

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])*)
Bases: object

Kernel for generation of events with a given particle generator, list of antennas, and optionally a non-default ice_model.

event ()
Generate particle, propagate signal through ice to antennas, process signal at antennas, and return the original particle.

6.2.9 pyrex.internal_functions module

Helper functions for use in PyREx modules.

`pyrex.internal_functions.`**`normalize`**(*vector*)
Returns the normalized form of the given vector.

`pyrex.internal_functions.`**`flatten`**(*iterator*, *dont_flatten=()*)
Flattens all iterable elements in the given iterator recursively and returns the resulting flat iterator. Can optionally be passed a list of classes to avoid flattening. Will not flatten strings or bytes due to recursion errors.

`pyrex.internal_functions.`**`mirror_func`**(*match_func*, *run_func*, *self=None*)
Returns a function which operates like run_func, but has all the attributes of match_func. If self argument is not None, it will be passed as the first argument to run_func.

`pyrex.internal_functions.`**`lazy_property`**(*fn*)
Decorator that makes a property lazily evaluated.

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

Class whose properties can be lazily evaluated by using lazy_property decorator, but will re-evaluate lazy properties if any of its specified static_attributes change. By default, static_attributes is set to all attributes of the class at the time of the init call.

6.3 PyREx Custom Subpackage

Note that more modules may be available as plug-ins, see *Custom Sub-Package*.

6.3.1 pyrex.custom.pyspice module

Module containing setup and wrappers for PySpice module into PyREx

6.3.2 pyrex.custom.irex package

Customizations of pyrex package specific to IREX (IceCube Radio Extension)

pyrex.custom.irex.antenna module

Module containing customized antenna classes for IREX

`class` `pyrex.custom.irex.antenna.`**`IREXAntenna`**(*position*, *center_frequency*, *bandwidth*, *resistance*, *orientation=(0, 0, 1)*, *effective_height=None*, *noisy=True*, *unique_noise_waveforms=10*)
Bases: `pyrex.antenna.Antenna`

Antenna to be used in IREX. Has a position (m), center frequency (Hz), bandwidth (Hz), resistance (ohm), effective height (m), and polarization direction.

`response`(*frequencies*)
Butterworth filter response for the antenna's frequency range.

`directional_gain`(*theta*, *phi*)
Power gain of dipole antenna goes as sin(theta)^2, so electric field gain goes as sin(theta).

polarization_gain (*polarization*)
Polarization gain is simply the dot product of the polarization with the antenna's z-axis.

class pyrex.custom.irex.antenna.**IREXAntennaSystem** (*name*, *position*, *trigger_threshold*, *time_over_threshold=0*, *orientation=(0, 0, 1)*, *amplification=1*, *amplifier_clipping=3*, *noisy=True*, *unique_noise_waveforms=10*, *envelope_method='analytic'*)

Bases: *pyrex.detector.AntennaSystem*

IREX antenna system consisting of dipole antenna, low-noise amplifier, optional bandpass filter, and envelope circuit.

setup_antenna (*center_frequency=250000000.0*, *bandwidth=300000000.0*, *resistance=100*, *orientation=(0, 0, 1)*, *effective_height=None*, *noisy=True*, *unique_noise_waveforms=10*)
Sets attributes of the antenna including center frequency (Hz), bandwidth (Hz), resistance (ohms), orientation, and effective height (m).

make_envelope (*signal*)
Return the signal envelope based on the antenna's envelope_method.

front_end (*signal*)
Apply the front-end processing of the antenna signal, including amplification, clipping, and envelope processing.

all_waveforms

full_waveform (*times*)

trigger (*signal*)

pyrex.custom.irex.detector module

Module containing customized detector geometry classes for IREX

class pyrex.custom.irex.detector.**IREXString** (*x*, *y*, *antennas_per_string=2*, *antenna_separation=50*, *lowest_antenna=-100*)

Bases: *pyrex.detector.Detector*

String of IREXAntennas. Sets positions of antennas on string based on the given arguments. Sets build_antennas method for setting antenna characteristics.

set_positions (*x*, *y*, *antennas_per_string=2*, *antenna_separation=50*, *lowest_antenna=-100*)
Generates antenna positions along the string.

build_antennas (*trigger_threshold*, *time_over_threshold=0*, *amplification=1*, *naming_scheme=<function IREXString.<lambda>>*, *orientation_scheme=<function IREXString.<lambda>>*, *noisy=True*, *unique_noise_waveforms=10*, *envelope_method='analytic'*)
Sets up IREXAntennaSystems at the positions stored in the class. Takes as arguments the trigger threshold, optional time over threshold, and whether to add noise to the waveforms. Other optional arguments include a naming scheme and orientation scheme which are functions taking the antenna index i and the antenna object. The naming scheme should return the name and the orientation scheme should return the orientation z-axis and x-axis of the antenna.

triggered (*antenna_requirement=1*)
Test whether the number of hit antennas meets the given antenna trigger requirement.

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*)

Bases: *pyrex.detector.Detector*

Station geometry with a number of strings evenly spaced radially around the station center. Supports any string type and passes extra keyword arguments on to the string class.

set_positions(*x*, *y*, *strings_per_station=4*, *station_diameter=50*, *string_type=<class 'pyrex.custom.irex.detector.IREXString'>*, ***string_kwargs*)
Generates string positions around the station.

triggered(*antenna_requirement=1*, *string_requirement=1*)
Test whether the number of hit antennas meets the given antenna and string trigger requirements.

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*)

Bases: *pyrex.detector.Detector*

Station geometry with one string at the station center and the rest of the strings evenly spaced radially around the station center. Supports any string type and passes extra keyword arguments on to the string class.

set_positions(*x*, *y*, *strings_per_station=4*, *station_diameter=50*, *string_type=<class 'pyrex.custom.irex.detector.IREXString'>*, ***string_kwargs*)
Generates string positions around the station.

triggered(*antenna_requirement=1*, *string_requirement=1*)
Test whether the number of hit antennas meets the given antenna and string trigger requirements.

class pyrex.custom.irex.detector.**StationGrid**(*stations=1*, *station_separation=500*, *station_type=<class 'pyrex.custom.irex.detector.IREXString'>*, ***station_kwargs*)

Bases: *pyrex.detector.Detector*

Rectangular grid of stations or strings, in a square layout if possible, separated by the given distance. Supports any station or string type and passes extra keyword arguments on to the station or string class.

set_positions(*stations=1*, *station_separation=500*, *station_type=<class 'pyrex.custom.irex.detector.IREXString'>*, ***station_kwargs*)
Generates rectangular grid of stations.

triggered(*station_requirement=1*, ***station_trigger_kwargs*)
Test whether the number of hit stations meets the given station trigger requirement.

pyrex.custom.irex.frontends module

Module containing IREX front-end circuit models

pyrex.custom.irex.frontends.**basic_envelope_model**(*signal*, *cap=2e-11*, *res=500*)
Model of a basic diode-capacitor-resistor envelope circuit. Takes a signal object as the input voltage and returns the output voltage signal object.

pyrex.custom.irex.frontends.**bridge_rectifier_envelope_model**(*signal*, *cap=2e-11*, *res=500*)
Model of a diode bridge rectifier envelope circuit. Takes a signal object as the input voltage and returns the output voltage signal object.

pyrex.custom.irex.reconstruction module

Module containing reconstruction methods for IREX

`pyrex.custom.irex.reconstruction.`**`quick_vertex_reconstruction`**(*detector*, *threshold=None*, *get_waveform=<function <lambda>>*)

`pyrex.custom.irex.reconstruction.`**`full_vertex_reconstruction`**(*detector*, *threshold=None*, *get_waveform=<function <lambda>>*)

`pyrex.custom.irex.reconstruction.`**`get_xcorr_times`**(*waveforms*)

`pyrex.custom.irex.reconstruction.`**`minimizer_vertex_reconstruction`**(*positions*, *times*, *guess=None*)

`pyrex.custom.irex.reconstruction.`**`least_squares`**(*vertex*, *positions*, *times*, *method='trace'*)

`pyrex.custom.irex.reconstruction.`**`bancroft_vertex`**(*positions*, *times*, *velocity=170940170.94017094*)

`pyrex.custom.irex.reconstruction.`**`bancroft_scan_vertex`**(*positions*, *times*, *velocity=170940170.94017094*)

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` function (and `receive` function 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 `triggered()` and `clear()` method to `Detector` class.
- 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 `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 `set_orientation` function to `Antenna` class 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 `with_times` method to `Signal` class for interpolation/extrapolation of signals to different times.
- Added `full_waveform` and `is_hit_during` methods to `Antenna` class for calculation of waveform over arbitrary time array and whether said waveform triggers the antenna, respectively.
- Added `front_end_processing` method to `IREXAntenna` 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 `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 `directional_gain` and `polarization_gain` methods to base `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` 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` methods become properties where reasonable:
  * `PathFinder.exists()` becomes `PathFinder.exists`.
  * `PathFinder.getEmittedRay()` becomes `PathFinder.emitted_ray`.
  * `PathFinder.getPathLength()` becomes `PathFinder.path_length`.
- `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

MIT License

PYTHON MODULE INDEX

p

INDEX

D

E

F

G

I

K

L

M

N

P

Q

R

S

T

U

V

W

Z