# PyREx Documentation

*Release 1.4.0*

**Ben Hokanson-Fasig**

**Mar 01, 2018**

# CONTENTS:

CHAPTER

ONE

# INTRODUCTION TO 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
```

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 Code Example

The most basic simulation can be produced as follows:

First, import the package:

```
import pyrex
```

Then, create a particle generator object that will produce random particles in a 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

# CODE EXAMPLES

The following code examples 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.

## 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:

```
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)
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 origin point 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, origin=[0,0,-300], 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
```

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

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

## 2.3 AntennaSystem and Detector Classes

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

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

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

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

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

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

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

incoming_signal_1 = pyrex.FunctionSignal(np.linspace(0,2*np.pi), np.sin,
                                         value_type=pyrex.Signal.ValueTypes.voltage)
incoming_signal_2 = pyrex.FunctionSignal(np.linspace(4*np.pi,6*np.pi), np.sin,
                                         value_type=pyrex.Signal.ValueTypes.voltage)
basic_antenna_system.receive(incoming_signal_1)
basic_antenna_system.receive(incoming_signal_2, origin=[0,0,-300],
                             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 an energy given by an energy generation function:

```
box_width = 1000 # m
box_depth = 500 # m
const_energy_generator = lambda: 1e8 # GeV
my_generator = pyrex.ShadowGenerator(dx=box_width, dy=box_width,
                                     dz=box_depth,
                                     energy_generator=const_energy_generator)
my_generator.create_particle()
```

## 2.6 Ray Tracing

As of PyREx version 1.4.0 full ray tracing is supported. However, this section has yet to be updated. Complain to Ben about it.

While PyREx does not currently support full ray tracing, it does provide a `PathFinder` class which implements some basic ray analysis by checking for total internal reflection along a straight-line path. `PathFinder` takes an ice model and two points as arguments and provides a number of properties and methods regarding the path between the points.

```
start = (0, 0, -100) # m
finish = (0, 0, -250) # m
my_path = pyrex.PathFinder(ice_model=pyrex.IceModel,
                           from_point=start, to_point=finish)
```

`PathFinder.exists` is a boolean value of whether or not the path between the points is traversable according to the indices of refraction. `PathFinder.emitted_ray` and `PathFinder.received_ray` are both unit vectors giving the direction from `from_point` to `to_point`. `PathFinder.path_length` is the length in meters of the straight line path between the two points.

```
my_path.exists
my_path.emitted_ray
my_path.path_length
```

`PathFinder.time_of_flight()` calculates the time it takes for light to traverse the path, with an optional parameter `n_steps` defining the precision used. `PathFinder.tof` is a convenience property set to the time of flight using the default value of `n_steps`.

```
my_path.time_of_flight(n_steps=100)
my_path.time_of_flight() == my_path.tof
```

`PathFinder.attenuation()` calculates the attenuation factor along the path for a signal of given frequency. Here again there is an optional parameter `n_steps` defining the precision used.

```
frequency = 1e9 # Hz
my_path.attenuation(f=frequency, n_steps=100)
```

Finally, `PathFinder.propagate()` propagates a `Signal` object from `from_point` to `to_point` by applying a `1/PathFinder.path_length` factor, applying the frequency attenuation of `PathFinder.attenuation()`, and shifting the signal times by `PathFinder.tof`:

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

PyREx also includes a `ReflectedPathFinder` class which essentially wraps two `PathFinder` objects containing rays which make up a path from the `from_point` to the `to_point`, undergoing total internal reflection at the specified `reflection_depth`. By default the `reflection_depth` is 0, assuming a reflection off of the surface of the ice.

`ReflectedPathFinder` is interacted with in the same way as `PathFinder`: `ReflectedPathFinder.exists` is a boolean of whether each of the constituent paths exist and total internal reflection is possible at the specified depth. `ReflectedPathFinder.emitted_ray` is the emitted ray of the first constituent path and `ReflectedPathFinder.received_ray` is the received ray of the second constituent path. `ReflectedPathFinder.tof` and `ReflectedPathFinder.time_of_flight()` are the sums of the times of flight for the constituent paths (with `n_step` passed to each `time_of_flight` method). Similarly `ReflectedPathFinder.attenuation()` is the product of the attenuations for the constituent paths with `n_step` passed to each. And finally `ReflectedPathFinder.propagate()` runs the `propagate` methods of both constituent paths in sequence.

## 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:

```
particle_generator = pyrex.ShadowGenerator(dx=1000, dy=1000, dz=500,
                                           energy_generator=lambda: 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,
                           ice_model=pyrex.IceModel,
                           antennas=detector)

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 PyREx Demo python notebook.

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

# PYREX API

## 4.1 Package contents

A Python package for simulation of Askaryan pulses and radio antennas in ice.

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).

**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*)

Applies the given frequency response function to the signal.

**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 neutrino with given energy (GeV) observed at angle theta (radians). 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*, *freq_range=None*, *noise_rms=None*, *temperature=None*, *resistance=None*, *noisy=True*)
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 the antenna to a state of having received no signals.

**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, where the real part is the amplitude response and the imaginary part is the phase response.

**receive**(*signal*, *origin=None*, *polarization=None*)
Process incoming signal according to the filter function and store it to the signals list. Subclasses may extend this fuction, but should 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*)
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.

pyrex.**IceModel**
alias of AntarcticIce

**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.

**k = 0.438**

**a = 0.0132**

**n0 = 1.32**

**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.

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).

pyrex.**slant_depth**(*angle*, *depth*, *step=5000*)
Returns the material thickness (g/cm^2) for a chord cutting through earth at Nadir angle and starting at depth (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_generator*)
Class to generate UHE neutrino vertices in (relatively) shallow detectors. Takes into accout Earth shadowing (sort of). energy_generator should be a function that returns a particle energy in GeV.

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

**class** pyrex.**PathFinder**(*ice_model*, *from_point*, *to_point*)
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.**EventKernel**(*generator*, *ice_model*, *antennas*)
Kernel for generation of events with a given particle generator, ice model, and list of antennas.

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

## 4.2 Submodules

### 4.2.1 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*)
Applies the given frequency response function to the signal.

**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 neutrino with given energy (GeV) observed at angle theta (radians). 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 neutrino with given energy (GeV) observed at angle theta (radians). 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).

### 4.2.2 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, freq_range=None, noise_rms=None, temperature=None, resistance=None, noisy=True*)
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 the antenna to a state of having received no signals.

**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, where the real part is the amplitude response and the imaginary part is the phase response.

**receive**(*signal*, *origin=None*, *polarization=None*)
Process incoming signal according to the filter function and store it to the signals list. Subclasses may extend this fuction, but should 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*)
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.

### 4.2.3 pyrex.ice_model module

Module containing ice model. AntarcticIce class contains static and class methods for easy swapping of models. 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.

**k = 0.438**

**a = 0.0132**

**n0 = 1.32**

**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 - USE ANTARCTICICE).

**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).

pyrex.ice_model.**IceModel**
alias of *AntarcticIce*

### 4.2.4 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).

pyrex.earth_model.**slant_depth**(*angle*, *depth*, *step=5000*)
Returns the material thickness (g/cm^2) for a chord cutting through earth at Nadir angle and starting at depth (m).

### 4.2.5 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_generator*)
Bases: object

Class to generate UHE neutrino vertices in (relatively) shallow detectors. Takes into accout Earth shadowing (sort of). energy_generator should be a function that returns a particle energy in GeV.

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

### 4.2.6 pyrex.ray_tracing module

Module containing class for ray tracing through the ice.

**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.

### 4.2.7 pyrex.kernel module

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

**class** pyrex.kernel.**EventKernel** (*generator*, *ice_model*, *antennas*)
Bases: object

Kernel for generation of events with a given particle generator, ice model, and list of antennas.

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

## 4.3 PyREx Custom Subpackage

More modules available as plug-ins, see *Custom Sub-Package*.

### 4.3.1 pyrex.custom.irex module

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

CHAPTER

FIVE

# VERSION HISTORY

## 5.1 Version 1.4.0

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

## 5.2 Version 1.3.1

- Changed neutrino interaction model to include all neutrino and anti-neutrino interactions rather than only charged-current neutrino (relevant for `ShadowGenerator` class).
- 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.

## 5.3 Version 1.3.0

- Added and implemented `ReflectedPathFinder` class for rays which undergo total internal reflection and subsequently reach an antenna.
- Improve performance of ice index calculated at many depths.
- 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).

## 5.4 Version 1.2.1

- Added `set_orientation` function to `Antenna` class for setting the `z_axis` and `x_axis` attributes appropriately.
- Fix bug where `Antenna._convert_to_antenna_coordinates` function was returning coordinates relative to (0,0,0) rather than the antenna's position.

## 5.5 Version 1.2.0

- 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.

## 5.6 Version 1.1.2

- Added `with_times` method to `Signal` class for interpolation/extrapolation of signals to different times.
- 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.
- Add `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).

## 5.7 Version 1.1.1

- 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.

## 5.8 Version 1.1.0

- Made units consistent across PyREx.
- 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 ability to define `Antenna` noise by RMS voltage rather than temperature and resistance if desired.
- 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).
- Allow `DipoleAntenna` to guess at `effective_height` if not specified.
- Increase speed of `IceModel.__atten_coeffs` method, resulting in increased speed of attenuation length calculations.

## 5.9 Version 1.0.3

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

## 5.10 Version 1.0.2

- 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()`.
- Added `Antenna.make_noise()` method so custom antennas can use their own noise functions.
- 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).

## 5.11 Version 1.0.1

- Fixed bugs in `AskaryanSignal` that caused the convolution to fail.
- Changed `Antenna` not require a temperature and frequency range if no noise is produced.
- Fixed bugs resulting from converting `IceModel.temperature()` from Celsius to Kelvin.

## 5.12 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()`.

## 5.13 Version 0.0.0

Original PyREx python notebook written by Kael Hanson:

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

# PYTHON MODULE INDEX

## p

# INDEX

## U



## V



## W