PyREx Documentation

Release 1.0.2

Ben Hokanson-Fasig

Sep 07, 2017

CONTENTS:

CHAPTER

ONE

QUICK START

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, as in the ARA and ARIANA collaborations.

1.1 Installation

To use the PyREx package, 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. In the future, PyREx may be installable via `pip`, but it is not currently available there.

1.2 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()
```

CHAPTER

TWO

CODE EXAMPLES

The following code examples assume these imports:

```
import numpy as np
import matplotlib.pyplot as plt
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 (though it is commonly assumed to be in volts or volts per meter). It is worth noting that the Singal 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)
```

`Signal` objects can be added as long as they have the same time array, and 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")
signal2 = pyrex.Signal(time_array, values2)
plt.plot(signal2.times, signal2.values, label="signal2")
signal3 = signal1 + signal2
plt.plot(signal3.times, signal3.values, label="signal3")
all_signals = [signal1, signal2, signal3]
signal4 = sum(all_signals)
plt.plot(signal4.times, signal4.values, label="signal4")
plt.legend()
```

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
```

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)
```

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)
```

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

```
time_array = np.linspace(0,10)
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)
```

`AskaryanSignal` produces an Askaryan pulse 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 = 1e5 # TeV
observation_angle = 45 * np.pi/180 # radians
askaryan = pyrex.AskaryanSignal(times=time_array, energy=neutrino_energy,
```

```
                                    theta=observation_angle)
plt.plot(askaryan.times, askaryan.values)
```

`ThermalNoise` produces Rayleigh noise 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)
plt.plot(noise.times, noise.values)
```

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).

```
position = (0, 0, -100) # m
temperature = 300 # K
resistance = 1 # ohm
frequency_range = (0, 1e3) # 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 defines three methods which are expected to be overwritten: `trigger`, `response`, and `receive`. `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`). `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`).

```
basic_antenna.trigger(pyrex.Signal([0],[0])) == True
freqs = [1, 2, 3, 4, 5]
basic_antenna.response(freqs) == [1, 1, 1, 1, 1]
```

The `receive` method is a bit different in that it contains some default functionality:

```
def receive(self, signal):
    copy = Signal(signal.times, signal.values)
    copy.filter_frequencies(self.response)
    self.signals.append(copy)
```

In this sense, the `receive` function is intended to be extended instead of overwritten. In derived classes, it is recommended that a newly defined `receive` function call `super().receive(signal)`. For example, if a polarization is to be applied, the following `receive` function could be implemented:

```
def receive(self, signal, signal_polarization):
    polarization_factor = np.vdot(self.polariztion, signal_polarization)
    polarized_signal = Signal(signal.times,
                              signal.values * polarization_factor)
    super().receive(polarized_signal)
```

To use the `receive` function, simply pass it the `Signal` object the antenna sees, and the `Antenna` class will handle the rest:

```
incoming_singal = pyrex.FunctionSignal(np.linspace(0,10), np.sin)
basic_antenna.receive(incoming_singal)
basic_antenna.is_hit == True
for wave in basic_antenna.waveforms:
    plt.figure()
    plt.plot(wave.times, wave.values)
    plt.show()
for pure_signal in basic_antenna.signals:
    plt.figure()
    plt.plot(pure_signal.times, pure_signal.values)
    plt.show()
```

The `Antenna` class also 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_singal = pyrex.FunctionSignal(np.linspace(0,10), np.sin)
my_antenna.receive(incoming_singal)
my_antenna.is_hit == False
len(my_antenna.waveforms) == 0
len(my_antenna.all_waveforms) == 1

incoming_singal = pyrex.Signal(incoming_singal.times,
                               5*incoming_singal.values)
my_antenna.receive(incoming_singal)
```

```
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()
```

PyREx defines `DipoleAntenna` which as a subclass of `Antenna`, which provides a basic threshold trigger, a basic bandpass filter, and a polarization effect on the reception of a signal. A `DipoleAntenna` object is created as follows:

```
antenna_identifier = "antenna 1"
position = (0, 0, -100)
center_frequency = 250 # MHz
bandwidth = 200 # MHz
resistance = 1000 # ohm
antenna_length = 1 # 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,
                             polarization=polarization_direction,
                             trigger_threshold=trigger_threshold)
```

2.3 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 = 100 # MHz
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)
```

2.4 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()
```

2.5 Ray Tracing

While PyREx does not currently support full ray tracing, it does provide a `PathFinder` class which implements some basic ray analysis by Snell's law. `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` is a unit vector 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()
```

2.6 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=500, bandwidth=500,
                            resistance=0, effective_height=1,
                            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))
```

2.7 More Examples

For more code examples, see the PyREx Demo python notebook.

CHAPTER

THREE

PYREX API

3.1 Package contents

class pyrex.**Signal**(*times*, *values*)
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.

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.

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*)
Bases: *pyrex.signals.Signal*

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

class pyrex.**FunctionSignal**(*times*, *function*)
Bases: *pyrex.signals.Signal*

Class for signals generated by a function

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 (TeV) 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=7.86e-05*, *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 (TeV), and electron radiation length (g/cm^2) in ice.

max_length(*density=0.92*, *crit_energy=7.86e-05*, *rad_length=36.08*)
Calculates the maximum length (m) of an EM shower with parameters for the density (g/cm^3), critical energy (TeV), and electron radiation length (g/cm^2) in ice.

class pyrex.**ThermalNoise**(*times*, *temperature*, *resistance*, *f_band*, *f_amplitude=1*, *n_freqs=0*)
Bases: *pyrex.signals.Signal*

Thermal Rayleigh noise at a given temperature (K) and resistance (ohms) in the frequency band f_band=[f_min,f_max] (Hz). 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 given times array). Returned signal values are voltages (V).

class pyrex.**Antenna**(*position*, *temperature=None*, *freq_range=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.

is_hit
Test for whether the antenna has received a signal.

isHit()
Deprecated. Replaced by is_hit property.

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.

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

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, polarization=[0, 0, 1]*)
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, effective_height, polarization=[0, 0, 1], trigger_threshold=0, noisy=True*)
Bases: *pyrex.antenna.Antenna*

Antenna with a given name, position (m), center frequency (MHz), bandwidth (MHz), 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.

receive(*signal, polarization=[0, 0, 1]*)
Apply polarization effect to signal, then proceed with usual antenna reception.

pyrex.**IceModel**
alias of AntarcticIce

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

Class containing characteristics of ice at the south pole.

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.

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 (MHz). 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**
Named tuple for containing particle attributes. Consists of a 3-D vertex (m), 3-D direction vector, and an energy (GeV).

direction

energy

vertex

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 and random direction.

class pyrex.**PathFinder**(*ice_model*, *from_point*, *to_point*)
Class for ray tracking.

exists
Boolean of whether path exists.

emitted_ray
Direction in which ray is emitted.

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.

3.2 Submodules

3.2.1 pyrex.signals module

Module containing classes for digital signal processing

class pyrex.signals.**Signal**(*times*, *values*)
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.

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.

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*)
Bases: *pyrex.signals.Signal*

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

class pyrex.signals.**FunctionSignal**(*times*, *function*)
Bases: *pyrex.signals.Signal*

Class for signals generated by a function

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 (TeV) 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=7.86e-05*, *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 (TeV), and electron radiation length (g/cm^2) in ice.

max_length(*density=0.92*, *crit_energy=7.86e-05*, *rad_length=36.08*)
Calculates the maximum length (m) of an EM shower with parameters for the density (g/cm^3), critical energy (TeV), 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 (TeV) 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=7.86e-05*, *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 (TeV), and electron radiation length (g/cm^2) in ice.

max_length(*density=0.92*, *crit_energy=7.86e-05*, *rad_length=36.08*)
Calculates the maximum length (m) of an EM shower with parameters for the density (g/cm^3), critical energy (TeV), and electron radiation length (g/cm^2) in ice.

pyrex.signals.**AskaryanSignal**
alias of *FastAskaryanSignal*

class pyrex.signals.**GaussianNoise**(*times*, *sigma*)
Bases: *pyrex.signals.FunctionSignal*

Gaussian noise signal with standard deviation sigma

class pyrex.signals.**ThermalNoise**(*times*, *temperature*, *resistance*, *f_band*, *f_amplitude=1*, *n_freqs=0*)

Bases: *pyrex.signals.Signal*

Thermal Rayleigh noise at a given temperature (K) and resistance (ohms) in the frequency band f_band=[f_min,f_max] (Hz). 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 given times array). Returned signal values are voltages (V).

3.2.2 pyrex.antenna module

Module containing antenna class capable of receiving signals

class pyrex.antenna.**Antenna**(*position*, *temperature=None*, *freq_range=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.

is_hit
Test for whether the antenna has received a signal.

isHit()
Deprecated. Replaced by is_hit property.

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.

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

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, polarization=[0, 0, 1]*)
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, effective_height, polarization=[0, 0, 1], trigger_threshold=0, noisy=True*)

Bases: *pyrex.antenna.Antenna*

Antenna with a given name, position (m), center frequency (MHz), bandwidth (MHz), 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.

receive(*signal, polarization=[0, 0, 1]*)
Apply polarization effect to signal, then proceed with usual antenna reception.

3.2.3 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.

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.

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 (MHz). 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*

3.2.4 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).

3.2.5 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 at a given energy E (GeV).

interaction_length(*E*)
Return the interaction length at a given energy E (GeV).

class pyrex.particle.**Particle**
Bases: tuple

Named tuple for containing particle attributes. Consists of a 3-D vertex (m), 3-D direction vector, and an energy (GeV).

direction

energy

vertex

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 and random direction.

3.2.6 pyrex.ray_tracing module

Module containing class for ray tracking through the ice. Ray tracing not yet implemented.

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

Class for ray tracking.

exists
Boolean of whether path exists.

emitted_ray
Direction in which ray is emitted.

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.

3.2.7 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.

CHAPTER

FOUR

VERSION HISTORY

4.1 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.
- 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).

4.2 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.

4.3 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()`.

4.4 Version 0.0.0

Original PyREx python notebook written by Kael Hanson:

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

PYTHON MODULE INDEX

p

INDEX