# PyCav Documentation

***Release 1.0.0b3+20.g707483f.dirty***

**PyCav team**

August 30, 2016

1 Installing PyCav 3
2 PyCav API Documentation 5
3 About PyCav 55

PyCav offers a rich set of tools so that you can make and visualize effective Physical simulations quickly. Its main components are:

- A mechanics module which can currently do the following:
  - Collision detection between spherical particles
  - Connect particles with a spring
  - Simulate forces from a field
- A quantum module for time independent 1D quantum systems
- A PDE module for 1D and 2D systems
- An optics module for performing refraction in the geometic optics limit
- A display module for displaying matplotlib animations within Jupyter notebooks
- Optional visualization using vpython

This documentation will walk through how to setup PyCav, as well as how to use it.

The table of contents is in the left sidebar, and developement can be followed at PyCav module repository.

The changelog is also at the PyCav module repository.

# Installing PyCav

PyCav requires Python 2.7 or  3.5.

There are a few ways to install PyCav. * The simplest and recommended way is to run the following command in Command Prompt (Windows) or Terminal(Mac/Linux), with pip installed:

```
  $ pip install pycav

This way, PyCav and all of its dependencies will be installed.
If you want to visualize the simulations, vpython also needs to be installed with
```

```
$ pip install vpython
```

- If you don't want to use the previous command, you can get the latest release as a tarball from from PyPI. With this method, however, you will not get the dependencies automatically installed, so make sure that you have: * numpy * matplotlib * scipy * vpython (If you want to visualize simulations) installed when you want to use PyCav.

CHAPTER 2

# PyCav API Documentation

PyCav currently contains these modules. Click on their names to read their API documentation.

## 2.1 Mechanics

### 2.1.1 Introduction to the mechanics module

The mechanics module provides the infrastructure to create simple systems consisting of springs and particles, and handles collisions between them as well as the visualization. A quick example of their use:

```
from pycav.mechanics import *
from vpython import *

container = Container(dimension=1)
system = System(collides=True, interacts=False, visualize=True, container=container)
avg_speed = 1
system.create_particles_in_container(number=100, speed=avg_speed, radius=0.03)
system.run_for(10)
```

This creates a system consisting of a 100 particles which can collide with each other within a cubic container of unit size, which is then run for 10 seconds in the system's time.

You can find the source code for the Mechanics module on the GitHub.

### 2.1.2 Classes

#### Particle

Particle is a class that represents a particle. It is not tied to any visualization method by design. By default, it implements a gravitational field, which can be changed by subclassing.

##### Functions

**__init__(pos=None, v=None, radius=1., inv_mass=0., color=None, alpha=1., fixed=False, applied_force=no_force, q = 0., make_trail = False)**

Initialises the Particle object.

**Parameters:**

*pos: numpy array*

Initial position of particle, default 0, 0, 0

*v: numpy array*

Initial velocity of particle, default 0, 0, 0

*radius: float*

Radius of particle

*inv_mass: float*

Inverse mass of particle

*color: array*

Color of particle, given in form [R G B], default 1, 0, 0

*alpha: float*

Alpha of particle, 1 is completely opaque, 0 is completely transparent, used in visualisation

*fixed: boolean*

Whether particle can move or not

*applied_force: function taking arguments of: particle(Particle) and time(float), that returns a numpy array*

Gives the applied force based on the particle's properties and time. By default, no force applied on particle.

*q: float*

Charge on particle

*make_trail: boolean*

Whether the particle will make a trail or not

**update(dt)**

Updates the position of the particle using the velocity Verlet method.

**Parameters:**

*dt: float*

Size of time step to take

**force_on(other, if_at=None)**

Gives force which another particle will feel from this particle (if at means that this can also give the force that the other particle would feel if it were at some other position). Default implementation gives gravitational force. Subclass particle and override this function to implement custom forces.

**Parameters:**

*other: Particle*

The particle which feels the force

*if_at: numpy array*

If this parameter is used, the function gives the force the 'other' particle would feel if it were at this position

**Returns:**

A *numpy array* with 3 elements giving the vector force on the other particle from this particle.

### Properties

**pos**

*numpy array*

3 element array giving position of particle in 3D space.

**v**

*numpy array*

3 element array giving the velocity of the particle.

**fixed**

*boolean*

If True, particle will be fixed and not move around however much force is applied to it. If False, particle will move due to interactions.

**q**

*float*

Charge on particle.

**color**

*array*

Gives the color of the particle as an array, given in the form [R G B]. Each element of the array should be between 0 and 1.

**radius**

*float*

Gives the radius of the particle.

**alpha**

*float*

Float between 0 and 1, giving the opacity of the particle.

**make_trail**

*boolean*

Decides whether the particle will make a trail or not when visualized.

**amplitude**

*float, read only*

Gives the amplitude of oscillations. Depends on the system class the particle is in to update.

**prev_pos**

*numpy array, read only*

3 element array giving the previous position of the particle.

## Spring

The Spring class represents a spring. It is not tied to any visualization method by design. It connects two Particles together and applies a force F = kx to them, as per Hooke's Law. It also shares many visualization properties with the Particle class.

### Functions

**__init__(particle_1, particle_2, k, l0=None, radius=0.5, color=None, alpha=1.)**

Initialises the Spring object by supplying the 2 particles it connects and the value of the stiffness, k.

**Parameters:**

*particle_1: Particle*

Particle on one end of spring

*particle_2: Particle*

Particle on other end of spring

*k: float*

The stiffness of the spring (F = kx)

*l0: float*

Original length of the spring

*radius: float*

Radius of spring.

*color: array*

Color of particle, given in form [R G B]

*alpha: float*

Alpha of particle, 1 is completely opaque, 0 is completely transparent, used in visualisation

**force_on(particle, if_at=np.array([None]))**

Given an arbitary particle, gives the force on that particle. No force if the spring isn't connected to that particle.

**Parameters:**

*particle: Particle*

Particle which feels the force

*if_at: NumPy Array*

If this parameter is used, this gives the force felt if the particle were at this position.

**Returns:**

A *numpy array* with 3 elements giving the vector force on the particle from the spring.

### Properties

**particle_1**

*Particle*

First particle that the spring is attached to.

**particle_2**

*Particle*

Second particle that the spring is attached to.

**k**

*float*

The stiffness of the spring (F = kx)

**l0**

*float*

The original length of the spring.

**color**

*array*

Gives the color of the spring as an array, given in the form [R G B]. Each element of the array should be between 0 and 1.

**radius**

*float*

Gives the radius of the spring.

**alpha**

*float*

Float between 0 and 1, giving the opacity of the spring.

**pos**

*numpy array, read only*

3 element array giving position of one end of the spring in 3D space.

axis ^

*numpy array, read only*

3 element array the axis, i.e. the vector showing the orientation and length of the spring.

### Container

Container is a class that represents a cubic Container that particles can be inside. It is not tied to any visualization method by design.

#### Functions

**__init__(dimension, pos=None, color=None, alpha=0.3)**

Initialises the Container object

**Parameters:**

*dimension: float*

The dimension of the Container, i.e. the length, width, height of the Container

*pos: numpy array*

Position of centre of cube, default 0, 0, 0

*color: array*

Color of particle, given in form [R G B], default 1,1,1

*alpha: float*

Alpha of particle, 1 is completely opaque, 0 is completely transparent, used in visualisation

**contains(particle)**

Function which checks if a given particle is entirely inside the Container

**Parameters:**

*particle: Particle*

Particle which is being checked to see if inside Container or not

**Returns:**

True if the Container contains the particle, and if not, returns the index of the axis along which the particle is outside the Container

#### Properties

**pos**

*numpy array*

3 element array giving position of tail end of Container in 3D space.

**dimension**

*float*

The dimension of the Container, i.e. the length, width, height of the Container

**axis**

*numpy array*

3 element array the axis, i.e. the vector showing the orientation and length of the Container.

**color**

*array*

Gives the color of the Container as an array, given in the form [R G B]. Each element of the array should be between 0 and 1.

**alpha**

*float*

Float between 0 and 1, giving the opacity of the Container.

**surface_area**

*float*

Float giving the surface area of the Container.

### PointerArrow

PointerArrow is a class that represents an arrow/pointer. It is not tied to any visualization method by design.

#### Functions

**__init__(pos, axis, shaftwidth=1, color=None, alpha=1)**

Initialises the PointerArrow object

**Parameters:**

*pos: numpy array*

Location of tail end of PointerArrow

*axis: numpy array*

A 3 dimensional vector giving the length and orientation of the pointer

*shaftwidth: float*

The width of the pointer's shaft

*color: array*

Color of PointerArrow, given in form [R G B], default 1, 1, 1

*alpha: float*

Alpha of PointerArrow, 1 is completely opaque, 0 is completely transparent, used in visualisation

#### Properties

**pos**

*numpy array*

3 element array giving position of the centre of the PointerArrow in 3D space.

**color**

*array*

Gives the color of the PointerArrow as an array, given in the form [R G B]. Each element of the array should be between 0 and 1.

**shaftwidth**

*float*

Gives the shaft width of the PointerArrow.

**alpha**

*float*

Float between 0 and 1, giving the opacity of the PointerArrow.

### System

System is a class used for the physical simulation of a collection of Particles, Springs, and Containers. It is also responsible for the visualization of such a collection. It does this by translating the properties of Particles, Springs, and Containers, to properties that vpython can use.

To simulate any new types of physical objects, subclass System and extend the simulate function. To make these new objects visualize in vpython, extend the create_vis and update_vis functions too.

To visualize the system using some different system than vpython, subclass System and override the create_vis and update_vis functions.

#### Functions

**__init__(collides, interacts, visualize, particles=None, springs=None, container=None, visualizer_type=”vpython”, canvas=None, stop_on_cycle=False, record_amplitudes=False, display_forces=False, record_pressure=False)**

Initialises a System class

**Parameters**

*collides: boolean*

Whether particles in this system collide with each other or not

*interacts: boolean*

Whether particles in this system interact with each other via fields

*visualize: boolean*

Whether the things in this system are visualized

*particles: array of Particles*

An array of all the particles in this system

*springs: array of springs*

An array of all the springs in this system

*container: Container*

Container within which the simulation takes place

*visualizer_type: string*

What type of visualizer is used. Used to make sure that some vpython-specific non-essential things don't run if different visualization method is used in a subclass. Set to anything you like if using another visualization method

*canvas: vpython cavas*

The canvas within which the system is visualized. Can be any type of view if System is subclassed

*stop_on_cycle: boolean*

Whether the simulation stops runnign when a full cycle is done. Only tested on relatively simple 1D systems, and only works when using the function run_for instead of simulate

*record_amplitudes: boolean*

Whether the system records the amplitudes of any oscillations. Only tested with simple 1D systems

*display_forces: boolean*

Whether the forces applied on the particle are displayed. By default, these vectors are displaced from the particles by 2*particle radius along the 2-axis. To change, need to subclass and edit simulate method.

*record_pressure: boolean*

Whether to record the pressure on the walls of the container or not

**create_vis(canvas=None)**

Creates a visualization. Override this method to change the visualization method.

**Parameters:**

*canvas: vpython canvas*

The canvas into which the visualization is drawn. For this implementation, must be a vpython canvas

**update_vis()**

Updates the visualization. Override this method to change the visualization method.

**run_for(time, dt=0.01, on_step=None)**

Run simulation for a certain amount of time(as measured in the simulated system's time). Recommended to use this instead of simulate(dt) for most situations, unless need some mechanism to stop simulation on some external condition.

**Parameters:**

*time: float*

Time for which the simulation will run for in the system's time

*dt: float*

Size of each step taken in time

*on_step: function taking one unnamed argument of System*

This system is passed to the function, and the defined function will be performed at the end of every step

**simulate(dt = 0.01)**

Simulates a time-step with a step size of dt. Collision detection, etc. happen here, so when adding new classes to simulate, extend this to add logic to simulate them.

**Parameters:**

*dt: float*

Size of time step taken

**create_particles_in_container(number=0, speed=0, radius=0, inv_mass=1.)**

Creates the given number of particles, with the given parameters, in random locations within the container. If the system has no container, this method will raise a RuntimeError.

**Parameters:**

*number: integer*

The number of particles to create

*speed: float*

The speed of these particles

*radius: float*

The radius of these particles

*inv_mass: float*

The inverse mass of these particles

#### Properties

*particles: array of Particles*

An array of all the particles in this system

*springs: array of springs*

An array of all the springs in this system

*container: Container*

Container within which the simulation takes place

*collides: boolean*

Whether particles in this system collide with each other or not

*interacts: boolean*

Whether particles in this system interact with each other via fields

*visualize: boolean*

Whether the things in this system are visualized

*visualizer_type: string*

What type of visualizer is used. Used to make sure that some vpython-specific non-essential things don't run if different visualization method is used in a subclass. Set to anything you like if using another visualization method

*canvas: vpython cavas*

The canvas within which the system is visualized. Can be any type of view if System is subclassed

*stop_on_cycle: boolean*

Whether the simulation stops runnign when a full cycle is done. Only tested on relatively simple 1D systems, and only works when using the function run_for instead of simulate

*record_amplitudes: boolean*

Whether the system records the amplitudes of any oscillations. Only tested with simple 1D systems

*display_forces: boolean*

Whether the forces applied on the particle are displayed. By default, these vectors are displaced from the particles by 2*particle radius along the 2-axis. To change, need to subclass and edit simulate method.

*record_pressure: boolean*

Whether to record the pressure on the walls of the container or not

*speeds: Array of floats, read only*

3D speed distribution of system as an unsorted array

*one_d_velocities: Array of floats, read only*

1D velocity distribution of system as an unsorted array

### 2.1.3 Functions

#### no_force(pos, time)

This is used as the default force applied to Particles. Returns no force, as name suggests.

**Parameters:**

*particle: Particle*

Particle which will feel the force

*time: float*

Time at which this force is felt

**Returns:**

A *numpy array* 0., 0., 0.

### element_mult(vec_1, vec_2)

Performs element wise multiplication (computes the Hadamard product) between two 3 dimensional vectors.

**Parameters:**

*vec_1: numpy array*

A 3 element numpy array which represents the first vector for which elements are multiplied

*vec_2: numpy array*

A 3 element numpy array which represents the second vector for which elements are multiplied

**Returns:**

The Hadamard product of the two vectors given as a 3-element numpy array

### vector_from(arr=np.array([0, 0, 0]))

Creates a vpython vector from a numpy array.

**Parameters:**

*arr: numpy array*

3 element numpy array which will be converted to into a vpython vector

**Returns:**

The numpy array converted into the equivalent vpython array

### normalized(arr)

Creates a normalised 1-D array for the 1-D array given.

**Parameters:**

*arr: numpy array*

Array to be normalised

**Returns:**

A 1-D numpy array that is of unit length

## 2.2 Quantum

### 2.2.1 Introduction to the Quantum module

This module contains functions for use in 1st order perturbation theory calculations and for solving 1d boundary value problems using the Shooting method. It also contains a class designed to represent systems of interacting spins. Documentation explaining the use of each function and the algorithms used within will be presented in individual sections.

You can find the source code for this module on the GitHub.

### 2.2.2 In-depth Documentation

#### Numerov Method

In units where \(\hbar = 1 \), the 1D time-independent Schrödinger equation can be expressed in the form:

$$ \frac{d^2 \psi}{dx^2} = -2m\left(E - V(x) \right) \psi = -g(x) \psi $$

This differential equation can be solved numerically via Numerov's method (see pages 10 - 11). For a 1D spatial grid, the wavefunction at the \(n+1\)th point along the x-axis can be approximated by:

$$ \psi_{n+1} = \frac{(12-10f_n) \ \psi_n-f_{n-1}\psi_{n-1}}{f_{n+1}} $$

where: $$ f_n \equiv \left( 1 + \frac{\delta x^2}{12}g_n \right), g_n = 2m(E-V(x_n)) $$

Hence to start Numerov's method we require \(\psi_0\) and \(\psi_1\), in other words \(\psi(x = x_{min})\) and \(\psi(x = x_{min} + \delta x)\), where \(\delta x\) is the step size.

When investigating bound states we require \(\psi( x \to \pm \infty) = 0\). However, we cannot consider an infinite domain. Instead we must choose a large enough domain that setting \(\psi( x = x_{min}) = 0\) is a good approximation (and similarly for \(x_{max}\)).

With Numerov's Method in place, the shooting method can be used to find the energy eigenstates. It goes as follows:

1. Setting \(\psi_0 = 0\) approximately satisfies the boundary condition that the wavefunctions must vanish at the boundary.
2. Since the Schrödinger Equation is linear and homogeneous we are free to set \(\psi_1\) to any non-zero constant as multiplying by a constant does not affect the solution. In this case we set \(\psi_1 = \delta x\).
3. Using the Numerov algorithm, \(\psi(x)\) can be found. Exponential growth near \(x_{max}\) is observed if the input energy is not near a energy eigenvalue

#### Argument list

numerov(x,dx,V,E,initial_values,params)

This function performs a Numerov integration at the given energy within the given domain and returns the un-normalised wavefunction evaluates it over the whole domain.

**Parameters:**

*x: numpy array*

An N element numpy array of equally spaced points (creating using numpy linspace is advised) at which the wavefunction will be evaluated

*dx: float*

The spacing between points in the x array

*V: function*

Function which takes x as an argument and returns the value of potential at that point, V(x)

*E: float*

The energy of the time independent wavefunction. This will give exponential growth if E does not correspond to a bound or free state energy eigenvalue.

*inital_value: list*

A list with 2 elements. The first of these is the boundary condition for the wavefunction at the lowest value spatial coordinate. The second is used to initialise the next point in from this, which must be non-zero. Apart from this requirement, the second element only effects normalisation.

*params: list*

List which can be used within your code to hold various physical parameters. The first element must be equal to the particle mass, but apart from this the size of the list and the other parameters are not called

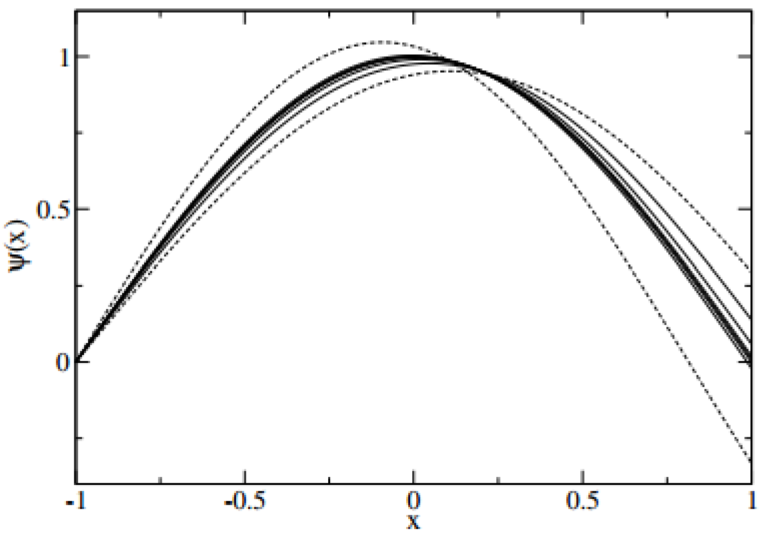
**Returns:**

A *numpy array* containing the approximated wavefunction evaluated at x

## Bisection Search

We wish to find a function \(f(E) = 0\). First we must find values of \(E\) which bracket the solution, that is: $$f(E_1) < 0, f(E_2) > 0$$ By evaluating \(f\) at the midpoint, \(E_3 = \frac{1}{2}(E_1+E_2)\), we can rebracket our solution. Hence the solution will be converge on iteration. Solutions can converge from both above and below \(0\), so your search algorithm should account for this.

The search should stop when \(\left| f(E)\right| < \epsilon\) where \(\epsilon\) is a suitably small number.



Dotted lines show the bracket solutions and the solid lines show the progression of the search to obtain the ground state of the potential.

### Argument list

bisection_search(x,dx,V,params,bracket_E,tolerance = 0.5,max_evals = 1000)

Uses the Numerov method to perform a bisection search in energy for a wavefunction which goes to zero at the boundaries

**Parameters:**

*x: numpy array*

An N element numpy array of equally spaced points (creating using numpy linspace is advised) at which the wavefunction will be evaluated

*dx: float*

The spacing between points in the x array

*V: function*

Function which takes x as an argument and returns the value of potential at that point, V(x)

*params: list*

List which can be used within your code to hold various physical parameters. The first element is required to be equal to the particle mass, but apart from this the size of the list and the other parameters are not called

*bracket_E: list*

A 2 element list which expresses a range in which the energy eigenvalue(s) lie. For the 2 energy values in this list, one of the values will have the Numerov approximated wavefunction above 0 and the other below 0. The ordering of these is handled by the function.

*tolerance: float*

The tolerance of the bisection search i.e. if the absolute value of the wavefunction at the right-hand boundary is less than the tolerance then the search is complete.

*max_evals: int*

The number of search evaluations taken before the search is given up. It is more likely your bracket_E list is not wide enough or your tolerance is too low than max_evals is too low

**Returns:**

The un-normalised wavefunction, as a numpy array, which has satisfied the bisection search (or max_evals has been reached) and the energy eigenvalue estimate as a float

### Perturbation Analysis

Using the following notation for our perturbation analysis:

Total Hamiltonian: $$ \hat{H} = \hat{H}^{(0)}+\hat{H}'$$

First order energy shift: $$ \Delta E_n^{(1)} = \langle n^{(0)} | \hat{H}' | n^{(0)} \rangle$$

First order perturbed wavefunctions: $$ | n^{(1)} \rangle \approx | n^{(0)} \rangle + \sum_{m \neq n} | m^{(0)} \rangle \frac{\langle m^{(0)} | \hat{H}' | n^{(0)} \rangle}{E_n^{(0)} - E_m^{(0)}} $$

Evaluating inner products are done using an integration over space i.e. $$ \langle n^{(0)} | \hat{H}' | n^{(0)} \rangle = \int_{x_{min}}^{x_{max}} \psi_{n}^{(0)}(x) \hat{H}'(x) \psi_{n}^{(0)}(x) dx $$

These are calculated using SciPy's quad integration function.

For first_order_wf, \(I_{mn} = \langle m^{(0)} | \hat{H}' | n^{(0)} \rangle / (E_n^{(0)}-E_m^{(0)})\) is calculated for m values around n until \(I_{mn} < \epsilon\), where \(\epsilon\) is the given tolerance. Two seperate iterations are run for even and odd values of m around n as the form of the perturbation may cause inner products to vanish for certain configurations of even/odd wavefunctions.

If return_list is set to True, the list of m values used in the sum is returned along with the corresponding \(I_{mn}\) values.

The perturbed wavefunction is calculated within the function using the following sum: $$ | n^{(1)} \rangle \approx | n^{(0)} \rangle + \sum_{k \in k_{list}} I_{kn} | k^{(0)} \rangle $$ This is returned as a function of position i.e.

```
x = np.linspace(-10.,10.,100)
perturb_wf = first_order_wf(n,H,unperturb_wf,unperturb_erg,params)
perturb_wf_x = perturb_wf(x)
```

Here perturb_wf_x is a numpy array containing the perturbed wavefunction evaluated at all the points in x

## Argument list

first_order_energy_sft(n,H,unperturb_wf,params,limits = [-np.inf,np.inf])

Works out the 1st order energy shift from time independent perturbation theory for a given unperturbed system and the applied perturbation

**Parameters:**

*n: int or list*

Principal quantum number (or list of these) which labels the unperturbed wavefunctions

*H: function*

The applied perturbation as a function of position

*unperturb_wf: function*

A function which is passed params and n, returning a function of position e.g.

```
def unperturb_wf(params,n):
  a = params[0]
  m = n+1
  def psi_n(x):
    return np.sqrt(2./a)*np.sin((m*np.pi/a)*(x+a/2.))
  return psi_n
```

Here psi_n(x) is the returned function of position

*params: list*

List which can be used within your code to hold various physical parameters used by unperturb_wf and other functions (see later)

*limits: list*

List containing the integration limits for the inner product of the wavefunctions and the perturbation. Default is the whole space. For hard wall potentials adjust these limits to respect the boundaries

**Returns:**

A list or float depending on the input argument n containing the first energy shifts to the unperturbed wavefunctions for the given quantum numbers in n

#### first_order_wf(n,H,unperturb_wf,unperturb_erg,params,tolerance = 0.01, limits = [-np.inf,np.inf], return_list = False)

Calculates the 1st order perturbed wavefunction for a given unperturbed system and the applied perturbation. The system is defined by its known unperturbed wavefunctions and energies. This function takes similar parameters as first_order_energy_sft (see above) so only new parameters will be defined.

**Unique Parameters:**

*unperturb_erg: function*

A function which is passed params and n and returns the energy of the n-th unperturbed eigenstate e.g. for a harmonic oscillator

```
def unperturb_erg(params,n):
  return (n+0.5)*params[1]
```

where params[1] contains the angular frequency (for hbar = 1)

*tolerance: float*

The value below which terms in the 1st order wavefunction sum are ignored

*return_list: float*

Set to True if you require the perturbation sum prefactors and values of the principal quantum numbers of the unperturbed wavefunctions

**Returns:**

A function of position which corresponds to the 1st order perturbed wavefunction and, if return_list = True, copies of the principal quantum number lists and the sum prefactors list which were used to calculate the resultant perturbed wavefunction

## SpinSystem

The SpinSystem class represents a system of spins interacting with one another. Interactions with an external inhomogeneous magnetic field can be included as an optional argument (note that the gyromagnetic ratios are unity).

### Functions

#### __init__(spins, couplings, B_field = None, scaling = [1.0,1.0,1.0])

Initialises the spin system object.

**Parameters**

*spins: list or numpy array*

Spin quantum numbers of the particles (integer or half-integer) e.g. [0.5, 1.0]

*couplings: list or numpy array*

NxN list for a system of N spins. The element J[i,j] gives the coupling strength between spins i and j. Positive coupling favours spin alignment. e.g. [[0,1],[1,0]]

*B_field: list, optional*

List of 3-dimensional vectors, [B_x, B_y, B_z], representing the magnetic flux density at each spin. e.g. [[3.0, 0, 0], [1.5, 0, 0]]

*scaling: list, optional*

List of floats which scale the spin-spin coupling in different directions. scaling[0] scales the Sx-Sx coupling, [1] the Sy-Sy coupling and [2] the Sz-Sz coupling.

**create_Hamiltonian()**

Create a Hamiltonian with spin-pairing given by the coupling array. A linear coupling between spins and the magnetic field is included if a B_field was used to initialise the SpinSystem.

The Hamiltonian acts on Kronecker products of the spins' states.

This function is called on initialisation and gives the instance of SpinSystem the attribute *H*, the Hamiltonian, as a numpy array.

**get_energies()**

Calculates the energy levels and eigenstates of the Hamiltonian using numpy.linalg.eigh. They are then stored as attributes *energies* and *states* in order of increasing energy, both are numpy arrays.

**count_multiplicities(tolerance = 0.0001)**

Counts the multiplicities of the energy eigenvalues, generating a list of multiplicities and list of non-repeated energies. These lists are subsequently stored as attributes *multiplicities* and *reduced_energies*.

**Parameters:**

*tolerance: float, optional*

Energy levels within *tolerance* of one another are considered degenerate.

#### Attributes

*The following attributes are created on initialisation*

*N: int*

Number of spins in the system.

*J: numpy array*

NxN array of coupling strengths. J[i,j] is the coupling strength between spins i and j.

*Sx: list of numpy arrays*

A list of x-angular momentum operators corresponding to the Hilbert spaces of each spin.

*Sy and Sz: list of numpy arrays*

See *Sx*.

*I: list of numpy arrays*

A list of the identity operators corresponding to the Hilbert spaces of each spin.

*B: numpy array*

The magnetic flux densities at each spin in the system. B[i,j] is the jth component of the magnetic field at the ith spin.

*H: numpy array*

The Hamiltonian of the system.

*The remaining attributes are not created on initialisation* *energies: numpy array*

Energy eigenvalues of the system, repeated according to degeneracy and stored in order of increasing energy.

*Created by get_energies() and count_multiplicities()*

*states: numpy array*

Energy eigenstates, stored in order of increasing energy.

*Created by get_energies() and count_multiplicities()*

*multiplicities: list*

List of the multiplicities of the energy eigenstates in order of increasing energy.

*reduced_energies: list*

List of the non-repeated energy levels. e.g. an 8-fold degenerate state will feature only once in this list.

The list below summarises the functions, their input arguments and their outputs for quick reference for the informed user:

### 2.2.3 Functions

#### numerov(x,dx,V,E,initial_values,params)

This function performs a Numerov integration at the given energy within the given domain and returns the un-normalised wavefunction evaluated at the given spatial coords

**Parameters:**

*x: numpy array*

An N element numpy array of equally spaced points in space (creating using numpy linspace is advised) at which the wavefunction will be evaluated

*dx: float*

Must give the spacing between points in the x array

*V: function*

Pass a function which takes x as an argument and returns the value of potential at that point, V(x)

*E: float*

The energy of the time independent wavefunction. This will give exponential growth if E does not correspond to a bound or free state energy eigenvalue.

*inital_value: list*

A list with 2 elements. The first of these is the boundary condition for the wavefunction at the lowest value spatial coordinate. The second of these is used to initialise the next point in from this, it must be non-zero but apart from this requirement it only effects normalisation (see in detail documentation)

*params: list*

List which can be used within your code to hold various physical parameter. The first element is required to be equal to the particle mass but apart from this size of the list and the other parameters are not called

**Returns:**

A *numpy array* containing the approximated wavefunction evaluated at x

### bisection_search(x,dx,V,params,bracket_E,tolerance = 0.5,max_evals = 1000)

Uses the Numerov method to perform a bisection search in energy for a wavefunction which goes to zero at the boundaries. Shares similar arguments with the numerov function above see *x, dx, V, params*

**Unique Parameters:**

*bracket_E: list*

A 2 element list which expresses a range in which the energy eigenvalue(s) lie. For the 2 energy values in this list, one of the values will have the Numerov approximated wavefunction above 0 and the other below 0. The ordering of these is handled by the function.

*tolerance: float*

The tolerance of the bisection search i.e. if the absolute value of the wavefunction at right hand boundary is less than the tolerance then the search is complete.

*max_evals: int*

The number of search evaluations taken before the search is given up. It is more likely your bracket_E list is not wide enough or your tolerance is too low than max_evals is too low

**Returns:**

The un-normalised wavefunction, as a numpy array, which has satisfied the bisection search (or max_evals has been reached) and the energy eigenvalue estimate as a float

### first_order_energy_sft(n,H,unperturb_wf,params,limits = [-np.inf,np.inf])

Works out the 1st order energy shift from time independent perturbation theory for a given unperturbed system and the applied perturbation

**Parameters:**

*n: int or list*

Principal quantum number (or list of these) which labels the unperturbed wavefunctions

*H: function*

The applied perturbation as a function of position

*unperturb_wf: function*

A function which is passed params and n and returns a function of position e.g.

```
def unperturb_wf(params,n):
  a = params[0]
  m = n+1
  def psi_n(x):
    return np.sqrt(2./a)*np.sin((m*np.pi/a)*(x+a/2.))
  return psi_n
```

Here psi_n(x) is the returned function of position

*params: list*

List which can be used within your code to hold various physical parameter used by unperturb_wf and other functions (see later)

*limits: list*

List containing the integration limits for the inner product of the wavefunctions and the perturbation. Default is the whole space. For hard wall potentials adjust these limits to respect the boundaries

**Returns:**

A list or float depending on the input argument n containing the first energy shifts to the unperturbed wavefunctions for the given quantum numbers in n

### first_order_wf(n,H,unperturb_wf,unperturb_erg,params,tolerance = 0.01, limits = [-np.inf,np.inf], return_list = False)

Calculates the 1st order perturbed wavefunction for a given unperturbed system and the applied perturbation. The system is defined by its known unperturbed wavefunctions and energies. This function takes similar parameters as first_order_energy_sft (see above) so only new parameters will be defined. Further documentation can be found on the functions documentation page

**Unique Parameters:**

*unperturb_erg: function*

A function which is passed params and n and returns the energy of the n-th unperturbed eigenstate e.g. for a harmonic oscillator

```
def unperturb_erg(params,n):
  return (n+0.5)*params[1]
```

where params[1] contains the angular frequency (for hbar = 1)

*tolerance: float*

The value below which terms in the 1st order wavefunction sum are ignored

*return_list: float*

Set to True if you require the perturbation sum prefactors and values of the principal quantum numbers of the unperturbed wavefunctions

**Returns:**

A function of position which corresponds to the 1st order perturbed wavefunction and if return_list = True, copies of the principal quantum number lists and the sum prefactors list which were used to calculate the resultant perturbed wavefunction

### create_Su(s)

Returns the ‘up’ angular momentum ladder operator in the z basis.

**Parameters:**

*s: int or float*

The angular momentum quantum number, 1/2 for isolated electrons, 1 for a hydrogenic p-orbital etc. If s is a float, it must be a half-integer.

**Returns:**

*numpy array with complex elements*

### create_Sd(s)

Returns the ‘down’ angular momentum ladder operator in the z basis. Usage is identical to create_Su.

#### create_Sx(s)

Returns the ‘x’ angular momentum operator in the z basis. Usage is identical to create_Su.

#### create_Sy(s)

Returns the ‘y’ angular momentum operator in the z basis. Usage is identical to create_Su.

#### create_Sz(s)

Returns the ‘z’ angular momentum operator in the z basis. Usage is identical to create_Su.

## 2.3 Partial Differential Equations

### 2.3.1 Introduction to the Partial Differential Equation module

This module contains functions for use in solving PDEs, namely the wave equation, the heat equation and the time dependent Schrödinger equation. These are solved using the Lax-Wendroff, Crank-Nicolson and Split Step Fourier methods respectively. Documentation explaining the use of each function and the algorithms used within will be presented in individual sections. A short introduction to finite difference methods will also be presented.

You can find the source code for this module on the GitHub.

### 2.3.2 Introductory Documentation

#### Finite Difference Methods

In these methods, partial derivatives in partial differential equations are approximated by linear combinations of function values at the grid points. For first order derivatives, the approximation needed can easily be seen from their definition:

$$ \frac{d f}{d x} = \lim_{\Delta x \to 0} \frac{f(x+ \Delta x)-f(x)}{\Delta x} $$

Hence if we take $\Delta x$ to be sufficiently small then the derivative will be well approximated by the following equivalent (in the limit of $\Delta x \to 0 $) expressions:

$$ \frac{d f}{d x} \approx \frac{f(x+ \Delta x)-f(x)}{\Delta x} $$ $$ \frac{d f}{d x} \approx \frac{f(x)-f(x- \Delta x)}{\Delta x} $$ $$ \frac{d f}{d x} \approx \frac{f(x+ \Delta x)-f(x- \Delta x)}{2 \Delta x} $$

If we now create a spatial gird, $x_i = x_0 + i \Delta x $, then labelling the function, $f$, on these grid points as $f_i$, then the finite differences as defined as:

Forward difference: $$ \left( \frac{d f}{d x} \right)_i \approx \frac{f_{i+1}-f_{i}}{\Delta x} $$

Backward difference: $$ \left( \frac{d f}{d x} \right)_i \approx \frac{f_{i}-f_{i-1}}{\Delta x} $$

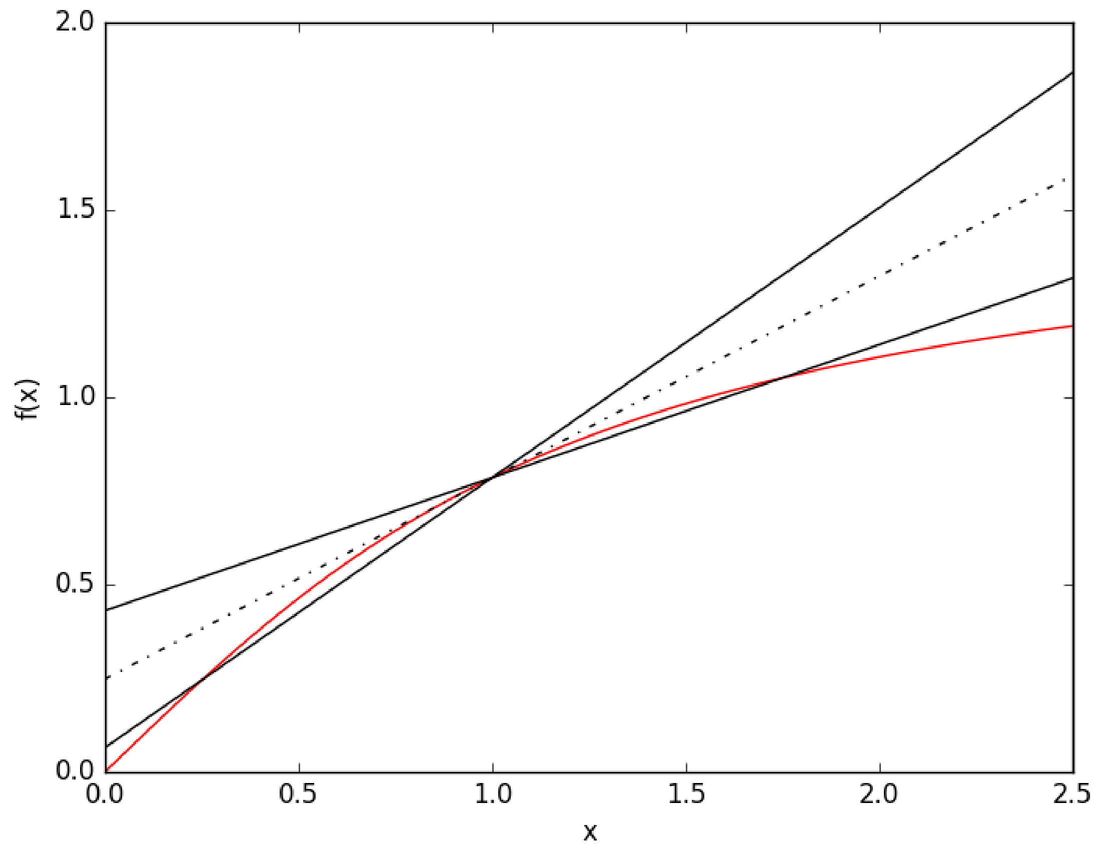
Central difference: $$ \left( \frac{d f}{d x} \right)_i \approx \frac{f_{i+1}-f_{i-1}}{2 \Delta x} $$

The errors in these approximations can be found by considering the Taylor expansions of $f_{i+1}$ and $f_{i-1}$:

$$ T_1: f_{i+1} = f_{i} + \Delta x \left( \frac{d f}{d x} \right)_i + \frac{\Delta x^2}{2} \left( \frac{d^2 f}{d x^2} \right)_i + \mathcal{O} (\Delta x^3) $$

$$ T_2: f_{i-1} = f_{i} - \Delta x \left( \frac{d f}{d x} \right)_i + \frac{\Delta x^2}{2} \left( \frac{d^2 f}{d x^2} \right)_i - \mathcal{O} (\Delta x^3) $$

Hence $T_1$ implies the forward difference has error of order $\Delta x$ and similarly for $T_2$ and the backward difference. However considering $T_1 - T_2$ we see the central difference has error of order $\Delta x^2$. Higer-order approximations can be found by considering linear combinations of Taylor expansions including $f_{i+2}$ and $f_{i-2}$.



The figure above shows estimates of the gradient of arctan(x) at x = 1 using the forward, backward and central difference methods. The hashed line shows the central difference result while the other lines are the first order forward and backward methods. A $\Delta x = 0.75$ was used, large to exaggerate the errors in these methods.

For second order derivatives, the approximation can be found via $T_1 + T_2$:

$$ T_1 + T_2: f_{i+1} + f_{i-1} = 2 f_{i} + \Delta x^2 \left( \frac{d^2 f}{d x^2} \right)_i + \mathcal{O} (\Delta x^3) $$

Therefore: $$ \left( \frac{d^2 f}{d x^2} \right)_i \approx \frac{f_{i+1} - 2 f_{i} + f_{i-1}}{ \Delta x^2} + \mathcal{O} (\Delta x) $$

These methods can be combined to evaulate mixed derivatives.

For variable coefficients, e.g. $f(x) = d(x) \frac{d u}{d x}$, we can find the first order derivative using half-step central differences $\left( \frac{d f}{d x} \right)_i \approx \frac{f_{i+1/2}-f_{i-1/2}}{\Delta x}$:

$$ \left( \frac{d f}{d x} \right)_i \approx \frac{f_{i+1/2}-f_{i-1/2}}{\Delta x} = \frac{d_{i+1/2} \left( \frac{d u}{d x} \right)_{i+1/2} - d_{i-1/2} \left( \frac{d u}{d x} \right)_{i-1/2} }{\Delta x} $$ $$ = \frac{d_{i+1/2} u_{i+1} - (d_{i+1/2}+d_{i-1/2}) u_i + d_{i-1/2} u_{i-1}}{\Delta x^2} $$

### 2.3.3 In-depth Documentation

#### Wave Equation via Lax/Lax-Wendroff schemes

The wave equations in 1D and 2D can be expressed as (for constant wave speed):

$$ \frac{ \partial^2 \psi}{\partial t^2} = c^2 \frac{ \partial^2 \psi}{\partial x^2} $$

$$ \frac{ \partial^2 \psi}{\partial t^2} = c^2 \left( \frac{ \partial^2 \psi}{\partial x^2} + \frac{ \partial^2 \psi}{\partial y^2} \right) $$

A large class of inital value PDEs can be case into a flux-conservative form. In 1D:

$$ \frac{\partial u}{\partial t} = -\frac{\partial F}{\partial x} $$

If we re-express the 1D wave equation in a flux-conservative form (which allows for the use of established numerical methods) then we obtain:

$$ u = \begin{bmatrix}r \\ s\end{bmatrix}, r \equiv c \frac{\partial \psi}{\partial x}, s \equiv \frac{\partial \psi}{\partial t} $$

$$ F = \begin{bmatrix}0 & -c \\ -c & 0\end{bmatrix}\cdot \begin{bmatrix}r \\ s\end{bmatrix} $$

We now have a first order differential equation to solve of a given inital value problem.

Unfortunately a simple finite difference method is unconditionally unstable. What a shame.

For the 1D wave equation we shall use the two-step Lax Wendroff scheme. This includes evaulating u at half steps, using this to find the half step fluxes and using a properly centered expression to perform the full time step. Using the notation introduced in the finite difference method documentation page:

$$ u^{n+1/2}_{j+1/2} = \frac{1}{2} (u_{j+1}^n+u_j^n) - \frac{\Delta t}{2 \Delta x}(F^n_{j+1} - F^n_j) $$

$$ u^{n+1}_{j} = u_j^n - \frac{\Delta t}{\Delta x}(F^{n+1/2}_{j+1/2} - F^{n+1/2}_{j-1/2}) $$

The stability of the scheme is parameterised by the Courant number, \(\alpha = c \Delta t / \Delta x \), and the criteria for stability is that \(\alpha \leq 1 \). This ensures \(c \Delta t \leq \Delta x \), hence information between spatial points can not have been communicated before the next time step is calculated.

Boundary conditions can be set to be periodic, fixed or reflective. These are defined in our flux-conservative formulation as:

Fixed (noting \(s = \frac{\partial \psi}{\partial t} \)):

$$ s(x = x_{min}) = s(x = x_{max}) = 0 $$

Reflective (noting \(r = c \frac{\partial \psi}{\partial x} \)):

$$ r(x = x_{min}) = r(x = x_{max}) = 0 $$

Periodic:

$$ \psi_N = \psi_1, \psi_0 = \psi_{N-1} $$

Now if we take the wave speed to be a function of position, \( c(x) \), then when the flux, \(F\), is evaulated at grid points then the wave speed at that point must be used e.g

$$ F^n_{j+1} = - \begin{bmatrix}c(x_{j+1}) s^n_{j+1} \\ c(x_{j+1}) r^n_{j+1} \end{bmatrix} $$

As an example of its use, here is a gaussian disturbance reflecting from a wall with fixed boundary conditions:

### 2D Lax Scheme

In 2D, the flux conservative form includes an additional flux:

$$ \frac{\partial u}{\partial t} = -\frac{\partial F_x}{\partial x}-\frac{\partial F_y}{\partial y} $$

We define our new $u$ vector as the following:

$$ u = \begin{bmatrix}r \\ l \\ s\end{bmatrix}, r \equiv c \frac{\partial \psi}{\partial x}, l \equiv c \frac{\partial \psi}{\partial y}, s \equiv \frac{\partial \psi}{\partial t} $$

$$ \frac{\partial r}{\partial t} = \frac{\partial}{\partial x} (c s)$$

$$ \frac{\partial l}{\partial t} = \frac{\partial}{\partial y} (c s)$$

$$ \frac{\partial s}{\partial t} = \frac{\partial}{\partial x}(c r) + \frac{\partial}{\partial y} (c l)$$

Hence the fluxes are given by:

$$ F_x = \begin{bmatrix}0 & 0 & -c \\ 0 & 0 & 0 \\ -c & 0 & 0\end{bmatrix}\cdot \begin{bmatrix}r \\ l \\ s\end{bmatrix} $$

$$ F_y = \begin{bmatrix}0 & 0 & 0 \\ 0 & 0 & -c \\ 0 & -c & 0\end{bmatrix}\cdot \begin{bmatrix}r \\ l \\ s\end{bmatrix} $$

Plugging in the definitions of $r$, $l$ and $s$ to the expression for $\dot{s}$, the 2D wave equation is recovered.

Using a 2D Lax scheme, the components of $u$ can be calculated by:

$$ u^{n+1}_{j,l} = \frac{1}{4} (u^n_{j+1,l} + u^n_{j-1,l} + u^n_{j,l+1} + u^n_{j,l-1}) - \frac{\Delta t}{2 \Delta}(F^n_{x,j+1,l}-F^n_{x,j-1,l}+F^n_{y,j,l+1}-F^n_{y,j,l-1}) $$

Where $\Delta = \Delta x = \Delta y $. The fluxs labelled with $x,y$ refer to terms within the $x,y$ partial derivatives in the flux conservative expressions above. e.g. $F_x$ for $r$ is $c s$ while $F_y$ is zero.

Boundaries conditions are dealt with in a similar way to the 1D case. But now $l$ vanishes at $y_{min}$ and $y_{max}$ for reflective boundary conditions.

For 2D cases, the Courant number must now be less than $ 2^{-1/2} $. It should be noted that the definition of $c$ in the Courant condition is replaced with the maximal wave speed when the wave speed is allowed to vary with position.

As an example of its use, here is a gaussian disturbance within a container with periodic boundary conditions:

### Form of the wave equation for spatially varying wave speed

A important distinction should be made about the form of the wave equation. There are two possible forms of the wave equation for a variable wave speed, in 1D these are:

$$ \frac{ \partial^2 \psi}{\partial t^2} = c(x)^2 \frac{ \partial^2 \psi}{\partial x^2} $$

$$ \frac{ \partial^2 \psi}{\partial t^2} = \frac{ \partial }{\partial x} \left( c(x)^2 \frac{\partial \psi}{\partial x} \right) $$

In the form we have cast the wave equation we are solving for the second of these equations. This describes systems such as surface waves on a fluid. The first equation follows from the electro-magnetic Maxwell equations in 1D.

*It should be noted when giving positional depedent wavespeeds with discontinuities, this will not give the familiar reflection and transmission results as the additional boundary conditions at the discontinuties are not included*

### Argument list

LW_wave_equation(psi_0, x_list, dx, N_t, c, a = 1., bound_cond = 'periodic',init_grad = None, init_vel = None)

This function performs the two-step Lax-Wendroff scheme for 1D problems and a Lax method for 2D problems to solve a flux-conservative form of the wave equation for variable wave speed, c.

**Parameters:**

*psi_0: numpy array*

In 1D, an N element numpy array containing the intial values of \(\psi\) at the spatial grid points. In 2D, a NxM array is needed where N is the number of x grid points, M the number of y grid points. This array needs to be in "matrix indexing" rather than "Cartesian indexing" i.e. the first index (the rows) correspond to x values and the second index (the columns) correspond to y values. If using numpy.meshgrid, matrix indexing can be ensured by using the indexing='ij' keyword arg.

*x_list: numpy array / list of numpy array*

In 1D, an N element numpy array of equally spaced points in space (creating using numpy linspace or arange is advised) at which the wave will be evaluated. In 2D, a list containing two numpy arrays of length N and M respectively. These correspond to the x and y spatial grids. e.g.

```
dx = 0.01
x = dx*np.arange(201)
y = dx*np.arange(101)
psi_2d,t = pde.LW_wave_equation(psi_0_2d,[x,y],dx,N,c_2d)
```

*dx: float*

Must give the spacing between points in the x array (and y array for 2D)

*N_t: integer*

Number of time steps taken

*c: function*

In 1D, must take a numpy array argument containing spatial coords and return a numpy array of equal length giving the value of the wave speed at the given positions e.g.

```
def c(x):
  return 0.5+0.5*x
```

In 2D, must take a pair of numpy arrays containing the x and y coords and return a numpy meshgrid of the wave speeds at those points e.g.

```
def c(x,y):
  XX,YY = np.meshgrid(x,y,indexing='ij')
  return 0.5+0.5*YY
```

This gives a wavespeed that's only a function of y

*a: float*

The Courant number, for stability of the code this must be \(\leq 1\) (look up Courant-Friedrichs-Lewy stability criterion for information on this). For lower a, the code is more stable but the time step is reduced so more time steps (N) are required to simulate the same time length

*bound_cond: string*

Can be equal to 'fixed', 'reflective' and 'periodic' to impose those boundary conditions. For fixed, the wave must go to zero at the boundary. For reflective, the gradient parallel to the surface normal must vanish at the boundary. For periodic, the boundaries on opposite sides are set to be equal.

*init_grad: function*

A function which takes psi_0 as an argument and returns the gradient of the initial wave on the spatial grid. 1D example for a travelling Gaussian given below along with the init_vel example. For 2D, both \(\partial \psi / \partial x \) and \(\partial \psi / \partial y \) must be returned individually. For a 2D initially Gaussian wave:

$$ \psi_0 (x,y) = \exp (- ((x - \mu_x )^2+(y - \mu_y )^2) / 2 \sigma^2 ) \to \frac{ \partial \psi }{ \partial x} = -(x- \mu_x) \psi_0 / \sigma^2 $$

```
def twoD_gaussian(XX,YY,mean,std):
  return np.exp(-((XX-mean[0])**2+(YY-mean[1])**2)/(2*std**2))

def gradient_2d(x,y,mean,std):
  XX,YY = np.meshgrid(x,y, indexing='ij')
  def D(psi_0):
     dfdx = -(XX-mean[0])*twoD_gaussian(XX,YY,mean,std)/std**2
     dfdy = -(YY-mean[1])*twoD_gaussian(XX,YY,mean,std)/std**2
     return dfdx,dfdy
  return gradient_2d
```

Here the init_grad argument would be set to gradient_2d(x,y,mean,std) so that the LW_wave_equation program recieves the function D. This removes the need for LW_wave_equation to know the values of mean and std.

If the default argument, None, is given then the initial gradient is estimated within the program using finite differencing. It is preferable to give the program a init_grad function when there exists an analytic form.

*init_vel: function*

A function which takes psi_0 as an argument and returns the velocity (\(\partial \psi / \partial t \)) of the initial wave on the spatial grid. 1D example for a travelling Gaussian given below.

If the default argument, None, is given then the initial velocity is set to zero at all points.

Having defined the variables; x, dx, N_t, mean and std:

```
def oneD_gaussian(x,mean,std):
  return np.exp(-((x-mean)**2)/(2*std**2))

def gradient_1d(x,mean,std):
  def D(psi_0):
     return -(x-mean)*oneD_gaussian(x,mean,std)/std**2
  return D

def velocity_1d(x,mean,std):
  def V(psi_0):
     return -c(x)*(x-mean)*oneD_gaussian(x,mean,std)/std**2
  return V

psi_1d,t = pde.LW_wave_equation(oneD_gaussian(x,mean,std),x,dx,N_t,c,
        init_vel = velocity_1d(x,mean,std), init_grad = gradient_1d(x,mean,std),
        bound_cond = 'reflective')
```

**Returns:**

A N x N_t numpy array, N x M x N_t in 2D, which contains the approximated wave at different times. A N_t element numpy array is also returned containing the time interval over which the simulation was run.

### Heat Equation via a Crank-Nicolson scheme

The heat equations in 1D and 2D can be expressed as:

$$ \frac{ \partial Q}{\partial t} = \frac{ \partial}{\partial x} \left( D \frac{ \partial Q}{ \partial x} \right) $$

$$ \frac{ \partial Q}{\partial t} = \frac{ \partial}{\partial y} \left( D \frac{ \partial Q}{ \partial y} \right)+\frac{ \partial}{\partial y} \left( D \frac{ \partial Q}{ \partial y} \right) $$

For 1D and constant coefficient D, using a finite differencing method we can obtain a stable algoritm (unlike for the wave equation):

$$ \frac{Q^{n+1}_j - Q^{n}_j}{\Delta t} = D \left( \frac{Q^n_{j+1} - 2 Q^n_j + Q^n_{j-1}}{\Delta x^2} \right) $$

However in order to observe features of scale \(\lambda \gg \Delta x\) with a stable result, the number of time steps needed is unfeasibly large. As usual we will have to be smarter.

The above method is called fully explicit, if instead we evaulate the RHS at the time step \(t_{n+1}\) we create a fully implicit method:

$$ \frac{Q^{n+1}_j - Q^{n}_j}{\Delta t} = D \left( \frac{Q^{n+1}_{j+1} - 2 Q^{n+1}_j + Q^{n+1}_{j-1}}{\Delta x^2} \right) $$

This scheme is unconditionally stable yet first order in time and second order in space. We can form a method which is second order in both space and time and unconditionally stable by forming the average of the explicit and implicit schemes. This is the Crank-Nicolson scheme:

$$ \frac{Q^{n+1}_j - Q^{n}_j}{\Delta t} = \frac{D}{2} \left( \frac{Q^{n+1}_{j+1} - 2 Q^{n+1}_j + Q^{n+1}_{j-1} + Q^n_{j+1} - 2 Q^n_j + Q^n_{j-1} }{\Delta x^2} \right) $$

We now have a suitable algorithm for solving the heat equation. But it would seem it requires knowledge of \(Q\) at later time steps. However this notion can be dispelled by writing the above in a matrix equation form:

$$ \begin{bmatrix} & \vdots & \vdots & \vdots & \\ \cdots & 1+s & -s/2 & 0 & \cdots \\ \cdots & -s/2 & 1+s & -s/2 & \cdots \\ \cdots & 0 & -s/2 & 1+s & \cdots \\ & \vdots & \vdots & \vdots & \end{bmatrix} \begin{bmatrix} \vdots \\ Q^{n+1}_{j+1} \\ Q^{n+1}_{j} \\ Q^{n+1}_{j-1} \\ \vdots \end{bmatrix} = \begin{bmatrix} & \vdots & \vdots & \vdots & \\ \cdots & 1-s & s/2 & 0 & \cdots \\ \cdots & s/2 & 1-s & s/2 & \cdots \\ \cdots & 0 & s/2 & 1-s & \cdots \\ & \vdots & \vdots & \vdots & \end{bmatrix} \begin{bmatrix} \vdots \\ Q^{n}_{j+1} \\ Q^{n}_{j} \\ Q^{n}_{j-1} \\ \vdots \end{bmatrix} + \begin{bmatrix} \vdots \\ \\ b.c.s \\ \\ \vdots \end{bmatrix}$$

Where \(s = \frac{D \Delta t}{\Delta x^2} \)

Hence the matrix equation \(Ax = B \) must be solved where \(A\) is a tridiagonal matrix. Here we can use SciPy's solve_banded function to solve the above equation and advance one time step for all the points on the spatial grid.

Solving for the diffusion of a Gaussian we can compare to the analytic solution, the heat kernel:

$$ Q(x,t) \propto \frac{1}{\sqrt{2 \pi (\sigma_0^2 + 2Dt)}} \exp \left( -\frac{(x-x_0)^2}{2(\sigma_0^2 + 2Dt)} \right) $$

In the below video, the red outline shows the analytic solution and the black solid line shows the Crank-Nicolson result

#### Variable Coefficient

If \(D\) is a function of position then \(s\) needs to be evaulated at the spatial point. The form of this finite difference can be seen in finite difference methods page under introductory documentation.

#### Argument list

CN_diffusion_equation(T_0, D, x_list, dx, N_t, s = 0.25, wall_T = [0.0,0.0,0.0,0.0])

This function performs the Crank-Nicolson scheme for 1D and 2D problems to solve the inital value problem for the heat equation.

**Parameters:**

*T_0: numpy array*

In 1D, an N element numpy array containing the intial values of T at the spatial grid points. In 2D, a NxM array is needed where N is the number of x grid points, M the number of y grid points. This array needs to be in "matrix indexing" rather than "Cartesian indexing" i.e. the first index (the rows) correspond to x values and the second index (the columns) correspond to y values. If using numpy.meshgrid, matrix indexing can be ensured by using the indexing='ij' keyword arg.

*D: function*

In 1D, must take a numpy array argument containing spatial coords and return a numpy array of equal length giving the value of the diffusivity at the given positions e.g.

```
def D(x):
  return 0.5+0.5*x
```

In 2D, must take a pair of floats of the x and y coords and return a float of the diffusivity at that point e.g.

```
def D(x,y):
  return 0.5+0.5*(x-0.5)**2+0.5*(y-0.5)**2
```

*x_list: numpy array / list of numpy array*

In 1D, an N element numpy array of equally spaced points in space (creating using numpy linspace or arange is advised) at which the wave will be evaluated. In 2D, a list containing two numpy arrays of length N and M respectively. These correspond to the x and y spatial grids. e.g.

```
dx = 0.01
x = dx*np.arange(201)
y = dx*np.arange(101)
T,t = pde.CN_diffusion_equation(T_0, D, [x,y], dx, N_t)
```

*dx: float*

Must give the spacing between points in the x array (and y array for 2D)

*N_t: integer*

Number of time steps taken

*s: float*

This is used to set the time step via \(\Delta t = s \Delta x^2 \). Although the scheme is stable for any size \(\Delta t \) in order to ensure accurate results s should set sufficiently low. Generally of order \(1/D_{max} \) is advisable.

*wall_T: list of floats*

A list of 2 or 4 floats (for 1D or 2D) containing the fixed T values for the boundaries.

**Returns:**

A N x N_t numpy array, N x M x N_t in 2D, which contains the approximated T at different times. A N_t element numpy array is also returned containing the time interval over which the simulation was run.

### Time-Dependent Schrödinger equation via the Split-Step Fourier method

Writing the Schrödinger equation in the form (in units where \(\hbar = 1 \)): $$ \frac{ \partial \psi}{\partial t} = i \mathcal{L} \psi+i \mathcal{N} \psi $$

Where for the TDSE: $$ \mathcal{L} = \frac{1}{2m} \frac{\partial^2}{\partial x^2}, \mathcal{N} = -V(x) $$

The time evolution operator is then given by:

$$ \psi (x,t_0 + t) = \exp (i (\mathcal{L} + \mathcal{N}) t) \psi (x,t_0) $$

We can split the exponential if \([ \mathcal{L}, \mathcal{N} ] = 0 \). This is not necessarily true but for a small time interval, \( \Delta t \), commutativity can be assumed with an error of order \( \Delta t^2 \).

By first neglecting \(\mathcal{L} \) in time interval \([t_0,t_0+ \Delta t/2]\) we are left with an ODE with a solution of the form:

$$ \psi (x,t_0 + \Delta t / 2) = \exp (i \Delta t \mathcal{N} /2) \psi (x,t_0) $$

Now neglecting \(\mathcal{N} \), moving to momentum space \(\mathcal{L} \) is simply multiplication. Hence in the full time interval \(\Delta t\):

$$ \tilde{ \psi } (k,t_0 + \Delta t) = \exp (i \Delta t \mathcal{F} ( \mathcal{L})) \tilde{ \psi }(k,t_0) = \exp (-i \Delta t k^2 / 2 m) \tilde{ \psi }(k,t_0) $$

For the initial \( \tilde{ \psi }(k,t_0)\) we use the Fourier transform of the time half step result we found first. Finally we must perform an additional spatial domain time half step to recover the split step approximation to time evolution operator for \(\mathcal{L} + \mathcal{N}\) by \( \Delta t\).

In full, the process is the following:

$$ \psi (x,t_0+ \Delta t) = \exp (i \Delta t \mathcal{N} /2) \mathcal{F}^{-1}( \exp (i \Delta t \mathcal{F}( \mathcal{L})) \mathcal{F} (\exp(i \Delta t \mathcal{N} /2) \psi(x,t_0))) $$

We will be using Fast Fourier Transforms (FFTs) from the SciPy library so need to take into consideration the discrete nature of our input.

The basic argument behind this is to match the continuous Fourier transform pair \( \psi(x,t) \leftrightarrow \tilde{ \psi} (k,t)\) to a discrete approximation, \( \psi(x_n,t) \leftrightarrow \tilde{ \psi} (k_m,t)\). Here we use n and m to index x and k:

Starting with a continuous Fourier transform, we can form the discrete approximation:

$$ \tilde \psi (k,t) = \frac{1}{ \sqrt{2 \pi}} \int^\infty _\infty \psi (x,t) e^{-ikx} dx \to \tilde \psi (k_m,t) \approx \frac{ \Delta x}{ \sqrt{2 \pi}} \sum^{N-1}_{n=0} \psi(x_n,t) e^{-ik_mx_n} $$

Comparing these to the discrete Fourier transform definitions we find the discrete Fourier transform pair:

$$ \frac{ \Delta x}{ \sqrt{2 \pi}} \psi(x_n,t) e^{-ik_0x_n} \leftrightarrow \tilde \psi (k_m,t) e^{im \Delta k x_0} $$

Where \( \Delta k = 2 \pi / (N \Delta x) \)

Note that just as we have limited the range of x above, we have here limited the range of k as well. This means that high-frequency components of the signal will be lost in our approximation. The Nyquist sampling theorem tells us that this is an unavoidable consequence of choosing discrete steps in space.

*It should be noted that the wavefunctions reaching the boundaries should be avoided. The spatial domain should be large enough and the inital wavefunction should be of a suitable form e.g. Gaussian wavepackets*

#### Gaussian wavepackets

A Gaussian wavepackets can be used to investigate the quantum mechanical evolution of particles as they are well localized in both real and momentum space (i.e. they are minimum uncertainty states). However a Gaussian wavepacket will spread over time so a smart choice of a initial wavefunction will help the wavepacket last long enough to observe long times. The width of the probability density increases with time as follows:

For a initial wavepacket of the form:

$$ \psi (x, t = t_0) \propto \exp \left(- \frac{(x-x_0)^2}{4 \sigma_0^2 } + i k_0 x \right) $$

$$ \sigma^2 (t) = \sigma^2_0 + \frac{1}{\sigma^2_0} \frac{t^2}{4 m^2} $$

Hence the width of the packet is minimized at the end of the simulation when

$$ \frac{d \sigma}{\sigma_0} = 0 \to \sigma^2_0 = \frac{t}{2m} $$

Below is an example of setting up such a wavepacket:

```
def oneD_gaussian(x,mean,std,k0):
    return np.exp(-((x-mean)**2)/(4*std**2)+ 1j*x*k0)/(2*np.pi*std**2)**0.25

dt = 0.01
N_t = 2000

p0 = 2.0
d = np.sqrt(N_t*dt/2.)

psi_0 = oneD_gaussian(x,x.max()-10*d,d,-p0)
```

As an example of its use, here is a gaussian wavepacket incident on a potential barrier:

### Non-Linear Schrödinger

The non-linear Schrödinger equation includes a term which depends on the probability density of the wavefunction. This can be included by modifying our \(\mathcal{N}\) operator:

$$ \mathcal{N} = - V(x) + \kappa \left| \psi (x,t) \right| ^2 $$

Depending on the sign, this corresponds to a repulsive or attractive contact potential between particles described by the wavefunction.

### Argument list

split_step_schrodinger(psi_0, dx, dt, V, N_t, x_0 = 0., k_0 = None, m = 1.0, non_linear = False)

This function performs the split-step Fourier method to solve the 1D time-dependent Schrödinger equation for a given potential

**Parameters:**

*psi_0: numpy array*

In 1D, an N element numpy array containing the intial values of \(\psi\) at the spatial grid points. In 2D, a NxM array is needed where N is the number of x grid points, M the number of y grid points. This array needs to be in "matrix indexing" rather than "Cartesian indexing" i.e. the first index (the rows) correspond to x values and the second index (the columns) correspond to y values. If using numpy.meshgrid, matrix indexing can be ensured by using the indexing='ij' keyword arg.

*dx: float*

Must give the spacing between points in the x array

*dt: float*

Gives the time step taken within the split-step algorithm. This needs to be small to reduce the size of numerical errors (try 0.01 as a safe starting value)

*V: function*

Pass a function which takes a numpy array argument containing spatial coords and returns the potential at that point e.g.

```
def V(x):
  V_x = np.zeros_like(x)
  a = 0.5
  x_mid = (x.max()+x.min())/2.
  V_x = -a**2*(1/np.cosh(a*(x-x_mid)))**2
  return V_x
```

If non_linear = True then the potential function must now take an additional argument which is equal to the spatial wavefunction at the current time step e.g.

```
def V(x,psi):
  V_x = np.zeros_like(x)
  V_x = -200.*np.absolute(psi)**2+0.05*x**2
  return V_x
```

*N_t: integer*

Number of time steps taken

*x_0: float*

Give the starting position of the spatial grid

*k_0: float*

Gives the starting position of the momentum space grid. If none is given then k_0 is set to \(-\pi/ \Delta x \) as it can be shown that this exactly satisfies the Nyquist limit.

*m: float*

The mass of the particle (default value of 1.0)

*non_linear: boolean*

Set to True if investigating the non-linear Schrödinger equation. Default is False

**Returns:**

Two N x N_t numpy arrays which contain the approximated real space and momentum space wavefunctions at different times. A N element numpy array is also returned containing the k space interval used.

The list below summarises the functions, their input arguments and their outputs for quick reference for the informed user:

### 2.3.4 Functions

#### LW_wave_equation(psi_0, x_list, dx, N_t, c, a = 1., bound_cond = 'periodic',init_grad = None, init_vel = None)

This function performs the two-step Lax-Wendroff scheme for 1D problems and a Lax method for 2D problems to solve a flux-conservative form of the wave equation for variable wave speed, c.

**Parameters:**

*psi_0: numpy array*

In 1D, an N element numpy array containing the intial values of \(\psi\) at the spatial grid points. In 2D, a NxM array is needed where N is the number of x grid points, M the number of y grid points. This array needs to be in "matrix indexing" rather than "Cartesian indexing" i.e. the first index (the rows) correspond to x values and the second index (the columns) correspond to y values. If using numpy.meshgrid, matrix indexing can be ensured by using the indexing='ij' keyword arg.

*x_list: numpy array / list of numpy array*

In 1D, an N element numpy array of equally spaced points in space (creating using numpy linspace or arange is advised) at which the wave will be evaluated. In 2D, a list containing two numpy arrays of length N and M respectively. These correspond to the x and y spatial grids. e.g.

```
dx = 0.01
x = dx*np.arange(201)
y = dx*np.arange(101)
psi_2d,t = pde.LW_wave_equation(psi_0_2d,[x,y],dx,N,c_2d)
```

*dx: float*

Must give the spacing between points in the x array (and y array for 2D)

*N_t: integer*

Number of time steps taken

*c: function*

In 1D, must take a numpy array argument containing spatial coords and return a numpy array of equal length giving the value of the wave speed at the given positions e.g.

```
def c(x):
  return 0.5+0.5*x
```

In 2D, must take a pair of numpy arrays containing the x and y coords and return a numpy meshgrid of the wave speeds at those points e.g.

```
def c(x,y):
  XX,YY = np.meshgrid(x,y,indexing='ij')
  return 0.5+0.5*YY
```

This gives a wavespeed that's only a function of y

*a: float*

The Courant number, for stability of the code this must be \(\leq 1\) (look up Courant-Friedrichs-Lewy stability criterion for information on this). For lower a, the code is more stable but the time step is reduced so more time steps (N) are required to simulate the same time length

*bound_cond: string*

Can be equal to 'fixed', 'reflective' and 'periodic' to impose those boundary conditions. For fixed, the wave must go to zero at the boundary. For reflective, the gradient parallel to the surface normal must vanish at the boundary. For periodic, the boundaries on opposite sides are set to be equal.

*init_grad: function*

A function which takes psi_0 as an argument and returns the gradient of the initial wave on the spatial grid. 1D example for a travelling Gaussian given below along with the init_vel example. For 2D, both \(\partial \psi / \partial x \) and \(\partial \psi / \partial y \) must be returned individually. For a 2D initially Gaussian wave:

$$ \psi_0 (x,y) = \exp (- ((x - \mu_x )^2+(y - \mu_y )^2) / 2 \sigma^2 ) \to \frac{ \partial \psi }{ \partial x} = -(x- \mu_x) \psi_0 / \sigma^2 $$

```
def twoD_gaussian(XX,YY,mean,std):
  return np.exp(-((XX-mean[0])**2+(YY-mean[1])**2)/(2*std**2))

def gradient_2d(x,y,mean,std):
  XX,YY = np.meshgrid(x,y, indexing='ij')
  def D(psi_0):
     dfdx = -(XX-mean[0])*twoD_gaussian(XX,YY,mean,std)/std**2
     dfdy = -(YY-mean[1])*twoD_gaussian(XX,YY,mean,std)/std**2
     return dfdx,dfdy
  return gradient_2d
```

Here the init_grad argument would be set to gradient_2d(x,y,mean,std) so that the LW_wave_equation program recieves the function D. This removes the need for LW_wave_equation to know the values of mean and std.

If the default argument, None, is given then the initial gradient is estimated within the program using finite differencing. It is preferable to give the program a init_grad function when there exists an analytic form.

*init_vel: function*

A function which takes psi_0 as an argument and returns the velocity (\(\partial \psi / \partial t \)) of the initial wave on the spatial grid. 1D example for a travelling Gaussian given below.

If the default argument, None, is given then the initial velocity is set to zero at all points.

Having defined the variables; x, dx, N_t, mean and std:

```
def oneD_gaussian(x,mean,std):
  return np.exp(-((x-mean)**2)/(2*std**2))

def gradient_1d(x,mean,std):
  def D(psi_0):
     return -(x-mean)*oneD_gaussian(x,mean,std)/std**2
  return D

def velocity_1d(x,mean,std):
  def V(psi_0):
     return -c(x)*(x-mean)*oneD_gaussian(x,mean,std)/std**2
  return V

psi_1d,t = pde.LW_wave_equation(oneD_gaussian(x,mean,std),x,dx,N_t,c,
        init_vel = velocity_1d(x,mean,std), init_grad = gradient_1d(x,mean,std),
        bound_cond = 'reflective')
```

**Returns:**

A N x N_t numpy array, N x M x N_t in 2D, which contains the approximated wave at different times. A N_t element numpy array is also returned containing the time interval over which the simulation was run.

## CN_diffusion_equation(T_0, D, x_list, dx, N_list, s = 0.25, wall_T = [0.0,0.0,0.0,0.0])

This function performs the Crank-Nicolson scheme for 1D and 2D problems to solve the inital value problem for the heat equation.

**Parameters:**

*T_0: numpy array*

In 1D, an N element numpy array containing the intial values of T at the spatial grid points. In 2D, a NxM array is needed where N is the number of x grid points, M the number of y grid points. This array needs to be in "matrix indexing" rather than "Cartesian indexing" i.e. the first index (the rows) correspond to x values and the second index (the columns) correspond to y values. If using numpy.meshgrid, matrix indexing can be ensured by using the indexing='ij' keyword arg.

*D: function*

In 1D, must take a numpy array argument containing spatial coords and return a numpy array of equal length giving the value of the diffusivity at the given positions e.g.

```
def D(x):
  return 0.5+0.5*x
```

In 2D, must take a pair of floats of the x and y coords and return a float of the diffusivity at that point e.g.

```
def D(x,y):
  return 0.5+0.5*(x-0.5)**2+0.5*(y-0.5)**2
```

*x_list: numpy array / list of numpy array*

In 1D, an N element numpy array of equally spaced points in space (creating using numpy linspace or arange is advised) at which the wave will be evaluated. In 2D, a list containing two numpy arrays of length N and M respectively. These correspond to the x and y spatial grids. e.g.

```
dx = 0.01
x = dx*np.arange(201)
y = dx*np.arange(101)
T,t = pde.CN_diffusion_equation(T_0, D, [x,y], dx, N_t)
```

*dx: float*

Must give the spacing between points in the x array (and y array for 2D)

*N_t: integer*

Number of time steps taken

*s: float*

This is used to set the time step via \(\Delta t = s \Delta x^2 \). Although the scheme is stable for any size \(\Delta t \) in order to ensure accurate results s should set sufficiently low. Generally of order \(1/D_{max} \) is advisable.

*wall_T: list of floats*

A list of 2 or 4 floats (for 1D or 2D) containing the fixed T values for the boundaries.

**Returns:**

A N x N_t numpy array, N x M x N_t in 2D, which contains the approximated T at different times. A N_t element numpy array is also returned containing the time interval over which the simulation was run.

### split_step_schrodinger(psi_0, dx, dt, V, N, x_0 = 0., k_0 = None, m = 1.0, non_linear = False)

This function performs the split-step Fourier method to solve the 1D time-dependent Schrödinger equation for a given potential

**Parameters:**

*psi_0: numpy array*

In 1D, an N element numpy array containing the intial values of \(\psi\) at the spatial grid points. In 2D, a NxM array is needed where N is the number of x grid points, M the number of y grid points. This array needs to be in "matrix indexing" rather than "Cartesian indexing" i.e. the first index (the rows) correspond to x values and the second index (the columns) correspond to y values. If using numpy.meshgrid, matrix indexing can be ensured by using the indexing='ij' keyword arg.

*dx: float*

Must give the spacing between points in the x array

*dt: float*

Gives the time step taken within the split-step algorithm. This needs to be small to reduce the size of numerical errors (try 0.01 as a safe starting value)

*V: function*

Pass a function which takes a numpy array argument containing spatial coords and returns the potential at that point e.g.

```
def V(x):
  V_x = np.zeros_like(x)
  a = 0.5
  x_mid = (x.max()+x.min())/2.
  V_x = -a**2*(1/np.cosh(a*(x-x_mid)))**2
  return V_x
```

If non_linear = True then the potential function must now take an additional argument which is equal to the spatial wavefunction at the current time step e.g.

```
def V(x,psi):
  V_x = np.zeros_like(x)
  V_x = -200.*np.absolute(psi)**2+0.05*x**2
  return V_x
```

*N_t: integer*

Number of time steps taken

*x_0: float*

Give the starting position of the spatial grid

*k_0: float*

Gives the starting position of the momentum space grid. If none is given then k_0 is set to \(-\pi/ \Delta x \) as it can be shown that this exactly satisfies the Nyquist limit.

*m: float*

The mass of the particle (default value of 1.0)

*non_linear: boolean*

Set to True if investigating the non-linear Schrödinger equation. Default is False

**Returns:**

Two N x N_t numpy arrays which contain the approximated real space and momentum space wavefunctions at different times. A N element numpy array is also returned containing the k space interval used.

## 2.4 Optics

### 2.4.1 Introduction to the Optics module

This module contains functions for use in geometric optics problems. The example usage is for caustics, such those observed at the bottom of a swimming pool. As we are working in the geometric optics limit, this simply requires manipulation of vectors. Refraction is a rotation about a axis perpendicular to the surface normal and incoming wave vector by an angle determined by Snell's law.

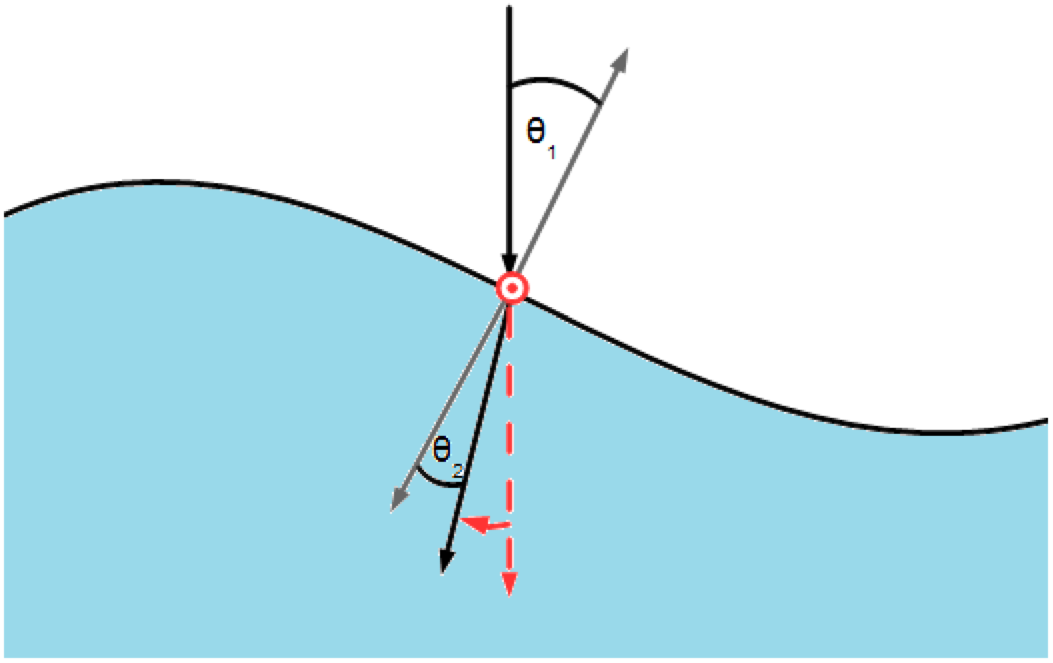
You can find the source code for this module on the GitHub.

### 2.4.2 In-depth Documentation

#### Ray Object

The optics module uses an object orientated approach based around the Ray object. A ray is defined by its direction and its behaviour is determined by the environment it is propagating through. Hence the ray object has attributes about itself, such as its position of incidence and its wave vector, and about the surrounding media, such as the refractive indices and media interfaces functional form.

To perform refraction on a ray we must know the surface normal and the angle of incidence. Using Snell's law we can find $\theta_2$ and hence the rotation required to rotate the incident ray onto the refracted ray (shown in the diagram below by the hashed red arrow rotating onto the refracted black ray).



The rotation axis is the normalised cross product of the incident wave vector and the surface normal. The angle of rotation is $\theta_{12} = \theta_1 - \theta_2$. Hence the full transformation required is the following:

$$ R_{tot} = R_{z \to r}^{-1} R_{\theta_{12}} R_{z \to r} $$

$R_{\theta_{12}}$ performs a rotation by $\theta_{12}$ about the z axis this has the form:

$$ R_{\theta_{12}} = \begin{bmatrix} \cos (\theta_{12}) & \sin (\theta_{12}) & 0 \\ -\sin (\theta_{12}) & \cos (\theta_{12}) & 0 \\ 0 & 0& 1 \end{bmatrix} $$

\(R_{z \to r}\) rotates the z axis onto the rotation axis, r. To form this transform let us consider rotating a unit vector \(a\) onto another \(b\). The axis of rotation is along \(x = a \times b\). Using Rodrigues' rotation formula in its matrix form to rotate \(a\) about \(x\) by an angle \(\theta\):

$$ \mathbf{b} = \left( I + \sin(\theta) X + (1 - \cos(\theta)) X^2 \right) \mathbf{a} $$

Where \(X\) is the cross-product matrix for \(x\) defined by

$$ X = \frac{1}{\left| x \right|}\begin{bmatrix} 0 & -x_3 & x_2 \\ x_3 & 0 & -x_1 \\ -x_2 & x_1 & 0 \end{bmatrix} $$

Also \( \left| x \right| = \sin(\theta)\) and \(a \cdot b = \cos(\theta) \)

Hence replacing \(a\) by a unit vector pointing along z and \(b\) by the normalised cross product of the unit surface normal and the unit incident wave vector, we obtain the required rotation matrix.

With total rotation matrix formed we can calculate the position of the refracted rays at the base of the refracting medium. This is simply done by enlarging the length of the vector until the z component is equal to the distance to the bottom, \(h+f(x,y)\). The refracted ray position can then be read off from the x and y components of the vector (plus the rays initial position).

### Ray Object

Ray(x,y,f,dfdx,dfdy,h,n_arr) (__init__ function)

**Parameters:**

*x,y: numpy arrays*

Numpy arrays containg the x and y coordinates of the rays before refraction

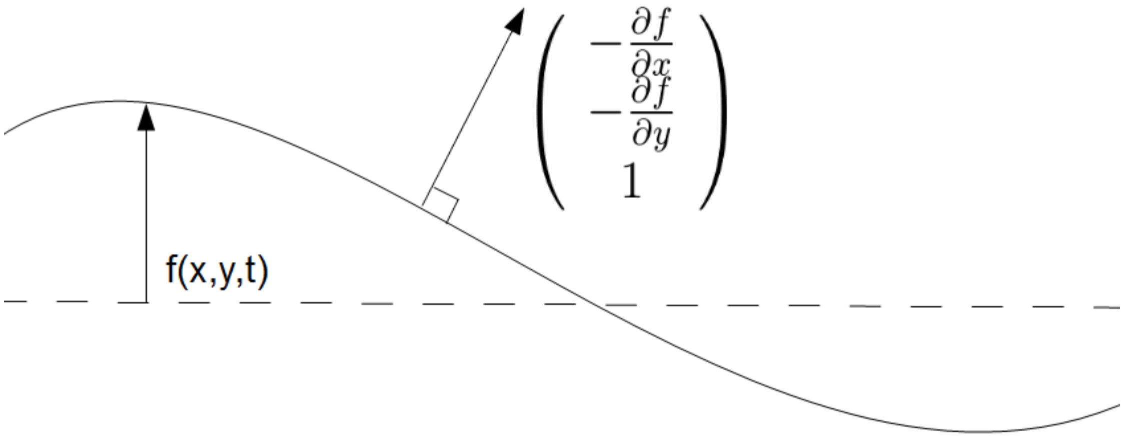
*f, dfdx, dfdy: functions*

f: A function of x, y and t which gives the displacement of the second mediums surface

dfdx: A function of x, y and t which gives the partial differential of f along x

dfdy: A function of x, y and t which gives the partial differential of f along y

These are used to compute the unit normal and the height the ray must descend to reach the bottom of the second medium


$\begin{pmatrix} -\frac{\partial f}{\partial x} \\ -\frac{\partial f}{\partial y} \\ 1 \end{pmatrix}$
f(x,y,t)

*n_arr: list of floats*

A list of two floats, the first being the refractive index of the first medium and the second that of the second medium *h: height*

The height of the second medium, the incident light rays which refract will be traced until they reach this height

findnormal(self,t)

Finds the normalised normal vector at position of the incident ray

**Parameters:**

*t: numpy array*

A numpy array with the time coordinates for which a refracted image will be displayed

snellsangle(self)

Calculates the angle of refraction via Snell's law

#### transformation_matrix(self)

Calculates the rotation matrix required to rotate the incident ray onto the refracted ray

#### refract(self,t)

Computes the refracted ray positions using the rotation matrix

**Parameters:**

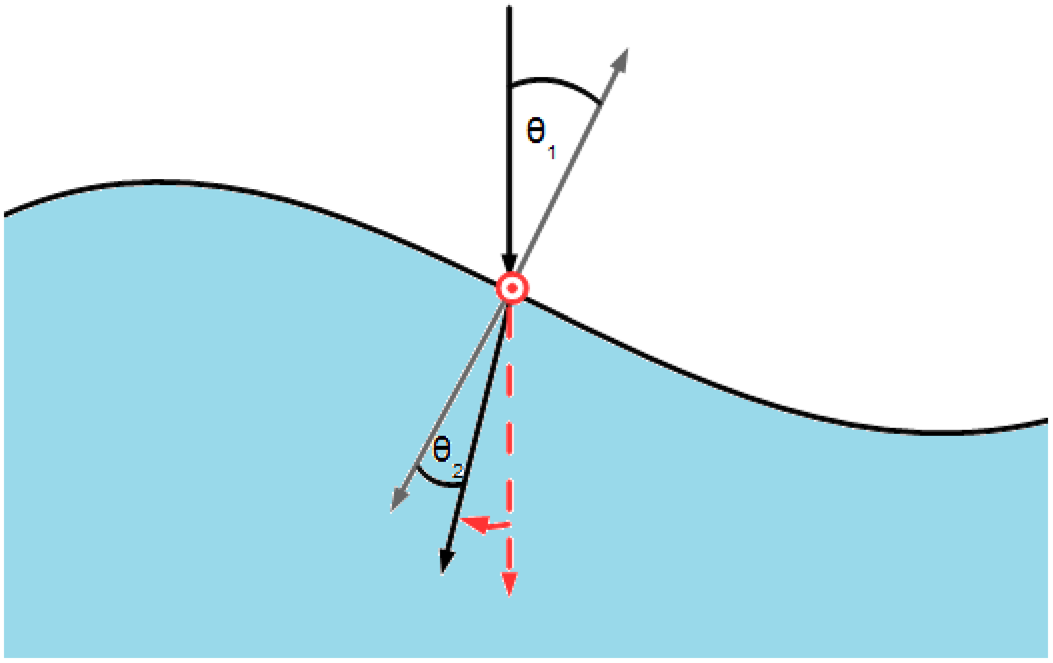
*t: numpy array*

A numpy array with the time coordinates for which a refracted image will be displayed

## Caustics and Dispersion

### Caustics

A caustic is the envelope of light rays reflected or refracted by a curved surface or object, or the projection of that envelope of rays on another surface. An example of this is the light seen at the bottom of a swimming pool. If the second media has a larger refractive index then raised regions will focus the light as the entering light is bent towards the surface normal.



Using the method described in the Ray Object page , caustics can be simulated for a simple two media system.

An example of its use can be found here.

The functions described below allow for easy use of the Ray Object to produce caustic images. The steps behind this method are as follows:

1. A regular grid of normal incident rays are set up at the interface
2. The rays are refracted based on the surface normal at incident point
3. The rays are traced to the bottom of the medium, the viewing screen is placed at a depth h below the interface
4. The screen is split into boxes and the number of rays in each box is counted i.e. a 2D histogram
5. The image is then created or the steps 1-4 are run for different times to create a dynamic caustic simulation

In terms of Optics module functions:

ray_grid → rays_refract → single_time_image/evolve (→ ray_count) → caustic_image/caustic_anim

### Dispersion

Dispersion can also be investigated. This is simply done by giving different refractive indices to different collections of Ray objects. The same method described for forming the caustic map can then be used to track the paths of the different wavelengths. Then by choosing appropriate colourmaps, different wavelengths in the incident light can seen to have seperated in the resulting image.

An example of programming this can be found in this notebook.

### Argument list

ray_grid(N_x,N_y,n_arr,h,f,dfdx,dfdy,x_lims = [0.,1.], y_lims = [0.,1.])

**Parameters:**

*N_x: integer*

The number of grid points along the x axis

*N_y: integer*

The number of grid points along the y axis

*n_arr: list of floats*

A list of two floats, the first being the refractive index of the first medium and the second that of the second medium

*h: height*

The height of the second medium, the incident light rays which refract will be traced until they reach this height

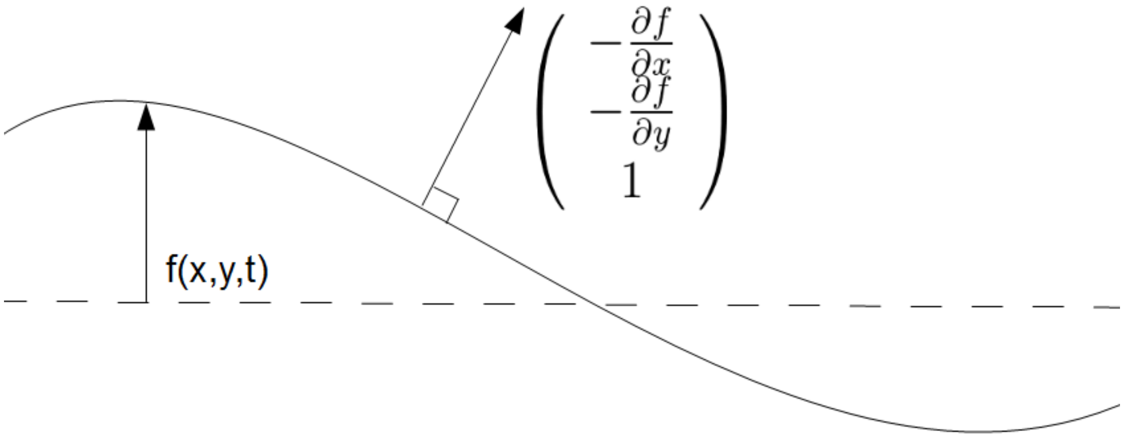
*f, dfdx, dfdy: functions*

f: A function of x, y and t which gives the displacement of the second mediums surface

dfdx: A function of x, y and t which gives the partial differential of f along x

dfdy: A function of x, y and t which gives the partial differential of f along y

These are used to compute the unit normal and the height the ray must descend to reach the bottom of the second medium


$\begin{pmatrix} -\frac{\partial f}{\partial x} \\ -\frac{\partial f}{\partial y} \\ 1 \end{pmatrix}$
f(x,y,t)

*x_lims: list of floats*

A list of floats which contain the range of x values over which the grid is defined

*y_lims: list of floats*

A list of floats which contain the range of y values over which the grid is defined

**Returns:**

The coordinate grids x and y and a list of N_x x N_y Ray objects defined on the grid incident on a surface with the given functional form and given refractive index

rays_refract(rays,t)

**Parameters:**

*rays: list of Ray objects*

A list of Ray objects either defined via ray_grid or one made individually *t: numpy array*

A numpy array with the time coordinates for which a refracted image will be calculated

**Returns:**

Returns the x and y coordinates of the rays after refraction and having travelled a vertical distance h + f

ray_count(rays_x,rays_y,boxsize_x,boxsize_y)

**Parameters:**

*rays_x,rays_y: numpy arrays*

Arrays containing the x and y coordinates of the refracted arrays, these are produced by the rays_refract function

*boxsize_x,boxsize_y: floats*

Floats which define the size of histogram bins used to evaulate a light intensity from the refracted image

**Returns:**

Two regular coordinate meshgrids, XX and YY, and the number of rays in each bin i.e. the 2D histogram data, I.

single_time_image(rays,boxsize_x,boxsize_y)

This function calls ray_refract followed by ray_count. These are evaluated at t = 0 so is useful for surface functions f, dfdx, dfdy which are time independent

**Parameters:**

*rays: list of Ray objects*

A list of Ray objects either defined via ray_grid or one made individually

*boxsize_x,boxsize_y: floats*

Floats which define the size of histogram bins used to evaulate a light intensity from the refracted image

**Returns:**

Two numpy arrays and three numpy meshgrids. The numpy arrays contain the x and y coordinates of the refracted rays, these can be used to visualise the refracted ray locations simply by creating a scatter plot of x against y. The numpy meshgrids give the coordinate meshgrids and 2D histogram data needed to plot an intensity map

evolve(rays,t,boxsize_x,boxsize_y)

This function calls ray_refract followed by ray_count for each time step within the array t.

**Parameters:**

*rays: list of Ray objects*

A list of Ray objects either defined via ray_grid or one made individually

*t: numpy array*

A numpy array with the time coordinates for which a refracted image will be calculated

*boxsize_x,boxsize_y: floats*

Floats which define the size of histogram bins used to evaulate a light intensity from the refracted image

**Returns:**

Three lists, the first two are lists of coordinate meshgrids for the different time evaluations. The third list is a list of 2D histogram data points for the different time evaluations. Hence all the data needed to plot the time evolution of the caustic image is created by this function

caustic_image(x,y,N,XX,YY,II,h,f,disturbance_height,plot_height,c_map = 'Blues_r')

Creates a 3D plot displaying a scaled media interface and the refracted ray intensity image

**Parameters:**

*x,y: numpy arrays*

Numpy arrays containg the x and y coordinates of the rays *before* refraction

*N: list of integers*

A list containing N_x and N_y used to create the ray grid before refraction

*XX,YY: numpy meshgrids*

Coordinate meshgrids for the refracted ray positions i.e. those created by ray_count

*II: numpy meshgrid*

Meshgrid containing the number of refracted rays within bins on the above coordinate meshgrid

*h: float*

The height of the second medium

*f: function*

The surface displacement of the second medium

*disturbance_height: float*

The maximum value of the function f for all x,y and t

*plot_height: float*

The factor by which the surface plot is scaled when displayed in the plot (value of 0.25 works well)

*c_map: colormap*

Colormap used to plot the refracted ray intensity map

**Returns:**

Creates a 3D plot displaying a scaled media interface and the refracted ray intensity image

caustic_anim(x,y,t,N,XX_t,YY_t,II_t,h,f,disturbance_height,plot_height,c_map='Blues_r',interval = 100,fname = None)

Creates an animated 3D plot displaying a scaled media interface and the refracted ray intensity image

**Parameters:**

*x,y: numpy arrays*

Numpy arrays containg the x and y coordinates of the rays *before* refraction

*t: numpy array*

A numpy array with the time coordinates for which a refracted image will be displayed

*N: list of integers*

A list containing N_x and N_y used to create the ray grid before refraction

*XX_t,YY_t: lists of numpy meshgrids*

Lists of coordinate meshgrids for the different time evaluations, these can be produced by the evolve function

*II_t: list of numpy meshgrids*

List of meshgrids containing the number of refracted rays within bins on the above coordinate meshgrids for different time evaluations, this can be produced by the evolve function

*h: float*

The height of the second medium

*f: function*

The surface displacement of the second medium

*disturbance_height: float*

The maximum value of the function f for all x,y and t

*plot_height: float*

The factor by which the surface plot is scaled when displayed in the plot (value of 0.25 works well)

*c_map: colormap*

Colormap used to plot the refracted ray intensity map

*interval: integer*

The number of milliseconds between frames in the animation

*fname: string*

Name of file to which the animation will be save. If left as default None argument then a temporary file will be used instead

**Returns:**

Creates a 3D animated plot displaying a scaled media interface and the refracted ray intensity image

The list below summarises the functions, their input arguments and their outputs for quick reference for the informed user:

## 2.4.3 Functions

### ray_grid(N_x,N_y,n_arr,h,f,dfdx,dfdy,x_lims = [0.,1.], y_lims = [0.,1.])

**Parameters:**

*N_x: integer*

The number of grid points along the x axis

*N_y: integer*

The number of grid points along the y axis

*n_arr: list of floats*

A list of two floats, the first being the refractive index of the first medium and the second that of the second medium

*h: height*

The height of the second medium, the incident light rays which refract will be traced until they reach this height

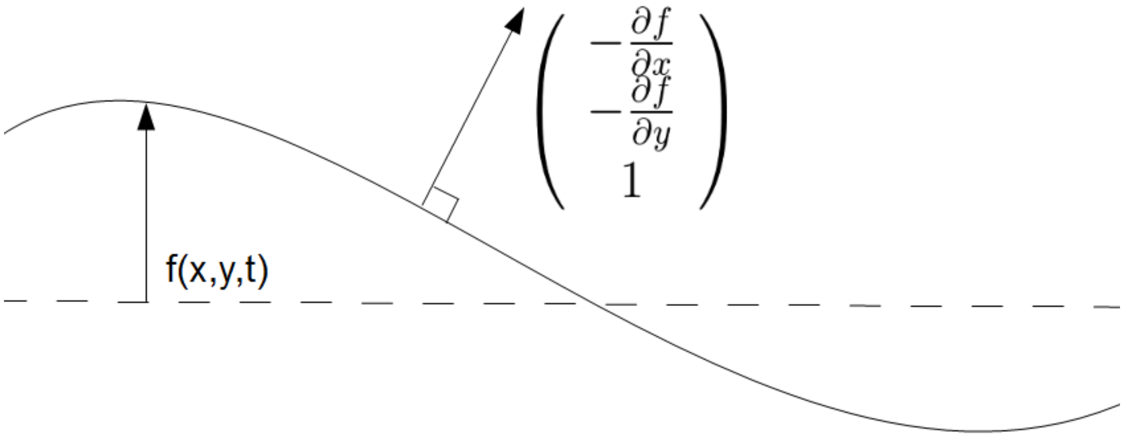
*f, dfdx, dfdy: functions*

f: A function of x, y and t which gives the displacement of the second mediums surface

dfdx: A function of x, y and t which gives the partial differential of f along x

dfdy: A function of x, y and t which gives the partial differential of f along y

These are used to compute the unit normal and the height the ray must descend to reach the bottom of the second medium


$\begin{pmatrix} -\frac{\partial f}{\partial x} \\ -\frac{\partial f}{\partial y} \\ 1 \end{pmatrix}$
f(x,y,t)

*x_lims: list of floats*

A list of floats which contain the range of x values over which the grid is defined

*y_lims: list of floats*

A list of floats which contain the range of y values over which the grid is defined

**Returns:**

The coordinate grids x and y and a list of N_x x N_y Ray objects defined on the grid incident on a surface with the given functional form and given refractive index

## rays_refract(rays,t)

**Parameters:**

*rays: list of Ray objects*

A list of Ray objects either defined via ray_grid or one made individually

*t: numpy array*

A numpy array with the time coordinates for which a refracted image will be calculated

**Returns:**

Returns the x and y coordinates of the rays after refraction and having travelled a vertical distance h + f

## ray_count(rays_x,rays_y,boxsize_x,boxsize_y)

**Parameters:**

*rays_x,rays_y: numpy arrays*

Arrays containing the x and y coordinates of the refracted arrays, these are produced by the rays_refract function

*boxsize_x,boxsize_y: floats*

Floats which define the size of histogram bins used to evaulate a light intensity from the refracted image

**Returns:**

Two regular coordinate meshgrids, XX and YY, and the number of rays in each bin i.e. the 2D histogram data, I.

### single_time_image(rays,boxsize_x,boxsize_y)

This function calls ray_refract followed by ray_count. These are evaluated at t = 0 so is useful for surface functions f, dfdx, dfdy which are time independent

**Parameters:**

*rays: list of Ray objects*

A list of Ray objects either defined via ray_grid or one made individually

*boxsize_x,boxsize_y: floats*

Floats which define the size of histogram bins used to evaulate a light intensity from the refracted image

**Returns:**

Two numpy arrays and three numpy meshgrids. The numpy arrays contain the x and y coordinates of the refracted rays, these can be used to visualise the refracted ray locations simply by creating a scatter plot of x against y. The numpy meshgrids give the coordinate meshgrids and 2D histogram data needed to plot an intensity map

### evolve(rays,t,boxsize_x,boxsize_y)

This function calls ray_refract followed by ray_count for each time step within the array t.

**Parameters:**

*rays: list of Ray objects*

A list of Ray objects either defined via ray_grid or one made individually

*t: numpy array*

A numpy array with the time coordinates for which a refracted image will be calculated

*boxsize_x,boxsize_y: floats*

Floats which define the size of histogram bins used to evaulate a light intensity from the refracted image

**Returns:**

Three lists, the first two are lists of coordinate meshgrids for the different time evaluations. The third list is a list of 2D histogram data points for the different time evaluations. Hence all the data needed to plot the time evolution of the caustic image is created by this function

### caustic_image(x,y,N,XX,YY,II,h,f,disturbance_height,plot_height,c_map = 'Blues_r')

Creates a 3D plot displaying a scaled media interface and the refracted ray intensity image

**Parameters:**

*x,y: numpy arrays*

Numpy arrays containg the x and y coordinates of the rays *before* refraction

*N: list of integers*

A list containing N_x and N_y used to create the ray grid before refraction

*XX,YY: numpy meshgrids*

Coordinate meshgrids for the refracted ray positions i.e. those created by ray_count

*II: numpy meshgrid*

Meshgrid containing the number of refracted rays within bins on the above coordinate meshgrid

*h: float*

The height of the second medium

*f: function*

The surface displacement of the second medium

*disturbance_height: float*

The maximum value of the function f for all x,y and t

*plot_height: float*

The factor by which the surface plot is scaled when displayed in the plot (value of 0.25 works well)

*c_map: colormap*

Colormap used to plot the refracted ray intensity map

**Returns:**

Creates a 3D plot displaying a scaled media interface and the refracted ray intensity image

### caustic_anim(x,y,t,N,XX_t,YY_t,II_t,h,f,disturbance_height,plot_height,c_map='Blues_r',interval = 100,fname = None)

Creates an animated 3D plot displaying a scaled media interface and the refracted ray intensity image

**Parameters:**

*x,y: numpy arrays*

Numpy arrays containg the x and y coordinates of the rays *before* refraction

*t: numpy array*

A numpy array with the time coordinates for which a refracted image will be displayed

*N: list of integers*

A list containing N_x and N_y used to create the ray grid before refraction

*XX_t,YY_t: lists of numpy meshgrids*

Lists of coordinate meshgrids for the different time evaluations, these can be produced by the evolve function

*II_t: list of numpy meshgrids*

List of meshgrids containing the number of refracted rays within bins on the above coordinate meshgrids for different time evaluations, this can be produced by the evolve function

*h: float*

The height of the second medium

*f: function*

The surface displacement of the second medium

*disturbance_height: float*

The maximum value of the function f for all x,y and t

*plot_height: float*

The factor by which the surface plot is scaled when displayed in the plot (value of 0.25 works well)

*c_map: colormap*

Colormap used to plot the refracted ray intensity map

*interval: integer*

The number of milliseconds between frames in the animation

*fname: string*

Name of file to which the animation will be save. If left as default None argument then a temporary file will be used instead

**Returns:**

Creates a 3D animated plot displaying a scaled media interface and the refracted ray intensity image

### 2.4.4 Ray Object

#### Ray(x,y,f,dfdx,dfdy,h,n_arr) (__init__ function)

**Parameters:**

*x,y: numpy arrays*

Numpy arrays containg the x and y coordinates of the rays before refraction

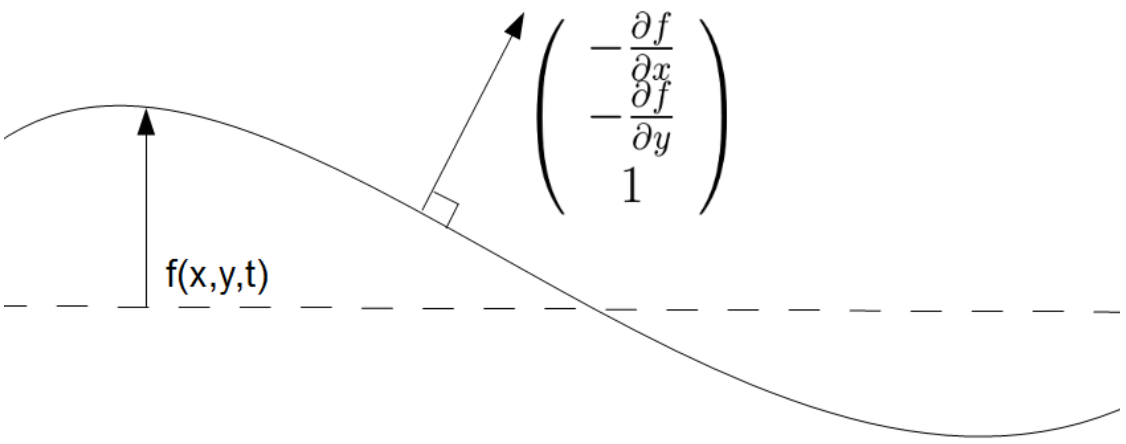
*f, dfdx, dfdy: functions*

f: A function of x, y and t which gives the displacement of the second mediums surface

dfdx: A function of x, y and t which gives the partial differential of f along x

dfdy: A function of x, y and t which gives the partial differential of f along y

These are used to compute the unit normal and the height the ray must descend to reach the bottom of the second medium


f(x,y,t)
$\begin{pmatrix} -\frac{\partial f}{\partial x} \\ -\frac{\partial f}{\partial y} \\ 1 \end{pmatrix}$

*n_arr: list of floats*

A list of two floats, the first being the refractive index of the first medium and the second that of the second medium

*h: height*

The height of the second medium, the incident light rays which refract will be traced until they reach this height

### findnormal(self,t)

Finds the normalised normal vector at position of the incident ray

**Parameters:**

*t: numpy array*

A numpy array with the time coordinates for which a refracted image will be displayed

### snellsangle(self)

Calculates the angle of refraction via Snell's law

### transformation_matrix(self)

Calculates the rotation matrix required to rotate the incident ray onto the refracted ray

### refract(self,t)

Computes the refracted ray positions using the rotation matrix

**Parameters:**

*t: numpy array*

A numpy array with the time coordinates for which a refracted image will be displayed

## 2.5 Display

### 2.5.1 Introduction to the Display module

This module contains functions for use in creating and displaying HTML embedded videos in Jupyter notebooks. Matplotlib animation objects are converted either directly to HTML5 or via a permanent file in the directory containing the notebook.

A notebook demonstrating its use can be found here (or in nbviewer).

You can find the source code for this module on the GitHub.

# CHAPTER 3

# About PyCav

## 3.1 History

There's not much to write here yet, we're just getting started!

## 3.2 License

PyCav source code is licensed under the terms of the clear BSD License: