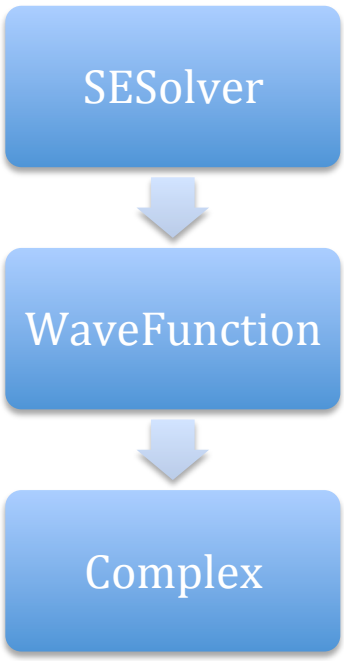
Overview

This program will solve numerically the one-dimensional time-dependent Schrödinger equation, using the Crank-Nicolson finite-difference method to evolve a user-defined arbitrary initial wave function in a user-defined potential.

This document describes the classes written for this program, how they interface to each other, and outlines any algorithms implemented. All code is written in Python.

Class layout:



Complex Class

Each instance represents a complex number. We assume that the number is in the form $a = x + yi$, where a is the *real part* and b is the *imaginary part*.

Properties

Name	Type	Notes
`realPart`	`float`	The real part
`imagPart`	`float`	The imaginary part

Constructor

Arguments	Notes
`float rp, float ip`	Creates `rp` + `ip` *i*

Methods

`__str__()`

Returns a string representing the complex number in the form *a* + *b*i.

normSquared()

Return the square of the norm of the complex number, i.e.

$$|a|^2 = x^2 + y^2$$

`norm()`

Return the norm of the complex number, i.e.

$$|a| = \sqrt{x^2 + y^2}$$

conj()

Return the conjugate of the complex number, i.e.

$$a^* = x - yi$$

Static Methods

`addComplex(Complex a, Complex b)`

Return the sum of two complex numbers, i.e.

$$a + b = (x_a + x_b) + (y_a + y_b)i$$

`subComplex(Complex a, Complex b)`

Return the subtraction of two complex numbers, i.e.

$$a - b = (x_a - x_b) + (y_a - y_b)i$$

`multComplex(Complex a, Complex b)`

Return the product of two complex numbers, i.e.

$$ab = (x_a x_b - y_a y_b) + (x_a y_b + x_b y_a)i$$

`scaleComplex(Complex a, float b)`

Return the product of a complex number and a scalar, i.e.

$$ba = bx + byi$$

`divideComplex(Complex a, Complex b)`

Return the division of two complex numbers, i.e.

$$\frac{a}{b} = \frac{ab^*}{|b|^2}$$

WaveFunction class

Each instance represents a one-dimensional complex wave function of a quantum particle. The wave function is represented on a regularly spaced discretized x-axis by an array of complex numbers, where each entry corresponds to the value of the wave function at a particular point along the x-axis.

Properties

Name	Type	Notes
nPoints	`int`	Number of wave function points along the axis
mass	`float`	The mass of the particle
psi	`np.array(Complex[nPoints])`	The wave function amplitude

Constructor

Arguments	Notes
`float m, int nP, np.array(Complex[]) phi`	Creates wave function for particle with mass m, defined on nP points, with values given by phi.

Methods

`setAmplitude(np.array(Complex[]) amp)`

Set the amplitude of the wave function to the specified array amp.

`setMass(float mass)`

Set the mass of the particle to the specified number mass.

`getGridSize()`

Returns the number of points on which the wave function is described.

`getAmplitude()`

Returns the amplitude of the wave function as Numpy array of Complex numbers.

`square()`

Returns the absolute square of the wave function as Numpy array of floats.

`norm()`

Returns the norm of the wave function, integrated over the entire range of definition.

`kineticEnergy()`

Returns the kinetic energy of the wave function, by third-order real-space representation of the Laplace operator, and integrating over the entire range of definition.

`potentialEnergy(np.array(float[]) V)`
Returns the potential energy of the wave function in the external potential V, integrated over the entire range of definition.

`leapAmplitude(float dt)`
Integrates the wave function amplitude forward by time step dt using the Crank-Nicolson time integrator.

Static Methods

`overlap(WaveFunction psi1, WafeFunction psi2)`
Returns the overlap integral (psi1*)*psi2 across the entire range of definition.

SESolver class

This class contains the main program that simulates the evolution of a one-dimensional wave function packet as function of time in an external potential, with given initial conditions.

This class does not contain any properties, constructors, or instance methods.

Static Methods

`main(str[] argv)`

The main method reads in the names of three input files from the command line:

1. The numerical parameters of the simulation: time discretization dt, spatial discretization dx, length of the system along the x-direction L, total simulation time T.
2. The external potential V(x), defined on regular grid points along the x-axis.
3. The initial wave function psi(t=0), defined on the same grid.
4. The name of the output file.

The program uses periodic boundary conditions, i.e. we identify the points x=0 and x=L.

Before running the simulation, we renormalize the wave function and calculate the initial kinetic and potential energy, using instance methods from the WaveFunction class.

We then solve the equation-of-motion for the wave function, the time-dependent Schrödinger equation,

$$-\frac{\hbar}{i}\frac{\partial}{\partial t}\psi(x,t)=\hat{\mathrm{H}}\psi(x,t)=\left[-\frac{\hbar^2}{2m_e}\frac{\partial^2}{\partial x^2}+V(x)\right]\psi(x,t)$$

In our simulation, we use atomic units, thus setting $\hbar=1, m=m/m_e$, which results in

$$i\frac{\partial}{\partial t}\psi(x,t)=\left[-\frac{1}{2m}\frac{\partial^2}{\partial x^2}+V(x)\right]\psi(x,t)$$

To solve the evolution of the wave function at discrete time steps, we use the Crank-Nicolson time integrator together with a finite difference spatial discretization of the Laplacian:

$$\psi\left(j\Delta x,(n+1)\Delta t\right)\equiv\psi_j^{n+1}=\psi_j^n+\frac{i\Delta t}{2}\left[\frac{1}{2m}\frac{\partial^2}{\partial x^2}\left(\psi_j^n+\psi_j^{n+1}\right)+V_j\left(\psi_j^n+\psi_j^{n+1}\right)\right],$$

$$\frac{\partial^2}{\partial x^2}\psi_j^n=\frac{1}{2\Delta x^2}\left(\psi_{j+1}^n-2\psi_j^n+\psi_{j-1}^n\right)$$

As the right hand side contains values of the wave function at the next time step, we set up a tridiagonal matrix with the coefficients of both ψ_j^n and ψ_j^{n+1} and implement Thomas' algorithm to solve for ψ_j^{n+1} .

In the main loop, we iterate the wave function as function of time in the external potential *V(x)*. At each time step, we calculate the total energy as sum of kinetic and potential energy, as well as the norm of the wave function, and write those values to the output file.

Periodically (every $1/100^{th}$ of the total simulation time) we calculate the overlap of the current wave function with the initial wave function, and we write the overlap to the output file as well, to be visualized later.

`triDiagSolver(WaveFunction phi, np.array(Complex[]) sub, np.array(Complex[]) diag, np.array(Complex[]) super, int N)`
This method uses Thomas' algorithm to solve a tridiagonal system of equations, Ax=b. Here, sub[], diag[], and super[] are the sub-, main-, and superdiagonal of the matrix A, N is the length of the matrix A, and phi contains both the right hand side b[] (at the beginning) and the solution vector x[] (at the end).