Numerical Solutions of the One-Dimensional Time Dependent Schrodinger Equation for Various Potentials

Physics Department, University of California at Berkeley
Oliver Gorton

May 2016

Abstract

A numerical method for solving the time dependent Schrodinger equation is reviewed and adapted for implementation in a program written in Python. This 'Particle in a Box' simulation is useful primarily as an educational tool since it can be used to demonstrate many important phenomena in quantum mechanics. The program described utilizes the matplotlib package called "animation" which is useful for creating animations of numerical simulations in particular. The main functions of the program are described in some detail.

1 Introduction

The purpose of this paper is to discuss the mathematics behind a simple numerical solution to the time dependent Schrodinger equation and to present a practical implementation of that solution in the Python programming language.

What follows is my reproduction of a simple method laid out by P.B. Visscher in his paper "A fast explicit algorithm forthe time-dependent Schrodinger equation" (1991). Visscher's paper is itself what I believe to be one of many reproductions of Abraham Goldberg et al.s original paper "Computer-Generated Motion Pictures of One-Dimensional Quantum-Mechanical Transmission and Reflection Phenomena" which was published in 1967 and which is to the best of my knowledge the first presentaiton of this numerical solution and discussion.

In this reproduction I hope to offer a more versatile and perhaps pedagogically entertaining version of the academic results presented previously. Using the wonder of modern computing I have written my own program to solve the TDSE by following the algorithm laid out by Visscher. In addition to a customizable rectangular barrier I have also included a double rectangular barrier, a ramp-barrier, and two sinusoidal barriers. This list could easily be expanded as any shape that can be stored as a one-dimensional field can be used as a potential.

In order to make the program more usable and pedagogically applicable I have written a Graphical User Interface (GUI) that allows the user to quickly and intuitively modify the wavefunction and potential and to then instantly observe the results. The GUI also has other useful tools: controls to save the animation as a video file, and a scrollbox to change the speed of the animation while it is running without affecting the accuracy of the calculations.

2 Theory

Here I will discuss how the wavefunction is evolved in time according to the TDSE, i.e. the algorithm used for integrating the TDSE and how that translates into code. I will also desribe how the momentum space wavefunction is calculated from the position space wavefunction using Python.

2.1 The wavefunction and the potential

The wavefunction used in this simulation is a Gaussian wavepacket of the form

$$\Psi(x,t) = \frac{1}{\sqrt{a\sqrt{\pi}}} e^{-\frac{(x-x_0)^2}{2a^2} + ixk_0}. \tag{1}$$

where a describes the spread of the packet, x_0 describes its center, and k_0 descibes its initial momentum. The potential that the wavefunction interacts with is of the form

$$V(x) = \begin{cases} f(x) & a \leq x \leq b \\ 0 & elsewhere \end{cases} \tag{2}$$

where $f(x)$ is some function of x that modulates the field within some interval. It is the goal of the following section to develope a method for solving the time dependent Schrodinger equation given an initial $\Psi(x,t)$ and $V(x,t)$.

2.2 Primary alogorithm for integrating the TDSE

Our objectives for this section are to (1) reform the time dependent schrodinger equation into a form that is conveniently translated into a discrete equation of motion, (2) to translate the wavefunction and the equations of motion from their continuous forms to distrete versions, and (3) to devise a method for storing and evolving the

discrete TDSE in computer code according the previous two sections.

This is not original work, but my interpretation of the simple algorithm described by P.B. Visscher, and the more rigorous discussion given by Abraham Godberg et al. All equations in sections 2.2.1 and 2.2.2 can be found in some form or another in one or both of the two papers cited. Section 2.2.3, however, is my own method for converting the mathematical algorithms into computer code.

2.2.1 Reforming the TDSE

We begin with the time dependent Schrodinger equation.

$$i\hbar\frac{\partial}{\partial t}\Psi(\vec{r},t) = -\frac{\hbar^2}{2m}\nabla^2\Psi(\vec{r},t) + V(\vec{r},t)\Psi(\vec{r},t) \tag{3}$$

By specializing to one dimension and setting both $\hbar$ and m equal to 1 for convenience we obtain a simplified equation:

$$i\frac{\partial}{\partial t}\Psi(x,t) = -0.5\frac{\partial^2}{\partial x^2}\Psi(x,t) + V(x,t)\Psi(x,t) \tag{4}$$

$$i\frac{\partial}{\partial t}\Psi(x,t) = H(\Psi) \tag{5}$$

In equation (3) we have defined the right side of (2) to be the Hamiltonian acting on Ψ in the usual sense. We now redefine our wavefunction to be the sum of a real part and an imaginary part. By substituting this back into (2) we can seperate the TDSE into its real and imaginary parts, yielding two coupled differential equations.

$$\Psi(x,t) = R(x,t) + I(x,t)i \tag{6}$$

$$\begin{aligned} i\frac{\partial}{\partial t}[R(x,t) + I(x,t)i] &= H[R(x,t) + iI(x,t)] \\ i\frac{\partial}{\partial t}R(x,t) - \frac{\partial}{\partial t}I(x,t) &= HR(x,t) + iHI(x,t) \\ \frac{d}{dt}R(x,t) &= HI(x,t) \qquad (7) \\ \frac{d}{dt}I(x,t) &= -HR(x,t) \qquad (8) \end{aligned}$$

Equations (5) and (6) mark the end of the analytic manipulations. These two coupled differential equations completely specify the time evolution of out system. Next, we seek a discrete version that can easily be translated into a numerical solution.

2.2.2 The discrete wavefunction and equations of motion

In order to represent the continuous wavefunciton using the discrete objects found in the simulation will must discretize both space and time and redefine our wavefunction and potential over these descrete domains:

$$\begin{aligned} \Psi(x,t) &\rightarrow \Psi_t^x \\ V(x,t) &\rightarrow V_t^x \end{aligned} \tag{9}$$

Where the indices indicate that we are now dealing with arrays wherein each element of an array represents the value of that field at position x and time t. We will be representing our wavefunction Ψ using two one-dimensional arrays R and I:

$$\Psi_t^x = R_t^x + iI_t^x \tag{10}$$

A standard trick in trajectory calculations is to define our two coupled quantities at staggered times (Visscher). Time evolution of the system is therefore naturally described by the following equations:

$$\begin{aligned} R_{t+\Delta t/2} &= R_{t-\Delta t/2} + \Delta R \\ I_{t+\Delta t/2} &= I_{t-\Delta t/2} + \Delta I \end{aligned} \tag{11}$$

Where the spacial index has been suppressed. In the limit that Δt approaches zero, we equations (5) and (6) tell us that

$$\begin{aligned} R_{t+\Delta t/2} &= R_{t-\Delta t/2} + \Delta t H(I_t) \\ I_{t+\Delta t/2} &= I_{t-\Delta t/2} - \Delta t H(R_t) \end{aligned} \tag{12}$$

Where in the same limit our Hamiltonian operator is translated into the difference equation as:

$$\begin{aligned} H(\Psi(x,t)) = -0.5\frac{\partial^2}{\partial x^2}\Psi(x,t) + V(x,t)\Psi(x,t) \rightarrow \\ H(\Psi_t^x) = -0.5\left(\frac{\Psi_t^{x-\Delta x} - 2\Psi_t^x + \Psi_t^{x+\Delta x}}{\Delta x^2}\right) + V_t^x\Psi_t^x \end{aligned} \tag{13}$$

Equation (11) is general in the sense that it is valid for any discrete object Ψ of the same form. Notice that is the potential V is time-independent that the Hamiltonian is also time-independent.

2.2.3 Converting to computer code

In the program the fundamental objects that we will be dealing with are arrays. Equation (7) tells us how to represent our system as two-dimensional arrays. However, to simply things further we are going to reduce our simulation to a program that deals only with one-dimensional arrays. This will greatly reduce the total ammount of memory that is require to run the program, and yet it does not subtract at all form its effectiveness. Since the goal of the program is to display an animation of the wavefuntion interacting with the potential, there is no sense in keeping the wavefunction for a given time around after is has been displayed. In other words, we only need to keep track of the current-time values of Ψ. Let me explain how this is done by defining the variables that we will be keeping track of in pseudocode:

```
R_prev = array(...) #previous value of R
R      = array(...) #current value of R
R_next = array(...) #next value of R
```

```
I_prev = array(...) #previous value of I
I      = array(...) #current value of I
I_next = array(...) #next value of I

V = array(...)     #current value of potential
```

Comparing these variables with those in section 2.1.2 (equations (10) and (11)), its easy to understand the following lines of psuedocode which would calculate the next values of the wavefunction after one timestep.

```
R_next = R_prev + dt * H(I)
I_next = I_prev - dt * H(R)

Where

H(I)[x] = -0.5 * (I[x+1] - 2 * I[x] + I[x-1])
                / dx^2 + V[x] * I[x]
H(R)[x] = -0.5 * (R[x+1] - 2 * R[x] + R[x-1])
                / dx^2 + V[x] * I[x]
```

The next steps that would occur in the code would then "shift" all of the values in order to update what the 'current' values of R and I are. In index notation this means:

$$\begin{aligned} R^x_t &\mapsto R^x_{t-\Delta t/2} \\ R^x_{t+\Delta t/2} &\mapsto R^x_t \end{aligned} \qquad (14)$$

$$\begin{aligned} I^x_t &\mapsto I^x_{t-\Delta t/2} \\ I^x_{t+\Delta t/2} &\mapsto I^x_t \end{aligned}$$

After each timestep calculation and memory shift, the probability density function is calculated and plotted, and then the process repeats. This concludes our discussion of the primary algorithm used in the program.

2.3 Choosing appropriate parameters

Both Visscher and Goldberd et al. give their own discussion of the accuracy of this calculation, and limits on the values of the input parameters. Of particular interest for practical application is the choice of Δt, Δx, the particle's initial momentum, and the magnitude of the potential. Here I will simply quote the relavent results.

Visscher finds a condition on the potential, Δt and Δx such that the solution is stabile. That condition is

$$\frac{-2}{\Delta t} < V < \frac{2}{\Delta t} - \frac{2}{\Delta x^2}. \qquad (15)$$

To implement this condition we select an arbitrary maximum and minumum value for the potential (unity, for example) and find a limiting relation on Δx and Δt. Absolute limits on the two steps are inuitively understood as the fact that our approximation to the Hamiltonian is only valid for $\Delta x << \Delta\Psi(\Delta t)$. Because of the staggered-time method used it is also important that the initial momentum be small compared to Δx. This is reflected in the simulation by small anomalous oscillations on top of the wave packet which appear for large particle momentum values.

Goldberg et al. arrive at six conditions for valid scattering events, two of which I will quote to explain these observations:

$$V << \frac{12}{\Delta x^2} \qquad (16)$$

$$k << \frac{\pi}{\Delta x} \qquad (17)$$

2.4 Momentum Space Representation

In order to improve the interpretability of the simulation by users, a momentum space representation of the wavefunction is included in the graphical output. The momentum space of the wavefunction is simply the Fourier ransform of the position space wavefunction, but we are interested in calculating the continuous fourier transform, rather than the discrete fourier transform that is available through packages like the Python Numpy Fast Fourier Transform (numpy.fft). Fortunately there is a method for converting between the two.

To go from $\Psi(x)$ to $\tilde{\Psi}(k)$ we have to compute the Fourier transform integral

$$\tilde{\Psi}(k) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{+\infty} \exp(-ikx)\Psi(x)dx \qquad (18)$$

In order to use a discrete Fourier transform package to compute this, we have to express (13) in terms of the Fast Fourier Transform:

$$FFT(A) = \sum_{m=1}^{N} A \exp(-2\pi imn/N) \qquad (19)$$

To do this we first discretize our position and momentum spaces over some appropriate finite range.

$$\begin{aligned} k_n &= k_o + n\Delta k = k_o + n\frac{2\pi}{N\Delta x} \\ x_m &= x_o + m\Delta x \end{aligned} \qquad (20)$$

$$\tilde{\Psi}(k) \approx \frac{1}{\sqrt{2\pi}} \sum_{m=1}^{N} \Delta x \Psi(x_m) \exp(-ik_n x_m) \qquad (21)$$

Substituting (15) into (16) and some algebraic manipulation yields

$$\begin{aligned} \tilde{\Psi}(k) \approx & \frac{\Delta x}{\sqrt{2\pi}} \exp(-inx_o\Delta k) * \\ & \sum_{m=1}^{N} \Big(\Psi(x_m)\exp(-ik_o x_m)\Big) \exp(-2\pi imn/N) \end{aligned} \qquad (22)$$

Which, observing the form of equation (14), is equivilent to

$$\tilde{\Psi}(k) \approx \frac{\Delta x}{\sqrt{2\pi}} \exp(-inx_o\Delta k) FFT\Big(\Psi(x_m)\exp(-ik_o x_m)\Big) \qquad (23)$$

Where the FFT() function is exactly the function available to us in the numpy.fft package. A psuedocode implementation of this relation is given here.

```
A = Psi * exp(-i * ko * x_m)
g = FFT(A)
FT(Psi) = g * (dx / sqrt(2pi)) *
                exp(-i * dk * x0)
```

3 The Simulation

3.1 Python implementation

The program discussed in this paper is written in the computer programming language Python. The code can be seen as consisting of two parts, the simulation and the graphical user interface (GUI). The simulation part itself consists of three major components: A class called Psi(), a class called Potential(), and a series of functions that interact with the matplotlib.animation package to iteratively evolve the wavefunction and update a plot of both the position space representation and the momentum space representation of the wavefunction.

3.2 matplotlib.animation

This simulation relies on the Python package called matplotlib.animation. This function works by calling on a function defined by its user to update a matplotlib plot psuedo-continuously at high speed. In this simulation, the function which is called is 'animate()', a custom function which carries out a number of operations. One of these operations is to call the another function named TimeStep() which calls the methods of the class Psi() in the correct order to advance the wavefunction one timestep.

3.3 Graphical User Interface

The GUI for this program was written using Python's default GUI package Tkinter. In this section I will briefly mention each of the objects in the GUI, what it does, and how it works if it is interesting.

3.3.1 Initialization settings

Text entry fields: There are five text entry fields in the Initialization section of the GUI. Each one allows the unit to input a numerical value for that variable. The variables that can be set are: the left and right edges of the potential (outside of this range the potential is zero), the depth of the potential, the momentum of the particle, and the initial position of the particle.

Dropdown menu for presets: This button opens a menu of selectable options. Each of the options enters values into the text entry fields for initializing the particle and potential. The options offer the user different behaviour of the particle based on how much of the particle is expected to be transmitted through the barrier if a rectangular barrier is selected.

Modulation options: These are so called 'radiobuttons' which are selectable setting for the user to choose a potential barrier modulation. Each of these options starts with a rectangular barrier described by the text entry fields and multiplies is by some modulation e.g. a sine wave or linear function.

3.3.2 Simulation operators

Set initial conditions: There are three buttons and one numerical scrollbox for controlling the simulation. The first button labeled 'Set initial conditions' runs a function that gets the values entered into the text fields and assigns the relavant variables to those values. The function then creates instances of the Psi() class and the Potential() class based on the selected parameters. The same function also enables the next button labeled 'Start simulation'.

Start Simulation: The 'Start Simulation button runs a function that calls the matplotlib.animation function and simultaneously disables the 'Start Simulation' button as running matplotlib.animation a second time causes errors. The function activated by 'Start Simulation' contains a while loop that only run the animation function while a variable called 'go' is True.

Stop Simulation: The third button labeled 'Stop Simulation' changes 'go' to False and then changes another variable called 'history' from False to True. This action causes the animation function to stop and disables the ability of 'Set initial conditions' to enable 'Start Simulation'. By stopping the animation in this way, the user can change any of the initial conditions (or change nothing) and then choose to select 'Set initial conditions' which will restart the simulation with the updated initial conditions. This structure was designed to prevent the code from crashing while allowing for an intuitive user interaction.

Animation speed multiplier: This is possible the most useful control in the GUI. Included in the animate() function is the TimeStep() function which advances the wavefunction. However, a for-loop within animate() calls TimeStep() a finite number of times. The 'Animation speed multiplier' sets the variable which determines the numer of times this for-loop runs before the next frame of the animation is set. This means that the rate at which the animation plays can be varied at any time without affecting the accuracy of the calculation!

3.3.3 Video Generator

This section of the user interface allows the user to save the animation as a video file (.mp4 type) using the method included in the matplotlib.animation package called 'save'. Just as before, there are several text-entry fields that allow the user to define the number of

frames, the interval of time between frames, the frames per second, and name of the file to be created.

4 Future projects

4.1 Time dependent potential

The potential is handled with a class that enables the potential field array to be fully customizable. The simplicity of the animation function means that the potential could be made to be time dependent simply by adding lines of code to the TimeStep() function to change the value of the potential array after each timestep. The most natural time dependent potentials that one might investigate would be a potential box that disappears after some time, or a potential wall that moves slowly. These could be used to investigate time dependent perturbation theory. Creating a time dependent potential means that our Hamiltonian would not longer be conservative, however, and great care would have to be taken when choosing initial conditions such that the energy added or removed from the system did not cause the stability conditions discussed above to be violated.

4.2 Higher dimensions

This would require significant developement and generalization of the current results, however it is possible and perfectly reasonable to imagine recreating this project in two or three dimensions. This could be accomplished by increasing the dimension of both the wavefunction array and the potential field array to two or three dimensions, and by then implementing the three-dimensional difference equation for the laplacian in the Hamiltonian.

5 References

Goldberg, A., *Computer-Generated Motion Pictures of One-Dimensional Quantum-Mechanical Transmission and Reflection Phenomena.* AJP Vol. 35, No. 3, March 1967.

Visscher, P.B., *A fast explicit algorithm for the time-dependent Schrodinger equation.* Comput. Phys. 5, 596 (1991).