PHY 121 Spring 2017 Honors Contract

Joshua Smith
Student

Dr. Ross Tucker
Instructor

Abstract

The purpose of this honors project was for the student to learn extra material not normally covered in PHY 121. This enriched the student's introduction to university-level physics by including topics beyond the main focus of the course.

For this project, the student studied the Infinite Square Well, which served as a basic introduction to quantum physics. The student wrote code that produces animated plots modeling the behavior of a particle in the Infinite Square Well and wrote this report in order to better understand this topic.

1 The Infinite Square Well

1.1 The Schrödinger Equation

The infinite square well is one of the simplest solutions to the Schrödinger equation. The Schrödinger equation in one dimension is given by

$$\frac{-\hbar^2}{2m}\frac{\partial^2}{\partial x^2}\Psi(x,t) + V(x)\Psi(x,t) = i\hbar\frac{\partial}{\partial t}\Psi(x,t)$$

Where $\hbar$ (pronounced "h-bar") is the reduced Planck constant, $\Psi(x,t)$ is related to the probability density of the particle with respect to both position and time (more on this in section 1.3), $V(x)$ describes the potential energy of the particle as a function of position, and i is the imaginary unit.

1.2 Solving using Separation of Variables and the Linking Constant

Solving the Schrödinger equation is possible using separation of variables, beginning with the assumption that $\Psi(x,t)$ can be written in the form $\Psi(x,t) = \psi(x)f(t)$. The Schrödinger equation can now take on this form:

$$\frac{-\hbar^2}{2m}\frac{\partial^2}{\partial x^2}\psi(x)f(t) + V(x)\psi(x)f(t) = i\hbar\frac{\partial}{\partial t}\psi(x)f(t)$$

which can also be expressed as

$$f(t)\left(\frac{-\hbar^2}{2m}\frac{\partial^2}{\partial x^2}\psi(x) + V(x)\psi(x)\right) = \psi(x)i\hbar\frac{\partial}{\partial t}f(t)$$

by using the fact that $f(t)$ is not a function of position and $\psi(x)$ is not a function of time. Dividing both sides of the equation by Ψ then yields

$$\frac{\frac{-\hbar^2}{2m}\frac{\partial^2}{\partial x^2}\psi(x)}{\psi(x)} + V(x) = \frac{i\hbar\frac{\partial}{\partial t}f(t)}{f(t)}$$

Since the right side of this equation depends only on t and the left side depends only on x, these expressions must be equal to a constant E called the *linking constant*:

$$\frac{\frac{-\hbar^2}{2m}\frac{\partial^2}{\partial x^2}\psi(x)}{\psi(x)} + V(x) = \frac{i\hbar\frac{\partial}{\partial t}f(t)}{f(t)} = E$$

The linking constant can be used to write both a time-dependent (i.e. position independent) and a position-dependent (i.e. time independent) equation:

$$E\psi(x) = \frac{-\hbar^2}{2m}\frac{\partial^2}{\partial x^2}\psi(x) + V(x)\psi(x)$$

$$Ef(t) = i\hbar\frac{\partial}{\partial t}f(t)$$

The time-dependent Schrödinger equation is a separable ordinary differential equation, making it easy to solve. The final solution is

$$f(t) = Ce^{\frac{E}{i\hbar}t}$$

where C is some constant given by initial conditions. Before solving the position dependent Schrödinger equation, the potential energy of the particle $V(x)$ must be provided. For a one-dimensional infinite square well of length L, $V(x)$ is imposed to be

$$V(x) = \begin{cases} 0 & 0 < x < L \\ \infty & otherwise \end{cases}$$

Since $V(x)$ is a piecewise function, the position dependent Schrödinger equation is solved in a piecewise manner. First consider the portions of the equation outside of the well. Since the potential energy is infinite outside of the well, the particle would need to acquire infinite energy in order to reach these areas. Therefore, here $\psi(x) = 0$. Next consider the portion inside of the well. Since $V(x)$ is zero in these areas, the equation can be expressed as

$$E\psi(x) = \frac{-\hbar^2}{2m}\frac{\partial^2}{\partial x^2}\psi(x).$$

The standard form of such a differential equation is

$$\frac{\partial^2}{\partial x^2}\psi(x) = \frac{-2Em}{\hbar^2}\psi(x)$$

and has the general solution

$$\psi(x) = A\cos\left(\sqrt{\frac{2mE}{\hbar^2}}x\right) + B\sin\left(\sqrt{\frac{2mE}{\hbar^2}}x\right)$$

As a consequence of the infinite square well, $\psi(0) = \psi(L) = 0$. This gives the initial conditions necessary to determine the coefficient A. Substituting $\psi(0) = 0$ in the equation yields the result that $A = 0$. Substituting $\psi(L) = 0$ can yield a new expression for the argument of the sine function as follows:

$$0 = B\sin\left(\sqrt{\frac{2mE}{\hbar^2}}L\right)$$

This has the trivial solution $B = 0$. Solving for the non-trivial solution yields

$$0 = \sin\left(\sqrt{\frac{2mE}{\hbar^2}}L\right) \implies n\pi = \sqrt{\frac{2mE}{\hbar^2}}L \implies E = \frac{n^2\pi^2\hbar^2}{2mL^2}.$$

Since this equation implies that there is a unique E for every integer n, this is written as

$$E_n = \frac{n^2\pi^2\hbar^2}{2mL^2}$$

Substituting this expression for E_n into $\sqrt{\frac{2mE}{\hbar^2}}x$ yields $\frac{n\pi x}{L}$, resulting in the simpler equation

$$\psi(x) = B\sin\left(\frac{n\pi x}{L}\right).$$

Having obtained the solutions for both the time-dependent and the position-dependent Schrödinger equations, the solution to the full Schrödinger equation can be written using the initial assumption that $\Psi(x,t) = \psi(x)f(t)$. Thus, the full solution is

$$\Psi(x,t) = Be^{\frac{E_n}{i\hbar}t}\sin\left(\frac{n\pi}{L}x\right)$$

The constants C and B found in the individual solutions can be written as a single unknown constant B, since a constant multiplied by a constant yields another constant. Since there is a unique term for each n, there is possibly a unique constant for each value of n. Thus, the equation is rewritten as

$$\Psi(x,t) = B_n e^{\frac{E_n}{i\hbar}t}\sin\left(\frac{n\pi}{L}x\right).$$

Additionally, according to the principle of superposition, the general solution to a differential equation is the sum of all linearly independent solutions to the differential equation. Since each value of n yields such a linearly independent solution, the solution is written as the infinite series

$$\Psi(x,t) = \sum_{n=1}^{\infty} = B_n e^{\frac{E_n}{i\hbar}t}\sin\left(\frac{n\pi}{L}x\right).$$

Since sine is odd (that is, $\sin(-a) = \sin(a)$), it is not necessary to include any $n < 0$ in the series. Since $\sin(0) = 0$, it is not necessary to include $n = 0$ in the series.

1.3 Calculating Probability Density and B_n

The probability density can be calculated by

$$\Psi^*\Psi$$

Where Ψ^* is the complex conjugate of Ψ. Since the particle exists only within the square well,

$$\int_0^L \Psi^*\Psi dx = 1.$$

This relationship can be used to calculate B_n. Substituting a simplified, one-term solution to the equation into the integral shown above gives

$$\int_0^L B^* B e^{\frac{E_n}{i\hbar}t} e^{\frac{E_n}{-i\hbar}t} \sin^2\left(\frac{n\pi x}{L}\right) dx = 1,$$

which is equivalent to

$$\int_0^L B^* B \sin^2\left(\frac{n\pi x}{L}\right) dx = 1,$$

because

$$e^{\frac{E_n}{i\hbar}t} e^{\frac{E_n}{-i\hbar}t} = 1.$$

Solving this expression for B, assuming that B is real, gives

$$B = \sqrt{\frac{2}{L}}$$

It can be shown that the expression for B when substituting the full series solution into the above integral yields the expression

$$\sum_{n=1}^{\infty} B_n^2 = \frac{2}{L}.$$

This means that the set B is a set of normalizing coefficients which assures that the total probability does not exceed 1. The set of B can be calculated from initial conditions using Fourier methods.

2 The Code

Here an overview of how the code works and the development process is given. All code can be found at `github.com/jdsmit60/fourier-visualizer`. Some selected code is included in the

appendices.

2.1 Generating the Data

In order to generate the plot data, C++ was chosen for its speed, as this is most computationally intensive part of the plot generation. The input of the program was to be a list of coefficients, specifically a space or newline-delimited file containing the first 20 B_n. The output was to be a data file for each time step of the plot. Each time step corresponds to one frame of the final video, and each data file contains all of the probability density versus position data for each time step. An additional data file, `range.dat`, contains the largest and smallest values that should be on the probability density axis. This range is calculated based on the largest and smallest probability densities which occur during the time interval for which the calculation is run.

The main reason that this portion of the plot generation is so computationally intensive is the nested `for` loops found in lines 78-111. The outermost loop manages splitting calculations between each time step. The middle loop manages calculating the probability density at each desired value of position. Finally, the innermost loop handles each term of the series in the solution through the use of the accumulator variable `Psi`. This high computational cost necessitated a language that would provide very high performance, hence the use of C++.

2.2 Generating the Plots

The two graphing utilities considered for creating the actual image files of the plots at each time step were gnuplot and matplotlib. Gnuplot is a standalone utility written in C which includes its own scripting language and a plethora of options. Matplotlib is a Python package which works extremely well with numpy arrays and is known for its simplicity of use.

Ultimately, gnuplot was chosen for generating the plots because it produced plots of identical quality at higher resolutions in much less time. Gnuplot has a significant speed advantage because it is written in C, a compiled language, while matplotlib is written in Python, an interpreted language. Gnuplot also worked much more nicely with the format of the output data generated, and could read this data directly from the data file, whereas when using matplotlib the data needed to first be read from the file into an array, which uses more time and is more complex to program.

2.3 Animating the Plots

Two main utilities were used to animate the plots. ffmpeg was used to create video files and imagemagick was used to create animated gifs. The advantages of ffmpeg are that it creates a video with proper compression, resulting in a smaller file. This makes the file quicker to load and play in most browsers than an equivalent gif. Generating gifs using imagemagick comes with the portability benefits of a gif (gifs will work almost anywhere), but gifs are more CPU-intensive when playing the animation and result in larger files, causing a gif to often perform worse. Gifs also come

with the advantage that they can be set to loop automatically, making them perfect for a function that changes periodically like the solution to the Schrödinger equation.

2.4 Tying Everything Together

The final step was to automate the three steps outlined above: generating the data, generating the plots, and animating the plots. For this purpose, two scripting languages were considered: Python and Bash (Linux shell scripting). The implementation in Bash was much less complex, as Bash essentially issues commands to the operating system directly, which allows it to read files into arrays and launch applications much more quickly. However, Bash only works on operating systems that use Bourne-compatible shells, making support on Linux excellent, support on Mac OS mostly good, and support on Windows impossible without extra tweaking. Here, Python shows its main strength: portability. Python is supported on many, many platforms and can perform many of the functions that Bash can using widely available packages.

Python's `subprocess.Popen()` command also led to an accidental optimization of the performance of the Python script. `subprocess.Popen()` does not wait for applications to exit before moving to the next part of the script, resulting in the parallelization of generating the plots using gnuplot. Before this change, the equivalent Bash script was much faster than the Python script, but this made the Python script faster by several times. Similar results may be acheivable using the `nohup` command in Bash, but this was never tested.

3 Using the Program

Here the use of the program will be explained.

Appendices

A PsiStarPsiSI.cpp

```
#include <fstream>
#include <iostream>
#include <iomanip>
#include <complex>

using namespace std;
const string outdir="./datafiles"; // use a separate directory for the data files to keep things clean

int main(int argc, char *argv[]) {
        // get or define all of the mathematical constants

        // get the coefficients from the file
        // the file should contain a space-separated or newline-separated list of the first 20 coefficients in the series

        if(argc != 2) {
                cout << "An input file is required. Exiting.\n";
                return 1;
        }

        ifstream inputFile;
        inputFile.open(argv[1]);

        if(!(inputFile.is_open())) {
                cout << "There was a problem opening the file. Exiting.\n";
                return 2;
        }

        double coefficient[20]; // assume that 20 coefficients will be used
```

```
29
30          cout << "Coefficients:\n";
31
32          for(int index = 0; index < 20; index++) {
33                  inputFile >> coefficient[index];
34                  cout << coefficient[index] << endl; // DEBUG
35          }
36
37          inputFile.close();
38
39          // define other mathematical numbers of importance
40
41          double hbar = 1.055e-34; // h bar (Planck's constant divided by 2*pi) in J*s
42          double mass = 9.11e-31; // mass of particle in kg    9.11e-31 is the mass of
            ↪  an electron
43          double L = 1.0e-10; // maximum position domain of Psi in meters (assume the
            ↪  domain starts at 0)
44          double domainStep = L/100.0; // distance between points that will be
            ↪  calculated
45          complex<double> i(0,1); // i = sqrt(-1)
46          complex<double> Psi(0.0,0.0); // the intermediate result of the calculation
47          double E_1 = (1.0/(2*mass))*pow(M_PI*hbar/L, 2); // ground-state energy
48          double PsiSquared; // this will be the final result of the calculation
49
50          // EVENTUALLY the coefficients will be normalized so that range is between 0
            ↪  and sqrt(2),
51          // but this does not appear to be the case right now so the range calculation
            ↪  remains.
52          double maxPsiSquared = 0.0; // minPsiSquared is fixed at 0 so a variable is
            ↪  unneeded
53
54          double timeStep = 3.3e-20; // let the time step be one tenth of the period of
            ↪  the most quickly oscillating term (term 20)
55          double finalTime = 600.0*timeStep; // let the final time be the period of the
            ↪  least quickly oscillating term (term 1)
56          int timeSteps = static_cast<int>(finalTime/timeStep);
57
58          int digits = log10(timeSteps) + 1; // the number of digits to be used in the
            ↪  file name (small numbers are padded with leading zeros)
59
60          // DEBUG
61
62          cout << "\n\nOther Numbers:" << endl
63               << "hbar = " << hbar << endl
64               << "mass = " << mass << endl
65               << "L = " << L << endl
66               << "domainStep = " << domainStep << endl
67               << "timeStep = " << timeStep << endl
```

```
68              << "finalTime = " << finalTime << endl
69              << "timeSteps = " << timeSteps << endl
70              << "Psi = " << Psi << endl
71              << "i = " << i << endl;
72
73      // get the output files ready
74
75      ofstream dataFiles[timeSteps];
76      stringstream filename; // using streams allows for the easy manipulation of
        ↪  the output file name
77
78      for(int it = 0; it < timeSteps; it++) { // it is an index variable, not the
        ↪  actual variable t used in the equation
79
80              double t = it*timeStep; // keep the equation clean by pre-computing
                ↪  the t that needs to be used in the equation
81
82              // get the output file for this time step ready
83
84              filename.str(string()); // reset the string in the filename
                ↪  stringstream to empty
85              filename << outdir << "/time" << setw(digits) << setfill('0') << it
                ↪  << ".dat";
86              dataFiles[it].open(filename.str(), ios::out | ios::trunc);
87
88              if(!(dataFiles[it].is_open())) {
89                      cout << "Problem opening/creating data file. Exiting.\n";
90                      return 3;
91              }
92
93                      for(double x = 0.0; x <= L; x += domainStep) {
94
95                              Psi = (0.0,0.0); // reset the value of Psi so that
                                ↪  old values don't hang around
96
97                              for(int n = 1; n <= 20; n++) {
98                                      Psi += coefficient[n-1]*sin(n*M_PI*x/L)*exp(i
                                        ↪  *(n*n*E_1/hbar)*t);
99                              }
100                             PsiSquared = norm(Psi);
101
102                             if(PsiSquared > maxPsiSquared) {
103                                     maxPsiSquared = PsiSquared;
104                             }
105
106                             dataFiles[it] << x << " " << PsiSquared << endl;
107                     }
108
```

```
109                 dataFiles[it].close();
110
111         }
112         ofstream range; // EVENTUALLY the coefficients will be normalized so that
            ↪  range is between 0 and sqrt(2),
113                         // but this does not appear to be the case right now so the
                            ↪  range calculation remains.
114         range.open(outdir + "/range.dat", ios::out | ios::trunc);
115
116         if(!(range.is_open())) {
117                 cout << "Problem opening/creating range file. Exiting.\n";
118                 return 4;
119         }
120
121         if(fabs(maxPsiSquared) < domainStep) {
122                 range << "0:" << 1.25*maxPsiSquared;
123         }
124         else {
125                 range << "0:" << maxPsiSquared;
126         }
127         range.close();
128
129 }
```

B gnuplotter.py

```
1  #!/bin/env python
2
3  import re
4  import os
5  import subprocess
6  import math
7
8  datFiles = []
9
10 for filename in os.listdir( "./datafiles" ):
11         potentialMatch = re.match("time[0-9]*.dat",filename)
12         if(potentialMatch):
13                 datFiles.append(potentialMatch.group())
14
15
16
17 datFiles.sort()
18
19 with open("./datafiles/range.dat","r") as rangefile:
20         functionRange = rangefile.read()
21
```

```
22  digits = int(math.log10(len(datFiles)) + 1)
23
24  index = 0
25  for datFile in datFiles:
26          command = "gnuplot -e \"set term pngcairo size 1280,960 truecolor enhanced;
            ↪  set output\'./imagefiles/gnuplot" + str(index).zfill(digits) + ".png\';
            ↪  set style line 1 lt 1 lw 2 linecolor rgb \'#0099ff\'; set yrange [" +
            ↪  functionRange + "]; set xlabel \'Position (meters)\'; set ylabel
            ↪  \'Probability Density (probability/meter)\'; plot \'./datafiles/" +
            ↪  datFile + "\' using 1:2 with lines smooth unique ls 1\""
27          subprocess.Popen(command, shell=True, stdin=None, stdout=None, stderr=None)
28          index += 1
29
30  # -pix_fmt yuv420p is essential here - otherwise videos will not play in browsers
31  # use os.system here so that ffmpeg exits properly - subprocess.Popen was causing
    ↪  issues
32  os.system("ffmpeg -i ./imagefiles/gnuplot%" + str(digits) + "d.png -vcodec h264 -r 25
    ↪  -crf 18 -pix_fmt yuv420p -y graph.mp4 -nostdin")
```