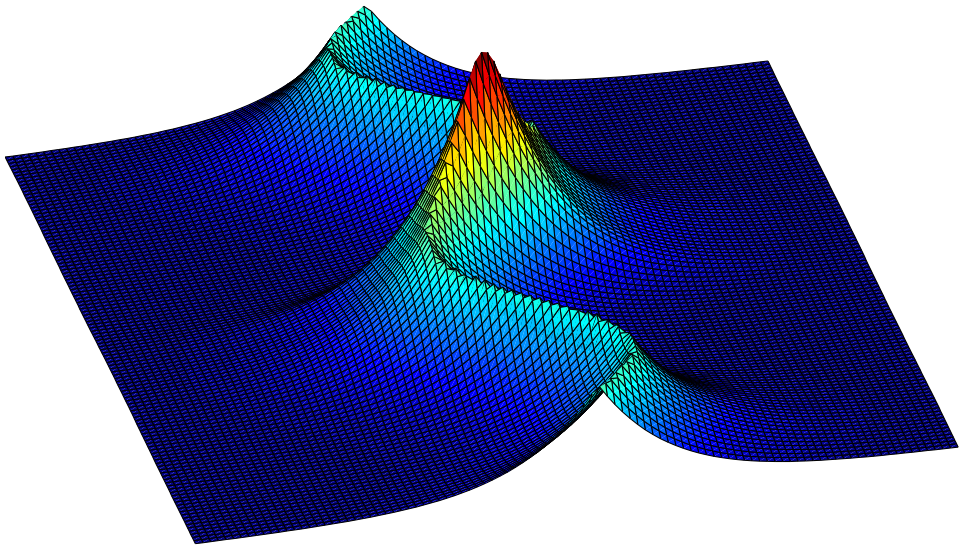
PH150: Introduction to experimental and numerical techniques in Physics



Lance J. Nelson

Department of Physics

PH150: Introduction to experimental and numerical techniques in Physics

Lance J. Nelson

Department of Physics

Brigham Young University–Idaho

Last Revised: June 2, 2017

Preface

This is a lab notebook intended to give you experience with uncertainty analysis, experimental methods, and numerical methods in physics. All text with a bold P designation (**P1.1** for example) indicates a task to be done. Usually this will involve writing something in your lab notebook. Please be as complete and neat as possible.

Any tasks that requires a computer will be done using the Python programming language. You can obtain a free copy of Python here. Any computer code that you create should be uploaded to the Google Drive folder provided.

There is a companion book to this one entitled, "Introduction to Python". It is intended to help you learn to use Python to do the tasks contained herein.

Contents

Review

If you are like most students, loops and logic gave you trouble in 330. We will be using these programming tools extensively this semester, so you may want to review and brush up your skills a bit. Here are some optional problems designed to help you remember your loops and logic skills. You will probably need to use online help (and you can ask a TA to explain things in class too).

(a) Write a `for` loop that counts by threes starting at 2 and ending at 101. Along the way, every time you encounter a multiple of 5 print a line that looks like this (in the printed line below it encountered the number 20.)

```
fiver: 20
```

You will need to use the commands `for`, `mod`, and `fprintf`, so first look them up in online help.

(b) Write a loop that sums the integers from 1 to N, where N is an integer value that the program receives via the `input` command. Verify by numerical experimentation that the formula

$$\sum_{n=1}^{N} n = \frac{N(N+1)}{2}$$

is correct

(c) For various values of x perform the sum

$$\sum_{n=1}^{1000} nx^n$$

with a `for` loop and verify by numerical experimentation that it only converges for $|x| < 1$ and that when it does converge, it converges to $x/(1-x)^2$.

(d) Redo (c) using a `while` loop (look it up in online help.) Make your own counter for n by using $n = 0$ outside the loop and $n = n + 1$ inside the loop. Have the loop execute until the current term in the sum, nx^n has dropped below 10^{-8}. Verify that this way of doing it agrees with what you found in (c).

(e) Verify by numerical experimentation with a `while` loop that

$$\sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{\pi^2}{6}$$

Set the `while` loop to quit when the next term added to the sum is below 10^{-6}.

(f) Verify, by numerically experimenting with a `for` loop that uses the `break` command (see online help) to jump out of the loop at the appropriate time, that the following infinite-product relation is true:

$$\prod_{n=1}^{\infty} \left(1 + \frac{1}{n^2}\right) = \frac{\sinh \pi}{\pi}$$

(g) Use a `while` loop to verify that the following three iteration processes converge. (Note that this kind of iteration is often called successive substitution.) Execute the loops until convergence at the 10^{-8} level is achieved.

$$x_{n+1} = e^{-x_n} \quad ; \quad x_{n+1} = \cos x_n \quad ; \quad x_{n+1} = \sin 2x_n$$

Note: iteration loops are easy to write. Just give x an initial value and then inside the loop replace x by the formula on the right-hand side of each of the equations above. To watch the process converge you will need to call the new value of x something like `xnew` so you can compare it to the previous x.

Finally, try iteration again on this problem:

$$x_{n+1} = \sin 3x_n$$

Convince yourself that this process isn't converging to anything. We will see in Lab **??** what makes the difference between an iteration process that converges and one that doesn't.

Chapter 1

Grids and Numerical Derivatives

When we solved differential equations in Physics 295 we were usually moving something forward in time, so you may have the impression that differential equations always "flow." This is not true. If we solve a spatial differential equation, for instance, like the one that gives the shape of a chain draped between two posts, the solution just sits in space; nothing flows. Instead, we choose a small spatial step size (think of each individual link in the chain) and we seek to find the correct shape by somehow finding the height of the chain at each link.

In this course we will be solving partial differential equations, which usually means that the desired solution is a function of both space x, which just sits, and time t, which flows. And when we solve problems like this we will be using *spatial grids*, to represent the x-part that doesn't flow. You have already used grids in Python to do simple jobs like plotting functions and doing integrals numerically. Before we proceed to solving partial differential equations, let's spend some time getting comfortable working with spatial grids.

Spatial grids

Figure 1.1 shows a graphical representation of three types of spatial grids for the region $0 \le x \le L$. We divide this region into spatial *cells* (the spaces between vertical lines) and functions are evaluated at N discrete *grid points* (the dots). In a *cell-edge* grid, the grid points are located at the edge of the cell. In a *cell-center* grid, the points are located in the middle of the cell. Another useful grid is a cell-center grid with *ghost points*. The ghost points (unfilled dots) are extra grid points on either side of the interval of interest and are useful when we need to consider the derivatives at the edge of a grid.

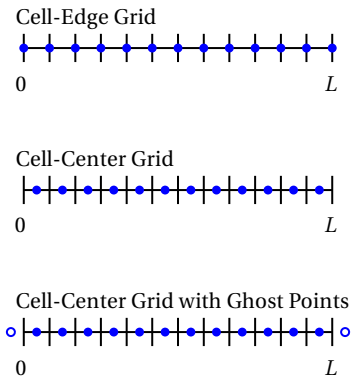


Figure 1.1 Three common spatial grids

P1.1 (a) Make a Python script that creates a cell-edge spatial grid in the variable `x` as follows:

```
from numpy import arange  # Import the needed function
N=100             % the number of grid points
a=0
b=pi          % the left and right bounds
h=(b-a)/(N-1)     % calculate the step size
x=arange(a,b,h)           % build the grid
```

Plot the function $y(x) = \sin(x)\sinh(x)$ on this grid. Explain the relationship between the number of cells and the number of grid points in a cell-edge grid and why you divide by `(N-1)` when calculating `h`. Then verify that the number of points in this x-grid is N (using Python's `len` command).

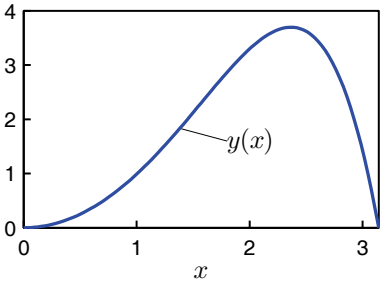


Figure 1.2 Plot from 1.1(a)

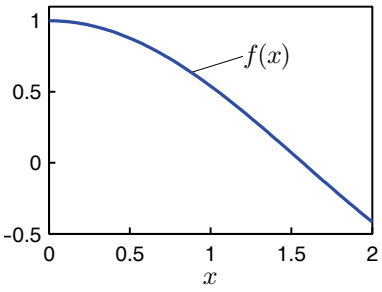


Figure 1.3 Plot from 1.1(b)

(b) Explain the relationship between the number of cells and the number of grid points in a cell-center grid and decide how you should modify the line that calculates `h` in (a) to get the correct spacing for a cell-center grid.

Now write a script like the one in part (a) to build a cell-center grid over the interval $0 \le x \le 2$ with $N = 5000$. Evaluate the function $f(x) = \cos x$ on this grid and plot this function. Then estimate the area under the curve by summing the products of the centered function values f_j with the widths of the cells h like this (midpoint integration rule):

```
sum(f)*h;
```

Verify that this result is quite close to the exact answer obtained by integration:

$$A = \int_0^2 \cos x \, dx.$$

(c) Build a cell-center grid with ghost points over the interval $0 \le x \le \pi/2$ with 500 cells (502 grid points), and evaluate the function $f(x) = \sin x$ on this grid. Now look carefully at the function values at the first two grid points and at the last two grid points. The function $\sin x$ has the property that $f(0) = 0$ and $f'(\pi/2) = 0$. The cell-center grid doesn't have points at the ends of the interval, so these boundary conditions on the function need to be enforced using more than one point. Explain how the ghost points can be used in connection with interior points to specify both function-value boundary conditions and derivative-value boundary conditions.

Interpolation and extrapolation

Grids only represent functions at discrete points, and there will be times when we want to find good values of a function *between* grid points (interpolation) or *beyond* the last grid point (extrapolation). We will use interpolation and extrapolation techniques fairly often during this course, so let's review these ideas.

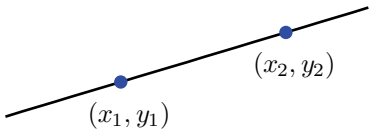


Figure 1.4 The line defined by two points can be used to interpolate between the points and extrapolate beyond the points.

The simplest way to estimate these values is to use the fact that two points define a straight line. For example, suppose that we have function values (x_1, y_1) and (x_2, y_2). The formula for a straight line that passes through these two points is

$$y - y_1 = \frac{(y_2 - y_1)}{(x_2 - x_1)}(x - x_1) \tag{1.1}$$

Once this line has been established it provides an approximation to the true function $y(x)$ that is pretty good in the neighborhood of the two data points. To linearly interpolate or extrapolate we simply evaluate Eq. (1.1) at x values between or beyond x_1 and x_2.

P1.2 Use Eq. (1.1) to do the following special cases:

(a) Find an approximate value for $y(x)$ halfway between the two points x_1 and x_2. Does your answer make sense?

(b) Find an approximate value for $y(x)$ 3/4 of the way from x_1 to x_2. Do you see a pattern?

(c) If the spacing between grid points is h (i.e. $x_2 - x_1 = h$), show that the linear extrapolation formula for $y(x_2 + h)$ is

$$y(x_2 + h) = 2y_2 - y_1 \tag{1.2}$$

This provides a convenient way to estimate the function value one grid step beyond the last grid point. Also show that

$$y(x_2 + h/2) = 3y_2/2 - y_1/2. \tag{1.3}$$

We will use both of these formulas during the course.

A fancier technique for finding values between and beyond grid points is to use a parabola instead of a line. It takes three data points to define a parabola, so we need to start with the function values (x_1, y_1), (x_2, y_2), and (x_3, y_3). The general formula for a parabola is

$$y = a + bx + cx^2 \tag{1.4}$$

where the coefficients a, b, and c need to be chosen so that the parabola passes through our three data points. To determine these constants, you set up three equations that force the parabola to match the data points, like this:

$$y_j = a + bx_j + cx_j^2 \tag{1.5}$$

with $j = 1, 2, 3$, and then solve for a, b, and c.

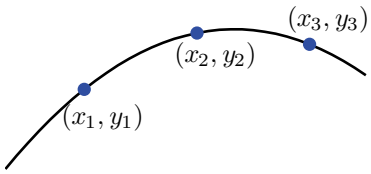


Figure 1.5 Three points define a parabola that can be used to interpolate between the points and extrapolate beyond the points.

P1.3 Use Eq. (1.5) to create a set of three equations in Mathematica. For simplicity, assume that the points are on an evenly-spaced grid and set $x_2 = x_1 + h$ and $x_3 = x_1 + 2h$. Solve this set of equations to obtain some messy formulas for a, b, and c that involve x_1 and h. Then use these formulas to solve the following problems:

(a) Estimate $y(x)$ half way between x_1 and x_2, and then again halfway between x_2 and x_3. Do you see a pattern? (You will need to simplify the answer that Mathematica spits out to see the pattern.)

(b) Show that the quadratic extrapolation formula for $y(x_3 + h)$ (i.e. the value one grid point beyond x_3) is

$$y(x_3 + h) = y_1 - 3y_2 + 3y_3 \tag{1.6}$$

Also find the formula for $y(x_3 + h/2)$.

Derivatives on grids

This is a course on partial differential equations, so we will frequently need to calculate derivatives on our grids. In your introductory calculus book, the derivative was probably introduced using the *forward difference* formula

$$f'(x) \approx \frac{f(x+h)-f(x)}{h}. \tag{1.7}$$

The word "forward" refers to the way this formula reaches forward from x to $x+h$ to calculate the slope. The exact derivative represented by Eq. (1.7) in the limit that h approaches zero. However, we can't make h arbitrarily small when we represent a function on a grid because (i) the number of cells needed to represent a region of space becomes infinite as h goes to zero; and (ii) computers represent numbers with a finite number of significant digits so the subtraction in the numerator of Eq. (1.7) loses accuracy when the two function values are very close. But given these limitation we want to be as accurate as possible, so we want to use the best derivative formulas available. The forward difference formula isn't one of them.

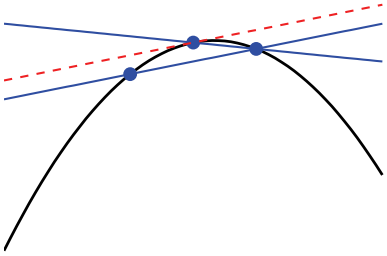


Figure 1.6 The forward and centered difference formulas both approximate the derivative as the slope of a line connecting two points. The centered difference formula gives a more accurate approximation because it uses points before and after the point where the derivative is being estimated. (The true derivative is the slope of the dotted tangent line).

The best first derivative formula that uses only two function values is usually the *centered difference* formula:

$$f'(x) \approx \frac{f(x+h)-f(x-h)}{2h}. \tag{1.8}$$

It is called "centered" because the point x at which we want the slope is centered between the places where the function is evaluated. The corresponding centered second derivative formula is

$$f''(x) \approx \frac{f(x+h)-2f(x)+f(x-h)}{h^2} \tag{1.9}$$

You will derive both of these formulas a little later, but for now we just want you to understand how to use them.

Python's colon operator provides a compact way to evaluate Eqs. (1.8) and (1.9) on a grid. If the function we want to take the derivative of is stored in an array `f`, we can calculate the centered first like this:

```
fp(2:N-1)=(f(3:N)-f(1:N-2))/(2*h);
```

and second derivative formulas at each interior grid point like this:

```
fpp(2:N-1)=(f(3:N)-2*f(2:N-1)+f(1:N-2))/h^2;
```

The variable `h` is the spacing between grid points and `N` is the number of grid points. (Both variables need to be set before the derivative code above will work.) Study this code until you are convinced that it represents Eqs. (1.8) and (1.9) correctly. If this code looks mysterious to you, you may need to review how the colon operator works in the 330 manual *Introduction to Python*.

The derivative at the first and last points on the grid can't be calculated using Eqs. (1.8) and (1.9) since there are not grid points on both sides of the endpoints.

About the best we can do is to extrapolate the interior values of the two derivatives to the end points. If we use linear extrapolation then we just need two nearby points, and the formulas for the derivatives at the end points are found using Eq. (1.2):

```
fp(1)=2*fp(2)-fp(3);
fp(N)=2*fp(N-1)-fp(N-2);
fpp(1)=2*fpp(2)-fpp(3);
fpp(N)=2*fpp(N-1)-fpp(N-2);
```

If we extrapolate using parabolas (quadratic extrapolation), we need to use three nearby points as specified by Eq. (1.6):

```
fp(1)=3*fp(2)-3*fp(3)+fp(4);
fp(N)=3*fp(N-1)-3*fp(N-2)+fp(N-3);
fpp(1)=3*fpp(2)-3*fpp(3)+fpp(4);
fpp(N)=3*fpp(N-1)-3*fpp(N-2)+fpp(N-3);
```

P1.4 Create a cell-edge grid with $N = 100$ on the interval $0 \le x \le 5$. Load $f(x)$ with the Bessel function $J_0(x)$ and numerically differentiate it to obtain $f'(x)$ and $f''(x)$. Use both linear and quadratic extrapolation to calculate the derivative at the endpoints. Compare both extrapolation methods to the exact derivatives and check to see how much better the quadratic extrapolation works. Then make overlaid plots of the numerical derivatives with the exact derivatives:

$$f'(x) = -J_1(x)$$

$$f''(x) = \frac{1}{2}\left(-J_0(x) + J_2(x)\right)$$

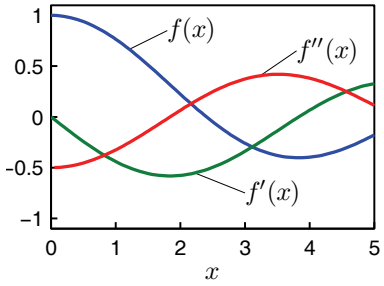


Figure 1.7 Plots from 1.4

Errors in the approximate derivative formulas

We'll conclude this lab with a look at where the approximate derivative formulas come from and at the types of the errors that pop up when using them. The starting point is Taylor's expansion of the function f a small distance h away from the point x

$$f(x+h) = f(x) + f'(x)h + \frac{1}{2}f''(x)h^2 + \cdots + f^{(n)}(x)\frac{h^n}{n!} + \cdots \tag{1.10}$$

Let's use this series to understand the forward difference approximation to $f'(x)$. If we apply the Taylor expansion to the $f(x+h)$ term in Eq. (1.7), we get

$$\frac{f(x+h) - f(x)}{h} = \frac{\left[f(x) + f'(x)h + \frac{1}{2}f''(x)h^2 + \cdots\right] - f(x)}{h} \tag{1.11}$$

The higher order terms in the expansion (represented by the dots) are smaller than the f'' term because they are all multiplied by higher powers of h (which

we assume to be small). If we neglect these higher order terms, we can solve Eq. (1.11) for the exact derivative $f'(x)$ to find

$$f'(x) \approx \frac{f(x+h)-f(x)}{h} - \frac{h}{2}f''(x) \tag{1.12}$$

From Eq. (1.12) we see that the forward difference does indeed give the first derivative back, but it carries an error term which is proportional to h. But, of course, if h is small enough then the contribution from the term containing $f''(x)$ will be too small to matter and we will have a good approximation to $f'(x)$.

Now let's perform the same analysis on the centered difference formula to see why it is better. Using the Taylor expansion for both $f(x+h)$ and $f(x-h)$ in Eq. (1.8) yields

$$\begin{aligned}\frac{f(x+h)-f(x-h)}{2h} &= \frac{\left[f(x)+f'(x)h+f''(x)\frac{h^2}{2}+f'''(x)\frac{h^3}{6}+\cdots\right]}{2h} \\ &\quad -\frac{\left[f(x)-f'(x)h+f''(x)\frac{h^2}{2}-f'''(x)\frac{h^3}{6}+\cdots\right]}{2h}\end{aligned} \tag{1.13}$$

If we again neglect the higher-order terms, we can solve Eq. (1.13) for the exact derivative $f'(x)$. This time, the f'' terms exactly cancel to give

$$f'(x) \approx \frac{f(x+h)-f(x-h)}{2h} - \frac{h^2}{6}f'''(x) \tag{1.14}$$

Notice that for this approximate formula the error term is much smaller, only of order h^2. To get a feel why this is so much better, imagine decreasing h in both the forward and centered difference formulas by a factor of 10. The forward difference error will decrease by a factor of 10, but the centered difference error will decrease by a factor of 100. This is the reason we try to use centered formulas whenever possible in this course.

P1.5 (a) Let's find the second derivative formula using an approach similar to what we did for the first derivative. In Mathematica, write out the Taylor's expansion for $f(x+h)$ using Eq. (1.10), but change the derivatives to variables that Mathematica can do algebra with, like this:

```
fplus=f + fp*h + fp2*h^2/2 + fp3*h^3/6 + fp4*h^4/24
```

where `fp` stands for f', `fp2` stands for f'', etc. Make a similar equation called `eqminus` for $f(x-h)$ that contains the same derivative variables `fp`, `fpp`, etc. Now solve these two equations for the first derivative `fp` and the second derivative `fpp`. Verify that the first derivative formula matches Eq. (1.14), including the error term, and that the second derivative formula matches Eq. (1.9), but now with the appropriate error term. What order is the error in terms of the step size h?

(b) **Extra Credit:** (Finish the rest of the lab before doing this problem.)
Now let's look for a reasonable approximation for the third derivative. Suppose you have function values $f(x-3h/2)$, $f(x-h/2)$, $f(x+h/2)$, and $f(x+3h/2)$. Using Mathematica and the procedure in (a), write down four "algebraic Taylor's" series up to the fifth derivative for the function at these four points. Then solve this system of four equations to find expressions for $f(x)$, $f'(x)$, $f''(x)$, and $f'''(x)$ (i.e. solve the system for the variables `f`, `fp`, `fp2`, and `fp3` if you use the same notation as (a)). Focus on the expression for the third derivative. You should find the approximate formula

$$f'''(x) \approx \frac{f(x+3h/2) - 3f(x+h/2) + 3f(x-h/2) - f(x-3h/2)}{h^3} \tag{1.15}$$

along with an error term on the order of h^2. This expression will be useful when we need to approximate a third derivative on a grid in Lab **??**.

P1.6 Use Python (or a calculator) to compute the forward and centered difference formulas for the function $f(x) = e^x$ at $x = 0$ with $h = 0.1,\ 0.01,\ 0.001$. Also calculate the centered second derivative formula for these values of h. Verify that the error estimates in Eqs. (1.12) and (1.14) agree with the numerical testing.

Note that at $x = 0$ the exact values of both f' and f'' are equal to 1, so just subtract 1 from your numerical result to find the error.

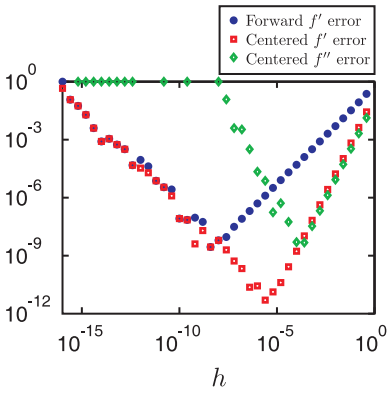


Figure 1.8 Error in the forward and centered difference approximations to the first derivative and the centered difference formula for the second derivative as a function of h. The function is e^x and the approximations are evaluated for $x = 0$.

In problem 1.6, you should have found that $h = 0.001$ in the centered-difference formula gives a better approximation than $h = 0.01$. These errors are due to the finite grid spacing h, which might entice you to try to keep making h smaller and smaller to achieve any accuracy you want. This doesn't work. Figure **??** shows a plot of the error you calculated in problem 1.6 as h continues to decrease (note the log scales). For the larger values of h, the errors track well with the predictions made by the Taylor's series analysis. However, when h becomes too small, the error starts to increase. Finally (at about $h = 10^{-16}$, and sooner for the second derivative) the finite difference formulas have no accuracy at all—the error is the same order as the derivative.

The reason for this behavior is that numbers in computers are represented with a finite number of significant digits. Most computational languages (including Python) use a representation that has 15-digit accuracy. This is normally plenty of precision, but look what happens in a subtraction problem where the two numbers are nearly the same:

$$\begin{array}{rr} & 7.38905699669556 \\ - & 7.38905699191745 \\ \hline & 0.00000000477811 \end{array} \tag{1.16}$$

Notice that our nice 15-digit accuracy has disappeared, leaving behind only 6 significant figures. This problem occurs in calculations with real numbers on all digital computers, and is called *roundoff*. You can see this effect by experimenting with the Python command

```
h=1e-17; (1+h); ans-1
```

for different values of h and noting that you don't always get h back. Also notice in Fig. **??** that this problem is worse for the second derivative formula than it is for the first derivative formula. The lesson here is that it is impossible to achieve arbitrarily high accuracy by using arbitrarily tiny values of h. In a problem with a size of about L it doesn't do any good to use values of h any smaller than about $0.0001L$.

Finally, let's learn some wisdom about using finite difference formulas on experimental data. Suppose you had acquired some data that you needed to numerically differentiate. Since it's real data there are random errors in the numbers. Let's see how those errors affect your ability to take numerical derivatives.

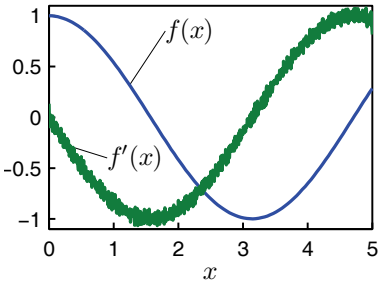


Figure 1.9 Plots of $f(x)$ and $f'(x)$ from 1.7 with 1000 points. $f''(x)$ has too much error to make a meaningful plot for this number of points.

P1.7 Make a cell-edge grid for $0 \le x \le 5$ with 1000 grid points. Then model some data with experimental errors in it by using Python's random number function `rand` like this:

```
f=cos(x)+.001*rand(1,length(x));
```

So now f contains the cosine function, plus experimental error at the 0.1% level. Calculate the first and second derivatives of this data and compare them to the "real" derivatives (calculated without noise). Reduce the number of points to 100 and see what happens.

Differentiating your data is a bad idea in general, and differentiating it twice is even worse. If you can't avoid differentiating experimental data, you had better work pretty hard at reducing the error, or perhaps fit your data to a smooth function, then differentiate the function.

Chapter 2

Quantum Bound States

Today we will study the quantum mechanical bound states corresponding to two potential wells. Instead of solving this problem using the shooting or matching method, as was done last week, we will treat the situation as an eigenvalue problem and use Python to solve it. This is a much more robust way to solve this kind of problem. The general form of the one-dimensional, time-independent Schrödinger's equation is

$$-\frac{\hbar^2}{2m}\frac{d^2\psi}{dx^2} + V(x)\psi = E\psi \tag{2.1}$$

[1] where $V(x)$ is the potential experienced by the particle(s) in question. Let's consider the problem of a particle in a one-dimensional harmonic oscillator well ($V(x) = \frac{1}{2}kx^2$). Schrödinger's equation for this situation is:

[1] This is the time-independent Schrödinger equation. To determine how a particle behaves over time requires a solution to the time-dependent version.

$$-\frac{\hbar^2}{2m}\frac{d^2\psi}{dx^2} + \frac{1}{2}kx^2\psi = E\psi \tag{2.2}$$

with boundary conditions $\psi = 0$ at $x = \pm\infty$. A full analytic solution (paper and pencil solution) to this problem would take the discussion too far astray. We'll just need to remember that the allowed energies for this potential are given by

$$E_n = (n + \frac{1}{2})\hbar\sqrt{\frac{k}{m}} \tag{2.3}$$

Later when we obtain the numerical results we'll want to compare them to the well-know exact answer here.

2.1 A numerical solution

For now we'll start down the path of solving this equation numerically. First, notice that the numbers involved in equation (2.2) are extremely small ($\hbar = 1.05 \times 10^{-34}$ m^2 kg/ s, $m_{\text{electron}} = 9.11 \times 10^{-31}$ kg, etc) and the typical grid sizes that we have been using (2,5,10, etc) will most certainly not be appropriate for this problem (10^{-10} m). We could just set $\hbar$, m, and k to 1, but then we would lose all information about the specific physical situation being studied. Instead of trying to shrink our domain down to appropriate atomic scales, a better approach is to rescale the differential equation so that all of the small numbers go away. This also makes the problem more general, allowing the obtained results to work for any values of m and k.

P1.1 This probably seems a little nebulous(unclear), so follow the recipe below to see how to rescale equation (2.2) (write it out on paper).

1. In equation (2.2) use the substitution $x = a\xi$, where a has units of length and ξ is dimensionless. After making this substitution put Schrödinger's equation in the following form (Note: Think carefully about changing $\frac{d^2\psi}{dx^2} \rightarrow \frac{d^2\psi}{d\xi^2}$. The chain rule will be your friend!)

$$C\left(-\frac{D}{2}\frac{d^2\psi}{d\xi^2} + \frac{1}{2}\xi^2\psi\right) = E\psi \tag{2.4}$$

   where C and D involve the factors $\hbar$, m, k, and a.

2. Make the differential operator inside the parenthesis $(\ldots)$ on the left be as simple as possible by choosing $D = 1$. This choice will allow you to write down a relationship between the characteristic length a and $\hbar$, m, and k. Once you have determined a in this way, check that it has units of length. You should find that

$$a = \left(\frac{\hbar^2}{km}\right)^{1/4} = \sqrt{\frac{\hbar}{m\omega}} \quad \text{where} \quad \omega = \sqrt{\frac{k}{m}} \tag{2.5}$$

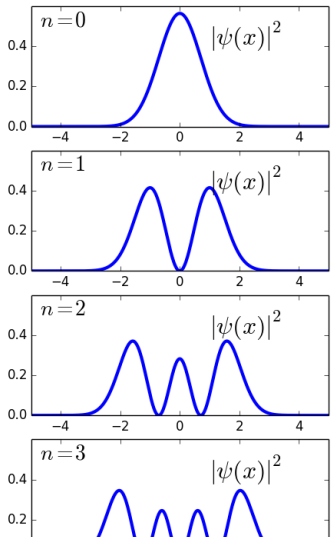
3. Now rescale the energy by writing $E = \epsilon\bar{E}$ where $\bar{E}$ has units of energy and ϵ is unitless. Show that if you choose $\bar{E} = C$ that (2.2) becomes the following dimensionless equation

$$-\frac{1}{2}\frac{d^2\psi}{d\xi^2} + \frac{1}{2}\xi^2\psi = \epsilon\psi \tag{2.6}$$

   You should find that

$$\bar{E} = \hbar\sqrt{\frac{k}{m}} \tag{2.7}$$

   Verify that $\bar{E}$ has units of energy.



Now that Schrödinger's equation is in a unitless form, it makes sense to choose a grid that goes from -5 to 5 , or some other similar pair of numbers. These numbers are suppose to approximate infinity in this problem, so make sure (by looking at the eigenfunctions) that they are large enough that the wave function goes to zero with zero slope at these locations.

Comparing equation (2.3) to equation (2.7) we see that our unitless energies are now given by:

$$\epsilon_n = n + \frac{1}{2} \tag{2.8}$$

[2]

With our attention now turned to solving equation (2.6), we need to write a finite-difference version of the second derivative. The centered-difference version is always the best choice and that's what we'll use here

$$\psi''(\xi) = \frac{\psi(\xi+h) - 2\psi(\xi) + \psi(\xi-h)}{h^2} \tag{2.9}$$

Inserting this equation into Eq. (2.6) gives:

$$-\frac{1}{2}\frac{\psi(\xi+h) - 2\psi(\xi) + \psi(\xi-h)}{h^2} + \frac{1}{2}\xi^2\psi(\xi) = \epsilon\psi(\xi) \tag{2.10}$$

simplifying and switching to index notation

$$\frac{-\psi_{j+1} + 2\psi_j - \psi_{j-1}}{2h^2} + \frac{1}{2}\xi^2\psi_j = \epsilon\psi_j \tag{2.11}$$

P1.2 Notice that equation (3.7) is not a single equation but rather a family of equations. Read that last sentence again. If you need help understanding, call someone over to talk it out with you. Once you understand, write down an explanation that might help you if you ever need to remember this. Write down 2 or 3 of these equations. Here are two that I chose

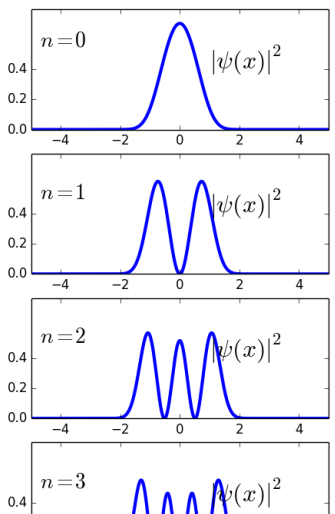
$$-\frac{1}{2h^2}\psi_1 + (\frac{1}{h^2} + \frac{1}{2}\xi^2)\psi_2 - \frac{1}{2h^2}\psi_3 = \epsilon\psi_2 \tag{2.12}$$

$$-\frac{1}{2h^2}\psi_2 + (\frac{1}{h^2} + \frac{1}{2}\xi^2)\psi_3 - \frac{1}{2h^2}\psi_4 = \epsilon\psi_3 \tag{2.13}$$

In linear algebra you should have learned that a system of linear equations can be neatly expressed in matrix form. For this particular problem that would look like

$$\boldsymbol{A\psi} = \epsilon\boldsymbol{B\psi} \tag{2.14}$$

which is written out as



[2] To undo the rescaling and arrive at the true eigenvalues, you can simply plug in chosen values for m and k into the rescaling equations (Eq. (2.7)).

$$\begin{bmatrix} 1 & 0 & 0 & \dots & 0 & 0 \\ -\frac{1}{2h^2} & (\frac{1}{h^2}+\frac{1}{2}\xi_2^2) & -\frac{1}{2h^2} & \dots & 0 & 0 \\ 0 & -\frac{1}{2h^2} & (\frac{1}{h^2}+\frac{1}{2}\xi_3^2) & \dots & 0 & 0 \\ . & . & . & \dots & . & . \\ . & . & . & \dots & . & . \\ . & . & . & \dots & . & . \\ 0 & 0 & 0 & \dots & (\frac{1}{h^2}+\frac{1}{2}\xi_{N-1}^2) & -\frac{1}{2h^2} \\ 0 & 0 & 0 & \dots & 0 & 1 \end{bmatrix} \begin{bmatrix} \psi_1 \\ \psi_2 \\ \psi_3 \\ . \\ . \\ . \\ \psi_{N-1} \\ \psi_N \end{bmatrix} = \epsilon \begin{bmatrix} 0 & 0 & 0 & \dots & 0 & 0 \\ 0 & 1 & 0 & \dots & 0 & 0 \\ 0 & 0 & 1 & \dots & 0 & 0 \\ . & . & . & \dots & . & . \\ . & . & . & \dots & . & . \\ . & . & . & \dots & . & . \\ 0 & 0 & 0 & \dots & 1 & 0 \\ 0 & 0 & 0 & \dots & 0 & 0 \end{bmatrix} \begin{bmatrix} \psi \\ \end{bmatrix}$$

[3] You should have seen this in Math 316

This is called a generalized eigenvalue problem[3] and it can be solved easily in python.You may be wondering about the first and last rows of matrix $\boldsymbol{A}$ as well as why we needed matrix $\boldsymbol{B}$ at all. First notice that equation (3.7) can't be written down when $j = 1$ or when $j = N$ because we would have to step off of the grid to the left or the right to evaluate equation (3.7). Equation (3.7) can only be written down for $1 < j < N$. Excluding those two points on our grid would make $N-2$ equations with N unknowns to solve for. In other words, there would be more unknowns than equations and the system wouldn't be solvable.[4] Luckily, we do have two more equations to add to the system: the boundary conditions. Remember that $\psi = 0$ for $x = \pm\infty$. Our grid does not extend to $\pm\infty$ but it should extend far enough that we expect ψ to be zero. The top and bottom row of matrix $\boldsymbol{A}$ as well as matrix $\boldsymbol{B}$ are used to enforce the boundary conditions.

[4] The linear algebra term for this situation is "underdetermined"

P1.3

1. Take a second to convince yourself that the matrix equation above is really the same thing as equation (3.7).

2. Look at the first and last row and convince yourself that they are enforcing the boundary conditions that $\psi(\pm\infty) = 0$. Can you see the need for matrix $\boldsymbol{B}$ now? What would those boundary conditions look like if $\boldsymbol{B}$ was not there? Take time to understand these questions and answer them in your lab notebook before moving on.

Now that all of the paper and pencil work is done, we are ready to build a program to solve this problem.

P1.4
(i)Start by initializing some variables that you'll need:

1. Number of grid points.
2. Domain end points. You may need to experiment with these. You'll want to make sure that your wave functions go to zero with zero slope at the domain boundaries. If they don't, you should extend them.
3. Step size. This can be calculated from the number of grid points and the domain definition defined above.
4. A list (or numpy array) that holds the location of the grid points. linspace is a good choice for this. After you create this array you'll want to ensure that your step size agrees with the actual spacing in your domain list. Print your domain list and your step size to verify and modify as needed.

(ii)Now load matrices $\boldsymbol{A}$ and $\boldsymbol{B}$. When you are done, print them off (to your screen) to verify that they are correct.

(iii)Once these arrays are loaded, python can easily solve the eigenvalue problem. Use the python code below to solve the eigenvalue problem and plot it. You must put a descriptive comment next to each line of code below to get full credit.

```
self.vals, self.vecs = scipy.linalg.eig(self.A,self.B)
self.key = sorted(range(len(self.vals)), key=lambda k: self.vals[k])
vec = self.vecs[:,self.key[mode]]
normalization = sqrt(numpy.sum(vec * numpy.conjugate(vec) * self.dx))
plt.plot(self.domain,real(vec * numpy.conjugate(vec)/normalization**2))
plt.show()
```

P1.5 Now redo this entire problem, but with the harmonic potential replaced by

$$V(x) = \mu x^4 \tag{2.15}$$

making Schrödinger's equation

$$-\frac{\hbar^2}{2m}\frac{d^2\psi}{dx^2} + \mu x^4 \psi = E\psi \tag{2.16}$$

With this new potential, you will need to once again rescale the equation to eliminate small constants and make the equation unitless. Similar, though not identical, to the harmonic oscillator you're unitless version of this equation should become

$$-\frac{1}{2}\frac{d^2\psi}{d\xi^2} + \xi^4 \psi = \epsilon\psi \tag{2.17}$$

with

$$a = \left(\frac{\hbar^2}{m\mu}\right)^{1/6} \qquad \bar{E} = \left(\frac{\hbar^4\mu}{m^2}\right)^{1/3} \qquad E = \epsilon\bar{E} \tag{2.18}$$

Find the first four bound states by finding the lowest 4 eigenvalues and their corresponding eigenvectors.

Chapter 3

Implicit Methods: the Crank-Nicolson Algorithm

So far we have solved the time-independent Schrödinger equation and found the bound states for several potentials. Now we will solve the time-dependent version of his equation. This will allow us to play "movies" of the wavefunction as it interacts with a potential. Consider the one-dimensional, time-dependent Schrödinger equation

$$-\frac{\hbar^2}{2m}\frac{d^2\psi}{dx^2} + V(x)\psi = i\hbar\frac{d\psi}{dt} \tag{3.1}$$

1.

P3.1 Just as we did last time, let's rescale this equation so it is unitless. Follow the recipe below to see how this is done(write it out on paper).

(a) Since there are two dependent variables in equation (3.1), we will need to make two substitutions. In equation (3.1) use the substitution $x = x_c\bar{x}$ and $t = t_c\bar{t}$, where x_c has units of length, t_c has units of time and $\bar{t}$ and $\bar{x}$ are dimensionless. You should arrive at the following equation

$$\left(-\frac{\hbar^2}{2mx_c^2}\frac{d^2\psi}{d\bar{x}^2} + V(x_c\bar{x})\psi\right) = i\frac{\hbar}{t_c}\frac{d\psi}{d\bar{t}} \tag{3.2}$$

(b) Now factor $\frac{\hbar}{t_c}$ (which has units of energy) out of each term on both sides and cancel them out. You should arrive at the equation

$$\left(-\frac{\hbar t_c}{2mx_c^2}\frac{d^2\psi}{d\bar{x}^2} + \frac{t_c}{\hbar}V(x_c\bar{x})\psi\right) = i\frac{d\psi}{d\bar{t}} \tag{3.3}$$

You just made the entire equation unitless. Verify that $\frac{\hbar t_c}{2mx_c^2}$ is a unitless number.

(c) Now let's make the left hand side as simple as possible by choosing

$$\frac{t_c}{\hbar} = 1 \quad \text{and} \quad \frac{\hbar t_c}{2mx_c^2} = 1 \tag{3.4}$$

Use these two expressions to find expressions for the characteristic length (x_c) and the characteristic time (t_c). You should find that

$$t_c = \hbar \quad \text{and} \quad x_c = \sqrt{\frac{\hbar^2}{2m}} \tag{3.5}$$

You just rescaled the differential equation and the final form is:

$$\left(-\frac{d^2\psi}{d\bar{x}^2} + V(x_c\bar{x})\psi\right) = i\frac{d\psi}{d\bar{t}} \tag{3.6}$$

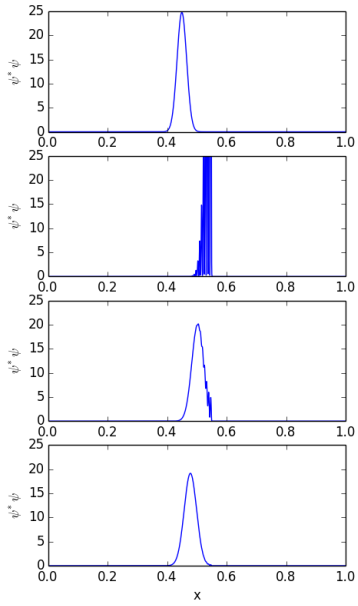


Figure 3.1 Snapshots in time of a wave packet interacting with a step potential located at $x = 0.55$ m. The parameters used to generate the initial wave packet were: $x_0 = 0.4$, $k_0 = 500$, $\sigma^2 = .001$. The height of the step potential was $V_0 = 3.29 \times 10^6$

Now let's start discretizing this differential equation in time and space. Using a centered-difference version of the 2nd order spatial derivative and a forward-difference version of the 1st order time derivative, we find

$$\left(-\frac{\psi(\bar{x}+h,\bar{t}) - 2\psi(\bar{x},\bar{t}) + \psi(\bar{x}-h,\bar{t})}{h^2} + V(x_c\bar{x})\psi(\bar{x},\bar{t})\right) = i\frac{\psi(\bar{x},\bar{t}+\tau) - \psi(\bar{x},\bar{t})}{\tau} \tag{3.7}$$

Here h is the spatial grid size and τ is the time step. The notation used in the equation above can get a little messy. To simplify it, we'll use index notation from here on. Equation (3.7) becomes:

$$\left(-\frac{\psi^n_{j+1} - 2\psi^n_j + \psi^n_{j-1}}{h^2} + V_j\psi^n_j\right) = i\frac{\psi^{n+1}_j - \psi^n_j}{\tau} \tag{3.8}$$

Here j is the spatial index, and n is the temporal index.

P1.2 Take a minute to convince yourself that equation (3.8) is the same thing as equation (3.7)

There is actually a small problem with equation (3.8). Notice that the derivative on the right hand side is centered at time $n + \frac{1}{2}$ but the left hand side is centered at time n. This introduces errors in the algorithm and it would be better to have both sides centered at the same moment in time.

P1.3 To accomplish this, let's replace every occurence of ψ on the left hand side of equation (3.8) with the average

$$\psi^{n+\frac{1}{2}} = \frac{\psi^{n+1} + \psi^n}{2} \tag{3.9}$$

Write this substitution down and see what emerges. You should find that: (write it out)

$$\left(-\frac{(\psi_{j+1}^{n+1} + \psi_{j+1}^n) - 2(\psi_j^{n+1} + \psi_j^n) + (\psi_{j-1}^{n+1} + \psi_{j-1}^n)}{2h^2} + V_j(\frac{\psi_j^{n+1} + \psi_j^n}{2})\right) = i\frac{\psi_j^{n+1} - \psi_j^n}{\tau} \tag{3.10}$$

Look carefully at this equation. Notice that ψ^{n+1} are all over the place. How are we suppose to find an expression for it in terms of the current wavefunction (ψ_j^n) so that we can propagate it forward in time. In hopes of something useful emerging, let's accumulate everything with $n+1$ superscript on one side and everything with an n superscript on the other. Do this by hand and write it out on your paper. You should find that: (write it out)

$$\psi_{j+1}^{n+1} + \left(2i\lambda - h^2 V_j - 2\right)\psi_j^{n+1} + \psi_{j-1}^{n+1} = -\psi_{j+1}^n + \left(2i\lambda + h^2 V_j + 2\right)\psi_j^n - \psi_{j-1}^n \tag{3.11}$$

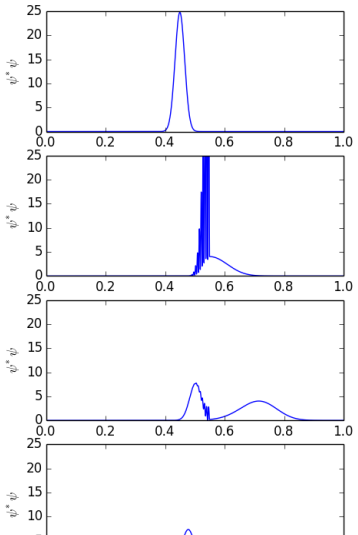
where $\lambda = \frac{h^2}{\tau}$

This equation may look bad, but it's actually not so bad. To solve for ψ^{n+1} (a future wavefunction) all we have to do is solve a linear system of equations. When a system of equations must be solved to find a future value of the function, we call it an implicit method. This is in contrast to a method where an expression for the future value in terms of the past value can easily be determine. This particular implicit method is called the Crank-Nicolson algorithm

To see more clearly how this is a linear algebra problem, let's define a matrix $\boldsymbol{A}$ for the left hand side of equation (3.11) and a matrix $\boldsymbol{B}$ for the right hand side of equation (3.11). Then the linear algebra problem becomes:

$$\boldsymbol{A}\psi^{n+1} = \boldsymbol{B}\psi^n \tag{3.12}$$

which can be written out explicitly as:



$$\begin{bmatrix} 1 & 0 & 0 & \dots & 0 & 0 \\ 1 & 2i\lambda - h^2V_2 - 2 & 1 & \dots & 0 & 0 \\ 0 & 1 & 2i\lambda - h^2V_3 - 2 & \dots & 0 & 0 \\ . & . & . & \dots & . & . \\ . & . & . & \dots & . & . \\ . & . & . & \dots & . & . \\ 0 & 0 & 0 & \dots & 2i\lambda - h^2V_{N-1} - 2 & 1 \\ 0 & 0 & 0 & \dots & 0 & 1 \end{bmatrix} \begin{bmatrix} \psi_1^{n+1} \\ \psi_2^{n+1} \\ \psi_3^{n+1} \\ . \\ . \\ . \\ \psi_{N-1}^{n+1} \\ \psi_N^{n+1} \end{bmatrix}$$

$$= \begin{bmatrix} 0 & 0 & 0 & \dots & 0 & 0 \\ -1 & 2i\lambda + h^2V_2 + 2 & -1 & \dots & 0 & 0 \\ 0 & -1 & 2i\lambda + h^2V_3 + 2 & \dots & 0 & 0 \\ . & . & . & \dots & . & . \\ . & . & . & \dots & . & . \\ . & . & . & \dots & . & . \\ 0 & 0 & 0 & \dots & 2i\lambda + h^2V_{N-1} + 2 & -1 \\ 0 & 0 & 0 & \dots & 0 & 0 \end{bmatrix} \begin{bmatrix} \psi_1^{n} \\ \psi_2^{n} \\ \psi_3^{n} \\ . \\ . \\ . \\ \psi_{N-1}^{n} \\ \psi_N^{n} \end{bmatrix}$$

This is not an eigenvalue problem like we have done previously, but rather a matrix inversion problem. In other words, ψ^{n+1} can be solved for by

$$\psi^{n+1} = \boldsymbol{A}^{-1}\boldsymbol{B}\psi^n \tag{3.13}$$

Python (well scipy actually) has routines that can solve this problem and we will use them today to evolve the wave function in time. Just as in a previous assignment, the entries in the first and last rows of matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ have been chosen to enforce a boundary condition: $\psi = 0$ at the boundaries.

Now we are ready to code. I'll ask you to write some of the code yourself and I'll give you other parts of the code. Please read carefully:

```
cos(x)
sin(x)
tan(x)
acos(x)
asin(x)
atan(x)
atan2(y,x)
exp(x)
log(x)
log10(x)
log2(x)
sqrt(x)
cosh(x)
sinh(x)
tanh(x)
```

P1.4 (i) Begin by initializing a few needed variables

1. Define the left boundary of the problem to be 0.
2. Define the right boundary to be 1.
3. Define h (spatial step size) to be .0005
4. Define τ (temporal step size) to be 5×10^{-7}
5. Calculate the number of spatial grid points you'll have.
6. Calculate your spatial grid (linspace is a good choice). After you do it, print off the domain and h(the spatial step size defined above) and compare to ensure they agree. Modify as needed.
7. Initialize λ: $\lambda = \frac{h^2}{\tau}$

(ii) Next you'll need to initialize a wave packet. The initialization is critical to giving your wave packet an appropriate initial velocity. One way to give your packet a positive initial velocity is using the following function:

$$\psi(x, t = 0) = e^{-(x-x_0)^2/\sigma^2} \times e^{ik_0 x} \tag{3.14}$$

The imaginary `i` here can be expressed in python as `1.j` or `complex(0,1)`. Using this function, initialize your wavefunction using $k_0 = 500$, $x_0 = 0.4$ and $\sigma^2 = .001$. If you choose to use the function `zeros` to initialize your wavefunction, you'll want to make sure that you specify that the type of data in the array are complex. Something like this ought to work: `zeros(N,dtype=complex)`. Before moving on, plot this function to ensure that it is what you expect. You should probably plot the square of the wave function ($\psi^*\psi$) which can be done like this

```
plt.scatter(self.domain,self.psi * conjugate(self.psi))
```

where `conjugate` comes from `numpy`.

(iii) When you load matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ you will have to evaluate the potential on the spatial grid points. In preparation for that, define a function for the potential barrier. Let's use a step potential for now:

$$V(x) = \begin{cases} 3.29 \times 10^6 & x \geq 0.55 \\ 0 & \text{otherwise} \end{cases}$$

(iv) Construct the matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ as defined in equation (3.12). This should be very similar to what you did with bound states last time. I suggest that you initialize the arrays to zero and then load them with the appropriate values with a loop. Print them off to verify that they are correct.

(v) The size of these matrices, in general, will be large and solving them is not an easy task. Computationalists look for every possible way to speed up a calculation like this. In this case, notice that both $\boldsymbol{A}$ and $\boldsymbol{B}$ are banded tri-diagonal matrices. In other words, all elements are zero

(vi) Now let's perform the time evolution of the wavefunction. This is actually quite simple. Equation (3.12) is solved using the function `solve_banded` (comes from scipy.linalg). This is how you do that:

```
b = dot(self.B,self.psi)
self.psi = solve_banded((1,1),self.ab,b)
```

Note `dot` comes from `numpy`. The solution to this equation provides the wavefunction at a future time. (a very short time into the future) Once we have this wavefunction, we can simply loop over these two lines for as long as we want to continue to get the next wave function in time. You'll want to enclose these two statements in a loop that runs for as long as you want. You may also want to include a plot inside the loop to generate an animation. Here's the plot that I used:

```
plt.scatter(self.domain,self.psi * conjugate(self.psi))
```

P1.5 The height of the barrier was chosen to minimize tunneling into the barrier. Decrease the height slowly until you begin to observe some tunneling.

P1.6 Change the potential from a barrier potential to a cliff potential.

$$V(x) = \begin{cases} -3.29 \times 10^6 & x \geq 0.55 \\ 0 & \text{otherwise} \end{cases}$$

and observe the time evolution

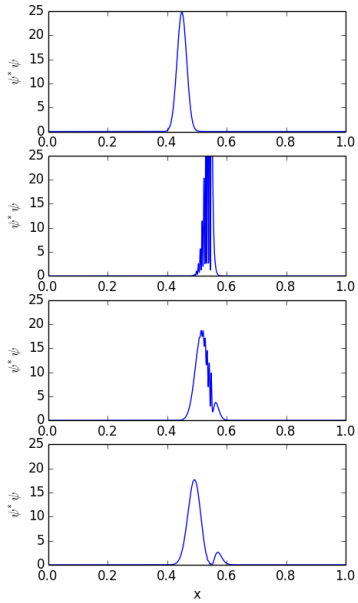


Figure 3.3 Snapshots in time of a wave packet interacting with a step potential located at $x = 0.55$ m. The parameters used to generate the initial wave packet were: $x_0 = 0.4$, $k_0 = 500$, $\sigma^2 = .001$. The height of the step potential was $V_0 = 2.85 \times 10^5$. Notice the