Computational Solution to Discrete Time-Dependent Schrodinger's Equation:

Wave packet's Interactions with Potential Wells and Barriers

In Griffith's Problem 2.22, you were able to analytically find the time dependence of a Gaussian wave packet in free space (or, equivalently, under the influence of a uniform potential.) As you might imagine, it's not so easy to find how an initially-Gaussian wave packet time evolves under the influence of other potentials. Doing that *analytically* is beyond the scope of this course. However, doing it *computationally* isn't so bad, and it nicely complements Chapter 2's discussion of transmission and reflection.

So, this handout will guide you through work on some code that will allow you to visualize a Gaussian wave packet's interaction with a potential well / barrier.

The Discrete Approximation

The computational approach you'll use is based on approximating the derivatives in Schrödinger's Equation as ratios of finite (rather than infinitesimal) differences. Schrödinger's Equation in 1-D is

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

From the handout on Setting up the Discrete Schrodinger Equation, you know that the left-hand-side can be approximated as

$$-\frac{\hbar^2}{2m}\frac{\psi(x_{j-1},t_k)-2\psi(x_j,t_k)+\psi(x_{j+1},t_k)}{(\Delta x)^2}+V(x_j,t_k)\psi(x_j,t_k),$$

where I've explicitly denoted both the position and time dependence of the wave function and the potential. Notice that there's a nice symmetry to how the second-derivative is expressed: it depends equally on the wave function a step ahead (x_{j+1}) as it does on the wave function a step behind (x_{j-1}).

Now for the right-hand-side of Schrödinger's Equation, it's traditional to say

$$\lim_{\Delta t\to 0}\frac{\psi(t+\Delta t)-\psi(t)}{\Delta t}\to\frac{\partial\psi(t)}{\partial t},$$

which is indeed true… in the *limit.* But along the way, the left hand side really gives the average slope over a time interval with one *edge* at time *t;* a better approximation to the instantaneous slope at time *t* would be the average slope *straddling* time *t.* That would be

$$\frac{\psi(t+\Delta t/2)-\psi(t-\Delta t/2)}{\Delta t}\approx\frac{\partial\psi(t)}{\partial t}$$

or, equivalently,

$$\frac{\psi(t+\Delta t)-\psi(t-\Delta t)}{2\Delta t}\approx\frac{\partial\psi(t)}{\partial t}.$$

Using subscripts to denote specific times, that's written

Computational Solution to Discrete Time-Dependent Schrodinger's Equation:

Wave packet's Interactions with Potential Wells and Barriers

$$\frac{\psi(t_{k+1})-\psi(t_{k-1})}{2\Delta t}\approx\frac{\partial\psi(t_k)}{\partial t}.$$

So, our approximation for the time-dependent Schrödinger Equation is

$$-\frac{\hbar^2}{2m}\frac{\psi(x_{j-1},t_k)-2\psi(x_j,t_k)+\psi(x_{j+1},t_k)}{(\Delta x)^2}+V(x_j,t_k)\psi(x_j,t_k)\approx -i\hbar\frac{\psi(x_j,t_{k+1})-\psi(x_j,t_{k-1})}{2\Delta t}.$$

With a little rearrangement,

$$\psi(x_j,t_{k+1})\approx\psi(x_j,t_{k-1})+i\frac{\Delta t\hbar}{(\Delta x)^2 m}\left(2\left(\frac{(\Delta x)^2 m}{\hbar^2}V(x_j,t_k)+1\right)\psi(x_j,t_k)-\left(\psi(x_{j-1},t_k)+\psi(x_{j+1},t_k)\right)\right), \quad (1)$$

and it becomes clear that, if you knew the wave function's values everywhere at the previous times t_{k-1} and t_k, then you could solve for the wave function's new values everywhere at the new time, t_{k+1}. This is the central relation for the program you'll write.

Of course, this means that before you get your program running you'll already need to know (or have a good approximation for) the wave function for the first two time steps – after that, the program takes over. Imagine a Gaussian wave packet that's initially far from a potential well or barrier. In spite of the presence of the (rather distant) potential well, for the first two time steps, you could reasonably approximate that wave packet using the exact analytical solution for when there is no potential, that is, the relation found in Griffiths' problem 2.22 and which you used in your previous program. Ever after, you can use Equation (1) to update the wave function's values through space as it interacts with the potential.

Starting with the program that you've already written to visualize Griffiths' problem 2.22, you'll make a number of modifications (described below) to turn it into a program that uses Equation (1) this way.

Constants – Respecting the Approximation

When using Equation (1), the finite-difference approximation to the Schrodinger equation, the central approximation is that Δt and Δx are as good as infinitesimal. More specifically, it's that $\partial\psi(t)/\partial t$ is essentially constant on the time scale of Δt and that $\partial^2\psi(x)/\partial x^2$ is essentially constant on the length scale of Δx. When it comes to writing your simulation, you'll need to be careful to not violate this approximation. This boils down to being careful how you choose *a*, (the spatial decay constant in the Gaussian expression), *V* (the potential well's depth / barrier's height), and *v* (the wave packet's initial speed), as well as the relationship between Δx and Δt. So before considering how to write the code, we'll take a moment to consider the constraints on these constants.

Thinking of a simple, sinusoidal traveling wave, you know that $\partial^2\psi(x)/\partial x^2$ is relatively constant only on length scales much shorter than the wavelength and, similarly, $\partial\psi(t)/\partial t$ is relatively constant only on time scales much shorter than the period. So, we need the finite steps to be much smaller than these; that is

$$\lambda >> \Delta x \quad (2a)$$

and

$$T >> \Delta t\,. \tag{3a}$$

Nyquist Limit

It's all good and well to say ">>", but practically speaking, making Δ*x* and Δ*t* smaller than they need to be means your code will run slower than it needs to, so it will pay for us to make this a little more specific. Recall hat in the time-independent discrete Schrodinger's equation simulation that you'd written, you'd seen the truth of the Nyquist Limit. That is, your solutions started getting pretty bad as your wavelengths shrank toward 2Δx. For longer wavelengths, they did alright (though not too great when you got close), and above that they did pretty darn poorly. So, if you've got enough time to run your program, by all means, respect inequalities (2) and (3), but at the very least maintain

$$\lambda > 2\Delta x \tag{2b}$$

and

$$T > 2\Delta t\,. \tag{3b}$$

For a Gaussian wave packet, you won't explicitly set wavelength's and periods, but we'll shortly argue that these two inequalities set limits on the potential well's depth (or barrier's height) as well as the packet's speed – both of which you will set in the program. First, however, a similar argument sets a limit on the Gaussian's width.

Gaussian's Width

For a Gaussian wave packet, of the form $\psi(x) = Ae^{-ax^2}$, $\sqrt{1/a}$ characterizes its width and curvature in a similar way that wavelength does for a sinusoidal wave function. So the approximation only holds if

$$a << \frac{1}{(\Delta x)^2}\,. \tag{4a}$$

In the spirit of the Nyquist Limit, it's advisable to at least maintain

$$a < \frac{1}{(2\Delta x)^2} \tag{4b}$$

There is actually a second, subtler reason this inequality is important. Remember that a Gaussian wave packet can be thought of as a sum of sinusoidal traveling waves with a range of different wavelengths. The constant *a* dictates both the width of space that the Gaussian wave packet initially spans and the range of wavelengths for the sinusoidal waves that make up the Gaussian – the largest wavelength of a significantly-represented sinusoidal wave that's in the Gaussian's 'recipe' is on order of $\sqrt{1/a}$. As the Gaussian wave packet time evolves (even if it's just sitting there, not interacting with any potential energy), some of these constituent sinusoidal waves rear their heads has ripples in the evolving Gaussian packet, especially in its real and imaginary components. Having ripples on order of or smaller than Δ*x* would violate the approximation inherent in Equation (1). So inequality (4) is important for that reason.

Inequality (4) will be directly applicable to the program you write – there will be a parameter *a* that you'll need to make sure respects this inequality.

Gaussian's Speed

In the simulation, you'll be interested not in a stationary wave packet, but a traveling one. So you'll give it initial velocity *v*. Again, thinking of the Gaussian as a sum of sinusoidal traveling waves, some would have greater and some would have lesser speeds, but they'd all be on this order. Of course, each one's wavelength would be related to its speed by

$$\frac{h}{\lambda} = p = mv\,,$$

and so, the constraint that the wavelength must be much larger than the spatial step, inequality (2), would imply

$$v << \frac{h}{m\Delta x}\,. \tag{5a}$$

If you must push the speed limit, at least maintain

$$v < \frac{h}{2m\Delta x}\,. \tag{5b}$$

So, if you ensure that the packet's speed respects this inequality and that *a* respects inequality (4), then inequality (2) won't get violated for any of the Gaussian's significant constituent sinusoidal waves.

Potential Well Depth

As for the potential energy, the argument is similar to that for the speed since a particle speeds up (shortens its wavelength) if it travels into a deep potential well (think of a ball speeding up as it rolls down a hill / into a gravitational well.) In the simple case of a wavepacket that initially has negligible kinetic or potential energy and travels into a potential well, it acquires kinetic energy equal to the well's depth

$$-V = K = \tfrac{1}{2}mv^2\,.$$

Substituting inequality (5) into this relationship tells us that we must have

$$V >> -\frac{1}{2m}\left(\frac{h}{\Delta x}\right)^2. \tag{6a}$$

Or, at the very deepest, don't exceed

$$V > -\frac{1}{2m}\left(\frac{h}{2\Delta x}\right)^2. \tag{6b}$$

Notice the negative sign and the direction of the inequality: together, you can read them as saying *V* must be "much more positive than" the term on the right. A separate consideration will constrain the maximum height of a potential barrier. But first,…

Computational Solution to Discrete Time-Dependent Schrodinger's Equation:

Wave packet's Interactions with Potential Wells and Barriers

Relation of Time and Spatial Step Sizes

Now let's consider Δt. Just as the constraint that $\lambda >> \Delta x$ (inequality 2) leads to constraints on kinetic energy, the constrant that $T >> \Delta t$ (inequality 3) leads to a constraint on total energy. Period and total energy are related by

$$E = \frac{h}{T}.$$

So inequality (3) then requires

$$\frac{h}{\Delta t} >> E\,. \tag{7a}$$

Or, pushing the Nyquist limit, at least respect

$$\frac{h}{2\Delta t} > E\,. \tag{7b}$$

On the other hand, for a free particle,

$$E = K = \tfrac{1}{2}mv^2 << \frac{1}{2m}\left(\frac{h}{\Delta x}\right)^2.$$

Equating these two expressions for the cap on the energies for wave functions you can hope to faithfully simulate,

$$\frac{h}{\Delta t} = \frac{h^2}{2m(\Delta x)^2}$$

or

$$\Delta t = \frac{2m}{h}(\Delta x)^2.$$

This tells us that if you're confident that the wave function you're simulating has no significant *spatial* variations as small as Δx, then you can rest assured that it won't have any significant *time* variations as small as Δt. To be on the safe side, we can treat this as a limit on how large Δt can be made, so

$$\Delta t \le \frac{2m}{h}(\Delta x)^2\,. \tag{8}$$

While respecting this constraint will ensure that your results are in good *quantitative* agreement with the exact solution, you can get away with violating it just a little if you just want fairly good *qualitative* agreement. However, the simulation will get unstable if you push it too far. To see this, look back at equation (1). You'll notice that the imaginary term has a factor of $\frac{\Delta t\hbar}{m(\Delta x)^2}$ which, according to inequality (8) must be less than $1/\pi$ (notice it has an $\hbar$ whereas inequality (8) has $2/h$). In the simplest case, where V = 0, imagine an initial condition for which $\psi(x_{j-1},t_k)=\psi(x_{j+1},t_k)=\psi(x_j,t_{k-1})=0$, but $\psi(x_j,t_k)=\delta<<1$, some negligibly small value. Applying equation (1), that would return a $\psi(x_j,t_{k+1})=i\left(\frac{\Delta t\hbar}{m(\Delta x)^2}\right)2\delta$. So, even if the wave function remained 0 at neighboring points (which it wouldn't do), in the next iteration, you'd have $\psi(x_j,t_{k+1})=\left(i\left(\frac{\Delta t\hbar}{m(\Delta x)^2}\right)2\right)^2\delta$, and in the n^{th} iteration it would be $\psi(x_j,t_{k+1})=\left(i\left(\frac{\Delta t\hbar}{m(\Delta x)^2}\right)2\right)^n\delta$. Now, respecting inequality (8), the factor in the outer bracket is less than 2/ π, which is less than 1. So, after *n* iterations, this term would have died a dwindling death. In fact, as long as the term inside the outer brackets is simply < 1, the same will happen eventually. However, if this term is >1, you get qualitatively different behavior – this term grows and grows. So, to preserve the correct qualitative behavior, you need at least $\left(\frac{\Delta t\hbar}{m(\Delta x)^2}\right)2<1$. That is to say, you need

$$\Delta t<\frac{1}{2}\frac{m(\Delta x)^2}{\hbar}=\frac{\pi m}{h}(\Delta x)^2 \tag{9}$$

to ensure the correct qualitative behavior.

Potential Barrier Height

Finally, considering a particle under a potential barrier tells us the maximum height we can give to a barrier (we've already figured out the lowest depth for a well.) For simplicity, say a particle's kinetic energy is much less than the barrier height so

$$\frac{h}{T}=E\approx V.$$

Though T now corresponds with an exponential decay time rather than a period, it's still true that we want $T>>\Delta t$, which requires

$$\frac{h}{\Delta t}>>V. \tag{10}$$

Potential Well Depth & Barrier Height

Combining our two constraints for *V*, inequalities (6) and (10) gives

$$-\frac{1}{m}\left(\frac{h}{\Delta x}\right)^2<<V<<\frac{h}{\Delta t}. \tag{11}$$

Invoking the relation between Dt and Dx, for simplicity sake, we can say

$$|V|<<\frac{1}{m}\left(\frac{h}{\Delta x}\right)^2, \tag{12a}$$

or at least

$$|V| < \frac{1}{m}\left(\frac{h}{2\Delta x}\right)^2 . \tag{12b}$$

Defining Constants in your Simulation

Now that we've learned these lessons, here's how to act on them when writing your program. To keep these constraints in mind, it's handy to define the constants in your program in terms of them. For example, after having defined *h*, *m*, and *deltax*, you could define

```
a = 0.004*(1/(2*deltax)**2)             #keep a < 1/(2*deltax)**2
v = 0.002*(h/(m*2*deltax))              #keep v < h/(m*2*deltax)
deltat= 0.001*(m*deltax**2/h)           #keep deltat<2* m*deltax**2/h, at most pi*m*deltax**2/h
V = 0.004*((1/m)*(h/(2*deltax))**2)     #keep |V|<(1/m)*(h/(2*deltax))**2
```

Define Potential Energy

In your original program, there was no potential energy, so you'll need to add some lines to define it. A convenient and versatile way of doing that is defining a function which you can call on later in you program (much like you can "call on" the pre-defined mathematical functions like sin(x)). After you've defined the well depth, *V*, as well as its center location (WellPos) and its width (WellWidth) in your constants section of the program, the following lines of code will do the job.

```
def V(x,t):
    height = 0
    If  (x>WellPos-0.5*WellWidth and x<WellPos+0.5*WellWidth):
            height = V
    return height
```

These can go near the top of your program, after the section where you define your constants. This will effectively create a square potential well (for negative V) or barrier (for positive V).

You should make two more additions to your code so you can visualize this well.

- Where you create the curves g, r, and b, make an identical line to create curve "Vcurve".
- Down inside the loop where you update the y-component of curves g, r, and b, add a similar line to update the y-component of Vcurve:
  ```
  Vcurve.pos[j][1]=V(g.pos[j][0],t)/EnergyScaleFactor
  ```
- Note: you'll want to define EnergyScaleFactor up in the constants section; you can adjust this to keep the plots of the wave function and the potential energy on the same scale.
- While you're at it, along with defining Vcurve, define Ecurve so you can visualize the relationship between the wave packet's energy and the potential energy well's depth.
  ```
  Ecurve = curve(x=arange(-L,L+deltax,L/2), y = 0.5*m*vo**2/EnergyScaleFactor)
  ```

Computational Solution to Discrete Time-Dependent Schrodinger's Equation:

Wave packet's Interactions with Potential Wells and Barriers

While the potential, as defined, doesn't change over time, coding things this way will make it easy for you to later implement time-evolving potentials.

Initialize Wave function

Equation (1) relates the wave function evaluated at three different times: t_{k-1}, t_k, and t_{k+1}, so it will be convenient to create three arrays, one representing the wave function at each of these times; say, WavesNew representing $\psi(x_j, t_{k+1})$, WavesOld representing $\psi(x_j, t_{k-1})$, and Waves representing $\psi(x_j, t_k)$. To do this, you'll want to go just above your time loop and, for example, for WavesOld, you first create an empty list before the loop and then a loop to step through filling it with initial values:

```
WavesOld = []

for j in arange(0, J+0.1):

        WavesOld.append((2*a/pi)**0.25 *(exp(-a*(x-xo)**2 /
                                                (1+2j*hbar*a*t/m)))/(1+2j*hbar*a*t/m)**0.5)
```

Similarly, up before this loop create empty Waves and WavesNew arrays and add lines inside this loop to populate them. For Waves, the idea is it should be the same as WavesOld but evaluated one time step later, so you'll use the same expression in Waves.append(…) as you did for WavesOld, except evaluated at (*t+deltat*) rather than at *t*.

As for WavesNew, ultimately we're going to use our discretized Schrodinger's equation (Eq'n 1) to figure it out, so it doesn't matter what you initially populate this array with; it could be all zeros:

```
WavesNew.append(0)
```

Later in the program, you'll be using Equation (1) to find its values based on the WavesOld, Waves, and the potential.

Time Evolve the Wave function

In the program you're modifying, you already have a while t<tmax: loop, and nested inside it is a for j in arange(0,J+0.1): loop. Just above the existing for j in arange… loop line, create an identical one at the same indentation level. So, within the while t<tmax loop you'll have two consecutive for j in arange… loops. The first one is going to get used to calculate new values for WavesNew, and once that's done, the second loop will be used to rename things: WavesOld gets overwritten with Waves values, and Waves values get overwritten with WavesNew so you're ready for the next cycle of the time loop.

Inside the top for j in arange(0, J+0.1) loop that's nested in the while t<tmax loop, it will be convenient to include a line that relates x's and j's:

```
x = -L + j*deltax
```

After that, write the line of code that corresponds to Eq'n (1). So that's

```
WavesNew[j] =WavesOld[j]+1j*(…you should be able to figure out what goes in here…)
```

where the "1j" is interpreted as $\sqrt{-1}$.

Now, the astute reader will have noticed that there's a problem for j = 0 since the expression involves Waves[j-1] which would be Waves[-1] which doesn't exist; similarly there's a problem for j = J. So, you'll need to handle these two end points separately. The simplest approach is to change the arange(0, J+0.1) to arange(1,J-1+0.1). That's sufficient as long as the wave packet never gets too close to the ends since it leaves the two end point's values frozen in time (if you do allow the wave packet to get near the ends, you'll see a reflection much like you'd see on a string when a pulse reflects from a fixed end.)

Now that you've stepped through all j values and calculated the new values of the WavesNew, it's time to do two things: display and step forward your three lists, that is, overwrite WavesOld with Waves and overwrite Waves with WavesNew. Both of these things can be done in the second loop for j in arange(0,J + 0.1). Add in the lines

```
WavesOld[j] = Waves[j]

Waves[j] = WavesNew[j]
```

Or, more compactly,

```
WavesOld[j], Waves[j]  = Waves[j], WavesNew[j]
```

Copy and then comment out (but don't delete) the existing g.pos[j][1], r.pos[j][1], and b.pos[j][2] lines within this loop. Paste new copies of these lines in that loop and modify them to depend on Waves[j]'s absolute value, real, and imaginary components.

Test

When there is no potential well and the wave packet is stationary, your new program, which makes use of Equation (1) to time evolve the Gaussian, should agree with your old program, which used the exact analytical relation to time evolve the Gaussian. Fortunately, you should still have that analytical expression in your new code. So, to see that your program at least works when there is no potential well and for a stationary wave packet, set V = 0 and v=0, and use the old (commented out) and new lines that update g, r, and b's positions so you can compare what the old, analytical solution gives with what the new, computational solution gives.

If you respect the constraints on a and Δt that are given in inequalities (4) and (8), then the exact analytical and the approximate computational solutions should be virtually identical (with the possible exception of a sign flip on the real or imaginary components).

To see the effect of violating the conditions of inequalities (4) and (8), I suggest you try making a and Δt larger and larger and see how the approximate solution begins to differ from the exact one. The two approaches, the exact solution and the finite-difference approximation that uses Equation (1), won't make identical plots, but they should be quite similar for quite a while (the approximation starts misbehaving when the wave function gets weak.)

Do the same for the real components and then the imaginary components (don't worry if the two approaches happen to differ by a sign.)

Computational Solution to Discrete Time-Dependent Schrodinger's Equation:

Wave packet's Interactions with Potential Wells and Barriers

Simulate Wave hitting a Potential Barrier

Once you've tested out your program to ensure that it reproduces the analytical solution when there is no potential well or barrier, you're ready to try bouncing it off a potential barrier. It's actually hard to get the simulation to give the *Gaussian* packet an initial velocity, but quite easy to give the *potential barrier* the opposite one, and as the principle of relativity tells us, that's perfectly equivalent. Inside your definition of the potential energy, change the condition of the if statement to read

```
        if  (x>WellPos-v*t and x<L):
```

Including the "-v*t" will effectively move the well's left edge leftward with velocity –v, which is equivalent to moving the Gaussian with velocity v. The right edge is held at L so the Gaussian packet only ever encounters the one side of the barrier.

I encourage you to explore a range of parameters, but to get you started, here are some that work:

```
V = 0.004*((1/m)*(h/(2*deltax))**2)     #keep |V|<(1/m)*(h/(2*deltax)**2)
J=2001
deltax = 0.01
L = (J-1)/2*deltax
hbar = 1
h= hbar*(2*pi)
m = 1
a = 1/(2*deltax*(J/80))**2 #a = 0.01/deltax**2 is about the largest that agrees with the exact solution
deltat = 0.9*pi*(m/h)*deltax**2 # keep deltat < pi* (m/h)*deltax**2tmax = 8000*deltat #long enough
to see reflection, short enough to not reflect off x = L
scalefactor = 3
EnergyScaleFactor = 100    #so the well / barrier can be displayed on same scale as wavefunction
v = 0.03*h/(m*2*deltax)    # should keep below h/(m*2*deltax)
xo = 0.75*L
WellPos = 0.9*L
n = 2 # as vo and WellHeight are defined, this is the ratio of WellHeight to Gaussian's energy
WellHeight = n*0.5*m*v**2  #defined to easily relate to kinetic energy but keep  |V|<
(1/m)*(h/(2*deltax))**2
```

Things to try:

- Vary the "WellHeight"; as defined above, it is exactly twice the particle's kinetic energy, so very little of the particle should tunnel in. Bring the "WellHeight" below the kinetic energy and see what happens.

- Try switching the WellHeight to negative so it's really a well instead of a barrier.

Computational Solution to Discrete Time-Dependent Schrodinger's Equation:

Wave packet's Interactions with Potential Wells and Barriers

- Try defining the well as box of definite width so the Gaussian packet will meet one and then the other end. This can be done by defining WellWidth, and then in your definition of the well, change the if statement to

```
        if  (x>WellPos-v*t – WellWidth/2 and x< WellPos-v*t + WellWidth/2):
```

- **Optional**: Get quantitative and determine Transmission and Reflection coefficients. At the end of the program, outside the time loop, you can add one more loop over j's and add up all the abs(wave[j])**2 values that are for positions outside the barrier (to the left of the barrier's final location) and separately add up all the abs(wave[j])**2 values for positions inside the barrier (to the right of barrier's final location). The ratio of the outside sum over the two sums is the reflection coefficient, and the ratio of the inside sum over the two is the transmission coefficient.

Delta Potential Barrier and Well

The closest we can come to approximating a delta potential or well, as discussed in section 2.5, is to define a well for which WellWidth = deltax. Then what corresponds to α is WellHeight*deltax. The way vo and WellHeight are defined above, changing *n* to 20 would make the transmission an reflection coefficients ½.

Things to Try

Set the WellWidth to deltax and run the program for various values of n above and below 20, and similarly above and below -20 (for a delta-function well rather than barrier).

Other Waves

While there are reasons to like the Gaussian wave packet, the program should work just fine for any initial wave packet (that is smooth on the scale of Δx), so feel free to play around with defining other initial packets if you like.