QSE 200 Sections

Ben Augenbraun

11/5/2022

Table of contents

Preface

This book contains the notes for recitation sections covered in QSE 200 during Fall 2022. This course was a first-year graduate-level treatment of quantum mechanics with a special emphasis on numerical problems in quantum mechanics.

1 Section 2: Numerically Solving the TDSE

Last week, we developed code to solve the time-independent Schrodinger equation (TISE). In that code, we used a second-order finite difference approximation to the second derivative and then computed numerical eigenvalues and eigenvectors associated with the Hamiltonian. In this week's section, we'll take a similar approach to solve the time-*dependent* Schrodinger equation (TDSE). After developing the numerical method and writing code to implement it, we will propagate Gaussian wavepackets to look at their behavior in free space and as they interact with some interesting potentials.

Learning Goals: After this Section you should be able to: - Understand how to discretize the Schrodinger equation in both space and time - Explain the difference between the Crank-Nicolson method and the forward- or backward-difference methods - Compute and describe the behavior of a Gaussian wavepacket moving in free space - Simulate and qualitatively rationalize the behavior of a Gaussian wavepacket interacting with at least one potential

1.1 Background: Simulating Time Dependence

In class, we talked about one way to think about how an arbitrary wavefunction evolves in time that relies on the simple time evolution of energy eigenstates. That idea was as follows: - Express an arbitrary wavefunction $\phi(x)$ as a superposition of the energy eigenstates for the specific potential of interest. This gives you an expression like $\phi(x) = \sum_n c_n \psi_n(x)$, where the c_n can be determined (either numerically or analytically) by taking inner products between the state $\phi(x)$ and the energy eigenstates. - Append the appropriate time-dependent phase factor to each term in this series, yielding $\phi(x,t) = \sum_n c_n \psi_n(x) e^{-i\omega_n t}$. - Evaluate the sum at each time t of interest.

You'll get more practice with this idea on future homeworks and in future Sections, but today we are going to talk about a different way to find time-dependent solutions to the Schrodinger equation—specifically, we'll try to integrate the Schrodinger equation *directly* without relying on a prior determination of the energy eigenstates. We are doing this so you can get more exposure to numerical solutions of differential equations and learn some intuition about how free particles move through different potentials.

The specific method we will use is called the Crank-Nicolson method. It is probably one of the simplest possible ways to solve the TDSE; it is not particularly efficient, but it is good enough for our purposes right now.

1.2 Deriving the Crank-Nicolson Method

Recall from class that the TDSE is given by

$$i\hbar\frac{\partial\Psi}{\partial t} = -\frac{\hbar^2}{2m}\frac{\partial^2\Psi}{\partial^2 x} + V(x)\Psi. \tag{1.1}$$

The right-hand side is exactly the same as the Hamiltonian terms involved in the TISE, but now we also need to handle the temporal derivative.

Let's use j as a subscript index to discretize space, with step size Δx, and n as a superscript index to discretize time, with step size Δt.

Question for the class: How should we fill in the following equations?

$$\frac{\partial^2\Psi}{\partial^2 x} \rightarrow \frac{\Psi_{j+1} - 2\Psi_j + \Psi_{j-1}}{\Delta x^2} \tag{1.2}$$

$$\frac{\partial\Psi}{\partial t} \rightarrow \frac{\Psi^n - \Psi^{n-1}}{\Delta t} \tag{1.3}$$

Once we have the discretization formulas, we can plug these back into Equation 1.1. But here's a question: *at which time index do we evaluate the right-hand side?*

1.2.1 Forward-Difference Approximation

The obvious choise might be to use time $n-1$ on the RHS, since then we have an equation that gives Ψ_j^n in terms of Ψ_{j-1}^{n-1}, Ψ_j^{n-1}, and Ψ_{j+1}^{n-1}. However, this has a problem! It introduces errors that can grow exponentially in time. That is NOT good.

Question for the class: Can you see how you might convince yourself this is true?

1.2.2 Backward-Difference Approximation

Ok, so we can't use the simplest solution and still obtain a stable method. What if we express Ψ_j^n in terms of Ψ^{n+1}? That's a little funny because now the equation relates Ψ_j^n to a bunch of values that haven't been determined yet! It's actually not a huge deal because we can in principle solve a system of (linear) equations to obtain the wavefunctions. But this method actually doesn't conserve probability. That is also NOT good.

Question for the class: Can you see how to convince yourself about the issues with the backward-difference method? (This one is harder at this stage in the course... but (if this makes sense to you from your previous QM knowledge) you should think about the idea of unitary evolution.

1.3 Crank-Nicolson Method

It turns out, we can play a clever trick developed by John Crank and Phyllis Nicolson. We can *average* the forward- and backward-difference approximations together. It turns out this solves the problems associated with stability and probability conservation, and it's pretty easy to code. In component form, this looks like:

$$i\hbar\frac{\Psi_j^n - \Psi_j^{n-1}}{\Delta t} = -\frac{\hbar^2}{2m}\frac{(\Psi_{j+1}^n - 2\Psi_j^n + \Psi_{j-1}^n) + (\Psi_{j+1}^{n-1} - 2\Psi_j^{n-1} + \Psi_{j-1}^{n-1})}{2\Delta x^2} + V_j\frac{\Psi_j^n + \Psi_j^{n-1}}{2}. \tag{1.4}$$

If we sort the terms by their time index (index n on the LHS and index $n-1$ on the RHS), we get:

$$(1 + i\Delta tH/2\hbar)\Psi^n = (1 - i\Delta tH/2\hbar)\Psi^{n-1} \tag{1.5}$$

This is great. It means that if we build a matrix representation of the Hamiltonian (which we already know how to do!) we can propagate the wavefunction in time by multiplying by

$$C = (1 + i\Delta tH/2\hbar)^{-1}(1 - i\Delta tH/2\hbar), \tag{1.6}$$

where C is the Crank-Nicolson matrix that advances by one time step:

$$\Psi^n = C\Psi^{n-1}. \tag{1.7}$$

So long as H itself doesn't depend on time, we only need to compute the matrix C once, which makes the algorithm more efficient.

1.4 Task 1: Write functions to propagate a Gaussian wavepacket

Now let's get to the code. First of all, here are the numerical libraries that will be most useful for you today. If you want to add more, just import them in the box below.

```
# Import the typical numerical and plotting libraries
import time
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
%matplotlib inline
```

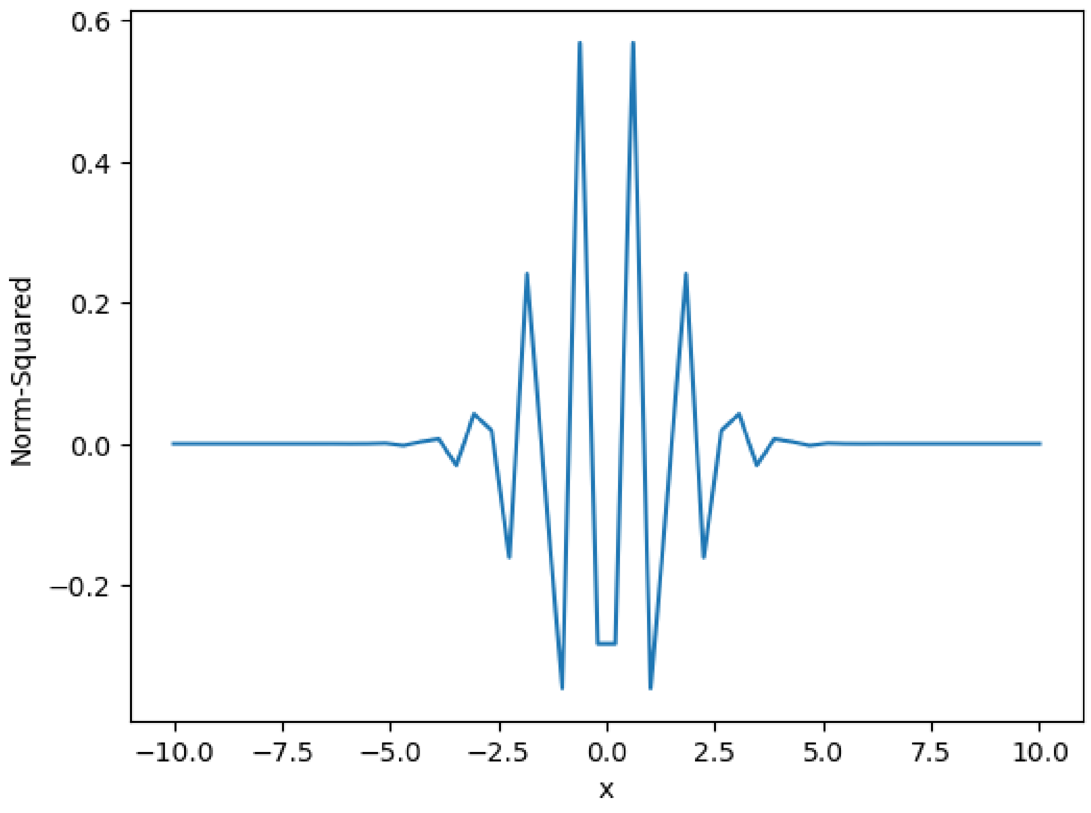
Get started by writing a function `make_gaussian` that creates a Gaussian wavepacket. We wrote the function signature for you, but you'll have to write the body. Make sure that it outputs a vector containing the amplitude of the Gaussian wavepacket at each point in your x grid.

```
def make_gaussian(x, x0, sigmax, k0):
    Norm_psi = 1/np.sqrt(sigmax*np.sqrt(2*np.pi))
    psi = Norm_psi * np.exp(1j*k0*x) * np.exp(-(x-x0)**2/(2*sigmax)**2)
    return psi
```

Make sure your code works by generating and plotting a Gaussian wavepacket.

```
%matplotlib inline
xtest = np.linspace(-10,10)
test_gaussian = make_gaussian(xtest, 0, 1,10)
plt.plot(xtest, test_gaussian)
plt.xlabel('x')
plt.ylabel('Norm-Squared')
```

```
Text(0, 0.5, 'Norm-Squared')
```



Now write the functions to make your Gaussian wavepacket move. In the block below, fill in the functions `make_H` to build the Hamiltonian and return it as a matrix `H` and `make_C` to make the Crank-Nicolson propagation matrix as a matrix `CN`. We need to use Equation 1.6 and Equation 1.7 to do this.

```
# Function to make the Hamiltonian given V (in vector form)
def make_H(V):
    N = np.size(V)
    y = -hbar**2/(2*m*dx**2) # helper variable
    H = np.zeros((N,N))
    d0 = np.ones(N) * (-2*y + V) # this is the main diagonal
    d1 = np.ones(N-1) * (y) # This is the +1/-1 diagonal
    H = np.diag(d0) + np.diag(d1, 1) + np.diag(d1, -1) # put it together
    H[0,N-1] = 0 # I'll use periodic boundary conditions but it shouldn't really matter...
    H[N-1,0] = 0
    return H

# Turn the Hamiltonian into the Crank-Nicolson matrix.
def make_C(V):
    H = make_H(V)
    N = H.shape[1] # Get Hamiltonian dimensions
    z = 0.5*1j*dt/hbar
    CN = np.linalg.inv(np.eye(N) + z*H) @ (np.eye(N) - z*H)
    return CN
```

The last thing to do before running simulations is to implement the time-stepping. Complete the function `simulate` which, using your other functions, builds a normalized Gaussian wavepacket and propagates it in time. You want this function to return a list of the wavefunctions computed at each time step. (Notice how in the function signature we have used *default values*, which is a useful trick you can use in your own code when there are certain values that you might not want to keep resetting over and over...)

```
# Function to run the simulation
def simulate(V, psi0, max_t_step = 350):
    psi=psi0
    # Make the C-N matrices
    CNmat = make_C(V)

    # run simulation
    tlist = dt * np.arange(max_t_step)
    psi_list = np.zeros((tlist.shape[0], psi.shape[0]), dtype=complex)
    for i in range(tlist.shape[0]):
        psi = CNmat @ psi
        psi_list[i,:] = psi

    return psi_list
```

Before moving on, let's run a simple test of your code to make sure things seem to work. We'll do this by simulating the motion of a *free particle*. Do this by running your simulate code on the potential energy function $V(x) = 0$ and watching your particle propagate in space. Is the behavior qualitative correct?

First of all, set up your simulation. We are going to work in a "natural" set of units that are defined for you already below.

```
# Parameters set for you.
hbar = 1 # hbar is 1 here.
m = 0.5 # electron mass

# These are parameters you can play with to adjust the simulation
x0 = -60 # initial position
k0 = np.pi/10 * 2 # Initial average wavenumber NOTE: Max k value is 2*pi/(2*dx)
sigmak = 1/12 * 1/4 # Initial width in k-space
simWidth = 1600 # Width of simulation
dt = 1 # Time step
dx = 1 # Grid step

# ... and these parameters are fixed by the values you set above.
N = simWidth + 1         # Number of grid points
sigmax = 1/(2*sigmak)    # Initial width in position space
windowWidth = simWidth*dx  # Plot window size
dp = 2*np.pi*hbar/(N*dx)   # k-space width of grid point
E0 = hbar**2*k0**2/(2*m)  # Average energy of initial wave packet
velocity0 = np.sqrt(2*E0/m)   # Average velocity of initial wave packet
```

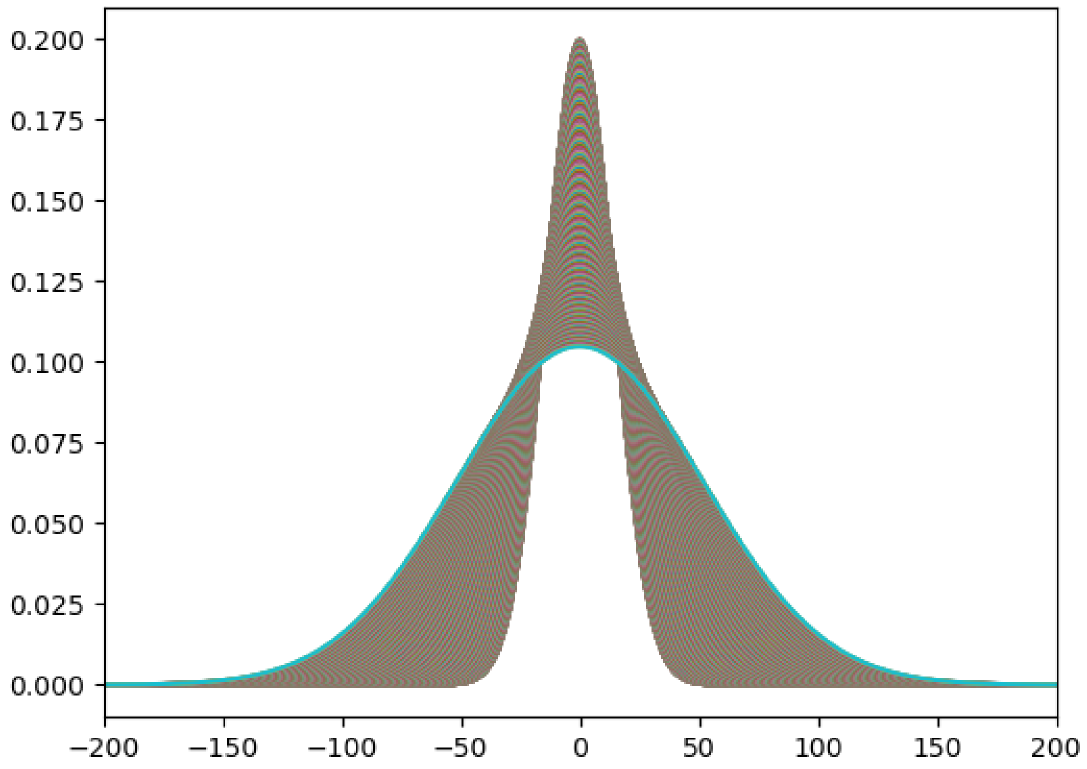
You should define a position-space grid with x values ranging from x_0 to x_N. Make sure to center your well at $x = 0$ in the sense that it extends from `-simWidth/2` to `simWidth/2`. Do that here:

```
x = dx * (np.arange(0,N-1) - simWidth/2)
```

Now, make a Gaussian wavefunction evolving in free space ($V = 0$ everywhere) and with no initial momentum. Plot it and watch it evolve in time from $t = 0$ to $t = 300$ (in our dimensionless units). What does the wavefunction do?

```
test_gaussian = make_gaussian(x, 0, 10, 0.0)
Vs = np.zeros(test_gaussian.shape)
psi_out = simulate(Vs, test_gaussian)

%matplotlib inline
for i in range(psi_out.shape[0]):
    plt.plot(x, np.abs(psi_out[i,:]))
plt.xlim(-200,200)
```

```
(-200.0, 200.0)
```

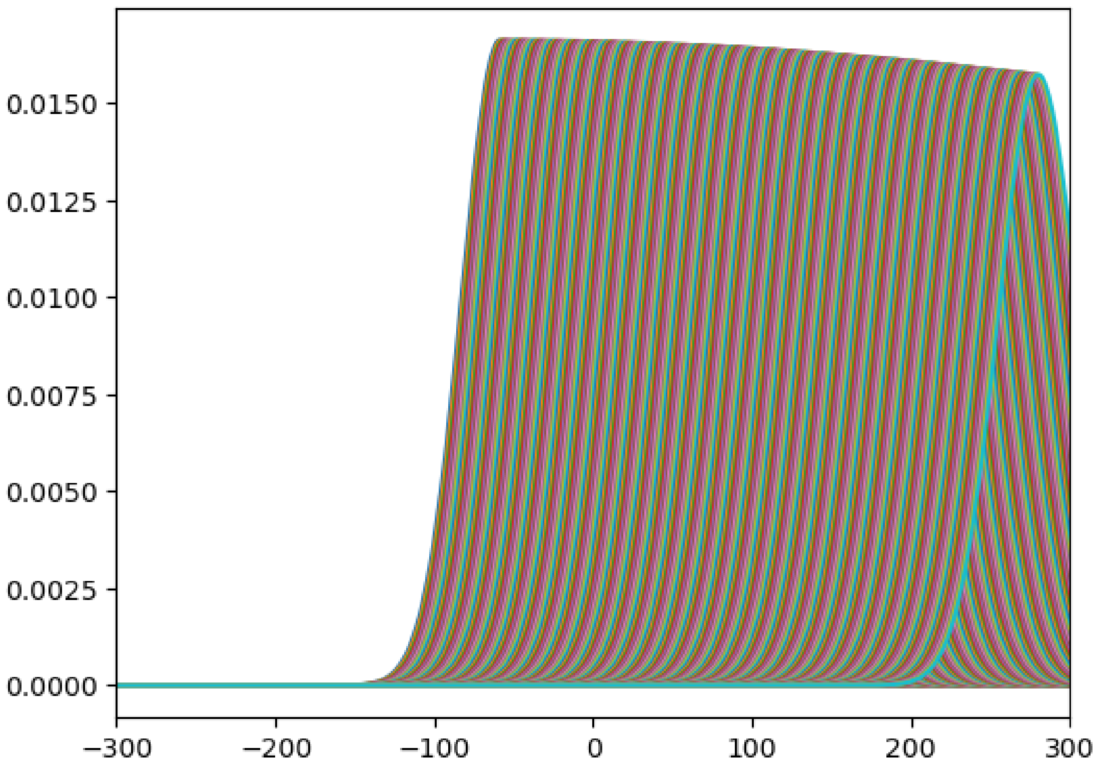


Repeat the question above, but now give the Gaussian wavepacket an initial momentum to the right by setting $k_0 \neq 0$. Now how does the wavefunction evolve?

```
test_gaussian = make_gaussian(x,x0,sigmax,k0)
Vs = np.zeros(test_gaussian.shape)
```

```
psi_out = simulate(0*Vs, test_gaussian, 300)

for i in range(300):
    plt.plot(x, np.abs(psi_out[i,:])**2)
plt.xlim(-300,300)
```

```
(-300.0, 300.0)
```



1.5 An aside on animations

Animating your plots in Python takes a little bit of work, so we have written a little function for you that generates an animation that can be plotted inside the Jupyter notebook. You can use this function by feeding it a list of wavefunctions and (optionally) plot limits. The function will automatically plot the *probability density* associated with your wavefunction list. (You'll need to modify it if you want to see the wavefunction directly…)

```
def make_animation(psi_list, xmin=-100, xmax=100, ymin=0, ymax=0.05, frame_interval=5):
    %matplotlib notebook
    fig, ax = plt.subplots()
    line, = ax.plot([])     # A tuple unpacking to unpack the only plot
    ax.set_xlim(xmin,xmax)
    ax.set_ylim(ymin,ymax)

    def animate(frame_num):
        y = np.abs(psi_list[frame_num,:])**2
        line.set_data((x, y))
        return line

    anim = FuncAnimation(fig, animate, frames=np.shape(psi_list)[0], interval=frame_interv
    return anim
```

As an example of how to use this, let's plot the free particle moving through space:

```
tempvar = make_animation(psi_out, frame_interval=20)
tempvar
plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

2 Task 2: Simulate evolution in a potential

Once it seems like your propagation code is working, let's use it to learn some physics!

We'll look at some more interesting cases, particles moving in actual potentials. For the rest of this Section, you'll work in groups of 2-3 people and each group will choose one of the examples below. At the end of Section, each group will report back to the rest of the class about what they found.

2.1 Project 1: Simulating a quantum mass on a spring

This section is set up to simulate a Gaussian wavepacket evolving in a harmonic potential, $V(x) = \frac{1}{2}\omega^2 x^2$. In the classical case, we know that this gives rise to simple harmonic motion with $x(t) \sim cos(\omega t + \varphi)$. Here you will write code to see if the same thing happens in the quantum world, or to what extent things differ...

And you'll also need to define a potential $V(x)$ for the particle to evolve in. Define that by filling in the code block below. To pick a reasonable value for ω, remember that the value of E_0 is tied to the value of k_0 and we can't simulate momenta with values greater than $\sim 2\pi/2\Delta x$. Try to think about what that means for how you should set ω such that it is compatible with your previously chosen simulation parameters...

```
# set up potential with default values
def Vfunc(a=15, V0=1.5):
    return 1/2 * 30 * E0*(x/np.max(x))**2

Vs = Vfunc(15,1.5)
```

```
plt.plot(x,Vs)
```

```
test_gaussian = make_gaussian(x,x0,sigmax,0)
done=simulate(Vs, test_gaussian, 800)
```

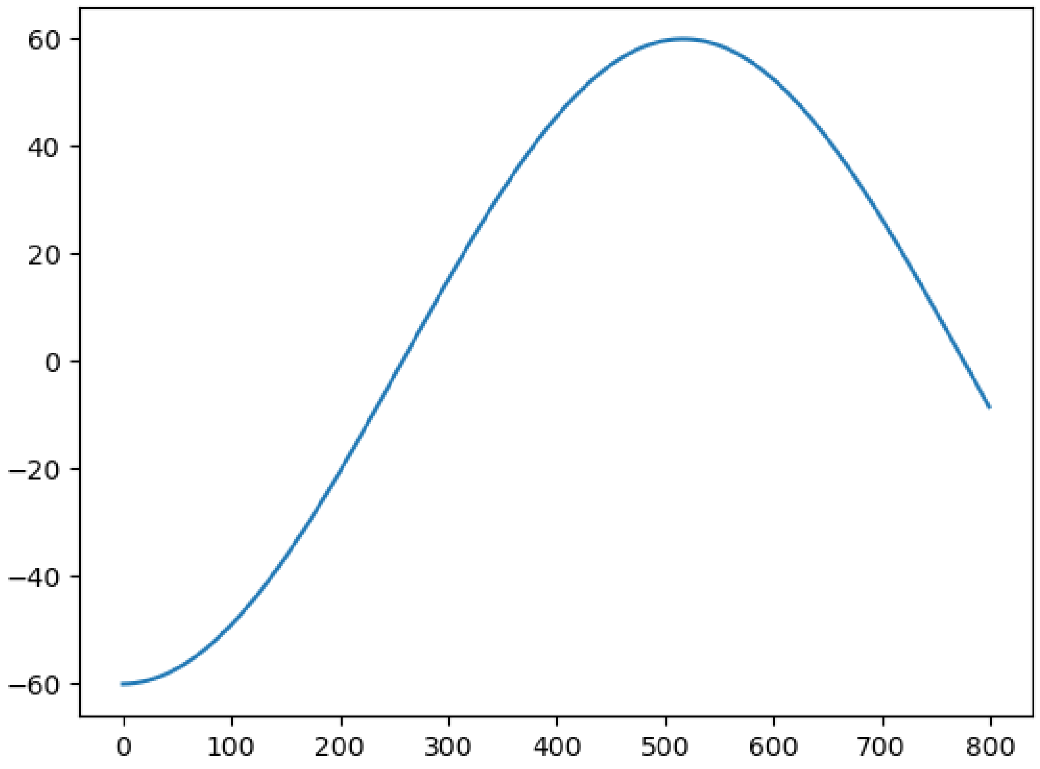
```
aaa=make_animation(done, xmin=-100, xmax=100, ymin=0, ymax=0.09)
plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

Below, write some code to compute the expectation value of x as a function of time and plot it. Does it evolve like you would expect for a classical particle? If it's different in any way, why do you think it behaves this way?

```
xexp = np.zeros(done.shape[0])
for n in range(done.shape[0]):
    xexp[n] = np.sum(x*np.abs(done[n,:])**2)

%matplotlib inline
plt.plot(xexp)
```



Once you have a feeling for how the particle evolves, try *changing* the potential, for example looking at a quartic ($\propto x^4$) potential. How does the evolution of a particle in this potential differ from that of a harmonic potential?

2.2 Option 2: Scattering from Square and Sech-Squared Potential Wells

An unusual aspect of quantum mechanics is that there can be reflection of a particle even from a potential *well*. Here, you will simulate this effect by comparing the scattering of a particle crashing into a square barrier (with positive energy) and a square well (with negative energy). You'll also compare your findings to a particularly interesting potential well, $V(x) \sim -\text{sech}(x)^2$ and uncover an unusual property of this type of well.

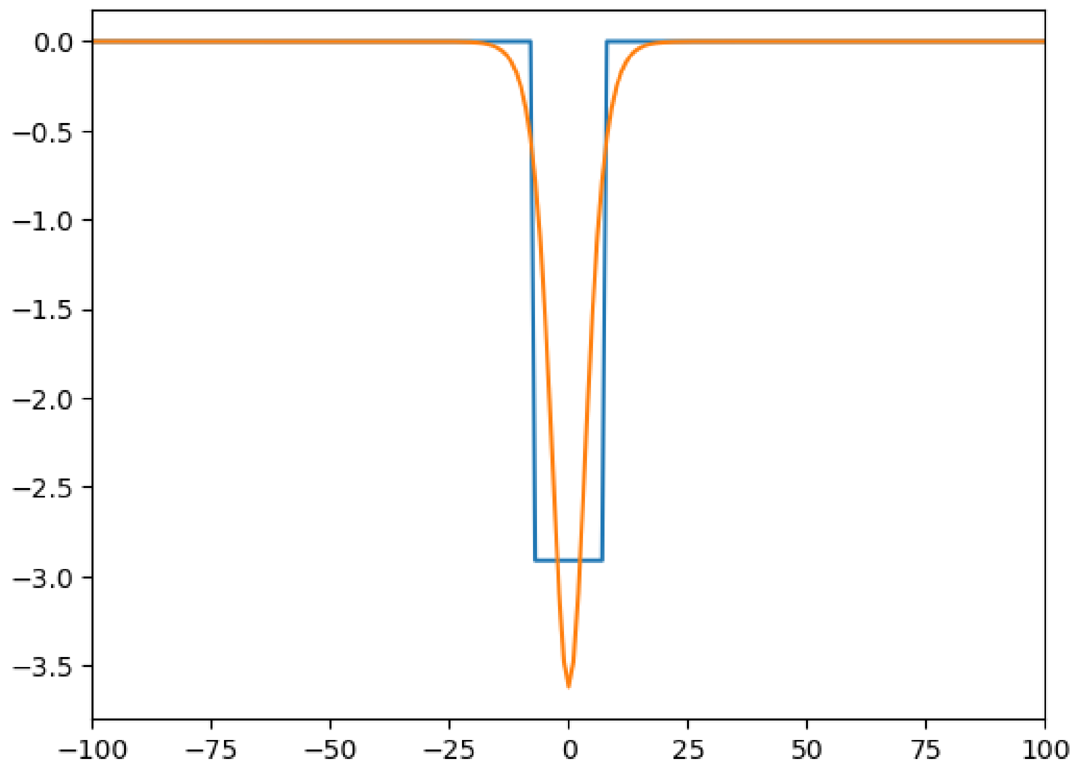
First of all, write some code that defines a square well and the sech-squared well. Note that numpy doesn't have the sech function implemented, but it does have a `cosh` function.

To choose your well depths, pick something that has a width of $\sim 5a - 50a$ and a depth (or height) that is a couple units of E_0. Generate vectors `V` below for the different potentials you want to simulate. Plot them with a y-axis normalized by E_0 to make sure they are a reasonable scale. (And think about why we need to choose this carefully—hint, think about the simulation parameters we have set.)

```
# set up potential with default values
def Vfunc(a=15, V0=1):
    return -1.15 * (np.abs(x) < a)

V_sq = Vfunc(8,6)

def Vfunc(a=15, V0=1):
    return -(hbar**2*V0*(V0+1))/(2*m*a**2)*1/np.cosh(x/a)**2

V_sech = Vfunc(5,5.5)
```

```
plt.plot(x,V_sq/E0)
plt.plot(x,V_sech/E0)
plt.xlim((-100,100))
```

```
(-100.0, 100.0)
```



Now write code to propagate a Gaussian wavepacket through your different potentials.

```
test_gaussian = make_gaussian(x,3*x0,sigmax,k0)

def Vfunc(a=15, V0=1):
    return -(hbar**2*V0*(V0+1))/(2*m*a**2)*1/np.cosh(x/a)**2

Vsech = Vfunc(7,8.75)
psi_out_sech = simulate(Vsech, test_gaussian, 800)

def Vfunc(a=15, V0=1):
    return -1.15 * (np.abs(x) < a)

Vsq = Vfunc(5,5)
psi_out_sq = simulate(Vsq, test_gaussian, 800)
```

Animate your solutions and describe the behavior of a particle that encounters a square well. What happens in this case?

```python
aaa=make_animation(psi_out_sq, xmin=-300, xmax=300, ymin=0, ymax=0.03)
plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
/opt/conda/lib/python3.10/site-packages/matplotlib/animation.py:887: UserWarning: Animation w
  warnings.warn(
```

Then, simulate a particle encountering a sech-squared well and describe the differences. What is the major change in this case? Why do you think the behavior is so different? (Hint: An analogy to optics may be useful.) Why might a potential with that sort of behavior be useful?

```python
aaa=make_animation(psi_out_sech, xmin=-300, xmax=300, ymin=0, ymax=0.03)
plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

3 Conclusion

After completing this Section, you've now learned about numerical solutions of the TDSE, the behavior of a wavepacket in free space, and the behavior of wavepackets evolving in a couple different potentials. You can now use your code to simulate wavepackets interacting with arbitrary potentials! You will use code like what you wrote here to solve problems on HW 2 in which you study the interactions of wavepackets with multi-well potentials as a simple model of particles moving through crystals.

4 Section 4: Discrete Variable Representation

Last week, we studied the connection between position and momentum space using Fourier transforms. We learned that we can work in either "position space" with basis states $\langle x'|x\rangle = \delta(x - x')$, or in "momentum space" with basis states $\langle k'|k\rangle = \delta(k - k')$. We also showed that these kets are connected by the relation

$$\langle k|x\rangle = (2\pi)^{-1/2} e^{-ikx}. \tag{4.1}$$

In this week's section, we will use what we learned to build and solve matrix representations of the Hamiltonian for arbitrary 1D potentials using a method called the "Discerete Variable Representation" (DVR). This is a method that is widely used in physics and chemistry to solve for the states and energies of potential energy surfaces needed to describe molecular vibrations, chemical reactions, and beyond.

Learning Goals: After this Section you should be able to: - Understand when it is more natural to work in a "position" vs. "momentum" basis - Use the sinc basis to express the Hamiltonian for an arbitrary 1D potential - Find the eigenvalues and eigenvectors of Hamiltonians for arbtirary potentials using the discrete variable representation

4.1 Motivating the Sinc-Basis DVR

As with previous problems, we'll first introduce a grid of N equally spaced points on the domain $[x_{\min}, x_{\max}]$. The points can be specified as

$$x_j = x_{\min} + (j - 1)\Delta x \tag{4.2}$$

where

$$\Delta x = \frac{x_{\max} - x_{\min}}{N - 1}. \tag{4.3}$$

Next, we want to set up some basis $\{|\phi_i\rangle\}$ that allows us to express our wavefunction as a superposition of convenient basis functions. Putting this into math based on last week's section, we want a way to implement the expansion

$$\psi(x) = \langle x|\psi\rangle = \sum_{i=1}^{n} c_i \langle x|\phi_i\rangle = \sum_{i=1}^{n} c_i \phi_i(x). \tag{4.4}$$

There are many possible reasonable choices of this basis, and there is a vast literature of DVR methods that use different choices (some that can't even be written as nice analytical functions!). But let's think about what we would want in a general DVR basis set.

Question: What nice properties might we want?

Answer: Some example of useful properties would be: - We want the basis to satisfy $\langle x_i | x_j \rangle \propto \delta_{i,j}$ - It would be nice if the basis functions were "well behaved" so that we can evaluate derivatives, Fourier transforms, etc.

One example of a set of functions that satisfy these desiderata are the *sinc functions*:

$$\phi_j(x) = \frac{1}{\sqrt{\Delta x}} \frac{\sin[\pi(x - x_j)/\Delta x]}{\pi(x - x_j)/\Delta x} = \frac{1}{\sqrt{\Delta x}} \mathrm{sinc}[\pi(x - x_j)/\Delta x]. \tag{4.5}$$

First of all, let's explore why these basis functions are useful. It's best to start this discussion graphically, so let's use some code to implement the basis and plot the basis functions.

```python
import numpy as np
import matplotlib.pyplot as plt
```

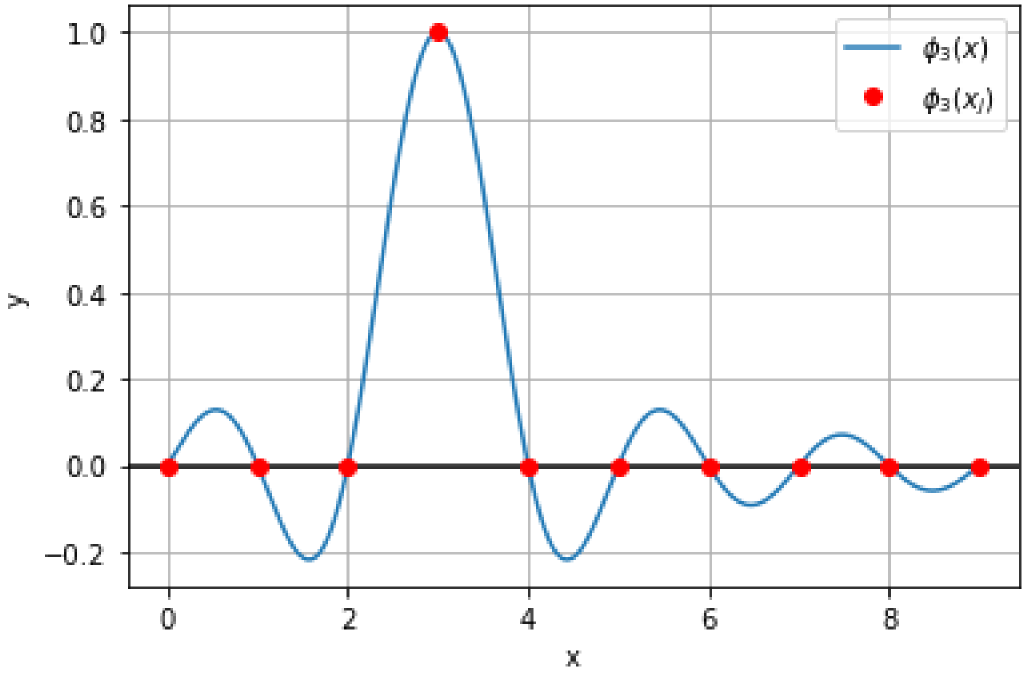
In the space below, we give you some code that defines a grid of x points and a function that implements the sinc-basis. Use these pieces of code to explore the sinc-basis functions and learn about their properties. You are looking to answer questions like the following: - What is the value of a basis function ϕ_j at grid point x_j? - What is the value of a basis function ϕ_j at a grid point x_{j+1}? At x_{j+2}? - In general, how do the basis functions relate to one another?

```python
# Parameters that we set
xmin = 0
xmax = 9
N = 10

# Grid spacing and position-space grid points
Delta_x = (xmax-xmin)/(N-1)
xs = np.linspace(xmin,xmax,N)
xs_fine = np.arange(xmin,xmax,Delta_x/1000)

# Definition of the sinc-basis functions
def phi(x,xj):
    return 1/np.sqrt(Delta_x) * np.sinc((x-xj)/Delta_x)
```

There are two useful ways that we can look at these basis functions. First of all, we can plot an example like $\phi_3(x_j)$ to look at the $i = 3$ basis function evaluated at each grid point. If we overlay this with a plot of $\phi_3(x)$ evaluated on a much finer grid, we see that the sinc-basis is defined such that a given basis function ϕ_i is nonzero at grid point x_i, and its oscillations ensure that it passes through zero at every other grid point!

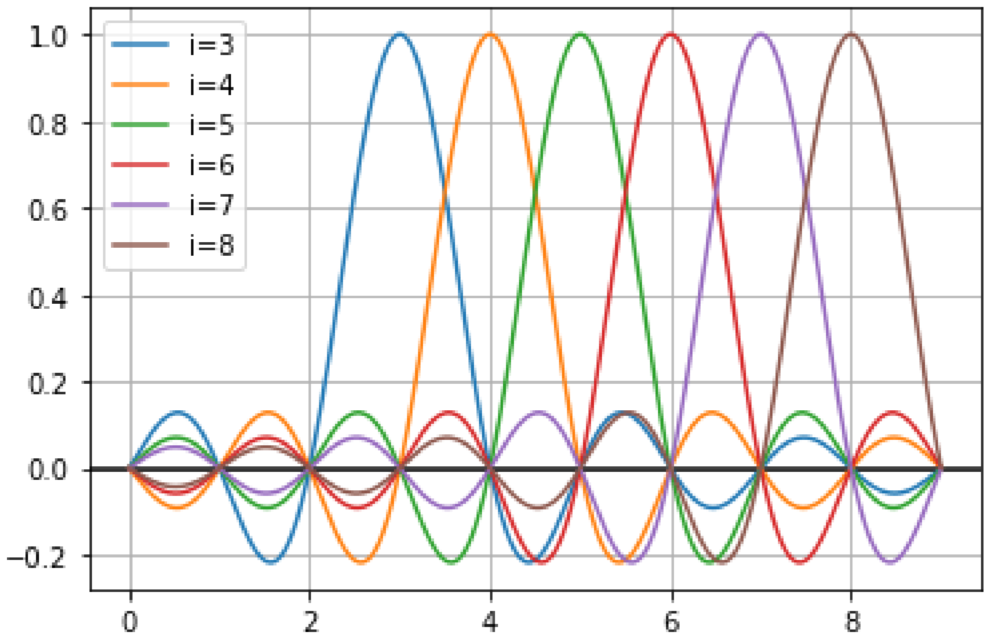
Question: Does it matter that the sinc function takes on nonzero values away from the $i \neq j$ grid points?

```
plt.axhline(0,color='k')
plt.plot(xs_fine, phi(xs_fine, xs[3]), label=f"$\phi_3(x)$")
plt.plot(xs,phi(xs,xs[3]),'ro', label=f"$\phi_3(x_j)$")
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.grid()
```



It can also be useful to look at a series of these sinc functions centered at different grid points. Notice where they all pass through zero, and where each individual basis function doesn't pass through zero.

```
plt.axhline(0,color='k')
for j in range(3,9):
    plt.plot(xs_fine, phi(xs_fine, xs[j]),label=f"i={j}")
plt.legend()
plt.grid()
```



4.1.1 Interpreting the Sinc-Basis

Question: In light of these plots, how can we interpret the sinc-basis functions as compared to a set of Dirac delta functions?

Answer: This is a finite resolution basis that is like the closest possible approximation to a delta function that we can make for a given range on the number of momentum basis functions that our spatial grid can accomodate. As the plots above show, we means this in the sense that these basis states satisfy

$$\phi_j(x_k) \propto \delta_{k,j}, \tag{4.6}$$

so a given basis function only gives a nonzero contribution to a function at the point at which that basis function is centered.

The sinc basis has a nice relationship to the Fourier transforms that we discussed last week. In particular, the sinc basis functions can be written as follows:

$$\phi_j(x) = \frac{\sqrt{\Delta x}}{2\pi} \int_{-\pi/\Delta x}^{\pi/\Delta x} e^{ik(x-x_j)} dk \tag{4.7}$$

Question: Can you justify this form qualitatively? How does it relate to Fourier transforms that we studied last week? And why are the limits on the integral those numbers?

4.2 Deriving the Kinetic Energy Matrix

Now that we have a useful basis, we can try to solve problems with it. We still need to learn how to express the Hamiltonian in this basis. Since the Hamiltonian is

$$H = -\frac{\hbar^2}{2m}\frac{d^2}{dx^2} + V(x) \tag{4.8}$$

we need to figure out how to express the second derivative operator and the $V(x)$ operator in our chosen basis.

Let's start with the kinetic energy term. We want to derive a kinetic energy matrix with matrix elements

$$T_{ij} = -\frac{\hbar^2}{2m}\left\langle \phi_i \left| \frac{d^2}{dx^2} \right| \phi_j \right\rangle . \tag{4.9}$$

It turns out this is where the Fourier transform expression is super helpful. We can go through the following steps:

$$\int_{-\infty}^{\infty} \phi_i^*(x)\frac{d^2}{dx^2}\phi_j(x)dx = \frac{\Delta x}{(2\pi)^2}\int_{-\infty}^{\infty} dx \int_{-\pi/\Delta x}^{\pi/\Delta x} e^{-ik(x-x_i)}dk \int_{-\pi/\Delta x}^{\pi/\Delta x}\left(\frac{d^2}{dx^2}e^{ik'(x-x_j)}\right)dk' \tag{4.10}$$

Because we've expressed the basis function in terms of plane waves, evaluating $\frac{d^2}{dx^2}$ is really easy: it just corresponds to pulling down a factor of $(ik)^2$. That gets us to

$$\frac{\Delta x}{(2\pi)^2}\int_{-\pi/\Delta x}^{\pi/\Delta x} (ik)^2 e^{ik(x_i - x_j)}dk. \tag{4.11}$$

If you work through these integrals, you will find that:

$$\boxed{T_{i,j=i} = \frac{\hbar^2}{2m\Delta x^2}\frac{\pi^2}{3}} \tag{4.12}$$

and

$$\boxed{T_{i,j\neq i} = \frac{\hbar^2}{2m\Delta x^2}\frac{2(-1)^{i-j}}{(i-j)^2}} \tag{4.13}$$

Task: Now fill in the code below to make a function that builds the kinetic energy operator and the potential energy operator.

```
# work in some convenient units
hbar = 1
m = 1
```

```
# Function to make the kinetic energy operator
def make_T(x):
    Delta_x = x[1]-x[0]
    N = xs.shape[0]
    Tmat = np.zeros((N,N))

    # now loop over kinetic energy matrix and fill in matrix elements
    for i in range(N):
        for j in range(N):
            if i==j:
                Tmat[i,j] = (hbar**2/(2*m*Delta_x**2)) * (-1)**(i-j) * (np.pi**2)/3
            else:
                Tmat[i,j] = (hbar**2/(2*m*Delta_x**2)) * (-1)**(i-j) * 2/(i-j)**2

    return Tmat
```

4.3 Deriving the Potential Energy Matrix

For the potential energy matrix, we need to express $V(x)$ in matrix form. Since $V(x)$ can be expanded as a power series in the position operator $\hat{x}$, let's first look at the matrix form of $\hat{x}$ itself. We want the matrix elements $x_{ij} = \langle\phi_i|x|\phi_j\rangle$. In other words, we want

$$\int_{-\infty}^{\infty} \phi_i(x)^* x \phi_j(x) dx = \frac{\Delta x}{(2\pi)^2} \int_{-\pi/\Delta x}^{\pi/\Delta x} dk \int_{-\pi/\Delta x}^{\pi/\Delta x} dk' e^{ikx_i - ik'x_j} \int_{-\infty}^{\infty} x e^{ix(k'-k)} dx. \tag{4.14}$$

In the last term, note that we can replace x by $\frac{d}{d(ik')}$ and then pull the derivative outside of the integral over x (since it doesn't depend on x anymore). Working through the integrals after that step, we get to

$$x_{ij} = \frac{\Delta x}{2\pi} x_i \int_{-\pi/\Delta x} \pi/\Delta x dk' e^{ik'(x_i - x_j)} = x_i \delta_{i,j}. \tag{4.15}$$

The important conclusion is that $\hat{x}$ is diagonal in this basis! So the potential energy, which can be expanded in powers of x is also diagonal in this basis! Knowing this, we can just make the potential energy operator by building a diagonal matrix from $V(x)$. The function make_V does this, and then the function make_H just calls the kinetic and potential energy functions and puts them together.

```
# Function to make the potential energy operator
def make_V(x,Vfunc):
    Vmat = np.diag(Vfunc(x))
    return Vmat

# Function to make the full Hamiltonian
def make_H(x,Vfunc):
    return make_T(x) + make_V(x,Vfunc)
```

4.4 Testing the DVR code

That was a lot of work to get our Hamiltonian matrix. But now let's see the payoff! We'll see that the DVR code is incredibly accurate, especially compared to the finite differences method that we used before. As always let's test our code by using a harmonic oscillator potential. Let's start with a 20 point grid and see how the calculations do for energies and wavefunctions.

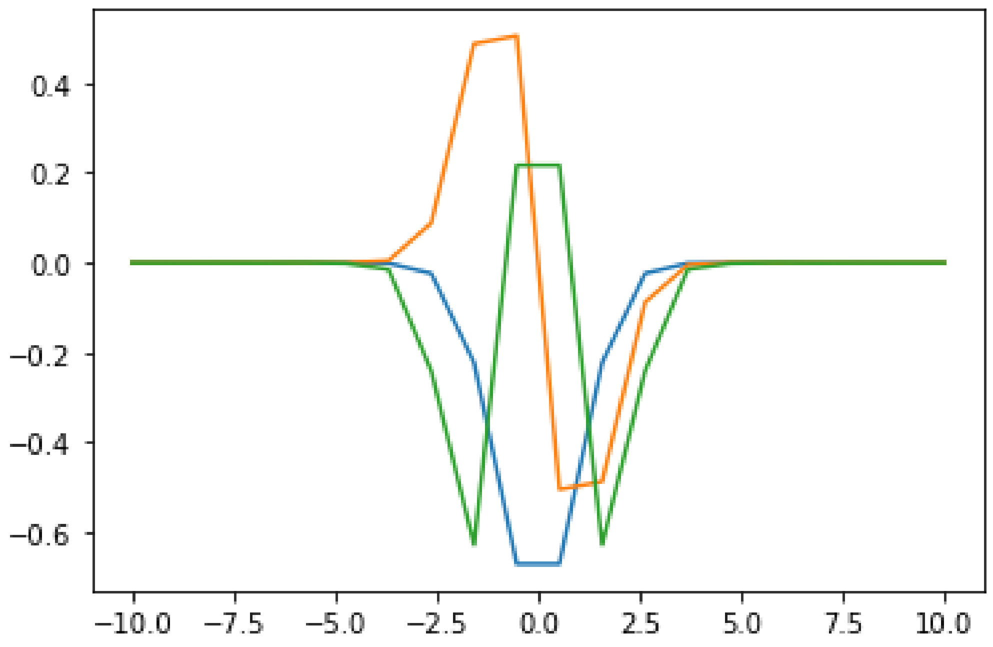
```
xs = np.linspace(-10,10,20)
```

```
def Vharmonic(x):
    return 0.5*x**2
```

```
Ham = make_H(xs,Vharmonic)
vals, vecs = np.linalg.eigh(Ham)
vals[0:4]
```

```
array([0.50042652, 1.49199271, 2.54377448, 3.30864454])
```

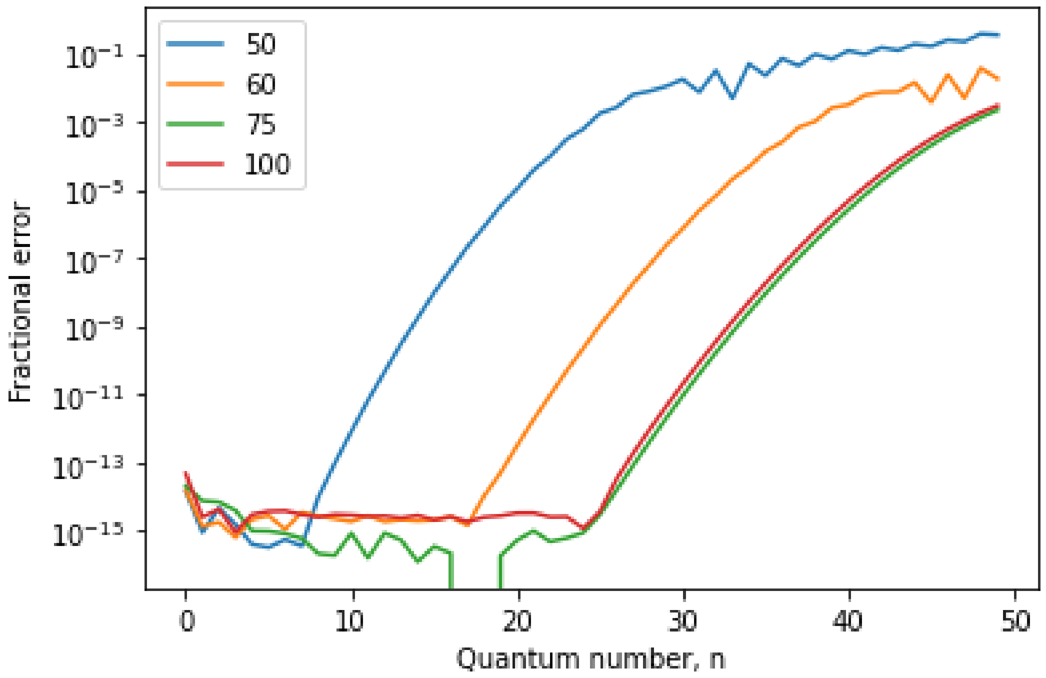
```
plt.plot(xs,vecs[:,0:3])
```



Let's look at the accuracy of our solutions as a function of quantum number.

```
plt.figure()

Nlist = [50,60,75,100]
for i in range(len(Nlist)):
    dx = 20/Nlist[i]
    xs = np.linspace(-10,10,Nlist[i])
    Ham = make_H(xs,Vharmonic)
    vals, vecs = np.linalg.eigh(Ham)
    energies_exact = 0.5 + np.arange(0,len(xs),1)
    errors = np.abs(vals - energies_exact)/energies_exact
    plt.semilogy(np.arange(0,50,1),errors[0:50])

plt.legend([Nlist[i] for i in range(len(Nlist))])
plt.xlabel("Quantum number, n")
plt.ylabel("Fractional error")
```

```
Text(0, 0.5, 'Fractional error')
```



Question: Explain the differences between the case with 50 grid points and the case with 60 grid points. Why is the agreement better to higher n for 60 grid points? Also, why does the agreement seem to always get worse above $n \sim 25$, even if the number of grid points is increased?

4.4.1 Comparing DVR to Finite Differences

Let's look at the accuracy of the numerical solution as a function of the number of grid points. We can compare this to the results of HW 1 to see how the DVR method does against finite differences.

In this block, write a function that computes the fractional error in the SHO eigenstates as a function of the number of grid points used.

```
Nlist = np.logspace(1,2.5,10)
errorlist = np.zeros(len(Nlist))
for i in range(len(Nlist)):
    N = Nlist[i]
    xs = np.arange(-10,10,20/Nlist[i])
    H = make_H(xs,Vharmonic)
    evals, evecs = np.linalg.eigh(H)
    errorlist[i] = np.abs(evals[0] - 0.5)/0.5
```

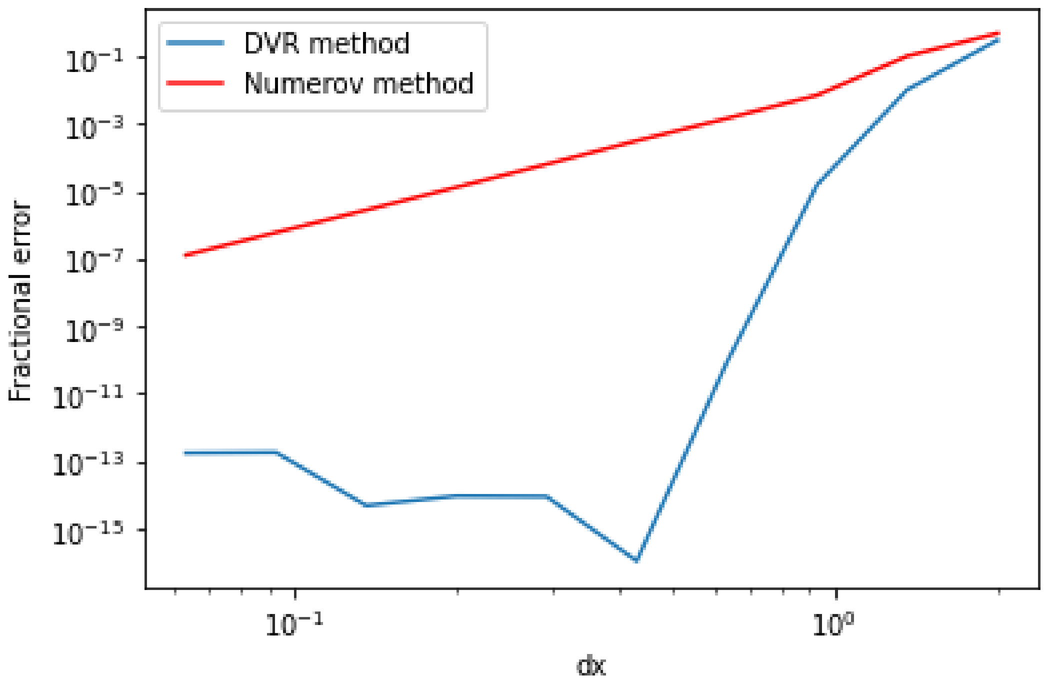
This next block is already written for you– nothing you need to do!— to implement and solve the same problem using the Numerov method. The code is taken from the HW1 solutions.

```
# Function to make the A matrix:
def makeA(N,h):
    A = np.zeros((N,N))
    for i in range(N):
        A[i,i] = -2.0/h**2 # fill in diagonal
        if i==0: # the first row looks like [-2,1,0,0,0...]/h^2
            A[i,i+1] = 1/h**2
        elif i==N-1: # the last row looks like [...,0,0,1,-2]/h^2
            A[i,i-1] = 1/h**2
        else: # all other rows look like [...0,1,-2,1,0,...]/h^2
            A[i,i+1] = 1/h**2
            A[i,i-1] = 1/h**2
    return A

# Function to make the B matrix:
def makeB(N):
    B = np.zeros((N,N))
    for i in range(N):
        B[i,i] = 10.0/12 # fill in diagonal
        if i==0:
            B[i,i+1] = 1/12
        elif i==N-1:
            B[i,i-1] = 1/12
        else:
            B[i,i+1] = 1/12
            B[i,i-1] = 1/12
    return B

# Function to make the Hamiltonian matrix on a grid of N points between (xmin,xmax), then
# The potential energy Vfunc is an arbitrary function that gets passed in and discretized
def solve(N,xmin,xmax,Vfunc):
    x = np.linspace(xmin,xmax,N) # grid of x values
    h = x[1]-x[0] # grid spacing

    # make the A and B matrices
    A = makeA(N,h)
    B = makeB(N)
    Tmat = -0.5 * np.linalg.inv(B) @ A # This is the matrix representing kinetic energy.

    # put the potential energy onto a grid and then put that into a diagonal matrix
    Vgrid = np.zeros(N)
    for i in range(N):
        Vgrid[i] = Vfunc(x[i])
    Vmat = np.diag(Vgrid)

    # Finally we get the whole Hamiltonian
    Hmat = Tmat + Vmat

    # Now diagonalize the Hamiltonian.
    # eigh returns sorted values and vectors!
    evals, evecs = np.linalg.eigh(Hmat)

    return evals, evecs

# NOW ACTUALLY CALL THE NUMEROV METHOD
vals_numerov, vecs_numerov = solve(1000, -10,10, Vharmonic)
errorlist_numerov = np.zeros(len(Nlist))
for i in range(len(Nlist)):
    N = round(Nlist[i])
    vals_num, vecs_num = solve(N,-10,10,Vharmonic)
    errorlist_numerov[i] = np.abs(vals_num[0] - 0.5)/0.5
```

Now, make a plot of the relative errors comparing the numerov and DVR methods for the ground state as a function of the grid spacing Δx.

```
plt.loglog([20/Nlist[i] for i in range(len(Nlist))],errorlist, label="DVR method")
plt.loglog([20/Nlist[i] for i in range(len(Nlist))],errorlist_numerov, 'r', label="Numerov
plt.legend()
plt.xlabel("dx")
plt.ylabel("Fractional error")
```

```
Text(0, 0.5, 'Fractional error')
```



Clearly the DVR method is much better than the Numerov technique! For reasonable grid sizes, the DVR method gets the ground-state energy with almost 10 orders of magnitude better relative agreement!! (And remember, the Numerov method was already much better than a "plain" second-order finite differences approximation to the Hamiltonian...)

Question: Why do you think the DVR method stops getting more accurate at a certain step size? What does this tell you about setting up simulations?

5 Section 6: An Intuitive Picture of Band Structure

```
import numpy as np
import matplotlib.pyplot as plt
from ipywidgets import interact, FloatSlider
from IPython.display import display
```

In this Section, we are going to explore the emergence of band structure using simple models that you have studied before. We will consider a series of potential barriers (equally spaced) to simulate the periodic potential that electrons experience inside a crystal. By varying the heights of these barriers, we can smoothly transition between two simple limits: (1) a single, wide square well and (2) a series of narrow, uncoupled square wells. Band structure emerges between these two extremes.

Learning Goals After this Section, you should: - Be able to explain the qualitative origins of energy bands - Explain the origin of energy spacings *within* bands as compared to *between* bands - Describe the shape of electronic wavefunctions at the edge of each Brillouin zone - Explain the connection between bound state and scattering methods of characterizing the energy levels of the periodic potential

5.1 Qualitative Solution

Before we dive into numerical solutions, let's think about the following qualitative situation: we have a square well of width L. Within the well, we place a series of barriers that partition the square well into N narrower wells with width a. Each barrier has height β and thickness b.

Let's start by considering the barriers to be infinitely thin (basically, delta functions). Now, as a class, let's work through the following questions:

1. What are the energies and eigenstates of the wide well if the barriers are not present ($\beta = 0$)?
2. When $\beta \to \infty$, what should the wavefunctions look like?

3. As we increase β to take on finite values, what will happen to the wavefunctions? How will they look? What will their ordering be like relative to the cases we just described?
4. What analogies could we make between excitation of different bands and excitations of atoms in a solid?

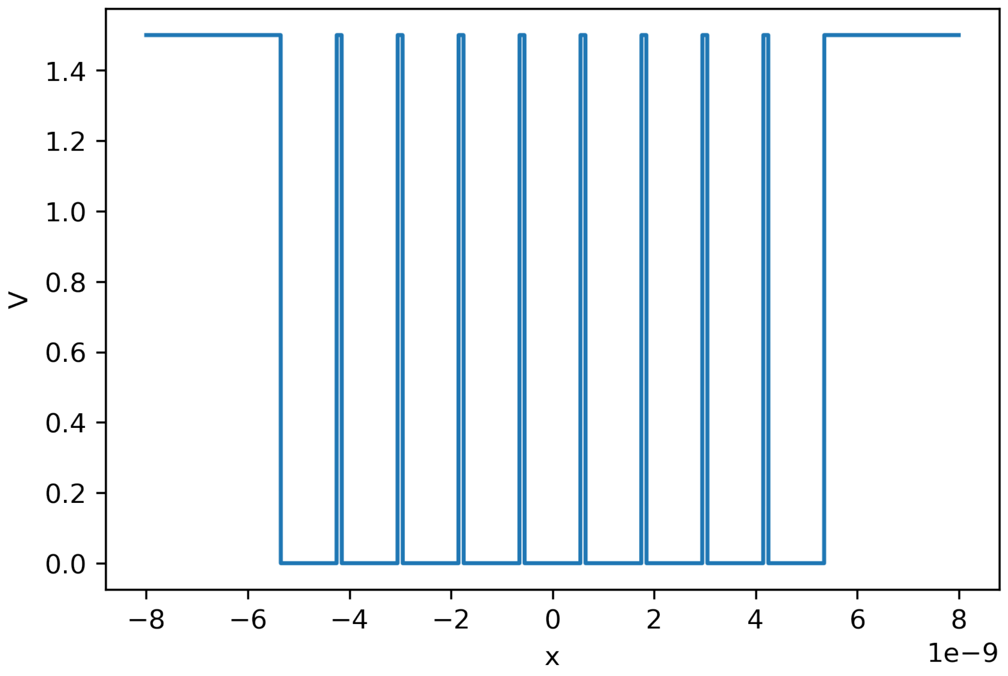
After going through these qualitative arguments, we can now look at what the numerics tell us.

5.2 Building the potential V(x)

First we'll write some functions that build our model potential. Our model includes the following parameters:

- L: width of the enclosing square well
- a: width of each narrow square well
- b: thickness of each internal barrier
- N: number of narrow square wells
- β: heigh of each barrier.

```
Text(0, 0.5, 'V')
```



5.3 Making the DVR code

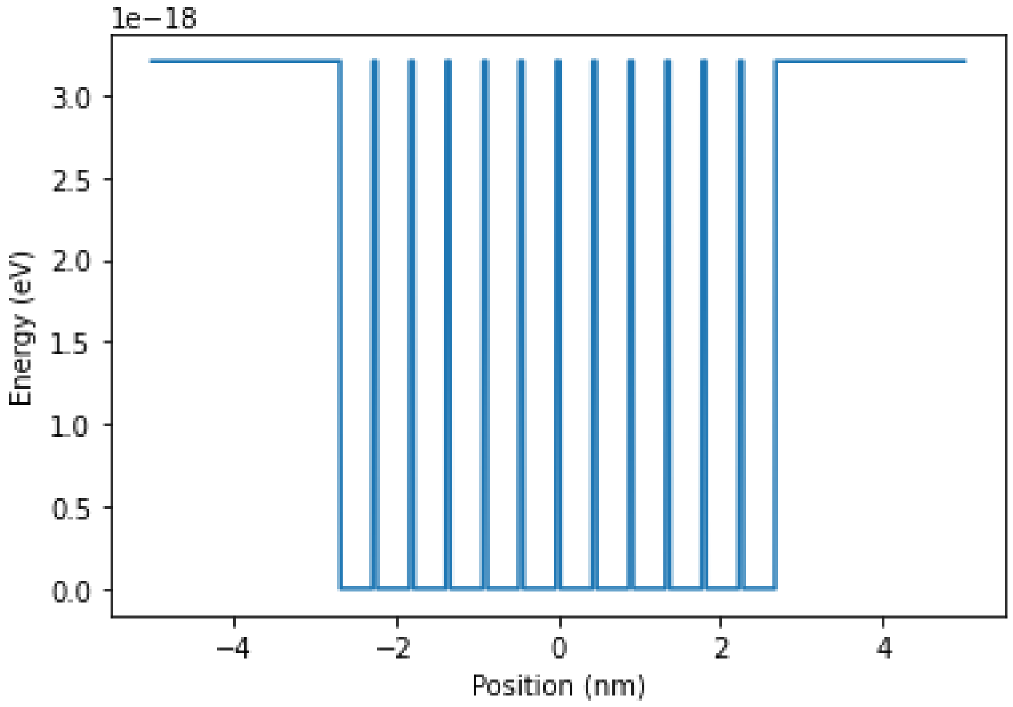
Next, we set up the DVR code from Section 4 to build the Hamiltonian for our system. This code is the same as we have used previously.

```
hbar = (6.63e-34)/(2*np.pi) # in J.s
m = 9.1e-31 # in kg

# Function to make the kinetic energy operator
def make_T(x):
    Delta_x = x[1]-x[0]
    N = x.shape[0]
    Tmat = np.zeros((N,N))

    # now loop over kinetic energy matrix and fill in matrix elements
    for i in range(N):
        for j in range(N):
            if i==j:
                Tmat[i,j] = (hbar**2/(2*m*Delta_x**2)) * (np.pi**2)/3
            else:
                Tmat[i,j] = (hbar**2/(2*m*Delta_x**2)) * (-1)**(i-j) * 2/(i-j)**2

    return Tmat

# Function to make the potential energy operator
def make_V(x,Vfunc):
    Vmat = np.diag(Vfunc(x))
    return Vmat

# Function to make the full Hamiltonian
def make_H(x,Vfunc):
    return make_T(x) + make_V(x,Vfunc)
```

Let's test this on a potential consisting of 12 wells that are fairly deep. After building the potential, we construct the Hamiltonian, evaluate the eigenvectors/eigenvalues, and then plot.

```
N = 12 # number of wells
a = 0.4e-9
b = 0.05e-9 # wall thickness in nm
beta = 20 / 6.24150907e18 # well height in J
V_wells = make_potential(N, a, b, beta)
```

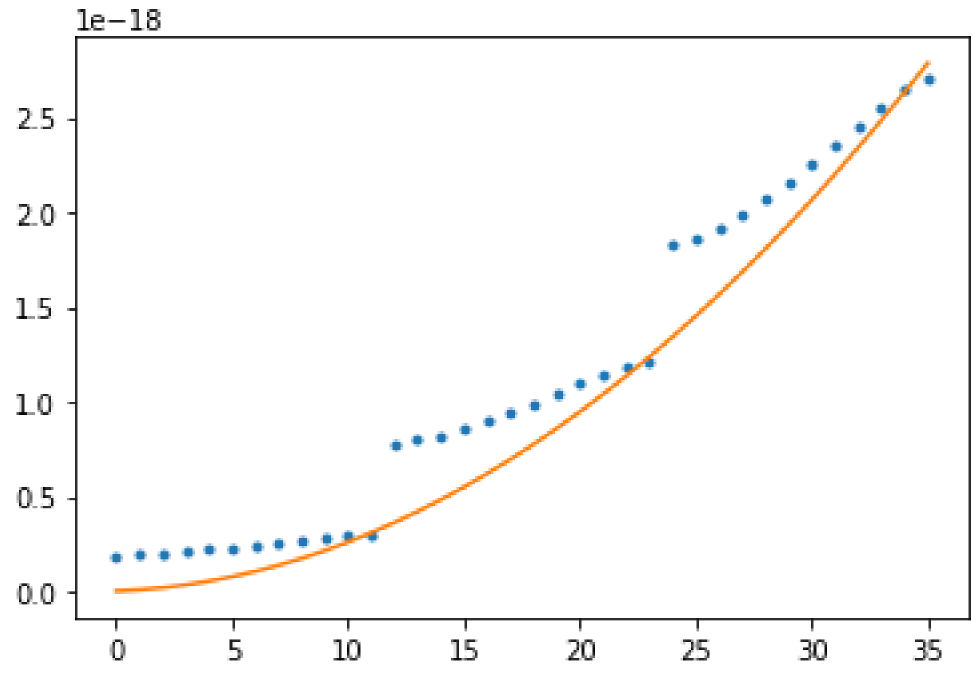
```
xs = np.linspace(-5, 5, 1600)*1e-9
plt.plot(xs*1e9, V_wells(xs))
plt.xlabel("Position (nm)")
plt.ylabel("Energy (eV)")

Ham=make_H(xs,V_wells)

vals, vecs = np.linalg.eigh(Ham)
# make sure ground state is upward going
vecs[:,0] = vecs[:,0] * np.sign(vecs[np.argmax(np.abs(vecs[:,0])),0])
# number of bound states:
nbound = np.sum([vals<beta])
```



We can now plot the numerically computed eigenvalues. Let's overlay these values with the energy levels that would occur if there were no barriers inside the potential—the energy levels of the "wide" finite square well.

```
def analytical_Es(w):
    return [i**2 * np.pi**2 * hbar**2/(2*m*w**2) for i in range(1,nbound+1)]
```

```
plt.plot(vals[:nbound],'.')
plt.plot(analytical_Es(N*a+(N-2)*b))
```



Notice that the periodic potential leads to a structure highly reminiscient of band structure: the energy levels roughly follow the parabolic shape expected from the wide square well (or, from a different perspective, for the free particle), but the levels are broken up into bands with the characteristic shape that we see in class. And there are band gaps between these bands.

Question: What will the plot look like if we increase the well separation to the point that the wells are essentially fully uncoupled? Will there be energy degeneracies? How many levels will share the same energy? What happens as the number of wells is increased–which energy gaps "close" and which remain?

To answer these questions, perform a "numerical experiment" and compare your results to the appropriate square well. How do the numbers compare?

Question: What happens if the well separation is decreased to essentially zero? How do these energies compare to the appropriate finite square well? What about the comparison to free particle energies?

Let's look at the wavefunctions to understand this even better. It will be useful to plot the actual wavefunctions and have some interactivity. The code below makes a plot with a slider that allows you to look at all of the bound eigenstates for a given potential. Once you set the number of wells, their width, and their barriers, you can scan through the eigenstates and look at the behavior.

```
def interact_wavefunction():
    ## Plot parameters
    xmin, xmax, nx = min(xs)*1e9, max(xs)*1e9, 50
    ymin, ymax = -1.2, 1.2
    pmin, pmax, pstep, pinit = 0, nbound-1, 1, 0

    ## Set up the plot data
    fig, (ax1,ax2)   = plt.subplots(1,2,figsize=(16,6))
    line2, = ax1.plot(xs*1e9,2*V_wells(xs)/np.max(V_wells(xs))-1, color="grey")
    line, = ax1.plot([], [], linewidth=2, color="red") # Initialize curve to empty data.

    well_levels, = ax2.plot([],[])
    well_itself, = ax2.plot([],[], color="black")

    ## Set up the figure axes, etc.
    ax1.set_xlim([xmin, xmax])
    ax1.set_ylim([ymin, ymax])
    ax2.set_xlim([xmin,xmax])
    ax2.set_ylim([-beta/8*6.24150907e18,1.2*beta*6.24150907e18])
    ax1.set_xlabel('Position (nm)', fontsize=18)
    ax1.set_ylabel('Re{$\psi$}', fontsize=18)
    ax2.set_xlabel('Position (nm)', fontsize=18)
    ax2.set_ylabel('Energy (eV)', fontsize=18)
    ax1.tick_params(axis='both', which='major', labelsize=16)
    ax2.tick_params(axis='both', which='major', labelsize=16)
    plt.close()      # Don't show the figure yet.

    ## Callback function
    def plot_wavefunction(Eigenstate):
        y = vecs[:,int(Eigenstate)]/np.max(np.abs(vecs[:,int(Eigenstate)]))
        line.set_data(xs*1e9, y)
        line2.set_data(xs*1e9, 2*V_wells(xs)/np.max(V_wells(xs))-1)
        well_itself.set_data(xs*1e9, V_wells(xs)*6.24150907e18)
        [ax2.axhline(y=vals[i]*6.24150907e18,color="grey") for i in range(nbound)]
        ax2.axhline(y=vals[int(Eigenstate)]*6.24150907e18, color="cyan")
        ax2.set_title(f"E={np.round(vals[int(Eigenstate)]*6.24150907e18,3)} eV", fontsize=
        display(fig)

    ## Generate the user interface.
    interact(plot_wavefunction,
             Eigenstate=FloatSlider(min=pmin, max=pmax, step=pstep, value=pinit))
```

For example, we can start with 12 wells that are each 0.8 nm wide, 4 eV high, and have a 0.05 nm barrier between them. For this set of parameters, use the interactive plotting to gain some insights about the potential.

```
N = 12 # number of wells
a = 0.8e-9
b = 0.05e-9 # wall thickness in nm
beta = 4 / 6.24150907e18 # well height in J
V_wells = make_potential(N, a, b, beta)
xs = np.linspace(-8, 8, 900)*1e-9

Ham=make_H(xs,V_wells)

vals, vecs = np.linalg.eigh(Ham)
# make sure ground state is upward going
vecs[:,0] = vecs[:,0] * np.sign(vecs[np.argmax(np.abs(vecs[:,0])),0])
# number of bound states:
nbound = np.sum([vals<beta])


interact_wavefunction();
```

```
interactive(children=(FloatSlider(value=0.0, description='Eigenstate', max=32.0, step=1.0), 
```

Question: Using the interactive graphs above, how does the behavior of each wavefunction within a single band conform to our expectations from Bloch's theorem? How do the wavefunctions differ from band to band? What do you expect to happen as $N \to \infty$?

5.4 Transfer matrix approach

There's a totally different way to look at band structure: by studying the scattering properties of the potential. This is a complementary method to learn about a potentials bound states. Scattering studies are widely used in many fields of physics/chemistry, from studying the solid state to studying atoms and molecules. Let's see what we can learn about electronic band structure from this point of view using the transfer matrix program that you wrote on HW2.

First of all, let's go over the idea of the transfer matrix approach.

Set up approach, talk about different role of propagation vs. transmission matrices.

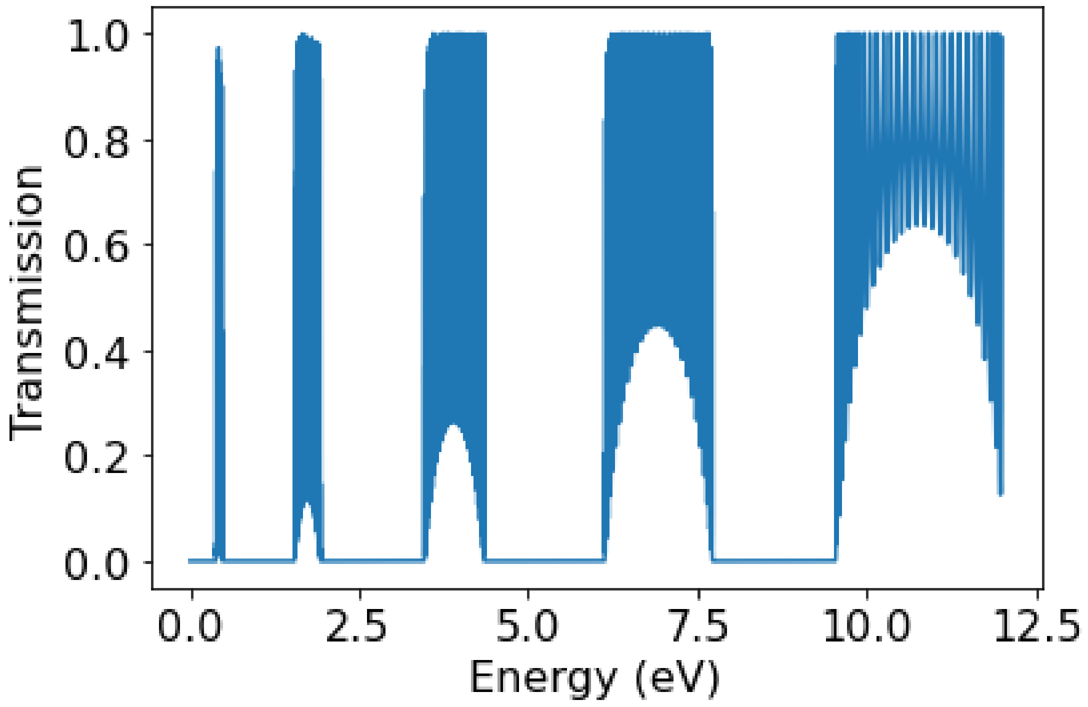
The code to implement each of these matrices is copied below.

```
def DMat(k1, k2):
    res = np.zeros((2,2),dtype=np.complex_)
    res[0,0] = (1 + k2/k1)/2
    res[0,1] = (1 - k2/k1)/2
    res[1,0] = res[0,1]
    res[1,1] = res[0,0]
    return res

def PMat(k, L):
    res = np.zeros((2,2),dtype=np.complex_)
    res[0,0] = np.exp(-1j * k * L)
    res[1,1] = np.exp(1j * k * L)
    return res
```

Question: What do the peaks here represent? What is the physics going on when N=2? N=3? Large N?

```
a = 0.8e-9
b = 0.1e-9

Es = np.arange(0.01, 12.0, 0.001) / 6.24150907e18
width_barrier = b
width_gap = a
Vb=10/ 6.24150907e18

Nwells=30


Ttrans = np.zeros(Es.size)
i = 0
for E in Es:
    klow = np.emath.sqrt(2 * m * E/hbar**2)
    khigh = np.emath.sqrt(2 * m * (E - Vb)/hbar**2)
    res_mat = DMat(klow, khigh) @ PMat(khigh, width_barrier) @ DMat(khigh, klow) @ PMat(kl
    U, V = np.linalg.eig(res_mat)
    diag_res_mat = np.diag([U[0],U[1]])
```

```
        res_mat = np.linalg.matrix_power(diag_res_mat,Nwells)
        res_mat = V @ res_mat @ np.linalg.inv(V)
        Ttrans[i] = 1 - np.abs(res_mat[1, 0])**2 / np.abs(res_mat[0,0])**2
        i = i + 1

plt.plot(Es * 6.24150907e18, Ttrans)
plt.xlabel("Energy (eV)",fontsize=16)
plt.ylabel("Transmission",fontsize=16)
plt.tick_params(axis='both', which='major', labelsize=16)
```



Let's compare the transmission peaks to the bound state energy of our finite square well.

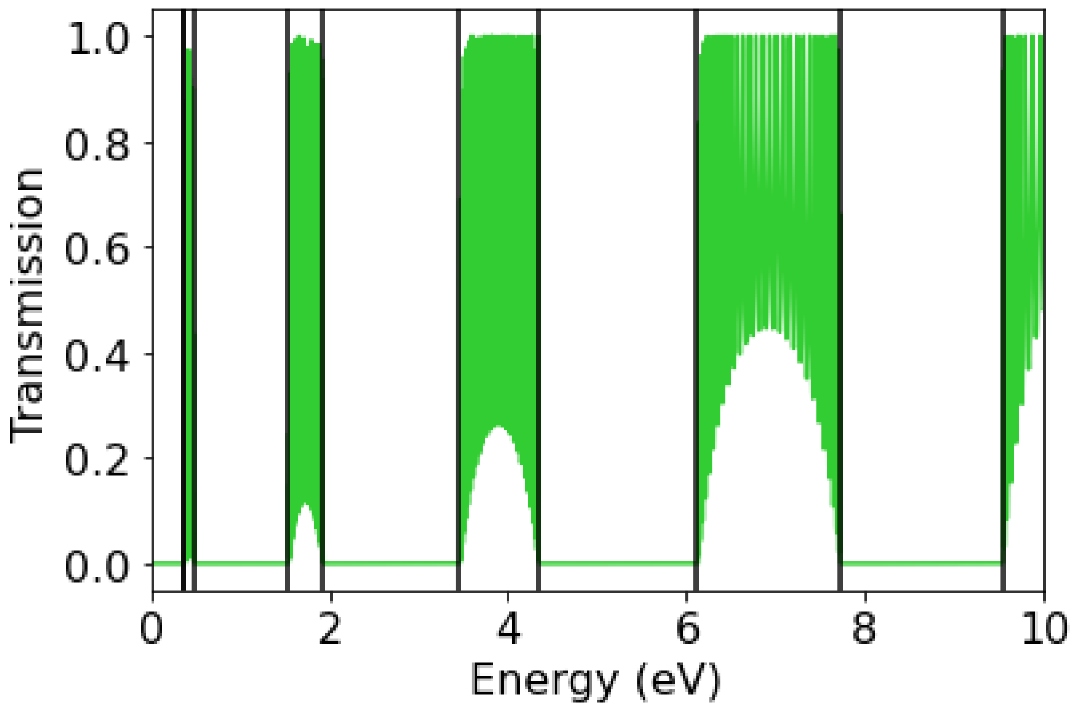
```
V_wells = make_potential(Nwells, a, b, Vb)
xs = np.linspace(-15, 15, 1500)*1e-9

Ham=make_H(xs,V_wells)

vals, vecs = np.linalg.eigh(Ham)
```

Now let's compare the transmission fraction to the band energies.

Question: How can we easily determine the limits of each energy band? (How does it compare to the number of wells being simulated?

```
band_limits = [0]
for n in range(5):
    band_limits.append(n*Nwells-1)
    band_limits.append(n*Nwells)

plt.plot(Es * 6.24150907e18, Ttrans, color="limegreen")
[plt.axvline(vals[i]*6.24150907e18, color="black") for i in band_limits]
plt.xlabel("Energy (eV)",fontsize=16)
plt.ylabel("Transmission",fontsize=16)
plt.tick_params(axis='both', which='major', labelsize=16)
plt.xlim(0,10)
```

```
(0.0, 10.0)
```



References

Bibliography