Module 2: The energy levels of the discretized particle in a box

Numerical Approach

Where we left off with Module 2 was that we could approximate Schrodinger's Equation, $-\frac{\hbar^2}{2m}\frac{\partial^2\psi}{\partial x^2}+V(x)\psi = E\psi$, as a finite-difference equation, $-\psi(x_{j-1})+2\psi(x_j)-\psi(x_{j+1})+\left(\frac{2m(\Delta x)^2}{\hbar^2}\right)V(x_j)\psi(x_j)=\left(\frac{2m(\Delta x)^2}{\hbar^2}\right)E\psi(x_j)$. For a given range through x, discretely broken into Δx steps, we'd have a collection of such equations, one for each x value, and that can be expressed as a single matrix relation,

$$\begin{pmatrix} 2+\tilde{v}(x_1) & -1 & & & & & \\ -1 & 2+\tilde{v}(x_2) & -1 & & & & \\ & & & \cdots & & & \\ & & -1 & 2+\tilde{v}(x_j) & -1 & & \\ & & & & \cdots & & \\ & & & & & -1 & 2+\tilde{v}(x_N) \end{pmatrix}\begin{pmatrix}\psi(x_1)\\ \psi(x_2)\\ \cdots\\ \psi(x_j)\\ \cdots\\ \psi(x_N)\end{pmatrix}=\varepsilon\begin{pmatrix}\psi(x_1)\\ \psi(x_2)\\ \cdots\\ \psi(x_j)\\ \cdots\\ \psi(x_N)\end{pmatrix} \qquad (1)$$

where $\tilde{v}(x)=2m(\Delta x)^2V(x)/\hbar^2$ and $\epsilon\equiv(2\pi\Delta x/\hbar)^2E$.

We are solving the eigenvalue problem outlined in Eq. (1). We are going to ask two questions:

1. Do the energies follow the correct trend, i.e., are they proportional to n^2?
2. Do the energies have the correct values, i.e., are they given by $E_n = n^2\pi^2\hbar^2/2mL^2$?

In order to obtain the energies and answer these questions we must do the following:

a. Construct the matrix on the left-hand side of Eq. (1). For convenience, and to be consistent with the notation used in more advanced treatments of quantum physics, we will call this matrix H for "Hamiltonian".
b. Call an eigenvalue solver to obtain the energies (i.e., the eigenvalues of the matrix) and the associated wavefunctions (i.e., the eigenvector, giving the values of ψ at a set of discrete locations).
c. Arrange the energies in a list from smallest to largest.
d. Make a log-log plot of eigenvalue ϵ vs. energy level number (or "quantum number") to verify the quadratic dependence on n.
e. Make plots of the wavefunctions, to see if they also match any of the predictions of theory.

The sample code used to perform this task is called **DiscretePIB.py** and is listed below in the section titled "Sample Code."

Pseudocode

Pseudocode is often used to describe an algorithm. It’s somewhere between the ‘to do’ list that’s above and the actual, language-dependent code that will execute it. While it looks a bit like computer code, it is usually not written in a specific language. This allows us to focus on clearly representing the algorithm, rather than being distracted by syntax requirements of a particular programming language. In our pseudocode we'll use indenting to help indicate algorithm structure. For example, in the pseudocode the indented lines following "Loop:" are part of the body of the loop.

```
Initialize the number of points N that you will use to describe your system, and the spacing $\Delta x$ between them
Initialize the matrix $H$ to be all zeroes (for convenience)

To initialize the non-zero elements of $H$, write a loop:
Loop: j from 1 to N
    $H(j,j)=2+\tilde{v}(x_j)$
    $H(j,j+1)=H(j,j-1)=-1$ (except in the first and last rows, where we can't have an element $H(1,0)$ or $H(N,N+1)$.
Call your preferred eigenvalue solver for the matrix $H$. Generate an $N\times1$ array $\epsilon$ containing eigenvalues (unitless energies), and an $N\times N$ array $\psi$ containing eigenvectors (un-normalized wavefunctions.)
Order the eigenvalues and corresponding eigenvectors in order of increasing eigenvalues.
Make plots of select eigenvectors.
Make plots of the elements of $\epsilon$, i.e. plot $\epsilon_n$ vs. $n$ on a log-log scale.  Before plotting, divide the elements by either $\epsilon_1$ to check the scaling or $(\pi n/(N+1))^2$ to check quantitative accuracy.
```

Sample Python Code for Particle in a Box

Below is sample code written in Python. It achieves steps 1 through 9 of the psuedocode for a particle in a box (V = 0).

```
from __future__ import division
from numpy import linalg
from pylab import *

N=10                #The input N tells how many elements to divide the 1D box into

H = zeros((N,N))   #First we set up an NxN matrix whose elements are all zero.

# We use a loop to go through all of the other rows and define the non-zero elements
# (there are only 3)
for j in arange(0,N-1 +0.1): #the +0.1 because arange sometimes returns < max value
   if j > 0:                  # for j = 0, there is no column to its left
      H[j,j-1] = -1           #Left of the diagonal
   H[j,j] = 2                 #Diagonal
   if j < N-1:                #for j = N, there is no column to its right
      H[j,j+1] = -1           #Right of the diagonal

# if you want to see what this matrix looks like,
print("H = ", H)

# Now we get the energies and wavefunctions using Pylab's built-in solver
# "eig", which gives the eigenvalues and the eigenvectors
Eigenvalues, Eigenvectors = eig(H)

# Unfortunately, eig() doesn't automatically order its output in order of smallest to largest
# eigenvalue.  The following implements the bubble sort algorithm to arrange the solutions
# by frequency while keeping the eigenvector with the associated eigenvalue.
for i in arange(1,len(Eigenvalues)-2):
   for j in arange(0,len(Eigenvalues)-i):
      if (Eigenvalues[j] > Eigenvalues[j+1]):
         Eigenvalues[j], Eigenvalues[j+1] = Eigenvalues[j+1], Eigenvalues[j]
         for k in arange(0,len(Eigenvalues)):
            Eigenvectors[k,j], Eigenvectors[k,j+1] = Eigenvectors[k,j+1], Eigenvectors[k,j]

print("Eigenvalues = ", Eigenvalues)
print("Eigenvectors = ", Eigenvectors)
```

When you run the program you'll notice that the N eigenvalues simply is a list or an array of values. Meanwhile the N Eigenvectors are given as a list of N arrays, each of which has N elements itself. The first array in list of Eigenvectors represents ψ_1; the first *value* in that list represents $\psi_1(x_1)$, the next value represents $\psi_1(x_2)$ and so forth.

Implicit Boundary Conditions

You may wonder 'how does this code impose the infinite barriers at either end?' It's rather implicitly handled. For the first location considered in the matrix (j=0), there can be no H[0,0-1]=-1 term, though there's still a H[0,0]=2 term and a H[0,0+1]=-1 term. That means the finite-difference equation that describes its motion would have the form

$$2\psi(x_0) - \psi(x_{0+1}) = \left(\frac{2m(\Delta x)^2}{\hbar^2}\right)E\,\psi(x_0)$$

rather than the

$$-\psi(x_{j-1}) + 2\psi(x_j) - \psi(x_{j+1}) = \left(\frac{2m(\Delta x)^2}{\hbar^2}\right)E\,\psi(x_j)$$

for most of the other elements. Comparing the two expressions, it's clear that *not* having a H[0,0-1]=-1 term in the matrix is equivalent to insisting that $\psi(x_{0-1}) = 0$. So that enforces the boundary condition on the left edge of the well. Similarly, for the final location considered in the matrix (j = N-1), there can be no H[N-1,N]= -1 term. That effectively sets $\psi(x_N) = 0$ at the other edge of the well.

Comparing Computational and Analytical Solutions – Assessing the Computational Approach

The code above has no potential term; as such, it's appropriate for the Particle in a Box. The point of testing this approach on the simple Particle in a Box system is that we already know this system's solutions (the eigenvectors / stationary states and eigenvalues / energies), so we can learn about the program's strengths and weaknesses by comparing its outputs with the known solutions. Then we'll know how far to trust the program for more complicated systems (for which we don't *a priori* know the solutions.) Analytically solving the particle-in-a-box, one gets the familiar results presented in sections 2.2 of the text: sinusoidal eigenfunctions (stationary states) and energies $E_n = n^2\pi^2\hbar^2/2mL^2$ (note: Griffiths uses *a* instead of *L* for the box length.) In the following exercises, you'll see under what conditions the computational approach gives results that agree or disagree with the exact, analytically-found solutions.

Exercise 1

Do the energies follow the correct trend, i.e., are they proportional to n^2? If some do and some don't, which do and don't?

a) The eigenvalues are supposed to be related to the energies by $\epsilon(n) \equiv 2m(\Delta x/\hbar)^2 E_n$. Since the factor in parentheses is a constant, if the energies are proportional to n^2, then so must be the eigenvalues. So you can answer this question by plotting the eigenvalues vs. *n* and seeing whether or not their trend is parabolic. If you've not created a plot in Python before, read over Dr. DeWeerd's tutorial at http://bulldog2.redlands.edu/facultyfolder/deweerd/tutorials/Plotting.html. As you'll see there, the three key lines for generating a basic plot are

```
figure()

plot(x values, y values, marker = ‘o’)

show()
```

So you’ll want to add something like this to the end of your program. The “marker = ‘o’” makes the plot not just connect the dots between the points its plotting, but actually mark the individual points too. In your case, the “y values” are the Eigenvalues while the “x values” are their numbers – the first Eigenvalue corresponds to n =1, the next to n = 2, … the last to $n = N$. So above the figure(), plot(…), and show() lines in your code, you’ll want to create the list [1,2,3,…N]. You can use the arange( ) function to do that. For example

```
ns = arange(1,N+0.1)
```

does the job. Note, you need that “+0.1” because the arange function creates a list that starts with the first entry (1 in this case), and adds 1 then 2 then 3,… up to, but *exclusive of* the second entry (N +0.1 in this case) so if you wrote arange(1,N) it would terminate the list at N-1 instead of N.
So, add the required four lines of code to your program and run it so it will plot the eigenvalues for you.

b) To get an even better feel for how good/bad the n^2 dependence is, rather than plotting Eigenvalues, plot Eigenvalues/Eigenvalues[0]; that divides off all constants (literally, replace “Eigenvalues” with “Eigenvalues/Eigenvalues[0]” in the “plot(ns, Eigenvalues, marker = ‘o’) line.)

   Since $\epsilon(\mathrm{n}) = 2m(\Delta x/\hbar)^2 E_n$, and the correct values of energies are $E_n = \mathrm{n}^2\pi^2\hbar^2/2mL^2$, we’d expect $\epsilon(\mathrm{n})/\epsilon(1) = \mathrm{n}^2$ if the program’s eigenvalues really correspond to the particle-in-a-box energies. So, to see how well/poorly the eigenvalues correspond with the expected energies, add another line, just before the “show()” line,

   ```
   plot(ns, ns**2, marker = ‘o’).
   ```

   Having the two “plot” lines between “figure()” and “show()” will generate a single graph with the two different curves plotted on it.

   So, after making these changes to your program, run it again. Roughly, what percent or fraction of the Eigenvalues are pretty good (that is, for what fraction do the two plots overlap)?

Exercise 2

Clearly, the eigenvalues *do* correspond to the particle-in-a-box energies for low n, but not for high n. It’s probably easiest to appreciate why things go wrong for high n if you look at the Eigenfunctions, which are supposed to correspond to the wavefunctions.

a) Plot the Eigenfunctions. This will entail adding another instance of those three key plotting lines:

   ```
   figure()
   plot(x values, y values, marker = ‘o’)
   show()
   ```

   Now, the “x values” will be the position along the x-axis, of which there are N evenly spaced ones. If we measure length in units of L (the length of the well), then the list of x values would be xs = arange(0,N-1+0.1)*1.0/(N-1). The factor of 1.0 is there just to trick Python into returning decimal values rather than rounding to the nearest integer values (which it does when dividing integers by integers). If you want to plot, say, the 2nd Eigen function, then the “y values” will be

Eigenfunctions[:,1]. Notice two things: in Python, the index number of the first element in a list is "0", so the index number of the second is "1" (and so forth); also, calling elements [:,1] essentially means "all elements on row 1."

Vary which eigenfunction you plot.

Roughly what percent or fraction of the Eigenfunctions look fairly smooth and sinusoidal? How does that relate to the quality of the corresponding Eigenvalues?

Vary N to see what effect that has on the range over which the Eigenfunctions and Eigenvalues are good.

Nyquist Limit

The moral is: only as long as the wavelength of the Eigenfunction is much larger than the discrete spacing of positions are the positions as good as continuous and the Eigenfunction and Eigenvalues good approximations to what you'd have in the continuous case. More specifically, you should have observed that roughly the first quarter of *N* Eigen energies were in agreement with the theory; beyond that, the calculated energies got worse and worse. The N/4th solution is at the threshold between the first quarter that are fairly good and the other three quarters that are progressively worse. For this particular solution, the wavelength is 2L/N = 2 Δx. This means you only have two calculated points per wavelength to try to capture the character of the actual (sinusoidal) solution. Admittedly, the simulated solution looks like a triangle wave, but at least you can correctly capture the amplitude and wavelength with two points per wavelength, but you can't do so with any fewer. This is known as the Nyquist limit. One way to phrase it is that, to capture the general character of a periodic function, the span between samples can't be any greater than half the span between its repetition. In space, that means $\Delta x \le \lambda / 2$. If you've taken electronics, you've met the analogous statement for oscillations in time $\Delta t \le T / 2$, or in terms of frequencies, $f_{sample} \ge 2 f_{function}$.

(This document is only slightly modified from that available at compphysicsed.org; all credit is due there.)