Propagating An Infinite Square Well State, In Time, With Python

Introduction

As we all now know, it can be painful and time consuming to analytically expand an initial wave function on a given basis, in order to determine its time-evolved state. Here we will develop a numerical tool to do the expansion and evolution for us, allowing us to rapidly explore some interesting initial states.

As always, it's best to start with a problem that we know the answer to - let's try to replicate the results of examples 2.2 through 2.3 in Griffiths (3rd & 2nd edition).

Python Modules

At the beginning of our program we import several modules that contain the tools (functions) we will need. Python is a far more general purpose programming language than MATLAB, therefore, it's not a given that you will need to perform linear algebra, or plot data sets, etc...

```python
import scipy as sp
from scipy import constants, integrate, linalg
import matplotlib.pyplot as plt
from matplotlib import animation
```

Scipy is a large collection of scientific tools, which include functions such as exp() and sin(), as well as linear algebra operators. We will also need some specific sub-modules: *constants* for $\hbar$, $m_{electron}$; *integrate* for numerical integration; *linalg* for a least-squares projection onto a basis.

matplotlib is the go-to module for plotting in python. Additionally, we will use the *animate* sub-module to make some fancy movies of the time-evolution.

Specifying the Infinite Square Well Domain

To make our lives easy, we will let $a = 1$. Note the units are in meters, so this is quite a large well. We then use the *linspace* function to create an x-axis vector starting from 0, ending at a, and containing 1000 points. The *endpoint* parameter ensures point a is included, i.e. $[0, a]$ not $[0, a)$.

```python
a = 1
xs = sp.linspace(start=0, stop=a, num=int(1e3), endpoint=True) # Basis is (0 <= x <= a)
```

Initializing the Wave Function

As in example 2.2, we have the initial wave function

$$\Psi(x,0) = Ax(a-x), \quad (0 \leq x \leq a).$$

In our code we must initially set $A = 1$, so that integration of the function returns the inverse of the normalization constant.

```python
A = 1 # set to 1 before normalization
```

We programmatically write the initial wave function as

```python
def wf_0(x,A,a):
        return A*x*(a-x)
```

This function takes a vector of x values, the normalization constant, and the well width; it returns a vector of probability amplitudes.

Remembering that we need to normalize the probability *density*, we immediately create another function, which is the magnitude-squared of the first:

```python
def rho_0(x,A,a):
        return abs(wf_0(x, A, a))**2
```

Normalizing the Wavefunction

Numerical integration is an easy one-liner in Python, We pass the probability density *rho_0*, the arguments that *rho_0* requires, and the bounds of integration $[0, a]$. The function then returns the value of the integral, and an estimate on the absolute error.

```python
(y, abserr) = integrate.quad(func=rho_0, args=(A,a), a=0, b=a)
```

The normalization constant is simply the reciprocal of the integral, and we should confirm that we get the expected result, $A = \sqrt{30} = 5.477$.

```python
A = sp.sqrt(1/y)
print('Normalization Constant: {:0.3e}'.format(A))
```

Expanding the Initial Wave Function Over the Energy Eigenstates

We now know the initial state of the particle at $t = 0$, but we don't know how this state will evolve over time. We do, however, know how the eigenstates of the infinite square well evolve over time, and that any initial state can be decomposed as a sum of these eigenstates. The question is, can we use a limited number of these eigenstates to approximate the initial wave function? Let's determine this for the current case, and hope it holds for the others.

We start by defining the eigenstates of the infinite square well

```python
def psi_n_inf_sqr_well(x, a, n):
        psi_n = sp.sqrt(2/a)*sp.sin(n*sp.pi/a*x)
        return psi_n
```

Then, we use these eigenstates to create a “basis”. Note, this is an incomplete basis, as it does not span the entire space of possible wave functions.

```python
30 def basis_inf_sqr_well(x, a, num_states):
31         basis = sp.array([]).reshape(len(x),0)
32         for n in range(1, num_states+1):
33                 psi_n = psi_n_inf_sqr_well(x, a, n)
34                 basis = sp.hstack((basis, psi_n[:,sp.newaxis]))
35         return basis
```

At line 31 we create an empty array with as many rows as there are points in our x-axis vector (1000), and zero columns. At line 34 we iteratively stack a new column of probability amplitudes for the n’th eigenstate, creating an array of eigenstates.

$$\begin{pmatrix} \vdots & \vdots & \vdots \\ \psi_1(x_{m-1}) & \cdots & \psi_n(x_{m-1}) \\ \psi_1(x_m) & \cdots & \psi_n(x_m) \\ \psi_1(x_{m+1}) & \cdots & \psi_n(x_{m+1}) \\ \vdots & \vdots & \vdots \end{pmatrix}$$

Using this function, we create a basis from the first 15 eigenstates, and then project our initial wave function onto it. Because the basis is incomplete, we are forced to look for a least-squares fit. However, comparing our results to Griffiths ($|c_1|^2 = 0.998555\ldots$) shows that this method is sufficient for the given initial wave function. A quick check on the sum of probabilities yields a similar conclusion.

```python
36 num_states = 15          # Number of states in the basis
37 basis = basis_inf_sqr_well(xs, a, num_states) # Create a basis to expand over
38 cns = linalg.lstsq(basis, wf_0(xs,A,a))[0] # Find the least-squares projection of wf_0 onto th
39 print('|c_1|^2 = {:0.6f}'.format(abs(cns[0])**2)) # Compare to Griffith's value
40 print('Sum(|c_n|^2) = {:0.6f}'.format(sp.sum(abs(cns)**2))) # Ensure coefficient's sum to 1
```

Note, at line 38 we have sliced out the zeroth parameter returned by the *lstsq* function - the rest are not needed.

Plotting the Initial Wave Function, and its Expansion

Another important check is visually determining that the initial wave function and it’s expansion are closely matched. Lets start by making a figure that has two axes, one above the other (2x1 array)

```python
43 fig, ax = plt.subplots(2,1)
```

The *fig* object refers to the figure window, and *ax* is an array with one axis object at each entry. In the zeroth axis, we plot the magnitude-square of the initial wave function

```python
44 ax[0].plot(xs,abs(wf_0(xs,A,a)), label='$|\\mathrm{wf}_0|$')
```

its expansion, which is

$$\Psi_{expansion}(x,0) = \begin{pmatrix} \vdots & \vdots & \vdots \\ \psi_1(x_{m-1}) & \cdots & \psi_n(x_{m-1}) \\ \psi_1(x_m) & \cdots & \psi_n(x_m) \\ \psi_1(x_{m+1}) & \cdots & \psi_n(x_{m+1}) \\ \vdots & \vdots & \vdots \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix}$$

```python
ax[0].plot(xs,abs(sp.dot(basis,cns)), label='$|\\mathrm{wf}_0$ Expansion|')
```

and lastly, we make an empty plot that we will later fill with data for a series of time steps.

```python
wf_t_plt, = ax[0].plot([],[], label='$|\\mathrm{wf}_{t}|$')
```

The rest of this section is simply formatting

```python
ax[0].set_title('Initial, Expanded, and Time-Evolved Wavefunction')
ax[0].set_xlabel('x [m]')
ax[0].set_ylabel('$|\\Psi(x,t)|$')
ax[0].legend(loc='upper right')
```

In the second axis, below, we will make a "stem" plot of the magnitude-squared coefficients. This is just a nice way to visualize how much of each eigenstate is in our initial wave function.

```python
ax[1].stem(
        sp.arange(1,len(cns)+1),
        abs(cns)**2,
        linefmt='C1-',
        markerfmt='C1o',
        basefmt='k')
ax[1].set_title('Expansion Coefficients')
ax[1].set_xlabel('n')
ax[1].set_ylabel('$|c_n|^2$')
plt.tight_layout()
```

Propagating the Wave Function in Time

Plotting the time evolution is not much harder than plotting the expansion at $t = 0$. All we need to do is multiply by another set of time-dependent coefficients. These are defined as follows:

```python
def time_dependence_inf_sqr_well(t, a, num_states, m=1):
        n = sp.arange(1, num_states+1)
        En = (n*sp.pi*constants.hbar/a)**2/(2*m)
        phi_n = sp.exp(-1j*En*t/constants.hbar)
        return phi_n
```

Note that the n defined within this function is a vector, hence, the returned *phi_n* is also a vector, with the n'th component being the time-dependence of the n'th eigenstate.

We then choose the number of steps in our animation, the time interval per step, and point the animation function to the empty plot we created previously

```
num_steps = 1000
time_per_step = 10e3/num_steps # [s]
def init(): # initialization function: plot the background of each frame
    wf_t_plt.set_data([], [])
    return wf_t_plt,
```

We also determine what is plotted at each step in the animation

```
def animate(i): # animation function.  This is called sequentially
        phi_t = time_dependence_inf_sqr_well(i*time_per_step, a, num_states, m=constants.elect
        wf_t = abs(sp.dot(basis,cns*phi_t))
        wf_t_plt.set_data(xs,wf_t)
        return wf_t_plt,
```

As for the $t = 0$ case, we expand the solution as a weighted sum of basis vectors

$$\Psi_{expansion}(x,t) = \begin{pmatrix} \vdots & \vdots & \vdots \\ \psi_1(x_{m-1}) & \cdots & \psi_n(x_{m-1}) \\ \psi_1(x_m) & \cdots & \psi_n(x_m) \\ \psi_1(x_{m+1}) & \cdots & \psi_n(x_{m+1}) \\ \vdots & \vdots & \vdots \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix} \odot \begin{pmatrix} \phi_1(t) \\ \phi_2(t) \\ \vdots \\ \phi_n(t) \end{pmatrix} = \sum_n \psi_n(x) c_n \phi_n(t)$$

where $\odot$ is an element-wise multiplication of the coefficients.

The last steps are simply calling the animator, showing the plot, and sitting back to watch the magic happen (admittedly, you may have to wrestle the python for awhile to get it working; see the appendix). You can uncomment the save function if you want a video to share with family and friends (there are much more interesting initial states to come) ;-)

```
### Call the animator.  blit=True means only re-draw the parts that have changed.
anim = animation.FuncAnimation(fig, animate, init_func=init,
                               frames=(num_steps+1), interval=20, blit=True, repeat=False)

### Save the animation
# anim.save('gaussian_sqr_well.mp4', fps=30, extra_args=['-vcodec', 'libx264'], dpi=300)

### Show Plot
plt.show()
```

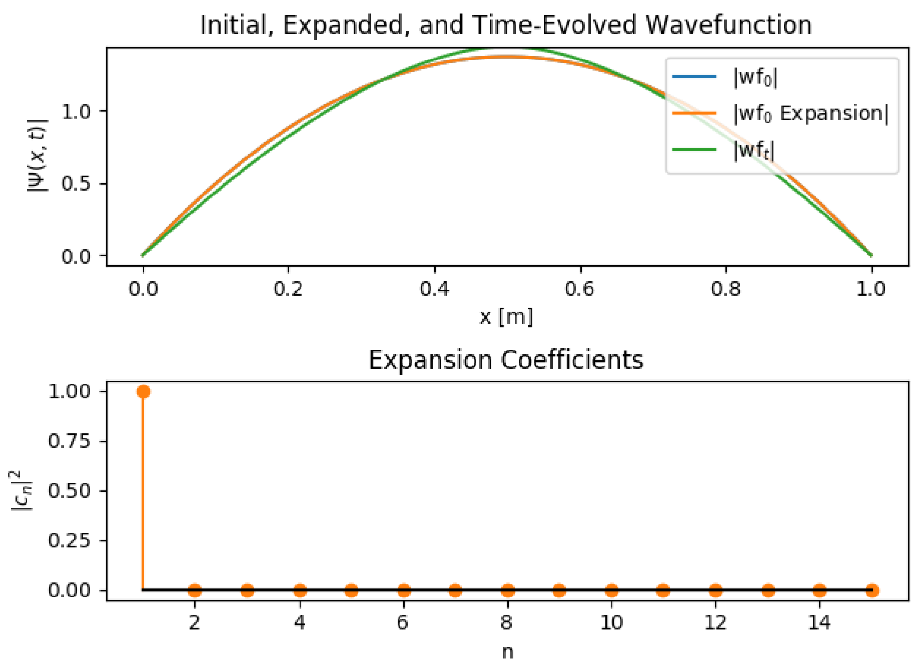

Figure 1: Expected output. The green trace should oscillate.

Making it a Black Box

To make the script easier to use, we can move all the parameters of interest up to the top. This is how the "..._blackbox.py" script is written.

```
### Paramters of interest ###
##############################################################
# Initial wavefunction (t=0)
def wf_0(x,A,a):
        return A*x*(a-x)
num_states = 15          # Number of states in the basis
num_steps = 1000 # Number of steps in the animation
time_per_step = 10e3/num_steps # [s]
##############################################################
```

Appendix

Some basic help:

- You will need to start an ipython session to run the scripts
- In ipython, navigate to folder where the script is saved using

  `cd C:/folder1/folder2/`

- In ipython run the scrip with the command

```
run wf_expansion_inf_sqr_doc.py
```

- To save the animation you may need to install a dependency for the animator to work (ffmpeg). In a terminal type

  ```
  pip install ffmpeg
  ```

  or if using Anaconda

  ```
  conda install -c menpo ffmpeg
  ```