Exploring the Probability Current

Group 7:
Shen, Yanzheng
Sousa-Fronenberg, Hannah
Talamo, Luca Giuseppe
Tan, Andrew
Taylor, Jacob
Teunissen, Samuel Adri Martijn

December 6, 2018

Our Project is on GitHub. Please Check it Out!
`https://github.com/jacotay7/phy456-python-project`

Introduction: What is $\vec{J}$?

In this assignment you will be exploring the probability current, $\vec{J}$, an important quantity in quantum mechanics. The probability current for a free, spinless particle is defined by the equation:

$$\vec{J} = \frac{\hbar}{2mi}(\Psi^* \nabla \Psi - \Psi \nabla \Psi^*) \tag{1}$$

$\vec{J}$ describes the flow of probability of a system. It is the closest analogue, in quantum mechanics, to a classical current, such as an electrical current or the flow of an incompressible fluid. Why? Because $\vec{J}$ obeys a local conservation laws similar to the ones that characterizes classical currents. For instance, where ρ is the charge density, an electrical current, $\vec{J_e}$, obeys the global the conservation law:

$$\frac{d}{dt}\int_V \rho \, d^3r = -\int_{\partial V} \vec{J_e} \cdot d\vec{S} \tag{2}$$

This last equation remains valid if we substitute the probability current, $\vec{J}$, for the electric current, $\vec{J_e}$ and the probability density $P = |\Psi|^2 = \Psi\Psi^*$ for the charge density ρ. The total amount of probability that "flows" into a volume of space must be equal to the rate of change in the probability that a certain object associated with the probability current will be found inside said volume.

You can, in fact, easily prove for yourself that (2), as modified in the preceding paragraph, holds. You can do this by taking the time derivative of P and rearranging the time dependent Schrodinger equation to obtain new expressions for $\frac{d}{dt}\Psi$ and $\frac{d}{dt}\Psi^*$. You should now be able to equate the time derivative of P and $\nabla \cdot \vec{J}$. Applying Stokes theorem then yields the desired conclusion.

You might have gleaned from the sketched proof of (2) that probability is globally conserved because the probability current obeys a local conservation law, which we in general call the continuity equation:

$$\nabla \cdot \vec{J} + \frac{\partial |\Psi|^2}{\partial t} = 0 \tag{3}$$

Indeed, the same conservation laws are found in other areas of physics, which you might be more familiar with. For example, global conservation of charge is a byproduct of the continuity equation for the conservation of charge:

$$\nabla \cdot \vec{J_e} + \frac{\partial \rho}{\partial t} = 0 \tag{4}$$

and global conservation of mass is a consequence of the continuity equation for the conservation of mass, where ρ_m refers to the mass density and $\mathbf{v}$ is the velocity of the particle:

$$\nabla \cdot (\rho_m \mathbf{v}) + \frac{\partial \rho_m}{\partial t} = 0 \tag{5}$$

By drawing an analogy from the underlying mechanics of other conservation laws to the quantum case, we can talk about the probability current as something that "flows" through the system. It is important to remind oneself that the particle does not flow through its vector field in the intuitive sense; the classical picture of electrons flowing through a current does not hold for quantum systems. All we can say, rather, is where the particle is *most* likely to be and, there we can mark a particles most probable path. The probability current is the closest thing we have to a trajectory in quantum mechanics. The continuity equation prohibits certain processes. The same way you cannot have a charge suddenly disappear from one region of space and reappear immediately in another, you cannot instantly transplant probability mass in this manner. The global form of the continuity equation states that probability of finding a particle in any region of the universe can vary only in proportion to the probability current over its boundary.

In this assignment, you will explore this feature of quantum mechanics by studying various systems and modelling their probability currents.

To begin with, let's look at the simplest system: a particle of definite energy, E_n ,in a box of length L. It's time dependent wave function is:

$$\Psi(x,t) = \sqrt{\frac{2}{L}} sin\left(\frac{n\pi x}{L}\right) e^{\frac{-iE_n t}{\hbar}} \tag{6}$$

If we calculate the probability current using (1), we will find that $\vec{J} = 0$. Might this be true for all purely real wave functions that are solutions to the time independent Schrodinger equation (i.e. for all $\Psi_{Re}(x)$ where $\Psi(x,t) = \Psi_{Re}(x)e^{-i\omega t}$)? What about solutions that are purely imaginary, $\Psi_{Im}(x)$?

It should be clear that if the wave function is purely real or purely imaginary, then the probability current will be 0. With this in mind, we can now move onto exploring a more complex system, the Hydrogen atom.

For this assignment, you will be working with the following wavefunctions,

$$\psi_{1,0,0} = \sqrt{\frac{1}{\pi a_0^3}} e^{\frac{-r}{a_0}} \tag{7}$$

$$\psi_{2,0,0} = \sqrt{\frac{1}{32\pi a_0^3}} \left(2 - \frac{r}{a_0}\right) e^{\frac{-r}{2a_0}} \tag{8}$$

$$\psi_{2,1,1} = -\sqrt{\frac{1}{64\pi a_0^3}} \left(\frac{r}{a_0}\right) e^{\frac{-r}{2a_0}} sin\theta e^{i\phi} \tag{9}$$

Do you notice any correlation between when the magnetic moment is non-zero and when the probability current is non-zero? You will explore this correspondence this later in the assignment.

Questions

I: Probability Current In 1D Systems

Consider the classic 1-dimensional particle in a box scenario. We have the following potential:

$$V(x) = \begin{cases} 0 & \text{if } 0 < x < L \\ \infty & \text{otherwise} \end{cases}$$

It follows from the Schrodinger equation that our system has the following eigenstates:

$$\psi_n(r) = \sqrt{\frac{2}{L}} \sin\left(\frac{n\pi x}{L}\right) \tag{10}$$

each with energy:

$$E_n = \frac{n^2\pi^2\hbar^2}{2mL^2} \tag{11}$$

Given the above, we will now consider a wave packet subject to this potential (assume it was placed in our system by some experiment set-up). Such a wave packet, $\Psi(x,t)$, will time-evolve according to the Schrodinger equation and will thus produce a non-zero probability current. I encourage the reader to try and visualize what this probability current might look like when the packet reflects off the boundary of our box. The goal of this assignment is simple, using the above eigenbasis write a program which time-evolves the wave packet (see Spectral Methods) and produce an animation of the probability current.

You will be provided with a Python script called *question_I.py*. This script is missing significant components which you will be asked to fill in. Each missing component should be clearly marked in the script with a description of what must be done.

For extra credit: Create an analogous animation for the quantum harmonic oscillator potential.

Hints:

- *NumPy* has a helpful function for computing integrals using the Trapezoidal rule. To compute the integral of a function $f(x)$ from a to b we have:

```
xi = np.linspace(a,b)
y = f(xi)
I = np.trapz(y, x = xi)
```

- When computing derivatives, ensure your derivative array is the same length as your co-ordinate array. If you use the central differences method found in Numerical Derivatives and Integrals, you will have to find a way to account for derivatives at the endpoints.
- *NumPy* has a helpful function for computing Hermite Polynomials:

```
from numpy.polynomial.hermite import hermval
```

- The main program for the simple harmonic oscillator should be very similar to that of the particle in a box potential.
- Take advantage of *NumPy*'s array indexing when computing derivatives. This is not only more concise, but more computationally efficient than using loops. If there is a loop in your *derivative* function, you are most likely not following this advice.
- You can increase the number of quantum harmonic oscillator eigenstates in your discretized basis by being careful not to overflow the *math.sqrt* function. You should be able to use 200 eigenstates in your solution before running into overflows in *NumPy*'s *hermval* function.
- The program should run quickly, if you are waiting more than a few seconds, reconsider how you are computing things. Don't do more work than necessary, whenever possible, let the good people at *SciPy* and *NumPy* do the work!

II: The Magnetic Moment of Hydrogen Eigenstates

The following section uses the same unit conventions as the wikipedia article for the Hydrogen atom
We will now extend our use of $\vec{J}$ to a 3D system. We consider the 3-dimensional potential of a Hydrogen atom, given by

$$V(\vec{R}) = \frac{e^2}{4\pi\epsilon_0 R}$$

In spherical co-ordinates, the stationary states are given by:

$$\psi_{n\ell m}(r,\vartheta,\varphi) = \sqrt{\left(\frac{2}{na_0^*}\right)^3 \frac{(n-\ell-1)!}{2n(n+\ell)!}} e^{-\rho/2}\rho^\ell L_{n-\ell-1}^{2\ell+1}(\rho) Y_\ell^m(\vartheta,\varphi) \tag{12}$$

where

$$\rho = \frac{2r}{na_0^*}$$

$a_0^* = \dfrac{4\pi\epsilon_0\hbar^2}{\mu e^2}$ is the reduced Bohr radius

$L_{n-\ell-1}^{2\ell+1}(\rho)$ is a generalized Laguerre polynomial of degree $n-l-1$

$Y_\ell^m(\vartheta,\varphi)$ is a spherical harmonic function of degree l and order m

Recall from class that we can associate a magnetic moment with a given m state using the relation

$$\mu_z = \frac{em}{2\hbar} \tag{13}$$

Shankar derives this using the Hamiltonian for a free particle, but you'll show, through examples, how it arises from the probability current. You will plot the currents associated with the wavefunctions ψ_{100}, ψ_{210}, and ψ_{211} and draw conclusions based on the current distributions.

First, calculate the 3 current distributions analytically and plot them using the *quiver* plotting method. An example for plotting vector fields can be found in Vector Field Plots in Python. Make sure to write up your derivations as well.

Now compare the plots to those generated using the *ProbabilityCurrent3D* starter code. Just as in the 1D case, you will have to fill in some functions yourself. You may reuse your vector field plotting code from the previous section in one of the methods.

Are you confident that your plotting codes work? How can you use your knowledge of the magnetic moments for each eigenfunction to be sure? Briefly explain how your current plots relate to the magnetic moment of the eigenfunctions.

Hints:

- You will need to familiarize yourself with *numpy.meshgrid* documentation to complete this part. The scipy documentation website has a good explanation.
- When computing $\vec{J}$, your program will return complex values. The imaginary parts are small and due to rounding errors. You can neglect them by storing only the real part of your $\vec{J}$ array using the following code

```
self.J = J.real
```

- When computing derivatives, ensure your derivative array is the same length as your co-ordinate array. If you use the central differences method found in Numerical Derivatives and Integrals, you will have to find a way to account for derivatives at the endpoints.

Computing Background

Spectral Methods

The fundamental problem of Quantum Mechanics is to solve the Schrodinger Equation for a given external potential. A popular method for doing so is finding a set of stationary solutions, and decomposing any given state into a linear combination of stationary states. Such solutions obey the time-independent Schrodinger equation and the entire set form an ortho-normal eigenbasis of our system. We will assume that such a basis exists, and is composed of eigenstates $\psi_n(r,t)$ with eigenvalues E_n. Thus, for a given time-independent Hamiltonian, $\mathcal{H}$, we have:

$$\mathcal{H}\psi_n(\mathbf{r},t) = E_n\psi_n(\mathbf{r},t) \tag{14}$$

$$i\hbar\frac{d\psi_n}{dt} = E_n\psi_n \implies \psi_n(\mathbf{r},t) = e^{\frac{-iE_nt}{\hbar}}\psi_n(\mathbf{r},0) \tag{15}$$

Where ψ_n is the n^{th} eigenstate of our system. Applying this result to a general state, $\Psi(\mathbf{r},t)$ we have:

$$\Psi(\mathbf{r},t) = \sum_n c_n\psi_n(\mathbf{r},t) = \sum_n c_n e^{\frac{-iE_nt}{\hbar}}\psi_n(\mathbf{r},0) \tag{16}$$

$$c_n = \int \psi_n(\mathbf{r},0)\Psi(\mathbf{r},0)d^3\mathbf{r} \tag{17}$$

Assuming we know the initial state of our system, our problem is now in a form we can work with. Briefly, the steps involved are:

- Solve for the eigenstates of our system. If an analytical solution is not within our reach, we attempt to solve the problem via numerical methods (Ex: Numerov's Method).
- Decompose our initial state into a linear combination of stationary states.
- Time evolve our linear combination using the energy of each stationary state.

The benefits of such a solution method are that, assuming our solution is stable, we can investigate the system at any time with a fixed computational cost. That is to say, such a method is useful for studying the system at $t >> 0$. While iterative methods (Crank-Nicolson, etc..) tend to be more flexible (one program will work for any initial state with appropriate boundary conditions), they also tend to be harder to code and less physically intuitive. For those reasons, we will only consider spectral methods for this assignment.

Object-Oriented Programming in Python

Typically, physicist's think about coding *functionally*, that is to say, a program is a machine which sequentially applies mathematical (or quasi-mathematical) functions to a data set. To an object-oriented programmer, however, a program is a machine which passes around *objects* (I tend to visualize such objects as material things like cubes) which themselves have *attributes* (colour, shape, weight, etc). What makes these objects useful is that they also can use, modify, and transform their own attributes through internal functions called *methods*. Although adopting such a programming paradigm is not strictly necessary to complete this assignment, it can be a useful framework to write clear, concise, user-friendly programs. Consider the following code snippet which aims to introduce you to object-oriented programming:

```
#Create a Cube object (called a class)
class Cube:

    #Function which initializes my Cube object
    def __init__(self, length, weight, colour, name):

        #Set attributes given on initialization
        self.length = length
        self.weight = weight
        self.colour = colour
```

```
        self.name = name

    #Function update the colour attribute of my cube
    def change_colour(self, new_colour):

        self.colour = new_colour

    #Function to compute the volume of my cube
    def volume(self):
        return self.length**3

#Main program
if __name__ == "__main__":

    #Create a cube object
    cube = Cube(3, 5, "Blue", "my Cube")

    print(cube.length)
    print(cube.volume())

    print(cube.colour)
    #Change the colour of my cube
    cube.change_colour("Red")
    print(cube.colour)
```

In this program we define a new object known as *Cube*. The object contains the function *__init__* which is called when the main program creates an *instance* of the object. The main program creates such an instance using the following syntax:

```
cube = Cube(3, 5, "Blue", "my Cube")
```

The newly instantiated cube object now stores its attributes internally which can then be accessed in the following way:

```
print(cube.length)
```

Remind you of anything? If you have worked with *NumPy* arrays in the past, then this syntax should be familiar (think array.shape or array.size). Furthermore, the cube object contains methods which can called upon to manipulate the objects attributes to produce useful results. This is demonstrated through the volume function:

```
print(cube.volume())
```

I encourage the reader to run the above program if they are unfamiliar with object-oriented programming and ensure they understand what each line is doing. For the sake of brevity, I will omit a discussion on inheritance, encapsulation and other important concepts within the programming paradigm.

Numerical Derivatives and Integrals

Given the universality of calculus in the physical sciences, it is not surprising that there are many different ways to numerically solve integrals and derivatives. Derivatives are somewhat less complicated since even a first order approximation is usually sufficient. For our purposes we will consider the finite differences method, defined as follows:

$$\frac{df}{dx} \approx \frac{f(x+h) - f(x-h)}{2h} \tag{18}$$

This method carries an optimal error of:

$$\epsilon \approx \frac{10^{-16}|f(x)|}{h} + \frac{h^2}{6}|f'''(x)| \tag{19}$$

We can easily extrapolate this method for computing gradients by computing the derivative of each components separately. A good exercise for the reader is to find a formula for the second derivative using the central difference method.

When considering integrals, there is a wide range of methods to solve an even wider range of problems. For brevity's sake, I will consider only the simplest set of such methods: the Newton-Cotes formulas:

$$\int_a^b f(x) \approx \sum_1^N w_n f\left(x_0 + n\frac{b-a}{N}\right) \tag{20}$$

Where w_n are weights assigned to each each discretized evaluation of the integral. The weights are determined by the order to which we approximate the function between the discretized points. For example, the first order Newton-Cotes formula (known as the trapezoidal rule), approximates the function as being linear between discretized points. The reader can check that the set of coefficients which correspond to the trapezoidal rule are:

$$w_n = \frac{b-a}{N} \times \left\{\frac{1}{2}, 1, 1, 1, ..., 1, 1, 1, \frac{1}{2}\right\} \tag{21}$$

with error:

$$\epsilon \approx \frac{1}{12}\left(\frac{b-a}{N}\right)^2 [f'(a) - f'(b)] \tag{22}$$

The problem of numerical integration is now reduced to finding the set of coefficients for the desired order (or equivalently a method for producing the coefficients). I would encourage the reader to solve for the second-order Newton-Cotes coefficients (see Simpson's rule). For this assignment we will only consider the trapezoidal method since the introduced error is sufficiently small and therefore does not detract from the physical results.

Animations in Python

Physicists often concern themselves with presenting multi-dimensional data in a clear and concise manner. Often a single 2D line, or histogram plot is sufficient, but occasionally animations can be used to more clearly communicate the relevant data. Python's *matplotlib* module has a very useful *animation* library which is relatively simple to use. The user first initializes a standard plot (histogram, line, scatter, density, etc) and subsequently defines a function which is called on each successive frame of the animation. This function updates the plot frame by frame, and the result is a set of plots which together form an animation. Here is a simple example which produces an animation of 3D scalar field.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation

#Constants
N = 100
L = 0.1

#Dicretize one side of a cube
side = np.linspace(-1*L/2, L/2, N)

#Create cube
X, Y, Z = np.meshgrid(side, side, side, indexing='ij')

#Compute scalar field (think point-charge electric potential)
field = 1/(np.sqrt(X**2 + Y**2 + Z**2))

#Initialize plot
fig, ax = plt.subplots()
im = ax.imshow(field[:,:,0])
```

```
#Function to update the animation
def update(i):

    #Plot to next value in the field
    im.set_array(field[:,:,i])

    #Return the plot, don't forget the comma afterwards
    return im,

#Create the animation
ani = FuncAnimation(fig, update, frames=N)
#Show the animation
plt.show()
```

The above code allows you to visualize the electric field of a point-charge like medical professionals visualize the human brain (volume slices). Such animations are invaluable in some fields and are therefore an excellent thing to become familiar with. We will use animations extensively in this assignment to try and properly visualize the probability current.

Vector Field Plots in Python

While slice imaging is a useful tool, it's use is restricted to scalar fields. Vector fields such as $\vec{J}$ can only be visualized by projecting the field on a specific axis or taking its modulus at each point. Sometimes you can use symmetry arguments to predict which direction the field should go in and these scalar images are enough. However, 3D vector plots are often necessary if you want to view the directional of your field. Here's an example of how to plot a 3 dimensional vector field using the *ax.quiver* plotting module found in the *axes3d* module of *mpl_toolkits.mplot3d*.

```
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
import numpy as np

#Constants
N = 10
L = 1

#Create Cube
X, Y, Z = np.meshgrid(np.linspace(-L/2, L/2, N), np.linspace(-L/2, L/2, N), \
    np.linspace(-L/2, L/2, N), indexing = 'ij')

#Initialize Plot
fig = plt.figure()
ax = fig.gca(projection='3d')

#Create Vector Field (think point charge Electric Field)
R = np.sqrt(X**2 + Y**2 + Z**2)
U = X/(R**3)
V = Y/(R**3)
W = Z/(R**3)

#Set axis to center origin
ax.set_xlim3d(-L/2, L/2)
ax.set_ylim3d(-L/2, L/2)
ax.set_zlim3d(-L/2, L/2)

#populate plot (try a few different length values for your plots to scale arrows properly)
ax.quiver(X, Y, Z, U, V, W, length = 0.001)
```

```
##optional code to plot a down-sampled version
#ax.quiver(X[::s,::s, ::s], Y[::s,::s, ::s], Z[::s,::s, ::s],\
#    U[::s,::s, ::s], V[::s,::s, ::s], W[::s,::s, ::s], length = 0.001)

#show field
plt.show()
```

Distribution of Work

Shen, Yanzheng:

- Worked on derivation solutions
- Contributed to probability current background

Sousa-Fronenberg, Hannah:

- Coordinator of the Overleaf write-up document
- Provided some background physics (for intro section)
- Wrote the introduction section
- Edited the final assignment document

Talamo, Luca Giuseppe:

- Came up with probability current idea
- Wrote Question II text
- Wrote starter script for Question II
- Wrote most of code for CoordinateField3D.py
- Contributed to Computational Background text
- Contributed to Planning/Formatting/Editing

Tan, Andrew:

- Wrote class structure for Question II
- Contributed to code for Question II and CoordinateField3D.py
- Contributed to write up of Question II
- Contributed to Formatting/Editing

Taylor, Jacob:

- Wrote Question I text
- Wrote starter script and code solution for Question I
- Wrote most of Computational Background text
- Contributed to code for Question II and CoordinateField3D.py
- Contributed to Planning/Formatting/Editing

Teunissen, Samuel Adri Martijn:

- Worked on derivation solutions
- Contributed to probability current background

Derivation Solutions

For ψ_{100}, use equation (1):

$$\vec{J_{100}} = \frac{\hbar}{2mi}(\Psi^*_{100}\nabla\Psi_{100} - \Psi_{100}\nabla\Psi^*_{100}) \tag{23}$$

From equation (7), you'll find ψ_{100} is real,

$$\Psi^*_{100} = \sqrt{\frac{1}{\pi a_0^3}}e^{\frac{-r}{a_0}} = \Psi_{100} \tag{24}$$

Substituting this into (23):

$$\vec{J_{100}} = \frac{\hbar}{2mi}(\Psi_{100}\nabla\Psi_{100} - \Psi_{100}\nabla\Psi_{100}) = 0 \tag{25}$$

Similarly, for ψ_{210}, since the wavefunction is real.

$$\vec{J_{210}} = \frac{\hbar}{2mi}(\Psi_{210}\nabla\Psi_{210} - \Psi_{210}\nabla\Psi_{210}) = 0 \tag{26}$$

For ψ_{211}, use equation (9), you'll get:

$$\Psi^*_{211} = -\sqrt{\frac{1}{64\pi a_0^3}}\left(\frac{r}{a_0}\right)e^{\frac{-r}{2a_0}}sin\theta e^{-i\phi} \tag{27}$$

Substituting this into equation (1)

$$\begin{aligned}
\vec{J_{211}} &= \frac{\hbar}{2mi}(\Psi^*_{211}\nabla\Psi_{211} - \Psi_{211}\nabla\Psi^*_{211}) \\
&= \frac{\hbar}{2mi} * \frac{1}{64\pi a_0^5}[(re^{\frac{-r}{2a_0}}sin\theta e^{-i\phi})\nabla(re^{\frac{-r}{2a_0}}sin\theta e^{i\phi}) - (re^{\frac{-r}{2a_0}}sin\theta e^{i\phi})\nabla(re^{\frac{-r}{2a_0}}sin\theta e^{-i\phi})] \\
&= \frac{\hbar}{2mi} * \frac{1}{64\pi a_0^5}[(r^2e^{-\frac{r}{a_0}}sin^2\theta e^{i\phi})\frac{1}{rsin\theta}\frac{d}{d\phi}(e^{i\phi}) - (r^2e^{-\frac{r}{a_0}}sin^2\theta e^{-i\phi})\frac{1}{rsin\theta}\frac{d}{d\phi}(e^{-i\phi})]\widehat{\phi} \\
&= \frac{\hbar}{2mi} * \frac{1}{64\pi a_0^5}\left(re^{-\frac{r}{a_0}}sin\theta * i - re^{-\frac{r}{a_0}}sin\theta * -i\right)\widehat{\phi} \\
&= \frac{\hbar}{64\pi a_0^5 m}\left(re^{-\frac{r}{a_0}}sin\theta\right)\widehat{\phi}
\end{aligned} \tag{28}$$

Starter Scripts

Note: Please see the GitHub for copies of the scripts.

I: Probability Current In 1D Systems

```
#Import neccessary modules
import numpy as np
import matplotlib.pyplot as plt

#Animation module
import matplotlib.animation as animation

#Embeds animations in jupyter notebooks
plt.rcParams["animation.html"] = "jshtml"

#Physical Constants
from scipy.constants import hbar

#Helpful functions
from numpy.polynomial.hermite import hermval
from math import sqrt


"""
Class QuantumSystem1D
Purpose: A generalized 1D quantum system
Parameters:
    - L: Length of the 1D system
    - M: Mass of the particle
    - N: Number of sample points
    - x: The sample point along the x axis
    - name: Unique name of the system (ex: PIB, SHO)
    - psi0: The initial state of the system
    - cn: The inital eigen decomposition coefficients

Methods:
    set_psi0: Initializes the state
        - psi0_func: A user defined function which computes psi0
        - args: the arugments of psi0_func
    generate_initial_cn: Initializes the eigen decomposition coefficients
        - n: the number of eigenstates you want to consider
    normalize: normalizes a given state
        - psi: the state to be normalized
    psi: returns the state at time t
        - t: The time to compute the state at
    psi_conj: return the complex conjugate of psi(t)
        - t: The time to find the state at
    psi_squared: returns |psi(t)|^2
        - t: The time to coupute psi(t) at
    derivative: Takes the numerical derivative of a given psi(t)
        - psi_t: the state at time t
    probability current: Compute the probability current at a time t
        - t: The time to compute the probability current at
    create_animation: Creates an animation of the system
        - start: the starting time for the animation
        - end: the ending time of the animation
```

```
        - frames: the number of frames in the animation

    """
    class QuantumSystem1D:

        def __init__(self, L, M, N=1000, name=''):

            self.x = np.linspace(0,L,N)
            self.N = N
            self.M = M
            self.L = L
            self.name = name
            return

        def set_psi0(self, psi0_func, args):

            self.psi0 = psi0_func(self.x, *args)
            self.psi0 = self.normalize(self.psi0)

        def generate_initial_cn(self,n):

            """
            Student must complete

            Requirements: Fill a complex, 1D array of size n with
            the initial coefficients of the eigen decomposition of Psi(x,0).
            Save the resulting array as a attribute of the system (self.cn)
            """

        def normalize(self, psi_t):

            """
            Student must complete
            Requirements: Write a function to normalize the 1D array, psi_t,
            representing the wavefunction at time t. This function should return
            the normalized array. Hint: Integrate the absolute square
            """

        def psi_conj(self, t):
            return np.conj(self.psi(t))

        def psi(self, t):

            """
            Student must complete
            Requirements: Write a function to which comput Psi(x,t) at a given
            time t. This is where you apply the spectral method.
            """

       def psi_squared(self,t):

            """
            Student must complete
            Requirements: Write a function which returns the absolute square
            of psi(t). Hint: Re-use code whenever possible
            """
```

```
def derivative(self, psi_t):

    """
    Student must complete
    Requirements: Write a function which returns the derivative
    of the array psi_t. The return value should be an array of
    the same shape as psi_t. You should apply the central differences
    method on the interior points. At the boundary apply the
    forward/backward difference methods.
    """

def probability_current(self, t):

    """
    Student must complete
    Requirements: Write a function to compute the probability current
    at time t. Hint: re-use code whenever possible, if the above functions
    have been written corectly, this should not take long to complete.
    """

def create_animation(self, start, end, frames = 100):

    #Set-Up the figure for the animation
    fig, ax = plt.subplots(nrows=3, ncols=1, sharex=True, figsize=[10,8])
    im, = ax[0].plot(self.x*1e9, self.psi_squared(start), color = 'black')
    im_real, = ax[1].plot(self.x*1e9, self.psi(start).real, color = 'blue', label = "real")
    im_imag, = ax[1].plot(self.x*1e9, self.psi(start).imag, color = 'orange', label = "imag")
    im_j, = ax[2].plot(self.x*1e9, self.probability_current(start), color = 'black')
    ax[0].grid()
    ax[1].grid()
    ax[2].grid()
    ax[2].set_xlabel("X [nm]")
    ax[1].legend()
    if(self.name == 'SHO'):
        ax[0].set_title("Wave Packet in SHO Potential Well")
    elif(self.name == 'PIB'):
        ax[0].set_title("Wave Packet in Inifinite Potential Well")
    else:
        ax[0].set_title("Wave Packet in Unknown Potential Well")
    ax[0].set_ylabel("Probability Density")
    ax[0].set_ylabel("$|\psi(x)|^2$")
    ax[1].set_ylabel("$\psi(x)$")
    ax[2].set_ylabel("J [$m^{-1}s^{-1}$]")
    plt.xlim(0,self.L*1e9)
    ax[2].set_ylim(-1*np.nanmax(im_j.get_ydata()),np.nanmax(im_j.get_ydata()))
    plt.tight_layout()
    #Updateable Text box for time
    ttl = ax[0].text(.78, 0.9, '', transform = ax[0].transAxes, va='center')

    #Iterable function which produces each frame of the animation
    def animate(i):

        t = start + i*(end-start)/(frames-1)
        psi_t = self.psi(t)
        im.set_data(self.x*1e9, np.abs(psi_t)**2)
        im_real.set_data(self.x*1e9, psi_t.real)
        im_imag.set_data(self.x*1e9, psi_t.imag)
```

```
            im_j.set_data(self.x*1e9, self.probability_current(t))
            #Update animation text
            ttl.set_text("t = {:.3f} fs".format(t*1e15))
            return im, im_real, im_imag, im_j,  ttl,

        return animation.FuncAnimation(fig, animate,\
                    frames=frames, blit=True)

"""
Class: PIB
Parent: QuantumSystem1D
Purpose: A particle in a box quantum system
Parameters:
    - eigenstates: an array holding the first n eigenstates of the system
    - En: an array holding the energy of each eigenstate
Methods:
    generate_eigenstates: Find the first n eigenstates of the system
        - n: The number of eigenstates to generate
"""
class PIB(QuantumSystem1D):

    def __init__(self, L, M, N=1000):
        QuantumSystem1D.__init__(self, L, M, N=N, name="PIB")
        return

    def generate_eigenstates(self, n):

        self.eigenstates = np.zeros((n,self.x.size), dtype=np.complex128)
        self.En = (np.arange(1,n+1)*np.pi*hbar/self.L)**2/(2*self.M)
        for i in range(n):
            pre_factor = sqrt(2/self.L)
            self.eigenstates[i,:] = pre_factor*np.sin((i+1)*np.pi/L*self.x)
        self.generate_initial_cn(n)

"""
Class: SHO
Parent: QuantumSystem1D
Purpose: A particle in a simple harmonic osccilator potential
Parameters:
    - eigenstates: an array holding the first n eigenstates of the system
    - En: an array holding the energy of each eigenstate
Methods:
    generate_eigenstates: Find the first n eigenstates of the system
        - n: The number of eigenstates to generate
"""
class SHO(QuantumSystem1D):

    def __init__(self, L, M, omega, x0, N=1000):

        QuantumSystem1D.__init__(self, L, M, N=N, name='SHO')
        self.w = omega
        self.x0 = x0
        return

    def generate_eigenstates(self, n):

        """
```

```
        Student can complete for extra credit
        Requirements: Should be analogous to the Particle in
        a box implementation.
        """

"""
A function to initialize a particle as a wave packet
"""
def psi0_func(x, x0, sigma, kappa):
    return np.exp(-1*(x-x0)**2/(2*sigma**2))*np.exp(1j*kappa*x)


#Main body of the program
if __name__ == "__main__":

    #Constants
    L = 1e-8 #m
    x0 = L/2
    sigma = 2e-10 #m
    kappa = 5e10 #m^-1
    M = 9.109e-31
    n = 500

    #Create Particle
    particle = PIB(L, M, N=1000)

    #Set inital conditions
    particle.set_psi0(psi0_func, (x0, sigma, kappa))
    particle.generate_eigenstates(n)

    #Plot a few times
    particle.create_animation(0, 2e-15, frames = 100)
    ani = particle.create_animation(0, 2e-15, frames = 100)

    #Uncomment to save as a video file (might not work on non-linux)
    #ani.save('PIB.mp4', dpi=80, writer='imagemagick')
    plt.show()

    #Number of eigensates used (any higher and the hermite polynomials have problems)
    n = 172
    omega = 2e15

    """
    Student can add quantum harmonic oscillator main program here for extra credit
    """
```

II: The Magnetic Moment of Hydrogen Eigenstates

Script For Making 3D Field Objects

```
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import axes3d


"""
Class CoordinateField3D
Purpose: Stores 3D Co-ordinate Grid and an associated Vector Field.
```

```
Parameters:
    - L1, L2, L3: lengths of co-ordinate arrays
    - N1, N2, N3: Number of sample points along coodinate directions
    - initCoord: Unique name of co-ordinates defined by Li and Ni (ex: "RECTANGULAR", "CYLINDRICAL", "S
Properties:
    -L = L1, L2, L3
    -N = N1, N2, N3
    -coordCart: 3tuple of cartesian meshgrid coordinates
    -coordSphere: 3tuple of spherical meshgrid coordinates
    -vectorField: 3tuple of vector values in meshgrid-compatible form
Methods:
    cart2sphere: converts cartesian meshgrid co-ordinates into spherical meshgrid coordinates
    sphere2cart: converts spherical meshgrid co-ordinates into cartesian meshgrid coordinates
    plotField3D: makes a 3D plot of the vector field
"""


class CoordinateField3D:

    def __init__(self, L1, L2, L3, N1, N2, N3, gridType="RECTANGULAR"):

        self.L = L1, L2, L3

        self.N = N1, N2, N3
        self.gridType = gridType
        if self.gridType == "SPHERICAL":
            # radial array: omit 0 to avoid blowup at origin
            r = np.linspace(0, L1, N1)
            theta = np.linspace(0., L2, N2)  # polar array
            phi = np.linspace(0, L3, N3)  # azimuthal array
            self.r, self.theta, self.phi = np.meshgrid(
                r, theta, phi, indexing='ij')
            self.x, self.y, self.z = self.sphere2cart()
        elif self.gridType == "RECTANGULAR":
            # bump up starting index by an intervalto avoid dividing by zero
            x = np.linspace(-1*L1/2, L1/2, N1)
            y = np.linspace(-1*L2/2, L2/2, N2)
            z = np.linspace(-1*L3/2, L3/2, N3)
            self.volume_element = (x[1]-x[0])**3
            self.x, self.y, self.z = np.meshgrid(x, y, z, indexing='ij')
            self.r, self.theta, self.phi = self.cart2sphere()

    def sphere2cart(self):
        x, y, z = self.r*np.sin(self.theta)*np.cos(self.phi), self.r * \
            np.sin(self.theta)*np.sin(self.phi), self.r*np.cos(self.theta)
        return x, y, z

    def cart2sphere(self):
        r = np.sqrt(self.x**2 + self.y**2 + self.z**2)
        theta = np.arccos(self.z/r)
        phi = np.arctan2(self.y, self.x)
        return r, theta, phi

    def fillContainer(self, func, func_args, coordinate_system="RECTANGULAR"):
        if(coordinate_system == "RECTANGULAR"):
            self.data = func(self.x, self.y, self.z, *func_args)
        elif(coordinate_system == "SPHERICAL"):
```

```
            self.data = func(self.r, self.theta, self.phi, *func_args)
        else:
            print("Unrecognised Co-ordinate System")

    def copy(self):
        copy = CoordinateField3D(
            self.L[0], self.L[1], self.L[2], self.N[0], self.N[1], self.N[2], gridType=self.gridType)
        copy.data = self.data
        return copy
```

Question Script

```
from abc import ABC, abstractmethod
from math import sqrt

import matplotlib.animation as animation
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import axes3d
from numpy.polynomial.hermite import hermval
from scipy.constants import e, epsilon_0, hbar, m_e
from scipy.fftpack import dst, idst
from scipy.misc import factorial
from scipy.special import genlaguerre, sph_harm

from CoordinateField3D import CoordinateField3D

plt.rcParams["animation.html"] = "jshtml"

"""
Class QuantumSystem3D
Purpose: A generalized 3D quantum system
Parameters:
    - L: Linear size of the system
    - M: Mass of the particle
    - N: Number of sample points per dimension
    - x: The sample point along the x axis
    - name: Unique name of the system (ex: PIB, SHO)
    - psi0: The initial state of the system
    - cn: The inital eigen decomposition coefficients
Methods:
    set_psi0: Initializes the state
        - psi0_func: A user defined function which computes psi0
        - args: the arugments of psi0_func
    generate_initial_cn: Initializes the eigen decomposition coefficients
        - n: the number of eigenstates you want to consider
    normalize: normalizes a given state
        - psi: the state to be normalized
    psi: returns the state at time t
        - t: The time to compute the state at
    psi_conj: return the complex conjugate of psi(t)
        - t: The time to find the state at
    psi_squared: returns |psi(t)|^2
        - t: The time to coupute psi(t) at
    derivative: Takes the numerical derivative of a given psi(t)
        - psi_t: the state at time t
    probability current: Compute the probability current at a time t
```

```
        - t: The time to compute the probability current at
    create_animation: Creates an animation of the system
        - start: the starting time for the animation
        - end: the ending time of the animation
        - frames: the number of frames in the animation
"""


class QuantumSystem3D(ABC):

    def __init__(self, L, M, N=250, name='', realdim=True):
        self.M = M
        self.L = L
        self.N = N
        self.name = name
        self.realdim = realdim
        return

    def normalize(self):
        '''
        Student must complete

        Requirements: Write a function to normalize the 1D array, psi_t,
        representing the wavefunction at time t. This function should return
        the normalized array. Hint: Integrate the absolute square
        '''
        return psi/sqrt(norm)

    def psi_conj(self):
        psi_conj = self.psi.copy()
        psi_conj.data = np.conj(self.psi.data)
        return psi_conj

    def psi_squared(self):
        psi_squared = self.psi.copy()
        psi_squared.data = np.abs(self.psi.data)**2
        return psi_squared

    def gradient(self, psi):
        def grad_func(x, y, z, psi):
            """
            Student Must Complete

            Requirements: Write a function which takes a 3D gradient using a generalized
            central differences method. Note that x, y, and z are meshgrid arrays
            specifying the coordinate inputs.  The function should return a 3D meshgrid
            populated with vector field array instead of numbers (i.e. a 4D meshgrid).
            Be careful with boundary points!
            """
            return psi_grad
        psi_grad = CoordinateField3D(
            self.L, self.L, self.L, self.N, self.N, self.N)
        psi_grad.fillContainer(grad_func, (psi.data,))
        return psi_grad

    def find_probability_current(self):
        """
```

```
        Student must complete

        Requirements: Write a function to compute the probability current
        of the psi. Hint: re-use code whenever possible, if the above functions
        have been written corectly, this should not take long to complete. Save
        the array as a property of the system (self.J).  Save the magnitude as a
        3D meshgrid array populated with scalar (self.J_mag).
        Hint: J should be real. Complex number may be due to rounding error and can be neglected
        """

    @abstractmethod
    def set_wavefunction(self, n, l, m, realdim=False):
        pass


"""
Class: HydrogenAtom
Parent: QuantumSystem3D
Purpose: A Hydrogen atom quantum system
Parameters:
    - eigenstates: an array holding the first n eigenstates of the system
    - En: an array holding the energy of each eigenstate
Methods:
    -set_wavefunction: specify eigenfunction that you want to plot
        - n, l, m: quantum numbers specifying the eigenfunction
"""


class HydrogenAtom(QuantumSystem3D):

    def __init__(self, L, N=250, realdim=True):
        QuantumSystem3D.__init__(
            self, L, m_e, N=N, name="HydrogenAtom", realdim=True)
        return

    def set_wavefunction(self, n, l, m):
        # real dimensions for the reduced Bohr radius
        if self.realdim:
            a_0 = (4.0 * np.pi * epsilon_0 * hbar**2) / (m_e * e**2)
        else:
            a_0 = 1

        # normalization
        C = np.sqrt((2)/(n * a_0)**3 * factorial(n-l-1) /
                    (2 * n * factorial(n+l)))

        # putting it together
        def Psi(r, theta, phi):
            # radial component
            def R(r):
                rho = (2 * r) / (n * a_0)
                return np.exp(-rho/2) * (rho ** l) * genlaguerre(n-l-1, 2*l+1)(rho)
            return C*R(r)*sph_harm(m, l, phi,  theta)

        psi = CoordinateField3D(self.L, self.L, self.L, self.N, self.N, self.N)
        psi.fillContainer(Psi, (), coordinate_system="SPHERICAL")
        self.psi = psi
```

```
        return


if __name__ == "__main__":
    a_0 = (4.0 * np.pi * epsilon_0 * hbar**2) / (m_e * e**2)
    L = 20 * a_0
    # Initialize Particle
    particle = HydrogenAtom(L)
    # Initialize Eigenstate
    particle.set_wavefunction(2, 1, 1)
    # Compute J
    particle.find_probability_current()

    def plot_quiver():
        """
        Student must complete

        Requirements: Write a function to plot particl.J against its coordinates.
        You can reuse your plotting code from the analytical plots.
        Hint:  you will need to use python's slice notation to undersample,
        or else the program run time will be too long
        """

    plot_quiver()
```

Solution Scripts

Note: Please see the GitHub for copies of the scripts and animations.

I: Probability Current In 1D Systems

```
#Import neccessary modules
import numpy as np
import matplotlib.pyplot as plt
#Animation module
import matplotlib.animation as animation
#Embeds animations in jupyter notebooks
plt.rcParams["animation.html"] = "jshtml"
from scipy.fftpack import dst, idst
from scipy.constants import hbar
from numpy.polynomial.hermite import hermval
from math import sqrt

"""
Class QuantumSystem1D
Purpose: A generalized 1D quantum system
Parameters:
    - L: Length of the 1D system
    - M: Mass of the particle
    - N: Number of sample points
    - x: The sample point along the x axis
    - name: Unique name of the system (ex: PIB, SHO)
    - psi0: The initial state of the system
    - cn: The initial eigen decomposition coefficients

Methods:
    set_psi0: Initializes the state
        - psi0_func: A user defined function which computes psi0
        - args: the arugments of psi0_func
    generate_initial_cn: Initializes the eigen decomposition coefficients
        - n: the number of eigenstates you want to consider
    normalize: normalizes a given state
        - psi: the state to be normalized
    psi: returns the state at time t
        - t: The time to compute the state at
    psi_conj: return the complex conjugate of psi(t)
        - t: The time to find the state at
    psi_squared: returns |psi(t)|^2
        - t: The time to compute psi(t) at
    derivative: Takes the numerical derivative of a given psi(t)
        - psi_t: the state at time t
    probability current: Compute the probability current at a time t
        - t: The time to compute the probability current at
    create_animation: Creates an animation of the system
        - start: the starting time for the animation
        - end: the ending time of the animation
        - frames: the number of frames in the animation

"""
class QuantumSystem1D:

    def __init__(self, L, M, N=1000, name=''):
```

```
        self.x = np.linspace(0,L,N)
        self.N = N
        self.M = M
        self.L = L
        self.name = name
        return

    def set_psi0(self, psi0_func, args):

        self.psi0 = psi0_func(self.x, *args)
        self.psi0 = self.normalize(self.psi0)

    def generate_initial_cn(self,n):
        self.cn = np.zeros(n, dtype = np.complex128)
        for i in range(n):
            self.cn[i] = np.trapz(np.conj(self.eigenstates[i,:])*self.psi0, x=self.x)

    def normalize(self, psi_t):
        norm = np.trapz(np.abs(psi_t)**2, x = self.x)
        return psi_t/sqrt(norm)

    def psi_conj(self, t):
        return np.conj(self.psi(t))

    def psi_squared(self,t):
        return np.abs(self.psi(t))**2

    def psi(self, t):

        psi_t = np.zeros_like(self.x, dtype=np.complex128)
        for i in range(self.cn.size):
            psi_t += self.cn[i]*np.exp(-1j*self.En[i]*t/hbar)*self.eigenstates[i,:]
        return self.normalize(psi_t)

    def derivative(self, psi_t):
        dx = self.x[1] - self.x[0]
        psi_x = np.empty_like(psi_t)
        psi_x[0] = (psi_t[1] - psi_t[0])/(dx)
        psi_x[-1] = (psi_t[-1] - psi_t[-2])/(dx)
        psi_x[1:-1] = (psi_t[2:] - psi_t[:-2])/(2*dx)
        return psi_x

    def probability_current(self, t):
        psi_t = self.psi(t)
        psi_t_conj = np.conj(psi_t)
        A = hbar/(2*self.M*1j)
        term1 = psi_t_conj*self.derivative(psi_t)
        term2 = psi_t*self.derivative(psi_t_conj)
        J = A*(term1-term2)
        return J.real

    def create_animation(self, start, end, frames = 100):

        fig, ax = plt.subplots(nrows=3, ncols=1, sharex=True, figsize=[10,8])
        im, = ax[0].plot(self.x*1e9, self.psi_squared(start), color = 'black')
        im_real, = ax[1].plot(self.x*1e9, self.psi(start).real, color = 'blue', label = "real")
```

```
        im_imag, = ax[1].plot(self.x*1e9, self.psi(start).imag, color = 'orange', label = "imag")
        im_j, = ax[2].plot(self.x*1e9, self.probability_current(start), color = 'black')
        ax[0].grid()
        ax[1].grid()
        ax[2].grid()
        ax[2].set_xlabel("X [nm]")
        ax[1].legend()
        if(self.name == 'SHO'):
            ax[0].set_title("Wave Packet in SHO Potential Well")
        elif(self.name == 'PIB'):
            ax[0].set_title("Wave Packet in Inifinite Potential Well")
        else:
            ax[0].set_title("Wave Packet in Unknown Potential Well")
        ax[0].set_ylabel("Probability Density")
        ax[0].set_ylabel("$|\psi(x)|^2$")
        ax[1].set_ylabel("$\psi(x)$")
        ax[2].set_ylabel("J [$m^{-1}s^{-1}$]")
        plt.xlim(0,self.L*1e9)
        ax[2].set_ylim(-1*np.nanmax(im_j.get_ydata()),np.nanmax(im_j.get_ydata()))
        plt.tight_layout()
        #Updateable Text box for time
        ttl = ax[0].text(.78, 0.9, '', transform = ax[0].transAxes, va='center')

        def animate(i):

            t = start + i*(end-start)/(frames-1)
            psi_t = self.psi(t)
            im.set_data(self.x*1e9, np.abs(psi_t)**2)
            im_real.set_data(self.x*1e9, psi_t.real)
            im_imag.set_data(self.x*1e9, psi_t.imag)
            im_j.set_data(self.x*1e9, self.probability_current(t))
            #Update animation text
            ttl.set_text("t = {:.3f} fs".format(t*1e15))
            return im, im_real, im_imag, im_j,  ttl,

        return animation.FuncAnimation(fig, animate,\
                    frames=frames, blit=True)

"""
Class: PIB
Parent: QuantumSystem1D
Purpose: A particle in a box quantum system
Parameters:
    - eigenstates: an array holding the first n eigenstates of the system
    - En: an array holding the energy of each eigenstate
Methods:
    generate_eigenstates: Find the first n eigenstates of the system
        - n: The number of eigenstates to generate
"""
class PIB(QuantumSystem1D):

    def __init__(self, L, M, N=1000):
        QuantumSystem1D.__init__(self, L, M, N=N, name="PIB")
        return

    def generate_eigenstates(self, n):
```

```
        self.eigenstates = np.zeros((n,self.x.size), dtype=np.complex128)
        self.En = (np.arange(1,n+1)*np.pi*hbar/self.L)**2/(2*self.M)
        for i in range(n):
            pre_factor = sqrt(2/self.L)
            self.eigenstates[i,:] = pre_factor*np.sin((i+1)*np.pi/L*self.x)
        self.generate_initial_cn(n)

"""
Class: SHO
Parent: QuantumSystem1D
Purpose: A particle in a simple harmonic osccilator potential
Parameters:
    - eigenstates: an array holding the first n eigenstates of the system
    - En: an array holding the energy of each eigenstate
Methods:
    generate_eigenstates: Find the first n eigenstates of the system
        - n: The number of eigenstates to generate
"""
class SHO(QuantumSystem1D):

    def __init__(self, L, M, omega, x0, N=1000):

        QuantumSystem1D.__init__(self, L, M, N=N, name='SHO')
        self.w = omega
        self.x0 = x0
        return

    def generate_eigenstates(self, n):

        self.eigenstates = np.zeros((n,self.x.size), dtype=np.complex128)
        self.En = hbar*self.w*(np.arange(n) + 0.5)
        tmp = self.M*self.w/hbar
        gauss = (tmp/np.pi)**0.25*np.exp(-1/2*tmp*(self.x-self.x0)**2)
        for i in range(n):
            pre_factor = sqrt(1/(2**i * np.math.factorial(i)))
            self.eigenstates[i,:] = pre_factor*gauss*hermval(sqrt(tmp)*(self.x-self.x0), \
            [int(i == j) for j in range(i+1)])
        self.generate_initial_cn(n)

"""
A function to initialize a particle as a wave packet
"""
def psi0_func(x, x0, sigma, kappa):
    return np.exp(-1*(x-x0)**2/(2*sigma**2))*np.exp(1j*kappa*x)

#Constants
L = 1e-8 #m
x0 = L/2
sigma = 2e-10 #m
kappa = 5e10 #m^-1
M = 9.109e-31
n = 500

#Create Particle
particle = PIB(L, M, N=1000)
#Set inital conditions
particle.set_psi0(psi0_func, (x0, sigma, kappa))
```

```
particle.generate_eigenstates(n)
#Plot a few times
particle.create_animation(0, 2e-15, frames = 100)
ani = particle.create_animation(0, 2e-15, frames = 100)
#Uncomment to save as a video file
#ani.save('PIB.mp4', dpi=80, writer='imagemagick')
plt.show()

#Number of eigenstates used (any higher and the hermite polynomials have problems)
n = 200
omega = 2e15
#Create Particle
particle = SHO(L, M, omega, x0, N=1000)
#Set initial conditions
particle.set_psi0(psi0_func, (x0, sigma, kappa))
particle.generate_eigenstates(n)
#Plot a few times
ani = particle.create_animation(0, 2e-15, frames = 100)
#Uncomment to save as a video file
#ani.save('SHO.mp4', dpi=80, writer='imagemagick')
plt.show()
```

The animations output from this script are not shown here for obvious reasons. Instead we will show screenshots, the animations can be found here: `https://github.com/jacotay7/phy456-python-project/tree/master/figures`

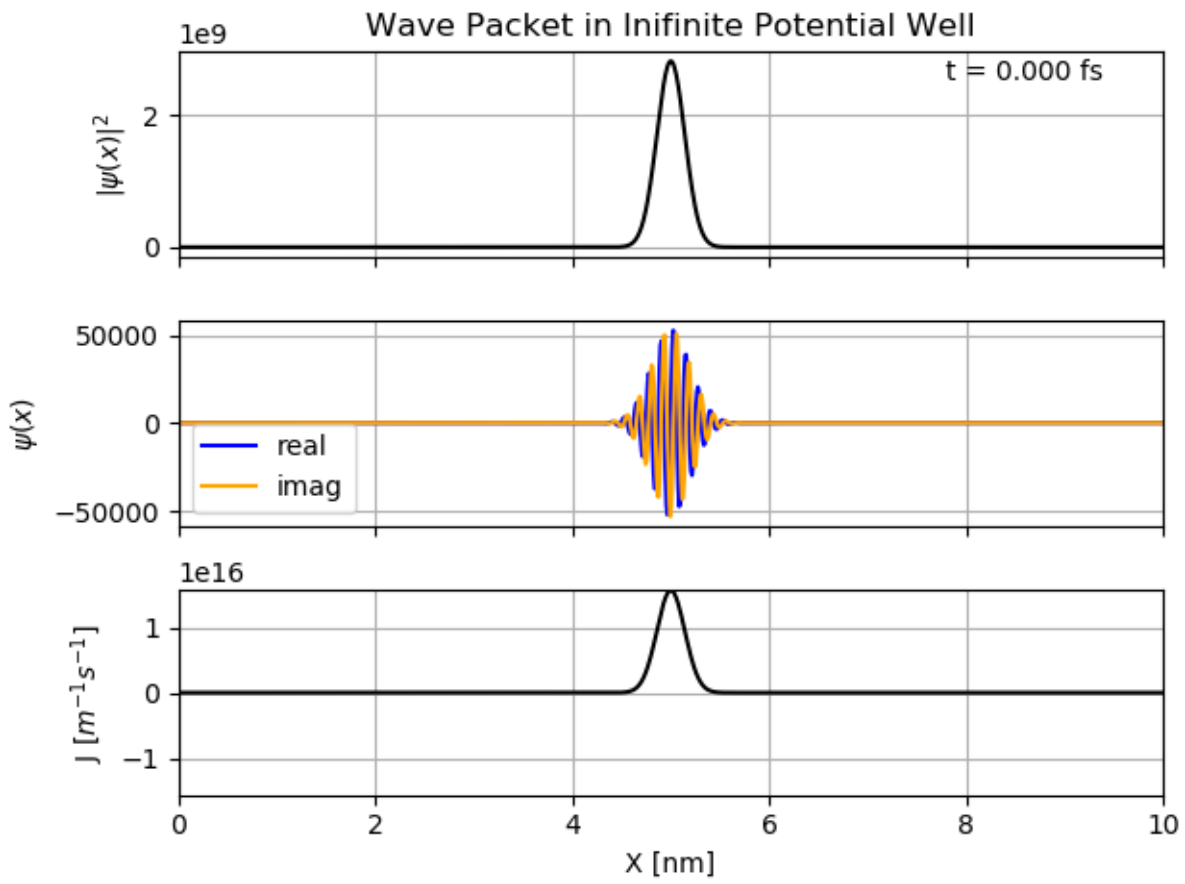

Figure 1: Screenshot of animation showing the probability current of a particle travelling in box.

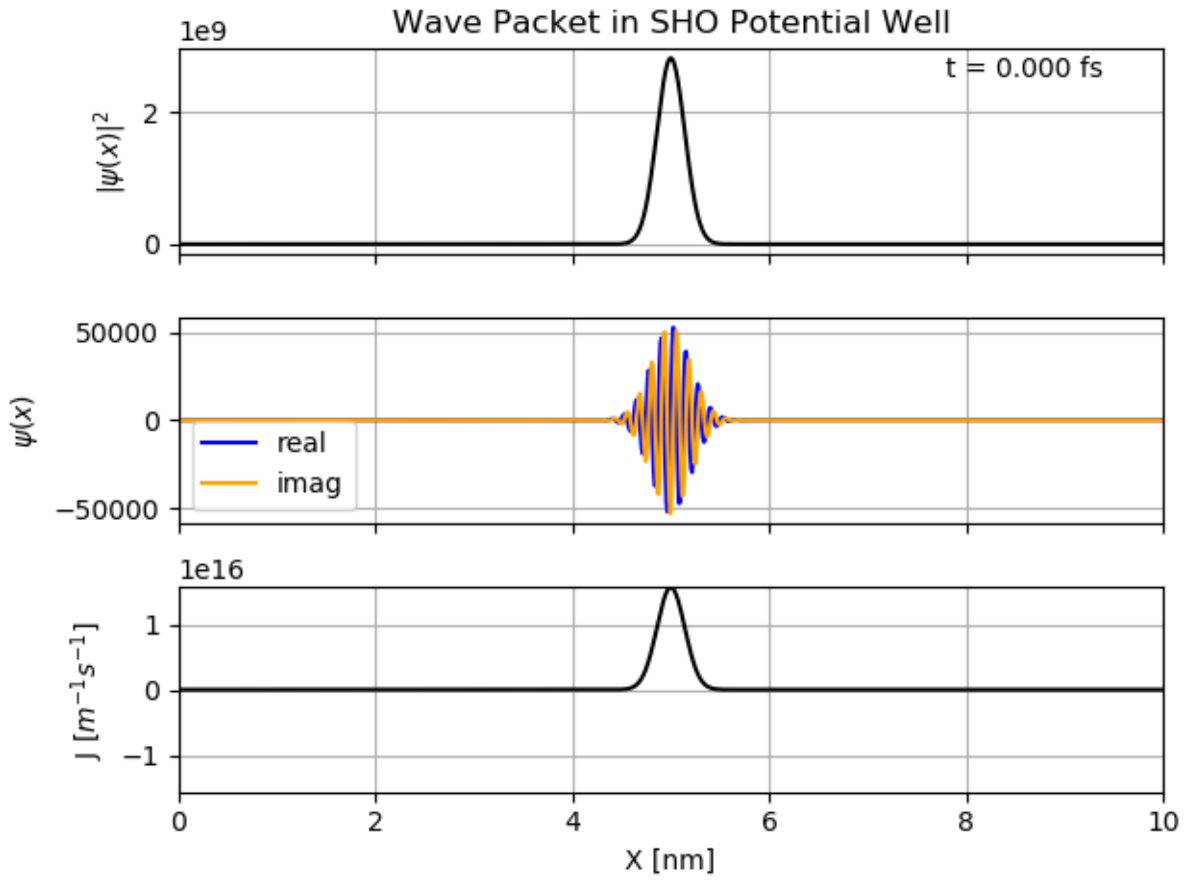

Figure 2: Screenshot of animation showing the probability current of a particle travelling in a quantum harmonic oscillator potential.

II: The Magnetic Moment of Hydrogen Eigenstates

```
from abc import ABC, abstractmethod
from math import sqrt

import matplotlib.animation as animation
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import axes3d
from numpy.polynomial.hermite import hermval
from scipy.constants import e, epsilon_0, hbar, m_e
from scipy.fftpack import dst, idst
from scipy.misc import factorial
from scipy.special import genlaguerre, sph_harm

from CoordinateField3D import CoordinateField3D

plt.rcParams["animation.html"] = "jshtml"

"""
Class QuantumSystem3D
Purpose: A generalized 3D quantum system
Parameters:
    - L: Linear size of the system
```

```
    - M: Mass of the particle
    - N: Number of sample points per dimension
    - x: The sample point along the x axis
    - name: Unique name of the system (ex: PIB, SHO)
    - psi0: The initial state of the system
    - cn: The inital eigen decomposition coefficients
Methods:
    set_psi0: Initializes the state
        - psi0_func: A user defined function which computes psi0
        - args: the arugments of psi0_func
    generate_initial_cn: Initializes the eigen decomposition coefficients
        - n: the number of eigenstates you want to consider
    normalize: normalizes a given state
        - psi: the state to be normalized
    psi: returns the state at time t
        - t: The time to compute the state at
    psi_conj: return the complex conjugate of psi(t)
        - t: The time to find the state at
    psi_squared: returns |psi(t)|^2
        - t: The time to coupute psi(t) at
    derivative: Takes the numerical derivative of a given psi(t)
        - psi_t: the state at time t
    probability current: Compute the probability current at a time t
        - t: The time to compute the probability current at
    create_animation: Creates an animation of the system
        - start: the starting time for the animation
        - end: the ending time of the animation
        - frames: the number of frames in the animation
    rotate_coords: rotates coordinates around
"""


class QuantumSystem3D(ABC):

    def __init__(self, L, M, N=250, name='', realdim=True):
        self.M = M
        self.L = L
        self.N = N
        self.name = name
        self.realdim = realdim
        return

    def normalize(self):
        norm = np.sum(self.psi.data)*psi.volume_element
        self.psi.data /= sqrt(norm)
        return psi/sqrt(norm)

    def psi_conj(self):
        psi_conj = self.psi.copy()
        psi_conj.data = np.conj(self.psi.data)
        return psi_conj

    def psi_squared(self):
        psi_squared = self.psi.copy()
        psi_squared.data = np.abs(self.psi.data)**2
        return psi_squared
```

```
    def gradient(self, psi):
        def grad_func(x, y, z, psi):
            psi_grad = np.empty(
                (x.shape[0], x.shape[1], x.shape[2], 3), np.complex128)
            volume_element = [x[1, 0, 0] - x[0, 0, 0],
                              y[0, 1, 0] - y[0, 0, 0], z[0, 0, 1] - z[0, 0, 0]]

            i = 0
            dv = volume_element[i]
            psi_grad[0, :, :, i] = (psi[1, :, :] - psi[0, :, :])/(dv)
            psi_grad[-1, :, :, i] = (psi[-1, :, :] - psi[-2, :, :])/(dv)
            psi_grad[1:-1, :, :, i] = (psi[2:, :, :] - psi[:-2, :, :])/(2*dv)

            i = 1
            dv = volume_element[i]
            psi_grad[:, 0, :, i] = (psi[:, 1, :] - psi[:, 0, :])/(dv)
            psi_grad[:, -1, :, i] = (psi[:, -1, :] - psi[:, -2, :])/(dv)
            psi_grad[:, 1:-1, :, i] = (psi[:, 2:, :] - psi[:, :-2, :])/(2*dv)

            i = 2
            dv = volume_element[i]
            psi_grad[:, :, 0, i] = (psi[:, :, 1] - psi[:, :, 0])/(dv)
            psi_grad[:, :, -1, i] = (psi[:, :, -1] - psi[:, :, -2])/(dv)
            psi_grad[:, :, 1:-1, i] = (psi[:, :, 2:] - psi[:, :, :-2])/(2*dv)

            return psi_grad
        psi_grad = CoordinateField3D(
            self.L, self.L, self.L, self.N, self.N, self.N)
        psi_grad.fillContainer(grad_func, (psi.data,))
        return psi_grad

    def find_probability_current(self):
        psi_conj = self.psi_conj()
        psi_grad = self.gradient(self.psi)
        psi_conj_grad = self.gradient(psi_conj)
        A = hbar/(2*self.M*1j)
        term1, term2 = np.empty_like(
            psi_grad.data), np.empty_like(psi_grad.data)
        for i in range(3):
            term1[:, :, :, i] = psi_conj.data*psi_grad.data[:, :, :, i]
            term2[:, :, :, i] = self.psi.data*psi_conj_grad.data[:, :, :, i]
        J = A*(term1-term2)
        self.J = J.real
        self.J_mag = np.sqrt(
            self.J[:, :, :, 0]**2 + self.J[:, :, :, 1]**2 + self.J[:, :, :, 2]**2)

    @abstractmethod
    def set_wavefunction(self, n, l, m, realdim=False):
        pass


"""
Class: HydrogenAtom
Parent: QuantumSystem3D
Purpose: A Hydrogen atom quantum system
Parameters:
    - eigenstates: an array holding the first n eigenstates of the system
```

```
    - En: an array holding the energy of each eigenstate
Methods:
    -set_wavefunction: specify eigenfunction that you want to plot
        - n, l, m: quantum numbers specifying the eigenfunction
"""


class HydrogenAtom(QuantumSystem3D):

    def __init__(self, L, N=250, realdim=True):
        QuantumSystem3D.__init__(
            self, L, m_e, N=N, name="HydrogenAtom", realdim=True)
        return

    def set_wavefunction(self, n, l, m):
        # real dimensions for the reduced Bohr radius
        if self.realdim:
            a_0 = (4.0 * np.pi * epsilon_0 * hbar**2) / (m_e * e**2)
        else:
            a_0 = 1

        # normalization
        C = np.sqrt((2)/(n * a_0)**3 * factorial(n-l-1) /
                    (2 * n * factorial(n+l)))

        # putting it together
        def Psi(r, theta, phi):
            # radial component
            def R(r):
                rho = (2 * r) / (n * a_0)
                return np.exp(-rho/2) * (rho ** l) * genlaguerre(n-l-1, 2*l+1)(rho)
            return C*R(r)*sph_harm(m, l, phi,  theta)

        psi = CoordinateField3D(self.L, self.L, self.L, self.N, self.N, self.N)
        psi.fillContainer(Psi, (), coordinate_system="SPHERICAL")
        self.psi = psi
        return


if __name__ == "__main__":
    a_0 = (4.0 * np.pi * epsilon_0 * hbar**2) / (m_e * e**2)
    L = 20 * a_0
    # Initialize Particle
    particle = HydrogenAtom(L)
    # Initialize Eigenstate
    particle.set_wavefunction(2, 1, 1)
    # Compute J
    particle.find_probability_current()
    # Plot

    def plot_quiver():
        fig = plt.figure()
        ax = fig.gca(projection='3d')
        s = 15  # down-sample a bit
        ax.quiver(particle.psi.x[::s, ::s, ::s],
                  particle.psi.y[::s, ::s, ::s],
                  particle.psi.z[::s, ::s, ::s],
```

```
                particle.J[::s, ::s, ::s, 0],
                particle.J[::s, ::s, ::s, 1],
                particle.J[::s, ::s, ::s, 2],
                length=2*a_0/np.max(particle.J[:]),
                normalize=False)

    # crop image a bit, might want to change this
    ax.set_xlim3d(-L/4, L/4)
    ax.set_ylim3d(-L/4, L/4)
    ax.set_xlim3d(-L/4, L/4)

    plt.show()

plot_quiver()
```

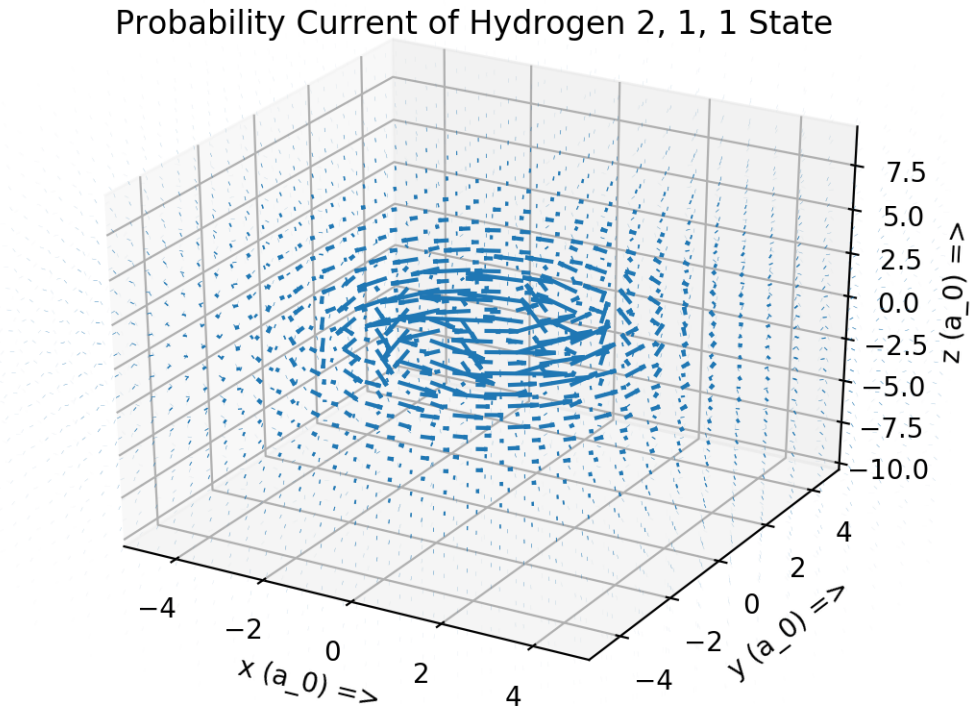

Figure 3: The probability current vector field of the 2,1,1 state. We omit other states since they are zero vector fields.

As expected, our result in Figure 3 points in the $\hat{\phi}$ direction an is independent of ϕ (see the above derivation). Interpreting the probability current of the electron as a classical current, we interpret the electron as producing an upward magnetic moment (where we have simply used the right-hand rule). This is exactly what we would expect for the 2,1,1 state of hydrogen (which has a *magnetic* quantum number of +1). The reader can verify the probability current is reversed for the 2,1,-1 state. Thus the probability current is one of many useful classical analogue in Quantum Mechanics.