Quantum Harmonic Oscillator Journal

Alec Burnett

May 2024

1 Introduction

My intro physics professor freshman year was obsessed with harmonic oscillators and encouraged all his students to do the same. He said that if you can understand ideal springs you can understand all of physics. This proved to be true when harmonic oscillation crept up in class again and again in lots of systems we derived in class. By the end of the year, he had several students including me submit lots of memes and artwork for our physics art project which was worth a surprising amount of our grade. I even tried to give him a giant spring from a car suspension at the end of the year but it turned out to be pretty hard to get rust off a helix shape. I tell this story to give some background on why I chose to do this honors project specifically. I think it is cool that these oscillators show up in lots of places and I wanted to become even more familiar with them. Coding something seems to be a great way to understand anything better. It also helps that the quantum harmonic oscillator has an exact solution and is simple enough to teach to a second year undergrad.

In this journal, I will be documenting my journey through coding a quantum harmonic oscillator, detailing my process and the challenges I face along the way. I started first with coding a simple harmonic oscillator and then a quantum version for comparison. This is meant to be an informal documentation of my process, not a research paper.

2 Simple Harmonic Oscillator

I felt the best way to start this project was to code a simple harmonic oscillator so that I could compare a classical simple harmonic oscillator to a quantum version of the same thing. This coding warm-up turned out to be more of a lesson in using Python to solve ordinary differential equations.

2.1 Equation

First, was to view a harmonic oscillator from the point of view of a 1D ideal spring system which uses Newton's 2nd law and Hook's law. I reviewed my Physics 2 notes to write the following system for an ideal spring.

$$F = m\frac{d^2x}{dt^2}, \quad F = -kx$$

Where k is the spring constant and m is mass.

$$\frac{d^2x}{dt^2} = \frac{-k}{m}x$$

$\frac{-k}{m}$ can be expressed as angular frequency $\omega = \sqrt{\frac{k}{m}}$

$$\frac{d^2x}{dt^2} = -\omega^2 x$$

2.2 Understanding The Code

1. After reminding myself of the basic math behind the system I wanted to do some research on how to code it. I found a sample code online that helped put me on the right track to understanding how to code the system. I first started to look at some relevant packages that could be used and the first one that jumped out was the integrated sub-package in SciPy. The standard packages we have used in class also seemed to be necessary like NumPy.

2. Next I figured it would be good to look more into the SciPy integrate subpackage in order to solve the DE. I could have solved this Equation by hand and then made a function based on the solution however, learning how to do these types of computations using Python will be useful in the future for things I won't want to solve by hand. I found in the integrate sub-package that the function solve_ ivp is for solving ordinary differential equations. The function needed in this case, is the function for the harmonic oscillator, a time span, an initial state, and t_eval which are times to store the solution. The harmonic oscillator function also needed to be vectorized for the solve_ ivp function to work

3. The confusing part of this was how to implement t_eval, which in order to work in the function needed to be an array. From the example code I found online the np.linspace function which creates an array of evenly spaced data points which is perfect for creating t values to sample to make a smooth graph.

4. The other confusing part of the solve_ ivp function was that the harmonic oscillator function needed to be vectorized, I just went over how to do this in PHYS 2504 so I was familiar with the process.

   $$\frac{d^2x}{dt^2} = -\omega^2 x$$

   $$y = \begin{pmatrix} x \\ v \end{pmatrix}$$

   Where v is velocity and x is position.

   $$\frac{dy}{dt} = \begin{pmatrix} v \\ -\omega^2 x \end{pmatrix}$$

   Now the system is vectorized.

2.3 Putting It Together

1. First I wrote the variables K and m for the spring constant and mass then I made a variable for ω from k and m. Then I made a time-span variable for the solve function that acts as the bounds for the integral that is being computed, here I chose t= 1 to t =50. The initial conditions are in the form of an array that represents the initial position and initial velocity, this is because that is how the SHO function that will come next returns a position and then a velocity. The Time evaluation uses np.linspace to create evenly spaced numbers representing a time that the solve function can utilize. For this system, the initial conditions are that the particle starts displacement 1 meter and has an initial velocity of 0. The time evaluation is also taking in 1000 measurements so the graph at the end looks smooth.

```
k=1 #spring constant
m=1 #mass of particle
omega =np.sqrt(k/m)
t_span = (0, 50)  # Limits of integration
initial_conditions = [1, 0]  # Initial position (x) and velocity
t_eval = np.linspace(t_span[0], t_span[1], 1000)
```

2. The simple harmonic oscillator function follows the vectorized differential equation for the simple harmonic oscillator defined above, all the function does is take in values for t and y, where t represents time, and y represents our vector. The function then returns the vectorized DE as an array

```
def sho(t,y):
    """
    y[0] = position x
    y[1] = velocity v
    """
return [y[1], -omega**2 * y[0]]
```

3. The solve ivp function is where the DE is solved, it intakes the vectorized DE, the time to integrate over, initial position/velocity, and the time evaluation and returns the solution to the DE.

```
sol = solve_ivp(sho, t_span, initial_conditions, t_eval=t_eval)
```

4. Plotting the solution is straightforward using mathplotlib, I plotted displacement (x) vs time (s). We can see the graph is what we would expect from an undamped ideal spring system.

```
plt.plot(t_eval, sol.y[0])
plt.xlabel('Time')
plt.ylabel('Displacement')
plt.title('Simple Harmonic Oscillator')
plt.grid(True)
plt.show()
```

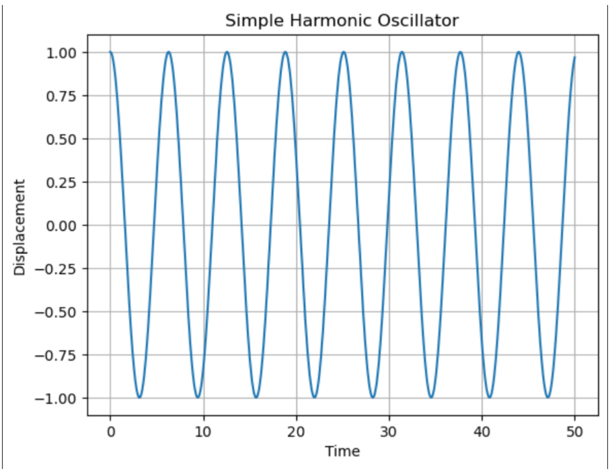

Figure 1: Simple Harmonic Oscillator initial conditions [1, 0]

2.4 Reflection

Although I coded the simple harmonic oscillator to warm up for the main part of this project, I did encounter some problems while coding it. The majority of the time was spent trying to understand how to make my code fit the parameters of the solve ivp function and understanding why those parameters were needed for the output. I had the sample code to look at that I found online however the website did not explain the solve ivp function very well or the time evaluation. The Scipy and Numpy websites were very valuable for explaining the functions.

3 Quantum Harmonic Oscillator

In this section, I aimed to code a quantum harmonic oscillator from the point of view of a quantum bead being pushed and pulled by a spring-like force. This would make it easy to compare the final quantum harmonic oscillator to the classical harmonic oscillator.

3.1 Solution

To start this portion of the project I wanted to get more familiar with the quantum harmonic oscillator. I covered the concept in modern physics so that seemed like a good place to review my notes. In modern I used a method of Hermite polynomials

$$-\frac{\hbar^2}{2m}\,\psi''(x) + U(x)\,\psi(x) = E\,\psi(x)$$

Here I used Schrodinger's time-independent equation because the harmonic oscillator is time-independent. Here U(x) is the potential energy function.

$$U(x) = \frac{1}{2}m\omega_0^2\,x^2$$

Here natural angular frequency $\omega_0 = \sqrt{\frac{k}{m}}$ where K is the spring constant and m is mass

Next, U(x) needs to be in terms of $\hbar$.

$$\begin{aligned} U(x) &= \frac{1}{2} m\omega_0^2 x^2 \\ &= \left(\frac{1}{2}\hbar\omega_0\right)\left(\frac{m\omega_0}{\hbar}\right) x^2 \\ &= \left(\frac{1}{2}\hbar\omega_0\right)\left(\sqrt{\frac{m\omega_0}{\hbar}}\, x\right)^2 \end{aligned}$$

Then We can make U in terms of a dimensionless variable

$$\tilde{U}(u) = \frac{1}{2}(\hbar\omega_0)\, u^2, \quad u = \sqrt{\frac{m\omega_0}{\hbar}}\, x$$

now $\tilde{U}$ can be put back into the time-independent Schrodinger equation

$$-f''(u) + u^2 f(u) = \left(\frac{E}{\frac{1}{2}\hbar\omega_0}\right) f(u), \quad f(u) = \psi(x)$$

The solution to this is DE is

$$f(u) = H_n(u)\, e^{-\frac{1}{2}u^2}, \quad E_n = (2n+1)(\frac{1}{2}\hbar\omega_0)$$

Here $H_n(u)$ is a Hermite polynomial based on energy level n

3.2 Thinking About The Code

The vision I want for the code is similar to the simple harmonic oscillator. I want to be able to input the mass of the object and the spring constant and see the wave function and probability as an output. I also want a way to see the potential energy function as well as the energy levels on the same graph as the wave function and probability.

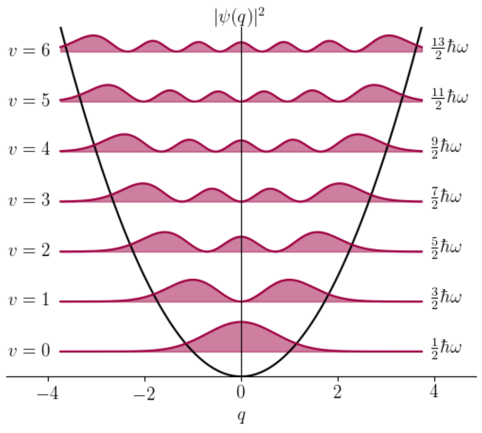

Figure 2: What I would like the code to output for the probability of finding the particle

3.3 The Code

Just as I did with the simple harmonic oscillator I found some example code online to see what packages would be useful, this sample code came from LibreTexts: Chemistry. The big package that I used for the code was the Numpy Hermite polynomials package. This would allow any energy level to be graphed for the system and each polynomial wouldn't have to be coded by hand. The sample code was very simple graphically compared to what I wanted to eventually output with my code, the sample code only graphed one energy level at a time and only displayed the shape of the wave function.

1. The first place I started with my code was to initialize the constants and bounds. For simplicity's sake, I made all the constants 1, $\hbar$ included. Then I initiated the quantum numbers in a set, I can add as many quantum numbers as I wanted to this set and the output graph would change accordingly. Here I choose quantum numbers from ground state 0 to 10. Next, I needed bounds on a 1D space that the quantum bead is oscillating through. I just choose to call this space X and X goes from -5 to 5 arbitrary units. Then, I needed to make the space in between the bounds, Here I used the same method I used to make the time scale for the simple harmonic oscillator using Numpy linspace function, creating an array from -5 to 5 with 1000 evenly spaced steps.

```
h_bar = 1
m = 1
w = 1
quantum_numbers = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
x_bounds = (-5, 5)
x = np.linspace(x_bounds[0], x_bounds[1], 1000)
```

2. Next I thought it would be best to define a Hermite polynomial function to return a Hermite polynomial based on our dimensionless space u. These Hermite polynomials are part of the wave function so the Hermite polynomial function should be defined before the wave function. The function has two parameters, representing a position x and the other an integer n to iterate through. The function returns the numpy.polynomial.hermite.hermval function to create a Hermite series

$$p(u) = c_0 * H_0(u) + c_1 * H_1(u) + \ldots + c_n * H_n(u)$$

This series once computed at u, can be used to calculate the wave function.

```
def hermite(x,n):
    u = np.sqrt(m*w/h_bar)*x
    herm_coeffs = np.zeros(n+1)
    herm_coeffs[n] = 1
    return Herm.hermval(u, herm_coeffs)
```

3. Defining the wave function was pretty straightforward. The function takes the same parameters as the Hermite function, the position u, and a value n to iterate through. The function initializes u and H and returns $H_n(u)\, e^{-\frac{1}{2}u^2}$

```
def wave_function(u,n):
    u = np.sqrt((m*w)/h_bar)*x
    H = hermite(u,n)
    wave = H * np.exp((-0.5) * (u)**2)
    return wave
```

4. Next the potential energy function is defined. The potential energy is only used to be placed on the graph to show the potential well of the system. The potential energy is defined by $U(u) = \frac{1}{2}m\omega_0^2 u^2$. This was straightforward to implement.

```
def potential_energy(x):
    return 0.5 * m * w**2 * x**2
```

5. An important step for visually showing the wave functions is to normalize the wave function. This is done by square rooting the integral of the wave function squared over our space in this case. Here I used the scipy.integrate.simps function to integrate the wave function over the array x. This will result in the square of the wave function having an area of one which is important for probability.

```
def normalize_wavefunction(psi, x):
    psi_squared = np.abs(psi)**2
    integral = simps(psi_squared, x)
    return psi / np.sqrt(integral)
```

6. Graphing the wave functions requires iterating through all the quantum numbers that need to be displayed. Here is where all the n values in the functions will be utilized, calculating the normalized wave function for each quantum number and displaying. Using mathplotlib I made the figure size and then iterated through quantum numbers calculating the wave function and the normalized wave function as well as the energy given by $E_n = (2n+1)(\frac{1}{2}\hbar\omega_0)$. Then I plotted the normalized wave function plus the energy vs the position u. I also added the potential energy function as well as the horizontal energy levels.

```
plt.figure(figsize=(10, 10))
for n in quantum_numbers:
    u = np.sqrt((m * w) / h_bar) * x
    psi = wave_function(u, n)
    psi_normalized = normalize_wavefunction(psi, u)
    energy = (2*n + 1) * (h_bar * w* .5)
    plt.plot(u, psi_normalized + energy, label=f'n={n}', linestyle=
   '-', color='C{}'.format(n))
    plt.hlines(energy,x_bounds[0],x_bounds[1],color='grey')
plt.plot(u, potential_energy(u), linestyle='--', color='black',
   label = 'Potential Energy')

plt.title('Quantum Harmonic Oscillator Wave Functions')
plt.xlabel('Position u')

plt.ylabel('Normalized Wave Function / Potential Energy')
plt.legend()
plt.grid(True)
plt.show()
```

7. Next I did the same graph but the wave function was squared to give the probability density of finding the particle based on its u position

```
plt.figure(figsize=(10, 10))
for n in quantum_numbers:
    u = np.sqrt((m * w) / h_bar) * x
    psi = wave_function(u, n)
    psi_normalized = normalize_wavefunction(psi, x)  # Normalize
   the wave function
```

```
        energy = (2*n + 1) * (h_bar * w* .5)
        plt.plot(u, np.abs(psi_normalized)**2 + energy, label=f'n={n}',
         linestyle='-', color='C{}'.format(n))
        plt.hlines(energy,x_bounds[0],x_bounds[1],color='grey')
    plt.plot(u, potential_energy(u), linestyle='--', color='black',
       label='Potential Energy')

    plt.title('Quantum Harmonic Oscillator Probabilites')
    plt.xlabel('Position u')
    plt.ylabel('Probabilites/ Potential Energy')
    plt.legend()
    plt.grid(True)
    plt.show()
```

3.4 The Result

The resulting graphs are very similar to online photos of the quantum harmonic oscillator that can be found online like the one I showed in section 3.2. While maybe not textbook-worthy my graphs convey the system quite well. The wave function graph seems to have some overlap on the curves but I believe that is because of scaling and not an issue with the math in the code.

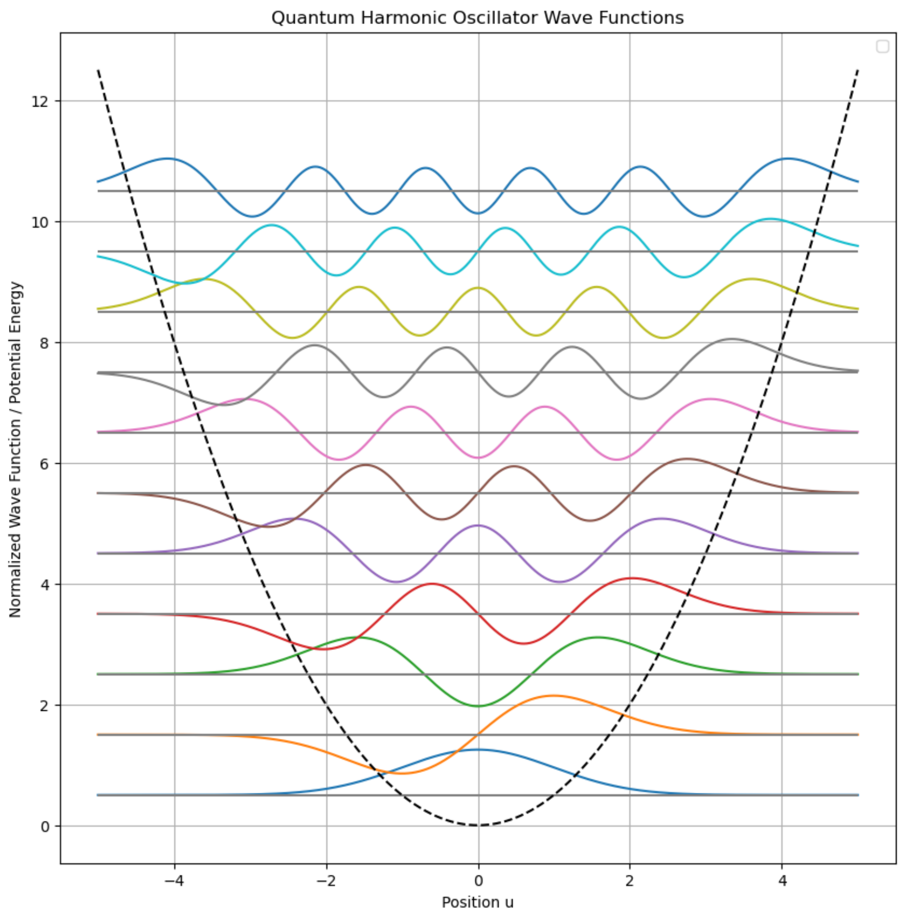

Figure 3: Graph of the wave functions with energy levels 0-10, Legend removed so all the waves are visible

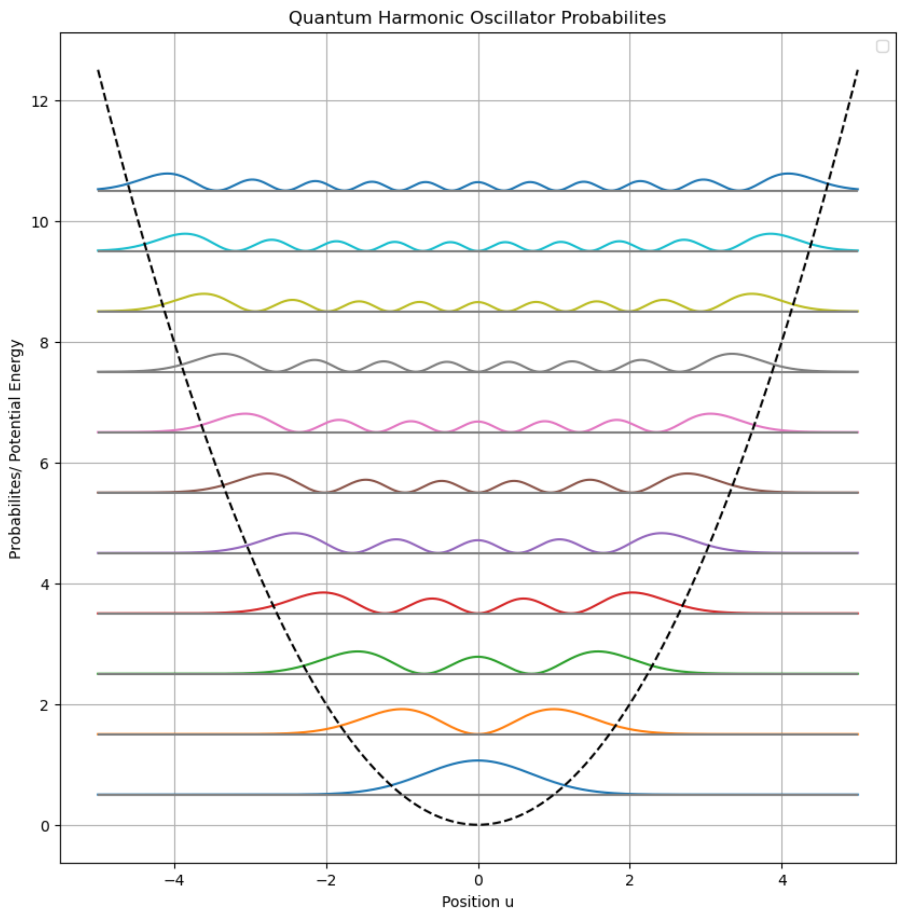

Figure 4: graph of the probability densities, Quantum numbers 0-10, Legend removed so all the waves are visible

3.5 Challenges

By far the most challenging part of this code was trying to understand the Hermite series function from Scipy. In Modern Physics, we were just given the Hermite polynomials without any mention of a series made out of the polynomials. There wasn't a function to output the Hermite polynomials so I needed to use the series. Luckily the sample code showed how to use the hermval function and after reading over all the functions in the package it made sense why hermval was the best one to use for this application. Another challenge was keeping track of all the notions and variables for the system, all the u values in place of x values as well as the n values for iteration. While it was a challenge it was interesting to discover more about what is possible using the math plot library. Specifically how to add multiple curves to the same graph, add horizontal lines, and different colors/line types. The normalization function was also a challenge. First realizing that normalization was important in the first place for scaling the graph so all the curves fit as well as understanding the wave function needed to be normalized so the probability density makes sense since the probability needs an area equal to one. After doing some research I found the best method to find the integral would be to use the scipy.integrate.simps because of the data type of x and u. After this hiccup, it was easy to implement the normalization formula

4 Quantum vs Classical

Since I coded both Quantum and classical harmonic oscillators I wanted to compare the two. The issue with this is it wouldn't make any sense to just put the graphs for the quantum harmonic oscillator over the graph of the classical harmonic oscillator. The axes are different and comparing a path of motion directly to a wave function wouldn't be very productive. What I've seen online and in my classes is a comparison based on probability densities so I decided to go that route. My goal for this comparison was to compare multiple probability densities based on multiple quantum numbers to the classical analog by graphing.

4.1 Equation

I have never learned about classic probability density in a physics class because it seems like it is really only useful for making comparisons to quantum systems. While there wasn't a lot of documentation about classical probability density for a harmonic oscillator online there were a few equations I could try. There was a Wolfram page that had a specific equation.

$$\rho(x,E) = \frac{1}{\pi}\sqrt{\frac{k}{2E - kx^2}}\,.$$

Here rho is the probability density based on position and energy. K is the spring constant that has been neglected in the quantum oscillator code.

4.2 Code

1. I first started by defining a classic probability function in the same file as the quantum harmonic oscillator Jupyter Notebook. It didn't seem to make much of a difference in my tests which definition of energy I used so I ended up using the h bar definition of energy so it would be easy to change the quantum number for both functions in the final graph. I also used the dimensionless variable u as the position for the classic probability density because it would make the code simpler and wouldn't change anything because of the constants.

   ```
   def classic_prob(x, h_bar, n, m):
       energy = (2*n + 1) * (h_bar * w* .5)
       prob = (1 / np.pi) * np.sqrt(k / ((2 * energy) - (k * (x**2))))
       return prob
   ```

2. Next I graphed using the math plot library and initially used quantum number 0 to create the first graph. First, the classic probability needs to be called and assigned to a variable, and the same with the wave function. Then the graphing process was straightforward.

   ```
   prob=classic_prob(u,h_bar,0,m)
   plt.plot(u,prob,label='Classical Probability', linestyle='--')
   psi = wave_function(u, 0)
   psi_normalized = normalize_wavefunction(psi, x)
   plt.plot(u, np.abs(psi_normalized)**2,label='Quantum Probability')
   plt.title('Comparison of SHO and QHO Probability Densities')
   plt.xlabel('Position (u)')
   plt.ylabel('Probability Density')
   ```

```
plt.legend()
plt.grid(True)
plt.show()
```

This code was edited to obtain the results for the other quantum numbers.

4.3 Result

The result for quantum number zero is

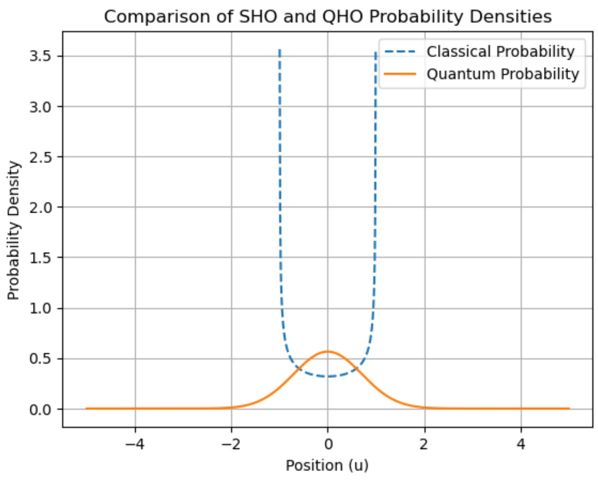

Figure 5: Probability density comparison, Quantum Number = 0

Here each of the probability densities is represented by the area under each respective curve. This graph for classical probability is what is expected. The trend continues for the other quantum numbers

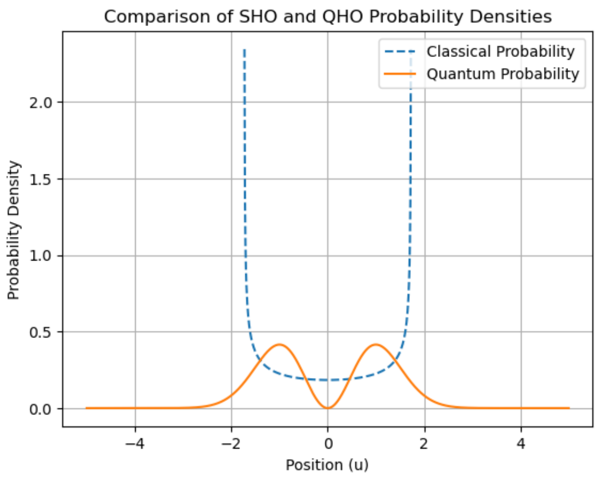

Figure 6: Probability density comparison, Quantum Number = 1

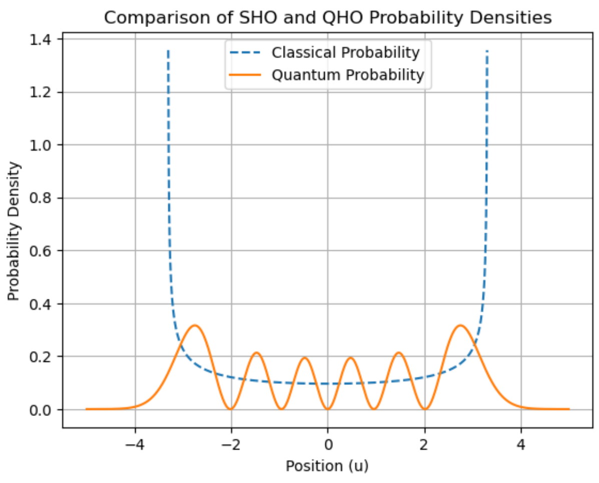

Figure 7: Probability density comparison, Quantum Number = 5

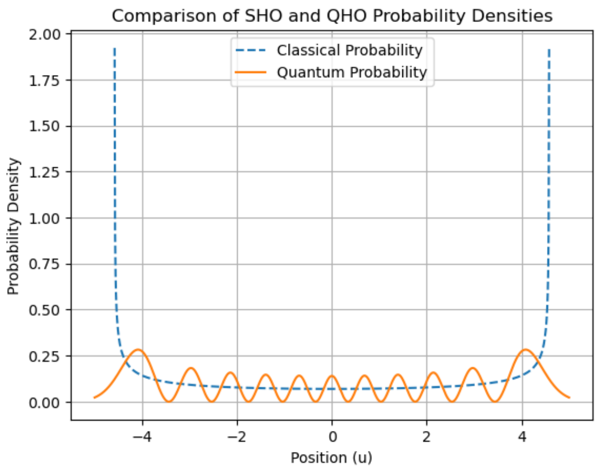

Figure 8: Probability density comparison, Quantum Number = 10

5 Conclusion

I learned a lot from this project, mostly solving ordinary differential equations using Python and graphing skills. Another skill I learned while doing this journal was how to use latex properly well as using cool latex packages like the one for inserting code and having it look like it should in an integrated development environment. As far as the actual coding goes I gained valuable experience in utilizing packages by reading the package online documentation which made the whole project a lot easier, and I now understand the importance of Numpy and Scipy. In terms of the content of the project, I am much more comfortable with the quantum harmonic oscillator and oscillators in general. This is a far cry from learning about them in modern physics where we covered the content very quickly, in this project I got the opportunity to slow down and really understand the concepts behind how to solve the system and why it works.

6 Citations

Medium.com
Numpy.org
chem.libretexts.org
scipy.org
wolfram.com
matplotlib.org