3rd Year Computational Physics : Jessica Murphy

09/11/2023

Assignment 3

This python notebook investigates numerical techniques, initially performing multiple integrations under a curve between x = a and x = b. This is because the aim is to numerically solve a **wavefunction problem**, namely Schrodinger's equation for a particle trapped infinite potential well. Given a potential well of width 2a, this gives a ground state solution given by a wavefunction $\psi(x) = \sqrt{\frac{1}{a}}cos(\frac{\pi x}{2a})$. Taking a = 1/2 as an example to give a unit width well, this results in a probability equation P, of:

$$P = \int_{1/3}^{2/3} \psi^*(x)\psi(x)\,dx$$

So the integral we are solving is:

$$f(x) = \int_{-1/3}^{1/3} cos^2(\pi x)\,dx$$

This integral can be solved analytically, with an answer of P = $\frac{2}{3} - \frac{4\pi - 3\sqrt{3}}{12\pi} \approx$ **0.4711655571887813**, this is a good basis to know throughout, to identify any outliers.

Background Information: The rectangle rule is the simplest way to integrate a well behaved function. In the rectangle rule, the area under the curve is approximated by the sum of the area of a number of rectangles. The height of each rectangle, $f(x_n)$) is set by the value of $f(x)$ at the left edge of the rectangle. The width of each rectangle, h, arrives by dividing the range of integration into n rectangles

$$\frac{b-a}{n} = h$$

$$I = \int_a^b f(x)\,dx \approx \sum_{i=0}^{n-1} f(x_i)h$$

Task 1: Rectangle Rule

The objective of task one is to directly programme the rectangle rule (i.e. do not use a library routine for this), with n = 100 iterations to give a numerical estimate for the integral of $f(x) = cos^2(\pi x)$. . Define f(x) as a function.

1. Import the python library NumPy as np, which is the natural log funtion, and maptplotlib.pyplot for the purpose of plotting
2. Define a function $f(x) = cos^2(\pi x)$ as advised
3. Define the limits of integration that are goven in the background info: -1/3, 1/3
4. Define a function that caluclates the rectangle rule, taking the number of iterations n as a parameter
    - This creates a linearly spaced array between given integral limits of size n, which is the number of iterations.
    - Then create an array for f(num) of each value of num in the previous array
    - Now use the formula for the reactangle rule, b-a/n = h, to convert each value of f(num) into an area element
5. Print the estimated value of 100 iterations of the integral using the recatangle rule

In [1]:
```
# assignemnt 3
# task 1

# import python libraies needed
import numpy as np # import NumPy for the natural log function
import matplotlib.pyplot as plt # import matplotlib for plotting graphs

# Define the given function f(x) which is equal to cos(πx^2)
def f(x):
    return np.cos(np.pi * x) ** 2 #returns the given function

# Define the interval [a, b] for integration (given)
a = -1/3 # Set the lower bound of the integration interval
b = 1/3 # Set the upper bound of the integration interval

# rectangle rule for numerical integration, (b-a)/n = h
# define a funtion to calaculate the rectangle rule
# This function estimates the value of the integral of f(x) within the given limits, using the number of iterations as a variable
def rectangle_rule(n):
    iteration_length = int(n) # convert the input 'n' into an integer to quantise the length of iterations.
    num = np.linspace(a, b, iteration_length) #create a linspace array with the number of iterations
    # calculate the corresponding function values at each point in the 'num' array.
    y = f(num) # create an array of f(x) values from that linspace array
    # calculate the numerical integral using the rectangle rule,
    # where each 'y' value is multiplied by the width 'h' and summed up.
    return (y * (b - a)/ n).sum() #convert each f(x) into an area element

# giving a numerical estimate of n = 100 iterations
# rectangle_rule(100)
print('Accurate value of the integral= {0}'.format(rectangle_rule(100)))
```

```
Accurate value of the integral= 0.46810021078092356
```

The difference between this value and teh true value is

$$0.4711655572 - 0.46810021078092356 = 0.0030653464078577164$$

which is small relative to the answer itself. This shows this method is quite accurate.

In [2]:
```
# difference of values
d = abs(0.4711655571887813 - 0.46810021078092356)
print(d)
```

```
0.0030653464078577164
```

Task 2

The objective of this task is to integrate using integration of quadratures. For an integration of higher efficacy, use the integration package *scipy.integrate.quad* to calculate a more accurate value of the integral

1. Import the integrate package from SciPy
2. Calculate an accurate integral with uncertainty
3. Print this accurate value with its uncertainty

In [3]:
```
# task 2
# import the 'integrate' module from SciPy to get integration of quadrature package
from scipy import integrate

# calculate the accurate integral using scipy.integrate.quad
# capture both the integral value and the error
integral_quad, error = integrate.quad(f, a, b)

# print the integral value and its error
print('Accurate value of the integral= {0} ± {1:.5}'.format(integral_quad, error))
```

```
Accurate value of the integral= 0.4711655571887813 ± 5.231e-15
```

This is the same as the 'true' value given above (0.4711655571887813) which shows this is a highly accurate method.

Task 3: Order of method

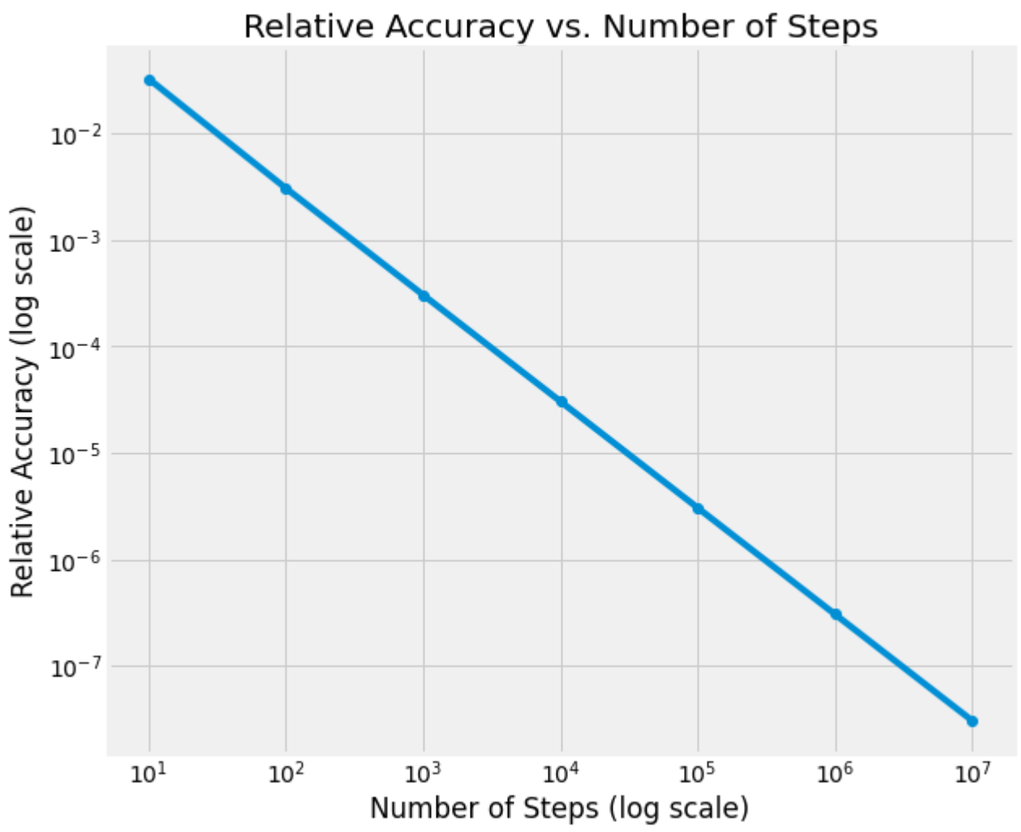
A numerical method is first order if the accuracy improves linearly with the number of steps and generally then, is Nth order if the accuracy is $\eta \propto n^{-N}$ where

$$\eta = \frac{|measured - expected|}{expected}$$

Using the accurate expected value for the integral from Task 2, investigate the performance of the rectangle rule by varying the number of iterations by factors of 10 (i.e. 10, 100, 1000, 10000,....) steps. Plot the relative accuracy of the algorithm versus the number of steps and hence identify the order of the algorithm*

1. Create a logspace to use for a number of iterations varied by a factor of 10
2. Create a for loop to iterate along the x-axis and append each f(x) value to a list y
3. Find the accuracy using the eta function
4. plot the relative accuracy vs the number of iterations
5. Verify the order of the fraph by using np.polyfit

In [4]:
```
x=np.logspace(1,7,7) #create logspace to use for number of iterations of factors of 10
y=[] #initialise a list to store the values of our rectangular rule integration
for integral_quad in x: #for each of our number of iterations
    y.append(rectangle_rule(integral_quad)) #find the corresponding integration estimation and append it to y

task_2 = 0.4711655571887813 # usin gvalue from task 2

accuracy=abs(np.array(y) - task_2) #finding the accuracy of each of our integration values

# Plot the relative accuracy versus the number of steps
plt.figure(figsize=(10, 8))
plt.style.use('fivethirtyeight') # for background use style sheet from matplotlib
plt.xscale('log') # log applied to all x-values (number of steps)
plt.yscale('log') # log applied to all x-values (relative accuracy)
plt.plot(x, accuracy, 'o-') # plotting x,y,marker
plt.xlabel('Number of Steps (log scale)')
plt.ylabel('Relative Accuracy (log scale)')
plt.title('Relative Accuracy vs. Number of Steps')
#plt.grid()
plt.show()
```



A log-log graph was applied as applying log to both sides of the equation $\eta \propto n^{-N}$ gives

$$log\eta = -Nlogn$$

and using y=mx+c, it is clear that we find the order of the numerical method by analysing the slope

In [5]:
```
# Verify the order of the graph
[m,c] = np.polyfit(np.log(x), np.log(accuracy), 1)
# Fit a linear model (y = mx + c) to the logarithmic data of 'x' and 'accuracy'.
# 'm' will be the slope, and 'c' will be the y-intercept of the fitted line (which we dont need in this instance)

# print the calculated slope of the linear fit with 6 decimal places
# then provide an interpretation of the slope indicating the order of the relationship in the context of the problem.
print('The slope of the graph = {0:.6}'.format(m))
print('Therefore the plot indicates an order of the negative of the slope, i.e N = {0:.0f}'.format(-m))
```

```
The slope of the graph = -1.00353
Therefore the plot indicates an order of the negative of the slope, i.e N = 1
```

Task 4

Monte Carlo Method 1

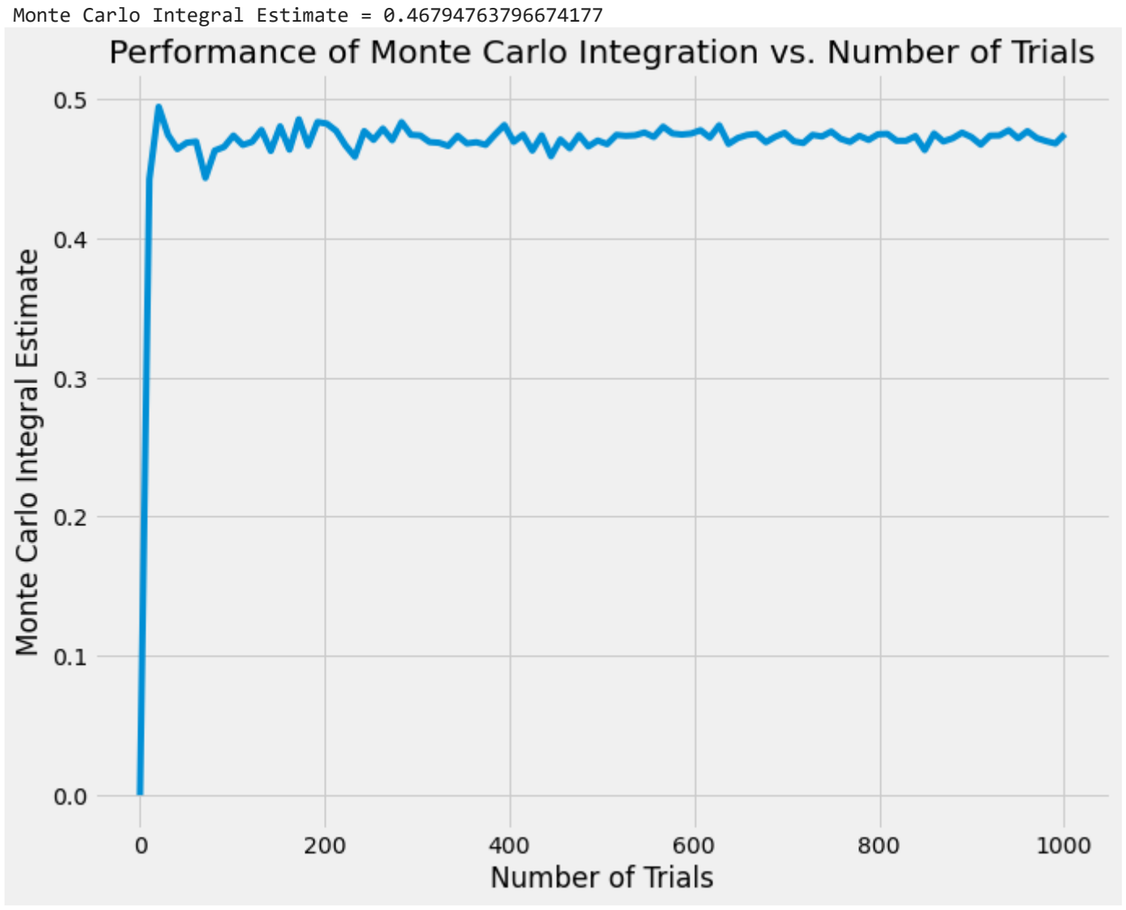
The purpose of this task is to use the monte carlo algorithm to approximate the area under the integral, and then make a plot of the performance of the Monte Carol method versus number of trials.

1. Specify integrationlimits in x to give a range xmin to xmax
2. Create an array for 100 x values, state there is 1000 trials, and evaluate ymax, which is a function of the x values. **There is now a box area defined by these x and y values**
3. Initialise the summation variable to be zero
4. Create a for loop to iterate through the number of trials stated
   - Create random number pair (x, y) by generating a random x in the range xmin to xmax and a random y in the range 0 and ymax
   - Test if the random number pair lies above or below the function f(x) by evaluating by substitution whether f(x) is less than or equal to y
   - If the random number pair is under or on the curve, add one to the summation. If it is above the curve, do not add to the summation
5. The integral is then found from: summation/trials ' integral/box area, i.e. integral = (summation/trials) * box area
6. Print the Monte Carlo estimate of the area under the graph
7. For the purposes of plotting, rewrite for loop including:
   - a range for the number of trials
   - a list for the estimates of the integral, namely 'integral_estimates'
8. Plot the graph

In [6]:
```
# step 1: specifying integration limits
# Integration limits
xmin = -1/3
xmax = 1/3

x_values = np.linspace(-1/3, 1/3, 100) #creating an array of 100 linearly spaced values between the given limits of integration
```

```python
# Number of trials (increase this for better accuracy)
trials = 1000

# Step 2: Find ymax by evaluating the function within the integration limits
ymax = max(f(x_values))

# Initialize the summation variable
summation = 0

# step 3,4,5,6,7
# Perform the Monte Carlo integration
# Loop through a specified number of trials denoting as arbitray variable name '_' because it doesn't come up again
for _ in range(trials):
    # generate random 'x' and 'y' values within the integration limits and a bounding box
    x = np.random.uniform(xmin, xmax)
    y = np.random.uniform(0, ymax)
    if f(x) >= y: # evaulating whether f(x) is less than or equal to y, i.e. under or on the curve
        summation += 1 # following method in instructions

# calculate the area of the bounding box
box_area = (xmax - xmin) * ymax

# Calculate the integral using the Monte Carlo method
integral = (summation / trials) * box_area

# print the Monte Carlo integral estimate with 5 decimal places
print('Monte Carlo Integral Estimate = {0}'.format(integral))

# Create a plot to show the performance versus the number of trials
trials_range = range(1, 10000, 100)
integral_estimates = [] # initialising an empty list to store future values

# iterate through different numbers of trials for performance comparison
# same method as above just adding an array 'integral_estimates' for plotting
for num_trials in trials_range:
    summation = 0
    for _ in range(num_trials):
        x = np.random.uniform(xmin, xmax)
        y = np.random.uniform(0, ymax)
        if f(x) >= y:
            summation += 1 # increment the summation if the condition is met
    integral_estimate = (summation / num_trials) * (xmax - xmin) * ymax
    integral_estimates.append(integral_estimate)


# Plot the performance
plt.figure(figsize=(10, 8))
plt.style.use('fivethirtyeight') # for background use style sheet from matplotlib
# forplotting, using a linearly spaced array for the x values
plt.plot(np.linspace(0, 1001, 100), integral_estimates, '-')
plt.xlabel('Number of Trials')
plt.ylabel('Monte Carlo Integral Estimate')
plt.title('Performance of Monte Carlo Integration vs. Number of Trials')
#plt.grid()
plt.show()
```

```
Monte Carlo Integral Estimate = 0.46794763796674177
```



In [7]:
```python
diff = abs(0.47661333496612585- 0.4711655571887813)
print(diff)
```

```
0.005447777777344565
```

The difference between the true value and monte carlo value is 0.005447777777344565.

This is notably larger than the difference when using the triangle rule, even though they had the same number of trials. This may imply the monte carlo being less accurate for this scenario.

It appears the graph seems to be 'shaky' and we can't clearly see an approximation number. So, I will rewrite the code for the purposes of 1000 trials

Rewiting graph to get large number of approximations for better visualisation

Monte Carlo Method 2

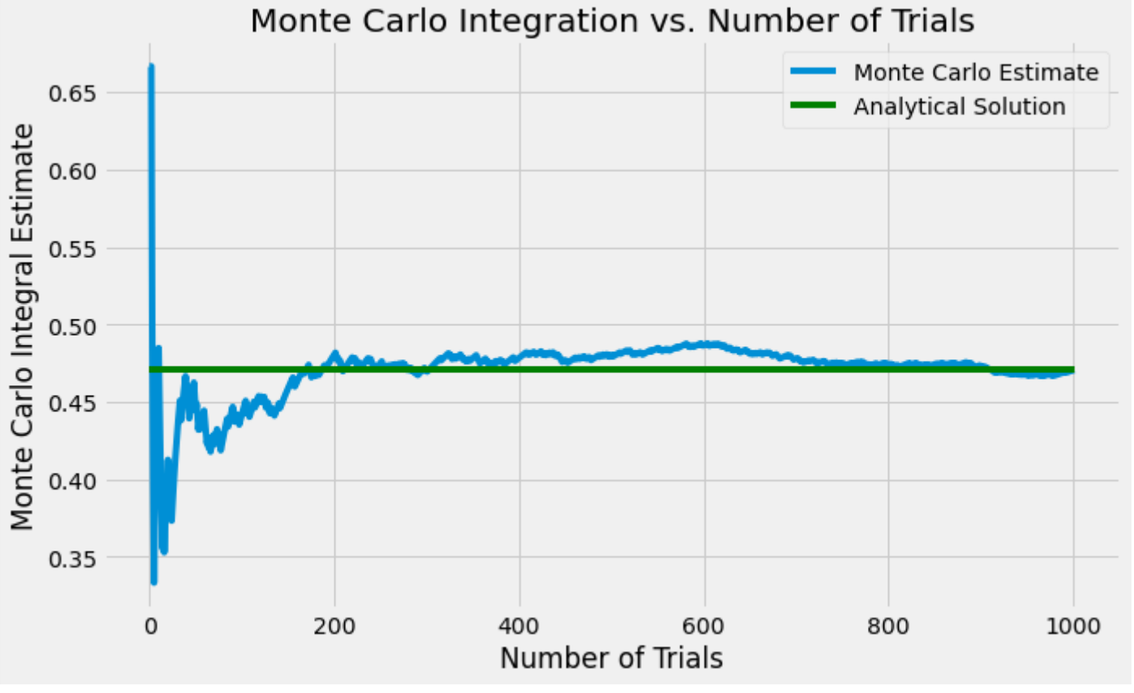
---

1. Define a function in order to call the function later on in order to generate a list of how the monte carlo algorithm performs in 1000 trials
    - Write for loop as before
2. For comparison on the graph, calaculate the analytical solution
    - print the answer
3. Call the function to obtain a list of Monte Carlo integral estimates for 1000 trials
4. Plot the numerical and analytical solution on the same graph

In [8]:
```python
import numpy as np # import numpy again for python kernel

# defining a function to call the monte carlo integration method to visualise how the method works
def monte_carlo(trial):

    # Initialize the summation variable
    summation = 0
    box_area = (xmax - xmin) * ymax # calculating the box area
    integral_list=[] # create an empty list to store the integral approximation after each trial (for visualization).

    # Perform the Monte Carlo integration, iterating along for each trial
```

```
    for trial_number in range(trial):
        x = np.random.uniform(xmin, xmax) # generate a random 'x' value within the given interval
        y = np.random.uniform(0, ymax) # generate a random 'y' value between 0 and 'ymax'
        if f(x) >= y: # check if the function value at x is greater than or equal to 'y'
            summation += 1 # increment the summation if the condition is met
        # calculate and store the Monte Carlo approximation for the integral after each trial
        integral_list.append(summation/(trial_number+1) * box_area)

    return integral_list #return the Monte Carlo approximation for the integral as a list

# Calculate the analytical solution for comparison and display it
answer = (2/3) - ((4*np.pi) - (3*np.sqrt(3)))/(12*np.pi)
print(f'Analytical solution: {answer}')

# Obtain a list of Monte Carlo integral estimates for 1000 trials
visualisation = monte_carlo(1000) # generate a list to visualize how the Monte Carlo integration function works

# Plot the performance
plt.figure(figsize=(10, 6))
plt.style.use('fivethirtyeight') # for background use style sheet from matplotlib
plt.plot(np.linspace(0, 1001, 1000), visualisation, '-', label='Monte Carlo Estimate') # Plot the Monte Carlo integral estimates over the number of trials
plt.plot(np.linspace(0,1001,1000),np.ones(1000)*answer, 'g-', label='Analytical Solution') # Plot the analytical solution as a reference.
plt.xlabel('Number of Trials')
plt.xlabel('Number of Trials')
plt.ylabel('Monte Carlo Integral Estimate')
plt.title('Monte Carlo Integration vs. Number of Trials')
# plt.legend(bbox_to_anchor = (1.05, 1), loc = 'upper left', borderaxespad=0) # adding legend to top right corner
plt.legend()
#plt.grid()
plt.show() # display plot
```

```
Analytical solution: 0.4711655571887813
```



It is seen that with adding more trials, the approximation tends to the 'true' value of the integral.

Task 5: Monte Carlo General Solution for Inifnite Potential Well

The purpose of this task is to generalise the monte carlo method into three dimensions to solve the following 3D infinite potential well problem.

The concept is to use a Monte Carlo simulation to generate random samples within a specified geometric region.

This Python code uses the Monte Carlo method to determine the probability of finding an electron in the ground state within a spherical region of a 3D infinite potential well

If the wavefunction for an electron trapped in a 3D infinite well of side length L is given by

$$\psi(x,y,z) = \sqrt{\frac{8}{L^3}} sin(\frac{n_1\pi}{L}x) sin(\frac{n_2\pi}{L}y) sin(\frac{n_3\pi}{L}z)$$

If the electron is in the ground state, determine the probability of finding it within a spherical region of radius 0.25L, centred in the middle of the cube.

1. Define a function with parameters x,y,z,n1,n2,n3 that returns a value for the wavefunctoin equation
2. Determine the probability of finding an electron in the ground state within a spherical region of radius 0.25L, centered in the middle of the cube
    - Define a function to compute the monte carlo probalilty of finding an electron in the ground state
    - Use the same method of the for loop as before, adding the radius equation and the x,y,z co-ords
    - use an if statement to calculate if the electron is inside the sphere, if so increment the summation
3. Assign values to the parameters for an example use of the method
4. Call the function to determine the probability of this example within the spherical region
5. Print this probability

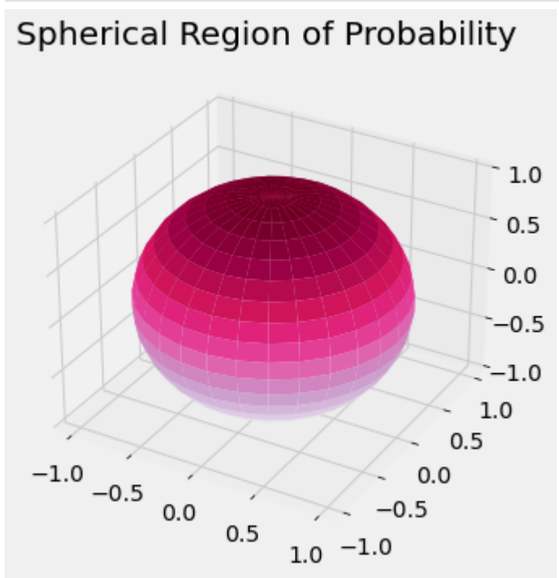
In [9]:
```
# task 5
# general Monte Carlo simulation where you want to generate random samples within a specified geometric region.
# this Python code uses the Monte Carlo method to determine the probability of finding an electron in the ground state within a spherical region of a 3D infinite potential well

# it calculates the wavefunction for an electron trapped in a 3D infinite well
# returns a float of the value of the wavefunction at the given coordinates
def wavefunction(x, y, z, L, n1, n2, n3):
    return np.sqrt(8 / L**3) * np.sin((n1 * np.pi / L) * x) * np.sin((n2 *np.pi / L) * y) * np.sin((n3 * np.pi / L) * z) # equation given

# Use the Monte Carlo method to determine the probability of finding the electron in the ground state within a spherical region of radius 0.25L, centered in the middle of the cube.
# returns a float of the probability of finding the electron within the specified spherical region
# L is the side length of the cube. The cube is centered at the origin, and its sides extend from -L/2 to L/2 along each axis

# coding monte carlo method as a probability sphere
# large number of samples used for better accuracy
def monte_carlo_probability(L, n1, n2, n3, num_samples=1000000):
   # same method as before with different variable names
    count_inside = 0
    radius = 0.25 * L
    for _ in range(num_samples):
        # minimum value of -L/2 ensures that the generated coordinates are within the left half of the cube along each axis
        # maximum value of -L/2 ensures that the generated coordinates are within the right half of the cube along each axis
        x = np.random.uniform(-L/2, L/2)
        y = np.random.uniform(-L/2, L/2)
        z = np.random.uniform(-L/2, L/2)
        if x**2 + y**2 + z**2 <= radius**2:
            count_inside += 1 # increment the summation if the condition is met
    # num_samples (int) is the number of random samples to use for the Monte Carlo method
    return count_inside / num_samples

# Example use of method
L = 1.0  # side length of the well
n1 = 1   # quantum number for x-direction
n2 = 1   # quantum number for y-direction
n3 = 1   # quantum number for z-direction

probability = monte_carlo_probability(L, n1, n2, n3) # calculating probability of the variables
print(f"The probability of finding the electron within the spherical region is: {probability:.4f}")
```

```
The probability of finding the electron within the spherical region is: 0.0655
```

Graphing this spherical region of probability

Below I have coded a graph to visualise the sphere of probability

In [10]:
```
# example of spherical region of probability - visiualised with radius = 0.25 * L
# Set the default figure size and enable automatic layout for better visualization
plt.rcParams["figure.figsize"] = [10.00, 5]
plt.rcParams["figure.autolayout"] = True

# create 3d plot
fig = plt.figure()
ax = fig.add_subplot(projection='3d')

# set radius size
r = 0.25 * L

# create a grid of points in spherical coordinates (u, v)
u, v = np.mgrid[0:2 * np.pi:30j, 0:np.pi:20j]
# x,y,z in spherical co-ords
x = np.cos(u) * np.sin(v)
y = np.sin(u) * np.sin(v)
z = np.cos(v)

# plot the surface of the sphere using the Cartesian coordinates
ax.plot_surface(x, y, z, cmap=plt.cm.PuRd)
plt.title('Spherical Region of Probability')
plt.show()
```



It should be noted this takes the shape of an 'ellipsoid'