Assignment 2

AA 651

Python Assignment 2

October 4, 2021

Professor: Dr. Suman Majumdar
Discipline of Astronomy, Astrophysics and Space Engineering (DAASE)

Student's name: Keshav Aggarwal
Roll No.: 2103121014

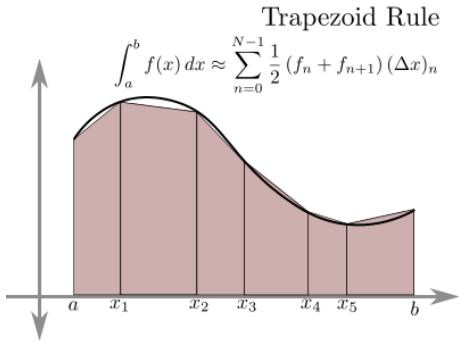
Index

1 Introduction

The questions of this assignment while looked simpler than the previous one, they involved a much more elaborate thought process. This assignment employed the concept of integration, primarily the Trapezoidal and Simpson's 1/3rd rule. Another benefit was to judge the better integration method. When I used the two methods, I was quite surprised to find the errors that each method makes while integrating.

1.1 Trapezoid Rule:

The trapezoid rule uses trapezoids instead of rectangles to approximate the area above each subinterval: The area of a trapezoid with bases (parallel segments) of length p and q, with



height h has area $\frac{1}{2}h(p+q)$. The area of the nth trapezoid, then, is $\frac{1}{2}(\Delta x)_n(f_n + f_{n+1})$.

Thus, the trapezoid rule gives the approximation:

$$\int_a^b f(x)dx \approx \sum_{n=0}^{N-1} \frac{1}{2}(\Delta x)_n(f_n + f_{n+1}) \approx \sum_{n=0}^{N-1} \frac{1}{2}(f_n + f_{n+1})(x_{n+1} - x_n).$$

If the sample points are evenly spaced uniformly, then the formula for the trapezoid rule simply becomes

$$\int_a^b f(x)dx \approx h\left(\frac{1}{2}(f_0 + f_N) + \sum_{n=1}^{N-1} f_n\right),$$

where $(h = \frac{b-a}{N})$ is the width of each trapezoid.

1.2 Simpson's Rule:

Simpson's rule uses piece-wise quadratic approximations.Another way to think of it is that Simpson's rule will compute the area under a parabola exactly.

One way that Simpson's rule differs from the above rule is that the sample points must be evenly spaced. Let $h = \frac{b-a}{N}$ be the distance between sample points. The Simpson's rule approximation is given by

$$\int_a^b f(x)dx \approx \frac{h}{3}\left(f_0 + 4f_1 + 2f_2 + 4f_3 + \ldots + 2f_{N-2} + 4f_{N-1} + f_N\right).$$

Here, N must be even.

1.3 Gaussian Quadrature :

In the quadrature formula

$$\int_a^b f(x)dx \approx \omega_0\, f(x_0) + \omega_1\, f(x_1) + \cdots + \omega_n\, f(x_n)$$

If we have n nodes in [a,b]=[1,1] the weights and the nodes are

n	ω_i	x_i
1	2.000000	0.000000
2	1.000000	$\pm$0.577350
3	0.555556	$\pm$0.774597
	0.888889	0.000000
4	0.347855	$\pm$0.861136
	0.652145	$\pm$0.339981
5	0.236927	$\pm$0.906180
	0.478629	$\pm$0.538469
	0.568889	0.000000

For instance, the Gaussian formula with one node

$$\int_{-1}^{1} f(x)dx \approx 2\ f(0)$$

that it is the same as the Middle Point rule.

For two nodes

$$\int_{-1}^{1} f(x)\ dx \approx f(-0.577350) + f(0.577350)$$

For three nodes

$$\int_{-1}^{1} f(x)\ dx \approx 0.555556\ f(-0.774597) + 0.8888889\ f(0) + 0.555556\ f(0.774597)$$

And so on.

This formulas can be generalized for any interval [a,b] changing x_i with y_i following the

$$y_i = \frac{b-a}{2} x_i + \frac{a+b}{2}$$

And then, the quadrature formula is

$$\int_a^b f(x)\, dx \approx \frac{b-a}{2}\left(\omega_0\ f(y_0) + \omega_1\ f(y_1) + \cdots + \omega_n\ f(y_n)\right)$$

2 Problems

2.1 Problem 1

Describe the motion of a planet for each of the following sets of initial condition. The sun is assumed to be at the origin and its motion is not considered. The only force involved being the force of gravitation. Assuming $GM_{\odot} = 1$, where G is gravitation constant and M is the mass of sun, one can avoid dealing with astronomical magnitudes for positions:

- $x_0 = 0.5; y_0 = 0.0; v_{0,x} = 0.0; v_{0,y} = 1.63.$
- $x_0 = 0.5; y_0 = 0.0; v_{0,x} = 0.0; v_{0,y} = 1.80.$
- $x_0 = 0.5; y_0 = 0.0; v_{0,x} = 0.0; v_{0,y} = 1.40.$
- $x_0 = 0.5; y_0 = 0.0; v_{0,x} = 0.50; v_{0,y} = 1.80.$
- $x_0 = 0.5; y_0 = 0.0; v_{0,x} = -0.50; v_{0,y} = 1.40.$
- $x_0 = 0.5; y_0 = 0.5; v_{0,x} = -0.50; v_{0,y} = 0.50.$

Answer-

From the set of positions given as well as from the question it was quite clear that I needed to plot the 3d trajectory of the set of equations given. After searching around a bit online, I found that pykep library was the best option to use for this problem. After experimenting with other orbit plots like kepler, etc, I found this to be the one that worked the best.

pykep is a scientific library developed at the European Space Agency to provide basic tools for astrodynamics research. Algorithmic efficiency is a main focus of the library, written in C++ and exposed to Python. At the library core is the implementation of an efficient solver for the multiple revolutions Lambert's problem, objects representing direct (Sims-Flanagan), indirect (Pontryagin) and hybrid methods to represent low-thrust optimization problems, efficient keplerian propagators, Taylor-integrators, a SGP4 propagator, TLE and SATCAT support, JPL SPICE code interface and more.

In the first listing I have plotted the graph for each of the set of positions separately.

In the second listing, I have combined all the curves on a single plot.

In the third listing, I gave each set of positions a unique color of their own to distinguish between them better.

At one point, I was trying to get to input the set of positions from a csv file and then plot the graphs but couldn't manage to do so. I did however manage to get the csv file to be displayed in the format of a table. I had also tried to do subplots but couldn't manage those too.

I have also attached a basic listing of the code below -

```
#pykep.orbit_plots.plot_taylor(*args)
ax = plot_taylor(r0, v0, m0, thrust, tof, mu, veff, N=60, units=1, color=
      b   , legend=False, axes=None):
```

axes: 3D axis object created using fig.gca(projection='3d')
r0: initial position (cartesian coordinates)
v0: initial velocity (cartesian coordinates)
m0: initial mass
u: cartesian components for the constant thrust
tof: propagation time

mu: gravitational parameter
veff: the product Isp*g0
N: number of points to be plotted along one arc
units: the length unit to be used in the plot
color: matplotlib color to use to plot the line
legend: when True it plots also the legend
Plots the result of a taylor propagation of constant thrust

```
!pip install pykep
```

```
import pykep as pk
import matplotlib.pyplot as plt
pi = 3.14

#plot for x0=0.5;y0=0.0;vx=0.0;vy=1.63
fig = plt.figure()
ax = fig.gca(projection = '3d')
pk.orbit_plots.plot_taylor([0.5,0,0],[0,1.63,0],100,[1,1,0],4, 1, 1, N =
    1000, axes = ax)
plt.title(r'$x_{0}=0.5 ;y_{0}=0.0 ;v_{0,x}=0.0 ;v_{0,y}=1.63 . $')
plt.show()

#plot for x0=0.5;y0=0.0;vx=0.0;vy=1.80
fig = plt.figure()
ax = fig.gca(projection = '3d')
pk.orbit_plots.plot_taylor([0.5,0,0],[0,1.8,0],100,[1,1,0],10, 1, 1, N =
    1000, axes = ax)
plt.title("$x_{0}=0.5 ;y_{0}=0.0 ;v_{0,x}=0.0 ;v_{0,y}=1.80 . $")
plt.show()

#plot for x0=0.5;y0=0.0;vx=0.0;vy=1.40
fig = plt.figure()
ax = fig.gca(projection = '3d')
pk.orbit_plots.plot_taylor([0.5,0,0],[0,1.4,0],100,[1,1,0],4, 1, 1, N = 1000,
     axes = ax)
plt.title("$x_{0}=0.5 ;y_{0}=0.0 ;v_{0,x}=0.0 ;v_{0,y}=1.40 . $")
plt.show()

#plot for x0=0.5;y0=0.0;vx=0.50;vy=1.80
fig = plt.figure()
ax = fig.gca(projection = '3d')
pk.orbit_plots.plot_taylor([0.5,0,0],[0.5,1.8,0],100,[1,1,0],17, 1, 1, N =
    1000, axes = ax)
plt.title("$x_{0}=0.5 ;y_{0}=0.0 ;v_{0,x}=0.50 ;v_{0,y}=1.80 . $")
plt.show()

#plot for x0=0.5;y0=0.0;vx= 0 .50;vy=1.40
fig = plt.figure()
ax = fig.gca(projection = '3d')
pk.orbit_plots.plot_taylor([0.5,0,0],[-0.5,1.4,0],100,[1,1,0],4, 1, 1, N =
    1000, axes = ax)
plt.title("$x_{0}=0.5 ;y_{0}=0.0 ;v_{0,x}=-0.50 ;v_{0,y}=1.40 . $")
plt.show()

#plot for x0=0.5;y0=0.5;vx= 0 .50;vy=0.50
fig = plt.figure()
ax = fig.gca(projection = '3d')
pk.orbit_plots.plot_taylor([0.5,0.5,0],[-0.5,0.5,0],100,[1,1,0],4, 1, 1, N =
    1000, axes = ax)
plt.title("$x_{0}=0.5 ;y_{0}=0.5 ;v_{0,x}=-0.50 ;v_{0,y}=0.50 . $")
plt.show()
```

Listing 1: Separate curve for each set of given values

Separate curve for each set of given values-

```
import pykep as pk
import matplotlib.pyplot as plt
pi = 3.14

#plot for x0=0.5;y0=0.0;vx=0.0;vy=1.63
fig = plt.figure()
ax = fig.gca(projection = '3d')
pk.orbit_plots.plot_taylor([0.5,0,0],[0,1.63,0],100,[1,1,0],4, 1, 1, N = 1000, axes = ax)
plt.title(r'$x_{0}=0.5 ;y_{0}=0.0 ;v_{0,x}=0.0 ;v_{0,y}=1.63 . $')
plt.show()

#plot for x0=0.5;y0=0.0;vx=0.0;vy=1.80
fig = plt.figure()
ax = fig.gca(projection = '3d')
pk.orbit_plots.plot_taylor([0.5,0,0],[0,1.8,0],100,[1,1,0],10, 1, 1, N = 1000, axes = ax)
plt.title("$x_{0}=0.5 ;y_{0}=0.0 ;v_{0,x}=0.0 ;v_{0,y}=1.80 . $")
plt.show()

#plot for x0=0.5;y0=0.0;vx=0.0;vy=1.40
fig = plt.figure()
ax = fig.gca(projection = '3d')
pk.orbit_plots.plot_taylor([0.5,0,0],[0,1.4,0],100,[1,1,0],4, 1, 1, N = 1000, axes = ax)
plt.title("$x_{0}=0.5 ;y_{0}=0.0 ;v_{0,x}=0.0 ;v_{0,y}=1.40 . $")
plt.show()

#plot for x0=0.5;y0=0.0;vx=0.50;vy=1.80
fig = plt.figure()
ax = fig.gca(projection = '3d')
pk.orbit_plots.plot_taylor([0.5,0,0],[0.5,1.8,0],100,[1,1,0],17, 1, 1, N = 1000, axes = ax)
plt.title("$x_{0}=0.5 ;y_{0}=0.0 ;v_{0,x}=0.50 ;v_{0,y}=1.80 . $")
plt.show()
```

```
#plot for x0=0.5;y0=0.0;vx=-0.50;vy=1.40
fig = plt.figure()
ax = fig.gca(projection = '3d')
pk.orbit_plots.plot_taylor([0.5,0,0],[-0.5,1.4,0],100,[1,1,0],4, 1, 1, N = 1000, axes = ax)
plt.title("$x_{0}=0.5 ;y_{0}=0.0 ;v_{0,x}=-0.50 ;v_{0,y}=1.40 . $")
plt.show()

#plot for x0=0.5;y0=0.5;vx=-0.50;vy=0.50
fig = plt.figure()
ax = fig.gca(projection = '3d')
pk.orbit_plots.plot_taylor([0.5,0.5,0],[-0.5,0.5,0],100,[1,1,0],4, 1, 1, N = 1000, axes = ax)
plt.title("$x_{0}=0.5 ;y_{0}=0.5 ;v_{0,x}=-0.50 ;v_{0,y}=0.50 . $")
plt.show()
```

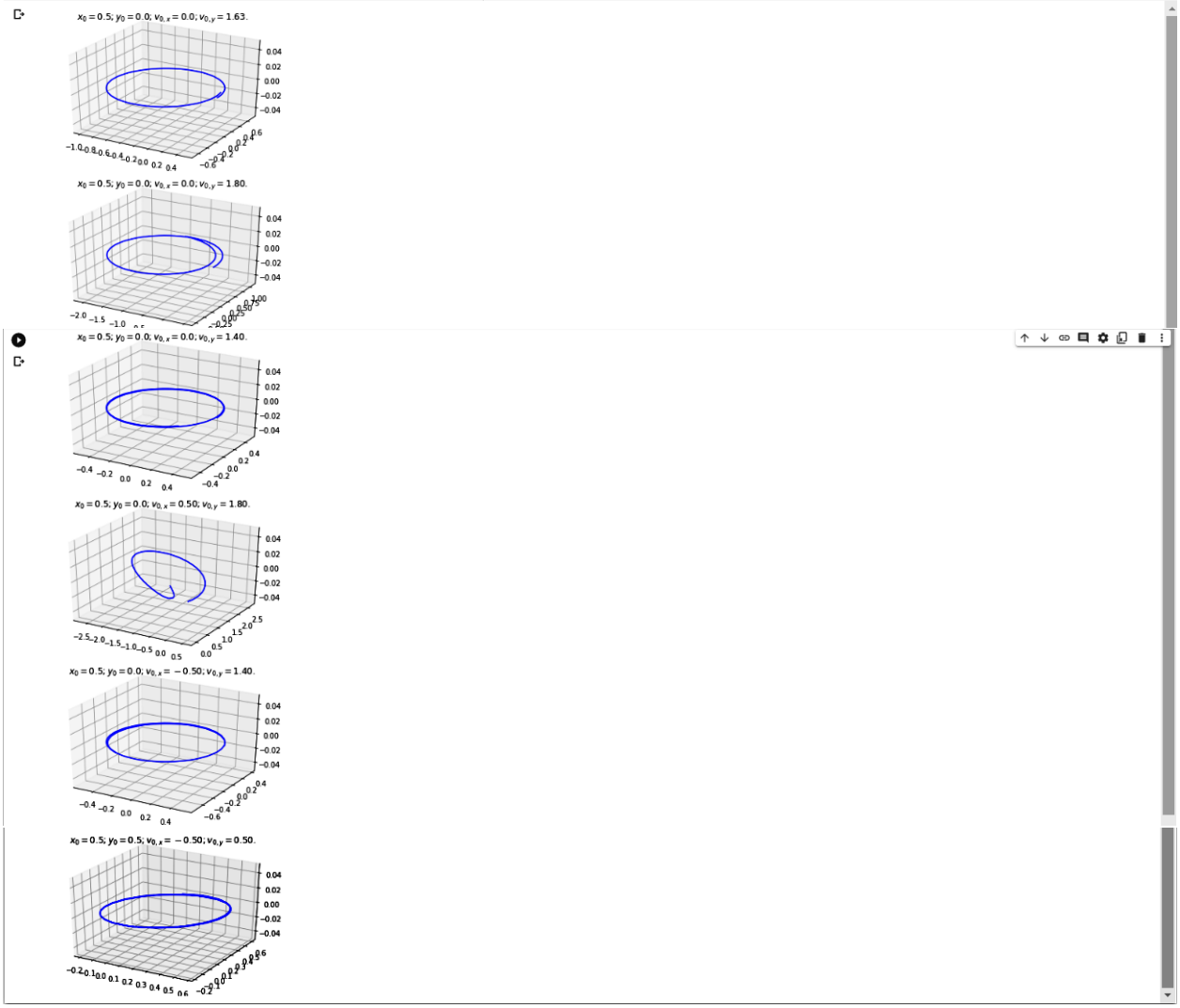


Figure 1: Separate curve for each set of given values

```
import pykep as pk
import matplotlib.pyplot as plt
pi = 3.14

#plot for x0=0.5;y0=0.0;vx=0.0;vy=1.63
fig = plt.figure()
ax = fig.gca(projection = '3d')
pk.orbit_plots.plot_taylor([0.5,0,0],[0,1.63,0],100,[1,1,0],4, 1, 1, N =
    1000, axes = ax)
pk.orbit_plots.plot_taylor([0.5,0,0],[0,1.8,0],100,[1,1,0],10, 1, 1, N =
    1000, axes = ax)
pk.orbit_plots.plot_taylor([0.5,0,0],[0,1.4,0],100,[1,1,0],4, 1, 1, N = 1000,
     axes = ax)
pk.orbit_plots.plot_taylor([0.5,0,0],[0.5,1.8,0],100,[1,1,0],17, 1, 1, N =
    1000, axes = ax)
pk.orbit_plots.plot_taylor([0.5,0,0],[-0.5,1.4,0],100,[1,1,0],4, 1, 1, N =
    1000, axes = ax)
pk.orbit_plots.plot_taylor([0.5,0.5,0],[-0.5,0.5,0],100,[1,1,0],4, 1, 1, N =
    1000, axes = ax)
plt.show()
```

Listing 2: Plot showing the combination of all curves

Plot showing the combination of all curves -

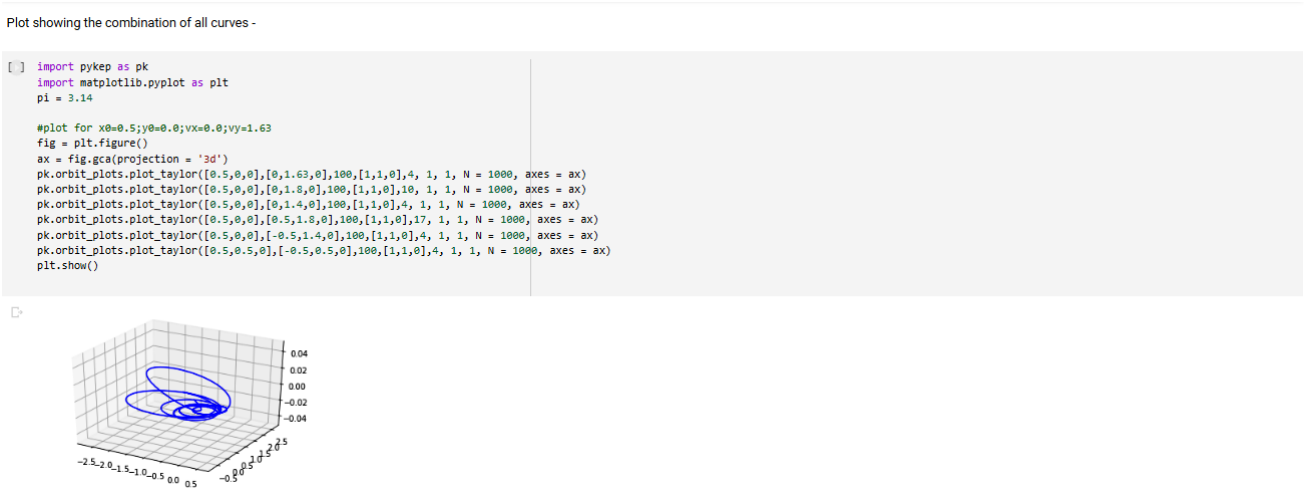


Figure 2: Plot showing the combination of all curves

```
import pykep as pk
import matplotlib.pyplot as plt
pi = 3.14

fig = plt.figure()
ax = fig.gca(projection = '3d')
#plot for x0=0.5;y0=0.0;vx=0.0;vy=1.63
pk.orbit_plots.plot_taylor([0.5,0,0],[0,1.63,0],100,[1,1,0],4, 1, 1, N =
    1000,color='r', axes = ax)
#plot for x0=0.5;y0=0.0;vx=0.0;vy=1.80
pk.orbit_plots.plot_taylor([0.5,0,0],[0,1.8,0],100,[1,1,0],10, 1, 1, N =
    1000,color='g', axes = ax)
#plot for x0=0.5;y0=0.0;vx=0.0;vy=1.40
pk.orbit_plots.plot_taylor([0.5,0,0],[0,1.4,0],100,[1,1,0],4, 1, 1, N = 1000,
    color='b', axes = ax)
#plot for x0=0.5;y0=0.0;vx=0.50;vy=1.80
pk.orbit_plots.plot_taylor([0.5,0,0],[0.5,1.8,0],100,[1,1,0],17, 1, 1, N =
    1000,color='yellow', axes = ax)
#plot for x0=0.5;y0=0.0;vx= 0  .50;vy=1.40
pk.orbit_plots.plot_taylor([0.5,0,0],[-0.5,1.4,0],100,[1,1,0],4, 1, 1, N =
    1000,color='black', axes = ax)
#plot for x0=0.5;y0=0.5;vx= 0  .50;vy=0.50
pk.orbit_plots.plot_taylor([0.5,0.5,0],[-0.5,0.5,0],100,[1,1,0],4, 1, 1, N =
    1000,color='pink', axes = ax)
plt.show()
```

Listing 3: Plot showing all 6 curves in the combined plot but in different colors

Figure 3: Plot showing all 6 curves in the combined plot but in different colors

```
from google.colab import files
uploaded = files.upload()    # upload the p.csv file when prompted
```

```
file_name = "p.csv"
uploaded[file_name].decode("utf-8")
```

```
import pandas as pd
import io
io.StringIO(uploaded['p.csv'].decode('utf-8'))
```

```
pd.read_csv(io.StringIO(uploaded['p.csv'].decode('utf-8')))
```

Listing 4: csv file in the form of a table

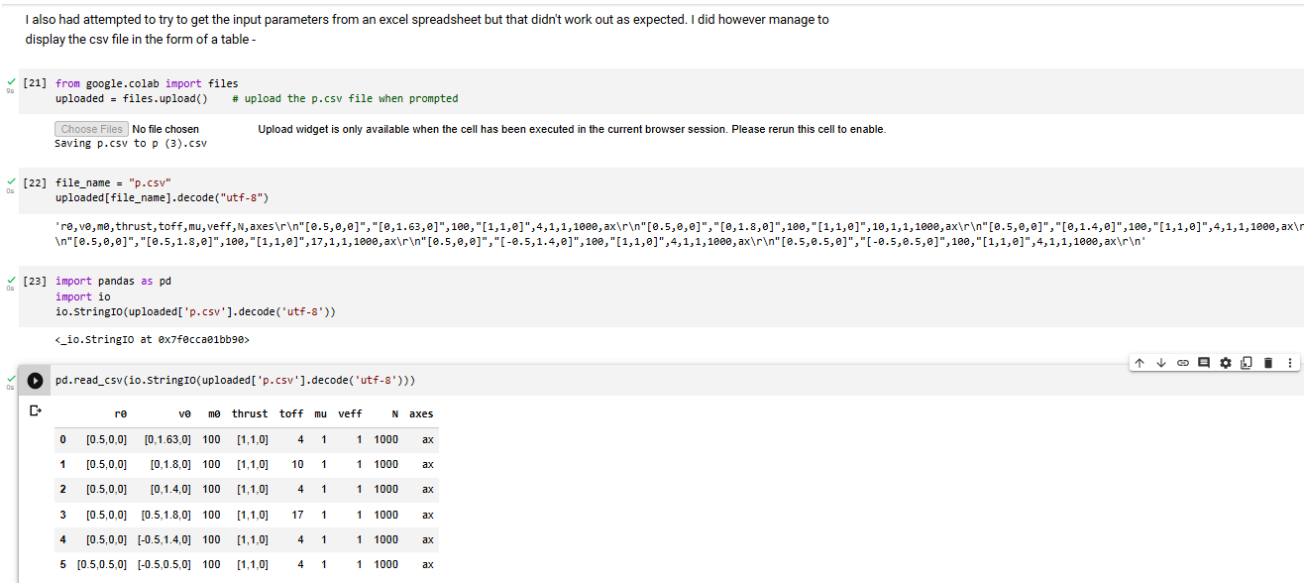


Figure 4: csv file in the form of a table

2.2 Problem 2

Write two functions which will perform integration following Trapezoidal and Simpson's rules respectively. Write these functions in such a way that they can take in any function as their integral.

Hints: If you are familiar with the concepts of pointer and function pointer, that will help you in writing these functions. Once you have managed to do that, evaluate the following integrals numerically:

- $\int_0^1 x\,dx$
- $\int_0^1 x^2\,dx$
- $\int_0^1 sin(x)\,dx$
- $\int_0^1 xsin(x)\,dx$

using both Trapezoidal and Simpson's rules for N = 2, 4, 8, ...1024, where N is the number of intervals used in the integration. As all of these integrals can be evaluated analytically, estimate the relative (with respect to the analytical value) error Err(N) in the numerical integration as a function of N. Show plots of Err(N) with varying N for all the integrals. Are these results consistent with how you expect the error to scale with N?

Answer-

This problem actually used three concepts, first, the concept of analytical integration, second, the concept of numerical integration, and third, plotting of results obtained.

In this problem, I have tried to generalise the code as much as I could. The code for the numerical integration is written in such a way that we can specify the values of the upper and the lower limits when we run the code. I had also attempted two more things but couldn't manage to do so. One being the option to allow to give the equation to be integrated when running the code (for now I have made use of commenting and un-commenting for the flexibility to allow to add more equations to be integrated as needed.) and the other being that the function uses the analytical integral directly when calculating the error of each numerical method. I couldn't figure out what I was doing wrong and had to drop these ideas this time.

For the problem I have attached the I/O screenshots as well as the listing code in three parts. The first part simply calculates the value of analytical integral of the function, the second calculates the integral using the Simpson's 1/3 rule and also calculates the error and plots it and the third does the came for Trapezoidal rule.

We can see that for higher values of intervals, the error becomes close to 0 hence higher the number of intervals, more accurate is the integration.

And yes, the results are consistent with how I had expected the error to scale with N.

```python
import numpy as np
import math
from scipy import *
from scipy.integrate import quad
#Choosing the lower and the upper limits of integration
# Input section
a = float(input("Enter lower limit of integration: "))
b = float(input("Enter upper limit of integration: "))

# function to be integrated can be simply added here and commented/un-commented as needed
def integrand1(x):
    output = x
    #output = x**2
    #output = np.sin(x)
    #output = x*np.sin(x)

    return output
solution1 = quad(integrand1,a,b)
# I is the value of the analytical integral chosen above.
#this value has to be substituted in the further two cells at the place of 0.5
I = solution1[0]

#Gives the analytical integral as output
print("Value from analytical method of integration:", I)
```

Listing 5: Analytical integration

Figure 5: Analytical integration

```
import numpy as np
import matplotlib.pyplot as plt

# function to be integrated can be simply added here and commented/un-
    commented as needed
def f(x):
    return x;
    #return x**2;
    #return np.sin(x);
    #return x*np.sin(x);

#Choosing the lower and the upper limits of integration
# Input section
a = float(input("Enter lower limit of integration: "))
b = float(input("Enter upper limit of integration: "))

#defining the Simpson's integration rule
def simpson(x0,xn,n):
    h = (xn - x0) / n
    integration = f(x0) + f(xn)
    for i in range(1,n):
        k = x0 + i*h
        if i%2 == 0:
            integration = integration + 2 * f(k)
        else:
            integration = integration + 4 * f(k)
    integration = integration * h/3

    return integration;
error_1=[]
for r in range(1,11):
  n=2**r

  #Here 0.5 needs to be replaced by the value of the integral,I obtained for
    the function in the previous step
  error_1.append(0.5- simpson(a,b,n))
  print("Integration result by Simpson's method for",n,"intervals is:",
    simpson(a,b,n),"with an error value of:", 0.5- simpson(a,b,n))

x = range(1,11)

plt.plot(x, error_1, color='blue');
plt.xlabel("Log N")
plt.ylabel("Error")
```

Listing 6: Simpson's 1/3 Rule

```python
import numpy as np
import matplotlib.pyplot as plt

# function to be integrated can be simply added here and commented/un-commented as needed
def f(x):
    return x;
    #return x**2;
    #return np.sin(x);
    #return x*np.sin(x);

#Choosing the lower and the upper limits of integration
# Input section
a = float(input("Enter lower limit of integration: "))
b = float(input("Enter upper limit of integration: "))

#defining the Simpson's integration rule
def simpson(x0,xn,n):
    h = (xn - x0) / n
    integration = f(x0) + f(xn)
    for i in range(1,n):
        k = x0 + i*h
        if i%2 == 0:
            integration = integration + 2 * f(k)
        else:
            integration = integration + 4 * f(k)
    integration = integration * h/3

    return integration;
error_1=[]
for r in range(1,11):
  n=2**r

  #Here 0.5 needs to be replaced by the value of the integral,I obtained for the function in the previous step
  error_1.append(0.5- simpson(a,b,n))
  print("Integration result by Simpson's method for",n,"intervals is:",simpson(a,b,n),"with an error value of:", 0.5- simpson(a,b,n))

x = range(1,11)

plt.plot(x, error_1, color='blue');
plt.xlabel("Log N")
plt.ylabel("Error")
```

```
Enter lower limit of integration: 0
Enter upper limit of integration: 1
Integration result by Simpson's method for 2 intervals is: 0.5 with an error value of: 0.0
Integration result by Simpson's method for 4 intervals is: 0.5 with an error value of: 0.0
Integration result by Simpson's method for 8 intervals is: 0.5 with an error value of: 0.0
Integration result by Simpson's method for 16 intervals is: 0.5 with an error value of: 0.0
Integration result by Simpson's method for 32 intervals is: 0.5 with an error value of: 0.0
Integration result by Simpson's method for 64 intervals is: 0.5 with an error value of: 0.0
Integration result by Simpson's method for 128 intervals is: 0.5 with an error value of: 0.0
Integration result by Simpson's method for 256 intervals is: 0.5 with an error value of: 0.0
Integration result by Simpson's method for 512 intervals is: 0.5 with an error value of: 0.0
Integration result by Simpson's method for 1024 intervals is: 0.5 with an error value of: 0.0
Text(0, 0.5, 'Error')
```

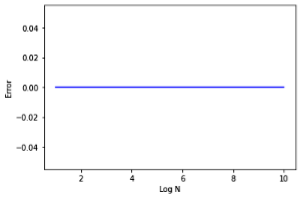


Figure 6: Simpson's 1/3 Rule

```
import numpy as np
import matplotlib.pyplot as plt

# function to be integrated can be simply added here and commented/un-
    commented as needed
def f(x):
    return x;
    #return x**2;
    #return np.sin(x);
    #return x*np.sin(x);
#Choosing the lower and the upper limits of integration
# Input section
a = float(input("Enter lower limit of integration: "))
b = float(input("Enter upper limit of integration: "))

def trapezoidal(x0,xn,n):
    h = (xn - x0) / n
    integration = f(x0) + f(xn)
    for i in range(1,n):
        k = x0 + i*h
        integration = integration + 2 * f(k)
    integration = integration * h/2

    return integration;

error_1=[]
for r in range(1,11):
  n=2**r
  #Here 0.5 needs to be replaced by the value of the integral,I obtained for
    the function in the previous step
  error_1.append(0.5-trapezoidal(a,b,n))
  print("Integration result by trapezoidal method for", n ," number of
    intervals is:" , trapezoidal(a,b,n), "with an error value of:", 0.5-
    trapezoidal(a,b,n) )

x = range(1,11)

plt.plot(x, error_1, color='blue');
plt.xlabel("Log N")
plt.ylabel("Error")
```

Listing 7: Trapezoidal rule

```python
import numpy as np
import matplotlib.pyplot as plt

# function to be integrated can be simply added here and commented/un-commented as needed
def f(x):
    return x;
    #return x**2;
    #return np.sin(x);
    #return x*np.sin(x);
#Choosing the lower and the upper limits of integration
# Input section
a = float(input("Enter lower limit of integration: "))
b = float(input("Enter upper limit of integration: "))

def trapezoidal(x0,xn,n):
    h = (xn - x0) / n
    integration = f(x0) + f(xn)
    for i in range(1,n):
        k = x0 + i*h
        integration = integration + 2 * f(k)
    integration = integration * h/2

    return integration;

error_1=[]
for r in range(1,11):
  n=2**r
  #Here 0.5 needs to be replaced by the value of the integral,I obtained for the function in the previous step
  error_1.append(0.5-trapezoidal(a,b,n))
  print("Integration result by trapezoidal method for", n ," number of intervals is:" , trapezoidal(a,b,n), "with an error value of:", 0.5-trapezoidal(a,b,n) )

x = range(1,11)

plt.plot(x, error_1, color='blue');
plt.xlabel("Log N")
plt.ylabel("Error")
```

```
Enter lower limit of integration: 0
Enter upper limit of integration: 1
Integration result by trapezoidal method for 2  number of intervals is: 0.5 with an error value of: 0.0
Integration result by trapezoidal method for 4  number of intervals is: 0.5 with an error value of: 0.0
Integration result by trapezoidal method for 8  number of intervals is: 0.5 with an error value of: 0.0
Integration result by trapezoidal method for 16  number of intervals is: 0.5 with an error value of: 0.0
Integration result by trapezoidal method for 32  number of intervals is: 0.5 with an error value of: 0.0
Integration result by trapezoidal method for 64  number of intervals is: 0.5 with an error value of: 0.0
Integration result by trapezoidal method for 128  number of intervals is: 0.5 with an error value of: 0.0
Integration result by trapezoidal method for 256  number of intervals is: 0.5 with an error value of: 0.0
Integration result by trapezoidal method for 512  number of intervals is: 0.5 with an error value of: 0.0
Integration result by trapezoidal method for 1024  number of intervals is: 0.5 with an error value of: 0.0
Text(0, 0.5, 'Error')
```

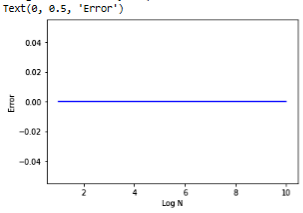


Figure 7: Trapezoidal rule

2.3 Problem 3

Consider the integral

$$\int_{-\infty}^{\infty} \frac{sin^2(x)}{x^2}\, dx$$

How will you evaluate it analytically? How will you evaluate it numerically? Compare your numerical and analytical results.

Answer-

While the problem itself was quite simply a mater of integration, the hard part were the limits. It was quite difficult to use such limits which python would comprehend as $-\infty$ and ∞. As I could not come to a conclusion on what limits to use for their replacement, I simply used these only.

As no specific method was specified for the numerical integration, I used Simpson's 1/3rd rule for this. The code differs from the previous question as I have taken the limits to be from zero to infinity and multiplied the function by two in the code at the final step. Then, I chose sufficiently large value for the upper limit so it could be treated as infinity. I have calculated the analytical integral first. Then the numerical integral using Simpson's 1/3rd rule and then calculated the error between the two.

```python
import numpy as np
import math
from scipy import *
from scipy.integrate import quad

#analytical method of integration


def integrand1(x):
    output = ((np.sin(x))**2)/x**2


    return output
solution1 = quad(integrand1,-np.inf,np.inf)
I = solution1[0]
print("Value from analytical method of integration:",I)
```

Listing 8: Analytical integral

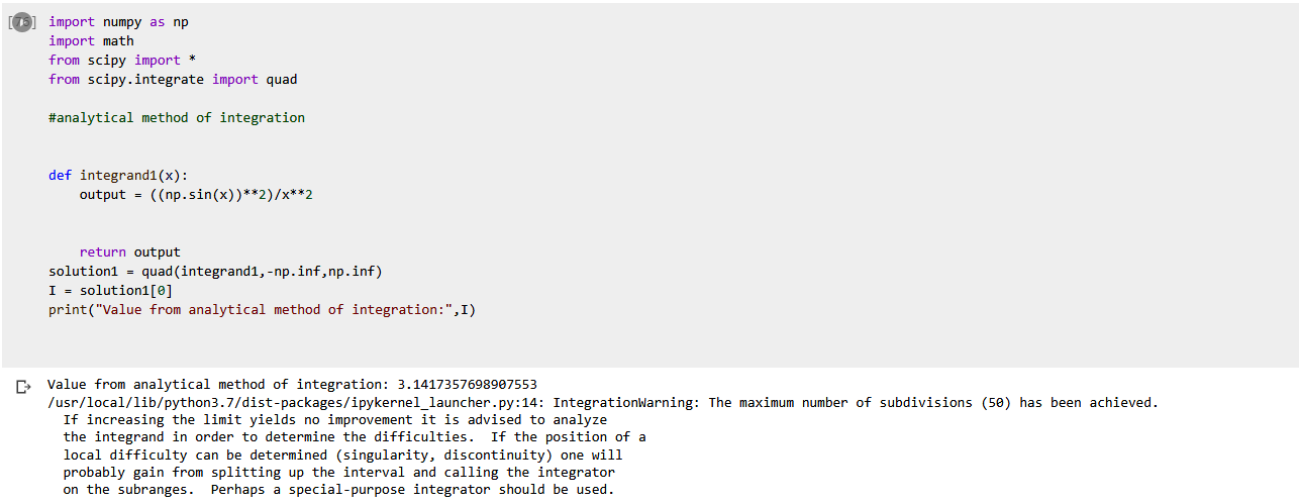


Figure 8: Analytical integral

```python
import numpy as np
import math


def f(x):
    return ((np.sin(x))**2)/x**2;

def simpson(x0,xn,n):
    h = (xn - x0) / n
    int = f(x0) + f(xn)
    for i in range(1,n):
        k = x0 + i*h
        if i%2 == 0:
            int = int + 2 * f(k)
        else:
            int = int + 4 * f(k)
    int = int * h/3
    print(2*int)
    return;

simpson(0.000001,10000000,10000000)
```

Listing 9: Numerical integral(Simpson's 1/3 rule)

```
[58] import numpy as np
     import math


     def f(x):
         return ((np.sin(x))**2)/x**2;

     def simpson(x0,xn,n):
         h = (xn - x0) / n
         int = f(x0) + f(xn)
         for i in range(1,n):
             k = x0 + i*h
             if i%2 == 0:
                 int = int + 2 * f(k)
             else:
                 int = int + 4 * f(k)
         int = int * h/3
         print(2*int)
         return;

     simpson(0.000001,10000000,10000000)

     3.14159053758913
```

Figure 9: Numerical integral(Simpson's 1/3 rule)

```python
#Value from analytical method of integration:
a= 3.1417357698907553

#value from Numerical integration using Simpson's Method:
b= 3.14159053758913
print(a-b)
```

Listing 10: Error calculation

```
[ ]  #Value from analytical method of integration:
     a= 3.1417357698907553

     #value from Numerical integration using Simpson's Method:
     b= 3.14159053758913
     print(a-b)

     0.00014523230162533096
```

This is the difference between the value of integral obtained from the two different methods.Similarly, the trapezoidal rule can also be used to calculate the numerical integral of the given function.

Figure 10: Error calculation

2.4 Problem 4

- Compute the following integral using 2, 3 and 5 point Gaussian quadrature method:

$$\int_{-1}^{1} x^2 e^{-x^2}\, dx$$

- The arc length along a curve f(x) is define as:

$$s = \int \sqrt{1 + f'(x)^2}$$

find the arc length from 0 to π along a curve f(x) = sin(x) analytically and also by using 2, 3 and 5-point Gaussian quadrature method and determine the % error with respect to the analytical value.

Answer-

This question also used a method of numerical integration but one I have never come across earlier. Before going through the coding I had to work out the maths involved in the method. Gauss quadrature deals with integration over a symmetrical range of x from -1 to + 1. The important property of Gauss quadrature is that it yields exact values of integrals for polynomials of degree up to 2n – 1.

For two nodes

$$\int_{-1}^{1} f(x)\, dx \approx f(-0.577350) + f(0.577350)$$

For three nodes

$$\int_{-1}^{1} f(x)\, dx \approx 0.555556\ f(-0.774597) + 0.8888889\ f(0) + 0.555556\ f(0.774597)$$

For five nodes

$$\int_{-1}^{1} f(x)\, dx \approx 0.236927\ f(-0.906180) + 0.236927\ f(0.906180) + 0.568889\ f(0.568889)$$

$$+0.478629\ f(0.538469) + 0.478629\ f(-0.538469)$$

Other than working out this mathematics, I was quite comfortable working with this question. To try out the input() function, I have written the maths for all versions of Gaussian Quadrature and used if-elif-else statements to keep it simple. The code first asks for input of the value of n and then just uses the Quadrature equation corresponding to it.

I felt the second part of the question was quite nice to be working on. As the function in the second question had different limits, I had to modify the code for the previous part in a way which allows the user to integrate the function using both analytical and Gaussian quadrature method using the same limits as given in the question and having not to go through the hassle of conversion of limits. For any interval [a,b], changing x_i with y_i following the equation

$$y_i = \frac{b-a}{2} x_i + \frac{a+b}{2} \tag{1}$$

Gave us the generalized form of the Gaussian Quadrature as -

$$\int_a^b f(x)\, dx \approx \frac{b-a}{2} \left(\omega_0\ f(y_0) + \omega_1\ f(y_1) + \cdots + \omega_n\ f(y_n)\right) \tag{2}$$

After this, it was again a simple code. I just use the code for the previous part and modified it according to (1) and (2) and obtained the integral.

```
import math
import numpy as np

#input the value of n for which you want to use the Gaussian Quadrature
n = int(input("Enter value of n for which you want to use the Gaussian
    Quadrature: "))

# function to be integrated can be just added here and commented/uncommented
    as per required
def f(x):
  return (x**2)*math.exp(-x**2)


# Gaussian Quadrature method to be used on the basis of the value of n input
    by the user
if n==1:
  Integ = 2*f(0)

elif n==2:
  Integ = f(-0.577350)+ f(0.577350)

elif n==3:
  Integ = 0.555556*f(-0.774597)+0.8888889*f(0) +0.555556*f(0.774597)

elif n==4:
  Integ = 0.347855*f(0.861136)+0.347855*f(-0.861136)+0.652145*f(0.339981)
    +0.652145*f(-0.339981)

else:
  Integ = 0.236927*f(-0.906180) + 0.236927*f(0.906180) +0.478629*f(0.538469)
    + 0.478629*f(-0.538469)+ 0.568889*f(0.568889)
# Output of the integral calculated using n point gaussian quadrature method
print("The value of the function using",n, "point Gaussian Quadrature method
    is", Integ)
```

Listing 11: Problem 4(a)

```
[20] import math
     import numpy as np

     #input the value of n for which you want to use the Gaussian Quadrature
     n = int(input("Enter value of n for which you want to use the Gaussian Quadrature: "))

     # function to be integrated can be just added here and commented/uncommented as per required
     def f(x):
       return (x**2)*math.exp(-x**2)


     # Gaussian Quadrature method to be used on the basis of the value of n input by the user
     if n==1:
       Integ = 2*f(0)

     elif n==2:
       Integ = f(-0.577350)+ f(0.577350)

     elif n==3:
       Integ = 0.555556*f(-0.774597)+0.8888889*f(0) +0.555556*f(0.774597)

     elif n==4:
       Integ = 0.347855*f(0.861136)+0.347855*f(-0.861136)+0.652145*f(0.339981)+0.652145*f(-0.339981)

     else:
       Integ = 0.236927*f(-0.906180) + 0.236927*f(0.906180) +0.478629*f(0.538469) + 0.478629*f(-0.538469)+ 0.568889*f(0.568889)
     # Output of the integral calculated using n point gaussian quadrature method
     print("The value of the function using",n, "point Gaussian Quadrature method is", Integ)

     Enter value of n for which you want to use the Gaussian Quadrature: 2
     The value of the function using 2 point Gaussian Quadrature method is 0.47768724341998986
```

Similarly, the values of other points can be calculated. For this question, the calculated values of the function for 2, 3 and 5 point Gaussian Quadrature are

2 point Gaussian Quadrature: 0.47768724341998986

3 point Gaussian Quadrature: 0.36587484174711

5 point Gaussian Quadrature: 0.5120811641591164

Figure 11: Problem 4(a)

```
import math
import numpy as np
from scipy.integrate import quad

#analytical method of integration
#limits of integration
a = 0      #lower limit of integration
b = np.pi  #upper limit of integration

def integrand1(x):
    output = math.sqrt(1+ (np.cos(x))**2)
    return output
solution1 = quad(integrand1,a,b)
I = solution1[0]
print("Value of arc length from analytical method of integration:",I)


##Gaussian Quadrature

#input the value of n for which you want to use the Gaussian Quadrature
n = int(input("Enter value of n for which you want to use the Gaussian
    Quadrature: "))

# function to be integrated can be just added here and commented/uncommented
    as per required
def f(x):
  return math.sqrt(1+ (np.cos(x))**2)

# Gaussian Quadrature method to be used on the basis of the value of n input
    by the user
if n==1:
  Integ = ((b-a)/2)*(2*f(((b-a)/2)*0 + ((a+b)/2)))

elif n==2:
  Integ = ((b-a)/2)*(f(-0.577350*((b-a)/2 + ((a+b)/2)))+ f(0.577350*((b-a)/2
    + ((a+b)/2))))

elif n==3:
  Integ = ((b-a)/2)*(0.555556*f(-0.774597*((b-a)/2) + ((a+b)/2))+0.8888889*f
    (0*((b-a)/2) + ((a+b)/2)) +0.555556*f(0.774597*((b-a)/2) + ((a+b)/2)))

elif n==4:
  Integ = ((b-a)/2)*(0.347855*f(0.861136*((b-a)/2) + ((a+b)/2))+0.347855*f
    (-0.861136*((b-a)/2) + ((a+b)/2))+0.652145*f(0.339981*((b-a)/2) + ((a+b)
    /2))+0.652145*f(-0.339981*((b-a)/2)+ ((a+b)/2)))

else:
  Integ = ((b-a)/2)*(0.236927*f(-0.906180*((b-a)/2) + ((a+b)/2)) + 0.236927*f
    (0.906180*((b-a)/2) + ((a+b)/2)) +0.478629*f(0.538469*((b-a)/2) + ((a+b)
    /2)) + 0.478629*f(-0.538469*((b-a)/2)+ ((a+b)/2))+ 0.568889*f(0.568889*((b
    -a)/2)+ ((a+b)/2)))
# Output of the integral calculated using n point gaussian quadrature method
print("The value of arc length using",n, "point Gaussian Quadrature method is
    ", Integ)
#calculation of error
error = ((Integ-I)/I)*100

print("The error using",n,"point Gaussian Quadrature method is",error,"%")
```

Listing 12: Problem 4(b)

```
[42] import math
     import numpy as np
     from scipy.integrate import quad

     #analytical method of integration
     #limits of integration
     a = 0      #lower limit of integration
     b = np.pi  #upper limit of integration

     def integrand1(x):
         output = math.sqrt(1+ (np.cos(x))**2)
         return output
     solution1 = quad(integrand1,a,b)
     I = solution1[0]
     print("Value of arc length from analytical method of integration:",I)


     ##Gaussian Quadrature

     #input the value of n for which you want to use the Gaussian Quadrature
     n = int(input("Enter value of n for which you want to use the Gaussian Quadrature: "))

     # function to be integrated can be just added here and commented/uncommented as per required
     def f(x):
       return math.sqrt(1+ (np.cos(x))**2)


     # Gaussian Quadrature method to be used on the basis of the value of n input by the user
     if n==1:
       Integ = ((b-a)/2)*(2*f(((b-a)/2)*0 + ((a+b)/2)))

     elif n==2:
       Integ = ((b-a)/2)*(f(-0.577350*((b-a)/2 + ((a+b)/2)))+ f(0.577350*((b-a)/2 + ((a+b)/2))))

     elif n==3:
       Integ = ((b-a)/2)*(0.555556*f(-0.774597*((b-a)/2) + ((a+b)/2))+0.8888889*f(0*((b-a)/2) + ((a+b)/2)) +0.555556*f(0.774597*((b-a)/2) + ((a+b)/2)))

     elif n==4:
       Integ = ((b-a)/2)*(0.347855*f(0.861136*((b-a)/2) + ((a+b)/2))+0.347855*f(-0.861136*((b-a)/2) + ((a+b)/2))+0.652145*f(0.339981*((b-a)/2) + ((a+b)/2))+0.652145*f(-0.339981*((b-a)/2)+ ((a+b)/2)))

     else:
       Integ = ((b-a)/2)*(0.236927*f(-0.906180*((b-a)/2) + ((a+b)/2)) + 0.236927*f(0.906180*((b-a)/2) + ((a+b)/2)) +0.478629*f(0.538469*((b-a)/2) + ((a+b)/2)) + 0.478629*f(-0.538469*((b-a)/2)+ ((a+b)/2))+ 0.568889*f(0.568889*((b-a)/2)+ ((a+b)/2)))
     # Output of the integral calculated using n point gaussian quadrature method
     print("The value of arc length using",n, "point Gaussian Quadrature method is", Integ)
```

```
#calculation of error
error = ((Integ-I)/I)*100

print("The error using",n,"point Gaussian Quadrature method is",error,"%")
```

```
Value of arc length from analytical method of integration: 3.8201977890277115
Enter value of n for which you want to use the Gaussian Quadrature: 3
The value of arc length using 3 point Gaussian Quadrature method is 3.789207568355212
The error using 3 point Gaussian Quadrature method is -0.8112203185266713 %
```

Value of arc length from analytical method of integration: 3.8201977890277115

For this question, the calculated values of arc length for 2, 3 and 5 point Gaussian Quadrature are

2 point Gaussian Quadrature: 3.2312572860403126

error: -15.416492430809233 %

3 point Gaussian Quadrature: 3.789207568355212

error: -0.8112203185266713 %

5 point Gaussian Quadrature: 4.058120459296245

error: 6.228019684013987 %

Figure 12: Problem 4(b)

2.5 Problem 5

We consider the 1D motion of a particle of mass m in a time independent potential V (x). The fact that the energy E will be conserved allows us to integrate the equation of motion and obtain a solution in a closed form:

$$t - C = \sqrt{\frac{m}{2}} \int_{x_i}^{x} \frac{dx'}{\sqrt{E - V(x')}} dx$$

The particle is at the position x_i at time t = 0 and it is located at the position x at any arbitrary time t. For simplicity, let us assume that C = 0.

Answer-

For me, this question was the trickiest for this assignment. I had trouble with two things. one, the type of motion the particle is moving with, and the second, what to do with the unknowns. For simplicity, I have taken E, and m as constants, C = 0 , and replaced V by it's equation for SHM and then assumed the value of k as well. To be fair, I still am not sure of how I solved this question and would like to be explained this question. For my part, I have evaluated the integral of the given equation both using analytical as well as numerical methods of integration. For numerical method, I have used Simpson's Rule.

As for the plotting, I have done a 2D plot from which we can see that even after changing the limits for from (-1,1) to (-0.5, 0.5) there is no change in amplitude, hence there is no dependency of amplitude on time period as it remains the same as π. For different values of amplitude, the time period also comes exactly the same as that in the case of a Simple Harmonic Motion.

Just as an exercise, I have also attempted to plot the same 2D plot in 3D as well. Although I am not sure of its accuracy, as I did try it and was not supposed to be done anyway I suppose, I still have tried out something new, just as I tried the input function and the table from a csv file. These problems were quite fun to work on.

```python
#For limits -1 to 1, with an aplitude of 1 unit.

from scipy.integrate import quad

def integrand(x):
    return (2**0.5)/(2-2*x**2)**(1/2)

ans, err = quad(integrand, -1,1)
print(ans)
```

```python
#For limits -0.5 to 0.5, with an aplitude of 0.5 unit.

from scipy.integrate import quad

def integrand(x):
    return (2**0.5)/(0.5-2*x**2)**(1/2)

ans, err = quad(integrand, -0.5,0.5)
print(ans)
```

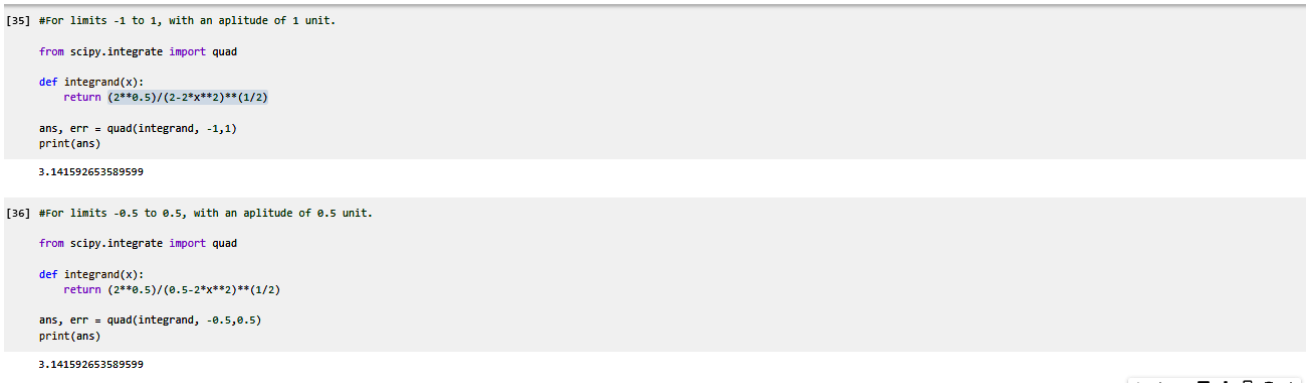
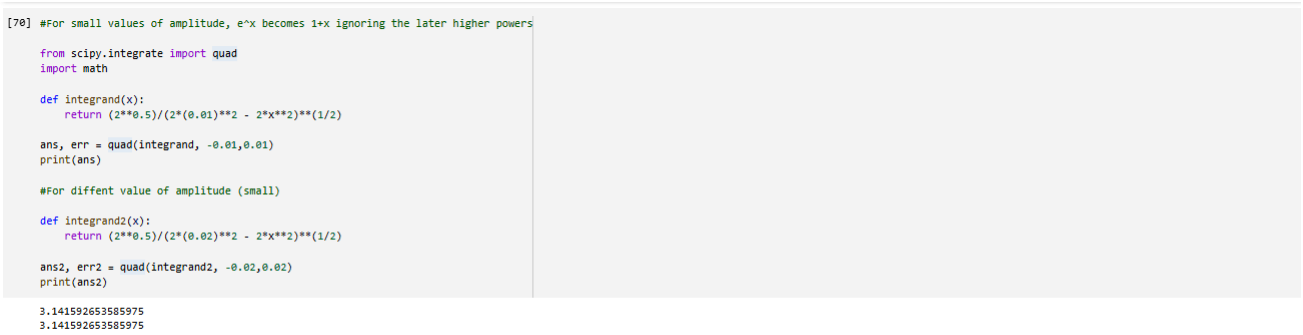


Figure 13: Problem 5

```python
#For small values of amplitude, e^x becomes 1+x ignoring the later higher
    powers

from scipy.integrate import quad
import math

def integrand(x):
    return (2**0.5)/(2*(0.01)**2 - 2*x**2)**(1/2)

ans, err = quad(integrand, -0.01,0.01)
print(ans)

#For diffent value of amplitude (small)

def integrand2(x):
    return (2**0.5)/(2*(0.02)**2 - 2*x**2)**(1/2)

ans2, err2 = quad(integrand2, -0.02,0.02)
print(ans2)
```



```
[70] #For small values of amplitude, e^x becomes 1+x ignoring the later higher powers

     from scipy.integrate import quad
     import math

     def integrand(x):
         return (2**0.5)/(2*(0.01)**2 - 2*x**2)**(1/2)

     ans, err = quad(integrand, -0.01,0.01)
     print(ans)

     #For diffent value of amplitude (small)

     def integrand2(x):
         return (2**0.5)/(2*(0.02)**2 - 2*x**2)**(1/2)

     ans2, err2 = quad(integrand2, -0.02,0.02)
     print(ans2)

     3.141592653589975
     3.141592653589975
```

We can see above that even after changing the limits for from (-1,1) to (-0.5, 0.5) there is no change in amplitude, hence there is no dependency of amplitude on time period as it remains the same as π. For diferent values of amplitude, the time period also comes exactly the same as that in the case of a Simple Harmonic Motion.

Figure 14: Problem 5

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.integrate import quad
import math

m = 1 #float(input("Mass of the particle :"))
E = 1 #float(input("energy of the particle :"))
k = 1 #float(input("Value of k :"))

def f(x,E):
    return math.sqrt((m/2)*(1/(E-1/2 * k * x**2)))

def simpson(x0,xn,n):
    h = (xn - x0) / n
    int = f(x0,E) + f(xn,E)
    for i in range(1,n):
        k = x0 + i*h
        if i%2 == 0:
            int = int + 2 * f(k,E)
        else:
            int = int + 4 * f(k,E)
    int = int * h/3
    print("simpson integ =" ,2*int)
    return;

simpson(-1,1,50)
```

Listing 13: Simpson's Rule

Figure 15: Simpson's Rule

```python
#For large values of amplitude
import numpy as np
from scipy.integrate import quad
import math
import matplotlib.pyplot as plt

def integrand(x):
    return (2**0.5)/(math.exp(2*(p/100)**2)-1 - (math.exp(2*x**2)-1))**(1/2)

array=[]
for p in range(1,101):
    ans, err = quad(integrand, -p/100,p/100)
    array.append(ans)

xpoints = np.linspace(0,1, 100)
ypoints = array
plt.title('Variation of Time Period w.r.t. Amplitude for Large Amplitudes')
plt.xlabel("Amplitude (meters)")
plt.ylabel("Time Period (seconds)")
plt.plot(xpoints, ypoints)
plt.show()
```

Listing 14: 2D plot

Figure 16: 2D Plot

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import quad
import math


# creating an empty canvas
fig = plt.figure()

# Creating an empty 3D axes of the plot
ax = fig.add_subplot(projection='3d')

# Labeling the X Axis
ax.set_xlabel('X Axis')

# Labeling the Y-Axis
ax.set_ylabel('Y Axis')

# Labeling the Z-Axis
ax.set_zlabel('Z Axis')



def integrand(x):
    return (2**0.5)/(math.exp(2*(p/100)**2)-1 - (math.exp(2*x**2)-1))**(1/2)

array=[]
for p in range(1,101):
    ans, err = quad(integrand, -p/100,p/100)
    array.append(ans)

xpoints = np.linspace(0,1, 100)
ypoints = array
zpoints = 0

plt.plot(xpoints,ypoints)
fig.set_size_inches(7,6)
# Showing the above 3D plot
plt.show()
```

Listing 15: 3D plot

The same plot as above, just in 3D -

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import quad
import math


# creating an empty canvas
fig = plt.figure()

# Creating an empty 3D axes of the plot
ax = fig.add_subplot(projection='3d')

# Labeling the X Axis
ax.set_xlabel('X Axis')

# Labeling the Y-Axis
ax.set_ylabel('Y Axis')

# Labeling the Z-Axis
ax.set_zlabel('Z Axis')



def integrand(x):
    return (2**0.5)/(math.exp(2*(p/100)**2)-1 - (math.exp(2*x**2)-1))**(1/2)

array=[]
for p in range(1,101):
    ans, err = quad(integrand, -p/100,p/100)
    array.append(ans)

xpoints = np.linspace(0,1, 100)
ypoints = array
zpoints = 0
```

```
plt.plot(xpoints,ypoints)
fig.set_size_inches(7,6)
# Showing the above 3D plot
plt.show()
```

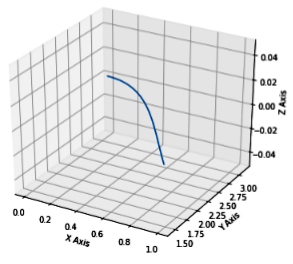


Figure 17: 3D plot

3 Bibliography

- https://www.pythoncheatsheet.org/
- https://www.tutorialspoint.com/index.htm
- https://www.stackoverflow.com/
- https://www.github.com/
- https://www.javatpoint.com/python-tutorial
- https://www.programiz.com/python-programming