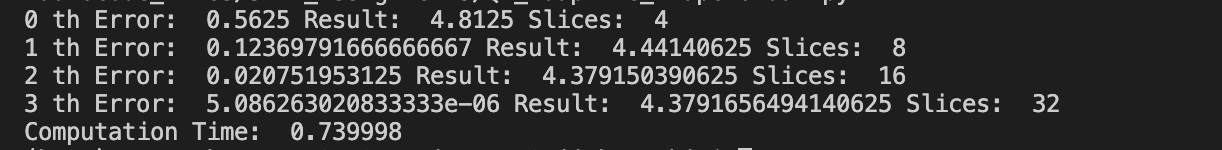
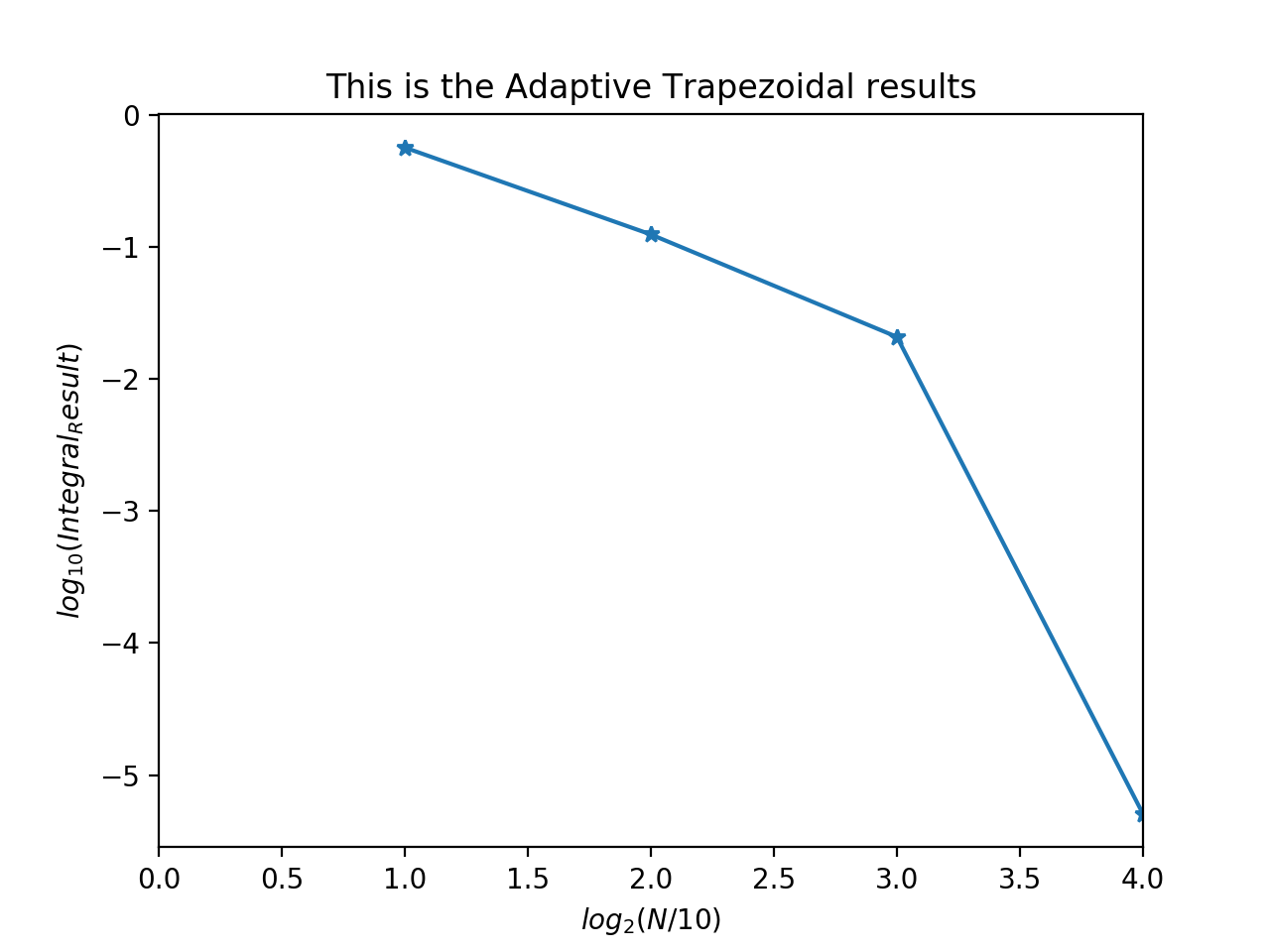
1. Euler-Maclaurin Rule

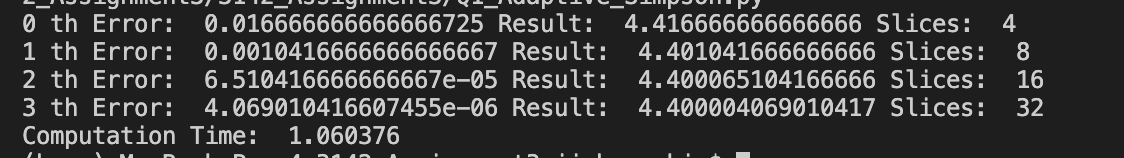
The analytical result from simple calculation shows that the result of the integral is equal to 4.4

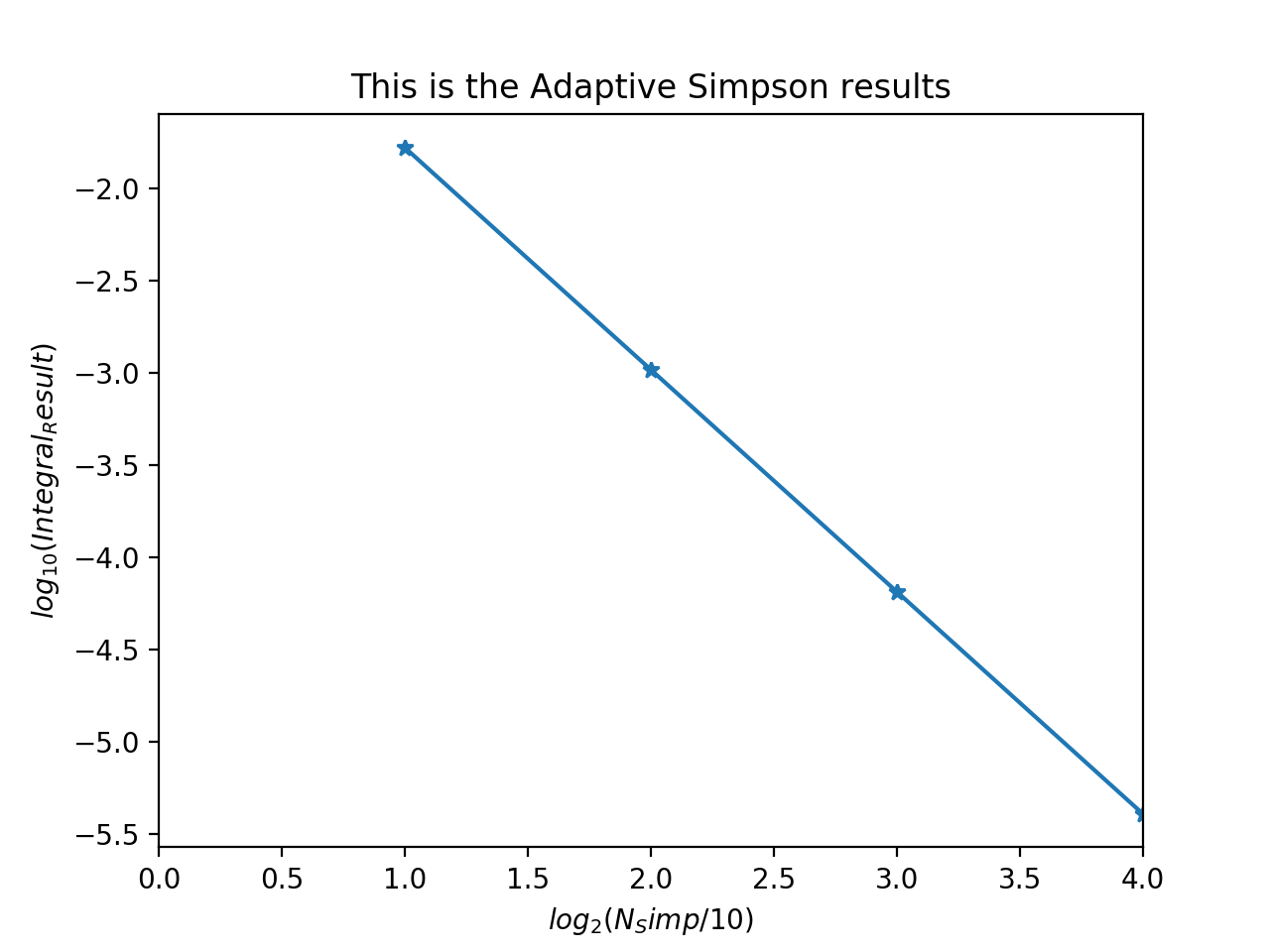
a) Adaptive Trapezoidal Rule

The result is somewhat accurate, but compared to the other two methods, it is not the best method, for accuracy, even though it provides extremely quick calculation with only 32 slices to achieve an error of 1e-6.



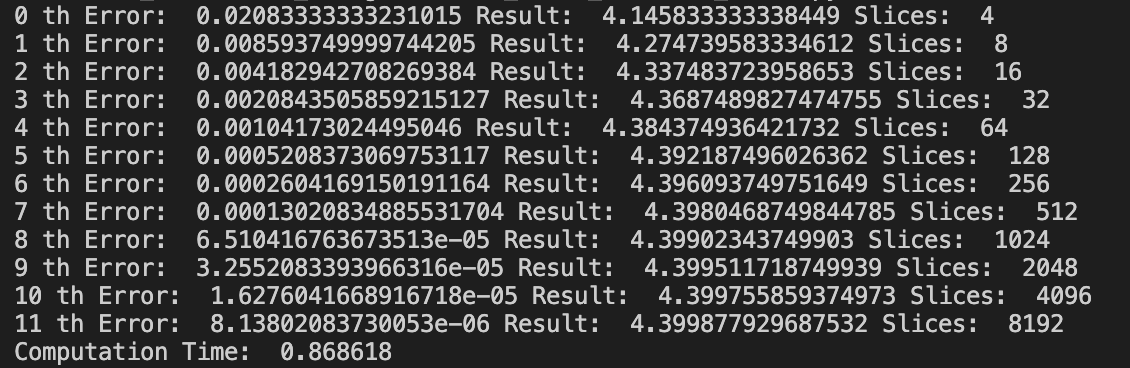
b) Adaptive Simpson



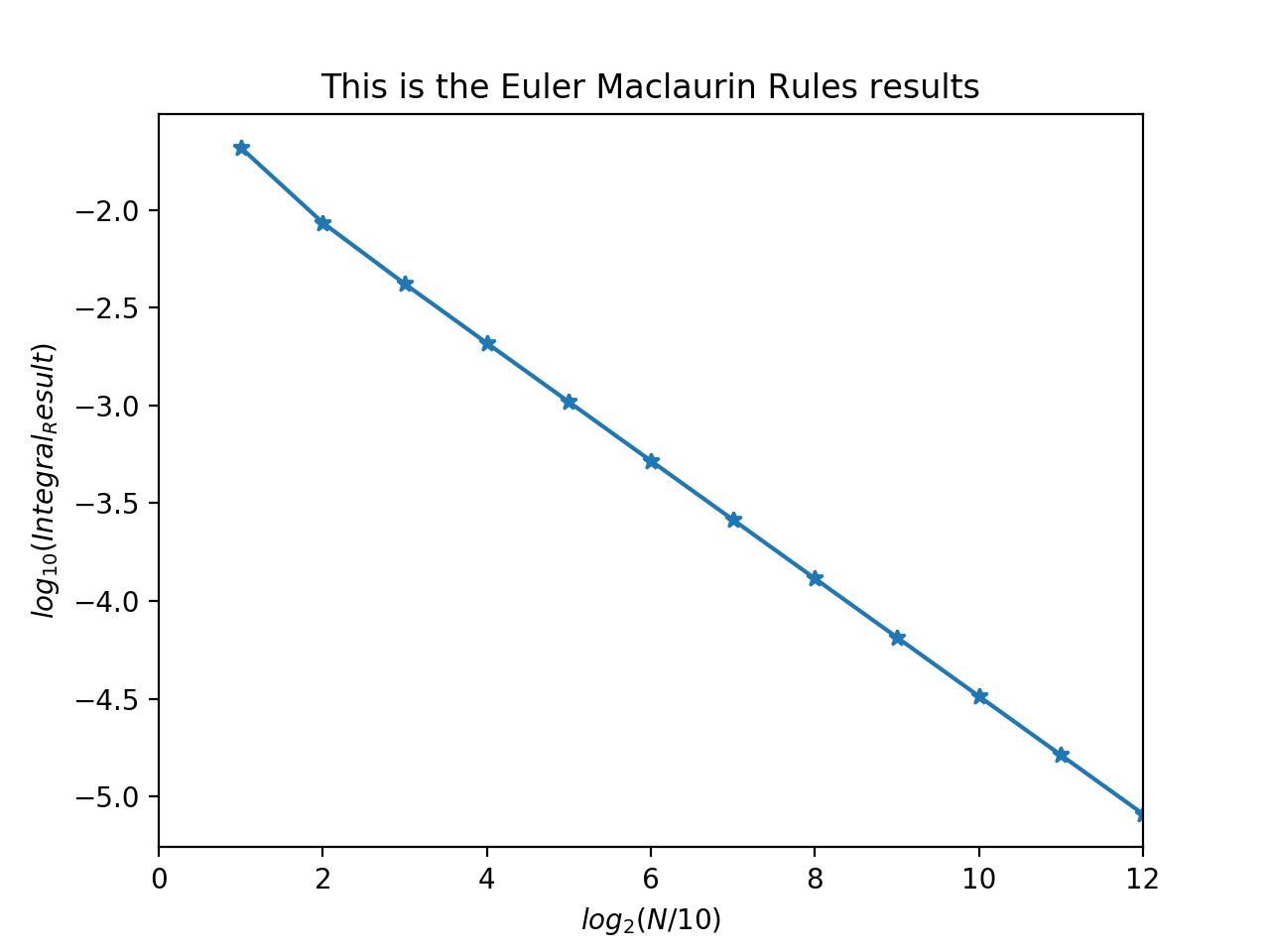


Like the Trapezoidal rule, the function had been imported from the previous assignment. However, since I had made a mistake in the previous Adaptive Simpson’s Rule, the code had been fixed as appropriate. The result is extremely accurate compared to the trapezoidal method even when the number of slices were equal.

c)

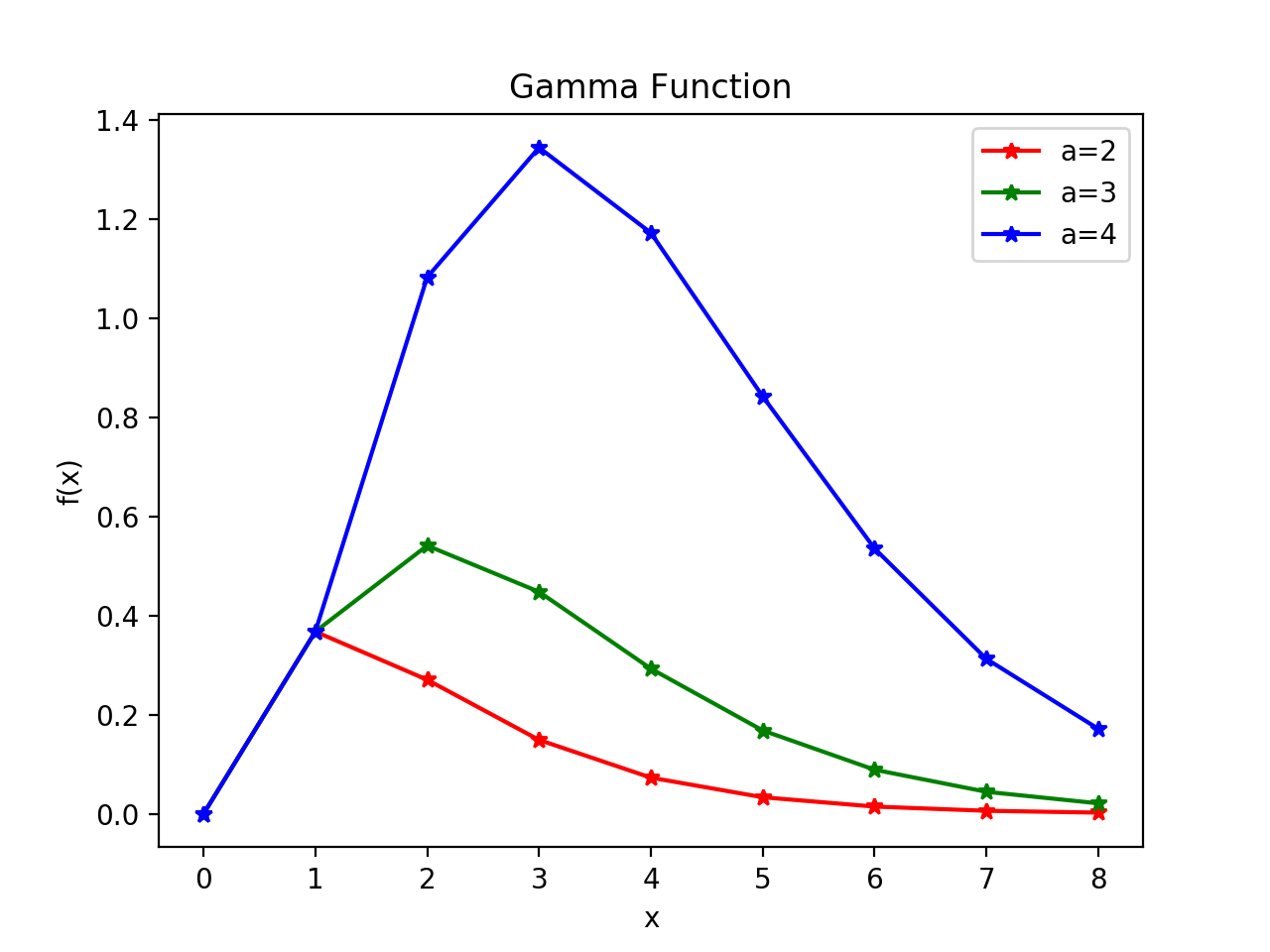


The Euler-Maclaurin Rule was not the most best method when compared to the previous two methods, because Trapezoidal and Simpson’s Rules required much less slices, and provided decent results. The Simpson’s rule outperformed Euler-Maclaurin Rule in both the accuracy and speed.



Q2)

a)



The three curves all start at zero, rises to their own maximums and decays as expected.

b)

import numpy as np

import matplotlib.pyplot as plt

import math

import time

from gaussxw import gaussxwab

# tried to find the value of c such that the function is at its maximum when z = 0.5

N = 10000

a = 6

# c = 6 when z=0.5 is maximum (found out through helper function below)

# c = 6

# c = 1 is the second part of part b

# c = 1

# c is closest to the analytical result of 120 when c = 6.50, where it returns 120.00000000000001

c = 6.50

x\_array = []

y\_array = []

# def f(z):

# x = c\*z/(1-z)

# y = (x\*\*(a-1))\*(np.e\*\*(-x))

# dxdz = c/((1-z)\*\*2)

# return (y\*dxdz)

def f(z):

i = ((c\*z)/(1-z))\*\*(a-1)

j = np.exp(-(c\*z)/(1-z))

k = c/((1-z)\*\*2)

return i\*j\*k

# print(f(0.))

# for i in range(N):

# x\_array.append(i)

x\_array = np.linspace(0.000000001,.999999999,N)

# print(x\_array)

for i in range(len(x\_array)):

# print('i: ', i)

y\_array.append(f(x\_array[i]))

# helper function to see if the maximum value of the function is at z=0.5

def findbiggestelement(array):

largest = 0

position = 0

for i in range(len(array)):

if array[i] > largest:

largest = array[i]

position = i

return largest, position

# print(findbiggestelement(y\_array))

# below returns 0.499949994999 which is close to z=0.5

# print(x\_array[4999])

# plt.figure()

# plt.title('Gamma Function Conversion Method')

# plt.plot(x\_array,y\_array)

# plt.xlabel('z')

# plt.ylabel('f(z)')

# axes = plt.gca()

# axes.set\_xlim([0,1])

# axes.set\_ylim([None,None])

# plt.legend()

# plt.show()

lower = 0.0

upper = 1.0

x,w = gaussxwab(N,lower,upper)

s = 0.0

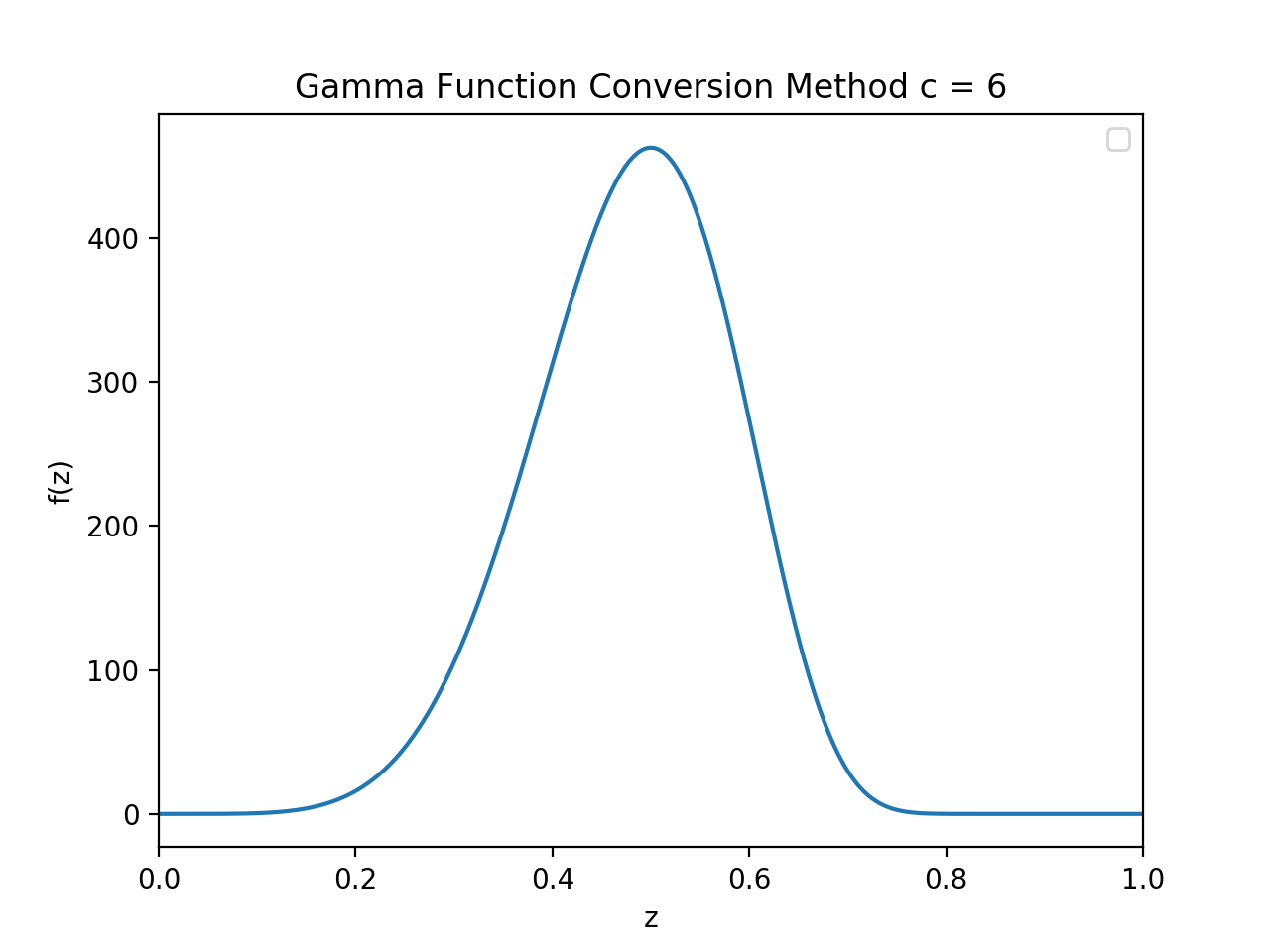
for i in range(N):

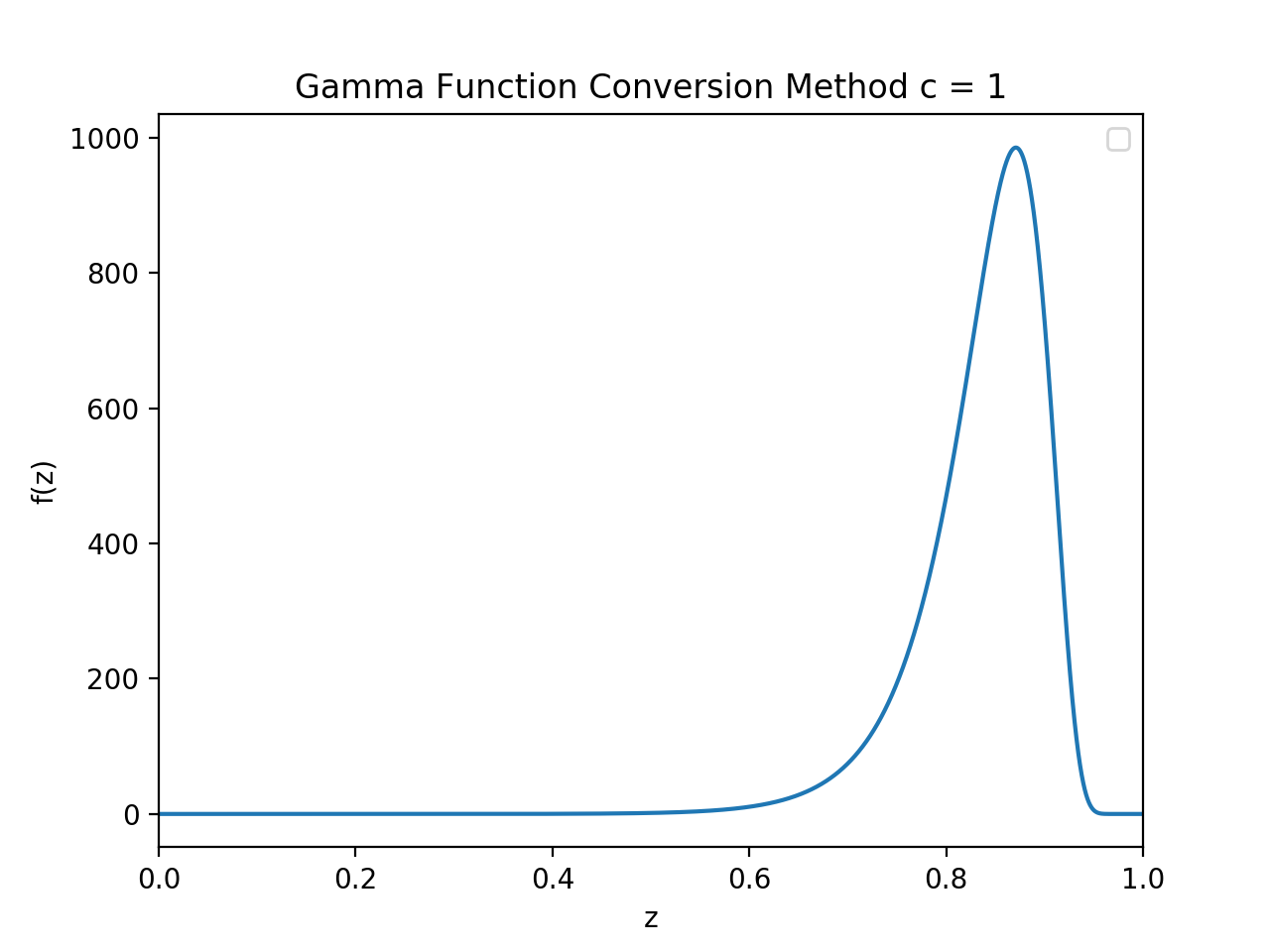
s += w[i]\*f(x[i])

print(s)

­­

Using the helper function above findbiggestelement(array), the value of ‘c’ to make the peak of the Transformed integrand at around z=0.5 could be found. Below is the plot to show the transformed integrand with c=6 (a=6 was used in this plot).



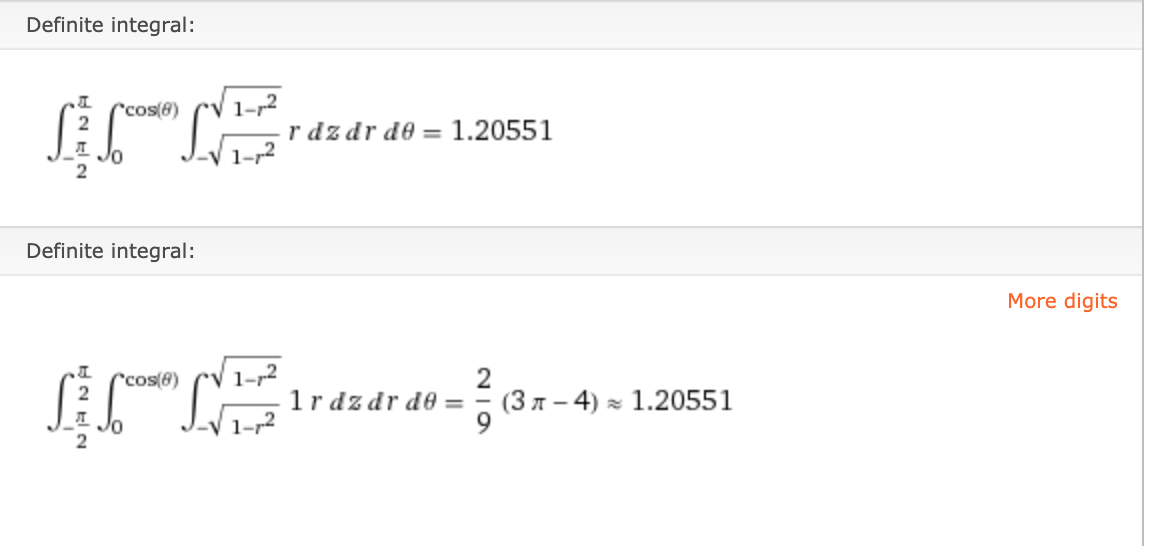


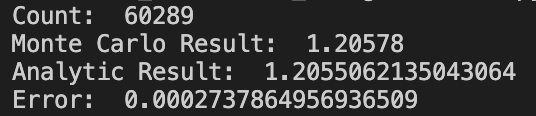
After adjusting the value of ‘c’ to adjust the output to approach the analytical result of 120, c=6.50 had been found to be extremely close to the result, with a difference of only 1e-13.

Q3)  
  
The logic I used in this method was to generate random points within the interval (a rectangle which the intersection would fit in), and if that point was inside the sphere AND the cylinder, it would be added to the count. After repeating for many repetitions, eventually:

The analytic result could be calculated using multivariable calculus by hand, giving:

The above result had been checked again on Wolfram:





The Monte Carlo result gave a reasonably accurate result with an error 1e-4. Considering how other traditional methods would either be impossible or be extremely difficult to implement, Monte Carlo result is extremely nice.

import numpy as np

import matplotlib.pyplot as plt

import math

from random import random

# equation of sphere which can be deduced from the question

def sphere(x,y,z):

return ((x\*\*2)+(y\*\*2)+(z\*\*2))

# equation of cylinder which can be deduced from the question

def cylinder(x,y):

return (((x-0.5)\*\*2)+(y\*\*2))

# helper function to check if random point is within the sphere

def in\_sphere(k):

if k < 1:

return 1

else:

return 0

# helper function to check if random point is within the cylinder

def in\_cylinder(k):

if k < 0.25:

return 1

else:

return 0

# N is the sampling number, count is used for counting

N = 100000

count = 0

for i in range(N):

x = random()

# y = -0.5\*random()+0.5\*random()

# z = -random()+random()

y= 0.5\*random()

z= random()

# if (in\_sphere(sphere(x,y,z)) and in\_cylinder(cylinder(x,y))):

# if point lies within the sphere and the cylinder, add 1 to counter

if (in\_sphere(sphere(x,y,z)) and in\_cylinder(cylinder(x,y))):

count+=1

# if(sphere(x,y,z)<1):

# count+=1

print('Count: ',count)

monte\_carlo\_result = 2\*count/N

# Monte Carlo Result

print('Monte Carlo Result: ', monte\_carlo\_result)

# Compare with analytical result that can be easily calculated using multivariable calculus

analytic\_result = ((2/9)\*((3\*np.pi)-4))

print('Analytic Result: ',analytic\_result)

error = monte\_carlo\_result-analytic\_result

print('Error: ', error)