PHYS 310 Work and Results

Khalil El Achi

Table of Content

1 Interpolation **2**

1.1 Lagrange Interpolation 2

1.1.1 Manual Function 2

1.2 Atkins Method 4

1.3 Quadratic Splines 7

2 Fitting **10**

2.1 Least Square Approximation 10

3 Integration Methods **13**

3.1 Trapezoidal Rule 13

3.2 Simpson's Rule 14

4 Root Finding Method **15**

4.1 Bisection method 15

5 Newton-Raphson **16**

1 Interpolation

1.1 Lagrange Interpolation

1.1.1 Manual Function

The first attempt to code a second-degree Lagrange Interpolation of the data sets modeling

$$y = x^2$$

is by utilizing a self-coded function:

```
import numpy as np

import matplotlib.pyplot as plt

from scipy.interpolate import lagrange


# Starting with the manual Interpolation


x2=np.array([1, 3, 5])

y2=np.array([0.8, 10, 23.5])


x_lagrange=np.arange(0,10,0.1)


def lagrange_interpol(x,x2, y2):

    P1 = (y2[0]*(x-x2[1])*(x-x2[2]))/((x2[0]-x2[1])*(x2[0]-x2[2]))

    P2 = (y2[1]*(x-x2[0])*(x-x2[2]))/((x2[1]-x2[0])*(x2[1]-x2[2]))

    P3 = (y2[2]*(x-x2[0])*(x-x2[1]))/((x2[2]-x2[0])*(x2[2]-x2[1]))

    y = P1 + P2 + P3

    return y
```

```
x1= np.arange(0,10, 0.1)
y1 = np.array([])
for i in x1:
       y1 = np.append(y1, lagrange_interpol(i,x2,y2))


plt.plot(x2[0],y2[0], 'bo',label='P1')
plt.plot(x2[1],y2[1], 'mo',label='P2')
plt.plot(x2[2],y2[2], 'ko',label='P3')
plt.plot(x1,y1, color='r')
plt.grid()
plt.legend()
plt.title('Lagrange Interpolation of y=x^2')
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```

The outputted Graph recovers the data points and the result of the interpolation:

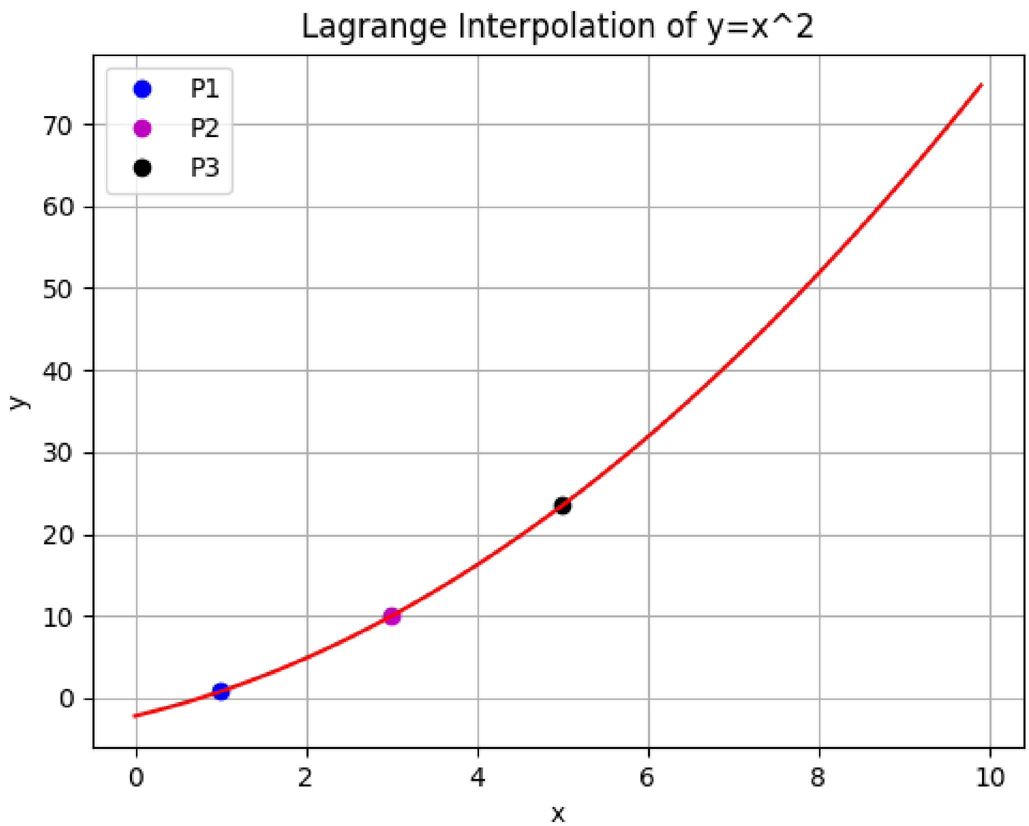

Figure 1: Lagrange Interpolation using a Manual Function

1.2 Atkins Method

Attempting to Model

$$y = \frac{1}{1+x}$$

using both the Atkins Method of Polynomials and the built in function in Scipy library.

```
        import numpy as np
        import matplotlib.pyplot as plt
        from scipy.interpolate import lagrange
# Newton/Atkins Method And Quadratic Splines
def Atkins_method(x,x1,y1):
        p12 = (x-x1[1])/(x1[0] - x1[1]) * y1[0] + (x-x1[0])/(x1[1]-x1[0]) * y1[1]
```

```
        p23 = (x-x1[2])/(x1[1] - x1[2]) * y1[1] + (x-x1[1])/(x1[2]-x1[1]) * y1[2]
        p34 = (x-x1[3])/(x1[2] - x1[3]) * y1[2] + (x-x1[2])/(x1[3]-x1[2]) * y1[3]
        p123 = (x-x1[2])/(x1[0] - x1[2]) * p12 + (x-x1[0])/(x1[2]-x1[0]) * p23
        p234 = (x-x1[3])/(x1[1] - x1[3]) * p23 + (x-x1[1])/(x1[3]-x1[1]) * p34
        p1234 = (x-x1[3])/(x1[0] - x1[3]) * p123 + (x-x1[0])/(x1[3]-x1[0]) * p234
        return p1234

# Modeling equation f(x) = 1/1+x

x_f=np.array([1, 2, 3, 4])
y_f=np.array([0.45, 0.35, 0.23, 0.18])

x_r=np.linspace(np.min(x_f),np.max(x_f),50)
y_quadratic = interp1d(x_f, y_f, kind='quadratic')
y_r=np.array([])
for i in x_r:
        y_r = np.append(y_r,Atkins_method(i,x_f,y_f))
y_real=1/(1+x_r)
plt.figure()
plt.plot(x_f[0],y_f[0], 'bo',label='P1')
plt.plot(x_f[1],y_f[1], 'mo',label='P2')
plt.plot(x_f[2],y_f[2], 'ko',label='P3')
plt.plot(x_f[3],y_f[3], 'ro',label='P4')
plt.plot(x_r,y_r, color='b',label='Atkins Interpolation')
plt.plot(x_r,y_real,color='k',label='Real Function')
plt.plot(x_r,y_quadratic(x_r),color='m',label='Quadratic Splines')
plt.grid()
```

```
plt.legend()

plt.title('Atkins Method for Interpolating y=1/1+x')

plt.xlabel('x')

plt.ylabel('y')

plt.show()


# Scipy Python Library

f_lag= lagrange(x2,y2)

plt.figure

plt.plot(x1,f_lag(x1), 'g' , x2, y2 , 'mo')

plt.grid()

plt.legend()

plt.title('Lagrange Interpolation of y=x^2')

plt.xlabel('x')

plt.ylabel('y')

plt.show()
```

Here one can see when the function doesn't follow an apparent trend or when the data points are not too far off and don't cover a wide range the estimated interpolation is very far from the actual function being modeled. This is seen below.

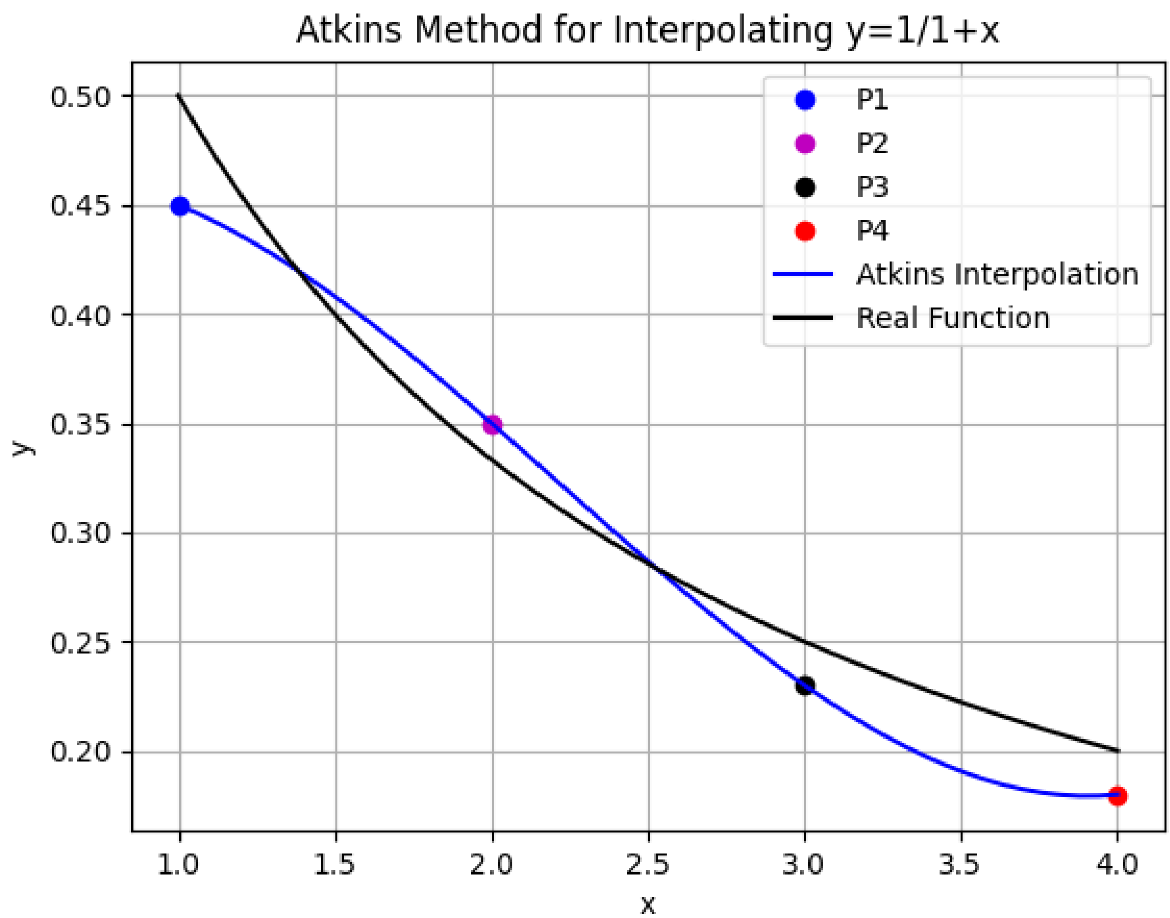

Figure 2: Bad Interpolation for $y = \frac{1}{1+x}$

1.3 Quadratic Splines

Using the built in function scipy.interpole from SciPy, one can produce better results using Quadratic Splines.

```
from scipy.interpolate import interp1d
y_quadratic = interp1d(x_f, y_f, kind='quadratic')
```

This will output a quadratic splines estimate between the data points and is a more accurate estimate than Lagrange interpolation as the figure below shows.

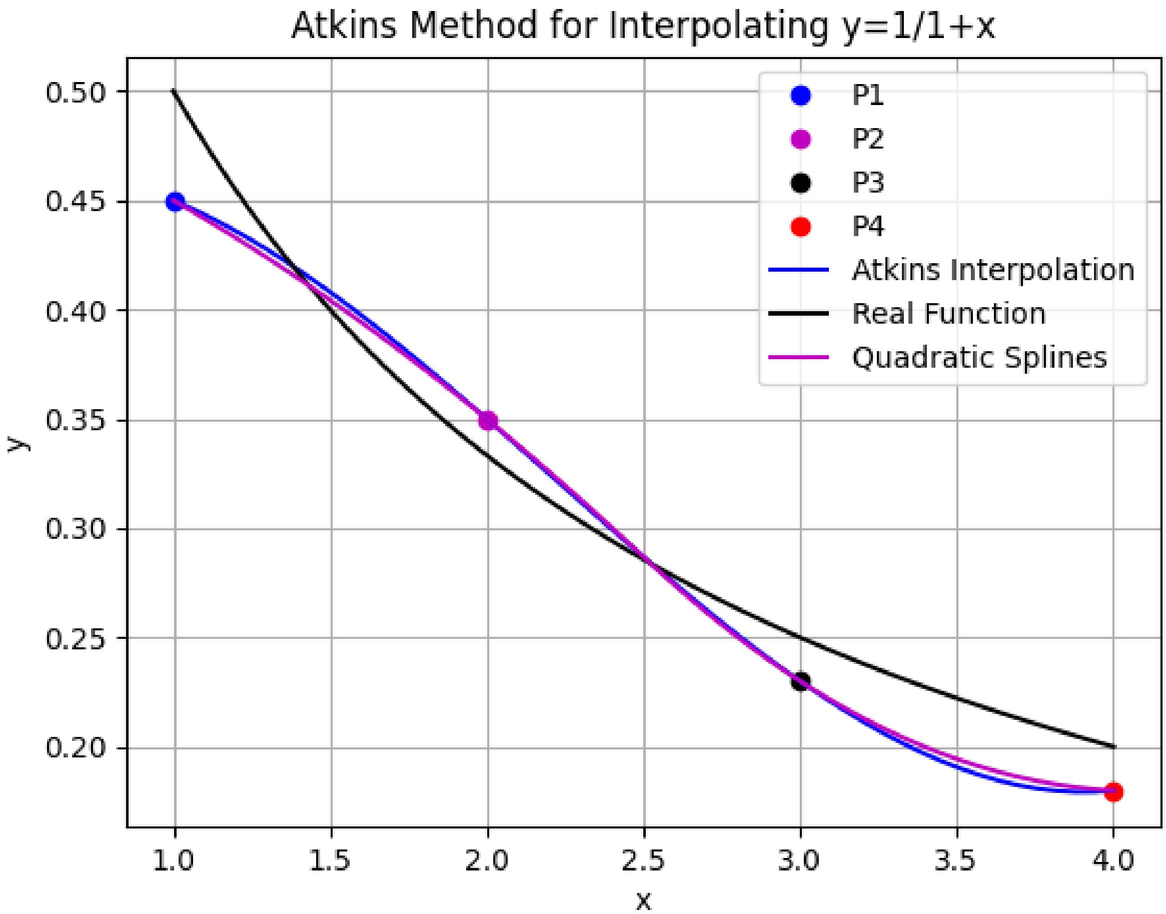

Figure 3: Quadratic Splines and Lagrange Interpolation for $y = \frac{1}{1+x}$

Additionally, below is the code used for the function lagrange in scipy.interpole that automatically outputs the Lagrange Interpolation of the the Data point. Below is the code and the output for Data Sets modeling $y = \frac{1}{1+x}$.

```
# Scipy Python Library

f_lag= lagrange(x_f,y_f)

plt.figure

plt.plot(x_r,f_lag(x1), 'g' , x_f, y_f , 'mo')

plt.plot(x_r,y_real,color='k')

plt.grid()

plt.legend()
```

```
plt.title('Lagrange Interpolation of y=x^2')

plt.xlabel('x')

plt.ylabel('y')

plt.show()
```

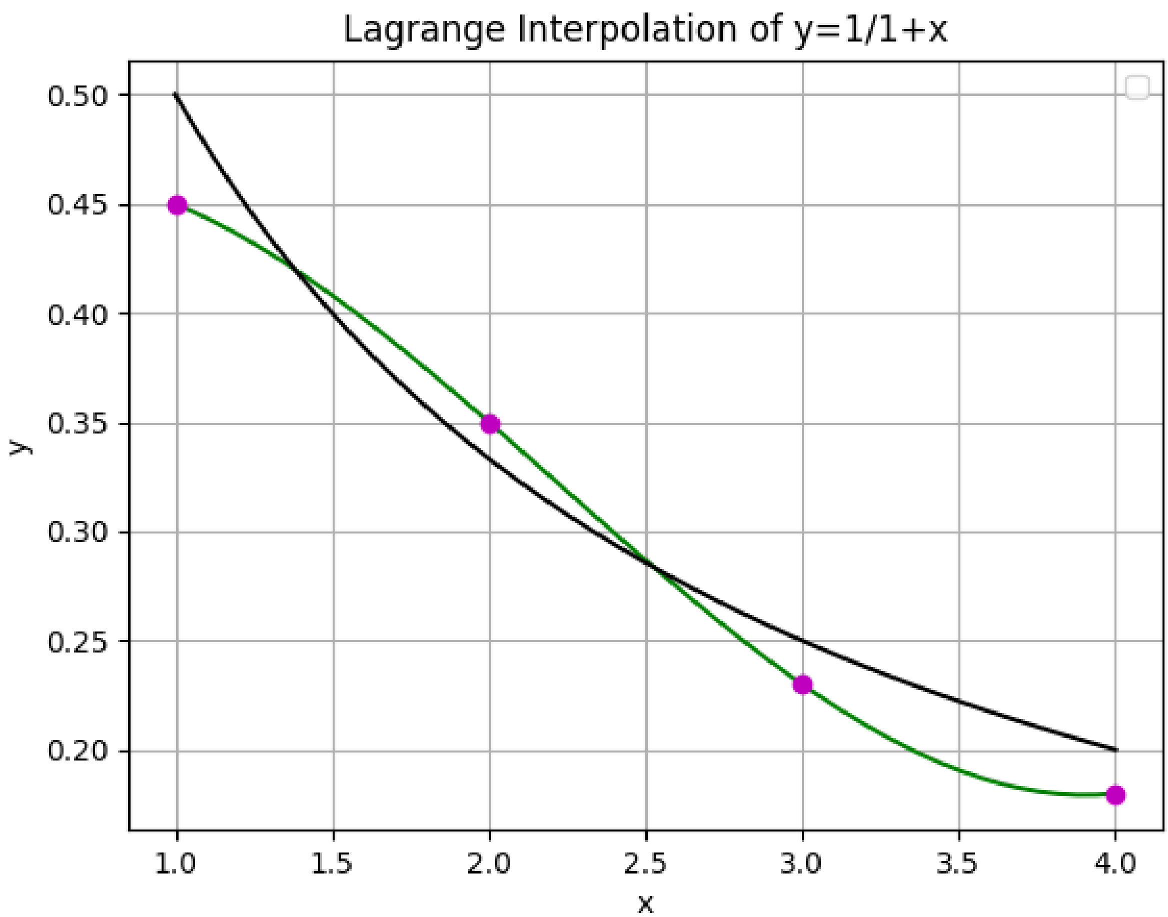

Figure 4: Lagrange Interpolation Using the Built-in function in SciPy Library for $y = \frac{1}{1+x}$

The output is identically the same as the ones above. The only difference that will help the interpolation is introducing more points in different ranges to have a better estimate towards the trend of the points.

2 Fitting

2.1 Least Square Approximation

To get the line of best-fit for

$$y = 2x + 3$$

The following code is executed. The two methods being used, are the manual method were the coefficients of the linear linear to describe the fit $y = mx + b$ are calculated using the sums and mean of the data point. The other method is to write up the data points in matrix form and solve for the coefficients. The last is the built in function, optimize.curve_fit in scipy.

```
import numpy as np
from scipy import optimize
import matplotlib.pyplot as plt

# generating data of equation y = 2x + 3 with errors
x= np.linspace(0,20,50)
n=len(x)
y= 2*x + 3 + 10* np.random.random(n)

# Physically getting the least squares through the sums equations

xbar = np.mean(x)
ybar = np.mean(y)

num = 0
den = 0
for i in range(n):
        num += (x[i] - xbar) * (y[i] - ybar)
```

```
        den += (x[i] - xbar) ** 2
m = num/den # slope
b = ybar - (m * xbar) # y-intercept
print(m, b)
y_reg = m*x + b
plt.plot(x,y,'r*',label='Data Point')
plt.plot(x,y_reg,color='b',label='Linear Regression')
plt.title('Least Square Approximation of y = 2x + 3')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.grid()
plt.show()

# Writing it in Matrix formation
A= np.vstack([x,np.ones(len(x))]).T
y = y[:,None] # changing y into a column vector
alpha = np.dot((np.dot(np.linalg.inv(np.dot(A.T,A)),A.T)),y) # Getting
print(alpha)
plt.figure()
plt.plot(x,y,'r*',label='Data Point')
plt.plot(x,alpha[0]*x + alpha[1],color='b',label='Linear Regression')
plt.legend()
plt.grid()
plt.show()

# Using optimize.curve_fit from scipy
```

```
def func(x,m,b):
    y=m*x + b
    return y


beta = optimize.curve_fit(func,xdata=x,ydata=y)[0]
print(beta)
```

The following code outputs the same value for the coefficients:

$$m = 2.07772003$$

$$b = 7.21019203$$

And the following graph describing the fit:

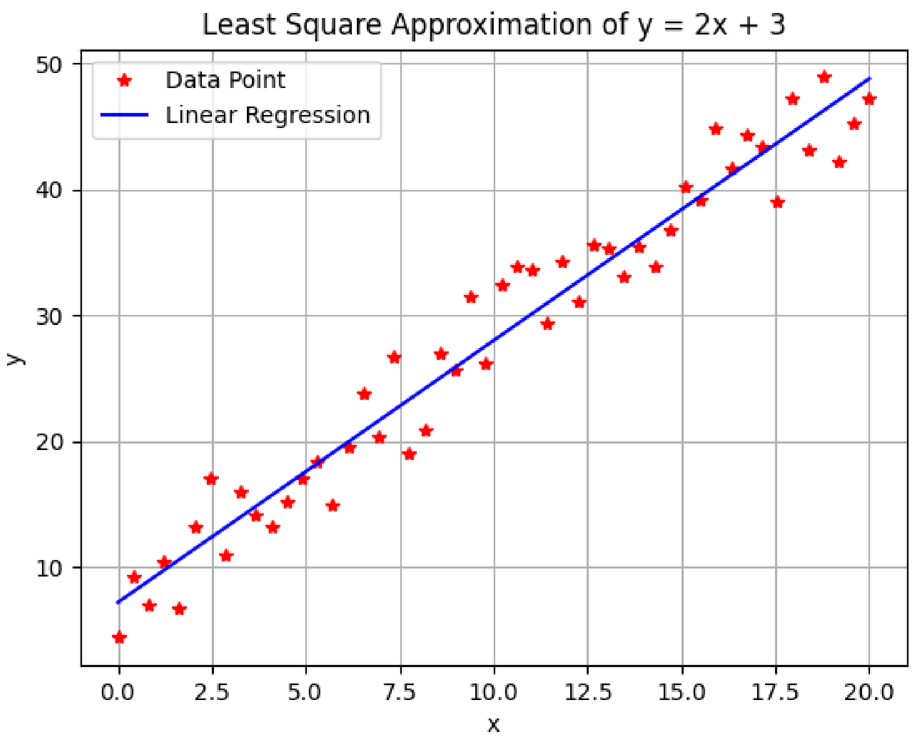

Figure 5: Linear Regression using Least Square Method

3 Integration Methods

Calculating the integral of

$$f(x) = \sin(x)$$

using the Trapezoidal Rule, Simpson's Rule, and the built in packages in Scipy.

3.1 Trapezoidal Rule

The code written will emulate this formula, where x_0 and x_n are the boundary points:

$$I_{Trapizoidal} = \frac{h}{2}(f(0) + f(n)) + h\sum_{i=1}^{n-1} f(x_n)$$

The code that will implement the rule:

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


x1 = 0
x2 = np.pi
n=100
h= (x2 - x1) / (n-1 )
x = np.linspace(x1,x2,n)
func = np.sin(x) #+ np.random.random(n)


# Trapizoidal Rule


I_trapizoid = (h/2) * (func[0] + func[n-1]) + np.sum(func[1:n-1]) * h
print(I_trapizoid)
```

```
plt.plot(x,func)
plt.show()
```

3.2 Simpson's Rule

The Simpson's rule will emulate the following relation:

$$I_{simpson} = \frac{h}{3}(f(x_0) + f(n)) + \frac{2h}{3}\sum_{i=1}^{n/2-1} f(x_{2i})\frac{4h}{3}\sum_{i=1}^{n/2} f(x_{2i-1})$$

The code that will implement the rule:

```
# Simpson's Rule

I_simpson = (h/3) * (func[0] + func[n-1]) + (2*h/3) * (np.sum(func[:n
    ↪ -2:2])) + (4*h/3) * (np.sum(func[1:n-1:2]))
print(I_simpson)

# Python integrator

def f(x):
       return np.sin(x)

I_package = quad(f,x1,x2)
print(I_package)
```

Both methods output the following results: 1.9998321638939927, 1.9991609434038167, and 2.0.
The theoretical answer is 2 so the methods are successful.

4 Root Finding Method

I intend to find the root of:

$$f(x) = \sin(x)$$

using the Bisection Method and Newton-Raphson method.

4.1 Bisection method

Using initial boundaries of $\frac{\pi}{4}$ and $\frac{3\pi}{2}$

The code that will implement this method is:

```
import numpy as np


def bisection_method(f,x1,x2,e):

        if f(x1)*f(x2) >=0:

                return print('No root exists in these boundaries')

        x1_n=x1

        x2_n=x2

        while (x2_n - x1_n) < e:

                mid = (x1_n + x2_n) / 2

                f_mid = f(mid)

                if f(x1_n) * f_mid < 0:

                        x2_n = mid

                elif f(x2_n) * f_mid < 0:

                        x1_n = mid

                elif f_mid == 0:

                        return print('Exact solution found: ',mid)

                else:

                        return print('Method Failed')
```

```
    return print((x1_n + x2_n) / 2)


bisection_method(f,np.pi/2,1.5*np.pi,0.001)
```

The root finding method outputs a root of: 3.141592653589793

Which is correct

5 Newton-Raphson

The initial guess for the Newton-Raphson method is: $\frac{\pi}{4}$

```
def f(x):
    return np.sin(x)
def df(x):
    return np.cos(x)


def Newton_Raphson(f,df,x,e):
    h = f(x) / df(x)
    while abs(h) > e:
        h=f(x) / df(x)
        x = x - h
    print('The root found is: ',x)


Newton_Raphson(f,df,np.pi/4,0.0001)
```

It outputs: 1.6543612251060553e-24, which is almost zero which is the correct root.