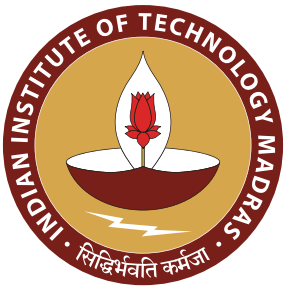


Department Of Aerospace Engineering
AS2101
Prof. Bharath Govindarajan
Prof. Ramakrishna M

NUMERICAL COMPUTATION OF A UNIVARIATE DEFINITE INTEGRAL

TASK 4: 31-07-2021(Due-Date)

REPORT

AUTHOR
Yash Singh Jha-(AE19B016)

July 31 , 2021

Contents

List of Figures

Abstract

This document is like a summary report of the work we were assigned as Task 4 in the course-AS2101.
This report presents a few of the various methods/algorithms used to calculate the numerical integral of a definite integral of a univariate function. We discuss two main methods - **Rectangular Method** and **Trapezoidal Method**. Three submethods of the Rectangular Method - Left Endpoint, Right Endpoint and the Midpoint Methods, are also discussed.
Chapter 1 - **Numerical Quadrature**, deals with the theory behind each of the above mentioned methods and illustrates the philosophy behind it elegantly using appropriate equations and figures. It also explains the theory behind absolute error and order of accuracy of each of these methods.
Chapter 2 - **Numerical Computation : Python Implementation**, presents and discusses the methods in Python - their characteristics and differences.
Chapter 3 - **Results**, takes us through the results obtained using the Python codes presented in chapter 2 and compares the computational differences between them.
Chapter 4 - **Conclusion**, presents a concluding remark on the overall report and the materials covered. It also gives a comment on the relative accuracy/usage of the methods.
This LaTeX file itself is the part of the task for the third week of AS2101.

Chapter 1

Numerical Quadrature

In analysis, numerical integration comprises a broad family of algorithms for calculating the numerical value of a definite integral, and by extension, the term is also sometimes used to describe the numerical solution of differential equations. This report focuses on calculation of definite integrals of univariate functions. Refer figure 1.1.

The term numerical quadrature (often abbreviated to quadrature) is more or less a

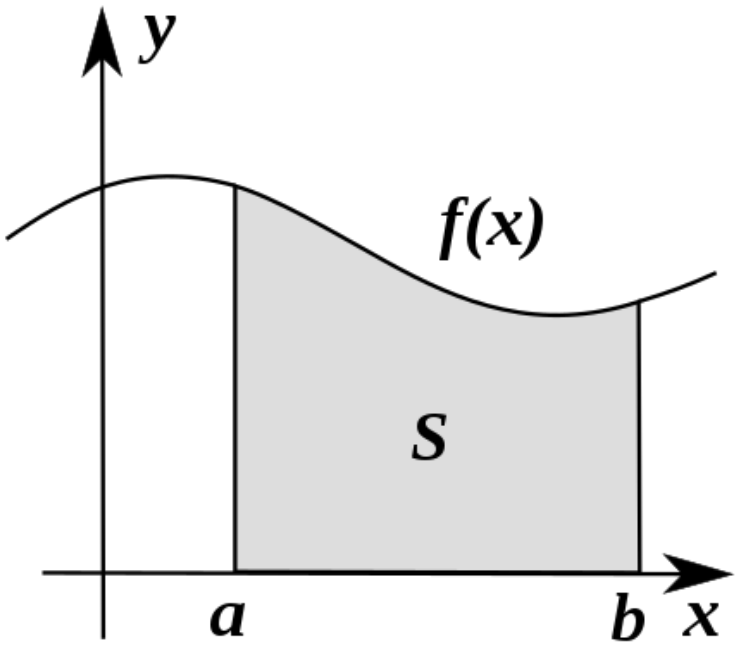


Figure 1.1: Numerical integration is used to calculate a numerical approximation for the value **S**, the area under the curve defined by **f(x)**.

synonym for numerical integration, especially as applied to one-dimensional integrals. Some authors refer to numerical integration over more than one dimension as cubature; others take quadrature to include higher-dimensional integration.

The basic problem in numerical integration is to compute an approximate solution to a definite integral

$$\int_a^b f(x)\,dx \tag{1.1}$$

to a given degree of accuracy. If $f(x)$ is a smooth function integrated over a small number of dimensions, and the domain of integration is bounded, there are many methods for approximating the integral to the desired precision.

1.1 Reasons for Numerical Integration

There are several reasons for carrying out numerical integration, as opposed to analytical integration by finding the antiderivative:

1. The integrand $f(x)$ may be known only at certain points, such as obtained by sampling. Some embedded systems and other computer applications may need numerical integration for this reason.
2. A formula for the integrand may be known, but it may be difficult or impossible to find an antiderivative that is an elementary function. An example of such an integrand is $f(x) = e^{-x^2}$, the antiderivative of which (the error function, times a constant) cannot be written in elementary form.
3. It may be possible to find an antiderivative symbolically, but it may be easier to compute a numerical approximation than to compute the antiderivative. That may be the case if the antiderivative is given as an infinite series or product, or if its evaluation requires a special function that is not available.

1.2 Rectangle Method

Here, we wish to approximate a definite integral 1.1 where $f(x)$ is a continuous function. We begin by choosing pointsx_i that subdivide $[a, b]$:

$$a = x_0 < x_1 < ..... < x_{n-1} < x_n = b \tag{1.2}$$

The subintervals $[x_{i-1}, x_i]$ determine the width Δx_i of each of the approximating rectangles. For height, we can choose any value $f(x*)$, where x* ϵ $[x_{i-1}, x_i]$. the resulting approximation is

$$\int_a^b f(x)\,dx \approx \Sigma_{i=1}^n f(x*_i)\Delta x_i \tag{1.3}$$

To use this to approximate integrals with actual numbers, we need to have a specific $x*_i$ in each interval. The two simplest (and relatively erroneous) ways to choose $x*_i$ are as the left-hand point or the right-hand point of the each subinterval. This gives concrete approximations -Left Riemann Sum and Right Riemann Sum, which we denote by L_n and R_n given by

$$L_n = \Sigma_{i=1}^n f(x_{i-1})\Delta x_i \tag{1.4}$$

and

$$R_n = \Sigma_{i=1}^n f(x_i)\Delta x_i \tag{1.5}$$

To keep things from getting too messy, we choose our subintervals of equal length i.e $\Delta x_i = \Delta x$. The approximations become better as the number of subintervals increase.

1.2.1 Left-Endpoint Method

For the left Riemann sum, approximating the function by its value at the left-end point gives multiple rectangles with base Δx and height $f(a + i\Delta x)$. Doing this for $i = 0, 1, ..., n - 1$ and adding up the resultant areas gives

$$L_n = \Delta x[f(a) + f(a + \Delta x) + f(a + 2\Delta x) + ... + f(b - \Delta x)] \tag{1.6}$$

The left Riemann sum amounts to an overestimation if f is monotonically decreasing on

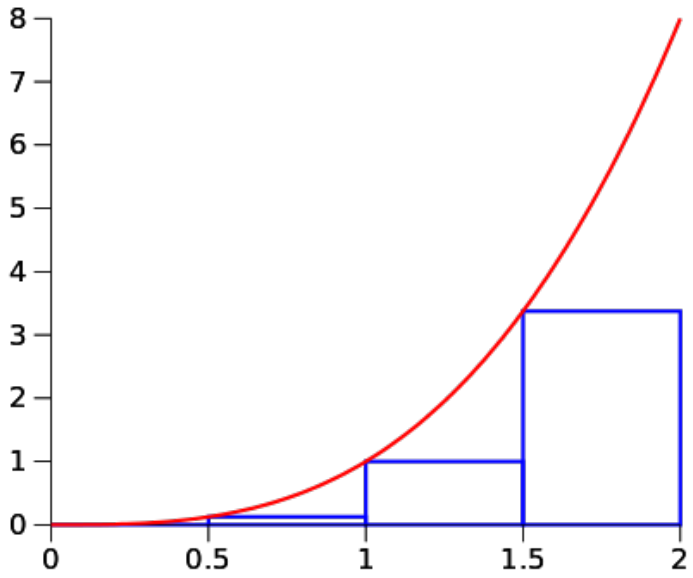


Figure 1.2: Left Riemann sum of x^3 over [0,2] using 4 subdivisions

this interval, and an underestimation if it is monotonically increasing. Refer 1.2.

1.2.2 Right-Endpoint Method

For the right Riemann sum, approximating the function by its value at the right-end point gives multiple rectangles with base Δx and height $f(a + i\Delta x)$. Doing this for $i = 0, 1, ..., n - 1$ and adding up the resultant areas gives

$$R_n = \Delta x[f(a + \Delta x) + f(a + 2\Delta x) + ... + f(b)] \tag{1.7}$$

The right Riemann sum amounts to an overestimation if f is monotonically increasing on this interval, and an underestimation if it is monotonically decreasing.Refer 1.3.
The error of this formula will be

$$\left| \int_a^b f(x)\,dx \; - R_n \right| \leq \frac{M_1(b-a)^2}{2n} \tag{1.8}$$

where M_1 is the maximum value of $|f'(x)|$ on the interval.

1.2.3 Midpoint Method

Approximating f at the midpoint of the intervals gives $f(a + \Delta x/2)$ for the first interval, for the next one $f(a + 3\Delta x/2)$, and so on until $f(b - \Delta x/2)$.Refer 1.4. Summing up the

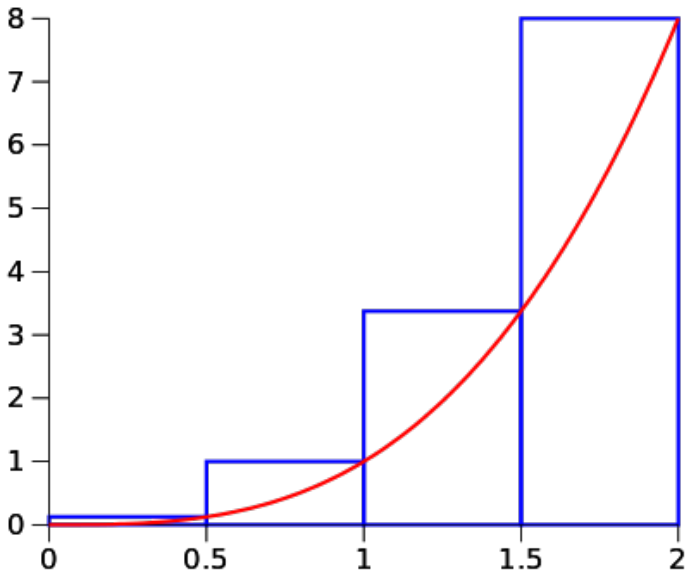


Figure 1.3: Right Riemann sum of x^3 over [0,2] using 4 subdivisions

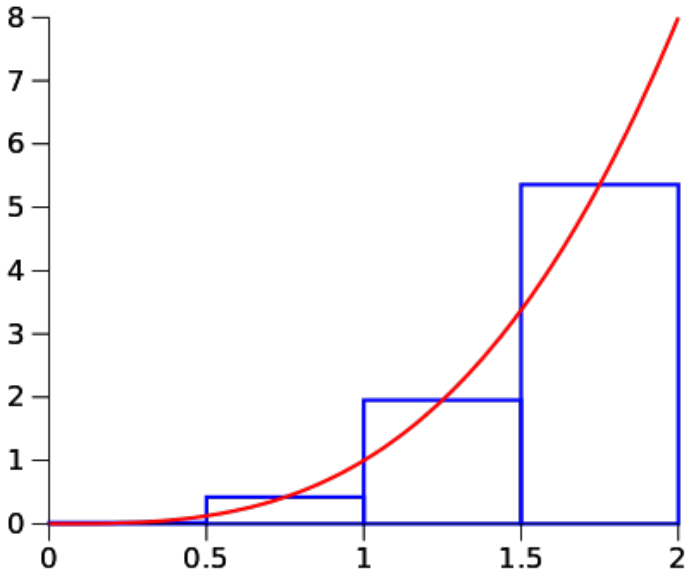


Figure 1.4: Midpoint Riemann sum of x^3 over [0,2] using 4 subdivisions

areas gives

$$M_n = \Delta x[f(a + \frac{1}{2}\Delta x) + f(a + \frac{3}{2}\Delta x) + ... + f(b - \frac{1}{2}\Delta)] \tag{1.9}$$

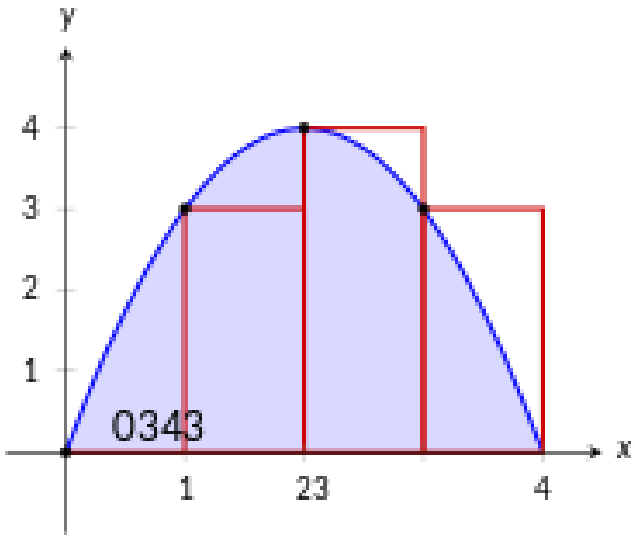
The error of this formula will be

$$\left| \int_a^b f(x)\,dx \ - M_n \right| \leq \frac{M_2(b-a)^3}{24n^2} \tag{1.10}$$

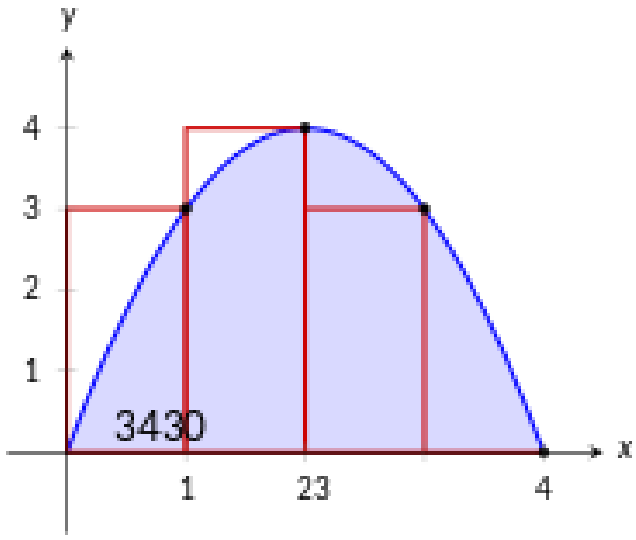
where M_2 is the maximum value of the absolute value of $f''(x)$ on the interval.

1.2.4 Example: Using Left, Right and Midpoint rules.

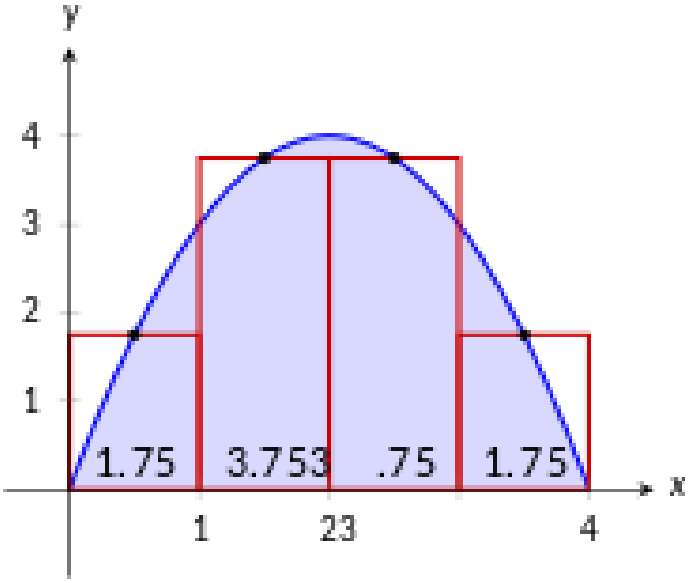
In this subsection, we explain the usage of Left, Right and Midpoint Rules by approximating the area under $f(x) = 4x - x^2$ on the interval $[0, 4]$ using each of the methods, using 4 equally spaced intervals.
In figure 1.5a, we see four rectangles drawn on $f(x) = 4x - x^2$ using the Left Hand Rule. (The areas of the rectangles are given in each figure.) Note, how in the first subinterval



(a) Left Hand Rule.



(b) Right Hand Rule



(c) Midpoint Rule

[0,1], the rectangle has height $f(0) = 0$. We add up the areas of each rectangle for our Left Hand Rule approximation:

$$f(0).1 + f(1).1 + f(2).1 + f(3).1 = 0 + 3 + 4 + 3 = 10 \tag{1.11}$$

Figure 1.5b, shows 4 rectangles drawn under $f(x)$ using the Right Hand Rule; note how the [3,4] subinterval has a rectangle of height 0. In this figure, these rectangles seem to be the mirror image of those found in figure 1.5a. (This is because of the symmetry of our shaded region.). Our approximation gives the same answer as before, though calculated

a different way:

$$f(1).1 + f(2).1 + f(3).1 + f(4).1 = 3 + 4 + 3 + 0 = 10 \tag{1.12}$$

Figure 1.5c, shows 4 rectangles drawn under $f(x)$ using the Midpoint Rule. This gives an approximation of the area as :

$$f(0.5).1 + f(1.5).1 + f(2.5).1 + f(3.5).1 = 1.75 + 3.75 + 3.75 + 1.75 = 11 \tag{1.13}$$

Our three methods provide two approximations of the area under $f(x) = 4x - x^2$: **10** and **11**.

1.3 Trapezoid Method

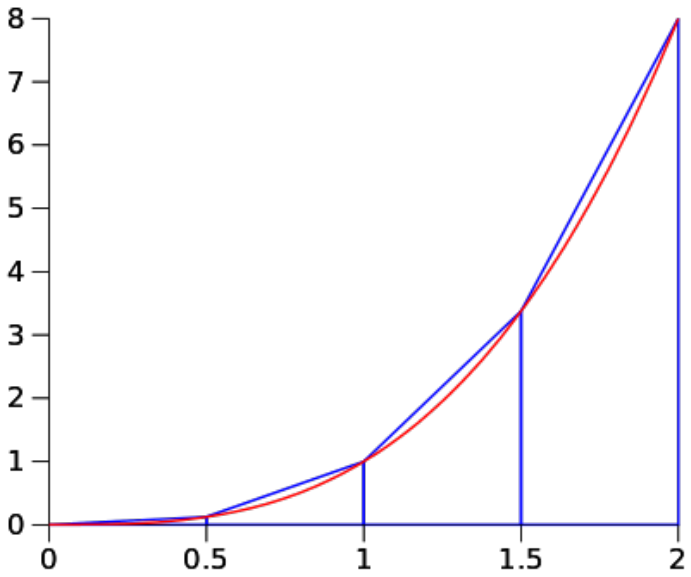


Figure 1.6: Trapezoidal Riemann sum of x^3 over [0,2] using 4 subdivisions

In this case, the values of the function f on an interval are approximated by the average of the values at the left and right endpoints.Refer figure 1.6 In the same manner as above, a simple calculation using the area formula

$$A = \frac{h(b_1 + b_2)}{2} \tag{1.14}$$

for a trapezium with parallel sides b_1, b_2 and height h produces

$$A_{trap} = \frac{1}{2}\Delta x[f(a) + 2f(a + \Delta x) + 2f(a + 2\Delta x)... + f(b)] \tag{1.15}$$

The error of this formula will be

$$\left| \int_a^b f(x)\,dx \ - A_{trap} \right| \leq \frac{M_2(b-a)^3}{12n^2} \tag{1.16}$$

where M_2 is the maximum value of the $|f''(x)|$.
The approximation obtained with the trapezoid rule for a function is the same as the average of the left hand and right hand sums of that function.

1.4 Absolute Error

An important aspect of using these numerical approximation rules consists of calculating error in using them for estimating the value of a definite integral. We first need to define absolute error and relative error.

Absolute Error: $|E|$ If B is our estimation of some quantity having an actual value of A, then the absolute error is given by $|A - B|$.

Relative Error: The relative error is the error as a percentage of the actual value and is given by

$$\left|\frac{A-B}{A}\right| .100\% \tag{1.17}$$

Evaluating f at many points may be computationally expensive, or difficult for other reasons, e.g if f is given by measurements. It is therefore important to consider how the error E depends on n. Obviously, one would expect E to become small as n becomes large. In a set of measurements, accuracy is closeness of the measurements to a specific value. So, even by logic the consideration of order of accuracy in our analysis/results is very important. We define it as follows.

Order of Accuracy: It shows how fast $|E|$ decays to zero when we take a smaller distance h between the nodes. Since, $n \approx \frac{1}{h}$ in these rules (at least in 1D), it similarly describes how fast $E \to 0$ when $n \to$ inf. We define the order of accuracy as the largest number p for which there exists a constant C, independent of h for which

$$|E| \leq Ch^p \tag{1.18}$$

For our report,

$$|E| = \left| \int_0^{\pi/2} \sin x \, dx \; - e \right| \tag{1.19}$$

Further comments on accuracy and order of accuracy are provided later in the report.

Chapter 2

Numerical Computation : Python Implementation

This chapter takes us through the codes in Python used to implement the theory provided to us in the preceding chapter or in other words - here, we go through the Python Code for the numerical implementation of the Rectangle Method (Left, Right and Midpoint methods) and the Trapezoid Method.

2.1 Rectangle Method

The following code in Python, is the implementation of the theory and iteration tasks covered in the previous chapters. The code is commented well so further explanation of the code doesn't seem necessary.Results of the code 2.1, 2.2 and 2.3 are presented in the next chapter, in plots 3.1, 3.2 and 3.3 respectively.

```
def LM(f,a,b,n):
    # Carries out the numerical integration using the Left Hand Rule
    # INPUT: f , a , b , n
    # f : function which is the integrand
    # a : lower limit of the interval
    # b : upper limit of the interval
    # n : number of sub-intervals

    # makes sure the input format is correct
    if (b < a) :
        print('Input Error: Lower limit(a) must be < Upper limit(b)')
        exit()
    # limits the maximum number of iterations to 1 million
    if (n > 1000000) :
        print('Input Error: No more than 1 million divisions are allowed!')
        exit()
    # calculate the step size or length of each sub-interval
    h = (b - a)/n
    # calculates the numerical integration using Left Hand Rule
    area = 0
    for i in range(n):
        x = a + float(i*h)
        area = area + f(x)*h
```

```
    # calculates error
    err = 1 - area
    rel_err = np.abs(err)*100
    print('-------------------------------------------------------------')
    print('NUMERICAL QUADRATURE RESULTS : LEFT-ENDPOINT METHOD')
    print('-------------------------------------------------------------')
    print('Integrating from %8.2f    to %8.2f   in %5d steps' %(a,b,n))
    print('Area             = %10.4f units\u00b2'%(area))
    print('Error            = %10.4f'%err)
    print('Absolute Error = %10.4f '%(np.abs(err)))
    print('Relative Error = %10.4f'%rel_err)
    print('-------------------------------------------------------------')
```

Listing 2.1: Function named **'LM'** is constructed to carry out the numerical integration.

```
def RM(f,a,b,n):
    # Carries out the numerical integration using the Right Hand Rule
    # INPUT: f , a , b , n
    # f : function which is the integrand
    # a : lower limit of the interval
    # b : upper limit of the interval
    # n : number of sub-intervals

    # makes sure the input format is correct
    if (b < a) :
        print('Input Error: Lower limit(a) must be < Upper limit(b)')
        exit()
    # limits the maximum number of iterations to 1 million
    if (n > 1000000) :
        print('Input Error: No more than 1 million divisions are allowed!')
        exit()
    # calculate the step size or length of each sub-interval
    h = (b - a)/n
    # calculates the numerical integration using Right Hand Rule
    area = 0
    for i in range(1,n+1):
        x = a + float(i*h)
        area = area + f(x)*h
    # calculates error
    err = 1 - area
    rel_err = np.abs(err)*100
    print('-------------------------------------------------------------')
    print('NUMERICAL QUADRATURE RESULTS : RIGHT-ENDPOINT METHOD')
    print('-------------------------------------------------------------')
    print('Integrating from %8.2f    to %8.2f   in %5d steps' %(a,b,n))
    print('Area             = %10.4f units\u00b2'%(area))
    print('Error            = %10.4f'%err)
    print('Absolute Error = %10.4f '%(np.abs(err)))
    print('Relative Error = %10.4f'%rel_err)
    print('-------------------------------------------------------------')
```

Listing 2.2: Function named **'RM'** is constructed to carry out the numerical integration.

```
def MpM(f,a,b,n):
    # Carries out the numerical integration using the Midpoint Rule
    # INPUT: f , a , b , n
    # f : function which is the integrand
```

```
    # a : lower limit of the interval
    # b : upper limit of the interval
    # n : number of sub-intervals

    # makes sure the input format is correct
    if (b < a) :
        print('Input Error: Lower limit(a) must be < Upper limit(b)')
        exit()
    # limits the maximum number of iterations to 1 million
    if (n > 1000000) :
        print('Input Error: No more than 1 million divisions are allowed!')
        exit()
    # calculate the step size or length of each sub-interval
    h = (b - a)/n
    # calculates the numerical integration using Midpoint Rule
    area = 0
    for i in range(n):
        x = a + float((i+0.5)*h)
        area = area + f(x)*h
    # calculates error
    err = 1 - area
    rel_err = np.abs(err)*100
    print('-------------------------------------------------------------')
    print('NUMERICAL QUADRATURE RESULTS : MIDPOINT METHOD')
    print('-------------------------------------------------------------')
    print('Integrating from %8.2f    to %8.2f    in %5d steps' %(a,b,n))
    print('Area             = %10.4f units\u00b2'%(area))
    print('Error            = %10.4f'%err)
    print('Absolute Error = %10.4f '%(np.abs(err)))
    print('Relative Error = %10.4f'%rel_err)
    print('-------------------------------------------------------------')
```

Listing 2.3: Function named **'MpM'** is constructed to carry out the numerical integration.

2.2 Trapezoidal Method

The task is repeated for Trapezoidal method here. For reference and better inference of the code, parallel viewing of the codes(2.4) and the results(plot 3.4) provided in the next chapter is advised.

```
def TM(f,a,b,n):
    # Carries out the numerical integration using the Trapezoidal Rule
    # INPUT: f , a , b , n
    # f : function which is the integrand
    # a : lower limit of the interval
    # b : upper limit of the interval
    # n : number of sub-intervals

    # makes sure the input format is correct
    if (b < a) :
        print('Input Error: Lower limit(a) must be < Upper limit(b)')
        exit()
    # limits the maximum number of iterations to 1 million
    if (n > 1000000) :
        print('Input Error: No more than 1 million divisions are allowed!')
        exit()
```

```
    # calculate the step size or length of each sub-interval
    h = (b - a)/n
    step.append(h)
    N.append(n)
    # calculates the numerical integration using Trapezoidal Rule
    area = f(float(a))*h/2
    for i in range(1,n):
        x = a + float(i*h)
        area = area + f(x)*h
    area = area + f(float(b))*h/2
    # calculates error
    err = 1 - area
    rel_err = np.abs(err)*100
    abs_err.append(np.abs(err))
    print('-----------------------------------------------------------------')
    print('NUMERICAL QUADRATURE RESULTS : TRAPEZOID METHOD')
    print('-----------------------------------------------------------------')
    print('Integrating from %8.2f    to %8.2f   in %5d steps' %(a,b,n))
    print('Area             = %10.4f units\u00b2'%(area))
    print('Error            = %10.4f'%err)
    print('Absolute Error = %10.4f '%(np.abs(err)))
    print('Relative Error = %10.4f'%rel_err)
    print('-----------------------------------------------------------------')
```

Listing 2.4: Function named **'TM'** is constructed to generate the iteration steps of the Newton's Method

2.3 Linear Regression of data.

We were also required to plot (scatter) the raw data (Absolute Error and Step size/ number of intervals) obtained from the above codes and then use linear regression to plot a linear variation of the data, whose semi-log and log-log plots are also considered. Following code help carry out linear regression and plot them.

```
def inv(A):
    #Augment with an identity matrix of required order: Here it's 2 :
    if A.shape(0)!= A.shape(1):
       sys.exit('Not Invertible!')
    n = A.shape(0)
    a = np.concatenate((A,np.identity(n,dtype = float)),axis = 1)
    # Applying Gauss Jordan Elimination:
    for i in range(n):
        if a[i][i] == 0.0:
           sys.exit('Not Invertible!')
        for j in range(n):
            if i != j:
               ratio = a[j][i]/a[i][i]
               for k in range(2*n):
                   a[j][k] = a[j][k] - ratio*a[i][k]
    #Row operation to make principal diagonal element to 1:
    for i in range(n):
        divisor = a[i][i]
        for j in range(2*n):
            a[i][j] = a[i][j]/divisor
    #Displaying the Inverted matrix:
```

```
    #print('\nINVERTED MATRIX IS:')
    #for i in range(n):
    #    for j in range(n,2*n):
    #        print(a[i][j],end ='\t')
    #    print()
    #Returning the inverted matrix
    b = np.zeros([n,n])
    for i in range(n):
        for j in range(n,2*n):
            b[i-n][j-n] = a[i][j]
    return b

def linreg(step,abs_err,name):
    data_x = (np.array(step).reshape(len(step),1))
    data_y = (np.array(abs_err).reshape(len(step),1))
    #Adding a column of all ones (intercept term) to data_x:
    data_X = np.concatenate((np.ones([len(data_x),1]),data_x),axis = 1)
    #We now apply the Normal Equation:
    X_transpose =np.transpose(data_X)
    theta =
        np.linalg.inv(X_transpose.dot(data_X)).dot(X_transpose).dot(data_y)
    #Plotting the linear regression:
    plt.plot(data_X[:,1],data_X.dot(theta),'-',label=name)
    #Plotting raw data to compare:
    plt.scatter(data_x,data_y,s=8,marker = 'x')
    plt.xlabel('X')
    plt.ylabel('Y')
    plt.autoscale()
    #plt.show()
    #return theta[0],theta[1]
```

Listing 2.5: **inv() and linreg()**

2.4 Tic-Toc Generator

We have defined functions such that they enable us to measure the time required for the execution for various methods discussed .Listing 2.6 contains the code for the generator. To understand the usage of the functions, please refer the next chapter for the results.

```
import time
def TicTocGenerator():
  # Function that returns time differences
  ti = 0           # initial time
  tf = time.time() # final time
  while True:
    ti = tf
    tf = time.time()
    yield (tf-ti)   #Returns the time difference

TicToc = TicTocGenerator()  #create an instance of the TicTocGenerator

# Main function through which we define both tic() and toc()
def toc(tempBool = True):
  # Prints the time difference yielded by instance of
    TicTocGenerator()-TicToc
```

```
    tempTimeInterval = next(TicToc)
    if tempBool:
      print("Elapsed time: %f seconds.\n" %tempTimeInterval)

def tic():
    # Records a time in TicToc, marks the beginning of a time interval
    toc(False)
```

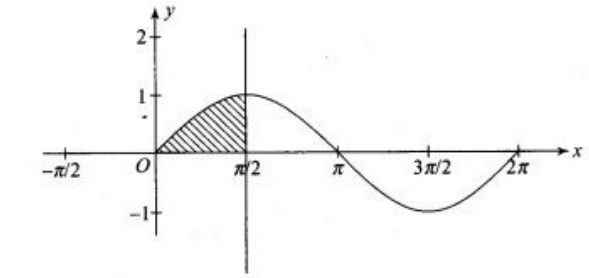
Listing 2.6: **tic() and toc()**

2.5 Task

For this task, we consider the definite integral

$$I = \int_0^{\pi/2} \sin x \, dx \tag{2.1}$$

which has the exact value of 1. Refer 2.1. We plot the absolute error versus length of



From the figure, area of the shaded region

$$A = \int_0^{\pi/2} \sin x dx$$

$$= [-\cos x]_0^{\pi/2} = -\cos\frac{\pi}{2} + \cos 0 = 0 + 1 = 1 \text{ sq. units}$$

Figure 2.1: Exact Value of the definite integral.

a subinterval (Δx) for each of the four methods discussed earlier in this chapter. We also try plotting the semi-log and log-log plot of the same to see if the plots give us a better insight into the nature of differences between the methods. We then apply linear regression to each of the plots.
We then comment on the order of accuracy, accuracy of each method and then present the concluding remark of the report.

Chapter 3

Results

This chapter takes us through the results obtained by executing the various codes discussed earlier for the 4 methods. Results include the plots for $|E|$ vs N, $log(|E|)$ vs h, $log(|E|)$ vs $log(h)$, $|E|$ vs h and comparison of the execution time for each root finding algorithm.
Note: The execution times stated here are majorly dependent on the way I have implemented the maths into the python code. However, it gives a brief sense of the relative efficiencies of the methods.

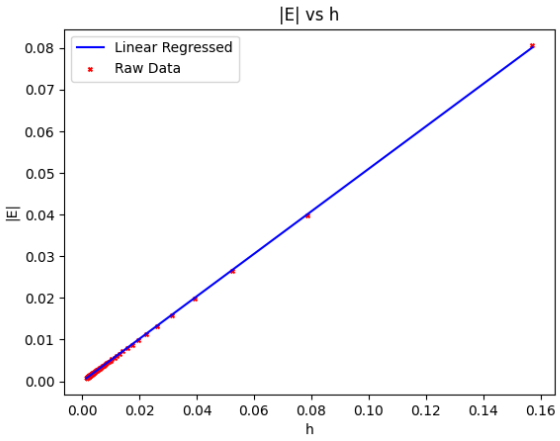
3.1 Plots

Here, we present the plots obtained from codes for various methods for Numerical Quadrature. Figure 3.1, 3.2 , 3.3 3.4 are the plots for the Left , Right , Midpoint and Trapezoidal Methods, implemented for equation 2.1, obtained by varying the number of subintervals from 10 to 1000, using a step size of 10.
Figures 3.5, 3.6 and 3.7 show the $|E|$ vs h, $log(|E|)$ vs h and $log(|E|)$ vs $log(h)$ for all the methods, each in one plot- : to ease visual distinction. Also, linear regression is performed in each set raw data obtained.
In figures 3.6 and 3.7, the plots for Left(Blue) and Right(Orange) Endpoint methods overlap. For better scaled up plots, please visit github.

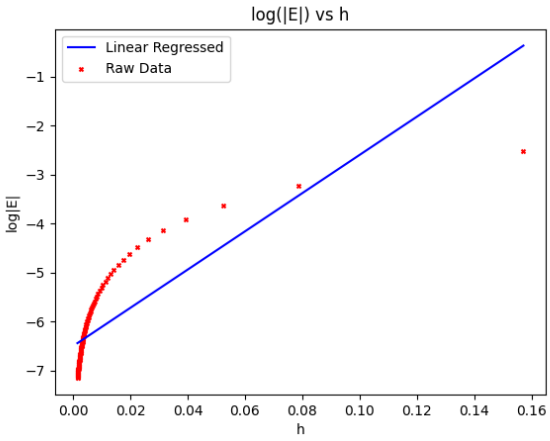
3.2 Order of Accuracy

The order of accuracy of the various methods are discussed here. The term 'Order of Accuracy' was defined earlier in Chapter -1.4.
We mention the order of accuracy of the methods here as per the definition provided earlier.
Trapezoid Method:
The order of accuracy is 2 as

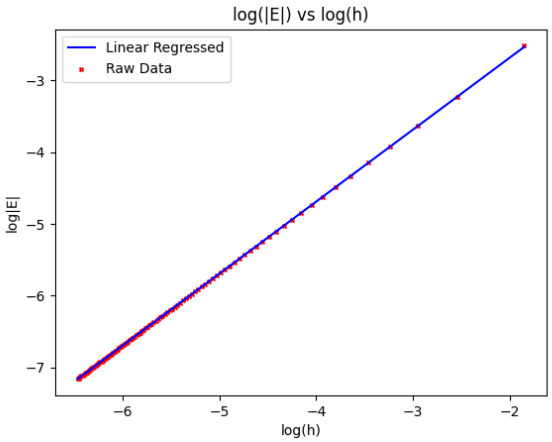
$$\left| \int_a^b f(x)\,dx \ - A_{trap} \right| \leq \frac{M_2(b-a)^3}{12n^2} \tag{3.1}$$



(a) $|E|$ vs h



(b) $log(|E|)$ vs h



(c) $log(|E|)$ vs $log(h)$

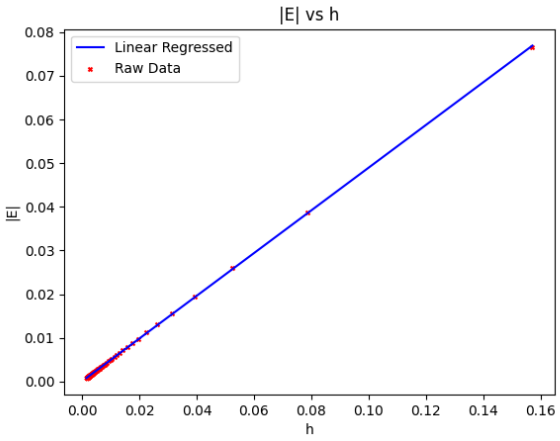
Figure 3.1: Plots for **Left-Endpoint Method**.

Midpoint Method:
The order of accuracy is 2 as

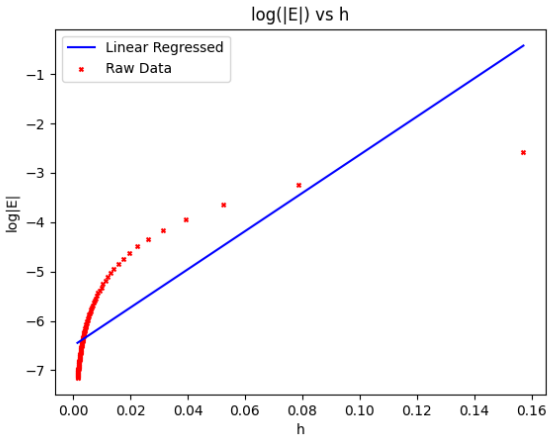
$$\left| \int_a^b f(x)\,dx \ - M_n \right| \leq \frac{M_2(b-a)^3}{24n^2} \tag{3.2}$$

Right and Left Endpoint Method:
The order of accuracy is 1 as

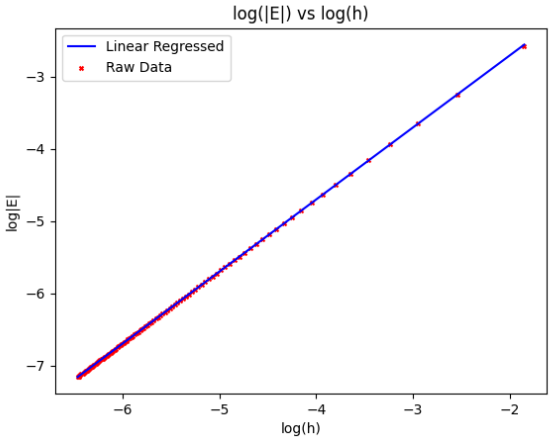
$$\left| \int_a^b f(x)\,dx \ - R_n \right| \leq \frac{M_1(b-a)^2}{2n} \tag{3.3}$$



(a) $|E|$ vs h

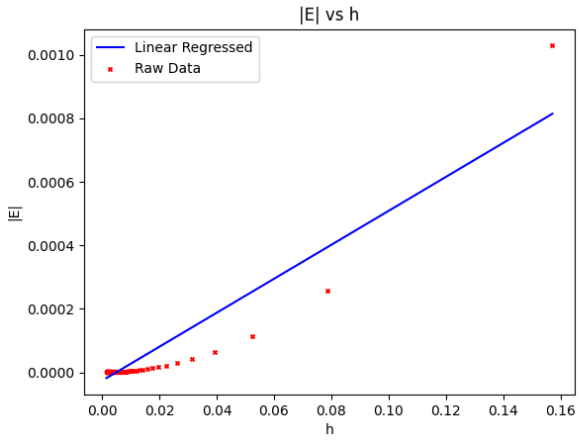


(b) $log(|E|)$ vs h

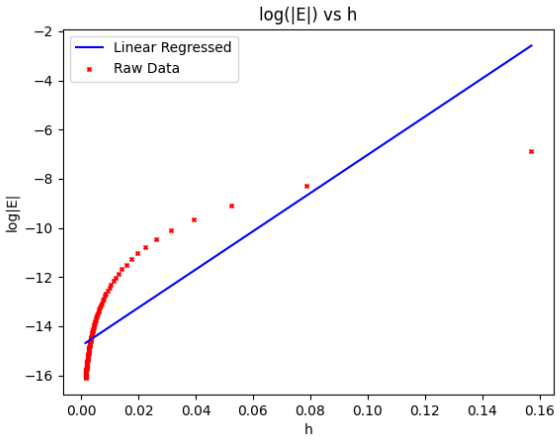


(c) $log(|E|)$ vs $log(h)$

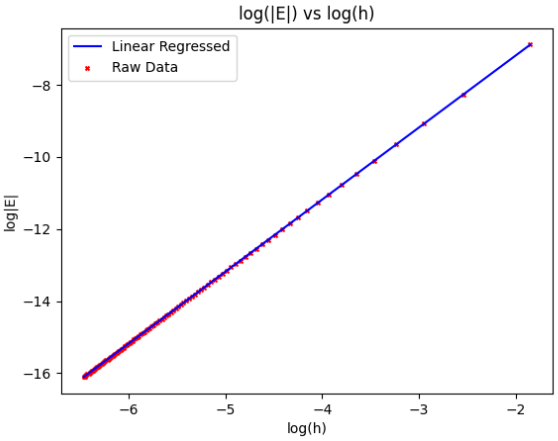
Figure 3.2: Plots for **Right-Endpoint Method**.



(a) $|E|$ vs h

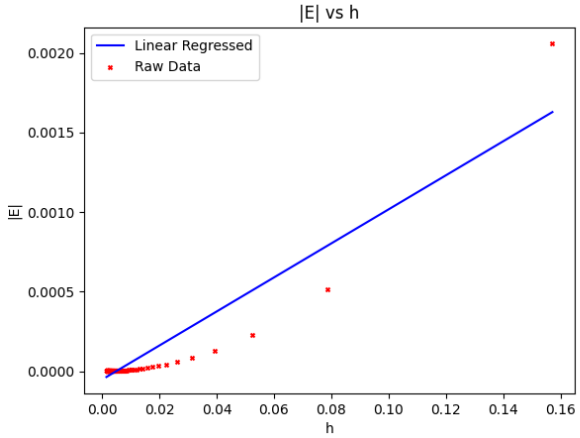


(b) $log(|E|)$ vs h

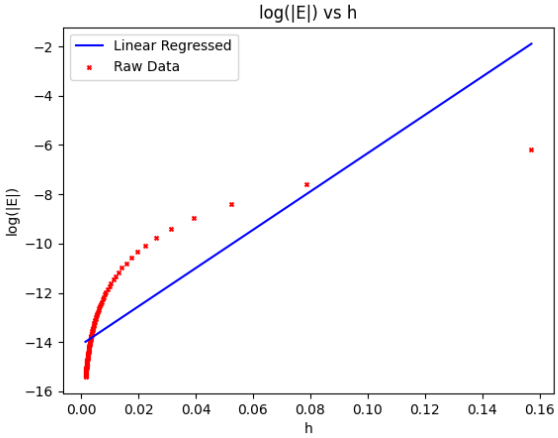


(c) $log(|E|)$ vs $log(h)$

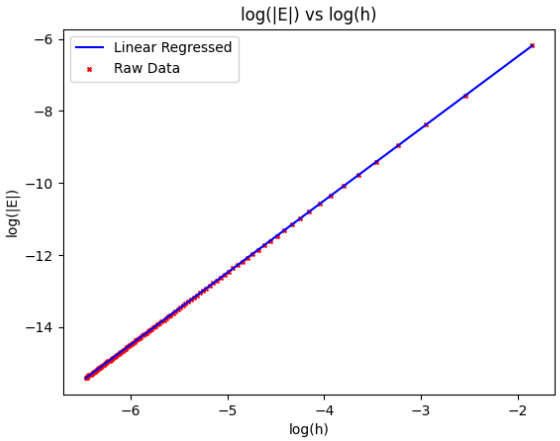
Figure 3.3: Plots for **Midpoint Method**.



(a) $|E|$ vs h



(b) $log(|E|)$ vs h



(c) $log(|E|)$ vs $log(h)$

Figure 3.4: Plots for **Trapezoid Method**.

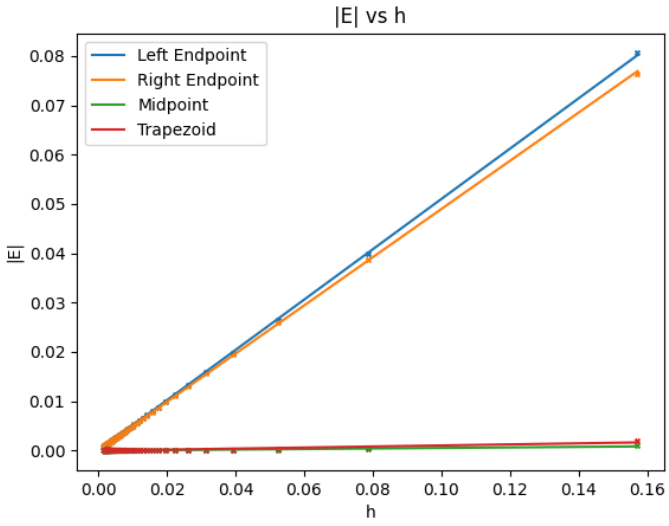


Figure 3.5: $|E|$ vs h for all methods.

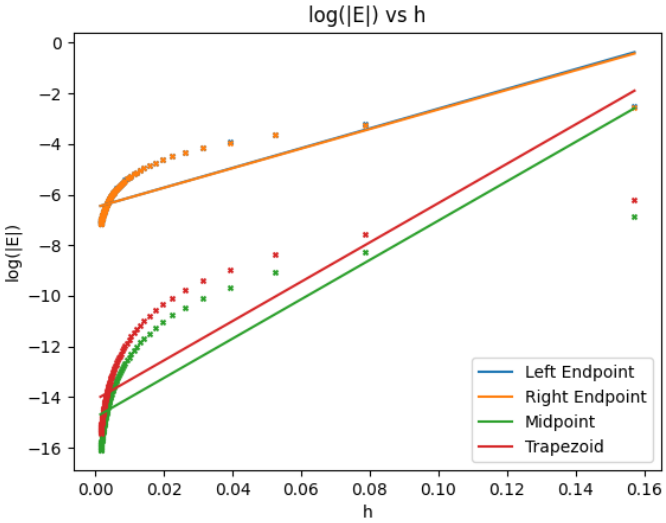


Figure 3.6: $log(|E|)$ vs h for all methods.

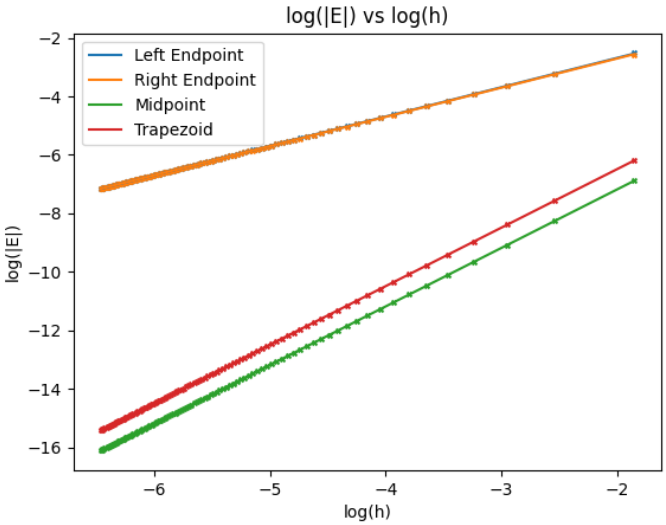


Figure 3.7: $log(|E|)$ vs $log(h)$ for all methods.

Chapter 4

Conclusion

In this report, we discussed and presented the various methods for Numerical Quadrature, explain the numerical implementation in Python and then apply the results for a simple definite integral of $sin(x)$ in $[0, \frac{\pi}{2}]$.
We used Linear Regression to produce a linear plot using the raw data generated using the codes presented in Chapter-2.
We also discussed the order of accuracy for each of the methods. Following plot, figure 4.2 shows the plot for $|E| vs N$ (Absolute Error versus No of sub-intervals). The plot (linear regressed) for Trapezoid(red) and Midpoint(green), and Left(blue) and Right(orange) Methods overlap.

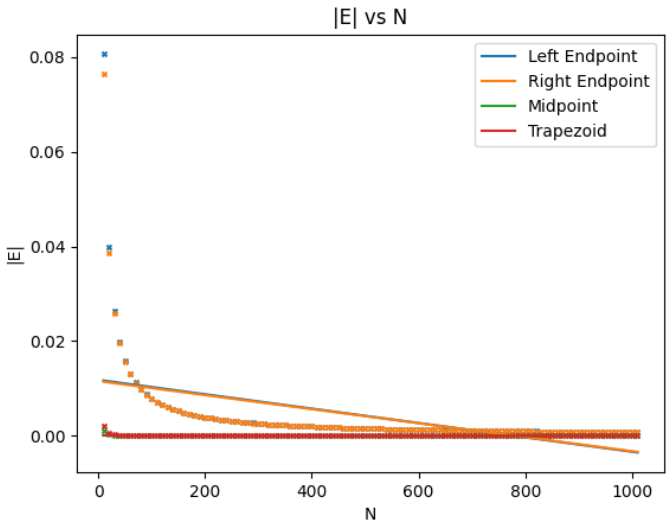


Figure 4.1: $|E|$ vs N for all methods - 100 data pints.

For a better scaled up figure, please visit github.
To comment on the accuracy of the 4 methods, consider figure **??** where the same process is repeated for 10 data points, starting from 10 sub-intervals to 100 sub-intervals. Accuracy is high when $|E|$ is low. From this figure, it's apparent that the accuracy for low number of sub-intervals is higher in Midpoint and Trapezoid Methods when compared to Left and Right Endpoint Methods. However, there exists an N such that past that number of

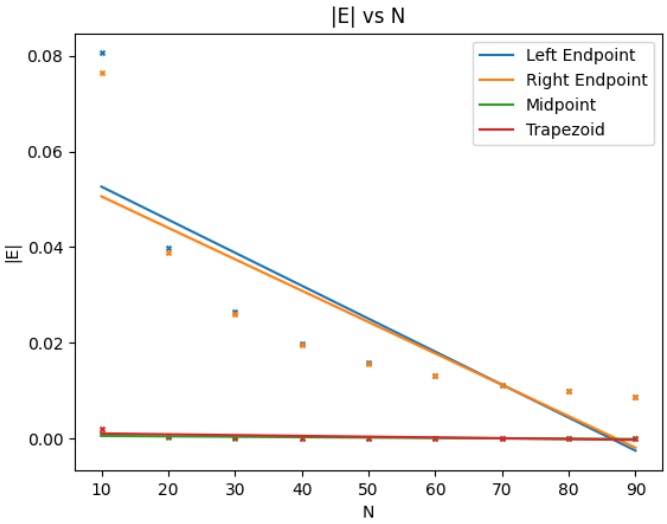


Figure 4.2: $|E|$ vs N for all methods - 10 data points.

sub-intervals the accuracy of Left and Right Endpoint Methods are higher!(4.2)
For the detailed codes in Python, please visit github.