EE1103: Numerical Methods

Programming Assignment # 3

Shreya S Ramanujam, EE21B126
Collaborators:
Srikar Babu Gadipudi, EE21B138

March 2, 2022

Contents

List of Figures

List of Tables

1 Problem 1

Integrate the sine wave from $[0, \pi]$ using Midpoint, Trapezoidal, and Simpson's methods. Evaluate the integral for different n (the number of subintervals) and Tabulate the absolute error for different experiments and compare the efficiency of the methods. Plot the relevant graphs to check the behaviour of the absolute error for each method as a function of n.

$$f(x) = \sin(x) \tag{1}$$

Find

$$\int_0^{\pi} f(x)dx. \tag{2}$$

1.1 Approach

We are using a `for` loop to loop over each subinterval, and calculating the corresponding Riemann sums using Midpoint rule 1, Trapezoidal rule 2 and Simpson's rule 3.

1.2 Algorithm

Algorithm 1: Midpoint Rule

Declare n (number of partitions)
Define $\texttt{len} = \frac{\pi}{n}$
Declare integral $= 0$
for $i = 0$ *to* $i < n$ **do**
  integral$+ = \texttt{len} \times \sin(\frac{2i+1}{2})$
end
E_a (absolute error) $= |\text{integral} - 2|$

Algorithm 2: Trapezoidal Rule

Declare n (number of partitions)
Define $\texttt{len} = \frac{\pi}{n}$
Declare integral $= 0$
for $i = 0$ *to* $i < n$ **do**
  integral$+ = \texttt{len} \times \frac{\sin(i*\texttt{len})+\sin((i+1)*\texttt{len})}{2}$
end
E_a (absolute error) $= |\text{integral} - 2|$

Algorithm 3: Simpson's Rule

Declare n (number of partitions)
Define $\texttt{len} = \frac{\pi}{n}$
Declare integral $= \frac{\texttt{len}}{3} \times \sin(\pi)$
for $i = 0$ *to* $i < n$ **do**
  if i is odd then
  integral$+ = 4 * \frac{\texttt{len}}{3} \times \sin(i * \texttt{len})$
  else
  integral$+ = 2 * \frac{\texttt{len}}{3} \times \sin(i * \texttt{len})$
end
E_a (absolute error) $= |\text{integral} - 2|$

1.3 Results

Simpson's rule is the most efficient method to get close to the actual value of integral in lesser number of subintervals.

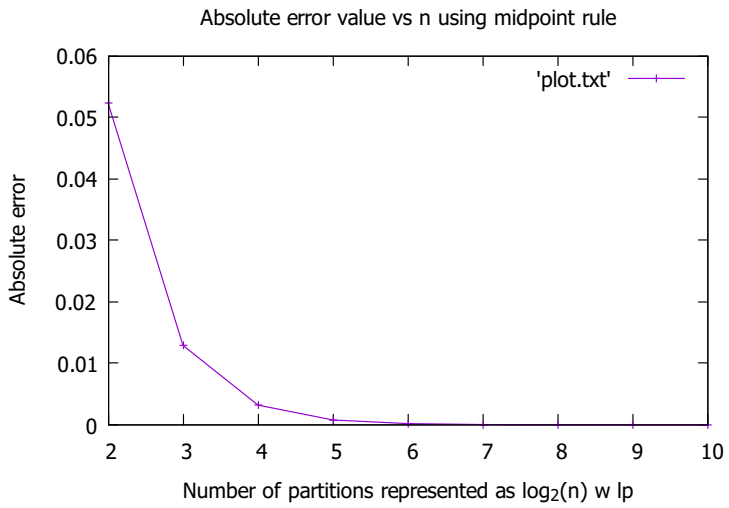

Figure 1: Absolute error vs $log_2 n$ using Midpoint rule

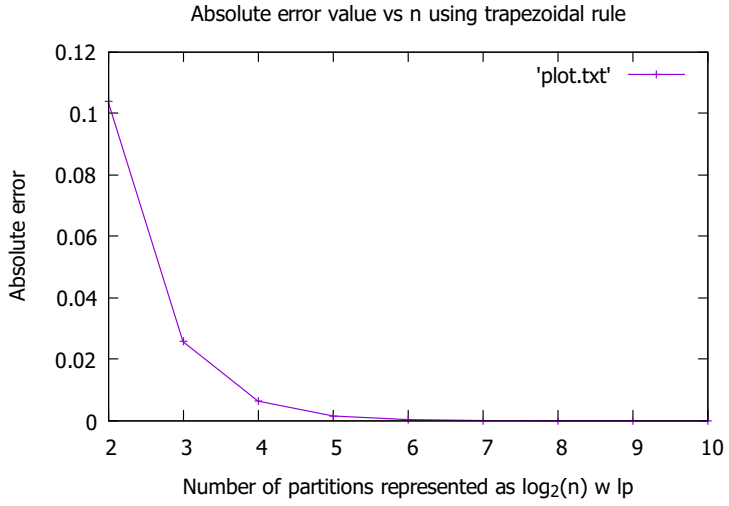

Figure 2: Absolute error vs $log_2 n$ using Trapezoidal rule

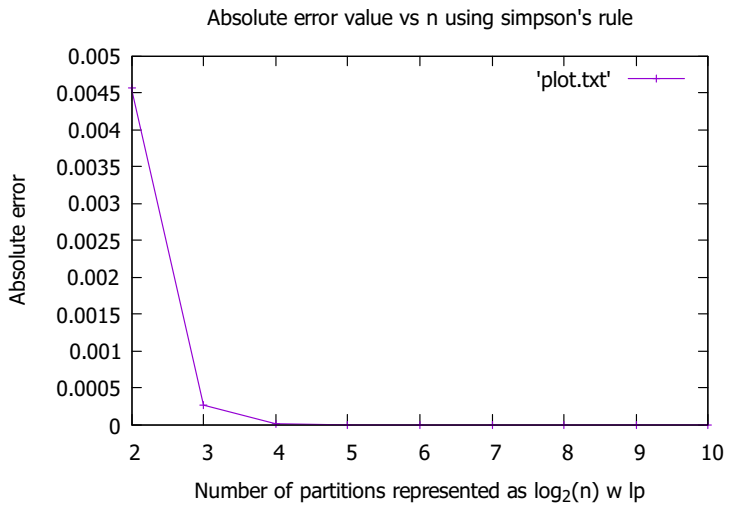

Figure 3: Absolute error vs $log_2 n$ using Simpson's rule

Table 1: Result using Midpoint rule

n	**Value of integral**	**Absolute error**
4	2.052344	0.052344
16	2.003217	0.003217
64	2.000201	0.000201
256	2.000012	0.000012
1024	2.000000	0.000000
2048	2.000001	0.000001

Table 2: Result using Trapezoidal rule

n	**Value of integral**	**Absolute error**
4	1.896119	0.103881
16	1.993570	0.006430
64	1.999598	0.000402
256	1.999974	0.000026
1024	1.999998	0.000002
2048	2.000000	0.000000

Table 3: Result using Simpson's rule

n	**Value of integral**	**Absolute error**
4	2.004560	0.004560
16	2.000017	0.000017
64	2.000000	0.000000
256	2.000000	0.000000
1024	1.999999	0.000001
2048	2.000000	0.000000

1.4 Inferences

- The midpoint rule approaches the true root from above, meaning the value of integral is always greater than 2 for all values of n, Refer Table 1.
- Similarly the trapezoidal rule approaches the true root from below, as the function is always concave down in our interval of observation, this rule underestimates the value of integral. Refer Table 2.
- Simpson's rule and Midpoint rule approaches the true value in lesser number of subintervals as compared to Trapezoidal rule, but later on they deviate and might converge again. As the subintervals become smaller and smaller, the functions tends to behave as a straight line, this causes the Simpson's rule to fail, and in case of Midpoint rule, the function might become very vertical in some subintervals, in which taking the midpoint is not a good approximation of its Riemann sum. Refer Table 2 and 3.

1.5 Code

- Midpoint rule 1
- Trapezoidal rule 2
- Simpson's rule 3

Listing 1: Code snippet used in the experiment for midpoint rule.

```
#include <stdio.h>
#include <math.h>

int main()
{
    FILE *fp=fopen("midpoint.txt", "a");
    int n;
    scanf("%i", &n);
    float len=M_PI/n;//length of subinterval
    float integmid=0;
    for(int i=0; i<n; i++){
        float mid=len*(2*i+1)/2;//midpoint rule
        integmid+=len*sin(mid);
    }
    float errabsm=fabs(integmid-2.0);//absolute error
```

```
    fprintf(fp, "%i %f %f\n", n, integmid, errabsm);
    fclose(fp);
}
```

Listing 2: Code snippet used in the experiment for trapezoidal rule.

```
#include <stdio.h>
#include <math.h>

int main()
{
    FILE *fp=fopen("trapezoidal.txt", "a");
    int n;
    scanf("%i", &n);
    float len=M_PI/n;//length of subinterval
    float integtrap=0;
    for(int i=0; i<n; i++){
        float midf=(sin(i*len)+sin((i+1)*len))/2;
        integtrap+=len*midf;//trapezoidal rule
    }
    float errabst=fabs(integtrap-2.0);//absolute error
    fprintf(fp, "%i %f %f\n", n, integtrap, errabst);
    fclose(fp);
}
```

Listing 3: Code snippet used in the experiment for simpson's rule.

```
#include <stdio.h>
#include <math.h>

int main()
{
    FILE *fp=fopen("simpson.txt", "a");
    int n;
    scanf("%i", &n);
    float len=M_PI/n;//length of subinterval
    float integsimp=(sin(0)+sin(M_PI))*len/3;
    for(int i=0; i<n; i++){
        if(i%2!=0){//simpson's rule
            integsimp+=4*(sin(i*len))*len/3;
        }
        else{
            integsimp+=2*(sin(i*len))*len/3;
        }
    }
    float errabss=fabs(integsimp-2.0);//absolute error
    fprintf(fp, "%i %f %f\n", n, integsimp, errabss);
    fclose(fp);
}
```

1.6 Contributions

Algorithm and code were done by Srikar, which was later discussed.

2 Problem 2

Integrate the standard Gaussian pdf to estimate Erf(1) and Erf(2) using Midpoint, Trapezoidal, and Simpson's rules.
Note: Assume the Gaussian PDF has 0 value outside the range $[-4, 4]$.

a) Tabulate the absolute error for different experiments and compare the efficiency of the methods.

b) Plot the absolute error vs n and explain if there is any anomalous behaviour. Is neglecting the region outside $[-4, 4]$ a good choice for calculating the integral with 0.1% accuracy?

c) Compare the values of Erf(1) and Erf(2) obtained by integration to those obtained using the empirical distribution in the previous assignment.

$$Erf(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{\frac{-t^2}{2}} dt \tag{3}$$

2.1 Approach

In all 3 approaches used to solve this problem, we use a nested `for` loop.
The outer loop iterates over the number of subintervals we are considering and the inner loop calculates the contribution of each subinterval to the Riemann Sum and adds all of them up to finally give us the approximate value of the integral.

2.2 Algorithm

The pseudocodes used are provided in Algorithms 4, 5 and 6 for Midpoint, Trapezoidal, and Simpson's rules respectively.

Algorithm 4: Approximating $Erf(1)$ using Midpoint Rule

upper limit $\leftarrow 1$
for *no of subintervals* $\leftarrow 4, 8, 16 \ldots 1024$ **do**
  subinterval length $\leftarrow \frac{4+\text{upper limit}}{\text{no of subintervals}}$
  sum$\leftarrow 0$;
  for $i \leftarrow 1$ *to* n **do**
    $x_{midpoint} \leftarrow (\frac{2i-1}{2})$(subinterval length) - 4
    $f(x_{midpoint}) \leftarrow \frac{1}{\sqrt{2\pi}} e^{\frac{-x_{midpoint}^2}{2}}$
    $sum \leftarrow sum +$ subinterval length $\times f(x_{midpoint})$
  end
  absolute error $\leftarrow |Erf(1) - sum|$
end

Algorithm 5: Approximating $Erf(1)$ using Trapezoidal Rule

upper limit $\leftarrow 1$
$f(x_0) \leftarrow \frac{1}{\sqrt{2\pi}} e^{\frac{-x_0^2}{2}}$
$f(x_n) \leftarrow \frac{1}{\sqrt{2\pi}} e^{\frac{-x_n^2}{2}}$
for *no of subintervals* $\leftarrow 4, 8, 16 \ldots 1024$ **do**
  subinterval length $\leftarrow \frac{4+\text{upper limit}}{\text{no of subintervals}}$
  sum$\leftarrow \frac{\text{subinterval length}}{2}(f(x_0) + f(x_n))$;
  for $i \leftarrow 1$ *to* $n-1$ **do**
    $x_{i^{th}division} \leftarrow i \times$ subinterval length $- 4$
    $f(x_{i^{th}division}) \leftarrow \frac{1}{\sqrt{2\pi}} e^{\frac{-x_{i^{th}division}^2}{2}}$
    $sum \leftarrow sum +$ subinterval length $\times f(x_{i^{th}division})$
  end
  absolute error $\leftarrow |Erf(1) - sum|$
end

Algorithm 6: Approximating $Erf(2)$ using Simpson's Rule

upper limit $\leftarrow 2$
$f(x_0) \leftarrow \frac{1}{\sqrt{2\pi}} e^{\frac{-x_0^2}{2}}$
$f(x_n) \leftarrow \frac{1}{\sqrt{2\pi}} e^{\frac{-x_n^2}{2}}$
for *no of subintervals* $\leftarrow 4, 8, 16 \ldots 1024$ **do**
  subinterval length $\leftarrow \frac{4+\text{upper limit}}{\text{no of subintervals}}$
  sum$\leftarrow \frac{\text{subinterval length}}{3}(f(x_0) + f(x_n))$;
  for $i \leftarrow 1$ *to* $n-1$ **do**
    $x_{i^{th}division} \leftarrow i \times$ subinterval length $- 4$
    $f(x_{i^{th}division}) \leftarrow \frac{1}{\sqrt{2\pi}} e^{\frac{-x_{i^{th}division}^2}{2}}$
    if (i is even) **then**
    k $\leftarrow \frac{2}{3}$
    else
    k $\leftarrow \frac{4}{3}$
    end
    $sum \leftarrow sum + k \times$ subinterval length $\times f(x_{i^{th}division})$
  end
  absolute error $\leftarrow |Erf(2) - sum|$
end

2.3 Results

We plot the graphs showing the absolute error as a function of the number of subdivisions considered while approximating $Erf(x)$ in Figures 4, 5 and 6 for Erf(1) and 7, 8 and 9 for Erf(2). The results are also summarized in Tables 4 and 5.

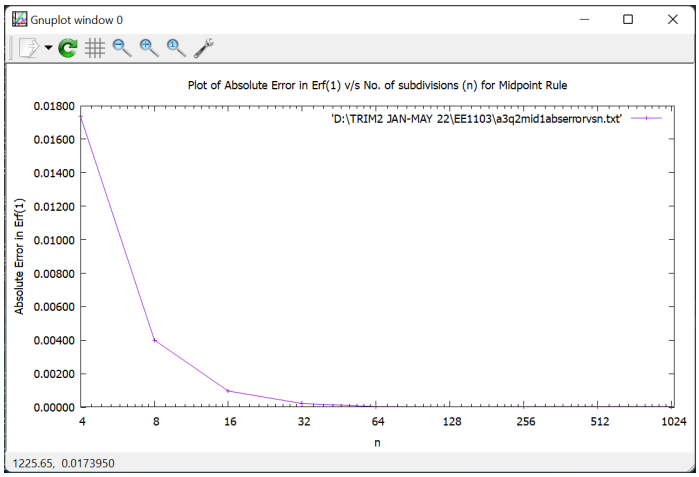

Figure 4: Absolute error vs number of subintervals considered while approximating $Erf(1)$ using Midpoint Rule

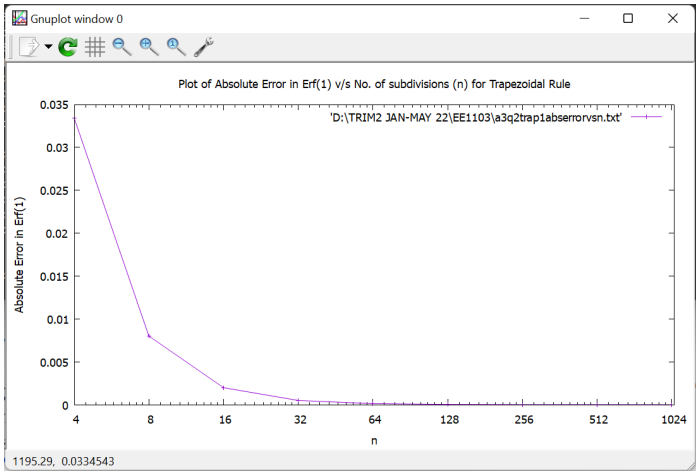

Figure 5: Absolute error vs number of subintervals considered while approximating $Erf(1)$ using Trapezoidal Rule

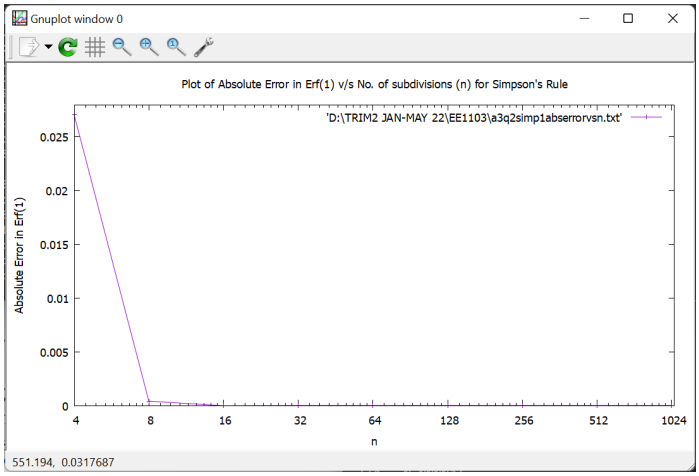

Figure 6: Absolute error vs number of subintervals considered while approximating $Erf(1)$ using Simpson's Rule

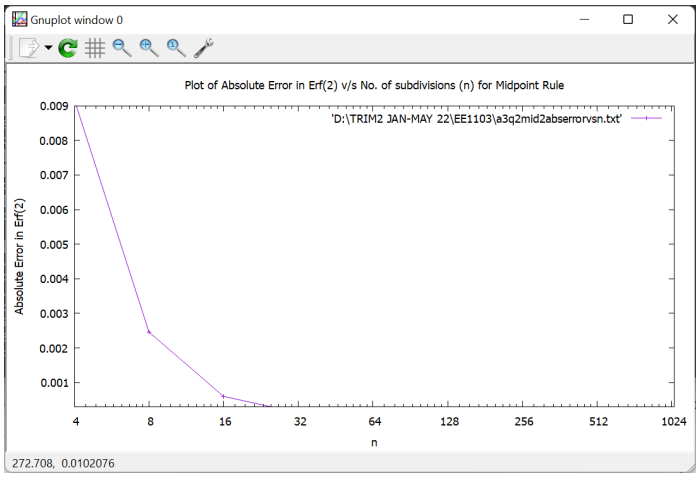

Figure 7: Absolute error vs number of subintervals considered while approximating $Erf(2)$ using Midpoint Rule

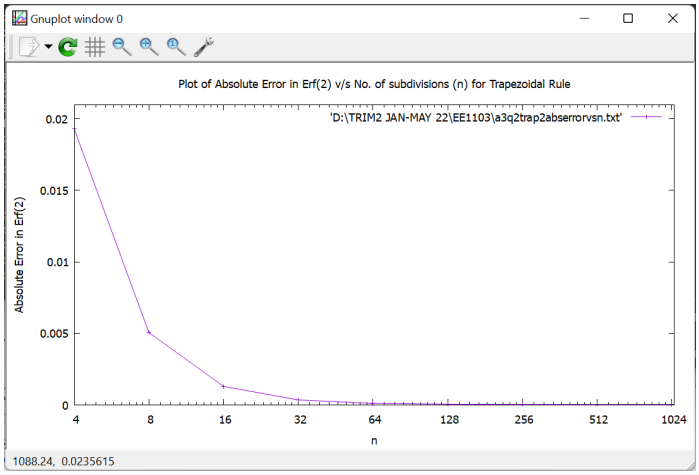

Figure 8: Absolute error vs number of subintervals considered while approximating $Erf(2)$ using Trapezoidal Rule

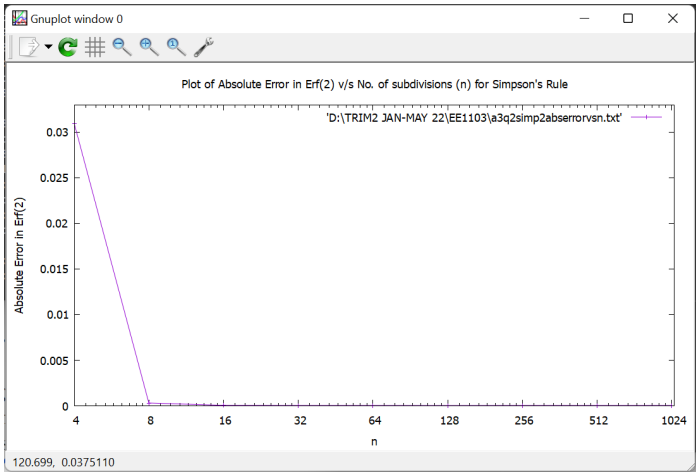

Figure 9: Absolute error vs number of subintervals considered while approximating $Erf(2)$ using Simpson's Rule

Table 4: No of subintervals (n) v/s absolute error for Erf(1)

n	**Midpoint**	**Trapezoidal**	**Simpson's**
4	0.017370	0.033430	0.027066
8	0.004007	0.008030	0.000437
16	0.000961	0.002012	0.000005
32	0.000215	0.000525	0.000030
64	0.000030	0.000155	0.000032
128	0.000016	0.000062	0.000032
256	0.000028	0.000039	0.000031
512	0.000031	0.000034	0.000032
1024	0.000031	0.000032	0.000032

Table 5: No of subintervals (n) v/s absolute error for Erf(2)

n	**Midpoint**	**Trapezoidal**	**Simpson's**
4	0.009185	0.019310	0.030971
8	0.002462	0.005062	0.000313
16	0.000601	0.001300	0.000046
32	0.000127	0.000349	0.000032
64	0.000008	0.000111	0.000032
128	0.000022	0.000052	0.000031
256	0.000029	0.000036	0.000032
512	0.000031	0.000033	0.000032
1024	0.000032	0.000032	0.000032

2.4 Inferences

We deduce the following inferences in this experiment:

- Simpson's Method gets closest to the result in the least no. of subdivisions.
- The best approximation we are able to get from our programs is 0.841339 for 16 subdivisions with an absolute error of 0.000005 for Erf(1) using Simpson's Rule.
- However, the best approximation of Erf(2) is 0.977258 for 64 subdivisions with an absolute error of 0.000008 using Midpoint Rule.
- As seen in Tables 4 and 5 , after a certain n, both Midpoint and Simpson's Rules start to diverge and give increasing absolute error in their approximations.
- This is because as we decrease the subinterval lengths, the function may start to approach a straight line-like graph in some subintervals.
- This is problematic, because in Midpoint rule, the subinterval function approaching a straight line means that there will be a considerable error in the value of the function at the midpoint (which is what we use in out Riemann approximation) and the actual value of the area under the curve.

- This also causes a problem in Simpson's Rule, which uses quadratic approximation, because a quadratic cannot approximate the given function well if (in a particular subinterval) it approaches a straight line graph.
- There is also always a possibility of roundoff errors in our approximations since we are using ($float$) datatype in all our calculations.
- As per our calculations, our best approximation has a relative error of 0.000645% for 16 subdivisions for Erf(1) using Simpson's Rule, which is much lesser than 0.1% .
- Our best approximation for Erf(2) has an absolute error of 0.000819%, which is also much lesser than 0.1%.
- Hence, neglecting the region outside $[-4, 4]$ was a good choice, since it gives us a pretty good approximation for the integral.
- Our previous values for Erf(1) and Erf(2) were 0.836200 and 0.978800 respectively.
- Our previous value of Erf(1) is lesser than our current best approximation value of Erf(1) and our previous value of Erf(2) is greater than our current best approximaton. However our previous values are subject to changes based on the seed used for the rand() function in our previous program.

2.5 Code

The code used for the experiments is mentioned in Listings 4 (Midpoint Rule), 5 (Trapezoidal Rule) and 6 (Simpson's Rule).

Listing 4: Midpoint Rule

```
//midpoint rule Q2
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

int main (void)
{
    FILE *fptr;

    fptr = fopen("a3q2mid1abserrorvsn.txt","w");

    if(fptr == NULL)
   {
      printf("Error");
      exit(1);
   }

    float x = 1.0;//upper limit
    float mid, fmid, abserror, mn = 0;

    for(int n = 4; n <= 1024; n = n * 2)//n is no of subintervals
    {

```

```
        float subl = (4.0 + x)/n;//length of each subinterval
        mn = 0;

        //i = subinterval number
        for(int i = 1; i <= n; i++)
        {
            mid = ((2*i - 1)*(x + 4))/(2*n) - 4;
            fmid = exp(-0.5 * mid * mid)/(pow(2 * M_PI, 0.5));
            mn = mn + subl * fmid;
        }

        float abserror = fabs((erf(x/pow(2,0.5)) + 1)/2.0 - mn);
        fprintf(fptr,"%d\t%f\n", n, abserror);
        printf("Approximate value of Erf(%f) = %f for %d
        ↪  subdivisions\nAbsolute error = %f\n", x, mn, n, abserror);
    }
    fclose(fptr);
    return(0);
}

```

Listing 5: Trapezoidal Rule

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

float func(float x);

int main (void)
{
    FILE *fptr;

    fptr = fopen("a3q2trap1abserrorvsn.txt","w");

    if(fptr == NULL)
   {
      printf("Error");
      exit(1);
   }


    float x = 1;//upper limit
    float fx0 = func(-4.0);
    float fxn = func(x);
    float fxi, xi;

    for(int n = 4; n <= 1024; n = n * 2)//n is no of subintervals
    {
        float subdivisionLength = (4.0 + x)/n;//length of each subinterval
```

```
        float tn = subdivisionLength *0.5 *(fx0 + fxn);

        //i = subinterval number
        for(int i = 1; i < n; i++)
        {
            xi = i * subdivisionLength - 4;
            fxi = func(xi);
            tn = tn + subdivisionLength * fxi;
        }

        float abserror = fabs((erf(x/pow(2,0.5)) + 1)/2.0 - tn);
        fprintf(fptr, "%d\t%f\n", n, abserror);
        printf("Approximate value of Erf(%f) = %f for %d
        ↪  subdivisions\nAbsolute error = %f\n", x, tn, n, abserror);
    }

    fclose(fptr);
    return(0);
}

float func(float x)
{
    float ans = exp(-0.5 * x * x)/(pow(2 * M_PI, 0.5));
    return(ans);
}
```

Listing 6: Simpson's Rule

```
//simpson Q2

#include <stdio.h>
#include <math.h>
#include <stdlib.h>

float func(float x);

int main (void)
{
    FILE *fptr;

    fptr = fopen("a3q2simp2abserrorvsn.txt","w");

    if(fptr == NULL)
   {
      printf("Error");
      exit(1);
   }

    float x = 2;//upper limit
    float fx0 = func(-4.0);
```

```
    float fxn = func(x);
    float fxi, xi, k;

    for(int n = 4; n <= 1024; n = n * 2)//n is no of subintervals
   {
       float subdivisionLength = (4.0 + x)/n;//length of each subinterval
       float sn = (subdivisionLength *(fx0 + fxn))/3.0;

        //i = subinterval number
        for(int i = 1; i < n; i++)
        {
            xi = i * subdivisionLength - 4;
            fxi = func(xi);
            k = (i%2==0?(2.0/3.0):(4.0/3.0));
            sn = sn + subdivisionLength * fxi * k;
        }

        float abserror = fabs((erf(x/pow(2,0.5)) + 1)/2.0 - sn);
        fprintf(fptr, "%d\t%f\n", n, abserror);
        printf("Approximate value of Erf(%f) = %f for %d
        ↪  subdivisions\nAbsolute error = %f\n", x, sn, n, abserror);
    }

    fclose(fptr);
    return(0);
}

float func(float x)
{
    float ans = exp(-0.5 * x * x)/(pow(2 * M_PI, 0.5));
    return(ans);
}
```

2.6 Contributions

Worked out by me, with inputs from Srikar.