1.

The code written to implement each of the algorithms can be found in the source code `prob1.cpp` in the appendix. Figures 1-3 below show the log-log plots of the relative error ϵ_r of each method as a function of the step size h. Plots and their data were generated by running `prob1.sh`, which can also be found in the appendix.

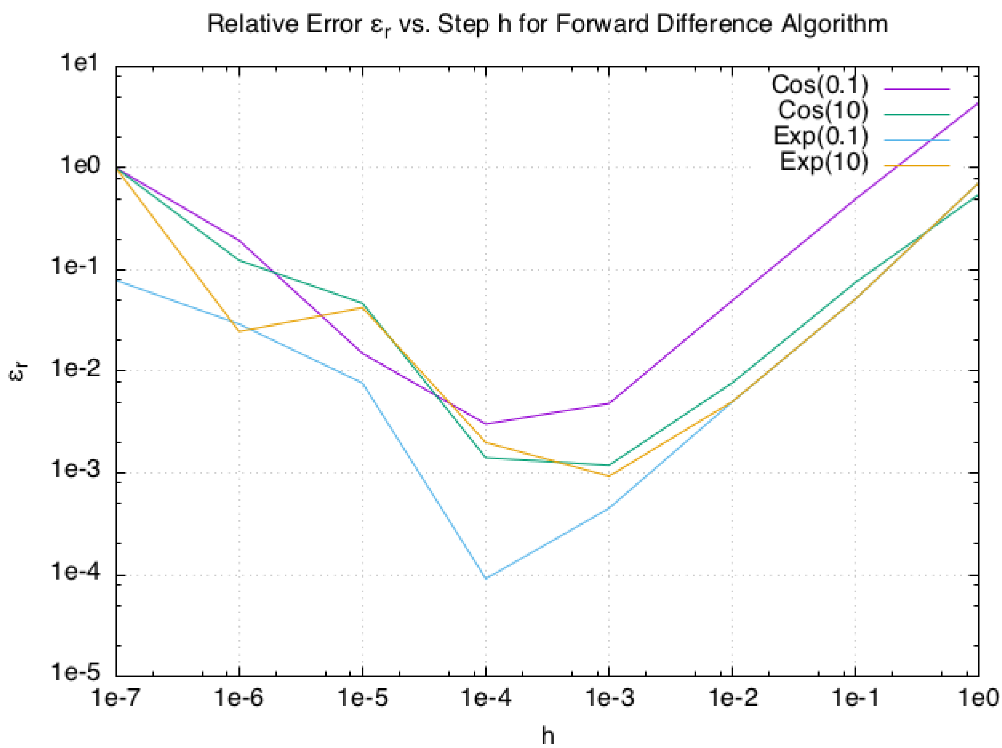

Figure 1: Relative error of the derivative of $\cos(x)$ and $\exp(x)$ for $x = 0.1, 10$ calculated using the forward differences algorithm as a function of the step h.

We have the following estimates for the optimal step h_{opt} and optimal relative error ϵ_r for a forward difference algorithm from class:

$$h_{opt} \sim \sqrt{\left|\frac{f(x_o)}{f''(x_o)}\right|\epsilon_m} \tag{1}$$

$$\epsilon_{opt} \sim \sqrt{\epsilon_m} \tag{2}$$

where x_o is the point for which the derivative is being calculated, ϵ_m is the machine precision which we take to be $\sim 10^{-7}$, and f is the function whose derivative is being calculated. (2) corresponds to an optimal error of $\epsilon_{opt} \sim 10^{-4}$. Looking at Figure 1, we see the minimum relative errors range from $10^{-3} - 10^{-4}$, in good agreement with out rough estimate for ϵ_{opt}. The table below lists the estimated value of the optimal step h_{opt} as given by (1).

$f(x_o)$	h_{opt}
$\cos(0.1)$	$\sim 10^{-4}$
$\cos(10)$	$\sim 10^{-4}$
$\exp(0.1)$	$\sim 10^{-4}$
$\exp(10)$	$\sim 10^{-4}$

Comparing the values in the table to the position of the minimum relative error in Figure 1 ($h = 10^{-3} - 10^{-4}$ for all), we see that the table and the plots are in good agreement. For a forward difference algorithm we expect the truncation error ϵ_T and round-off error ϵ_{RO} to be estimated by the following:

$$\epsilon_{RO} \propto \frac{1}{h} \tag{3}$$

$$\epsilon_T \propto h \tag{4}$$

Looking at the region of the plots in Figure 1 for $h > h_{opt}$ the $\epsilon_r(h)$ are straight lines in the log-log plot, indicating that $\epsilon_r \sim h^n$ for some positive integer n. It is clear from the plot that the ϵ_r increases by a factor of 10 when h increases by a factor of 10, indicating the slope of the line in the log-log plot is roughly 1. Thus, $n \approx 1$ as expected.

Looking at the region of the plots in Figure 1 for $h < h_{opt}$ the $\epsilon_r(h)$ look to be roughly following straight lines. This time the log-log slope of most of these lines looks to be approximately -1, indicating that $n \approx -1$ as we expect.

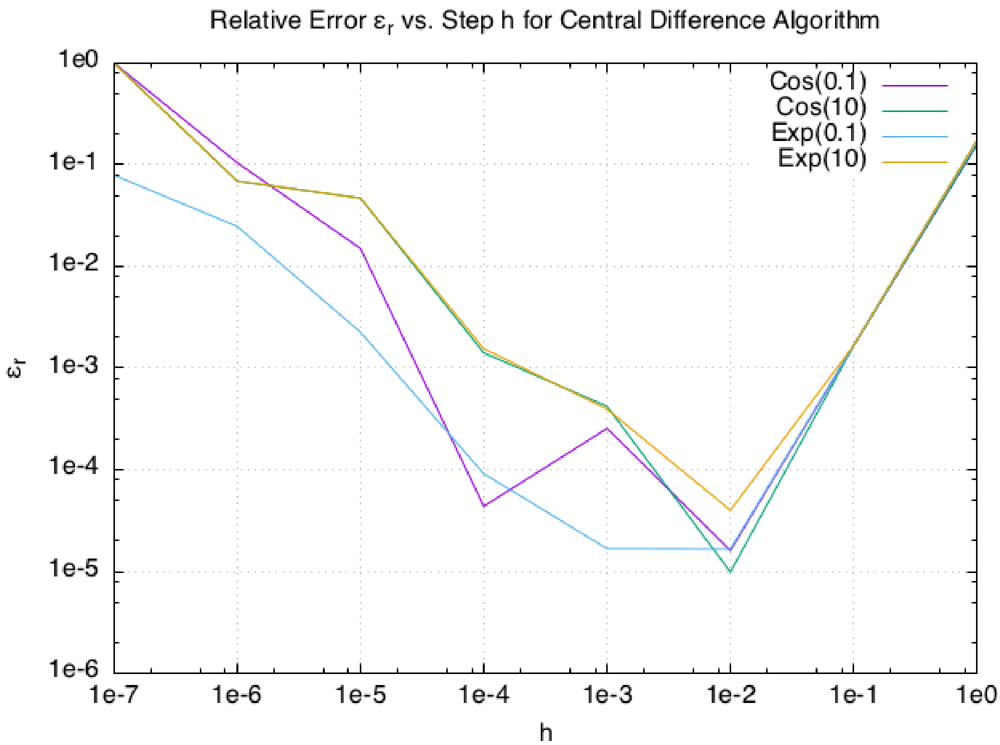

Figure 2: Relative error of the derivative of $\cos(x)$ and $\exp(x)$ for $x = 0.1, 10$ calculated using the central differences algorithm as a function of the step h.

For a central difference algorithm we have the following estimates for h_{opt} and ϵ_{opt} from class:

$$h_{opt} \sim \left(\left| \frac{f(x_o)}{f'''(x_o)} \right| \epsilon_m \right)^{1/3} \tag{5}$$

$$\epsilon_{opt} \sim \epsilon_m^{2/3} \tag{6}$$

For single-precision measurements, (6) corresponds to an optimal error of $\epsilon_r \sim 10^{-5}$. Looking at Figure 2, we see the minimum relative errors are roughly 10^{-5}, as expected. The table below lists the estimated value of the optimal step h_{opt} as given by (5).

Comparing the values in the table to the position of the minimum relative error for each plot in Figure 2 ($h = 10^{-3} - 10^{-4}$ for all) we see that the table and the plots are again in good agreement. For a central difference

$f(x_o)$	h_{opt}
$\cos(0.1)$	$\sim 10^{-3}$
$\cos(10)$	$\sim 10^{-3}$
$\exp(0.1)$	$\sim 10^{-4}$
$\exp(10)$	$\sim 10^{-4}$

algorithm we expect the round-off error ϵ_{RO} to scale as in (3), but we expect the truncation error ϵ_T to this time scale as:

$$\epsilon_T \propto h^2 \tag{7}$$

Looking at the region of the plots in Figure 2 for $h > h_{opt}$, the $\epsilon_r(h)$ look to be roughly straight lines in the log-log plot. This time, though, it is clear that the slope of these lines in the log-log plot is roughly 2, indicating $n \approx 2$ as we would expect.

Looking at the region of the plots in Figure 2 for $h < h_{opt}$, the $\epsilon_r(h)$ look to roughly be following straight lines. For very small h, it is clear that these lines have slope of about -1 in the log-log plot, indicating $n \approx -1$ as we expect.

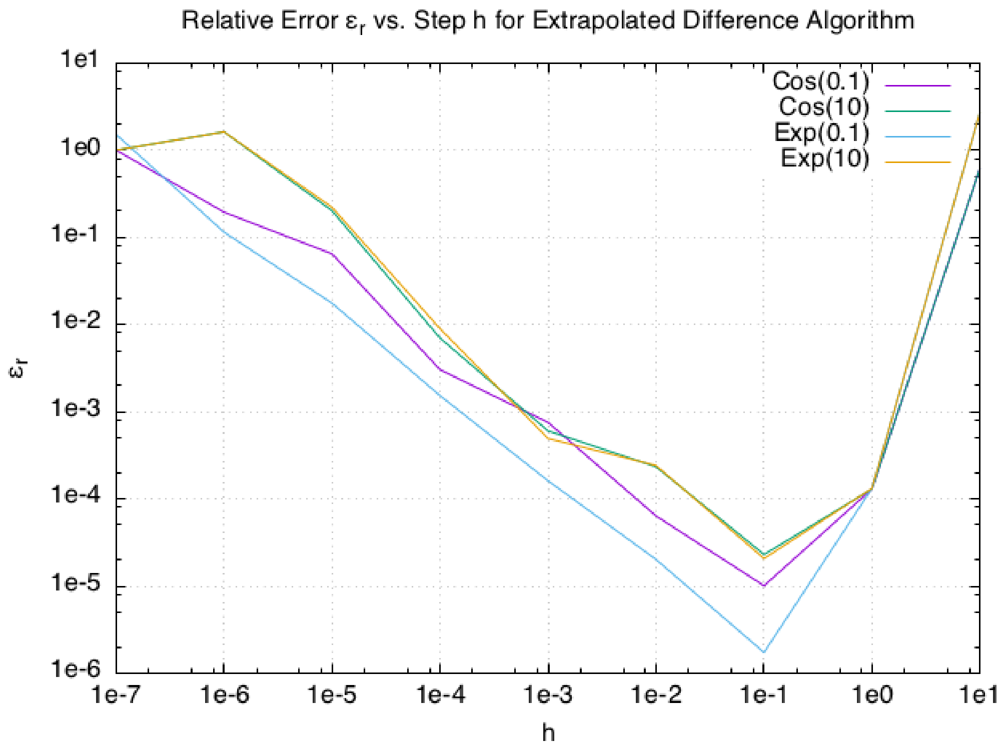

Figure 3: Relative error of the derivative of $\cos(x)$ and $\exp(x)$ for $x = 0.1, 10$ calculated using the extrapolated differences algorithm as a function of the step h.

For an extrapolated difference algorithm we have the following estimates for h_{opt} and ϵ_{opt} from class:

$$h_{opt} \sim \left(\left| \frac{f(x_o)}{f^{(5)}(x_o)} \right| \epsilon_m \right)^{1/5} \tag{8}$$

$$\epsilon_{opt} \sim \epsilon_m^{3/5} \tag{9}$$

For single-precision measurements, (9) corresponds to an optimal error of $\epsilon_r \sim 10^{-5}$. Looking at Figure 3, we see the minimum relative errors are roughly 10^{-5} on average, as expected. The table below lists the estimated value of the optimal step h_{opt} as given by (8).

Comparing the values in the table to the position of the minimum relative error for each plot in Figure 3 ($h = 10^{-1}$ for all) we see that the table and the

$f(x_o)$	h_{opt}
$\cos(0.1)$	$\sim 10^{-2}$
$\cos(10)$	$\sim 10^{-2}$
$\exp(0.1)$	$\sim 10^{-2}$
$\exp(10)$	$\sim 10^{-2}$

plots are roughly in agreement. For an extrapolated difference algorithm we expect the round-off error ϵ_{RO} to scale as in (3), but we expect the truncation error ϵ_T to scale as:

$$\epsilon_T \propto h^4 \tag{10}$$

Looking at the region of the plots in Figure 3 for $h > h_{opt}$, the $\epsilon_r(h)$ look to be roughly straight lines in the log-log plot. It is clear that the slop of these lines in the log-log plot is roughly 4 for large h, indicating $n \approx 4$ as expected.

Looking at the region of the plots in Figure 3 for $h < h_{opt}$, the $\epsilon_r(h)$ look to be roughly following straight lines, as well. For small h, it is clear that these lines have slope of about -1 in the log-log plot, indicating $n \approx -1$ as we would expect.

2.

The code written to implement each of the algorithms can be found in the source code `prob2.cpp` in the appendix. Graphs and their data were generated by running `prob2.sh`, also in the appendix. Figures 4-6, below, show the relative error ϵ_r of each algorithm as a function of the number of abscissa N. This data was generated by performing the following integration numerically:

$$\int_0^1 \exp(-t)dt$$

Lastly, the look-up table for the roots of the Legendre polynomials and their corresponding weights used for Gauss-Legendre quadrature were obtained from [1].

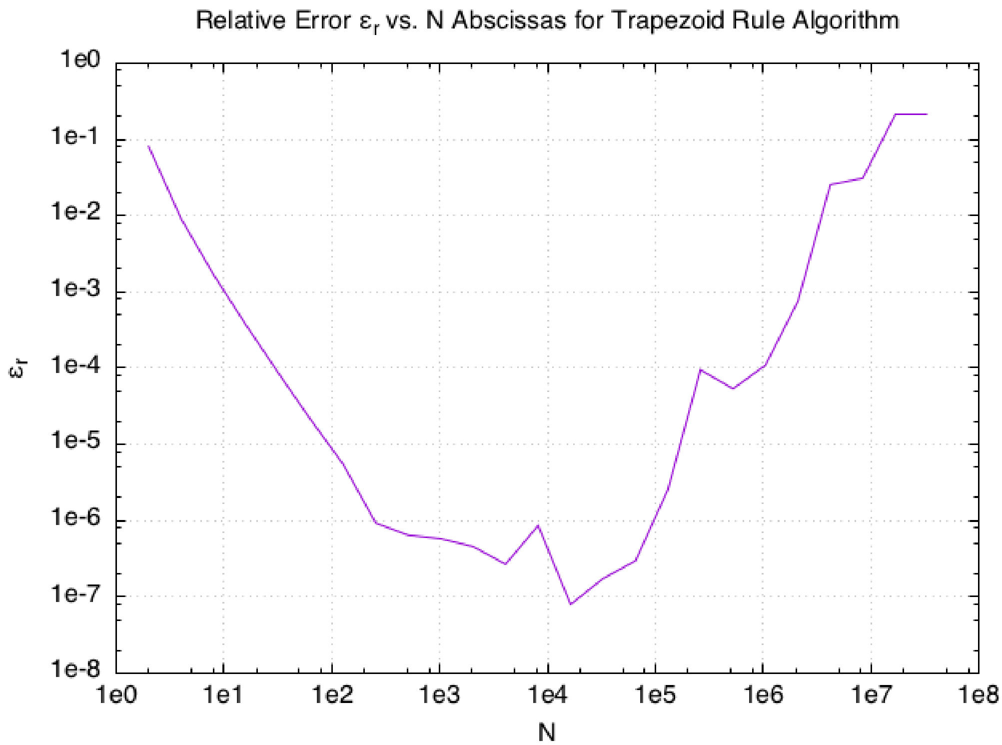

Figure 4: Relative error ϵ_r for the trapezoid rule algorithm as a function of the number of abscissa N.

In Figure 4, above, we see the relative error ϵ_r as a function of the number of abcissa N. For a trapezoid rule algorithm, we expect the optimal relative error ϵ_{opt} and optimal number of abcissa N_{opt} to be given by the following:

$$\epsilon_{opt} \sim \epsilon_m^{4/5} \tag{11}$$

$$N_{opt} \sim \epsilon_m^{-2/5} \tag{12}$$

For single-precision measurements, (11) corresponds to $\sim 10^{-6}$, while (12) corresponds to $\sim 10^2$. Looking at Figure 4, we see the minimum relative error we obtain is $\sim 10^{-7}$, in fairly good agreement with our rough estimate. However, our optimal N is $\sim 10^4$, in contrast with our estimate. It is worth noting, though, that for $N \sim 10^2$ we do reach a relative error of $\epsilon_r \sim 10^{-6}$. Finally, for a trapezoid rule algorithm, we expect the round-off error ϵ_{RO} and truncation error ϵ_T to scale in the following ways:

$$\epsilon_{RO} \propto \sqrt{N} \tag{13}$$

$$\epsilon_T \propto N^{-2} \tag{14}$$

For $N < N_{opt}$, which we take to be the computated value, rather than the theoretical value, we see that $\epsilon_r(N)$ follows a straight line in the log-log plot. It can be seen that the slope of this line is roughly -2 in the log-log plot, indicating $n \approx -2$, as expected.

For $N > N_{opt}$, we see that $\epsilon_r(N)$ seems to be following a parabolic path, which is not expected. Even if we approximate this path by a straight line it is clear the slope of this line in the log-log plot would be roughly 3, in stark contrast to our expected value of $\frac{1}{2}$. The source of this discrepancy is likely due to some inefficiency in our algorithm's computation. An excessive number of computations may make the round-off error more severe than the best case scenario which the theoretical value estimates.

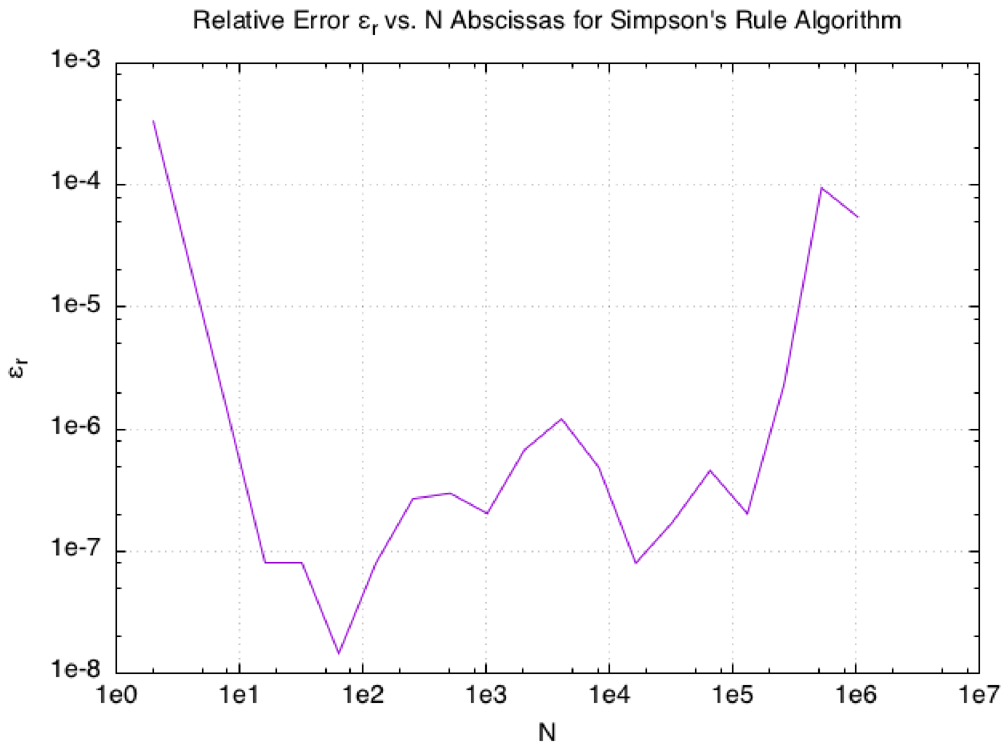

Figure 5: Relative error ϵ_r for the Simpon's rule algorithm as a function of the number abscissa N.

In Figure 5, above, we see the relative error ϵ_r as a function of the number of abcissa N. For a Simpson's Rule algorithm, we expect the optimal relative error ϵ_{opt} and optimal number of abcissa N_{opt} to be given by the following:

$$\epsilon_{opt} \sim \epsilon_m^{8/9} \tag{15}$$

$$N_{opt} \sim \epsilon_m^{-2/9} \tag{16}$$

For single-precision measurements, (15) corresponds to $\sim 10^{-7}$, while (16) corresponds to ~ 10. Looking at Figure 5, we see the minimum relative error we obtain is $\sim 10^{-7}$, in good agreement with our estimate. The optimal number of abcissa can be seen to be ~ 10, also in very good agreement with our estimate. Lastly, for a Simpson's Rule algorithm, we expect the round-off error ϵ_{RO} to follow (13), while the truncation error ϵ_T should scale in the following way:

$$\epsilon_T \propto N^{-4} \tag{17}$$

For $N < N_{opt}$, we see that $\epsilon_r(N)$ follows a straight line in the log-log plot. The slope of this line looks to be about -4 in the log-log plot, indicating $n \approx -4$, as expected.

For $N > N_{opt}$, but not too large, we see that $\epsilon_r(N)$, if approximated by a straight line, has a slope of roughly $-\frac{1}{2}$ in the log-log plot. This, of course, indicates $n \approx -\frac{1}{2}$, as we would expect. However, for large N, $\epsilon_r(N)$ follows a straight line in the log-log plot of slope roughly 2, indicating $n \approx 2$. This is again in stark contrast with our estimate and is likely due to inefficiencies in our algorithm, which causes the round-off error to be more severe than estimated.

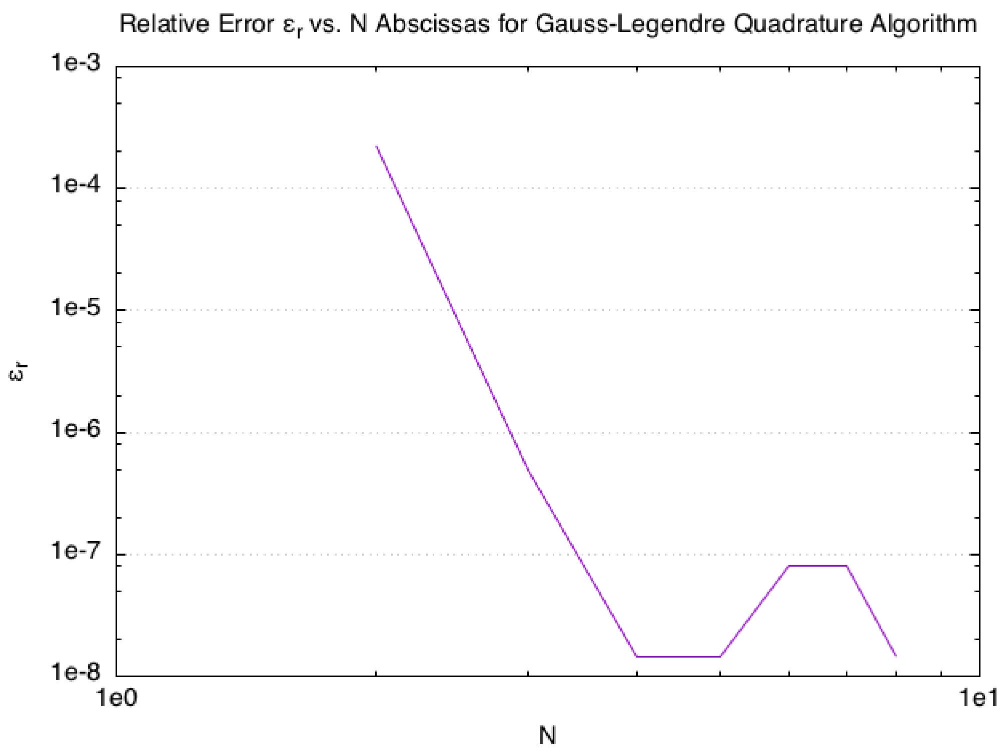

Figure 6: Relative error ϵ_r for the Gauss-Legendre quadrature algorithm as a function of the number of abscissa N.

In Figure 6, we again see the relative error ϵ_r plotted as a function of the number of abcissa N. It is clear from the figure that we obtain $N_{opt} \sim 1$ and

$\epsilon_{opt} \sim 10^{-8}$, making it by far the most efficient and precise algorithm. Note that as N increases we begin to see the very start of the loss of precision to round-off error.

3.

The code written to generate the random walk can be found in the source code `prob3.cpp` in the appendix. Plots and data were generate by running `prob3.sh`, also in the appendix. Figures 7-9, below, show σ^2, s_3, and s_4 for the random walk as a function of n.

For a random walker starting a position x_0 and taking unit length steps forward or backward, the position of the walker after n steps is given by the recursion relation:

$$\left.\begin{aligned} x_0 &= 0 \\ x_n &= x_{n-1} + l_n \end{aligned}\right\} \tag{18}$$

where l_n is a random variable, equal to ± 1 with equal probability, giving the n^{th} step. Plugging each x_k, $k = 0, 1, ..., n-1$, into (18) we obtain the closed form equation for x_n:

$$\begin{aligned} x_n &= x_0 + \sum_{k=1}^{n} l_k \\ &= \sum_{k=1}^{n} l_k \end{aligned} \tag{19}$$

Finally, we note several facts about the l_k. Firstly, l_k and l_j are independent random variables for $k \neq j$. Secondly, the following expectation values can be derived easily from the definition:

$$\langle l_k \rangle = 0 \tag{20a}$$
$$\langle l_k^2 \rangle = 1 \tag{20b}$$
$$\langle l_k^3 \rangle = 0 \tag{20c}$$
$$\langle l_k^4 \rangle = 1 \tag{20d}$$

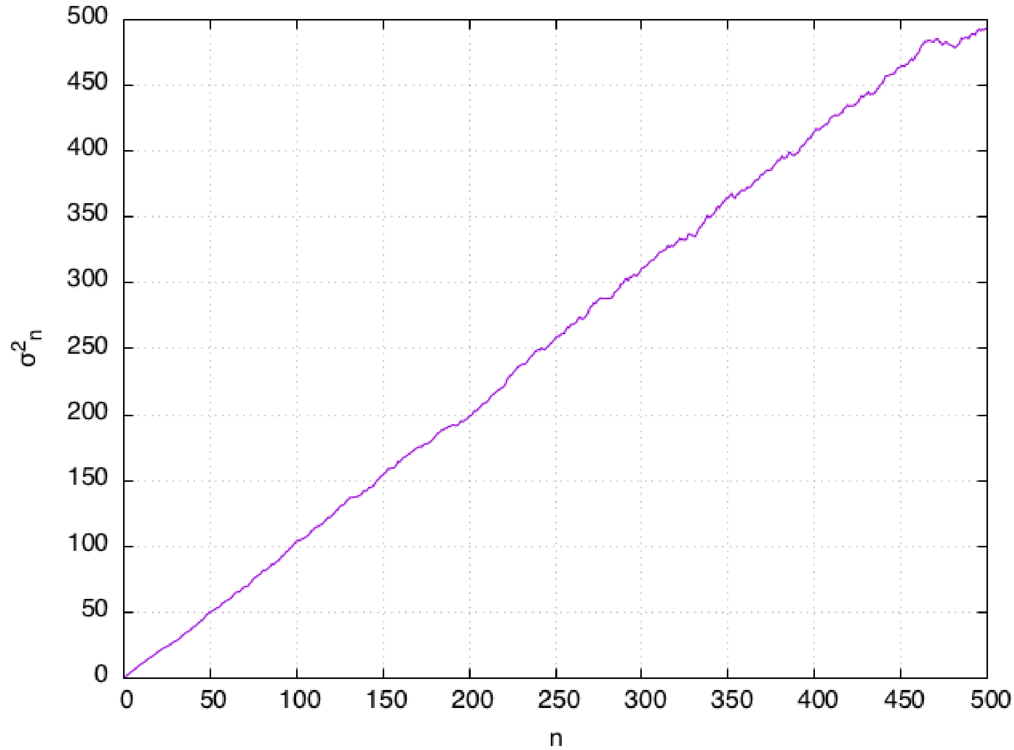

Figure 7: σ^2 as a function of the number of steps n in the random walk.

Figure 7, above, shows the variance of σ_n^2 the final position after n steps x_n as a function of the number of steps n. Here, we calculate the variance using:

$$\begin{aligned} \sigma_n^2 &= \langle x_n^2 \rangle - \langle x_n \rangle^2 \\ &= \langle x_n^2 \rangle \end{aligned} \tag{21}$$

where we obtain the second line by comparing (19) and (20a). To obtain a theoretical estimate for the variance we expand x_n^2 using (19):

$$\begin{aligned} x_n^2 &= \left(\sum_{k=1}^{n} l_k \right)^2 \\ &= \sum_{k=1}^{n} l_k^2 + 2 \sum_{k=1}^{n} \sum_{j<k} l_k l_j \\ \implies \langle x_n^2 \rangle &= n \end{aligned} \tag{22}$$

where we obtain the final line using the independence of the l_k, (20a), and (20b). Therefore, in the limit of large n we expect:

$$\sigma_n^2 = n \tag{23}$$

Comparing this prediction to Figure 7, we see that our results are in very good agreement with (23).

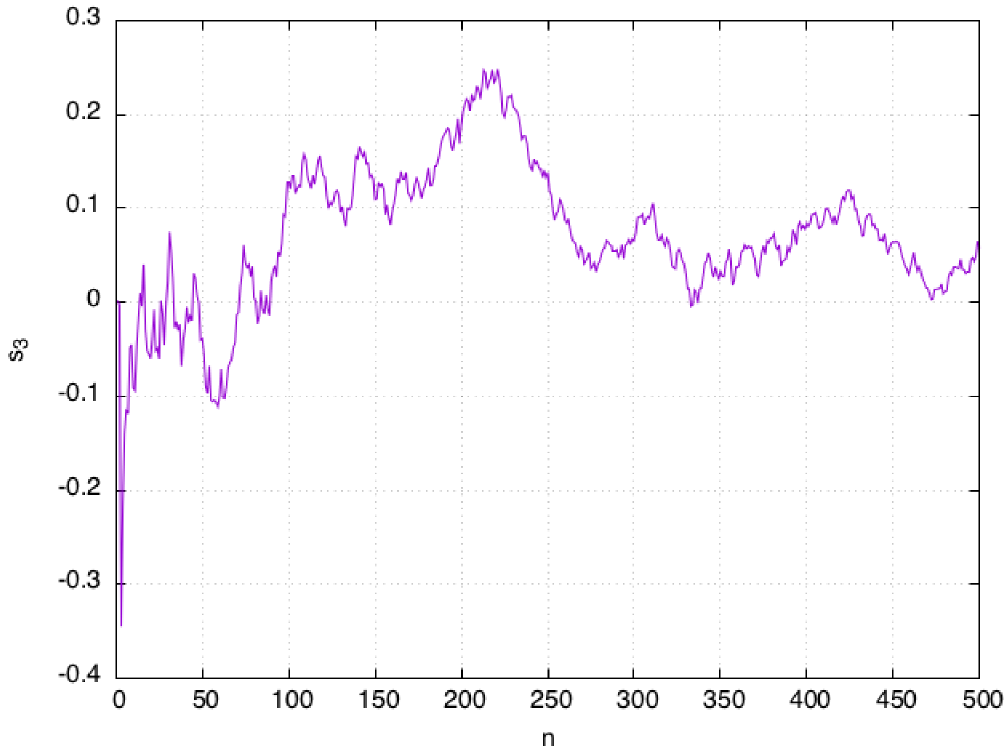

Figure 8: s_3 as a function of the number of steps n in the random walk.

Figure 8, above, shows s_3 as a function of n, where s_3 is given by:

$$s_3 = \frac{\langle x_n^3 \rangle}{\sigma_n^3} \tag{24}$$

To obtain a theoretical estimate for s_3 we expand x_n^3 using (19):

$$\begin{aligned} x_n^3 &= \left(\sum_{k=1}^{n} l_k\right)^3 \\ &= \sum_{k=1}^{n} l_k^3 + 6\sum_{k=1}^{n}\sum_{j<k} l_k^2 l_j + 6\sum_{k=1}^{n}\sum_{j<k}\sum_{i<j} l_k l_j l_i \\ \implies \langle x_n^3 \rangle &= 0 \end{aligned} \tag{25}$$

where we obtain the final line using the independence of l_k, (20a), and (20c). Therefore, in the limit of large n we expect:

$$s_3 = 0 \tag{26}$$

Comparing this prediction to Figure 8, we see that our results are in fairly good agreement with (26).

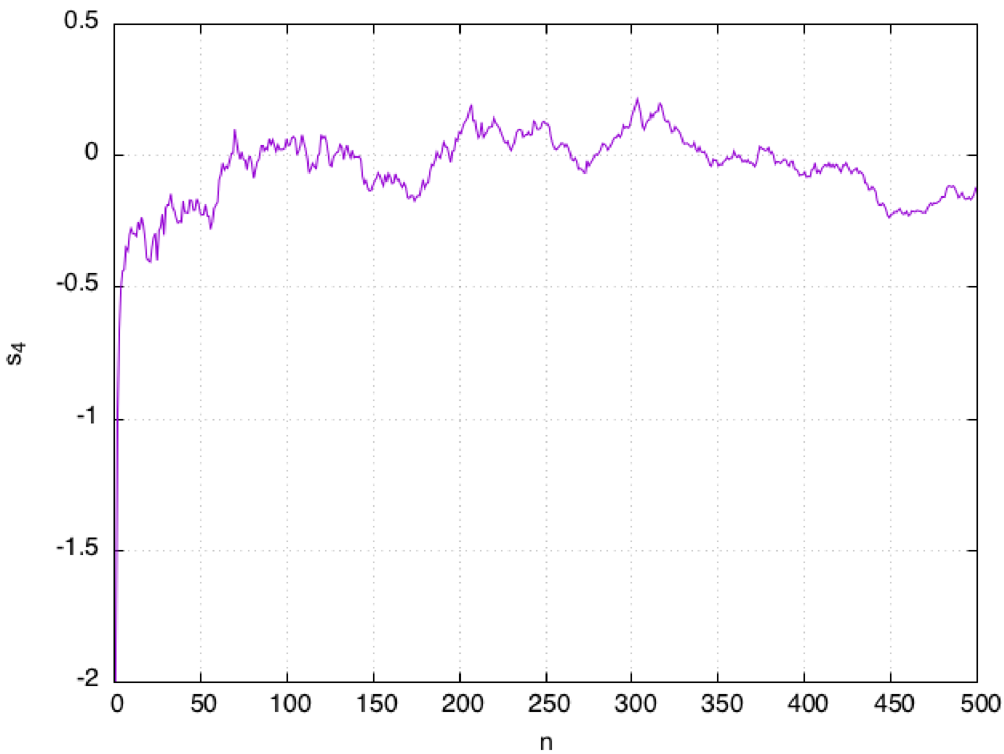

Figure 9: s_4 as a function of the number of steps n in the random walk.

Figure 9, above, shows s_4 as a function of n, where $s_4$is given by:

$$s_4 = \frac{\langle x_n^4 \rangle}{\sigma_n^4} - 3 \tag{27}$$

To obtain a theoretical estimate for s_4 we expand x_n^4 using (19):

$$\begin{aligned}
x_n^4 =& \left(\sum_{k=1}^{n} l_k \right)^4 \\
=& \sum_{k=1}^{n} l_k^4 + 6 \sum_{k=1}^{n} \sum_{j<k} l_k^2 l_j^2 + 4 \sum_{k=1}^{n} \sum_{j<k} l_k^3 l_j \\
&+ 12 \sum_{k=1}^{n} \sum_{j<k} \sum_{i<j} l_k^2 l_j l_i + 24 \sum_{k=1}^{n} \sum_{j<k} \sum_{i<j} \sum_{h<i} l_k l_j l_i l_h \\
\implies \langle x_n^4 \rangle =& n + 6 \frac{n(n-1)}{2} \\
=& 3n^2 - 2n
\end{aligned} \tag{28}$$

where we obtain the final expression using the independence of the l_k and (20). Therefore, in the limit of large n we expect:

$$s_4 = 0 \tag{29}$$

where we have plugged in $\sigma_n^2 = n$ and taken dropped terms approaching zero. Comparing this prediction to Figure 9, we see that our results are in very good agreement with (31).

References

[1] Lowan, Arnold N.; Davids, Norman; Levenson, Arthur. *Table of the zeros of the Legendre polynomials of order 1-16 and the weight coefficients for Gauss' mechanical quadrature formula.* Bull. Amer. Math. Soc. 48 (1942), no. 10, 739–743. http://projecteuclid.org/euclid.bams/1183504772.

Appendix

prob1.cpp:

```
// Marco Muzio - Homework 1 Problem 1 Source Code

#include<iostream>
#include<cmath>
#include<fstream>

using namespace std;

float forward_diff(float x, float h, float (*f)(float));
float central_diff(float x, float h, float (*f)(float));
float extrap_diff(float x, float h, float (*f)(float));
float rel_error(float x, float derivative, float (*df)(float));
float neg_sin(float x);

int main()
{
        const float e_m = 1.0e-7; // Approximate machine
         ↪  precision for SP
        float derivative, x;

        ofstream cos0d1, cos10, exp0d1, exp10;

        // Forward difference calculations

        cos0d1.open("cos0d1_forward_diff.txt");
        cos10.open("cos10_forward_diff.txt");
        exp0d1.open("exp0d1_forward_diff.txt");
        exp10.open("exp10_forward_diff.txt");

        for(float h=1.0; h>=e_m/10.0; h /= 10.0)
        {
                x = 0.1;
                derivative = forward_diff(x, h, cos);
                cos0d1 << h << "\t" << derivative << "\t" <<
                 ↪  rel_error(x, derivative, neg_sin) << endl;
```

```
        derivative = forward_diff(x, h , exp);
        exp0d1 << h << "\t" << derivative << "\t" <<
         ↪  rel_error(x, derivative, exp)<< endl;

        x = 10.0;
        derivative = forward_diff(x, h, cos);
        cos10 << h << "\t" << derivative << "\t" <<
         ↪  rel_error(x, derivative, neg_sin) << endl;

        derivative = forward_diff(x, h, exp);
        exp10 << h << "\t" << derivative << "\t" <<
         ↪  rel_error(x, derivative, exp) << endl;
}

cos0d1.close();
cos10.close();
exp0d1.close();
exp10.close();

// Central difference calculations

cos0d1.open("cos0d1_central_diff.txt");
cos10.open("cos10_central_diff.txt");
exp0d1.open("exp0d1_central_diff.txt");
exp10.open("exp10_central_diff.txt");

for(float h=1.0; h>=e_m/10.0; h /= 10.0)
{
        x = 0.1;
        derivative = central_diff(x, h, cos);
        cos0d1 << h << "\t" << derivative << "\t" <<
         ↪  rel_error(x, derivative, neg_sin) << endl;

        derivative = central_diff(x, h , exp);
        exp0d1 << h << "\t" << derivative << "\t" <<
         ↪  rel_error(x, derivative, exp)<< endl;
```

```
        x = 10.0;
        derivative = central_diff(x, h, cos);
        cos10 << h << "\t" << derivative << "\t" <<
         ↪  rel_error(x, derivative, neg_sin) << endl;

        derivative = central_diff(x, h, exp);
        exp10 << h << "\t" << derivative << "\t" <<
         ↪  rel_error(x, derivative, exp) << endl;
}

cos0d1.close();
cos10.close();
exp0d1.close();
exp10.close();

// Extrapolated difference calculations

cos0d1.open("cos0d1_extrap_diff.txt");
cos10.open("cos10_extrap_diff.txt");
exp0d1.open("exp0d1_extrap_diff.txt");
exp10.open("exp10_extrap_diff.txt");

for(float h=10.0; h>=e_m/10.0; h /= 10.0)
{
        x = 0.1;
        derivative = extrap_diff(x, h, cos);
        cos0d1 << h << "\t" << derivative << "\t" <<
         ↪  rel_error(x, derivative, neg_sin) << endl;

        derivative = extrap_diff(x, h , exp);
        exp0d1 << h << "\t" << derivative << "\t" <<
         ↪  rel_error(x, derivative, exp)<< endl;

        x = 10.0;
        derivative = extrap_diff(x, h, cos);
        cos10 << h << "\t" << derivative << "\t" <<
         ↪  rel_error(x, derivative, neg_sin) << endl;
```

```
                derivative = extrap_diff(x, h, exp);
                exp10 << h << "\t" << derivative << "\t" <<
                 ↪  rel_error(x, derivative, exp) << endl;
        }

        cos0d1.close();
        cos10.close();
        exp0d1.close();
        exp10.close();

        return 0;
}

// Forward difference algorithm for calculating f'
float forward_diff(float x, float h, float (*f)(float))
{
        float derivative;

        derivative = (*f)(x+h) - (*f)(x);
        derivative /= h;

        return derivative;
}

// Central difference algorithm for calculating f'
float central_diff(float x, float h, float (*f)(float))
{
        float derivative;

        derivative = (*f)(x+h) - (*f)(x-h);
        derivative /= 2.0*h;

        return derivative;
}

// Extrapolated difference algorithm for calculating f'
float extrap_diff(float x, float h, float (*f)(float))
{
```

```
        float derivative;

        derivative = 4.0*central_diff(x, h/4.0, (*f)) -
         ↪  central_diff(x, h/2.0, (*f));
        derivative /= 3.0;

        return derivative;
}

// Relative error
float rel_error(float x, float derivative, float (*df)(float))
{
        float error;

        error = derivative - (*df)(x);
        error /= (*df)(x);
        error = abs(error);

        return error;
}

// Negative of sin(x) so that the derivative of cos(x) can be
 ↪  passed to the above functions
float neg_sin(float x)
{
        return -1.0*sin(x);

}
```

prob1.sh:

```
#!/bin/sh

### Marco Muzio - Homework 1 Problem 1 Script

# Runs prob1 program to generate data
./prob1
```

```
# Plots results on log-log plots

## Plot data for forward difference algorithm
gnuplot << EOF
set terminal pngcairo enhanced
set encoding utf8
set output 'forward_diff.png'
set title 'Relative Error {/Symbol e}_r vs. Step h for Forward
↪  Difference Algorithm'
set xlabel 'h'
set ylabel '{/Symbol e}_r'
set logscale xy
set format xy '1e%T'
set grid
plot 'cos0d1_forward_diff.txt' using 1:3 with lines title
↪  'Cos(0.1)',\
         'cos10_forward_diff.txt' using 1:3 with lines title
↪  'Cos(10)', \
         'exp0d1_forward_diff.txt' using 1:3 with lines title
↪  'Exp(0.1)', \
         'exp10_forward_diff.txt' using 1:3 with lines title
↪  'Exp(10)'
EOF

## Plot data for central difference algorithm
gnuplot << EOF
set terminal pngcairo enhanced
set encoding utf8
set output 'central_diff.png'
set title 'Relative Error {/Symbol e}_r vs. Step h for Central
↪  Difference Algorithm'
set xlabel 'h'
set ylabel '{/Symbol e}_r'
set logscale xy
set format xy '1e%T'
set grid
plot 'cos0d1_central_diff.txt' using 1:3 with lines title
↪  'Cos(0.1)', \
```

```
        'cos10_central_diff.txt' using 1:3 with lines title
↪  'Cos(10)', \
        'exp0d1_central_diff.txt' using 1:3 with lines title
↪  'Exp(0.1)', \
        'exp10_central_diff.txt' using 1:3 with lines title
↪  'Exp(10)'
EOF

## Plot data for extrapolated difference algorithm
gnuplot << EOF
set terminal pngcairo enhanced
set encoding utf8
set output 'extrap_diff.png'
set title 'Relative Error {/Symbol e}_r vs. Step h for
↪  Extrapolated Difference Algorithm'
set xlabel 'h'
set ylabel '{/Symbol e}_r'
set logscale xy
set format xy '1e%T'
set grid
plot 'cos0d1_extrap_diff.txt' using 1:3 with lines title
↪  'Cos(0.1)', \
        'cos10_extrap_diff.txt' using 1:3 with lines title
↪  'Cos(10)', \
        'exp0d1_extrap_diff.txt' using 1:3 with lines title
↪  'Exp(0.1)', \
        'exp10_extrap_diff.txt' using 1:3 with lines title
↪  'Exp(10)'
EOF
```

prob2.cpp:

```
// Marco Muzio - Homework 1 Problem 2 Source Code

#include<iostream>
#include<cmath>
#include<fstream>
```

```
#define pi 3.14159265

using namespace std;

float trap_rule(float lower, float upper, int N_points);
float simps_rule(float lower, float upper, int N_points);
float GL_quad(float lower, float upper, int N_points);
float Leg_roots(int n, int root);
float Leg_weights(int n, int root);
float rel_error(float integral);

int main()
{
        float integral;

        ofstream trap, simps, GL;

        trap.open("trapezoid_rule.txt");
        simps.open("simpsons_rule.txt");
        GL.open("GL_quadrature.txt");

        // Trapezoid rule calculations
        for(int i=2; i<=pow(2, 25); i *= 2)
        {
                integral = trap_rule(0.0, 1.0, i);

                trap << i << "\t" << integral << "\t" <<
                 ↪  rel_error(integral) << endl;
        }

        trap.close();

        // Simpson's rule calculations
        for(int i=2; i<=pow(2, 20); i *= 2)
        {
                integral = simps_rule(0.0, 1.0, i+1);
```

```
                simps << i << "\t" << integral << "\t" <<
                 ↪  rel_error(integral) << endl;
        }

        simps.close();

        // GL quadrature calculations
        for(int i=2; i<9; i++)
        {
                integral = GL_quad(0.0, 1.0, i);

                GL << i << "\t" << integral << "\t" <<
                 ↪  rel_error(integral) << endl;
        }

        GL.close();

        return 0;
}

// Trapezoid rule algorithm to calculate integral of exp(-t)
float trap_rule(float lower, float upper, int N_points)
{
        float h, integral, x;

        h = (upper - lower)/(N_points - 1.0);

        integral = 0.5*exp(-lower) + 0.5*exp(-upper);

        for(int i=2; i<N_points; i++)
        {
                x = lower + (i-1)*h;
                integral += exp(-x);
        }

        integral *= h;

        return integral;
```

```
}

// Simpson's rule algorithm to calculate integral of exp(-t)
float simps_rule(float lower, float upper, int N_points)
{
        float h, integral, x_even, x_odd;

        h = (upper - lower)/(N_points - 1.0);

        integral = exp(-lower)/3.0 + exp(-upper)/3.0;

        for(int i=2; i<N_points-1; i += 2)
        {
                x_even = lower + (i-1)*h;
                x_odd = x_even + h;

                integral += 4.0*exp(-x_even)/3.0 +
                 ↪  2.0*exp(-x_odd)/3.0;
        }

        x_even = lower + (N_points-2.0)*h;

        integral += 4.0*exp(-x_even)/3.0;

        integral *= h;

        return integral;
}

// Gauss-Legendre quadrature algorithm to calculate integral
 ↪  of exp(-t) -- only valid for N_points<9
float GL_quad(float lower, float upper, int N_points)
{
        float integral, weight, transformation_factor, f;
        float x;

        integral = 0.0;
```

```
        for(int i=1; i<=N_points; i++)
        {
                x = Leg_roots(N_points, i);
                weight = Leg_weights(N_points, i);
                transformation_factor = (upper - lower)/2.0;
                f = exp(-((upper - lower)/2.0*x + (upper +
                 ↪  lower)/2.0));

                integral += f*weight*transformation_factor;
        }

        return integral;
}

// Look-up table of 2nd-7th Legendre polynomial roots
float Leg_roots(int n, int root)
{
        switch(n)
        {
                case 2:

                        switch(root)
                        {
                                case 1: return
                                 ↪  -0.577350269189626;
                                case 2: return
                                 ↪  0.577350269189626;
                        }

                case 3:

                        switch(root)
                        {
                                case 1: return
                                 ↪  -0.774596669241483;
                                case 2: return 0.0;
                                case 3: return
                                 ↪  0.774596669241483;
```

```
        }

case 4:

        switch(root)
        {
                case 1: return
                 ↪  -0.86113631159405;
                case 2: return
                 ↪  -0.339981043584856;
                case 3: return
                 ↪  0.339981043584856;
                case 4: return
                 ↪  0.86113631159405;
        }

case 5:

        switch(root)
        {
                case 1: return
                 ↪  -0.906179845938664;
                case 2: return
                 ↪  -0.538469310105683;
                case 3: return 0.0;
                case 4: return
                 ↪  0.538469310105683;
                case 5: return
                 ↪  0.906179845938664;
        }

case 6:

        switch(root)
        {
                case 1: return
                 ↪  -0.932469514203152;
```

```
                case 2: return
                 ↪  -0.661209386466265;
                case 3: return
                 ↪  -0.238619186083197;
                case 4: return
                 ↪  0.238619186083197;
                case 5: return
                 ↪  0.661209386466265;
                case 6: return
                 ↪  0.932469514203152;
        }

case 7:

        switch(root)
        {
                case 1: return
                 ↪  -0.949107912342759;
                case 2: return
                 ↪  -0.741531185599394;
                case 3: return
                 ↪  -0.405845151377397;
                case 4: return
                 ↪  0.000000000000000;
                case 5: return
                 ↪  0.405845151377397;
                case 6: return
                 ↪  0.741531185599394;
                case 7: return
                 ↪  0.949107912342759;
        }

case 8:

        switch(root)
        {
                case 1: return
                 ↪  -0.960289856497536;
```

```
                                case 2: return
                                 ↪  -0.796666477413627;
                                case 3: return
                                 ↪  -0.525532409916329;
                                case 4: return
                                 ↪  -0.183434642495650;
                                case 5: return
                                 ↪  0.183434642495650;
                                case 6: return
                                 ↪  0.525532409916329;
                                case 7: return
                                 ↪  0.796666477413627;
                                case 8: return
                                 ↪  0.960289856497536;
                        }
        }

        return 0;
}

// Look-up table for weights corresponding to roots of
 ↪  2nd-7th Legendre polynomial
float Leg_weights(int n, int root)
{
        switch(n)
        {
                case(2):

                        switch(root)
                        {
                                case 1: return
                                 ↪  1.000000000000000;
                                case 2: return
                                 ↪  1.000000000000000;
                        }

                case(3):
```

```
            switch(root)
            {
                    case 1: return
                     ↪  0.555555555555556;
                    case 2: return
                     ↪  0.888888888888889;
                    case 3: return
                     ↪  0.555555555555556;
            }

    case(4):

            switch(root)
            {
                    case 1: return
                     ↪  0.347854845137454;
                    case 2: return
                     ↪  0.652145154862546;
                    case 3: return
                     ↪  0.652145154862546;
                    case 4: return
                     ↪  0.347854845137454;
            }

    case(5):

            switch(root)
            {
                    case 1: return
                     ↪  0.236926885056189;
                    case 2: return
                     ↪  0.478628670499366;
                    case 3: return
                     ↪  0.568888888888889;
                    case 4: return
                     ↪  0.478628670499366;
                    case 5: return
                     ↪  0.236926885056189;
```

```
            }

    case(6):

            switch(root)
            {
                    case 1: return
                     ↪  0.171324492379170;
                    case 2: return
                     ↪  0.360761573048139;
                    case 3: return
                     ↪  0.467913934572691;
                    case 4: return
                     ↪  0.467913934572691;
                    case 5: return
                     ↪  0.360761573048139;
                    case 6: return
                     ↪  0.171324492379170;
            }

    case(7):

            switch(root)
            {
                    case 1: return
                     ↪  0.129484966168870;
                    case 2: return
                     ↪  0.279705391489277;
                    case 3: return
                     ↪  0.381830050505119;
                    case 4: return
                     ↪  0.417959183673469;
                    case 5: return
                     ↪  0.381830050505119;
                    case 6: return
                     ↪  0.279705391489277;
                    case 7: return
                     ↪  0.129484966168870;
```

```
                        }

                case(8):

                        switch(root)
                        {
                                case 1: return
                                 ↪  0.101228536290376;
                                case 2: return
                                 ↪  0.222381034453374;
                                case 3: return
                                 ↪  0.313706645877887;
                                case 4: return
                                 ↪  0.362683783378362;
                                case 5: return
                                 ↪  0.362683783378362;
                                case 6: return
                                 ↪  0.313706645877887;
                                case 7: return
                                 ↪  0.222381034453374;
                                case 8: return
                                 ↪  0.101228536290376;
                        }
        }

        return 0;
}

// Relative error in calculated integral
float rel_error(float integral)
{
        float error;

        error = integral - (1.0-exp(-1.0));
        error /= 1.0-exp(-1.0);
        error = abs(error);

        return error;
```

```
}
```

prob2.sh:

```sh
#!/bin/sh

### Marco Muzio - Homework 1 Problem 2 Script

# Runs prob2 program to generate data
./prob2

# Plots results on log-log plots

## Plot data for trapezoid rule
gnuplot << EOF
set terminal pngcairo enhanced
set encoding utf8
set output 'trapezoid_rule.png'
set title 'Relative Error {/Symbol e}_r vs. N Abscissas for
↪ Trapezoid Rule Algorithm'
set xlabel 'N'
set ylabel '{/Symbol e}_r'
set logscale xy
set format xy '1e%T'
set grid
set nokey
plot 'trapezoid_rule.txt' using 1:3 with lines
EOF

## Plot data for Simpson's rule
gnuplot << EOF
set terminal pngcairo enhanced
set encoding utf8
set output 'simpsons_rule.png'
set title "Relative Error {/Symbol e}_r vs. N Abscissas for
↪ Simpson's Rule Algorithm"
set xlabel 'N'
set ylabel '{/Symbol e}_r'
```

```
set logscale xy
set format xy '1e%T'
set grid
set nokey
plot 'simpsons_rule.txt' using 1:3 with lines
EOF

## Plot data for Gauss-Legendre quadrature
gnuplot << EOF
set terminal pngcairo enhanced
set encoding utf8
set output 'GL_quad.png'
set title 'Relative Error {/Symbol e}_r vs. N Abscissas for
↪ Gauss-Legendre Quadrature Algorithm'
set xlabel 'N'
set ylabel '{/Symbol e}_r'
set logscale xy
set format xy '1e%T'
set grid
set nokey
plot 'GL_quadrature.txt' using 1:3 with lines
EOF
```

prob3.cpp:

```
// Marco Muzio - Homework 1 Problem 3 Source Code

#include<cmath>
#include<iostream>
#include<fstream>

using namespace std;

int random_number(int x_in);
int step_sign(int num);

int main()
{
```

```
int iseed, sign, n_max=500, N_R=1000;
double x_0=0.0;
double x[N_R], sigma[n_max], s3[n_max], s4[n_max];

// Initialization of arrays to zero
for(int i=0; i<n_max; i++)
{
        sigma[i]=0.0; s3[i]=0.0; s4[i]=0.0;
}

for(int j=0; j<N_R; j++)
{
        x[j]=0.0;
}

// Steps Loop
for(int n_steps=1; n_steps<=n_max; n_steps++)
{
        // Realizations Loop
        for(int realization=0; realization<N_R;
         ↪  realization++)
        {
                iseed=realization+1;

                // First step
                iseed=random_number(iseed);
                sign=step_sign(iseed);
                x[realization]=x_0+pow(-1.0,sign);

                // Remaining Walk Loop
                for(int step=1; step<n_steps; step++)
                {
                        iseed=random_number(iseed);
                        sign=step_sign(iseed);
                        x[realization]+=pow(-1.0,sign);
                }
        }
```

```
        // Calculation of data for n=n_step
        for(int realization=0; realization<N_R;
         ↪  realization++)
        {
                sigma[n_steps-1] +=
                 ↪  pow(x[realization],2);
                s3[n_steps-1] += pow(x[realization],3);
                s4[n_steps-1] += pow(x[realization],4);
        }

        sigma[n_steps-1] /= N_R;
        s3[n_steps-1] /= N_R;
        s4[n_steps-1] /= N_R;

        s3[n_steps-1] /= pow(sigma[n_steps-1],
         ↪  3.0/2.0);
        s4[n_steps-1] /= pow(sigma[n_steps-1], 2);

        s4[n_steps-1] -= 3.0;
}

// Writes results to output files
ofstream sigma_data, s3_data, s4_data;
sigma_data.open("prob3_sigma.txt");
s3_data.open("prob3_s3.txt");
s4_data.open("prob3_s4.txt");

for(int n_steps=1; n_steps<=n_max; n_steps++)
{
        sigma_data << n_steps << "\t" <<
         ↪  sigma[n_steps-1] << endl;
        s3_data << n_steps << "\t" << s3[n_steps-1] <<
         ↪  endl;
        s4_data << n_steps << "\t" << s4[n_steps-1] <<
         ↪  endl;
}
```

```
        sigma_data.close(); s3_data.close(); s4_data.close();

        return 0;
}

// Random number generator
int random_number(int x_in)
{
        int x_out, a=9301, c=49297, m=233280;

        x_out= a*x_in + c;
        x_out = x_out % m;

        return x_out;
}

// Gives sign (direction) of step
int step_sign(int num)
{
        double q, m=233280;

        q = abs(((double)num)/m);
        if(q<0.5) return 0;
        else return 1;
}
```

prob3.sh:

```
#!/bin/sh

### Marco Muzio - Homework 1 Problem 3 Script

# Generate data from random walks
./prob3

## Plot data

# Plot sigma^2
```

```
gnuplot << EOF
set terminal pngcairo enhanced
set encoding utf8
set output 'sigma2.png'
set xlabel 'n'
set ylabel '{/Symbol s}^2_n'
set autoscale
set nokey
set grid
plot 'prob3_sigma.txt' with lines
EOF

# Plot s_3
gnuplot << EOF
set terminal pngcairo enhanced
set encoding utf8
set output 's_3.png'
set xlabel 'n'
set ylabel 's_3'
set autoscale
set nokey
set grid
plot 'prob3_s3.txt' with lines
EOF

# Plot s_4
gnuplot << EOF
set terminal pngcairo enhanced
set encoding utf8
set output 's_4.png'
set xlabel 'n'
set ylabel 's_4'
set autoscale
set nokey
set grid
plot 'prob3_s4.txt' with lines
EOF
```