Computational Physics Homework 1

Stanley "Alex" Breitweiser"

September 9, 2015

Introduction

Throughout this report, Julia is used to implement and run algorithms. Julia is a numerical computation language that is dynamically typed (like Python), but has a comprehensive type system that will allow us to work in single precision. Optimization was forgone for code clarity; in particular, techniques like pre-allocation, dynamic programming, and parallelization were not used, although the code could clearly benefit from it. Because Julia does not have a strong object orientated syntax, closures were used for stateful algorithms.

1 Question 1

For source code, see included file HW1Q1.jl. For output, see HW1Q1.out.

1.1 Code

1.1.1 Forward Difference Derivative

The traditional way to take a numerical derivative uses the forward difference algorithm; namely, given a step h, it looks h distance further down the dependent variable to measure the difference. Mathematically, we obtain the predicted derivative with

$$d_{f,h}f = \frac{f(x+h) - f(x)}{h}.$$

In code, we compute this with the following function:

```
#Function for single precision derivative using forward difference algorithm
function forward_diff_derivative(f, x::Float32, h::Float32)
  return (f(x+h) - f(x))/h
end
```

1.1.2 Center Difference Derivative

The center difference algorithm, on the other hand, looks $h/2$ forward and backward in the dependent variable to calculate the difference, leading to a difference calculation that is more centered on the point in question. Mathematically, we obtain the predicted derivative with

$$d_{c,h}f = \frac{f(x+h/2) - f(x-h/2)}{h}.$$

In code, we compute this with the following function:

```
#Julia interprets float literals as double precision, so creating 32-bit "two"
const TWO_SP = float32(2.0)
```

```
#Function for single precision derivative using center difference algorithm
#  Need to use TWO_SP so return expression doesn't get promoted to 64 bit
function center_diff_derivative(f, x::Float32, h::Float32)
  return (f(x + (h/TWO_SP)) - f(x - (h/(TWO_SP))))/h
end
```

1.1.3 Extrapolated Difference Derivative

At the cost of additional function calls, we can computer an even better derivative using

$$d_{e,h}f = \frac{4f_2' - f_1'}{3} \qquad f_1' = \frac{f(x+h/2) - f(x-h/2)}{h} \qquad f_2' = \frac{f(x+h/4) - f(x-h/4)}{h/2}.$$

In code, we compute this with the following function:

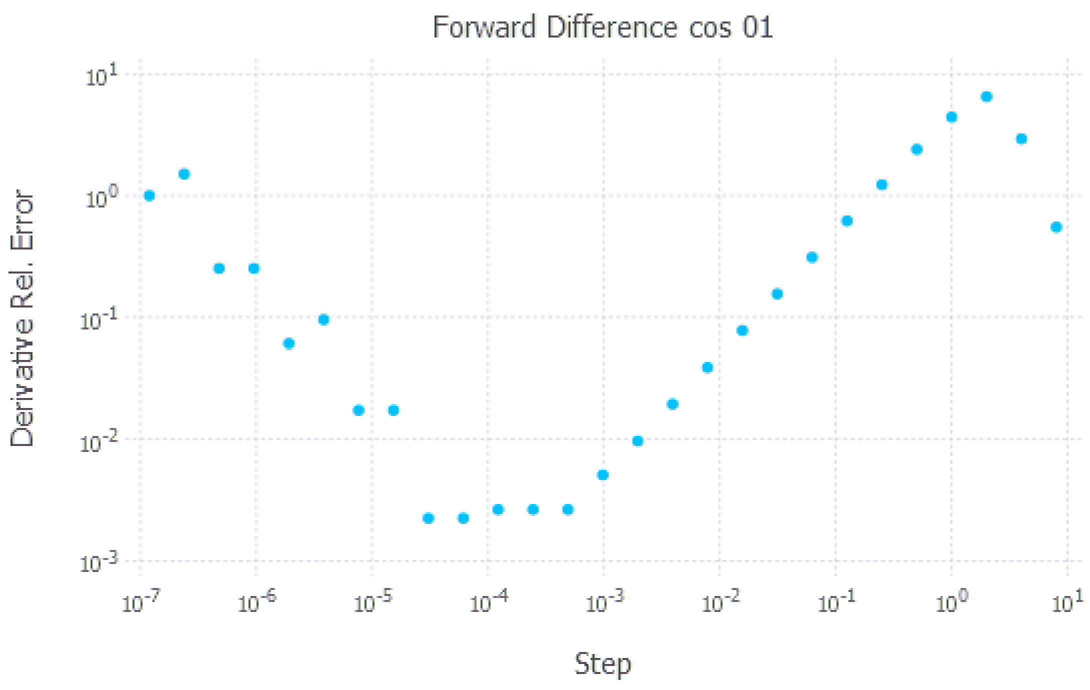
```
#Julia interprets float literals as double precision, so creating sp constants
const THREE_SP = float32(3.0)
const FOUR_SP = float32(4.0)

#Function for single precision derivatice using extrapolated difference algorithm
#  Need to use single precision constants so the expression does not get promoted
function extrapol_diff_derivatice(f, x::Float32, h::Float32)
  return (FOUR_SP*center_diff_derivative(f, x, h/TWO_SP)
          - center_diff_derivative(f, x, h))/THREE_SP;
end
```

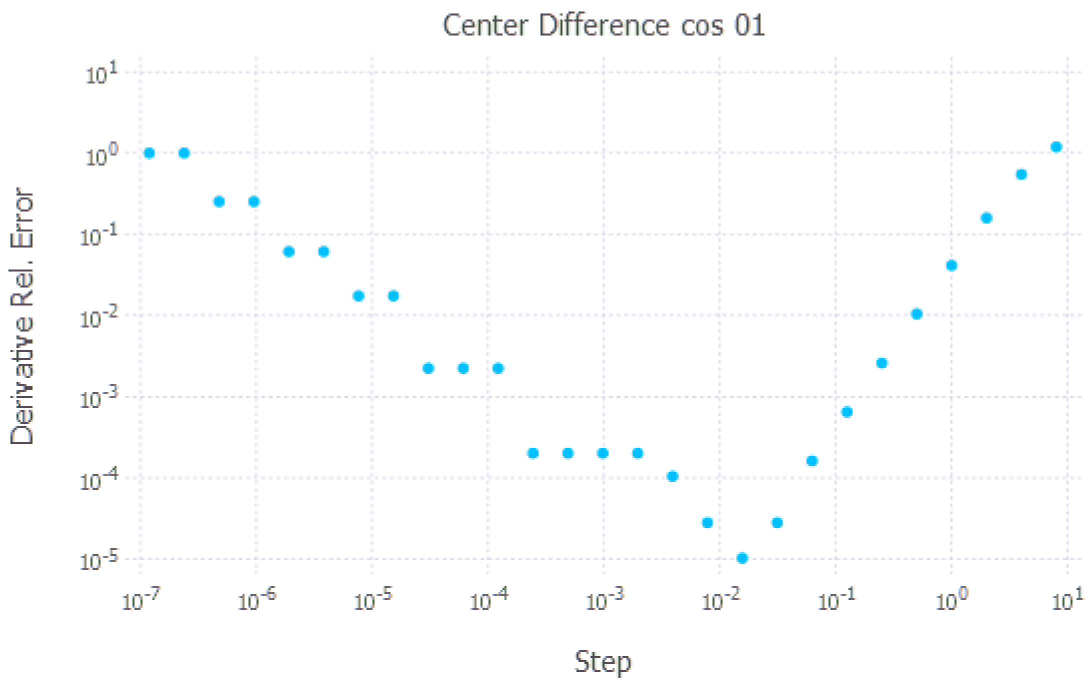
1.2 Printing out derivative and relative error

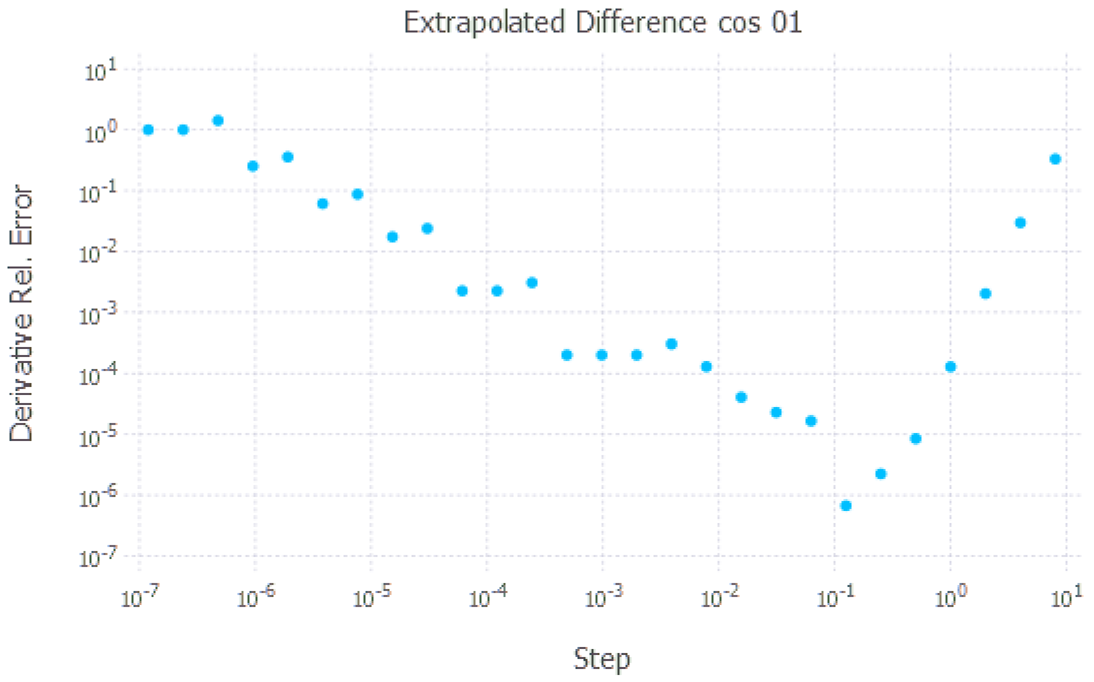
See included file, HW1Q1.out, for the output, which includes the printout of all the computed derivatives and relative errors.

1.3 Log-Log plot of ϵ vs h

Below are figures for the plot of derivatice relative error as a function of step size for each of the difference methods; here I include them only for cosine at 0.1, the rest are in the appendix.


Forward Difference cos 01
Derivative Rel. Error
Step
10^{1}
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-7}
10^{-6}
10^{-5}
10^{-4}
10^{-3}
10^{-2}
10^{-1}
10^{0}
10^{1}


Center Difference cos 01
Derivative Rel. Error
Step
10^{1}
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-7}
10^{-6}
10^{-5}
10^{-4}
10^{-3}
10^{-2}
10^{-1}
10^{0}
10^{1}



According to estimates we found in class, we expect the forward difference algorithm to yield the best relative error, $\epsilon \sim \sqrt{\epsilon_m} \sim 10^{-3.5}$ (where ϵ_m is the machine error), when h (the step) is about $\sqrt{\epsilon_m} \sim 10^{-3.5}$; indeed, we see that ϵ is smallest, about 10^{-3}, when h is about 10^{-4}. Similarly, we find that for the center difference ϵ reaches about 10^{-5} when h is about 10^{-2}, which is in agreement with our estimations of $\epsilon_m^{2/3} \sim 2x10^{-5}$ and $\epsilon_m^{1/3} \sim 5x10^{-3}$, respectively. For the extrapolated difference, we find that the minimum error of $\epsilon \sim 10^{-6}$ occurs when h is about 10^{-1}, which is close to our predictions of $\epsilon_m^{4/5} \sim 2x10^{-6}$ and $\epsilon_m^{1/5} \sim 4x10^{-2}$, respectively.

1.4 Truncation and Roundoff error

We expect to see large truncation errors when our step size is too large; on the other hand, if our step size is too small, we become subject to increasingly large roundoff error as our computatons suffer floating point inaccuracy. Therefore, as we increase our step size (left to right on the plots), we expect to start with some large roundoff error, and slowly decrease our relative error as our step increases until we hit the truncation error, at which point the relative error begins increasing again. In particular, as discussed in class, we expect the roundoff error to scale as $1/h$ in all cases; we know that, in a log-log plot, power rule exponents appear as slopes, so we expect a slope of about -1 in the roundoff error section of our plots. Similarly, we expect a truncation error on the order of h^2, h^3, and h^4, respectively, for the forward, center, and extrapolated differences derivatices, so we should see slopes of 2, 3, and 4 on those plots.

This is pretty close to what we actually observed; we can calculate the approximate slope by looking at two points in each section of the graphs and calculating

$$m \approx \frac{y_2 - y_1}{x_2 - x_1} \qquad x_2 > x_1$$

Below is a table of the observed truncation and roundoff error slopes for each derivative at $cos(0.1)$; the values for each of the other points and functions are also fairly close. The rest of the calculated slopes can be found in the output file, HW1Q1.out.

Derivative Type	Truncation Error Slope	Roundoff Error Slope
Forward	0.8826095	-0.994048
Center	1.7098624	-1.009142
Extrapolated	3.8116312	-1.009142

2 Question 2

For source code, see included file HW1Q2.jl.

2.1 Code

2.1.1 Trapezoid Rule Integration

We obtain an approximation of the integral using the N-point trapezoid rule by calculating

$$\int_a^b f(x)dx \approx h(\frac{f(a)+f(b)}{2} + \sum_{n=1}^{N-2} f(a+nh)) \qquad h := \frac{a-b}{N-1}.$$

We calculate this in code using:

```
#Constant for two in single precision
const TWO_SP = float32(2)

#Function that implements the trapezoid rule for integration
function trap_integral(f, N::Unsigned, x_min::Float32, x_max::Float32)
  h = (x_max - x_min)/(N-1)
  tot = (f(x_min)+f(x_max))/TWO_SP
  for i = 1:N-2
    tot += f(x_min + i*h)
  end
  tot *= h
  return tot
end #trap_integral
```

Note that, while could certainly be sped up (e.g. dynamic walk through indices, parallelization decorators, etc), this is the simplest way to write it.

2.1.2 Simpson Rule integration

We obtain an approximation of the integral using the N-point (for N odd) Simpson rule with

$$\int_a^b f(x)dx \approx \frac{h}{3}((f(a)+f(b)) + \sum_{n=1}^{N-2} c_n f(a+nh)) \qquad h := \frac{a-b}{N-1} \qquad c_n = \begin{cases} 4 & n \, even \\ 2 & n \, odd \end{cases}.$$

In code, this is calculated as:

```
#Constants for three, four in single precision
const THREE_SP = float32(3)
const FOUR_SP = float32(4)

#Function that implements the Simpson rule for integration
function simp_integral(f, N::Unsigned, x_min::Float32, x_max::Float32)
  h = (x_max - x_min)/(N-1)
  tot = (f(x_max) + f(x_min))
  for i = 1:2:N-2
      tot += FOUR_SP * f(x_min + i*h)
  end
  for i = 2:2:N-3
      tot += TWO_SP * f(x_min + i*h)
  end
  tot *= h / THREE_SP
  return tot
end #simp_integral
```

2.1.3 Gauss-Legendre Quadrature

To calculate the N-point Gauss-Legendre Quadrature approximation of the integral, we use

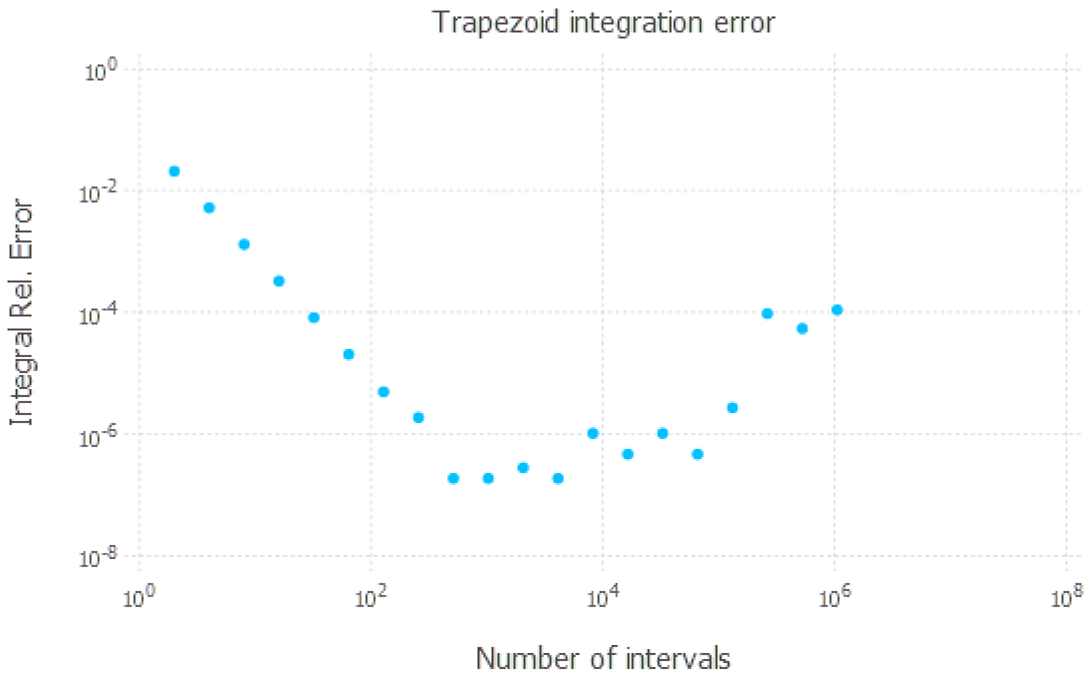
$$\int_a^b f(x)dx \approx \frac{b-a}{2}\sum_{n=1}^{N} w_n f(\frac{b-a}{2}x_n + \frac{b+a}{2})$$

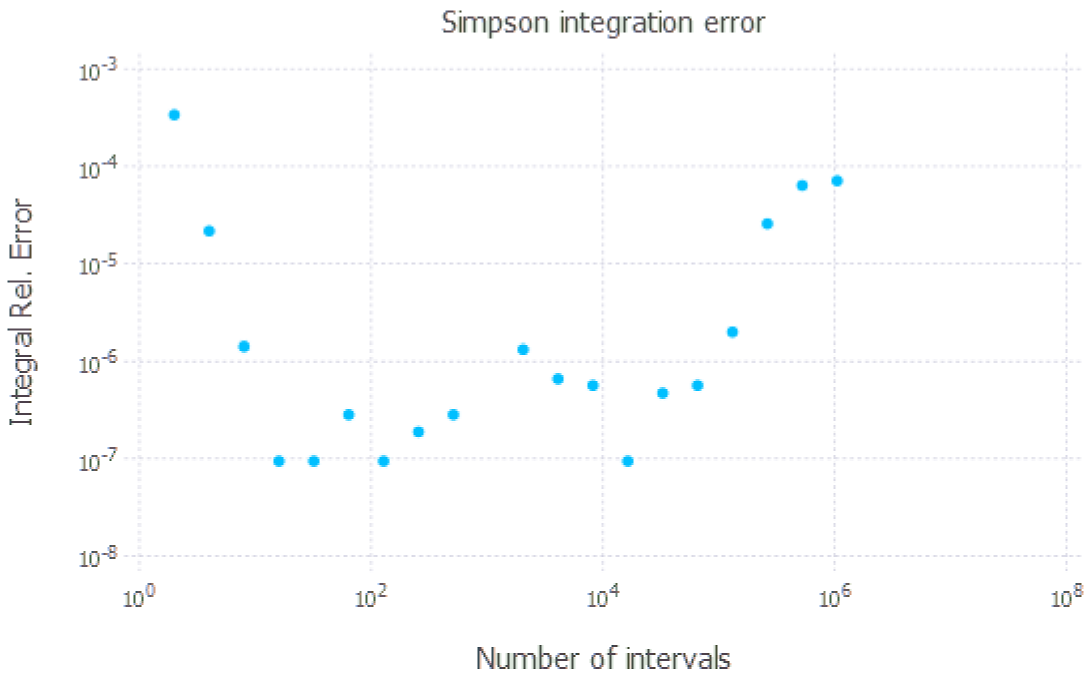
Where the x_n are the roots of the N-th Legendre polynomial and the w_n are their associated weights. In code, we compute this with (Note that Base.Gauss(x,y) returns the y-point positions and weights typecast to x type):

```
#Function that implements the Gaussian-Legendre Quadrature
function gauss_integral(f, N::Unsigned, x_min::Float32, x_max::Float32)
  alpha = (x_max - x_min)/TWO_SP
  beta = (x_max + x_min)/TWO_SP
  X, W = Base.gauss(Float32, N)
  tot = float32(0)
  for (x,w) in zip(X,W)
    tot += w*f(alpha*x + beta)
  end
  tot *= alpha
end #gauss_integral
```

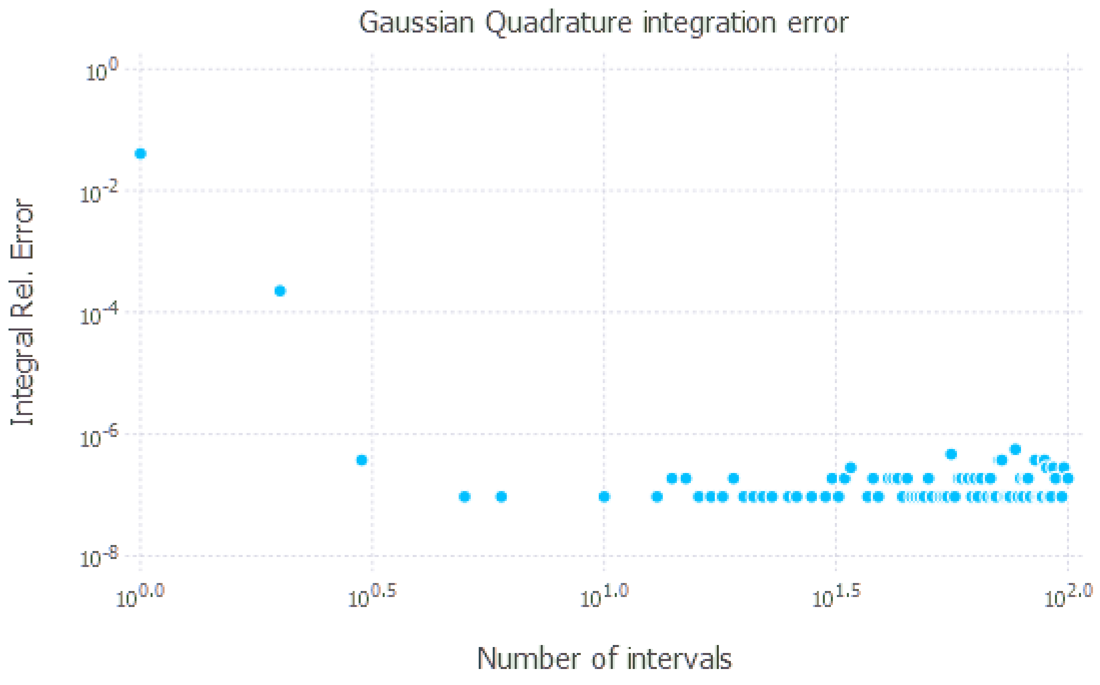
2.2 Graphs

Here are the graphs of relative error for the Trapezoid and Simpson integration rules, for N equal to powers of two from 1 to 20 plus 1.





Here is the graph of the relative error for the Gauss-Legendre quadrature, for N between 1 and 100.



2.3 Graph Explanation

For both the Simpson and Trapezoid integration rules, we start with a large amount of error at a small number of intervals, which decreases as the number increases. This region of truncation error should follow $\epsilon \propto \frac{1}{N^2}$ for the trapezoid rule and $\epsilon \propto \frac{1}{N^4}$ for the Simpson rule, as discussed in class. This produces the characteristic -2 and -4 slopes we see in the small number region of the log-log graphs. This continues until the error hits some optimal point, predicted to be 600 for trapezoid rule and 30 for Simpson's rule. In our graphs, we hit the optimum at 512 for trapezoid rule and 16 for Simpson's rule, which are close. At this point, roundoff error takes over for each of them, which should scale as $\frac{1}{\sqrt{N}}$; we see this in the noisy slope of 1/2 on the large number portion of both graphs.

For the Gaussian Quadrature error, since the approximation is based on polynomials, we expect the truncation error to follow the error in approximating e^{-x} as a polynomial. In particular, since it is exact for polynomials up to the $2N-1$ order, we expect to see error from the $2N$ term of the polynomial expansion of e^{-x}. The $2N$ term of the polynomial expansion is proportional to x^{2N}, which is less than 1 in our interval, times $\frac{1}{(2N)!}$. Therefore, we expect to see truncation error on the order of $O(\frac{1}{(2N)!})$. This means we expect negligible error (less than machine precision) when $(2N)!$ is about 10^7 $(\frac{1}{\epsilon_m})$, which occurs at about $N = 5$. In our graph, we actually see the error disappear at $N = 4$, but this is due to our liberal estimation of the error. As we continue increasing the number of intervals, we begin to see roundoff error when $N > 10$, which again slowly increases at a rate of about $\frac{1}{\sqrt{N}}$.

3 Question 3

For source code, see included file HW1Q3.jl.

3.1 Code

Linear Congruential Generators are defined by the recurrence relation

$$r_{i+1} = (Ar_i + C)\%M,$$

where A, C, and M are constants of the generator. We get a pseudo-random number between 0 and 1 by dividing r_i by M. Then, to get a random walk, we simply compare a random number from the LCG to 0.5 and appropriately increment or decrement the position

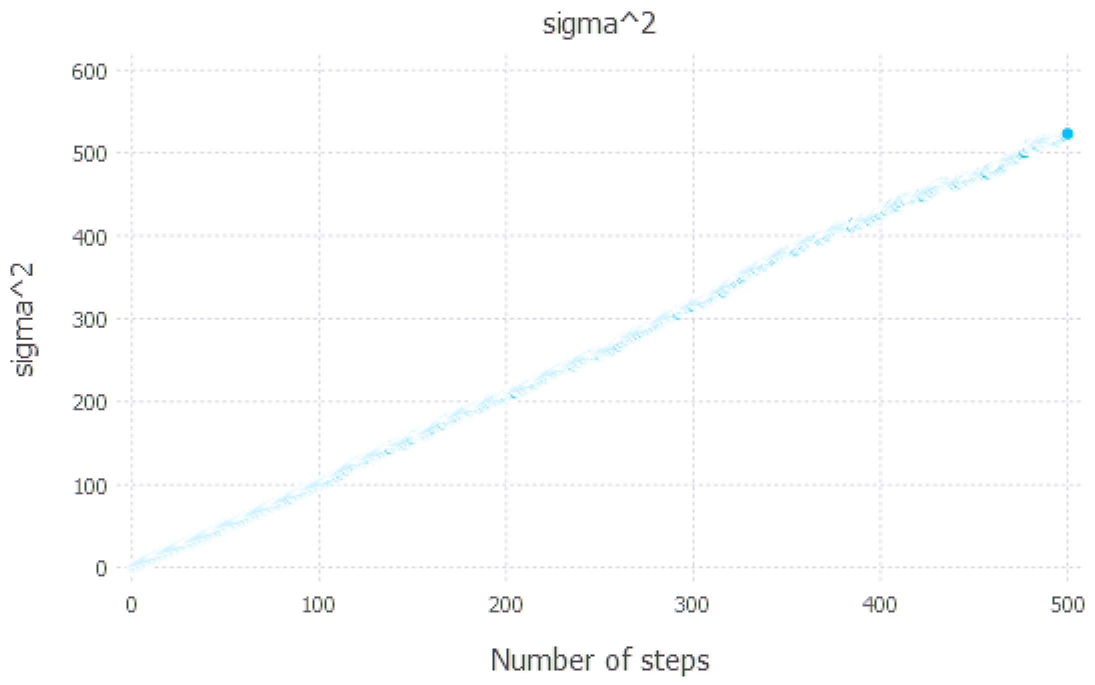
For our random number generator, we take a seed value and return a closure to iteratively produce the next random number between 0 and 1.

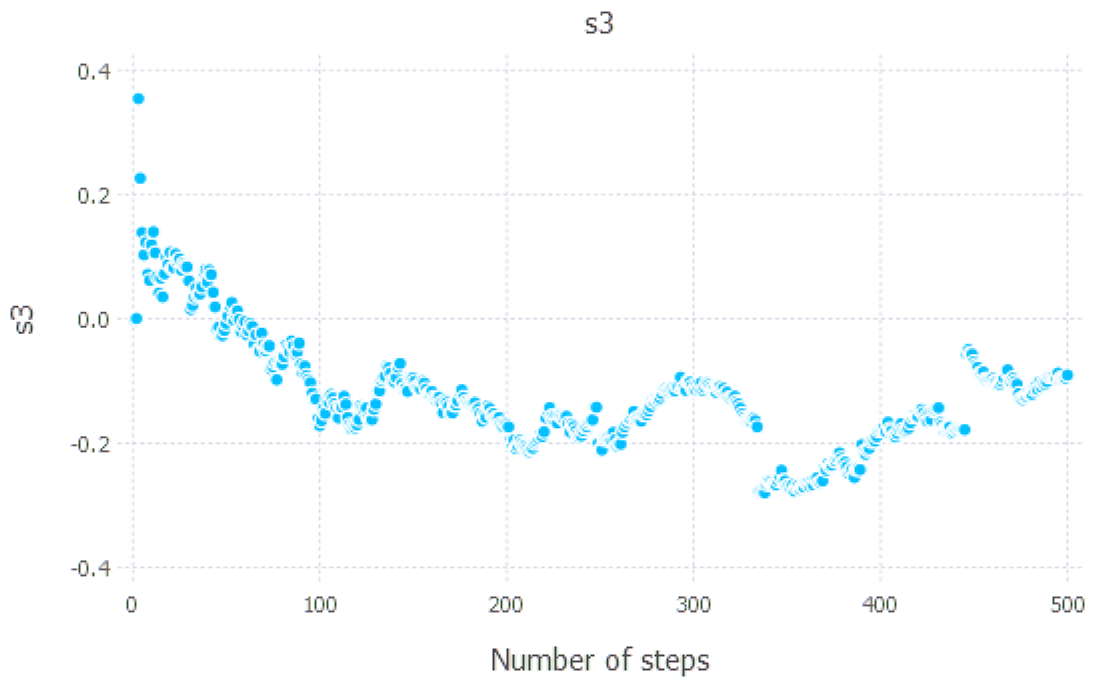
```julia
#Defining constants of the LCG
const A = 9301
const C = 49297
const M = 233280

#Random number generator generator, using LCG algo
function random_gen(seed::Integer)
  state = seed
  function random()
    state = (A*state + C) % M
    return state / float32(M) #Need to promote operation to floating point
  end #random
  return random
end #random_gen
```

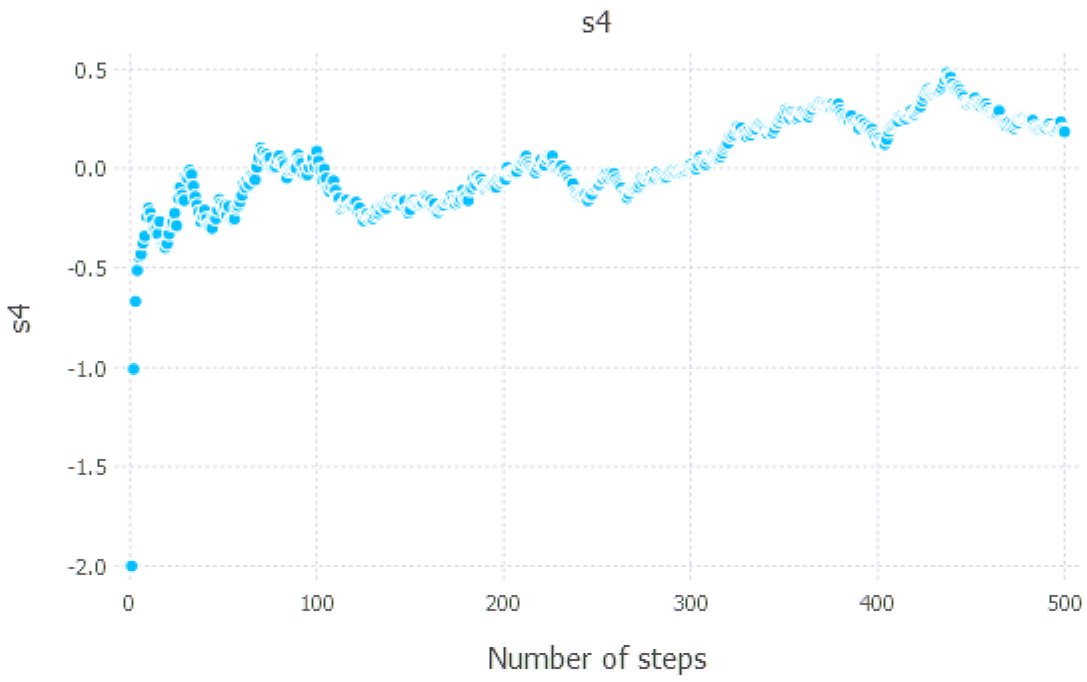
To get a random walk from this, we create such a closure and then create a producer which polls the RNG, appropriately increases or decreases the position, and then yields the position. To get it in generator syntax, we wrap it with a task decorator and return.

```julia
#Random walk function; iteratively produces position from +-1 random walk
function random_walk(seed::Integer, n_max::Integer)
  next = random_gen(seed) #Get an RNG
  x = 0 #Start the walk at 0
  function walk()
    for step = 1:n_max
      if next() > 0.5 #50/50 chance to walk forward or back
        x += 1
      else
        x -= 1
      end
      produce(x) #yield the current position
    end
  end
  @task walk() #pre-wrap the producer in a Task
end #random_walk
```

3.2 Plots


sigma^2
600
500
400
300
200
100
0
sigma^2
0
100
200
300
400
500
Number of steps


s3
0.4
0.2
0.0
-0.2
-0.4
s3
0
100
200
300
400
500
Number of steps



3.3 Plot Explanations

3.3.1 σ_n^2

We explain this behavior by first setting up the recurrence relations for each of the calculated values. First, we look at σ_n^2, the value of σ^2 after n steps:

$$\sigma_n^2 := \langle x_n^2 \rangle = \langle (x_{n-1}+l)^2 \rangle = \langle x_{n-1}^2 + 2lx_{n-1} + l^2 \rangle,$$

where l is ± 1 depending on the step. By distribution of the expectation over addition and multiplication of independent variables and the fact that l has an expectation of 0 while l^2 has an expectation of 1, we see that

$$\sigma_n^2 = \langle x_{n-1}^2 \rangle + 0 + 1 = \sigma_{n-1}^2 + 1.$$

This predicts the linear plot of slope 1 that we see in the σ^2 plot, namely

$$\sigma_n^2 = n.$$

3.3.2 s_3

Similarly, for s_3, we have

$$s_3 := \frac{\langle x_n^3 \rangle}{\sigma_n^3} = \frac{\langle (x_{n-1}+l)^3 \rangle}{\sigma_n^3} = \frac{\langle x_{n-1}^3 + 3lx_{n-1}^2 + 3l^2 x_{n-1} + l^3 \rangle}{\sigma_n^3} = \frac{\langle x_{n-1}^3 \rangle + 3\langle x_{n-1} \rangle}{\sigma_n^3}$$

However, since we start with $\langle x_0 \rangle = \langle x_0^3 \rangle = 0$, this means that we should just always have

$$s3 = 0.$$

This agrees with the intuition that, for any given walk, by symmetry the probability of ending up at x and $-x$ are the same for any x; this means the probability density for x_n^3 is symmetric about 0 and so it expectation values should always be 0. Although we see some drift, we notice that s_3 hovers around 0; this seems reasonable, given how infamously broken LCG generators are.

3.3.3 s_4

Again, we start by calculating:

$$s_4 := \langle \frac{\langle x_n^4 \rangle}{\sigma_n^4} - 3 \rangle = \langle \frac{\langle (x_{n-1} + l)^4 \rangle}{\sigma_n^4} - 3 \rangle = \langle \frac{\langle x_{n-1}^4 \rangle + 6\langle x_{n-1}^2 \rangle + 1}{\sigma_n^4} - 3 \rangle,$$

This does not immediately reveal the general behavior of the system, but we can try another trick; let

$$x_n = \sum_{i=1}^{n} l_i \qquad l_i = \pm 1.$$

Then

$$\langle x_n^4 \rangle = \langle (\sum_{i=1}^{n} l_i)^4 \rangle.$$

Since the l_i are independent, we can distribute the expectation across summation and multiplication. We know that any factor with an even power of an l_i will have an expectation of 1, and any with an odd power will have an expectation of 0. We also note that there are six (four choose two) ways to obtain any mixed terms of second order each.

$$\langle x_n^4 \rangle = \sum_{i=1}^{n} \langle l_i^4 \rangle + 6 \sum_{i=1}^{n} \sum_{j=1, j \neq i}^{n} \langle l_i^2 \rangle \langle l_j^2 \rangle = \sum_{i=1}^{n} 1 + 6 \sum_{i=1}^{n} \sum_{j=1, j \neq i}^{n} 1.$$

The first sum clearly gives us n; the second term gives us n choose two ($\frac{n(n-1)}{2}$), so we get

$$\langle x_n^4 \rangle = n + 3n(n-1)$$

We also know that

$$\sigma_n^2 = n \qquad \sigma_n^4 = n^2$$

so

$$s_4 := \langle \frac{\langle x_n^4 \rangle}{\sigma_n^4} - 3 \rangle = \langle \frac{n + 3n(n-1)}{n^2} - 3 \rangle$$

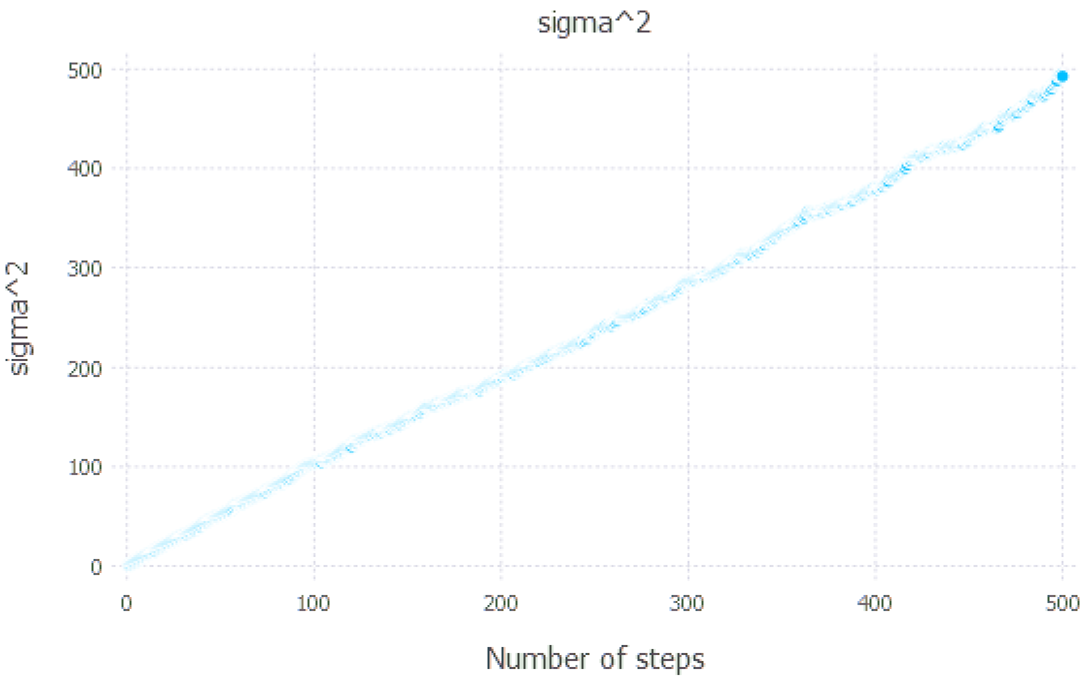
And so, in the large n limit

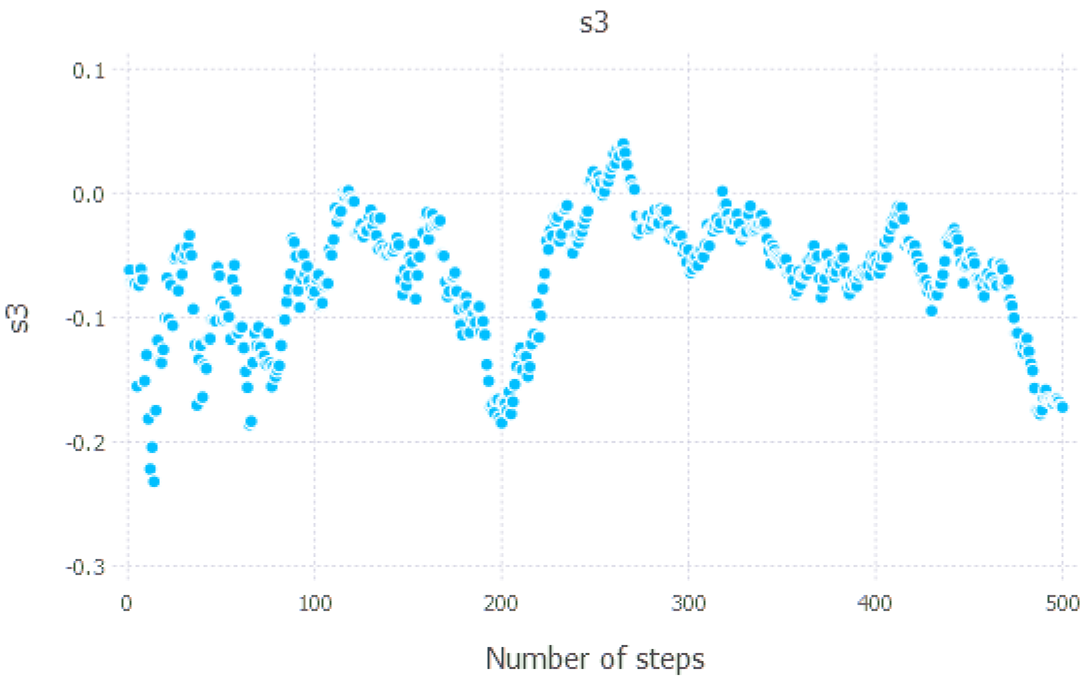
$$s_4 = \frac{1}{n} + 3 - \frac{3}{n} - 3 \approx 0.$$

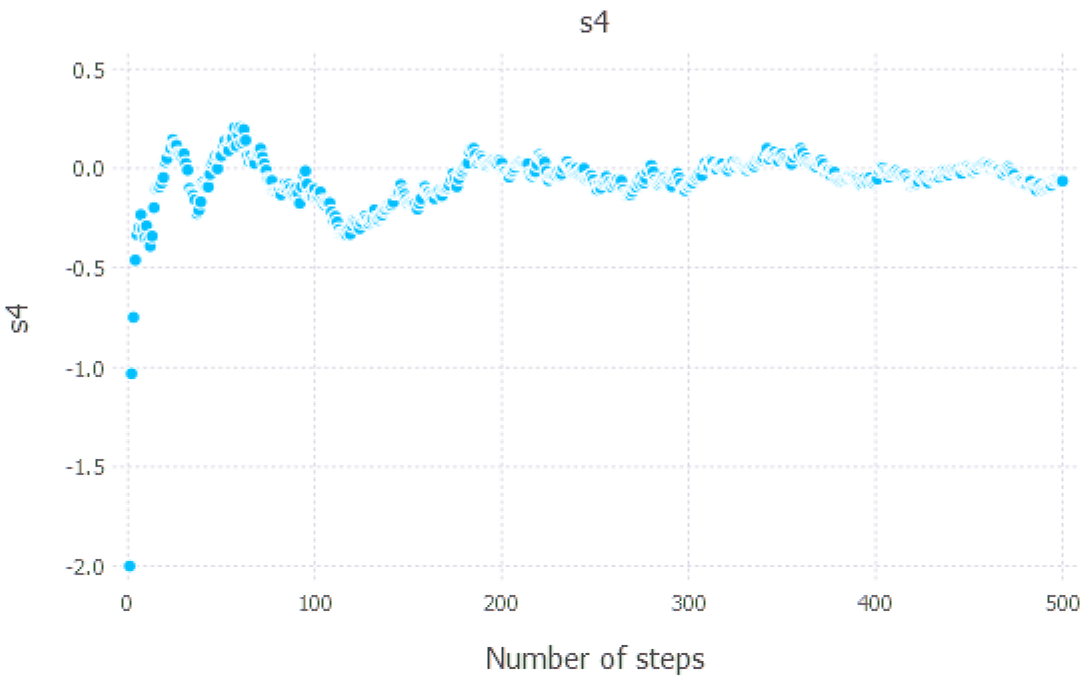
This behavior is pretty close to what we observe, although there is clearly some drift due to the non-ideal behavior of the LCG RNG.

3.4 Mersenne Twister

This wasn't really part of the assignment, but I was interested in comparing the LCG to a "modern" PRNG. Early random number generators such as LCG's were infamously bad, so much so that terrible random numbers have become something of an in-joke in the software community (see this xkcd). The Mersenne Twister, invented in 1997 as part of a renewed interest in random number generation, is considered the current "standard" for good random number generation, offering an excellent trade off between speed and quality. In particular, it was one of the first RNGs to pass the die-hard tests; it is also the default RNG in Julia (as opposed to, say, gcc, which still uses a beefed up LCG). I executed the same code, only replacing my random number generator with the default `rand` funtion in Julia. I got the following graphs; I was surprised to find only a marginal improvement on s_3 over the LCG. However, in the s_4 graph, the Mersenne Twister clearly shows better asymptotic behavior.

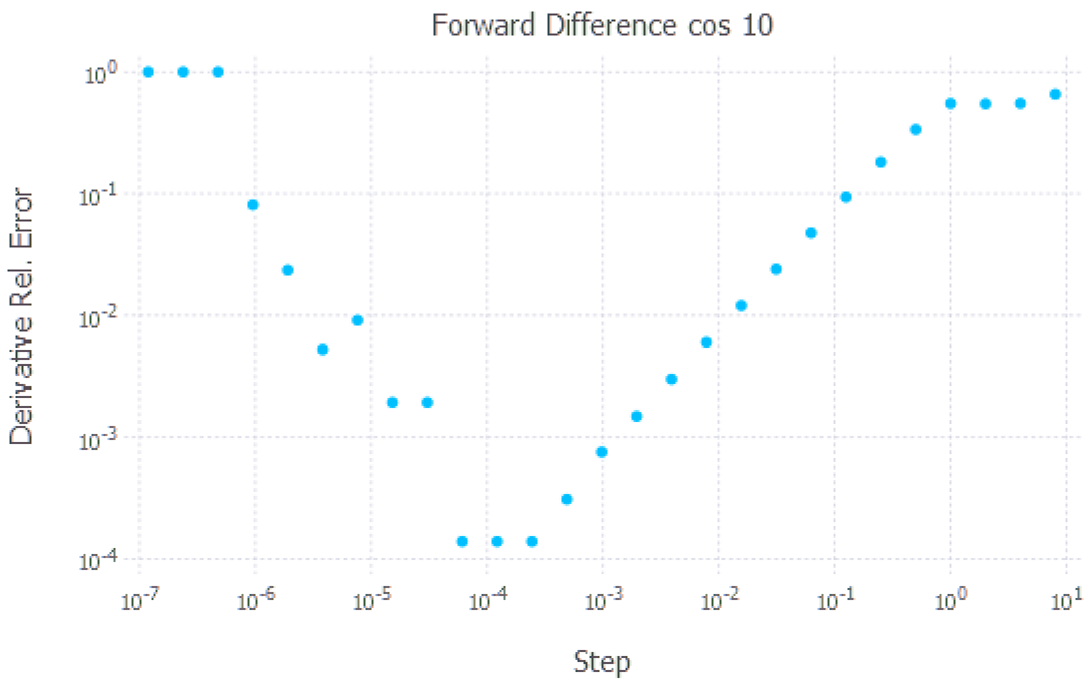

sigma^2
sigma^2
Number of steps
0
100
200
300
400
500
0
100
200
300
400
500

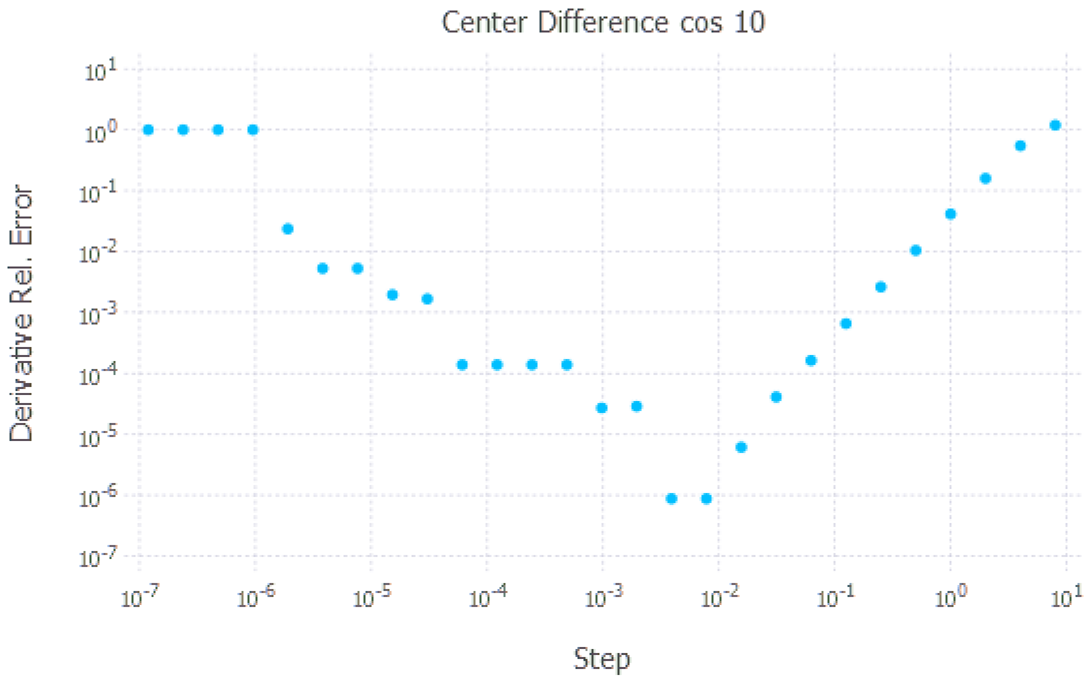

s3
s3
Number of steps
0.1
0.0
-0.1
-0.2
-0.3
0
100
200
300
400
500

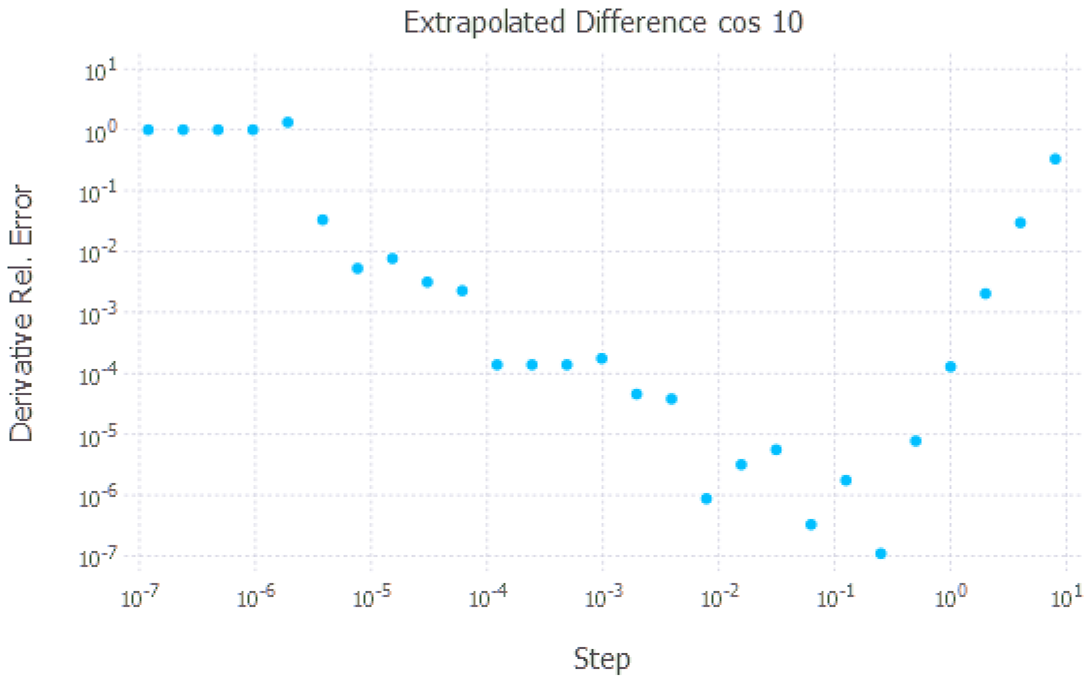

s4
0.5
0.0
-0.5
-1.0
-1.5
-2.0
s4
0
100
200
300
400
500
Number of steps

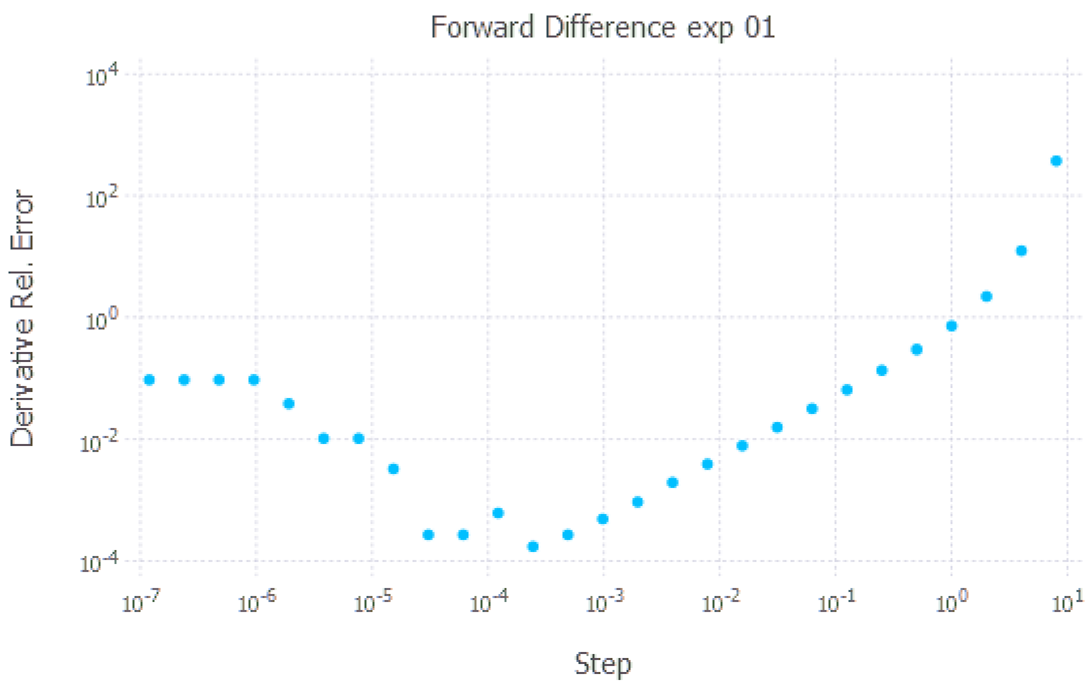
Appendix

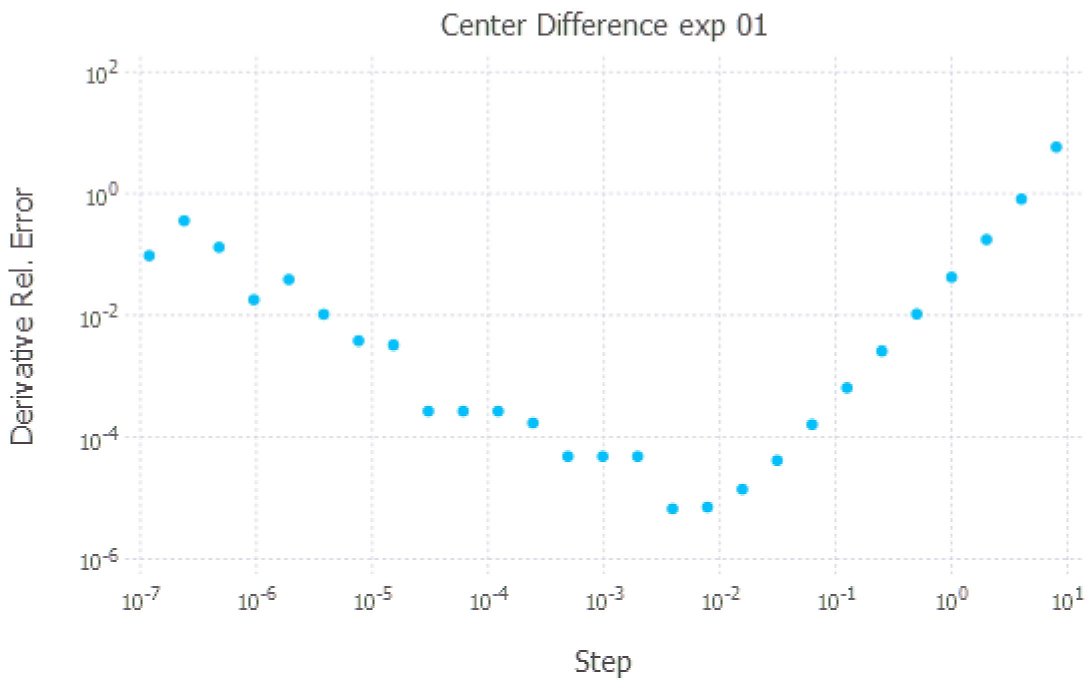
Question 1 graphs

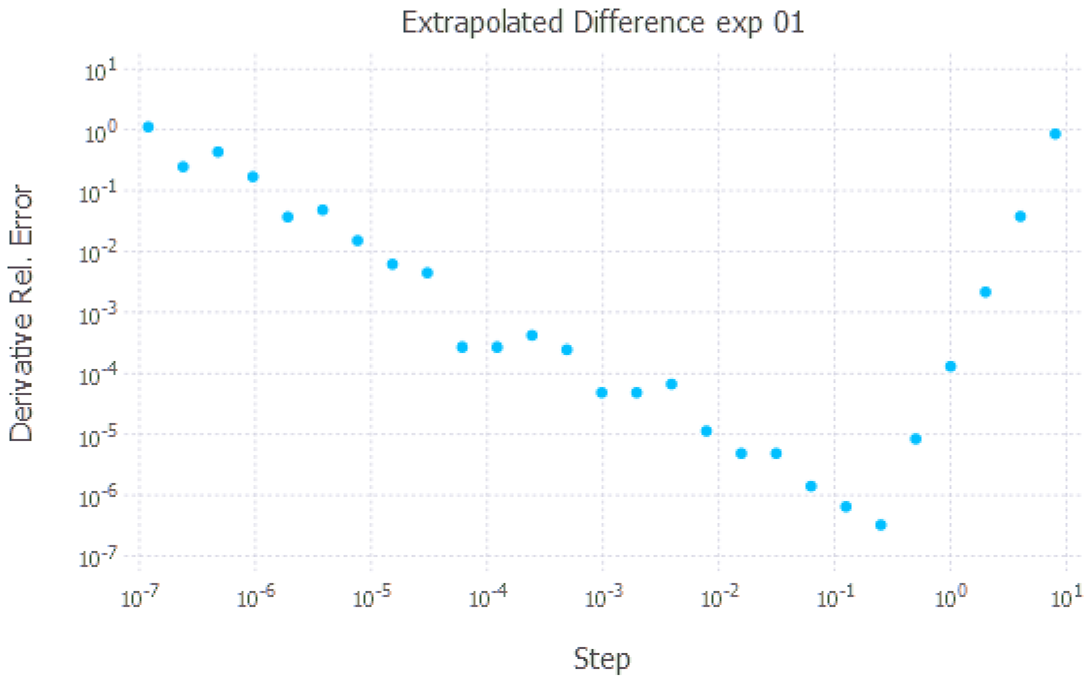


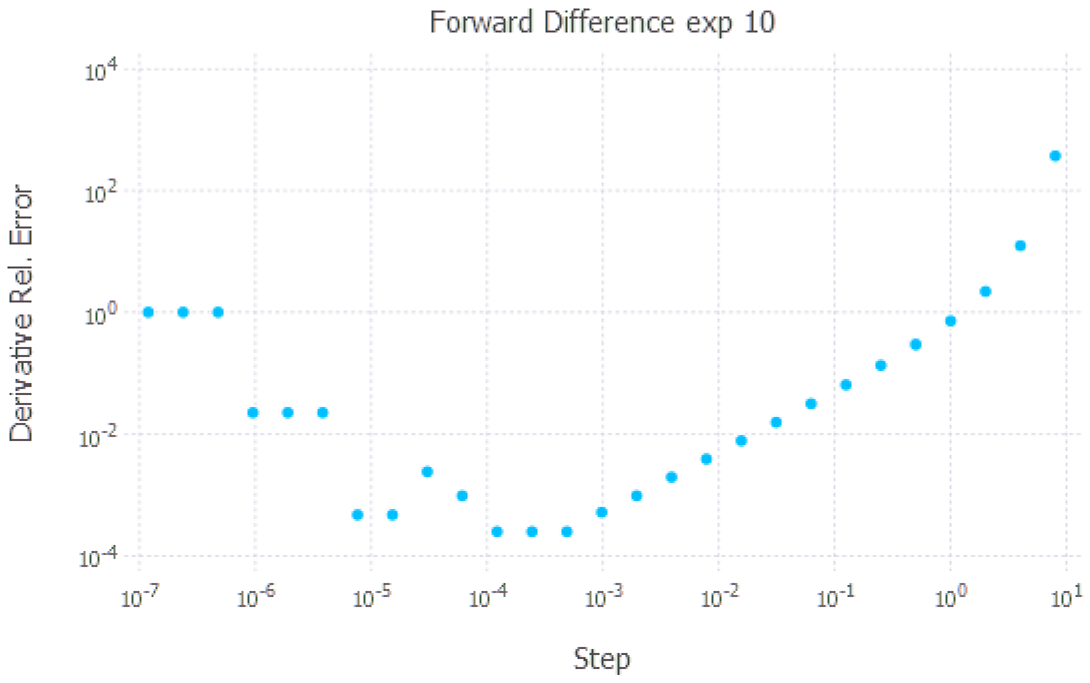

Center Difference cos 10
Derivative Rel. Error
10^{1}
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
10^{-7}
10^{-6}
10^{-5}
10^{-4}
10^{-3}
10^{-2}
10^{-1}
10^{0}
10^{1}
Step

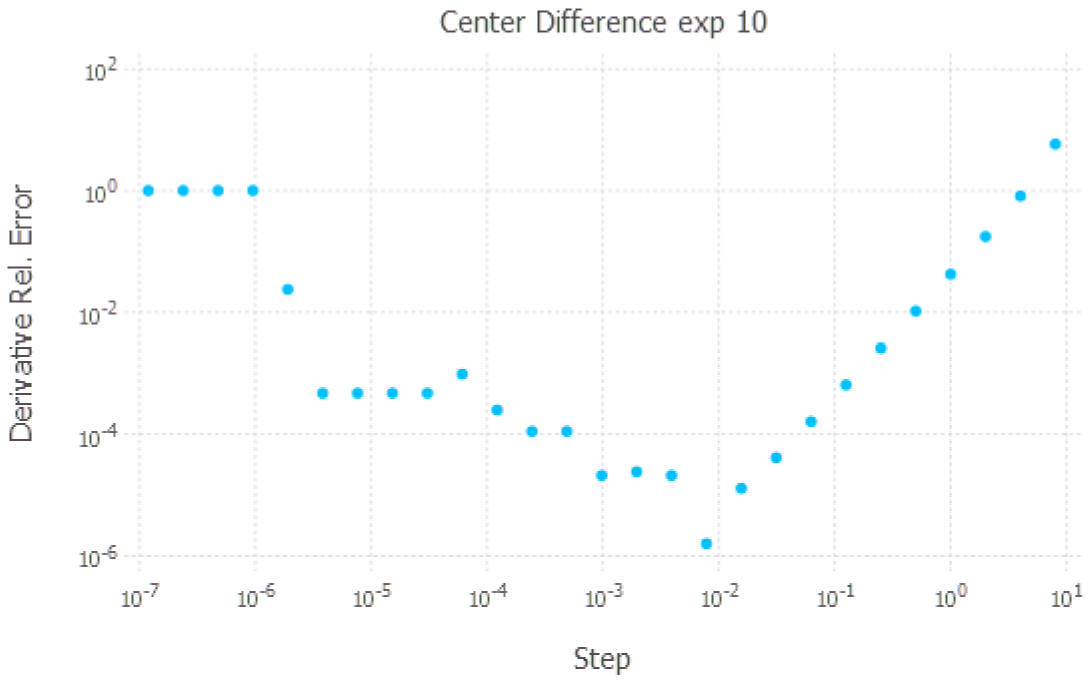

Extrapolated Difference cos 10
Derivative Rel. Error
10^{1}
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
10^{-7}
10^{-6}
10^{-5}
10^{-4}
10^{-3}
10^{-2}
10^{-1}
10^{0}
10^{1}
Step

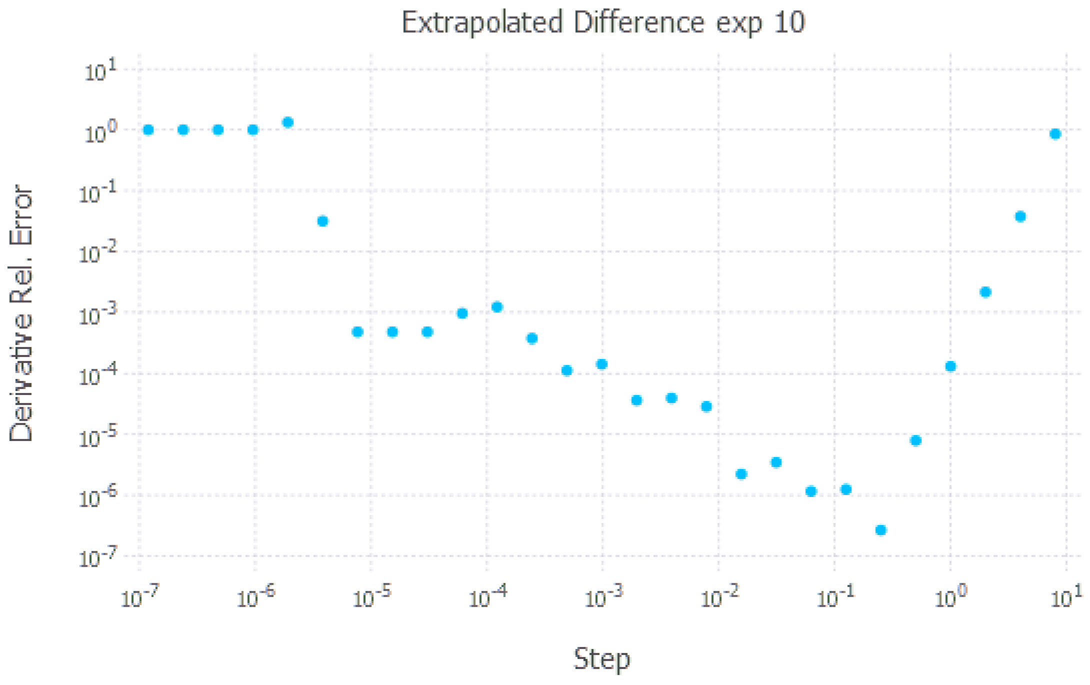

Forward Difference exp 01
Derivative Rel. Error
10^4
10^2
10^0
10^{-2}
10^{-4}
10^{-7}
10^{-6}
10^{-5}
10^{-4}
10^{-3}
10^{-2}
10^{-1}
10^0
10^1
Step


Center Difference exp 01
Derivative Rel. Error
10^2
10^0
10^{-2}
10^{-4}
10^{-6}
10^{-7}
10^{-6}
10^{-5}
10^{-4}
10^{-3}
10^{-2}
10^{-1}
10^0
10^1
Step


Extrapolated Difference exp 01
Derivative Rel. Error
10^{1}
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
10^{-7}
10^{-6}
10^{-5}
10^{-4}
10^{-3}
10^{-2}
10^{-1}
10^{0}
10^{1}
Step


Forward Difference exp 10
Derivative Rel. Error
10^{4}
10^{2}
10^{0}
10^{-2}
10^{-4}
10^{-7}
10^{-6}
10^{-5}
10^{-4}
10^{-3}
10^{-2}
10^{-1}
10^{0}
10^{1}
Step


Center Difference exp 10
Derivative Rel. Error
Step
10^{2}
10^{0}
10^{-2}
10^{-4}
10^{-6}
10^{-7}
10^{-6}
10^{-5}
10^{-4}
10^{-3}
10^{-2}
10^{-1}
10^{0}
10^{1}


Extrapolated Difference exp 10
Derivative Rel. Error
Step
10^{1}
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
10^{-7}
10^{-6}
10^{-5}
10^{-4}
10^{-3}
10^{-2}
10^{-1}
10^{0}
10^{1}