Problem Set 1 Solutions

1. For $\boldsymbol{x}$, an arbitrary $n \times 1$ column vector, and an $n \times n$ matrix $A = [A_{ij}]$, $\mathbf{x}^T A\mathbf{x}$ is a scalar, which is called a *quadratic form*. Express $\mathbf{x}^T A\mathbf{x}$ in terms of the components of A and $\boldsymbol{x}$. (Hint: $\mathbf{x}^T A\mathbf{x} = \mathbf{x} \cdot (A\mathbf{x})$)

   $$\boldsymbol{x}^\intercal A\boldsymbol{x} = \boldsymbol{x} \cdot (A\boldsymbol{x})$$

   $$= \sum_{i=1}^{n} x_i (A\boldsymbol{x})_i$$

   $$= \sum_{i=1}^{n} x_i \sum_{j=1}^{n} A_{ij} x_j$$

   $$= \sum_{i=1}^{n} \sum_{j=1}^{n} x_i A_{ij} x_j$$

2. If A is a symmetric $n \times n$ matrix and B is $n \times m$, prove that $B^T AB$ is a symmetric $m \times m$ matrix.

   First of all, $B^\intercal A$ is an $m \times n$ matrix multiplied by an $n \times n$ matrix, so it is a $m \times n$ matrix. Then multiplying this by B, an $n \times m$ matrix, we get an $m \times m$ matrix.
   To show that $B^\intercal AB$ is a symmetric matrix, we need to show that $(B^\intercal AB)_{ij} = (B^\intercal AB)_{ji}$:

   $$(B^\intercal AB)_{ij} = \sum_{k=1}^{n} (B^\intercal A)_{ik} B_{kj}$$

   $$= \sum_{k=1}^{n} \left(\sum_{p=1}^{n} B^\intercal_{ip} A_{pk} \right) B_{kj} = \sum_{k=1}^{n} \sum_{p=1}^{n} B^\intercal_{ip} A_{pk} B_{kj}$$

   $$= \sum_{k=1}^{n} \sum_{p=1}^{n} B_{pi} A_{pk} B^\intercal_{jk}$$

   where we have used the definition of a transpose matrix and that $B^{\intercal\intercal} = B$.

   $$= \sum_{k=1}^{n} \sum_{p=1}^{n} B^\intercal_{jk} A_{kp} B_{pi}$$

   where we have switched the order of multiplication and used that A is symmetric so that $A_{pk} = A_{kp}$.

   $$= \sum_{p=1}^{n} \left(\sum_{k=1}^{n} B^\intercal_{jk} A_{kp} \right) B_{pi}$$

   where we have switched the order of the summations.

$$= \sum_{p=1}^{n} (B^\intercal A)_{jp} B_{pi}$$

$$= (B^\intercal AB)_{ji}$$

So we have $(B^\intercal AB)_{ij} = (B^\intercal AB)_{ji}$, and it is therefore symmetric.$\square$

3. Prove that $(AB)^T = B^T A^T$ for $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{n\times p}$.

   $((AB)^\intercal)_{ij} = (AB)_{ji}$

   $$= \sum_{k=1}^{n} A_{jk} B_{ki}$$

   $$= \sum_{k=1}^{n} A^\intercal_{kj} B^\intercal_{ik}$$

   $$= \sum_{k=1}^{n} B^\intercal_{ik} A^\intercal_{kj}$$

   $$= (B^\intercal A^\intercal)_{ij}$$

   So we have $((AB)^\intercal)_{ij} = (B^\intercal A^\intercal)_{ij}$ for arbitrary i, j. Therefore, $(AB)^\intercal = B^\intercal A^\intercal$.

4. Assume we have a floating point number system, represented in binary, which allocates 1 bit to sign, 4 bits to the mantissa (q) and 4 bits to the characteristic(c), and our numbers are of the form $\pm(1+q)(2^{c-7})$

   (a) What is machine epsilon?

   The gap between floating point numbers that have an exponent of m is

   $(.0001)(2^m) = 2^{-4}(2^m) = 2^{m-4}$, where I am using binary representation of q.

   The relative gap is therefore $\dfrac{2^{m-4}}{|x|} \leq \dfrac{2^{m-4}}{(1+.0000)(2^m)} = 2^{-4}$.

   Therefore, $\epsilon_{mach} = 2^{-4} \approx 6.25(10)^{-2}$.

   (b) Find the floating point representation of $\frac{1}{7}$. Also include the binary representation for q and c.

   $\dfrac{1}{7} = 2^{-3} + 2^{-6} + 2^{-9} +$ higher order terms

$$\implies \frac{1}{7} = (1 + 2^{-3} + 2^{-6} + ...)(2^{-3})$$

$$\implies fl\left(\frac{1}{7}\right) = (1 + .0010)(2^{-3})$$

(i.e. $q = .0010, m = -3$ so $c = 4$ and in binary, $c = 0010$)

$$fl\left(\frac{1}{7}\right) = 0.140625$$

(c) Find the floating point representation of π. Also include the binary representation for q and c.

Using the same steps as above, we get

$\pi \approx 3.141592654 = 2^1 + 2^0 + 2^{-3} +$ higher order terms

$$\implies \pi = (1 + 2^{-1} + 2^{-4} + ...)(2^1)$$

$$\implies fl(\pi) = (1 + .1001)(2^1)$$

($q = .1001, m = 1, c = 8,$ or, in binary, $c = 0001$)

$$\implies fl(\pi) = 3.125$$

(d) Comment on the relative errors of your representations from (b) and (c).

The relative error in (b) is given by

$$\frac{|fl(\frac{1}{7}) - \frac{1}{7}|}{\frac{1}{7}} \approx \frac{0.002232}{\frac{1}{7}} \approx 0.015625 \approx 1.6(10)^{-2}$$

The relative error in (c) is given by

$$\frac{|fl(\pi) - \pi|}{\pi} \approx \frac{0.01659}{\pi} \approx 0.00528 \approx 5(10)^{-3}$$

Additionally, we see $\frac{1}{7} \approx 0.142857$ and $fl(\frac{1}{7}) = 0.140625$. Comparing these, we see 2 digits of accuracy.

And $\pi \approx 3.14159$ and $fl(\pi) = 3.125$, again demonstrating 2 digits of accuracy.

Both the relative errors and the direct comparison of the numbers with their floating point representations demonstrates about 2 digits of accuracy. And since $\epsilon_{mach} \approx 6(10^{-2})$ and hence the rounding unit, $\eta \approx 3(10)^{-2}$, 2 digits of accuracy is about what we should expect.

5. Near certain values of x, each of the following functions cannot be accurately computed using the formula as given due to cancellation error. Identify the values of x and propose a refor-

mulation to remedy the problem.

(a) $f(x) = 1 + \cos x$

We would get cancellation error near $x = \pi \pm 2\pi n$.

Using a trig identity, we can get a reformulation given by $f(x) = 2\cos^2\left(\frac{x}{2}\right)$.

(b) $f(x) = 1 - 2\sin^2 x$

Looking for where $|\sin x| = \dfrac{1}{\sqrt{2}}$, we see we would get cancellation error near $x = \dfrac{\pi}{4} \pm \dfrac{\pi}{2} n$.

Again using a trig identity, we can get a reformulation given by $f(x) = \cos 2x$.

(c) $f(x) = \ln(x) - 1$

We would get cancellation error near $x = e$.

Using log properties, we can reformulate as $f(x) = \ln(x) - \ln(e) = \ln\left(\frac{x}{e}\right)$

6. (a) For which positive integer(s) α can the number $5 + 2^{-\alpha}$ be represented *exactly* in double precision floating point arithmetic?

$5 + 2^{-\alpha} = (1 + .0100....01)(2^2)$, where the last 1 is in the $\alpha + 2$ place.

For $fl(5 + 2^{-\alpha}) = 5 + 2^{-\alpha}$ in double precision arithmetic, we need that last 1 to be in the 52^{nd} place or sooner.

In other words, we need $2^{-\alpha-2} \geq \epsilon_{mach} = 2^{-52}$ or $\alpha + 2 \leq 52$.

Therefore, α can be any integer between and including 1 and 50.

(b) Find the largest integer α for which $fl(19 + 2^{-\alpha}) > fl(19)$ in double precision floating point arithmetic.

$19 = 2^4 + 2 + 1 = (1 + .0011)(2^4)$

Therefore, $19 + 2^{-\alpha} = (1 + .001100...1)(2^4)$, where the last 1 is in the $\alpha + 4$ place.

For $fl(19 + 2^{-\alpha}) > fl(19)$, we need the last 1 to be within the 52^{nd} place, so we need $\alpha + 4 \leq 52$ or $\alpha \leq 48$.

7. (a) What is the relative condition number of evaluation of the function $f(x) = e^{\cos x}$ at the point x. About how many digits of accuracy would you expect to lose when performing this operation at $x = 1000$?

The relative condition number is given by $\kappa(x) = \dfrac{|f'(x)||x|}{|f(x)|} = \dfrac{|\sin x||e^{\cos x}||x|}{|e^{\cos x}|} = |\sin x||x|$

So $\kappa(1000) \approx 8.3(10)^2$, so we would expect to lose about 3 digits of accuracy when evaluating this function at $x = 1000$.

(b) Suppose that f and g are continuously differentiable functions. Let $\kappa_f(x)$ denote the condition number of evaulation of the function f at x and similarly for $\kappa_g(x)$ Find a relationship between the condition number of evaluation of $h(x) = f(x)g(x)$ and κ_f and κ_g.

$$\kappa_h(x) = \frac{|f(x)g'(x) + g(x)f'(x)||x|}{|f(x)||g(x)|}$$

$$\le \Big(\frac{|f(x)||g'(x)| + |g(x)||f'(x)|}{|f(x)||g(x)|}\Big)|x|$$

$$= \frac{|g'(x)||x|}{|g(x)|} + \frac{|f'(x)||x|}{|f(x)|}$$

$$= \kappa_f + \kappa_g$$

Therefore, the relationship is $\kappa_h \le \kappa_f + \kappa_g$, and this bound could allow us to estimate the number of digits of accuracy lost by only using the condition numbers of f and g. This will be an overestimation of the number of digits lost, so it gives us a way to bound the theoretical relative error.

(c) Let $\kappa_f(x)$ denote the conditon number of evaluation of the function f at the point x. Find a function f which is infinitely differentiable on the interval (0,1) (but which may have singularities at $x = 0$) such that

$$\lim_{x \to 0^+} \kappa_f(x) = \infty$$

.

The function $f(x) = e^{\frac{1}{x}}$ will give us what we want.

Then, $f'(x) = \dfrac{-e^{\frac{1}{x}}}{x^2}$, so

$$\kappa_f(x) = \frac{|f'(x)||x|}{|f(x)|} = \frac{|e^{\frac{1}{x}}||x|}{|x^2||e^{\frac{1}{x}}|} = \frac{1}{|x|} \to \infty \text{ as } x \to 0.$$

This tells us that even though this fuction is differentiable up until $x = 0$, evaluating it near $x = 0$ will result in large relative errors.

8. You are running a simulation that updates time every 0.1 seconds. The time in the simulation is kept by incrementing a time variable.

   (a) Suppose you are simulating one day (86,400 s). Implement the above code, and compute the absolute and relative errors in the time at the end of the simulation.

   Running the simulation and comparing t with the exact number of seconds in a day, 86400, we get

   Absolute error $\approx 5.4126(10)^{-7}$

   Relative error $\approx 6.2646(10)^{-12}$

   (b) Change the time increment to 0.125 and again run the simulation to 1 day. What are the relative and absolute error in the time?

   With this new dt value, we get Absolute error = Relative error =0.

   (c) Explain the difference in the results from parts (a) and (b).

   The reason that in (b) we are able to get no error is that $0.125 = \frac{1}{8}$ can be stored exactly as a binary number, $\frac{1}{2^3}$. However, for part (a), 0.1 must be rounded when it is stored as a floating point number. Therefore, each time that we compute $t = t + dt$ in the algorithm, we are actually adding $fl(0.1)$, and we are introducing non-zero round-off error each time.

9. Write a Matlab function to perform matrix multiplication of two matrices. Your code should take matrices of any size and return an error if they cannot be multiplied. Use `tic` and `toc` to time your function and the built-in Matlab function to multiply A, a 1000×40 matrix with B, a 40×4000 matrix of randomly chosen numbers from 0 to 30. Report these times, as well as the flop count of your function for A and B, both $n \times n$ matrices.

   See PS1N9.m and function my_matprod.m for Matlab code. My code took 0.57 seconds and the built-in Matlab function took 0.0069 seconds. As written, if A and B are both $n \times n$, the flop count is $2n^3$ because there is one addition and one multiplication in one step of the interior loop, for which there are n steps. This is done n times for the middle loop and n times for the outer loop.

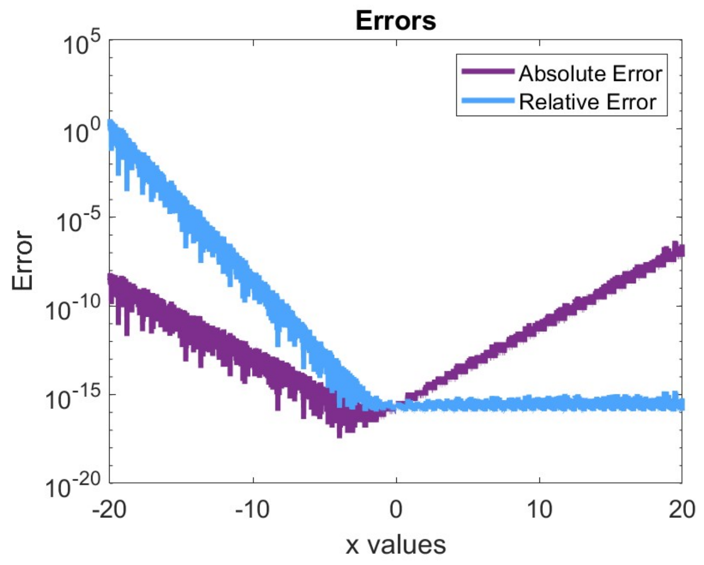
10. (22 points) Suppose that you can only use addition, subtraction, multiplication, division, rounding and integer powers of numbers. You decide to use a Taylor Series to evaluate $y = e^x$ with only these operations, since you learned in calculus that it converges for all x. Below gives an example of such code.

   (a) Explain what the while loop is doing in this code.

   In each step of the while loop, we are adding on the next term of Taylor Series. This continues as long as the value keeps increasing. As soon as the value doesn't increase (when newsum=oldsum), the loop terminates and we end with our estimation of e^x. Round-off error/floating point representation is the reason that eventually the sum will cease to increase because the new term will be too small to "see" in comparison to the size of the sum so far.

   (b) Assess the accuracy of the algorithm below by using it to approximate $y = e^x$ on the interval $x \in [-20, 20]$ by comparing with the built-in library function for the exponential. Compute the absolute and relative errors as a function of x and plot the results (use log scale for the error; i.e. in MATLAB use command semilogy for plotting).

   See file PS1N10.m for Matlab code for both (b) and (d). Below is a semi-log plot of the errors.

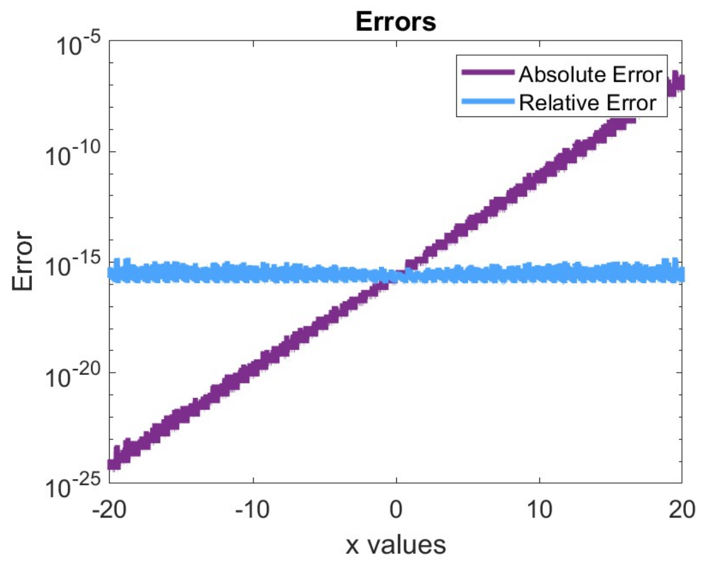


   (c) For what values of x do you see poor performance from the algorithm? Explain the reason for the poor performance.

   The plot above shows great performance for $x > 0$, as the relative error is down near ϵ_{mach}. The relative error is much worse near $x = -20$. Additionally, while the absolute error is bad near $x = 20$, this is due to the fact that the size of e^x, the value we are approximating, is very large here. On the other hand, the absolute errors are also worse near $x = -20$, where the sizes we are approximating are very small. These errors start to get worse soon after the x values dip below 0. The reason the algorithm shows worse performance when x is negative is that since the series is then alternating, we end up adding in terms that are the opposite sign of the sum to which they are being added. When these two numbers are close in magnitude, we get cancellation error. To see where exactly this happens, let's look at an x value like $x = -15$. This cancellation error doesn't happen near the beginning of the sum because the terms that we are addding (or subtracting) to the "current" total are actually much larger than the"current total." It also doesn't happen near the end of our algorithm, when the new terms we are adding are tiny compared to our current total. Instead, it happens somewhere in the middle of our algorithm, when the "current total" is at the same size as the new terms being added. When this is the case, our negative terms cause us to "subtract" numbers that are very close together, resulting in large cancellation errors.

(d) Based on your answer from the previous part, modify the algorithm to eliminate the poor performance. Discuss the changes and demonstrate the performance of the modified code by plotting the errors as a function of x.

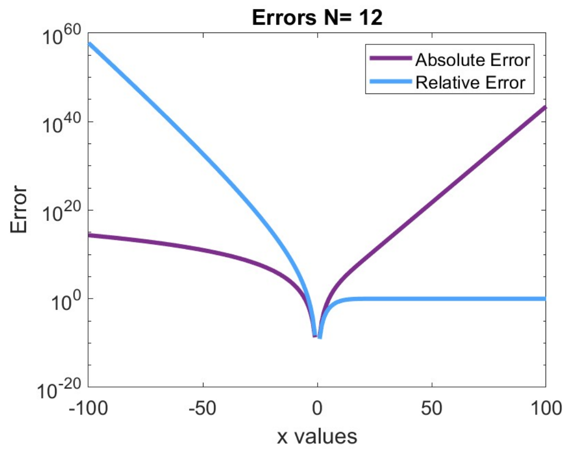
By altering the algorithm to first approximate e^{-x}, when x is negative, we can eliminate having to use the series on any negative inputs. Then, we simply use that $e^{-x} = \frac{1}{e^x}$ at the end. See myexp1.m for the new algorithm. The new results are shown below. We now see great relative error (near ϵ_{mach}) for all values of x in our range.

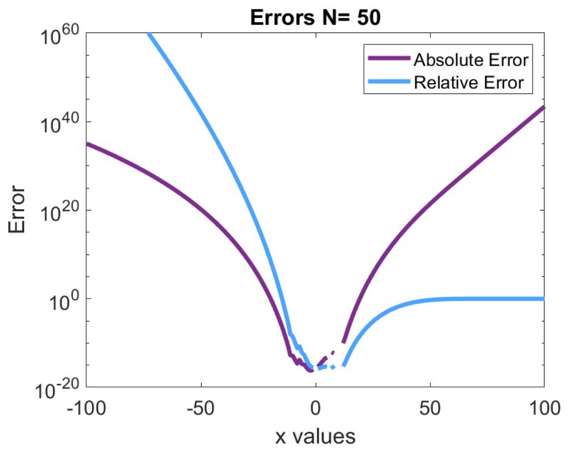


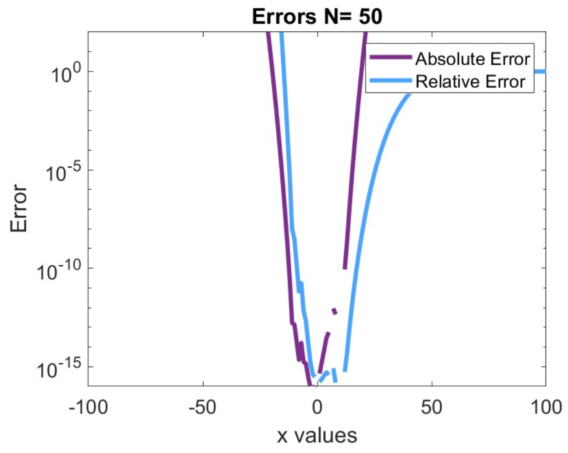
11. (BONUS) Suppose that you can only use addition, subtraction, multiplication, division, rounding and integer powers of numbers. You decide to use a *truncated* Taylor Series to evaluate $y = e^x$ with only these operations.

   (a) Create a function that approximates e^x by truncating the series to n terms. Use your function for $n = 12$ to approximate e^x for some values of x between -100 and 100. Repeat for $n = 50$. Comment on your results.

   See file PS1N11.m for both (a) and (b). The function for part (a) is myexp2.m. As we saw in Problem (10), the errors are are bad when $x < 0$, but a new problem arises when we use a truncated series (instead of allowing the series to include as many terms as needed, as we did in Problem (10)). Any point away from the center (here, 0) gives bad errors. In the third plot, zoomed in, we can see that the relative errors are bad even when x is positive, as soon as it is away from $x = 0$. They are especially bad in the areas when $x < -13$ or $x > 33$.







   (b) By exploiting properties of the exponential, design an algorithm for accurately computing the value of e^x using your truncated series for x values between -100 and 100.

Explain your algorithm and implement it. Report your relative error for e^x for $x = \pm 0.5$ and $x = \pm 100$, and compare to the previous part. You should be able to acheive relative errors below 10^{-13}. Hint: Based on the Taylor Series remainder, for what values of x do you expect the series to be the most accurate? Exploit properties of the exponential to make an algorithm that only uses the series on this range of x values.

The function for part (b) is myexp3.m. Depending on how accurate you want your solution to be, you can pick a value of r and make sure to only use the truncated series on values of x such that $|x| < r$. Your choice of r could depend on how many terms you want to use in your truncated series. I will assume we want to use N=12, and I will pick $r = 1$. Again, this choice will depend on how accurate you want your solution to be.

My new algorithm breaks up the exponential as follows: $e^x = (e^{\frac{1}{8}})^a (e^m)$.

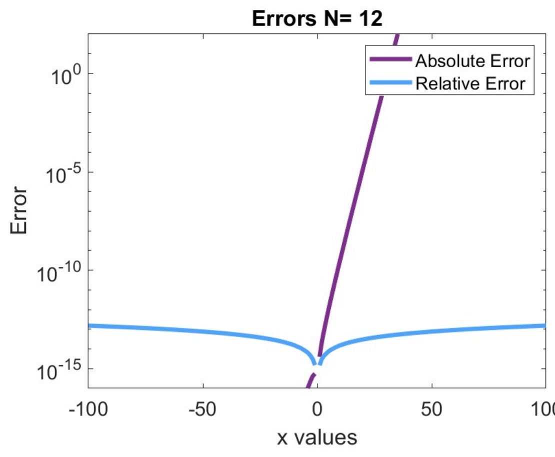
First, I find $a = floor(8x)$. Then $m = x - a/8 < 1$.

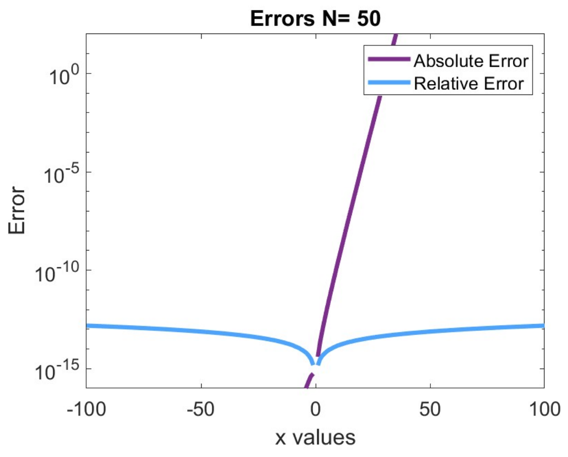
Note that the choice of 8 is largely arbitrary.

We need to check if we only use addition, subtraction, multiplication, division, rounding, and integer powers. First, we use a rounding function to find a. Then, only basic arithmetic is used in obtaining m. Next, we will find e^m and $e^{1/8}$ using our truncated Taylor series, which was already designed to only use these operations. Next, I will raise $e^{1/8}$ to an integer power, which is allowed, and do another multiplication.

Also, note that both times I will use the truncated series, the input will be less than 1, so I will only be using the original algorithm where it is most accurate.

As you see in the plots below, we are able to keep our relative errors down extremely low with this new algorithm.





The table below gives the relative errors for the requested x values for the algorithms used in part (a) and part (b), with $N = 50$. We see that for part (a), just as we saw in the plot, the errors become large away from 0. However, we are able to achieve errors below 10^{-2} for all of these x values using the new algorithm in part(b).

Relative Error		
x	part (a)	part (b)
-0.5	1.8×10^{-16}	7.3×10^{-16}
0.5	2.7×10^{-16}	8.1×10^{-16}
-100	3×10^{78}	1.5×10^{-13}
100	1	1.5×10^{-13}