Assignment 2 Write Up: Integration

Aniket Pratap

January 19, 2022

Abstract

The main goal of this assignment was to make a math library – without the use of <math.h> and create an integrate function. The math library would include approximate calculations of e, log(x), sin(x), cos(x), and sqrt(). the integrate function uses Simpson's 1/3 formula to integrate equations utilizing this new library. The equations are as follows:

$$e^x = \frac{x^k}{k!} = \frac{x^{k-1}}{k-1} \times \frac{x}{k}$$

$$sinx = \sum_{k=0}^{\infty} -1^k \frac{x^{2k+1}}{2k+1!}$$

$$cosx = \sum_{k=0}^{\infty} -1^k \frac{x^{2k}}{2k!}$$

$$\sqrt{x} \text{ and } logx = \text{ Newton-Raphson Method } x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

Simpson's rule:

$$\int_a^b f(x)dx \approx \frac{b-a}{6}\left[f(a) + 4f(\frac{a+b}{2}) + f(b)\right]$$

These are the formulas used to create the math library. In addition, the integrate function has a lower and upper bound along with partitions. The more partitions something has, the more accurate the approximation and vice versa. Now that the concept is introduced, we can move on to the next section.

1 Building Sin() and Cos() and Exp()

began the assignment by rewriting the pseudo code provided to us on the assignment document. As these were coded in python, I had to recode it in terms of C. The Exp() function remained identical, but the others were slightly changed. The Sin() and Cos() functions could be run faster since these functions are symmetric. In order to take advantage of this symmetry, I created a function called range() that scaled any value back into the range of 0-2π. This decreased run time because the computer would be doing unnecessary calculations if the domain was extended to this fixed range. And although Sin() was given in pseudocode, I had to code Cos() on my own. I mainly did this by looking at the similarities between the two functions, but the hardest part was figuring out how to make the numerator an even product of x. Sin() started with x and multiplied x * x making it an odd product at all times so I didn't want to follow this rule. My solution was changing the starting value from 3 to 2 and doing one step before my for loop ran. That one step would be t * x only once to acquire x squared and I then manually incremented the step by two. The problem with this, however, was that the operation signs would be one step off due to the formula. This was an easy fix however and all I did was manually multiply by -1 rather than 1. starting the formula at -1 allowed me to obtain the proper operational pattern. I also began the sum with 1 as that's how the formula went. The k(k-1) part in both functions emulates a factorial and by storing in a variable, I didn't need to do each factorial again since I already had the previous product – making the program even faster. Epsilon was used as a stopping point for these functions because the funtion then stops when the values become meaningless to the whole integrated area. Factorial grow fast and the approximation would eventually reach 0 – we don't want that– so epsilon is the best default stopping point.

2 Building Sqrt() and Log()

his was a little more straightforward as I rewrote both formulas using the pseudo code provided. The only problem was that I needed to scale values because the functions became useless when a large number passed in as an argument. For large numbers to work for Sqrt(), I factored out all the 4's and made a product of all the 2's the 4's would make. This pseudo code was provided, but the implementation was not shown. My solution for installing the while loop doing this task was within the main while loop doing the task. Before I do the calculations, I can factor out all the 4's and replace them with 2. Placing it inside the first while loop made the most sense in order to achieve this result. I then multiplied the final value with all the 2's that I accumulated in order to get the final approximated value. Log() was similar except that I needed to factor out e rather than 4. The reason is that the argument of log could be rewritten as $log(e^k \times a) = log(a)+log(e^f) = f+log(a)$. This means that larger numbers could instead be represented by a coefficient and the exponent f. This factoring loop was implemented inside the main while loop and the final value was added with the factored out e; allowing the program to work for values greater than 29.

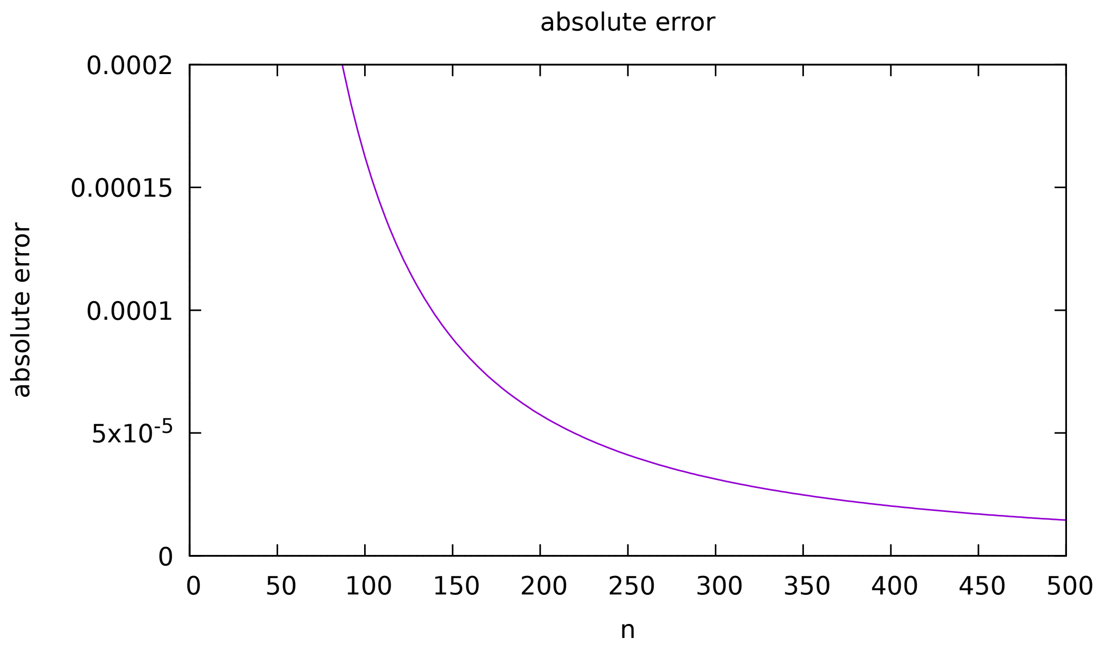
3 Building Integrate

his function was not provided and by observing the formula, I created a functional integrate function. One of the things I noticed was that the main function was multiplied by the change of x. How did I get this change of x? I simply added the points at the start and end and divided them by the number of partitions the user wanted. By using this number with an incrementor, I was able to go on to the next partition in the function. I also noticed that the first and last points of the function weren't being touched; this meant that they would be added to the end. Hence, I created a variable called sum, adding the first and last elements, and adding it to the end. I also noticed that each operation was positive, so it made my life a little easier, but if the iteration was odd, f(next number) would be multiplied by 4 whereas it would be multiplied by 2 if even. This was a simple if else statement inside the for loop for the integration function and the base model was done. The only problem was to fix the divide by zero options. Some functions might divide by zero and this causes the integrate function to result in an error. To fix this, I created another if else statement in the for loop stating that if a number f(number) == 0, then I would replace 0 with 1e-60. Yes this will affect my accuracy, but I thought it was a good solution for zero division error; even if it looks messy.

4 Building Integrate.c

his was where the main function resided. The program runs by the user entering ./integrate -a-j -p number -q number -n number. The -p is the lower bound while the -q is the upper bound. The -n are the amount of partitions. Before I began, I looked at Collatz.c for reference and did a similar usage function for integrate.c. Anytime the user did something wrong, the usage prompt would shop up. This was done using fprintf() because it allows us to print the error to stderr rather than the console. The next thing I coded was another error handling mechanism for the functions. If the user inputed some invalid value, with regard to the function, then a prompt with domains of each function would show up. I pass in a pointer pointing to a char array. I made this array to hold the equations in a string format so that I could use them in multiple areas. I began my main function and set the arguments as int argc and char **argv, so that I could take inputs from the terminal. I created a pointer pointing to an array of 10 functions, a-j, and made my string array as I mentioned before. I got the idea of using a string array from Eugene during his section. I also created a boolean array so that I can check if any of the functions are being inputted as arguments. This was done so that I could input multiple functions at once. After setting the default value for some needed variables, I began my getopt switch statement. I set opt = getopt(argc, argv, OPTIONS) so that the while loop would keep checking if the user input matches the OPTIONS. I defined options at the start, because I used Collatz.c as reference. I then made my switch and cases. The cases had all the inputs and for the equations, I set the my input array, which I set everything to false before, to true if that letter was typed in. This meant that I didn't have to write my integrate function 10 different times, but only once at the end. I took in the p, q, and n, and I set the default to show the usage function stated earlier. After the getopt, I did some error handling. I created a checker to check the boolean values of the input function. If the user didn't input a function, the program will spit and error saying that no function was inputted. I did this my oring all the values in input. If the total value came out as false, then it was clear that the uesr didn't specify which function. After it passed that test, my code then checked if the user inputed an upper or lower bound. This was done by setting the upper and lower to 1e-14 and if either were that value, it would spit out an error saying no bound inputted. I chose 1e-14 because nobody would be sane enough to type that in. This served as a good test value. The next thing I did was set the partitions to the default: 100. If the user didn't enter anything the partitions would be 0, the default value I set. If partitions are zero, then change it to 100 because the user did not input anything. The reason I did this in a more "weird" was because I was getting a seg fault error if I made partitions optional for some reason. I fixed it and that was my error of thinking how -n needed to be opttional. I believed that if the user didn't specify how many partitions, the program would take 100, but I was wrong. Next, I checked for invalid bounds. I first plotted these functions on desmos and created the bounds from there. I would stop the program from running because...i wouln't run if it's out of bounds anyway. This was more cleaner. I did this for a, b, and e after checking their bounds. These were simple if else statements, and if it was true for any of the statements, the program wouln't run.

5 Function A Graph



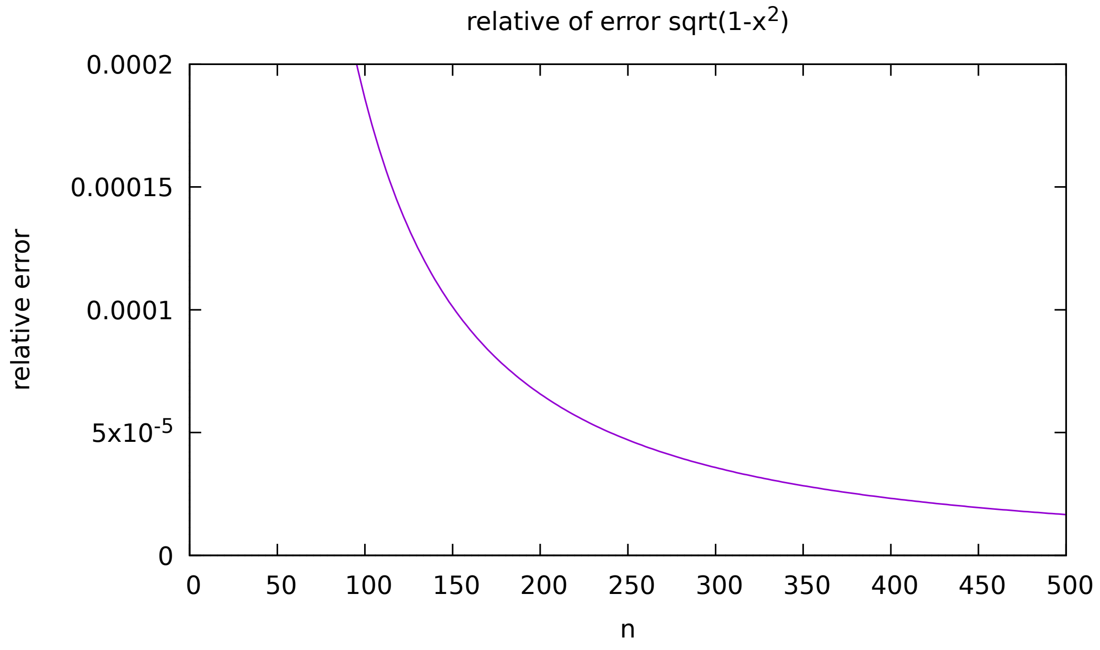
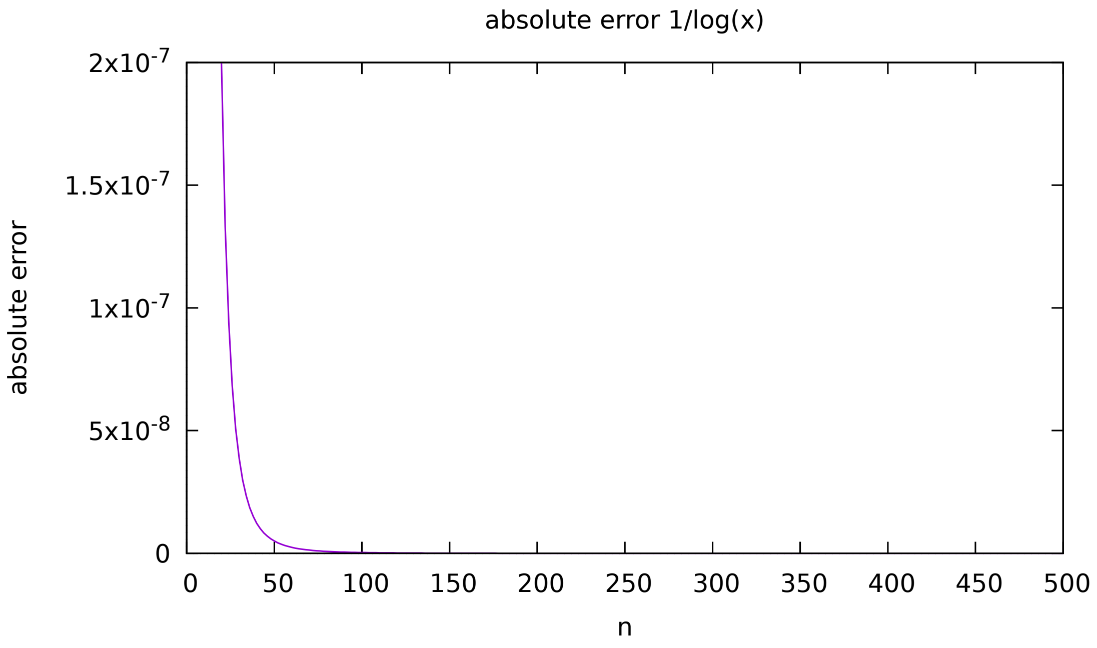


Figure 1: You can see as the number of partitions increases, so does the absolute and and relative errors. The relative error says how big is the error relative to the actual number and absolute error shows only the difference.



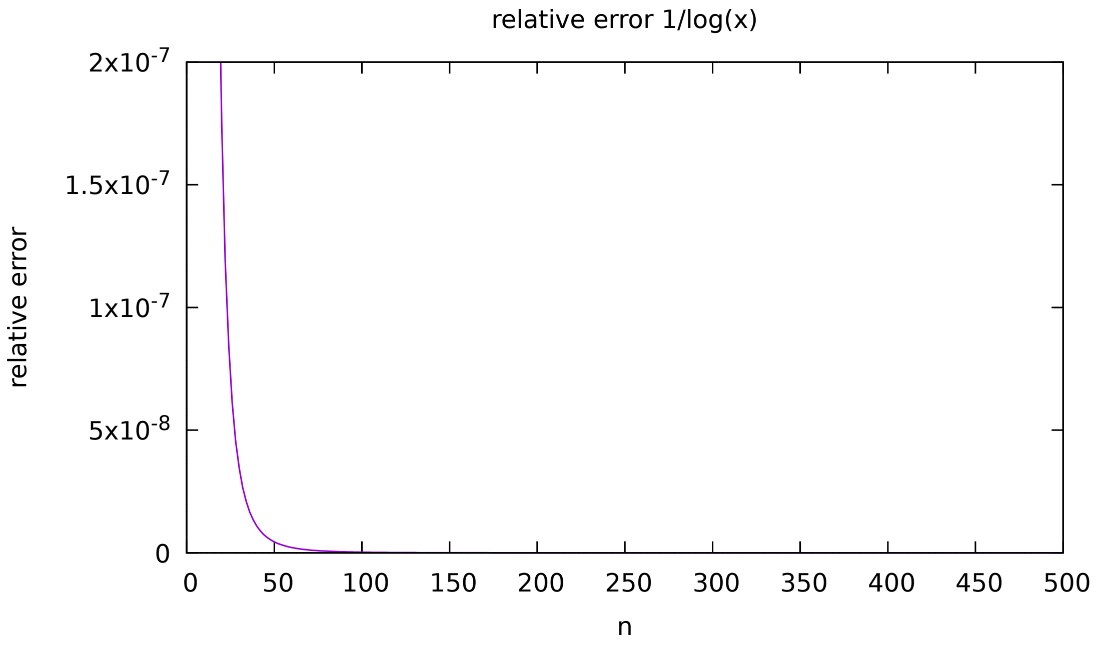
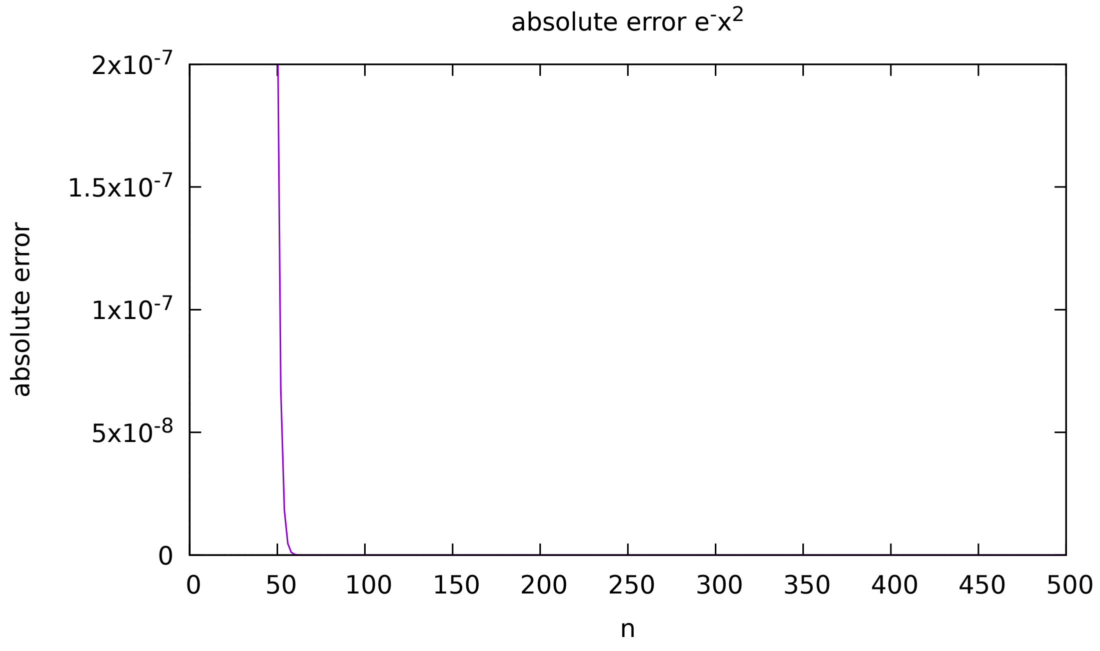


Figure 2: Function b shows similar findings when compared to a



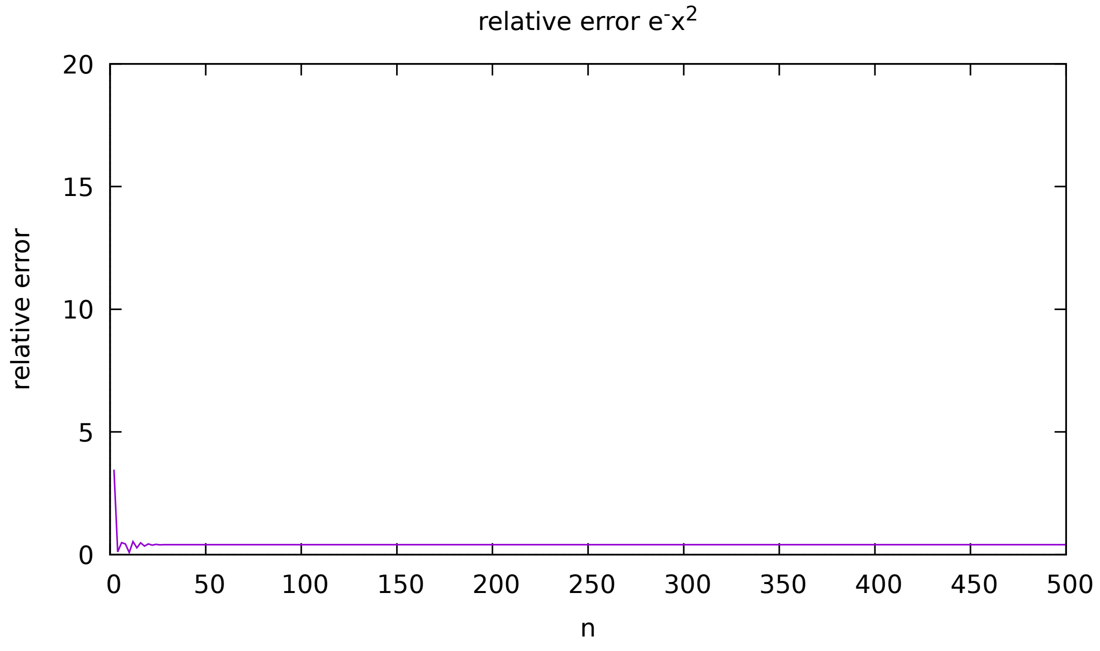


Figure 3: Function c shows the absolute error getting smaller and smaller while the relative error plateaus at a reasonably high number. This must be due to my Log function not providing as accurate results as the actual log value. The Log function was very close when I tested it so it must be something else. It's odd that the absolute error retains a small value while the relative error is high.

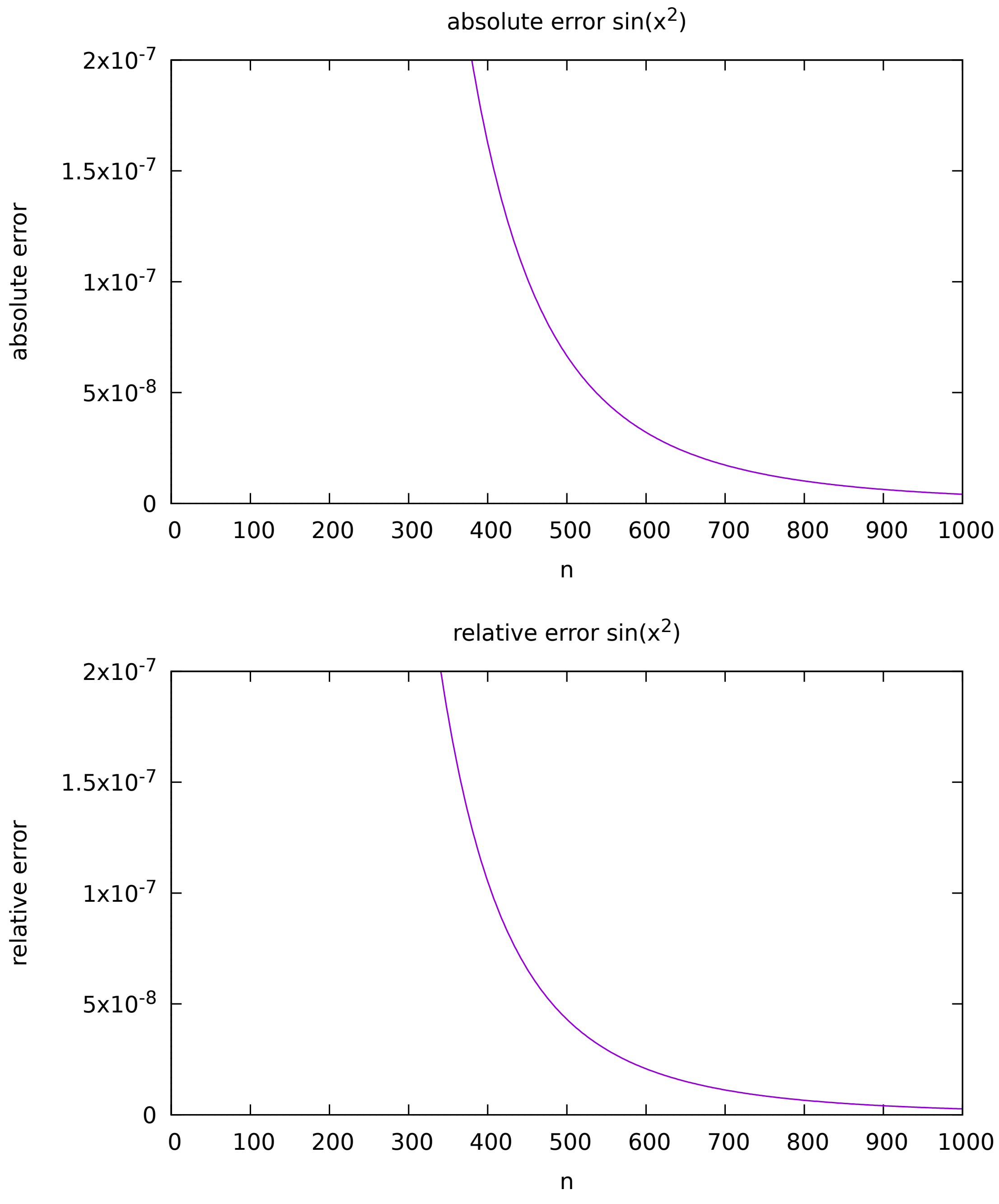
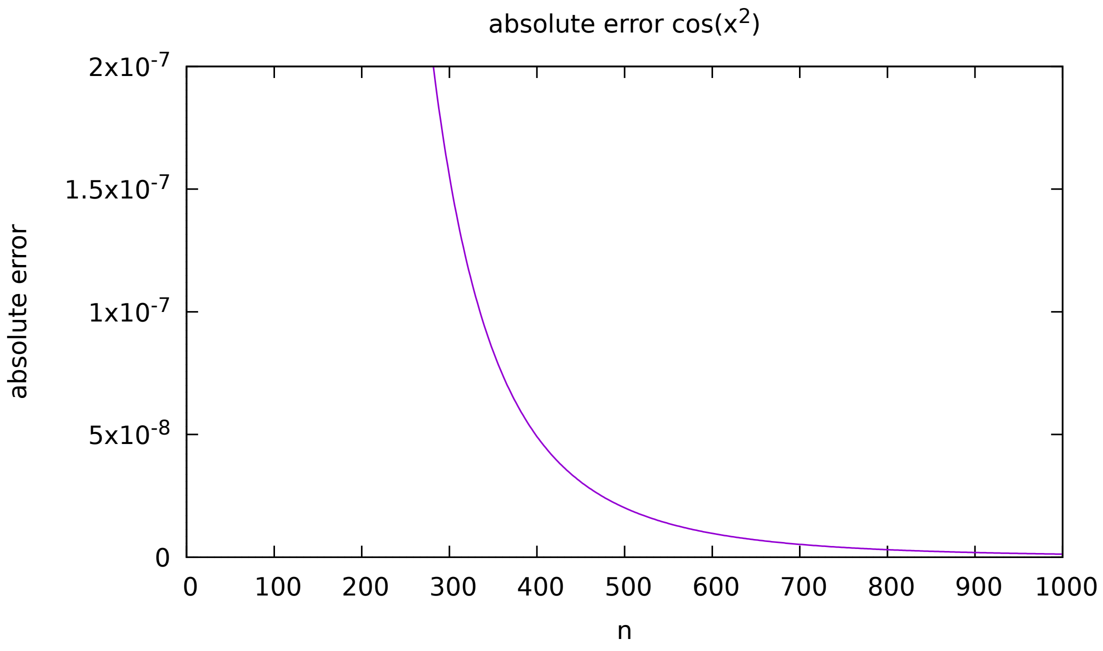


Figure 4: function d shows similar results as functions a and b. This meant that my symmetry function worked according to plan.



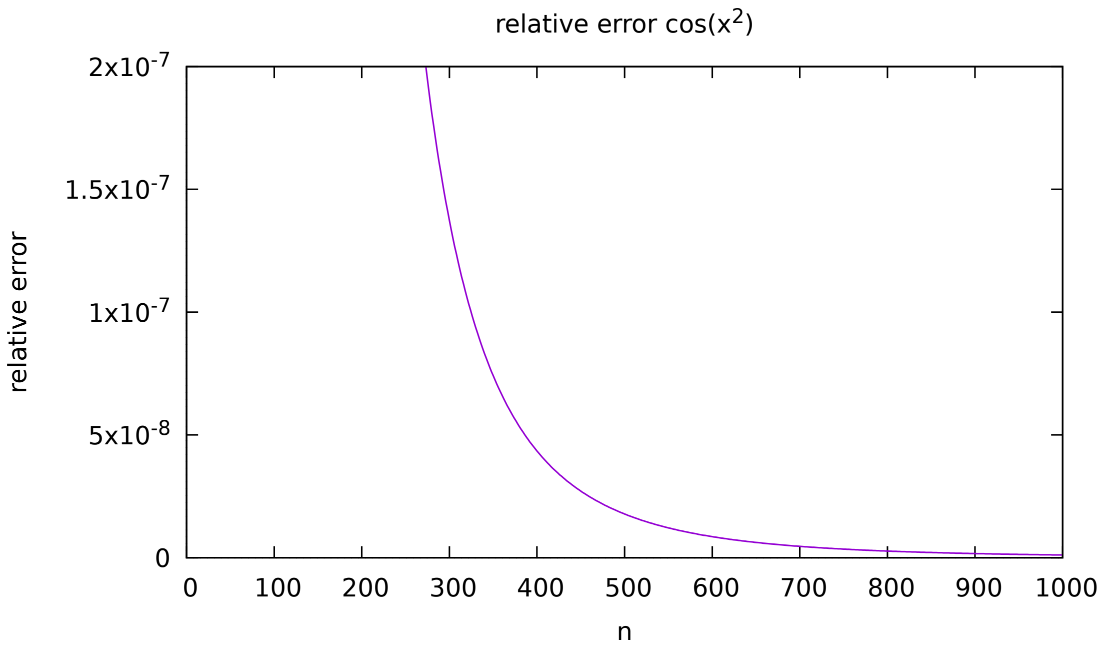
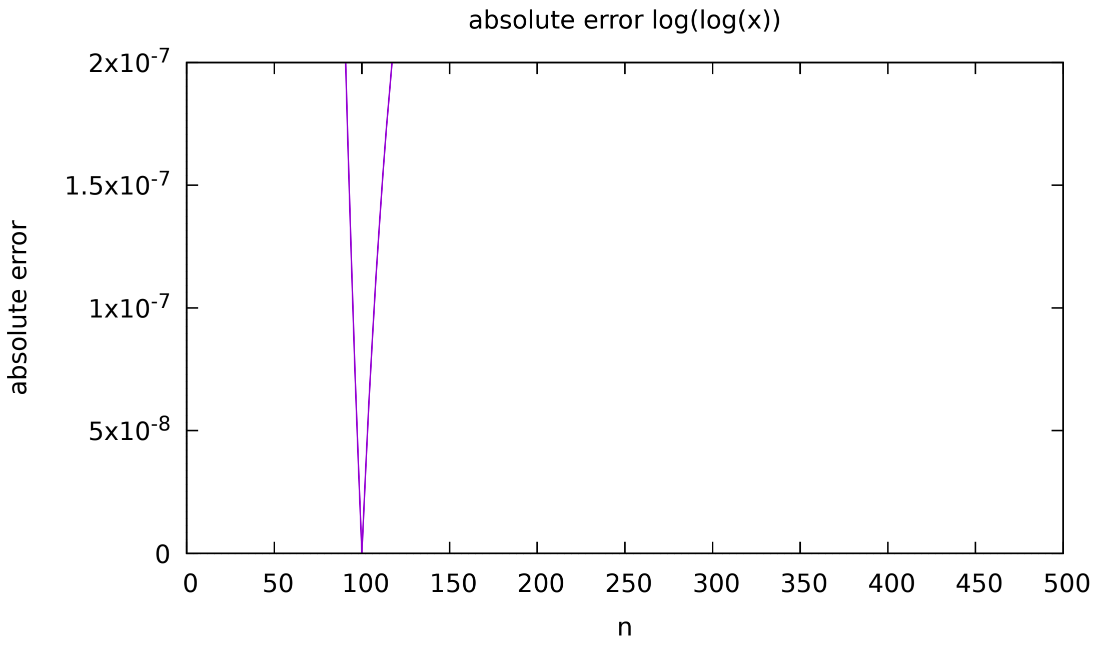


Figure 5: function e naturally follows function d. Both functions are similar so it makes sense they have the same graph.



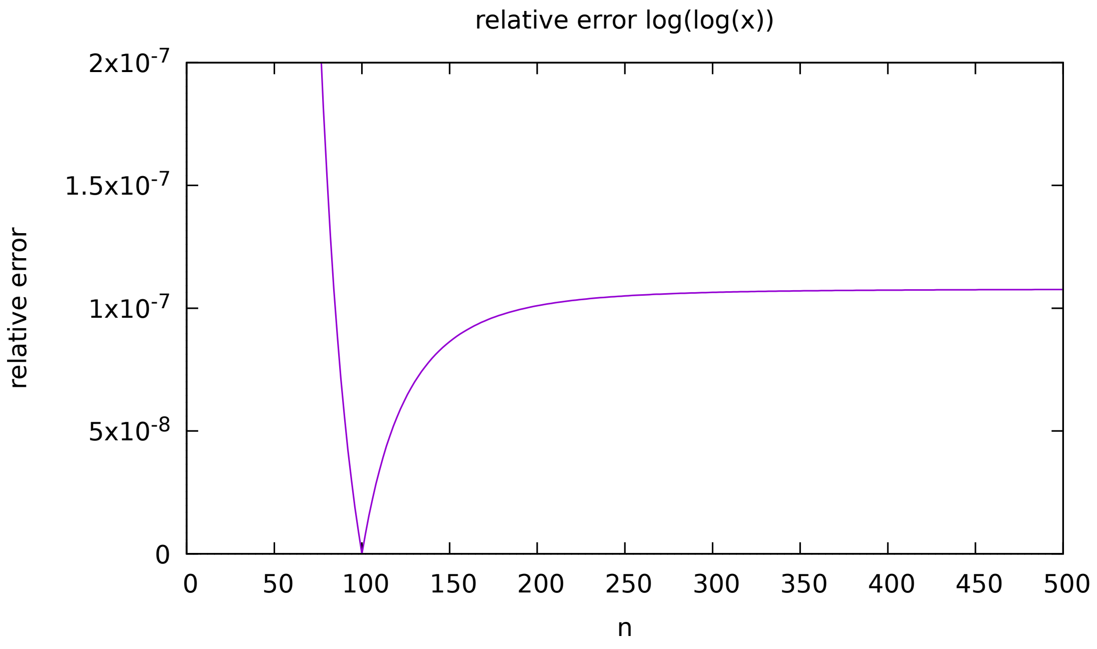
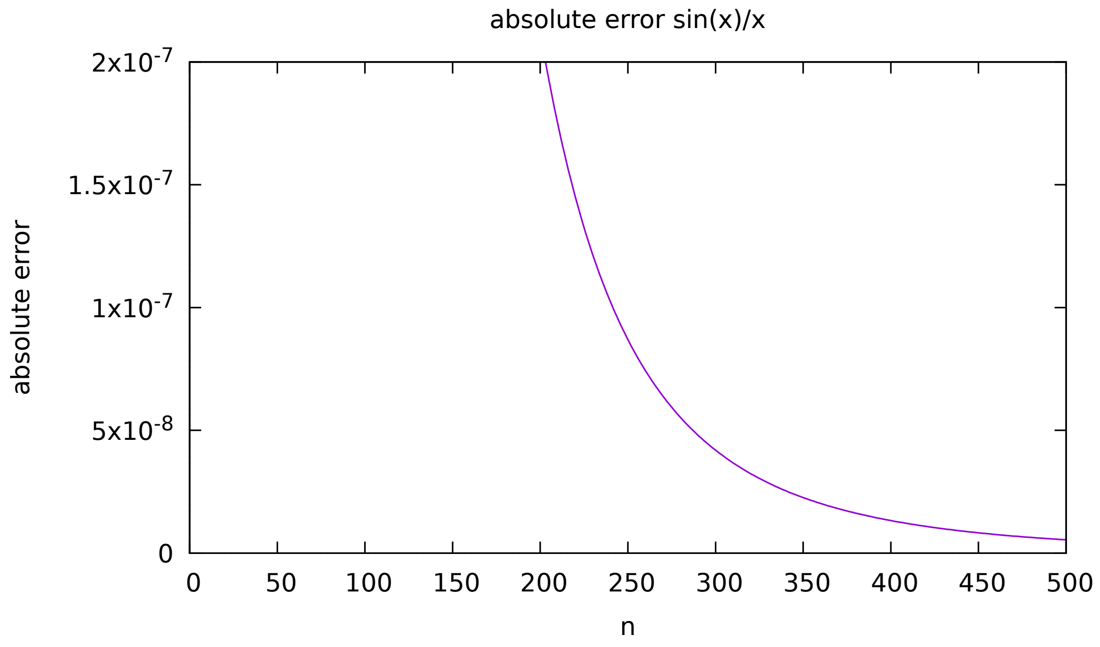


Figure 6: I thought this was the coolest graph because it shows that adding more partitions doesn't always mean better accuracy. When function f arrives at the 100th partition, it reaches its optimal value. Since log(x) increases until infinity, it makes sense that both errors also increase when compared to the actual result.



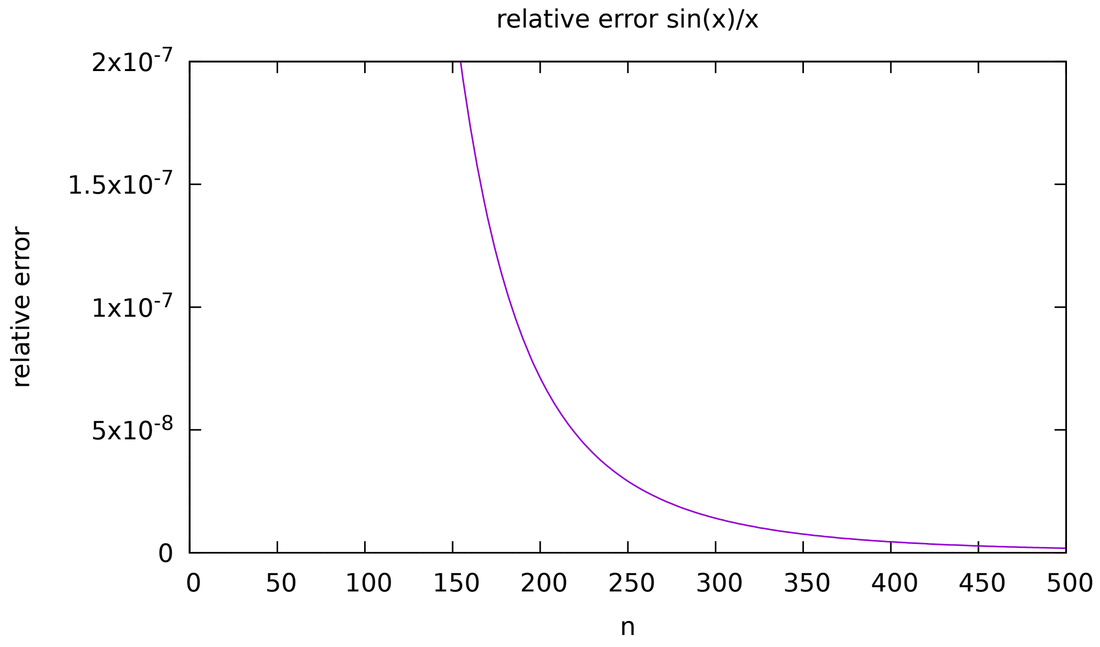
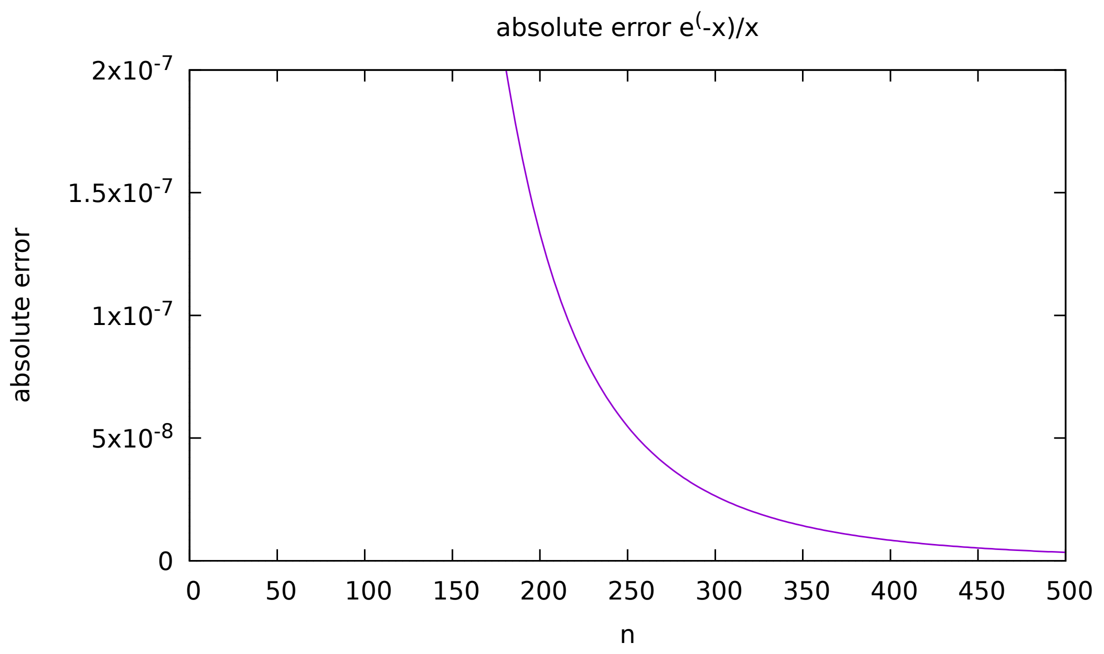


Figure 7: Funtion g also shares the same result as d and e. Makes sense because it contains sin(x), but this also means that my zero division error handling worked!



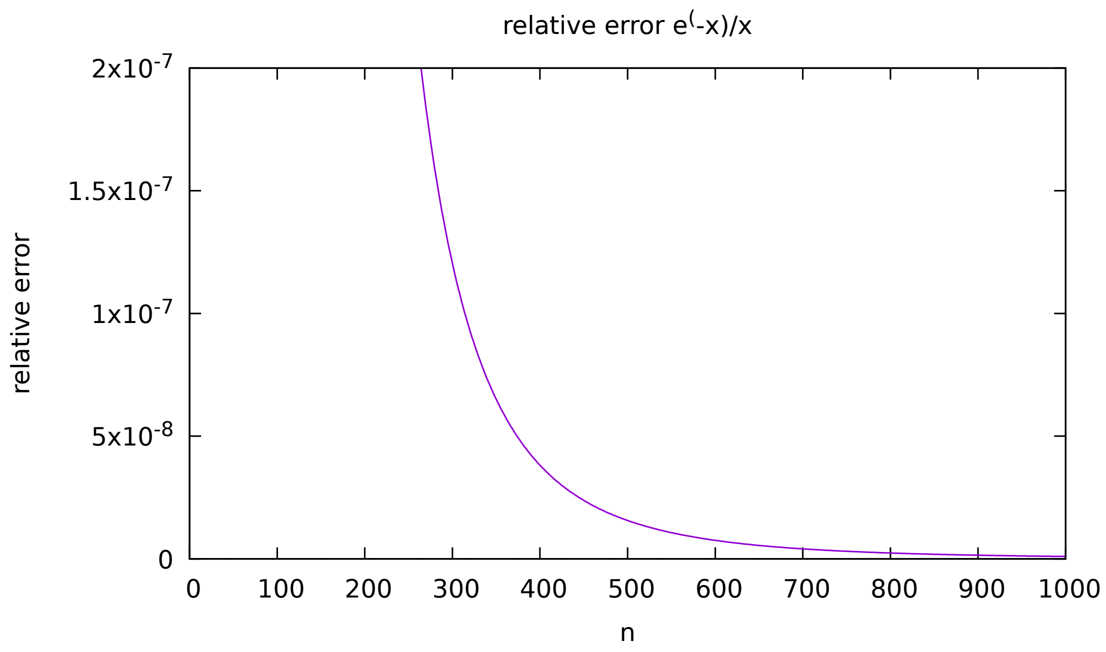
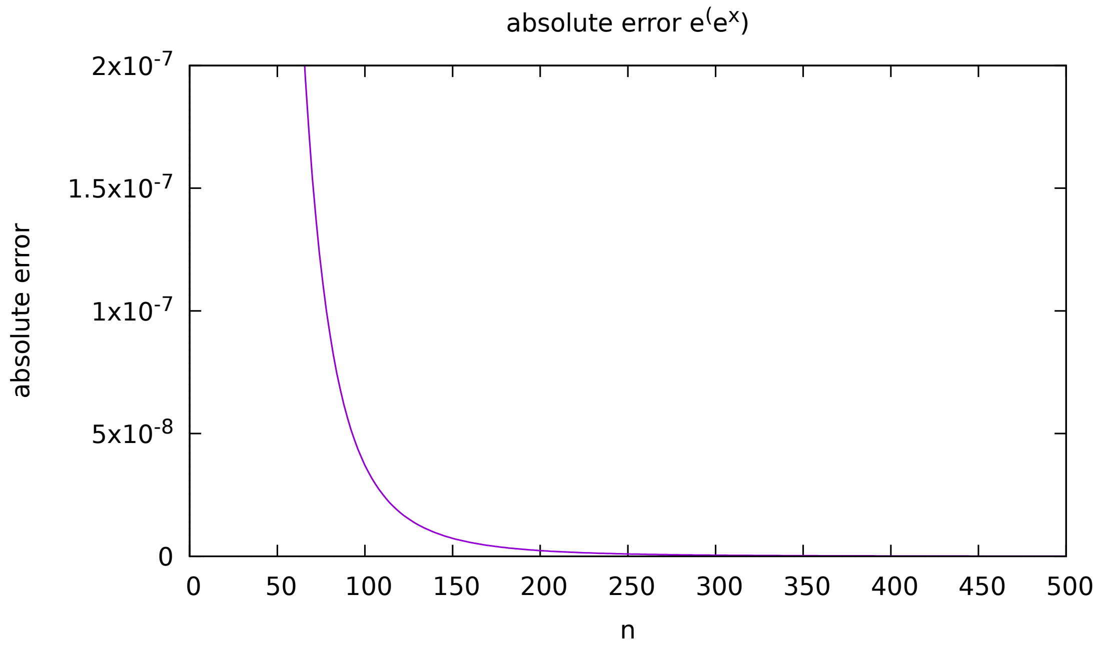


Figure 8: Funtion h is similar to function g and has a better error than function c, which also used Exp()



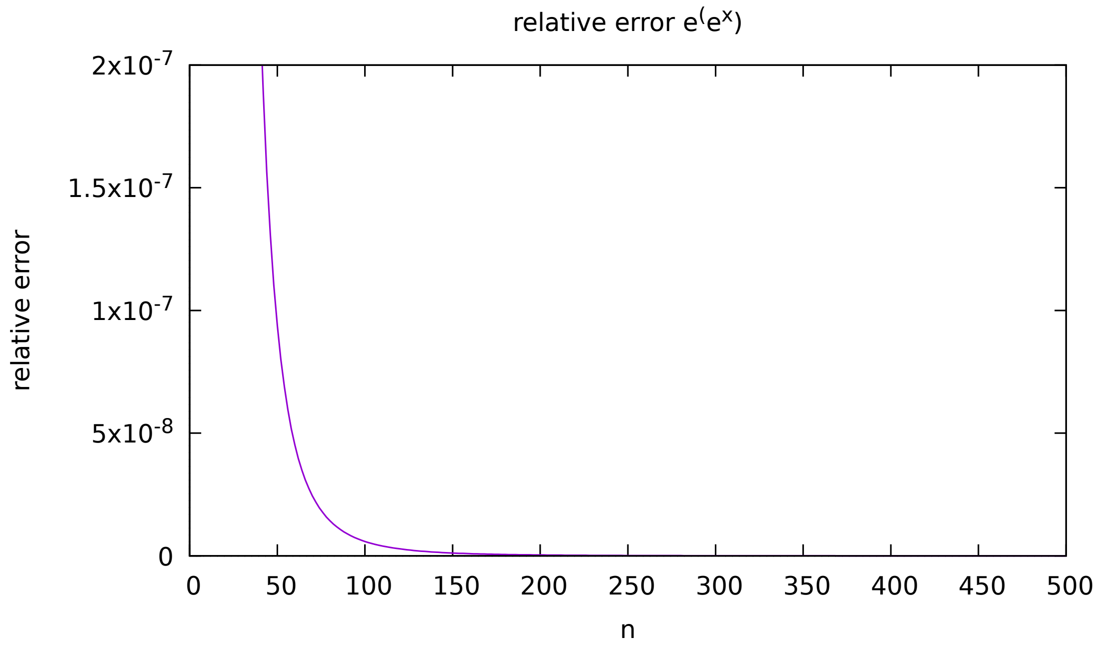
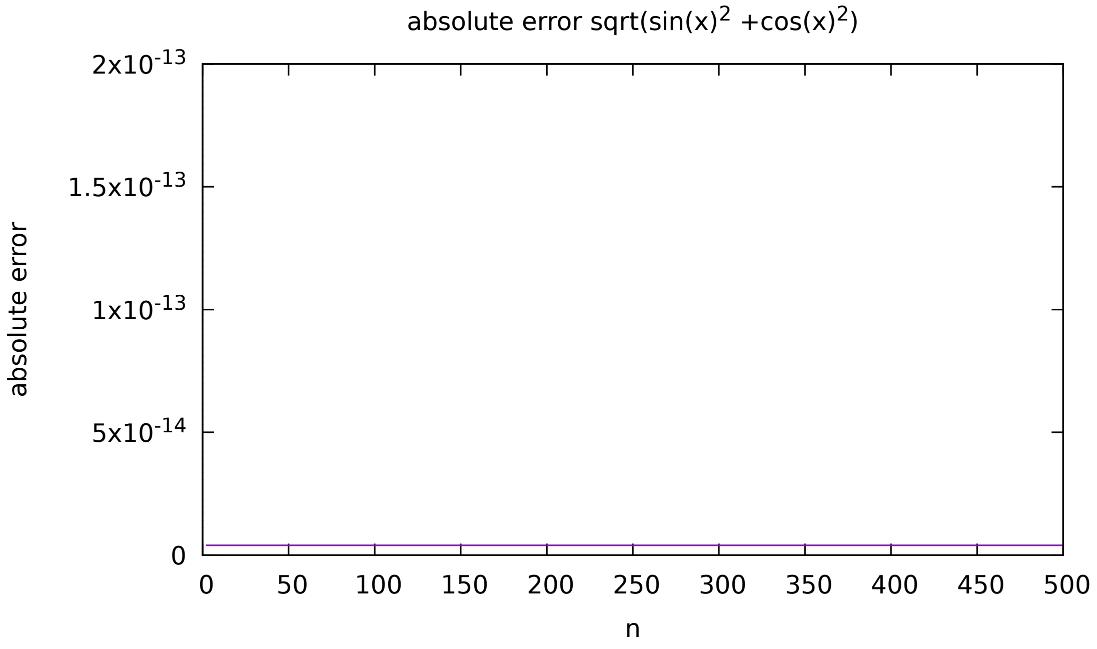


Figure 9: Function i has similar characteristics when compared to function h



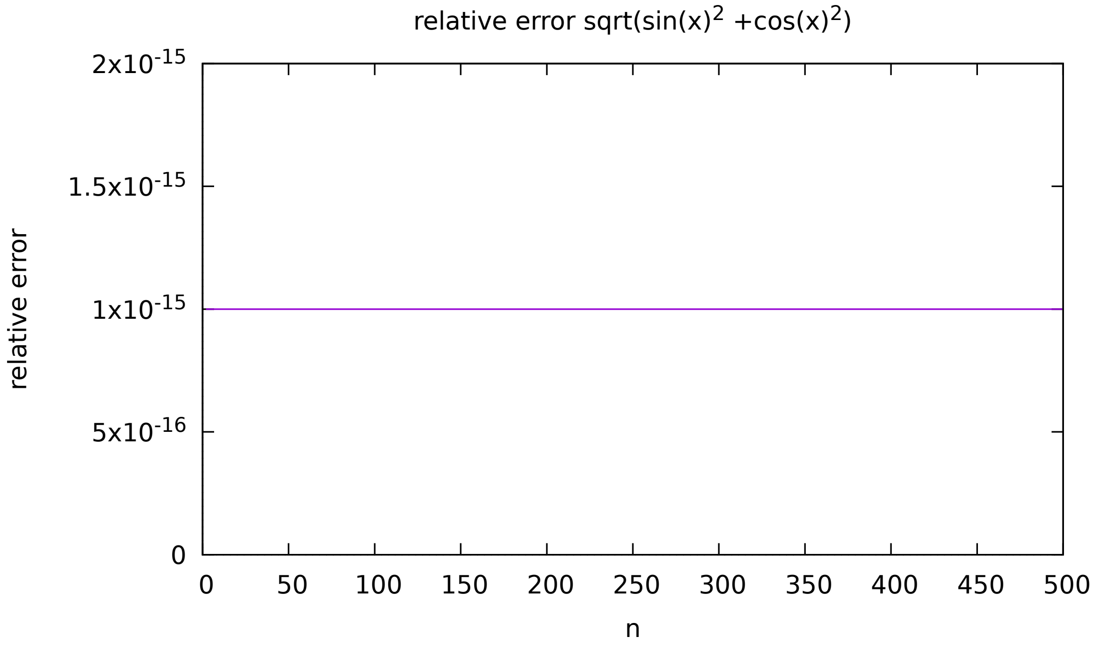


Figure 10: These graphs may seem weird, but it makes sense when the function itself is a straight horizontal line. This means that both errors should be constant throughout.