Assignment 2 Write Up

Nicholas Chu

January 18, 2022

1 Results

1.1 mathlib.h

Below is a table comparing the values produced by the functions in mathlib.h and the math library, math.h, and in this case when given the arbitrary number of 8.

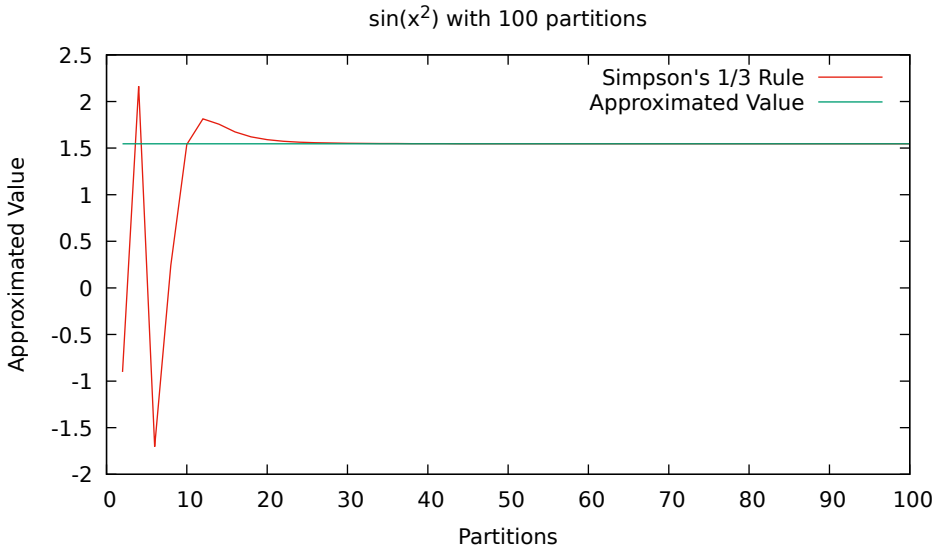
Function	mathlib.h	math.h
e^x	2980.95798704172785	2980.95798704172830
sin(x)	0.98935824662341	0.98935824662338
cos(x)	-0.14550003380863	-0.14550003380861
$\sqrt{x}$	2.82842712474619	2.82842712474619
log(x)	2.07944154167984	2.07944154167984

Figure 1: mathlib.h versus math.h using 8

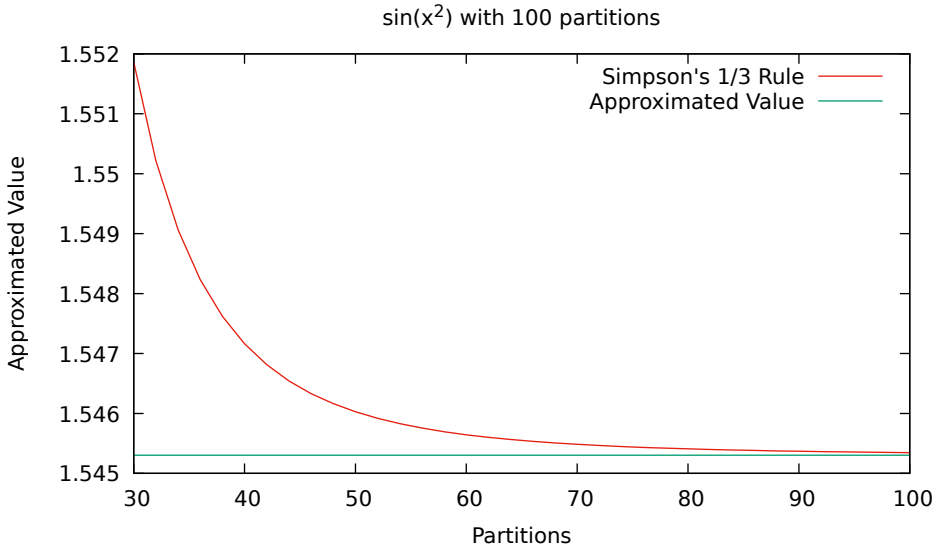
For the most part, the functions produced approximately or exactly the same results when given the same value. The functions sqrt() and log() and the functions Srqt() and Log(), from math.h and mathlib.h respectively, produced the exact same values as their counterparts, at least up to the fourteenth decimal place. exp() and Exp(), sin() and Sin(), as well as cos() and Cos() all have a slight degree of error nearing the last 2 decimal places. This error is most likely due to the mathlib.h functions stopping their approximations when the error is near epsilon, if it were to continue approximating after that point the value would most likely begin to equal again. However, the margin of error ended up being so small for these functions that it did not end up making any real impact for the calculations in integrate.c.

1.2 integrate.c

The results of integrate.c ended up being very close to the approximated values. Plotting the results of $sin(x^2)$ in the interval $[-\pi, \pi]$ using 100 partitions using integrate.c we can see the how the approximated values values of integrate.c changes corresponding with the number of partitions.

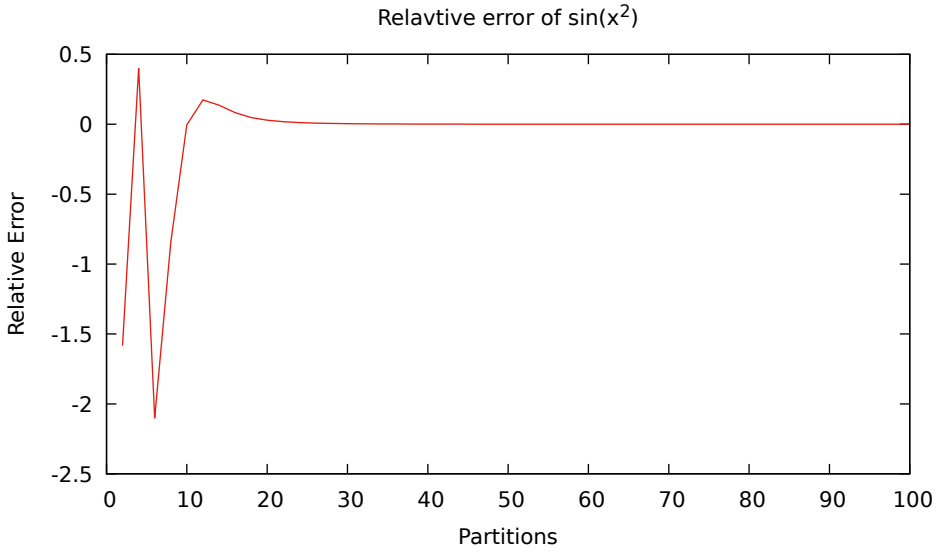


Through this we see how the approximated value of integrate.c's function approaches the actual approximated value. From this we can conclude that with the increase in number of partitions, the approximations become closer and closer to the actual approximated value. From around 2 to 15 partitions we can see that the values are oscillating and vary wildly from the actual approximated value as it adjusts by increasing or decreasing the calculated value to try approaching the actual value. Below is the plot zoomed in around the interval of [30,100] partitions so we take a closer look observe how the calculated value stabilizes and approaches the approximated one.



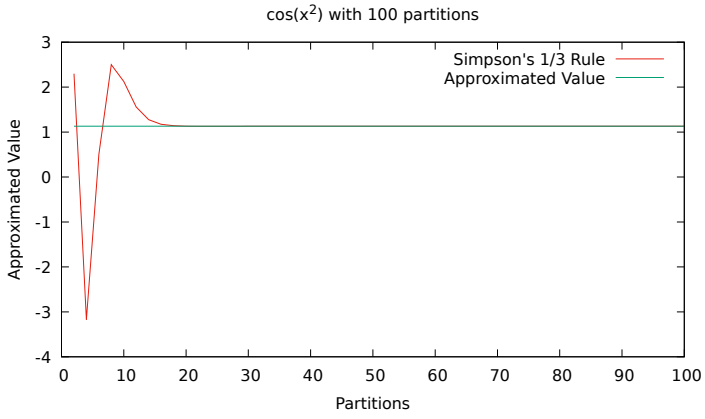
We can see that the actual approximated value is acting like a horizontal asymptote, once the produced values are no longer is oscillating around the line. It shows a similar relationship to something like a $fracx^2$, where the values approach the approximated one as number of partitions increase however we see that these values never actually becoming equal, just getting extremely

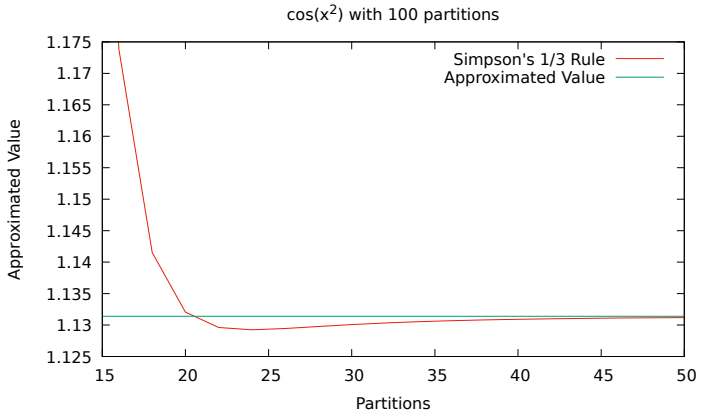
close. If we assume this relationship continues, if we were to compare the actual value from the result of the integrating $sin(x^2)$, no amount of partitions would get us the exact value of its integral. Below is a plot that represents the relative error of the our program compared to the approximated value.

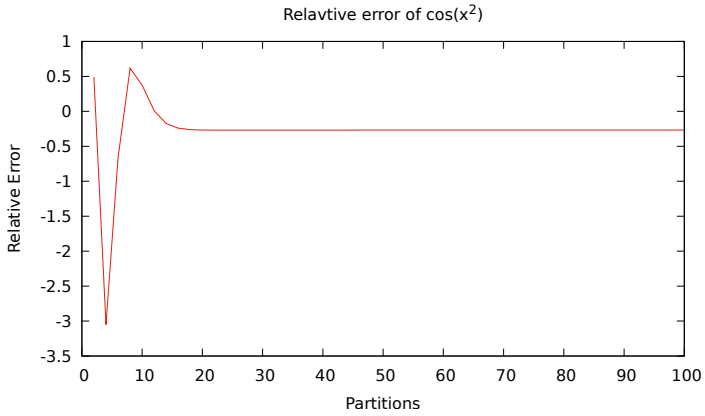


From the plot above, we see that the relative error of the values produced by integrate.c getting smaller as the number of partitions increase. Mirroring the approximated value graph, we see that as the number of partitions being increased we see the relative error similarly approaching zero but never reaching it, to the point where there is little error left. From this we can see some of the limitations of the composite Simpson's 1/3 rule as it never can get an exact solution. However, by increasing the number of partitions used, we can see from the graph that the inaccuracy of approximated values becomes so small that the error becomes almost negligible, making the composite Simpson's 1/3 rule still a viable way to get a value for an integral.

I also plotted the results of integrate.c using the function $cos(x^2)$ into the same plots to get additional results. The graphs produced by $cos(x^2)$ followed much of the same relationships and patterns as $sin(x^2)$ and is evidence that supports the relationships mentioned above.







An interesting thing to note when looking as these graphs is that they look like an inverted version of the $sin(x^2)$ plots and this is most likely reflecting the properties of sine and cosine. The values for $sin(x^2)$ in the beginning starts from some middle and rises to a maximum value then decreases similar to a sine graph, while $cos(x^2)$ starts from a maximum value and decreases like a cosine graph. For $cos(x^2)$, we can also see that the values produced by it also approach the approximated value, however this time from the opposite side of $sin(x^2)$.

2 Conclusion

Through this lab I get some insight into how the math library for C works, and learned how a computer can be used compute complex math functions when only knowing how to use addition, subtraction, multiplication, and division. From the functions created for this assignment we learn that many of these complicated math expressions be broken down into some form of summation involving only addition, subtraction, multiplication, and division and uses these summations to compute an approximated value. The greater number of terms used for the summation means the closer the calculated value gets to the actual value. The two sides to this means that while a computer can not calculate the exact values for some of these math equations, if done correctly the approximation it produces could have a such a small relative error that accuracy should no longer be a concern. Some personal takeaways from this assignment was learning how to use header files

and including them in other files, how to get command line options using getopt(), pointers and function pointers, and how computers calculate math problems.