Assignment 2 DESIGN.pdf

Victor Nguyen

January 20, 2022

1 Description of Program:

This program numerically integrates specified functions over a given interval using composite 1/3 Simpson's rule. Some functions that you can integrate over includes:

- $\sqrt{1-x^4}$
- $\frac{1}{log(x)}$
- e^{-x^2}
- $sin(x^2)$
- $cos(x^2)$
- $log(log(x))$
- $\frac{sin(x)}{x}$
- $\frac{e^{-x}}{x}$
- e^{e^x}
- $\sqrt{sin^2(x)+cos^2(x)}$

2 Files to be included in directory "asgn2":

1. integrate.c
    - Source file that contains main() and uses parameters to calculate specific functions.
2. mathlib.c
    - Contains necessary math functions to be used for integration.
    - Professor Long restricted usage of the "offical" math library, so I was tasked to create my own.
3. mathlib.h
    - Contains the function names contained in mathlib.c.
    - Used for linking to other files that need the math library.

4. functions.c
    - Contains the functions necessary for integrate.c.
    - This was provided by the professor.
5. functions.h
    - Contains the functions necessary for integrate.c.
    - This was provided by the professor.
    - This is needed during the linking process of files.
6. test.c
    - Source file that I used to test my mathlib.c functions.
7. Makefile
    - File that formats the program into clang-format.
    - File also compiles integrate.c and test.c.
    - Cleans files that were generated during the compilation process.
8. README.md
    - A text file that's in Markdown format that describes how you would build/run the program.
9. DESIGN.pdf
    - Describes the design and thought process of the main program and the math library.

3 Pseudocode / Structure:

3.1 integrate.c

Contains the composite simpson's 1/3 rule, along with the usage page, and prints out the integrals specified by the user.
Get the user input on what function to use according to the functions stated in the description, the lower bound and upper bound of the integral, and optionally the number of partitions.
After getting the function, I should be able to specify in the simpsons integration function to use the user specified function, it's bounds, and the optional partition count.
I'm going to need switch statements that can specify what function I want to use, along with accepting user inputs for the lower bound, upper bound, and partitions.
After specifying the necessary function, bounds, and partitions, I need to feed all the info into the composite simpson's 1/3 function.
My implementation of the composite Simpson's 1/3 rule is heavy inspired by the asgn2.pdf composite simpson's 3/8 rule. The only real differences between the two is that we write a for loop that iterates from 1 to n (n being the number of partitions), and if the iterative number is divisible by 2, then we multiply 4 with the function that was specified by the user and add that to the sum. Otherwise, we would multiply 2

with the function specified and add that to the sum. Lastly, multiply the sum by the height h, and divide by 3 and return the sum.
Once we calculate the integral (using the composite simpsons 1/3 rule) we will return the value calculated and print it out along side with the partition number.

3.2 NOTE ABOUT THE PSEUDOCODE / STRUCTURE OF integrate.c:

- The default number of partitions is 100. The number of partitions will determine how many times we will run the
- Utilizing getopt was necessary during the implementation of getting the users inputs.
- I needed to implement a help page to show what options were available to use. This help message should be displayed when no values were given and invalid use of the executable file.

3.3 mathlib.c

Contains math functions needed for integrate.c. Here is what we need:

1. Exp function
    - Exp should take some floating point number x, and compute e to the power of x. Set trm (the starting point) to be 1.0, set the sum to be the same as trm, set i (the factorial) to be 1, and set epsilon (the value used to stop the function) to be 1e-14.
    - Then write a while loop that checks if trm is bigger than epsilon. While trm is bigger than epsilon, multiply trm to the absolute value of x divided by i, then store the value back into trm. Increase the sum by the trm, and lastly increment i by 1. This will calculate the taylor series of Exp until the number is smaller than epsilon.

      Afterwards, return x if it's a positive number, otherwise return the inverse of x.
2. Sin function
    - Sin will take in a floating point number x. You will need to initialize s, v, t, and k. S (the sign) should be 1.0, v (the sum) should start at x, t (the term) should start at x, and k (the factorial) should start at 3.0. We also need to initialize epsilon again.
    - Before we actually start computing the approximated value of a given x, we should scale the number down to be between [-2pi, 2pi]. The nature of a sine wave is that it oscillates between -1 and 1. So eventually the bigger numbers will match preexisting y values, and the numbers start to repeat if the numbers are lower than -2pi or bigger than 2pi. When the number is outside the range as aformentioned, we need to compute the modulus of that value with either -2pi or 2pi (depending if the original x value is a negative or not).
    - Then write a while loop that checks if the absolute value of t is greater than epsilon. While the condition is met, multiply t with x squared and divide it by k - 1 times k. Computing this will give us each iteration of the sine taylor series. Next, we need to change the sign of s to be negative to mimic the signs changing in the taylor series. Then you need to add v with s times t and store it back into v. This sums all the iterations of the sine taylor series. Lastly, add 2 to k and let that be the new k to mimic the denominator of the sine taylor series.

- You should return the value of v after the while loop terminates.

3. Cos function

   - The cos function is fairly similar to the implementation to the sin function. Cos will take in a floating point number x. Initialize s, v, t, and k as floating points. Here the s variable will act as the sign change for the cosine taylor series. V will start at 1.0, t starts at 1, and k at 2.0 (don't forget to initialize epsilon as well).
   - Before we start computing the approximated value of a given x, we also need to scale the number (like in sine) down to be between [-2pi, and 2pi].
   - Then write a while loop that checks if the absolute value of t is greater than epsilon. While the condition is met, multiply t with x squared and divide it by k - 1 times k. Computing this will give us each iteration of the cosine taylor series. Next, we need to change the sign of s to be negative to mimic the signs changing in the taylor series. Then you need to add v with s times t and store it back into v. This sums all the iterations of the cosine taylor series. Lastly, add 2 to k and let that be the new k to mimic the denominator of the cosine taylor series. (Note: it's quite literally the same as sine)

4. Square root function

   - The implementation of the square root function follows Newton's method of computing it. The square root function will first take in a floating point value x. We then initialize z, y, and f as floating point numbers. Z should start at 0.0 (z being the previous value of the search), y starts at 1.0 (the "sum" of the function), and f should start at 1.0 (more on f later).
   - Next, we should check if x is a negative value, and if it is, we return -NAN. This is because negative square roots are outside the scope of the square root function.
   - Now, similar to the cos and sin functions, we need to scale the value to make the search more efficient. Notice that the square root of 4 is 2. So we can just take out 4's from the value x and at the very end of our search we can multiply all the 2's and return our value. To do this, we need to check if x is greater than 1. If it is, divide our x value by 4 and store it back into x. Then multiply f (to keep track of how many fours we've taken out of the square root) by 2 and store that back into f.
   - Then, write a while loop that checks if the absolute value of y - z is bigger than epsilon. If that condition is met, set z to be y and then compute 0.5 * (z + x / z) and let this new value to be the new y.
   - Lastly, return the computation of y times f.

5. Log function

   - Our log function will look pretty similar to our square root function. It follows a similar search as aforementioned, so we take in a floating point value x. Initialize y, p, e, and f as floating point numbers. Y will start at 1, and p will start at e to the power of y, e starts at the actual value of e, and f will start at 0.
   - We first need to check if x is less than 0, if it is, return -NAN. By design, we will be ignoring square roots since it's outside the scope of the function. We should do the same if x is also equal to 0, and return negative infinity.

- Similar to the sqrt function, we should also scale the x value before doing any computations. Write a while loop that checks if x is greater than e, if so we will divide x with e and store it back into x. Then add 1 to f and store it back into f to keep track of how many times we've divided x by e.
- Now after scaling, we will write another while loop to check if the absolute value of p - x is greater than epsilon. While this condition is met, let y be equal to y + x / p - 1. Then set p to be equal to e raised to the power of y.
- Return y + f after the while loop terminates.

6. Modulus function

   - Computes the modulus of two floating point values (n and m). Initialize a floating point variable r, and an integer variable t. The integer will contain the value after computing n / m, and r will be the value after you compute n - (m * t).
   - Check to see if n is a negative number, you should return the absolute value of r times negative 1. Otherwise, just return the absolute value of r.

3.4 NOTE ABOUT THE PSEUDOCODE / STRUCTURE of mathlib.c:

- Instead of initializing epsilon in every function over and over, you could also define epsilon and use that value for all of the functions.
- All functions found in the math library were derived from the provided asgn2.pdf.
- The log function utilizes another function that was previously created, so it's really important that you implement Exp before the log function.
- The variable e in the log function should be the actual value of e. Inputting about 7 to 14 decimal points should be good enough.

4 Error Handling:

Some problems and solutions I found while coding.

4.1 Error handling in mathlib.c

1. Sin and Cos functions

   - Using big numbers causes inefficiency, so I scaled the number given if the value is bigger than 2pi.

2. Square root function

   - We can't compute negative values. It's not within my scope of computing, so when the function receives a negative value, it will return -NAN.
   - I also scale down the value if x is bigger than 1.

3. Log function

- We can't compute negative values or 0. It's not within my scope of computing so when the function receives a negative value, it will return -NAN, negative infinity if it receives 0.
- I also scaled the x value down if the value was bigger than e.

4.2 Error Handling in integrate.c

- There were many errors during the making of integrate.c, particularly due to me not knowing the correct syntax for the code. I fixed most of the syntax errors through trial and error, along with using the c programming language textbook.
- While the user runs the executable, if the user ever inputs nothing or invalid syntax, I will display a help page that shows them what to input.

5 Credit:

- For the math library, I mostly followed the python code that was provided in the asgn2.pdf.
- I attended Eugene's section on January 14th. He helped me understand how to start my integrate.c file. He helped me better understand getopt, switch cases, coding standards, and linking files.
- I attended Omar's section on January 13th. He helped me understand how the math functions were implemented, and how to link files.
- Professor Long's sections on numerical computation and the nature of numbers helped me understand the taylor series, and convergence. I also followed his implementations of the absolute function and modulus function on the cse13s discord server. Other things I found useful was his function pointer example on discord. That gave me ideas on how I could point at specific functions and passing it to my simpson's 1/3 composite rule.
- Audrey's example design pdf helped me structure my own design pdf. This can be found on this quarters cse13s discord server in the asgn-1-faq-n-tips channel.
- I used the collatz.c from asgn1 as reference and inspiration to implementing my switch cases.
- I've copied the Synopsis help page for the ./integrate executable. This was given through the resources repository.
- Ben's tutorial on how to compile and link multiple files together was really helpful.