Krisha Sharma

CSE13s

Winter 2022 Long

01/15/2022

Assignment 2 DESIGN.pdf

Description of Program:

This assignment is divided into 2 parts. Our first task in this assignment is to implement a small library of math functions, sin(x), cos(x), e^x, $\sqrt{x}$, and log(x). We are not allowed to use the <math.h> library to help with the first task. Each of the functions written for the first task will also halt computation using $\varepsilon = 10^{14}$. The second part of this assignment is to write a dedicated program, called integrate, that links with my implemented math library and computes numerical integrations of various functions using the composite Simpson's ⅓ rule.

Files to be included in directory "asgn2":

- functions.c
    - This file is provided and contains the implementation of the functions that my main program should integrate
- functions.h
    - This file is provided and contains the function prototypes of the functions that my main program should integrate.
- integrate.c
    - This contains the integrate() and the main() function to perform the integration specified by the command-line over the specified interval.
- mathlib.c

    - This contains the implementation of each of my math library functions.
- mathlib.h
    - This file is provided and contains the interface for my math library.
- Makefile
    - The file that formats the program into clang-format and compiles it into program executable “integrate” with makeintegrate/make all from Makefile.
- README.md
    - Text file in Markdown format that describes how to build and run the program, how the program handles erroneous inputs, and any problems encountered while developing the program.
- DESIGN.pdf
    - Describes design for the program thoroughly with pseudocode and visualizations.

Pseudocode / Structure:

- e^x function:
    - initialize term at 1.0

      sum = term

      k = 1

      while term > epsilon

          term *= abs(x) / k

          sum += term

          k += 1

      if x > 0, return the sum

      else 1 / x

- Notes: Initialize the term at 1.0 and set the sum equal to the term. Set k equal to 1. After defining and initializing all the variables necessary, begin the while loop. It is necessary to limit the while loop by making sure that the loop only runs if the absolute value of the term is greater than the value of epsilon (10^{-14}). Inside the loop multiply and assign the absolute value of x divided by k, to the term. Then add and assign the value of the term to the sum. Next, increase the value of k by 1. Outside the while loop, create a conditional where if x > 0, return the sum; else do 1 / x.

- sin function:
  - initialize all the variables at 1.0

    while the absolute value of the term > epsilon:

        term = term * (x * x) / ((k - 1) * k)

        sum = -sum (switch the sign)

        the current value of the series += sum * term

        k += 2.0

    return the current value of the series
  - Notes: Initialize all the necessary variables and then begin the while loop. It is necessary to limit the while loop by making sure that the loop only runs if the absolute value of the term is greater than the value of epsilon (10^{-14}). Afterward, set the term equal to the value of (x * x) and then divide by (k - 1) * k. Change the sign of the sum, and then take the current value of the series and add and assign it to the value of sum * term. Next, add and assign the value of 2.0 to k. Lastly, return the current value of the series.

- cos function:
    - The cos function will be exactly like the sin function; however, the starting point will be different. Rather than starting at 0, as in sin, the cos function will start at 1.
- $\sqrt{x}$ function:
    - initialize z at 0

      initialize y at 1

      while the absolute value of (y - z) > epsilon

      z = y

      y = 0.5 * (z + x / z)

      return y
    - Initialize the variable z at 0 and the variable y at 1 in order to account for the upper and lower limits. The z is the current term and the y is the next term. After initializing all the variables, begin coding the while loop. Limit the while loop by making sure that the loop only continues if the absolute value of (y - z) is greater than the value of epsilon. Inside the loop, set the value of z equal to the value of y, and then calculate y by doing (z + x / z) and multiplying that by 0.5 since we
- log function:
    - initialize y at 1

      p = exp(y)

      while abs(p - x) > epsilon

      y = y + x / p - 1

      p = exp(y)

      return y

- Notes: Initialize y at 1, and set p equal to the exp(y). After initializing all the variables, begin coding the while loop and limit the loop so that the loop only runs if the absolute value of (p - x) is greater than the value of epsilon. Inside the loop set y equal to y + x / p - 1. Then set p equal to the value of exp(y). Break outside the while loop and then return y.

- integrate function
  - int main (int argc, char **argv)

    initialize opt at 0

    initialize all the case bool's (a, b, c, d, e, f, g, h, i, j, n) as false

    set the default low to 0

    set the default high to 10000

    set the default partition to 100

    while ((opt = getopt(argc, argv, "abcdefghijn:p:q:H")) != -1)

    begin the switch case

    case 'a':

    set case a bool to true

    break

    case 'b':

    set case b bool to true

    break

    case 'c':

    set case c bool to true

    break

```
case ‘d’:
        set case d bool to true
        break
case ‘e’:
        set case e bool to true
        break
case ‘f’:
        set case f bool to true
        break
case ‘g’:
        set case f bool to true
        break
case ‘h’:
        set case h bool to true
        break
case ‘i’:
        set case i bool to true
        break
case ‘j’:
        set case j bool to true
        break
case ‘n’:
        set case n bool to true
```

```
                define partition as an atoi optarg
                break
        case ‘p’:
                define low as a string to decimal optarg
                break
        case ‘q’:
                define high as a string to decimal optarg
                break
        case ‘H’:
                write out the usage and synopsis in print statements
                break
        default:
                print out an error message
                return 1
if (bool a)
        set the count to 2
        print (“sqrt(1-x^4)”, low, high, partition)
        while the partition is higher than the count
                if partition is greater than or equal to the value of count
                        print (count, integrate(a, low, high, count))
                        increment the count by 2
                if the partition is equal to the count
                        print (count, integrate(a, low, high, count))
```

break

repeat the same if loop for all the other cases b-j

return 0

- Notes: The integrate function that I will write, links with my implemented math library and computes numerical integrations of various functions using the composite simpson's ⅓ rule. The integrate function itself computes the numerical integration of some function "f" over some interval "[a, b]".

Error Handling:

- I ran into an error while programming my math function for cos. Rather than have the term initially set to 1.0, I had my term set to x. When I was testing my code this was causing my cos function values to have a large difference when compared to the cos values I got from the actual math library (math.h). I realized that I needed to change where I initialized "term" as x to 1.0 because when looking at the equation for cos, the equation begins at 1.0 rather than at x. When I did this the difference between the values outputted from my cos function and the actual math library cos function became very small.
- As I was programming my square root and log functions, I ran into an error where my log function was stuck in an infinite loop because I had not yet scaled my log function code. After implementing scaling the error went away and the difference became small.
- While testing my integrate function and getopt loop, I ran into an error. When I called ./integrate -a -p 0.0 -q 1.0 -n 10, everything would work as expected; however, when I called ./integrate -b -p 2.0 -q 3.0 -n 10, the program would output incorrect values. Because my previous test worked as expected, I knew that my issue was somewhere in

my log(x) function. When I went back to look over it in mathlib.c I realized that rather than adding the offset back into my sum I was multiplying it in. Once I fixed the error my values were generated correctly.

Cite:

- The code for my math library is based on the python pseudocode provided by the professor in the assignment 2 document. I converted the code from python to C in order to make my own math library. The code for my composite simpson's ⅓ rule is also based on the example python code that the professor provided in the assignment document.
- I attended Eugene's section on the 14th of January, 2022, and during it, he showed me how to create my Makefile as well as how to make a getopt loop when needed.
- I attended Brian's tutoring section on the 17th of January, 2022, at 12 pm. The tutoring section was one on one and during it, he showed me how to get started on part two of assignment 2, specifically the integrate function and the way it incorporates composite simpson's ⅓ rule. He also went over how to link files in my Makefile and the way that files such as functions.c, integrate.c, and mathlib.c will need to first be converted into object files and then linked together afterward. Brian also helped me understand how to scale my log and square root functions and how an offset is added or multiplied to the end sum to account for larger numbers.
- I attended Mile's section on the 18th of January, 2022, at 9 am, and during it, he helped me figure out how to get started on my getopt loop.
- I attended Ben's section on the 18th of January, 2022, and during it, he helped me understand how to structure my cases and the way the getopt loop functions to link together multiple arguments.

- Most of the code for my getopt loop was taught to me by Eugene. In his section video on the 14th of January, he showed me how to create the getopt loop as well as other hints, tricks, and strategies I would need to combine to successfully complete part 2 of assignment 2.
- I attended Brian's section on the 19th of January, 2022, and during it, he helped me solve an error I encountered where my actual integrate(a, low, high, partition) command was not running due to the fact that I declared low and high as integers rather than doubles, and used the atoi command instead of strtod. Brain pointed out that I needed to specify what type of elements the low, high, and partition were when I was trying to print out the first line of my output statement. He told me that I needed to use %f to show that low and high were doubles and %d to show that partition was an integer when I was writing out the print statement that specified what type of operation was being performed. After Brian pointed this out I was able to fix my error and my print statement now looks like printf("sqrt(1-x^4) , %f,%f,%d\n", low, high, partition).
- I attended Audrey's section on the 19th of January, 2022 at 4 pm, and during it, she answered my question on to test π and -π on the command line. She told me that I should test that part of my integrated function by approximating π as 3.14 or -3.14.
- I collaborated with my sister Twisha Sharma on this assignment, specifically, there was high-level pseudocode collaboration on the math library functions.