Assignment 2 Design Document

Nicholas Chu

January 2022

1 Purpose

The purpose of this assignment is to first write functions that mimic some of the math functions found in the math library math.h in a file called mathlib.c. Then write another program called integrate.c that creates a function, integrate(), which integrates by applying the composite Simpson's 1/3 rule. The main program will integrate certain equations based on command line options.

2 Files

1. functions.c: A provided file that contains the implemented functions mimicking those in math.h to be used in the main program.
2. functions.h: A provided file containing the function prototypes that the main program should integrate.
3. integrate.c: Contains the function integrate() and in the main() functions it preforms the integration based on command line inputs.
4. mathlib.c: Contains the implementation of the math functions definined in functions.c.
5. mathlib.h: Provided file that contains the interface for the math library.
6. Makefile: Should format and build all the source code and header files, as well as clean all files the compiler-generated.
7. README.md: A brief description of the program and how to compile and run it. It should also contain the command line options for the programs and any known errors or bugs.
8. DESIGN.pdf:This current document, meant to describe the design process and how to replicate the implementation of the programs.
9. WRITEUP.pdf: Must describe and analyze the graphs produced by the programs and gnuplot, along with what was learned about floating-point numbers when completing this assignment.

3 Pseudocode

3.1 mathlib.c

These functions mimic functions in the math.h library and the approximated value they produce should have a margin of error less than $\epsilon = 1.0 * 10^{-14}$. Much of the implementation of these functions are based on the specifications found within Assignment 2. Below is a basic and quick summary of what each function will do and how the code is set up.

- double Exp(double x): Returns an approximated value for e^x, and does this through using the equation: $e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!} = 1 + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!}$... To avoid using another loop inside of our while loop to completely recalculate the factorial with each new term, which would cause long and unruly run times, all we have to do is use the idea that $\frac{x^k}{k!} = \frac{x^{k-1}}{(k-1)!} \times \frac{x}{k}$, where we save the previous term and just multiply it with the x over the current k and get the current term. This works because the power of the next term is increasing by 1. To check for accuracy and knowing when to stop is checking when the current term is less than epsilon as that amount of accuracy should be sufficient. Another thing to account for is when x would be negative which would still use the same process but just need to put the calculated value over 1 when returning. Below is what the function should look like based on the explanation above:

```
k = 1
current term = 1
previous term = 1
e^x = 1
while (previous term is greater that epsilon):
    #Finding current term.
    current term = previous term * absolute value of x / k
    #Adding current term to the sum.
    e^x += current term
    #Increment k.
    k++
    #Current term will be previous term during the next iteration.
    previous term = current term
if x < 0:
    e^x = 1/e^x
return e^x /* Returns approximated value for e^x*/
```

- double Sin(double x): Returns an approximated value of sin(x), and does this using the equation $sin(x) = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k+1}}{(2k+1)!} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + ...$ It uses similar concepts like the above. The while loop will continue to calculate terms until the term is less than epsilon to get a more accurate approximation. In that loop to get the current term from the previous term, you would multiply the previous term by $\frac{x^2}{(k-1)(k)}$. Something specific to note about this summation is that it is alternating between adding and subtracting terms and to do implement this we can multiply the current term with either ± 1 to turn the term negative or positive and add that result to the sum. We set the positive or negative sign at the beginning to a -1 since it starts by subtracting the second term, then at the end of each iteration of the while loop multiplies that sign by -1 to switch it. Below is what the function should look like based on the explanation above:

```
k = 3
current term = x
previous term = x
sin(x) = current term
sign = -1
while (previous term is greater that epsilon):
    current term = previous term * x^2/((k-1)*k)
    sin(x) +=  sign * current term  #Adding current term to the sum
    #Increment k, in this case as seen from the equation you need 2 not 1.
    k+=2
    previous term = current term
    sign *= -1 #Swap the sign
return sin(x)
```

- double Cos(double x): Returns an approximated value of cos(x), and does this using the equation $cos(x) = \sum_{k=0}^{\infty}(-1)^k \frac{x^{2k}}{(2k)!} = \frac{x^{2k+1}}{(2k+1)!} = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + ...$ The implementation is very similar to sign, the only significant difference would be the starting term. Below is what the function should look like based off the explanation above:

```
k = 2
current term = 1
previous term = 1
cos(x) = current term
sign = -1
while (previous term is greater that epsilon):
    current term = previous term * x^2/((k-1)*k)
    cos(x) += sign * current term
    k+=2
    previous term = current term
    sign *= -1
return cos(x)
```

- double Sqrt(double x): Returns an approximated value of $\sqrt{x}$ using the Newton-Raphson method, $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$. By using the Newton-Raphson method and applying it $f(x) = \sqrt{x} - y$ (which is done to find the root), we get the equation $x_{k+1} = \frac{1}{2}(x_k + \frac{y}{x_k})$ to use in the while loop. Along with this there will need to be some condition to check if the x is negative since you can't square a negative number and have the function return something like undefined or imaginary when that happens. It is also important to implement scaling when creating this function so it runs better when working with larger x values.

```
#First simplifying the square root before getting the approximated value
while x is greater than 1:
    x = x/4
    factor = factor * 2

x_{k} = 0 #x(k) starting point at 0
y = 1 #Think of this as the next term
while absolute value of y-x_{k} is greater than epsilon:
    x_{k} = y
    y = (1/2)*(x_{k} + x / x_{k}) #Used to find x_{k+1}
return factor * y #Returns approximated value of sqrt(x)
```

- double Log(double x): Returns an approximated value of log(x) using the Newton-Raphson method. By using the Newton-Rahpson method and applying it to $f(x) = y - e^x$, we get the equation $x_{k+1} = x_k - \frac{y-e^{x_k}}{-e^{x_k}} = x_k - (-\frac{y-e^{x_k}}{e^{x_k}}) = x_k + \frac{y}{e^{x_k}} - 1$. To use e^x in our function we will be using the already defined Exp() function. Scaling x would also be different in log. In this case we are factoring out Euler's constant from log and adding the number of times it was factored out of log(x) and added to the final sum. Otherwise same things that applied to the Sqrt() function also apply here.

  ```
  factor = 0
  while x is greater than 1:
      x = x/Euler's constant
      factor += 1

  x_{k} = 1
  while absolute value of e^x-x_{k} is greater than epsilon:
      x_{k} = x_{k} + y / e^x - 1 #saving next term of x_{k} into itself
  return factor + x_{k} #Returns approximated value for log(x)
  ```

3.2 integrate.c

The file will contain the integrate() function and its main() should contain the definitions for command line options.

- double integrate(double (*f)(double), double a, double b, uint32_t n): The function will integrate relying on the composite Simpson's 1/3 Rule:

$$\int_a^b f(x)dx \approx \frac{h}{3}\sum_{j=1}^{n/2}[f(x_{2j-2}) + 4f(x_{2j-1}) + f(x_{2j})]$$

$$= \frac{h}{3}[f(x_0) + 2\sum_{j=1}^{n/2-1} f(x_{2j-1}) + 4\sum_{j=1}^{n/2} f(x_{2j}) + f(x_n)]$$

The inputs for the function are:

1. f*: A pointer to a function that takes and returns a double.
2. a: The beginning of the interval that is being integrated over.
3. b: The endpoint of the interval that is being integrated over.
4. n: The number of desired partitions.

To break this equation down a bit so we can implement it into our code, h/3, h being step length and equal to (b-a)/n, is multiplied by the sum of $f(x_0) + 2\sum_{j=1}^{n/2-1} f(x_{2j-1}) + 4\sum_{j=1}^{n/2} f(x_{2j}) + f(x_n)$. $f(x_0)$ is the function (in this case f*) at the starting point using the input of a, and $f(x_n)$ is the function endpoint using b as the input. The summation $2\sum_{j=1}^{n/2-1} f(x_{2j-1})$ is the summation of all the odd terms (since 2j-1 is always odd) between 0 and n while $4\sum_{j=1}^{n/2} f(x_{2j})$ is the summation of the even terms between 0 and n (2j always being equal). The function will have a for loop that will run n times, for number of partitions. The pseudocode for the implementation is below and is a modified version of the code for the implementation of the Simpson's 3/8 rule in the Assignment 2 document which now implements Simpson's 1/3 Composite rule.

```
h = (b-a)/n #Calculates step length
simpsonsum = f(a) + f(b) #Start off by adding f(x_{0}) and f(x_{f})
#Now getting summations
for j in range from 1 to n:
    #j*h+a is to get the current step length from the beginning
    if j % 2 == 0:
        sum += 2*f(j*h+a)
    else:
        sum += 4*f(j*h+a)
#Multiplying h/3 constant outside of the brackets
sum = sum * h /3
return simpsonsum
```

- main(): This will contain the implementation of command-line options for integrate.c using getopt(). The following command line options that needed to be included are:
  1. -a: Sets the function for integration to $\sqrt{1-x^4}$. This uses Srqt(1+x*x*x*x).
  2. -b: Sets the function for integration to 1/log(x). This uses 1/Log(x).
  3. -c: Sets the function for integration to e^{-x^2}. This uses Exp(-1*(x*x)).
  4. -d: Sets the function for integration to $sin(x^2)$. This uses Sin(x*x).
  5. -e: Sets the function for integration to $cos(x^2)$.This uses Cos(x*x).
  6. -f: Sets the function for integration to log(log(x)). This uses Log(Log(x)).
  7. -g: Sets the function for integration to sin(x)/x. This uses Sin(x)/x.
  8. -h: Sets the function for integration to e^{-x}/x. This uses Exp(-x)/x.

9. -i: Sets the function for integration to e^{e^x}. This uses Exp(Exp(x)).
10. -j: Sets the function for integration to $\sqrt{sin^2(x) + cos^2(x)}$. This uses Sqrt(Sin(x)*Sin(x)+Cos(x)*Cos(x)).
11. -n partitions: Sets n, number of partitions. Should have a default value of 100.
12. -p low: Sets a, the low end of the interval. Should have no default value.
13. -q high: Sets b the high end of the interval. Should have no default value.
14. -H: Displays usage and synopsis of program.