Assignment 2 Design

Nhan Nguyen

January 2022

1 Introduction

1.1 Program Description

The purpose of this program is to calculate the most accurate result of an integration from **low** range to **high** range of any equation listed below.

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

We will be using Simpson's 1/3 rule to calculating the integration of all these equation. The Simpson's 1/3 rule is as follows:

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

Additionally, to help calculating the Simpson's 1/3 rule, we will be writing functions to calculate the **x-value** of each math library functions, these functions are:

- $|x|$ (already provided)
- e^x
- $sin(x)$
- $cos(x)$
- $\sqrt{x}$
- $\log x$

1.2 List of files

- **functions.c:** This provided file contains the implementations of all the equations mentioned above in C code
- **functions.h:** This provided file contains the prototype functions of the all the equations mentioned above
- **integrate.c:** This file contains the integrate() function as well as the main() function to do the integration of all the equations with command-lines and C code
- **mathlib.h:** This provided file contains the prototype functions of my math library functions
- **mathlib.c** This file contains the implementations of the my math library functions
- **Makefile:** This file formats program into clang-format and also compiles it. In addition, this makefile will also link functions.c integrate.c, and mathlib.c.
- **README.md:** A text file in Markdown format that describes how to build the program, as well as describing the problems encountered while making the program
- **DESIGN.pdf:** This PDF file describes the idea behind the program with visualization and pseudocode
- **WRITEUP.pdf:** This PDF file contains all the graphs displaying the integrated values with varying partitions, as well as a writeup on analysing the graphs and lessons learned from floating-point numbers

2 Math library functions

Note: Most of these functions utilizes the Taylor series to produce a result that is as accurate as possible when compared to the actual results produced by **C's Standard Library Functions**. In addition, all the calculation will halt computation using **epsilon** $= 10^{-14}$

2.1 e^x

- This is the simplest function when it comes to producing a Taylor series since $f^{(k)}(x) = f^{(k-1)}(x) = f^{(k-2)}(x) = ... = f'(x) = e^x$. When centered at 0, the Taylor series will be:

$$e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!} = 1 + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + ...$$

- However when it comes to coding the equation, we will run into a problem when calculating **k!** due to how factorials can increase very quickly. We can solve this problem with this simple observation:

$$\frac{x^k}{k!} = \frac{x^{(k-1)}}{(k-1)!} \cdot \frac{x}{k}$$

- Note that this is an iterative process since we can calculating the current term by utilizing the result we got from the previous term. We start with the first term being k! = 0! = 1, then compute the next terms by repeatedly multiply the previous term by $\frac{x}{k}$

- **Pseudocode:**

```
double exp(double x)      double terms = 1.0
    double epsilon = 1e-14
    double sum = terms
    double k = 1
    while terms > epsilon do
        terms *= abs(x) / k
        sum += terms
        k++
    if x > 0 do
        return sum
    else do
        return 1 / sum
```

2.2 $sin(x)$ and $cos(x)$

- This 2 functions are also some of the more simpler ones when it comes to producing a Taylor Series, thanks to their nature of repeating over the interval $[-2\pi, 2\pi]$. When centered at 0, their series are as follows:

$$sin(x) = \sum_{k=0}^{\infty} (-1)^k \frac{x^{(2k+1)}}{(2k+1)!} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - ...$$

$$cos(x) = \sum_{k=0}^{\infty} (-1)^k \frac{x^{(2k)}}{(2k)!} = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - ...$$

- The process of calculating the Taylor series will also be iterative, and the process will continue until the last term is less than the acceptable error (i.e less than the epsilon value mentioned earlier)

- **Pseudocode for** $sin(x)$**:**

```
double sin(double x)
    double sign = 1.0
    double total = x
    double terms = x
    double k = 3.0
    double epsilon = 1e-14
    while terms > epsilon do
        terms *= (x * x) / (k * (k - 1))
        k += 2
        signs *= -1
        total += s * terms
    return total
```

- **Pseudocode for** $cos(x)$**:**

```
double cos(double x)
    double sign = 1.0
    double total = 1.0
    double terms = 1.0
    double k = 2.0
    double epsilon = 1e-14
    while terms > epsilon do
        terms *= (x * x) / (k * (k - 1))
        k += 2
        signs *= -1
        total += s * terms
    return total
```

2.3 $\log(x)$

- We might think that a function like this can be calculated through the Taylor series used in the other functions mentioned above, And indeed the series exist, since log(0) is undefined, we're going to use $\log(x+1)$:

$$log(x+1) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - ...$$

- The series does exist, however it converges very slowly due to how slow the denominator is increasing. As a result, we will be using Newton's method to produce a better approximation:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

- Let's now focus on calculating the $log(y)(ln(y))$. To calculate that, we're going to need to find the inverse of $log(y)$ which is e^x since $log(e^x) = x$. Therefore to find $log(y)$ we need to find the number **x** such that $e^x = y$. We can express this equation as finding the root of $f(x) = y - e^x$

- Plugging everything to our Newton's method equation we get:

$$x_{k+1} = x_k - \frac{y - e^{x_k}}{e^{x_k}} = x_k - \frac{y}{e^{x_k}} - 1$$

- **Pseudocode:**

```
double log(double x)
    double epsilon = 1e-14
    double y = 1.0
    double p = exp(y)
    while abs(p - x) > epsilon
        y = y - y / p - 1
        p = exp(y)
    return y
```

2.4 $\sqrt{x}$

- When we tried to solve $\sqrt{x}$ using the Taylor series, we encountered a problem

$$f'(x) = \frac{1}{2\sqrt{x}}$$

$$f''(x) = \frac{1}{4\sqrt{x^3}}$$

$$f'''(x) = \frac{1}{8\sqrt{x^5}}$$

- We noticed that no matter how many times we take the derivative of $\sqrt{x}$, we can not get rid of the square root. From here, we noticed that we need to use Newton's method once again to calculate $\sqrt{x}$
- Now let's focus on calculating $\sqrt{y}$. To do that, we need to find the inverse of $\sqrt{x}$, which in this case is x^2. Therefore to solve $\sqrt{y}$ we need to find an number **x** that satisfy the equation $x^2 = y$. Express the equation as finding the root of $f(x) = y - x^2$, we can now plug the equation into our Newton's method equation

$$x_{k+1} = x_k - \frac{x_k^2 - y}{2x_k} = x_k - \frac{x_k}{2} - \frac{y}{2x_k} = \frac{x_k}{2} - \frac{y}{2x_k} = \frac{1}{2}(x_k - \frac{y}{x_k})$$

- **Pseudocode:**

```
double sqrt(double x)
    double epsilon = 1e-14
    double z = 0.0
    double y = 1.0
```

```
while abs(y - z) > epsilon
    z = y
    y = (z + x / 2) / 2
return y
```

3 Integrate()

3.1 Idea

- There are alot of ways to calculate an integral, trapezoid, left, right, and mid Riemann sum. For this particular algorithm, we will be using Simpson's rule, particularly Simpson's 1/3 rule, which is the following:

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

- However, this formula can only give the a good result if the function is smooth over the interval $[a...b]$. When it comes functions that are not smooth, we're going to use the Composite Simpson's 1/3 rule, which is the following given **n** is even

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

with $j = a + jh$ for $j = 0, 1, 2, ..., n-1, n$ and $h = \frac{b-a}{n}$

3.2 Pseudocode

```
double integrate(double pointer to the any function in functions.c, double a,
double b, unsigned int 32 bits n)
    double term1 = 0.0
    double term2 = 0.0
    double term3 = 0.0
    double h = (b - a) / n
    for j from 1 to n / 2 do
        term1 += (pointer to function)(a + ((2 * j - 2) * h))
        term2 += (pointer to function)(a + ((2 * j - 1) * h))
        term3 += (pointer to function)(a + (2 * j * h))
    term2 *= 4
    return (h / 3) * (term1 + term2 + term3)
```

4 Main()

4.1 Idea

Along with integrate(), the file also contain code that takes in commands in the command line using getopt(), the commands the program accepted are:

- -a: calculating the integration of $\sqrt{1-x^4}$ in the interval $[low...high]$, with **n** partitions
- -b: calculating the integration of $\frac{1}{\log x}$ in the interval $[low...high]$, with **n** partitions
- -c: calculating the integration of e^{-x^2} in the interval $[low...high]$, with **n** partitions
- -d: calculating the integration of $\sin(x^2)$ in the interval $[low...high]$, with **n** partitions
- -e : calculating the integration of $\cos(x^2)$ in the interval $[low...high]$, with **n** partitions
- -f : calculating the integration of $\log(\log(x))$ in the interval $[low...high]$, with **n** partitions
- -g : calculating the integration of $\frac{\sin(x)}{x}$ in the interval $[low...high]$, with **n** partitions
- -h : calculating the integration of $\frac{e^{-x}}{x}$ in the interval $[low...high]$, with **n** partitions
- -i : calculating the integration of e^{e^x} in the interval $[low...high]$, with **n** partitions
- -j : calculating the integration of $\sqrt{sin^2(x)+cos^2(x)}$ in the interval $[low...high]$, with **n** partitions
- -n partitions: set the number partitions the integration will use when doing the composite Simpson's 1/3 rule, default value is 100
- -p low: set the low end of the interval the integration will use when doing the composite Simpson's 1/3 rule, no default value
- -q high: set the high end of the interval the integration will use when doing the composite Simpson's 1/3 rule, no default value
- -H: display synopsis and how to run the code

4.2 Pseudocode

```
int main()
    int segments = 100
    double low
    double high
    function pointer pass

    get command
    if command is a do
        pass = function a in functions.c
    if command is b do
        pass = function b in functions.c
    if command is c do
        pass = function c in functions.c
    if command is d do
        pass = function d in functions.c
    if command is e do
        pass = function e in functions.c
    if command is f do
        pass = function f in functions.c
    if command is g do
        pass = function g in functions.c
    if command is h do
        pass = function h in functions.c
    if command is i do
        pass = function i in functions.c
    if command is j do
        pass = function j in functions.c
    if command is n do
        read number
        segments = number
    if command is p do
        read number
        low = number
    if command is q do
        read number
        high = number
    if command is H do
        display synopsis and usage

    for i from 2 to segments with step 2 do
        integrate(function pointer pass, low, high, i))
```