Assignment 2 Design

Arnav Nepal

January 30, 2023

1 Introduction

In this assignment we will be implementing a small number of mathematical functions in C in order to compute the fundamental constants e and π. The two mathematical functions we will be implementing are e^x and $\sqrt{x}$. We are not allowed to use math library functions or make our own factorial function. Instead, we will be using a taylor series to approximate each of the required mathematical functions. Files we will be creating include:

- e.c
- madhava.c
- euler.c
- bbp.c
- viete
- newton.c
- mathlib-test.c
- Makefile

2 Design and Psuedocode

2.1 e.c

This file will contain my implementation of e (Euler's number). The taylor series for e is :

$$e = \sum_{k=0}^{\infty} \frac{1}{k!}$$

While this is a complex series, we can simplify the process of calculating it. To find the kth term in the taylor series, we can use the formula:

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

keeping these formula's in mind, the design for e.c:

```
###This function is a translation of python code as provided by Dr.Long
include relevant libraries and files
global var numterms
def ϵ
func e(x):
    var orig = 1 #1st (original) term, also holds x/(k−1)!
    var sum = 1 #taylor series starts with 1
    k = 1 # counter for summation
    while absolute(orig) > ϵ:
        orig = orig ×x/k
        sum += orig
        k++
    set terms to sum
    return sum
func e_terms():
    return numterms
```

2.2 newton.c

This file will contain my implementation of square root using the Newton-Raphson method of approximation. sqrt_newton() will return the square root and sqrt_newton_iters() will return the number of iterations square root took to converge.

```
###This function is a translation of python code as provided by Dr.Long
include relevant files
global var iters
def ϵ
func sqrt_newton():
    scale = 1 #scale to be used to hold multiplier after normalization
    y = 1 #intial guess
    while x > 4: #normalization
        x = x/4 #removes multiples of 4 since √4 = 2
        scale = scale*2 #multiplies scale by 2 each time 4 removed
    guess = 0 #initial guess
    while absolute(y - guess) > ϵ:
        guess = y #holds previous value for y
        y = ( y + x/y) / 2 #newtons method
        increment iters
    return scale * y
func sqrt_newton_iters():
    return iters
```

2.3 madhava.c

This file will contain the C code for my implementation of π approximation using the madhava series. The madhava series can be expressed as:

$$\sqrt{12}\sum_{k=0}^{\infty}\frac{(-3)^{-k}}{2k+1}=\frac{1}{(2k+1)(-3)^{k}}$$

```
include relevant files
global var terms
def ϵ
func pi_madhava():
    var orig = 1 # holds first term and successive terms
    var k # counter for k
    var sum #holds total sum
    var exp #holds the exponent
    while absolute(orig) > ϵ:
        multiply exp by -1/3
        orig = (1/(2k+1)) * exp
        add orig to sum
        increment k
    set terms to k
    multiply sum by √12 using sqrt_newton()
    return sum
func sqrt_madhava_terms():
    return terms
```

2.4 euler.c

This file will contain the C code for my implementation of π approximation using the euler series. The euler series can be expressed as:

$$\sqrt{6\sum_{k=1}^{\infty}\frac{1}{k^2}}$$

Since we can square root the sum at the end, let us remove it for now:

$$6\sum_{k=1}^{\infty}\frac{1}{k^2}$$

This is a relatively simple sum to calculate and does not need simplifying, since $k^2 = k \times k$.

```
include relevant files
```

```
global var terms
def ϵ
func pi_euler():
    var temp = 1 #1st terms is 1
    var k = 1 #starts one 1 to prevent dividing by zero
    var sum #holds total sum
    while absolute(temp) > ϵ:
        temp = 1/(k×k)
        add temp to sum
        increment k
    set terms to k
    mutiply sum by 6
    square root sum
    return sum
func sqrt_euler_terms():
    return terms
```

2.5 bbp.c

This file will contain the C code for my implementation of π approximation using the Bailey-Borwein-Plouffe Formula. The formula is long and complex, and can be represented in a reduced form for the least number of multiplications as:

$$\sum_{k=0}^{\infty} 16^{-k} \times \frac{k(120k+151)+47}{k(k(k(512k+1024)+712)+194)+15}$$

```
#this should follow the same process as the other sums
include relevant files
global var terms
def ϵ
func pi_bbp():
    var temp = 318/39312 #the first term
    var k = 0 #counter
    var sum
    var exp #holds exponent each loop
    while absolute(temp) > ϵ:
        multiple exp by 1/16
        temp =  exp × next term (see above)
        add temp to sum
        increment k
    set terms to k
    return sum
```

```
func sqrt_bbp_terms():
    return terms
```

2.6 viete.c

This file will contain the C code for my implementation of π approximation using the viete series. Unlike the other formulas, this series is product series. The series can be expressed as:

$$\frac{2}{\pi} = \prod_{k=1}^{\infty} \frac{a_k}{2}$$

Where $a_1 = \sqrt{2}$ and $a_k = \sqrt{2 + a_{k-1}}$ for all $k > 1$. Since we are approximating π, we can treat π as a variable and solve for π once we get an a good approximation of $\frac{2}{\pi}$

```
include relevant files
global var terms
def ϵ
func pi_viete():
    var orig = √2/2 #term at a_1
    var k
    var product
    var temp #holds temporary values
    var prev #holds previous product for comparison in while()
    while absolute(product - previous) > ϵ:
        set prev to current product to get diff. for while() next loop
        calculate the numerator for viete using orig
        divide numerator by 2 to get next term
        multiply next term to product
        increment k
    set terms to k
    product = divide 2 by the product
    return product
func sqrt_viete_terms():
    return terms
```

3 Design changes

I had to make a multitude of changes in my program design from my initial draft. One of my biggest mistakes from my initial design was expecting the rest of the functions to mirror square root, which caused me to try to find a way to calculate the current term from the previous term. In hindsight, this did not make sense since approximating the root of something is very different from finding π. I also severely simplified viete initially, which caused trouble later on since viete was much more complicated than I originally thought.