Assignment 2 – Slice of Pi

Priya Chaudhari

CSE 13S – Spring 2023

Purpose

This program contains 7 mathematical functions (including e, BBP, Madhava, Euler, Vi'ete, Wallis, Newton) without using a math library that allows us to compute the major math functions e and Pi. These functions are supposed to mimic the math.h library without exactly utilizing it within our function files other than comparing our results with the math library's. We will have an additional test harness file named mathlib-test that will execute the outputs and their formats of the function files.

How to Use the Program

In order to use my program, the user must first run the command "make" within the terminal. Next ./mathlib-test (with a command-line option) should be entered into the terminal. Of course, the user must input a specific command line which should either be an a, e, b, r, v, w, n, m, s, or h (purposes specified below) and if neither of the options is inputted, the default will be the help message. If the user wants to see the terms and statistics alongside the tests the -s command line should be followed with whatever specific formula they want to see.

- -a: Runs all tests.
- -e: Runs e approximation test.
- -b: Runs Bailey-Borwein-Plouffe PI approximation test.
- -m: Runs Madhava PI approximation test.
- -r: Runs Euler sequence PI approximation test.
- -w: Runs Wallis PI approximation test.
- -v : Runs Viète PI approximation test.
- -n: Runs Newton-Raphson square root approximation tests.
- -s: Enable printing of statistics to see computed terms and factors for each tested function.
- -h: Display a help message detailing program usage.

Program Design

The structure and overall design of my program involves several files which contain mathematical functions. These functions are utilized in my test harness named mathlib-test.c which acts as the main executable of my program. It is responsible for printing and displaying all the statistics and outputs, formulated in their respective function files, which are seen by the user in terminal.

- e.c: This file contains the implementation of the Taylor series to approximate the e from the Euler's number and will return the number of computed terms.

- euler.c: This file will contain Euler's solution which is used to approximate Pi and will return the number of computed terms.
- madhava.c: Contains the implementation of the Madhava series to approximate Pi and returns the number of computed terms.
- bbp.c: Will contain the Bailey-Borwein-Plouffe formula which approximates Pi and the function and will return the number of computed terms.
- newton.c: Implements the square root approximation using Newton's method and function and will return the number of iterations.
- viete.c: Implements Viète's formula to approximate Pi and the function to return the number of computed terms.
- wallis.c: Implements Wallis's formula to approximate Pi and the function to return the number of terms/factors.
- mathlib-test.c: Contains my main() function which is the test harness to test each of my math library functions.
- mathlib.h: The header file which contains every function within my program.
- MakeFile: Compiles my program and will basically define the set of tasks that need to be executed for my program to work.
- README.md: A file in markdown format that will describe how to use my Makefile and program.
- DESIGN.pdf: The PDF you're reading now describes the design of my program and the design process for it to be replicated.

Data Structures

The structure of my function files typically follows this format.

- 1) The first function of each file will typically contain the mathematical execution of the formula I am translating.
- 2) The second function in each file will typically contain either "terms" or "factors" which is responsible for counting the iterations/terms that are computed with enacting each function.

I'm mostly utilizing double and static variables and my functions either start with int or double, and of course, the one exception would be my main function in mathlib-test.c. I am storing the options passed into the function through a switch case structure and utilizing the getopt function in my main function.

Function Descriptions

- e.c: calculates e using the Taylor series and the number of terms that are calculated with executing e. No input is required besides the command-line option that executes this (e or a).

```
define static double sum and static int numTerms = 0
double e(void) function
Have 2 double statements (double fact and addTerm) both are equal to 1
but in addTerm divide 1 with fact
           While Loop (addTerm is greater than EPSILON)
                 Increment NumTerms
                 Add each addTerm to sum and set them equal
                 Multiply each fact with numTerms and set them equal
                Divide 1 with fact and assign it with addTerm
```

```
            Return Sum
  Int e terms
     return Iterations
```

- euler.c: approximates the value of Pi using Euler's solution and tracks the number of terms computed. No input is required by the user besides the command-line option that executes this (r or a).

```
        Static double iteration = 0
    double pi_euler function
    2 double statements: double term initialized to 1 and double initialized to 0
        For Loop (double k =1; absolute(term) > EPSILON; k+= 1)
                Set term equal to 1 divided by k * k
                Add term and set it equal to ans
        Iterate through loop outside for loop to avoid increased difference
        + more terms
        Return sqrt_newton of 6 and multiply it to ans
    Int pi_euler_terms
        Return iteration
```

- madhava.c: approximates the value of Pi using the Madhava series and tracks the number of terms computed. No input is required by the user besides the command-line that calls this (m or a).

```
        Static double iteration = 0
Double pi_madhava function
3 double variables are defined: negative which is = to -3, sum and x which are = to 1
        For loop (double k =1; absolute(term) > EPSILON; k+= 1)
                X = 1 / 2 times k +1 all multiplied by the double variable negative
                Sum added and equal to x
                Negative multiplied and equal to -3
                Iterate through loop with iteration
        Multiply sum with sqrt_newton(12)
        Return sum
Int pi_madhava_terms
        Return iteration
```

- bbp.c: approximates the value of Pi using the Bailey-Borwein-Plouffe formula and tracks the number of terms computed. No input is necessary besides either the b or a command-line option.

```
        Static double iteration equal to -1
Double pi_bbp function
Have four double variables: numerator, denominator, answer, term all of which are
equal to 0 but term is equal to 1
        For loop (double k =0; absolute(term) > EPSILON; k+= 1)
                Numerator = (k * ((120 * k) +151) +47)-  the numerator component
  of the bbp function
                Denominator = (k*(k*(k*((512 *k) +1024) + 712) +194) +15) - the
denominator component of the bbp function
                Answer added and set equal to term multiplied by numerator divided
  by denominator
                Term divided and equal to 16
                Iterate through the loop
        Return answer
Int pi_bbp_terms
```

```
        Return iteration
```

- viete.c: approximates the value of Pi with Viete's formula and tracks the number of terms computed. Command-line options a or v are necessary inputs for this file and its functions.

```
    Static double factors equal to 0
    Double pi_viete function
        Double total = 1
        Double last_term = 0
        For Loop (double term = sqrt_newton(2); absolute(2- term) > EPSILON;
        factors += 1)
                Last term is set equal to term
                Total multiplied and set equal to term divided by 2
                Term is set equal to sqrt_newton((2+term))
        Return 2 divided by total
    Int pi_viete_factors
        Return factors
```

- newton.c: sqrt newton function takes a non-zero/non-negative and approximates it using the Newton-Raphson method and command-line options inputs -n and -a are necessary so the user can see the calculations from these functions. Additionally, the number of terms are calculated. The following is the pseudocode provided in the assignment's document which is what I translated to C for my newton.c file.

```
    def sqrt_newton(x):
    next_y = 1.0 # initial guess for sqrt(x)
    y = 0.0 # force while condition to be true initially
    while abs(next_y - y) > epsilon:
        y = next_y
        next_y = 0.5 * (y + x / y)
    return next_y
```

- wallis.c: approximates the value of Pi with the use of Wallis's formula. The command-line options needed are -w or -a. The number of terms/factors is also returned.

```
        For Loop where the limit is set before 1-term is greater than EPSILON
            The numerator is calculated which is 4 times k squared
            The denominator is the numerator minus 1
            The term is the numerator divided by denominator
            total is multiplied with the term
            Iterate
        return 2 times the total
        Factors function
            returns iterations(k)
```

Results

My code successfully achieves most of what the assignment requires. One of the aspects which are lacking is it's a couple of points off from the example output from the assignment's pdf. Additionally, I did not end up figuring out how the switch case -s can act as a command line that prints statistics when it is called alongside

another command-line option, but also prints the help message when the user only puts -s. The main thing that can be improved is redoing calculations involving the terms/factors iterated so that it outputs the exact amount provided in the assignment's manual.

Numeric results

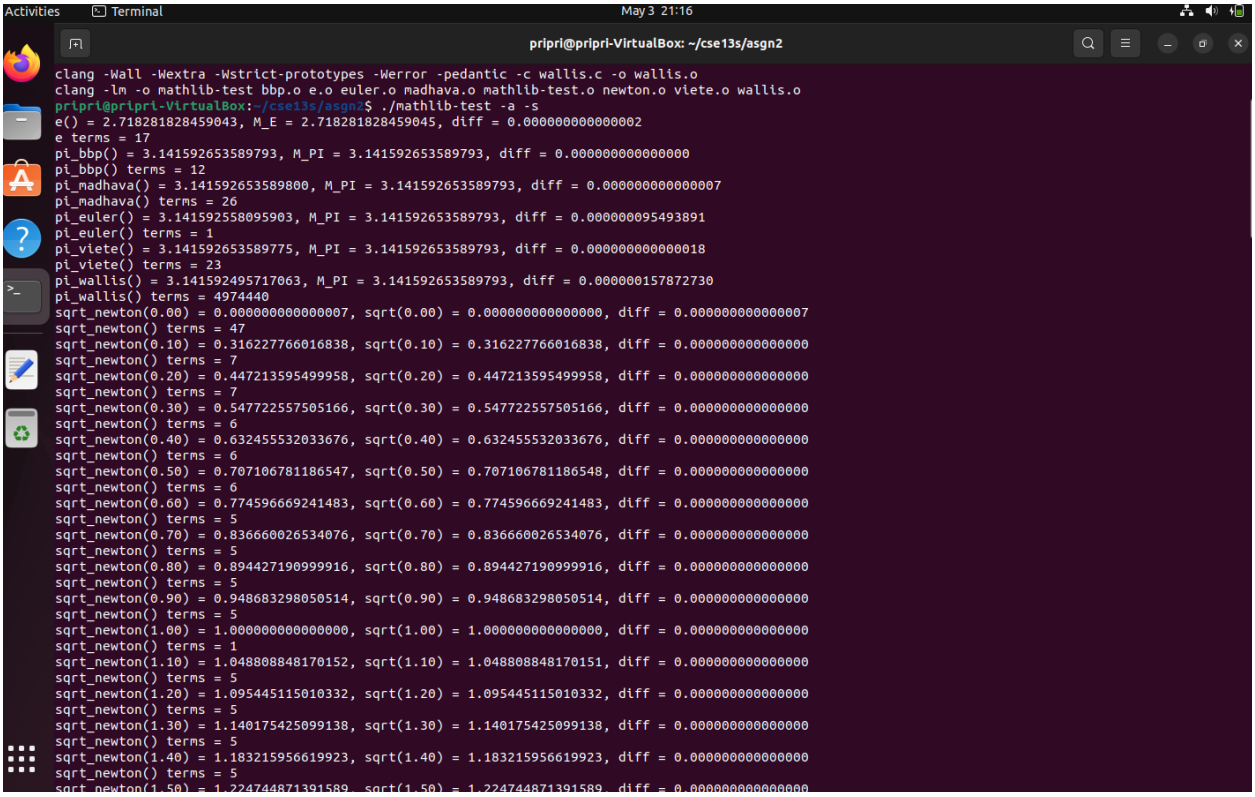

Figure 1: Screenshot of my program running: the output of -a being inputted.

Credit + Notes

I just wanted to mention that this is the second time I am taking CSE13S, therefore most of this code is from last quarter, and although I did tweak around some stuff and added the Wallis series requirements I wanted to ensure that it's expressed this is my own code written from last quarter.

Additionally, I attended Ben's office hours on Wednesday, May 3, 2023, and he was a tremendous help with guiding me in figuring out why my terms were unnecessarily large and why my Wallis series was infinite looping.

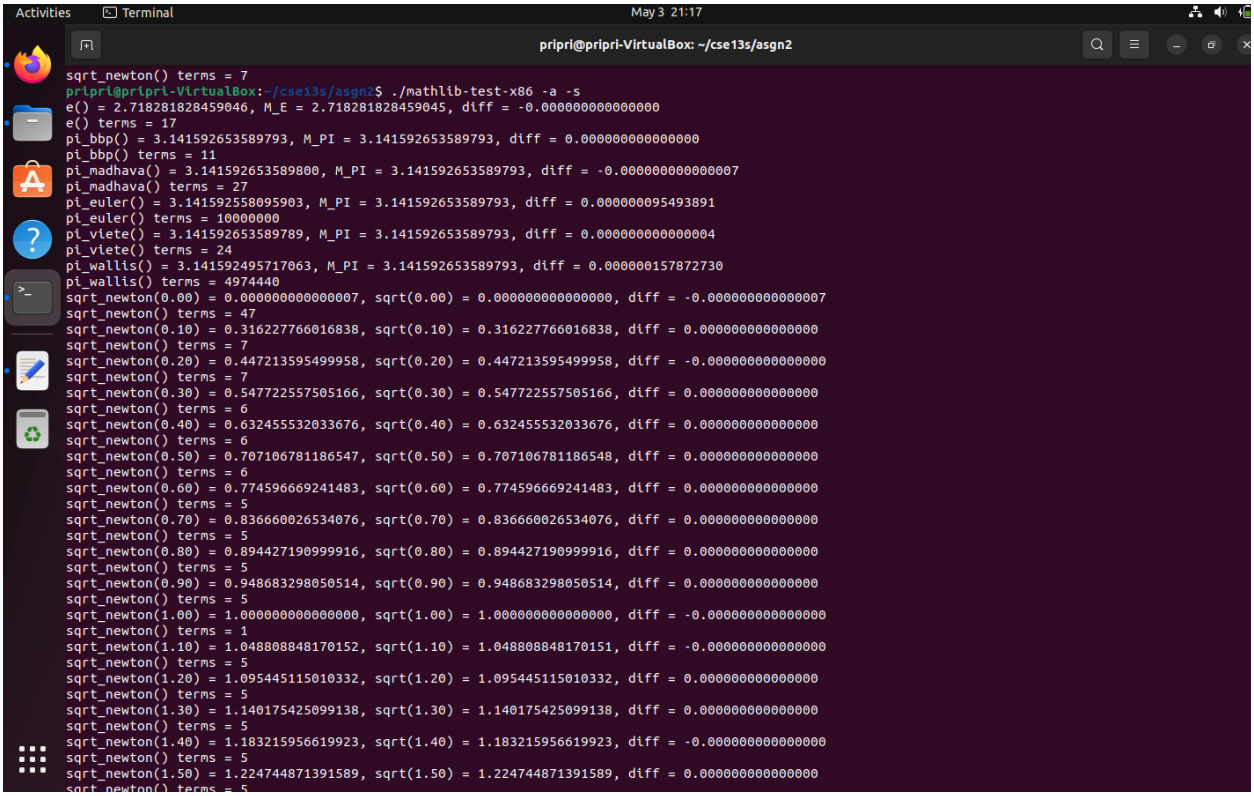

Figure 2: Screenshot of test arm running: the output of -a being inputted.