Assignment 2 – Slice of Pi

Mateo Garcia

CSE 13S – Spring 2023

Purpose

The purpose of this program is to implement a number of mathematical functions in order to compute e and π. The program will also compare them to the C language's math library to ensure they are completely functional.

How to Use the Program

By running mathlib-test.c, you can add 10 different options to the command to tell it which mathematical function you want to run. These are the several different flags you can run:

- -a : Runs all tests.
- -e : Runs e approximation test.
- -b : Runs Bailey-Borwein-Plouffe approximation test.
- -m : Runs Madhava approximation test.
- -r : Runs Euler sequence approximation test.
- -v : Runs Vi'ete approximation test.
- -w : Runs Wallis approximation test.
- -n : Runs Newton-Raphson square root approximation tests, calling sqrt_newton() with various inputs for testing. This option does not require any parameters, and will only test within the range of [0, 10) (it will not test the value 10).
- -s : Enable printing of statistics to see computed terms and factors for all tested functions.
- -h : Displays a help message.

Program Design

There are 8 different C files: **e.c**, **madhava.c**, **euler.c**, **bbp.c**, **viete.c**, **wallis.c**, **newton.c**, and **mathlib-test.c**. The first 6 calculate an approximation of either π or e, and **newton.c** approximates square roots, and the **mathlib-test.c** will test all of these files.

Data Structures

A double will be used to store Epsilon. It is used in while loops to know when to end the loop as the approximation gets closer and closer to convergence. In the **mathlib-test.c** file, the **getopt()** command is used to grab the flags of the command line. Other data structures that will be used are either ints or doubles to store variables in the functions, as well as iterations in the while loops.

Algorithms

Most of the algorithms will be using a while loop to converge on the answer and will break when the difference between the last value and current value is less than a pre-determined value (epsilon). Inside the while loop, a mathematical operation will be performed according to which one is chosen in the command line.

```
int approximate_function(x){
    next_val = 1.0
    val = 0.0
    while abs(next_value - val) > epsilon{
        val = next_val
        next_val = // math operation to get next value in function
    }
    return val
}
```

The **mathlib-test.c** file will use a while loop to grab all the flags from the command line, and if-else statements to print out the functions corresponding to the flags inputted.

```
#define OPTIONS "abcd..n"
int main(int argc, char **argv){
    opt = 0
    flag_1 = 0
    flag_2 = 0
    ...
    flag_n = 0

    while (getopt(argc, argv, OPTIONS) != -1){
        set flags inputted to 1 (set them to True)
    }

    if flag1{
        print("flag on")
    }

    //... repeat if's for all flags

    return 0;
}
```

Function Descriptions

There are several files part of this program and each will have more than 1.

- The **e.c** file will have 2 functions, **e()** and **e_terms()**. The former function will approximate the value of e using Taylor series and track the number of computed terms with a static variable, and the other will return the number of computed terms.
- The **madhava.c** file will contain **pi_madhava()** and **pi_madhava_terms**, the former function will approximate the value of π using the Madhava series, and track the number of computed terms with a static variable. The latter will return the number of computed terms.
- The **euler.c** file will contain **pi_euler()** and **pi_euler_terms()**. The former function will approximate the value of π using the formula derived from Euler's solution to the Basel problem, and track the number of computed terms using a static variable. The latter will return the number of computed terms.

- The **bbp.c** file will contain **pi_bbp()** and **pi_bbp_terms()**. The former function will approximate the value of π using the Bailey-Borwein-Plouffe formula, and track the number of computed terms using a static variable. The latter will return the number of computed terms.
- The **viete.c** file will contain **pi_viete()** and **pi_viete_factors()**. The former function will approximate the value of π using Viete's formula, and track the number of computed terms using a static variable. The latter will return the number of computed terms.
- The **wallis.c** file will contain **pi_wallis()** and **pi_wallis_factors()**. The former will approximate the value of π using Wallis's formula, and track the number of computed terms using a static variable. The latter will return the number of computed terms.
- The **newton.c** file will contain **sqrt_newton()** and **sqrt_newton_iters()**. The former function will take a parameter that will be greater than zero, and will be expected to produce an approximation of the square root using the Newton-Raphson method. This parameter can be outside the range of values that are tested by **mathlib-test.c**. It will also track the number of iterations using a static variable. The latter will return the number of computed terms.
- **mathlib-test.c** will contain the main test for this implemented math library. The expected output for each of the e or π tests should resemble the following.

```
$ ./mathlib-test -e -b -v
e() = 2.718281828459046, M_E = 2.718281828459045, diff = 0.000000000000000
pi_bbp() = 3.141592653589793, M_PI = 3.141592653589793, diff = 0.000000000000000
pi_viete() = 3.141592653589775, M_PI = 3.141592653589793, diff = 0.000000000000018
```

Results

My code successfully achieves the required criteria for Assignment 2. If anything could be improved I would believe it to be in the **mathlib-test.c** file. Just making less if statements and being more efficient overall would be good practice.

Error Handling

A lot of my errors came from syntax mistakes, and errors ocurring in the pipeline with mismatching formats and print statements. I took my time closely looking through the whole **mathlib-test.c** file to look for the disparities. Another error I had was the wrong name for functions in my c files, I accidentally used something like **pi_wallis_terms()** instead of **pi_wallis_factors()** and it caused errors in the pipeline.

Numeric results

Screenshots on next page

```
e() = 2.718281828459046, M_E = 2.718281828459045, diff = -0.000000000000000
e() terms = 18
pi_bbp() = 3.141592653589793, M_PI = 3.141592653589793, diff = 0.000000000000000
pi_bbp() terms = 11
pi_madhava() = 3.141592653589800, M_PI = 3.141592653589793, diff = -0.000000000000007
pi_madhava() terms = 27
pi_euler() = 3.141592558095903, M_PI = 3.141592653589793, diff = 0.000000095493891
pi_euler() terms = 10000000
pi_viete() = 3.141592653589789, M_PI = 3.141592653589793, diff = 0.000000000000004
pi_viete() terms = 24
pi_wallis() = 3.141592495717063, M_PI = 3.141592653589793, diff = 0.000000157872730
pi_wallis() terms = 4974440
sqrt_newton(0.00) = 0.000000000000007, sqrt(0.00) = 0.000000000000000, diff = -0.000000000000007
sqrt_newton() terms = 47
sqrt_newton(0.10) = 0.316227766016838, sqrt(0.10) = 0.316227766016838, diff = 0.000000000000000
sqrt_newton() terms = 7
sqrt_newton(0.20) = 0.447213595499958, sqrt(0.20) = 0.447213595499958, diff = -0.000000000000000
sqrt_newton() terms = 7
sqrt_newton(0.30) = 0.547722557505166, sqrt(0.30) = 0.547722557505166, diff = 0.000000000000000
sqrt_newton() terms = 6
sqrt_newton(0.40) = 0.632455532033676, sqrt(0.40) = 0.632455532033676, diff = 0.000000000000000
sqrt_newton() terms = 6
sqrt_newton(0.50) = 0.707106781186547, sqrt(0.50) = 0.707106781186548, diff = 0.000000000000000
sqrt_newton() terms = 6
sqrt_newton(0.60) = 0.774596669241483, sqrt(0.60) = 0.774596669241483, diff = 0.000000000000000
sqrt_newton() terms = 5
sqrt_newton(0.70) = 0.836660026534076, sqrt(0.70) = 0.836660026534076, diff = 0.000000000000000
sqrt_newton() terms = 5
sqrt_newton(0.80) = 0.894427190999916, sqrt(0.80) = 0.894427190999916, diff = 0.000000000000000
sqrt_newton() terms = 5
sqrt_newton(0.90) = 0.948683298050514, sqrt(0.90) = 0.948683298050514, diff = 0.000000000000000
sqrt_newton() terms = 5
sqrt_newton(1.00) = 1.000000000000000, sqrt(1.00) = 1.000000000000000, diff = -0.000000000000000
sqrt_newton() terms = 1
sqrt_newton(1.10) = 1.048808848170152, sqrt(1.10) = 1.048808848170151, diff = -0.000000000000000
sqrt_newton() terms = 5
```

Figure 1: Screenshot of the program running.

```
mgarc318@mgarc318:~/cse13s/asgn2$ ./mathlib-test -e -b -r -s
e() = 2.718281828459046, M_E = 2.718281828459045, diff = -0.000000000000000
e() terms = 18
pi_bbp() = 3.141592653589793, M_PI = 3.141592653589793, diff = 0.000000000000000
pi_bbp() terms = 11
pi_euler() = 3.141592558095903, M_PI = 3.141592653589793, diff = 0.000000095493891
pi_euler() terms = 10000000
```

Figure 2: Screenshot of the program running.

```
mgarc318@mgarc318:~/cse13s/asgn2$ ./mathlib-test -h
By running mathlib-test.c, you can add 10 different options to the command to tell it which mathematical function you want to run.
These are the several different flags you can run. Make sure to run it in the format of ./mathlib-test -<flag> -<flag2> ..:
-a : Runs all tests.
-e : Runs e approximation test.
-b : Runs Bailey-Borwein-Plouffe approximation test.
-m : Runs Madhava approximation test.
-r : Runs Euler sequence approximation test.
-v : Runs Viete approximation test.
-w : Runs Wallis approximation test.
-n : Runs Newton-Raphson square root approximation tests, calling sqrt_newton() with various inputs for testing. This option does not require any parameters,
and will only test within the range of [0, 10) (it will not test the value 10).
-s : Enable printing of statistics to see computed terms and factors for all tested functions.
-h : Displays this help message.
mgarc318@mgarc318:~/cse13s/asgn2$
```

Figure 3: Screenshot of the program running.