Pi and e Approximations

Mignot Mesele

Purpose

The purpose of this program is to create our own lbirary dedicated to enabling users to be able to perform math beyond just adding, subtracting, multiplying, and dividing since that is what we are limited to. The program has c files dedicated to providing functions for different math formulas such as The bailey borwein plouffe formula. With these, they are then tested in mathlib-test.c by comparing the c files with the formulas in math.h header file, depending on what command option is used, which prints their results and how much of a difference there is between the functions. If -s option is in the command argument, then it will show how much terms are computed.

How to Use the Program

To use the program you'd simply type in the command argument, ./mathlib-test along with a command option. This would specify what formulas you want to compare.

Options:

- -a: run all tests
- -e: run Euler's number approximation test
- -b: run Bailey-Borwein-Plouffe pi approximation test
- -m: run Madhava pi approximation test
- -r: run Euler sequence pi approximation test
- -v: run Viete pi approximation test
- -w: run Wallis pi approximation test
- -n: run Newton-Raphson square root approximation test
- -s enable printing statistics of term/factor iteration in all functions called
- -h: displays help message

You need to use the Makefile, which enables you to run the commands, make filename and make clean. The make command converts the c files into executable files to be able to use mathlib-test.c. The make clean command removes all files created by make, clearing any issues that the object and executable files could cause.

Program Design

Data Structures

array- this will be used to store the elements of the command argument, so it would be able to respond to what the user wants to happen. For example, if they type ./mathlib-test -a, then this would store ./mathlib-test and -a so the program can check for command options like -a, which runs all tests of ./mathlib-test.

Algorithms

```
square root algorithm
    initializes guess for square root of given value;
    initializes the next guess for the condition of loop;
        infinte loop if absolute value of next guess minus current guess
        is greater than epsilon{
            variable of guess is assigned the value of next guess
            variable of next guess takes the result of given value divided by guess,
            added with guess, and divided overall by 2
            }
        returns the next guess
```

```
euler's number algorithm
    initialize k at 0
    infinite loop until the absolute value of a term is less than epsilon{
            term is assigned the value of 1 divided by the factorial of k
            sum of terms is added by term value
            k is incremented by 1
            }
    return sum of terms
```

```
The bailey Borwein Plouffe algorithm
    initialize k at 0
    infinite loop until the absolute value of a term is less than epsilon{
            term is assigned the value of (1 divided by 16 to the power of k)
            times ((4 divided by (8 times k plus 1)) minus (2 divided by (8
            times k plus 4 )) minus (1 divided by (8 times k plus 5)) minus ( 1
            divided by (8 time k plus 6)))
            sum of terms is added by term value
            k is incremented by 1
            }
    return sum of terms
```

```
The Madava series algorithm
    initialize k at 0
    infinite loop until the absolute value of a term is less than epsilon{
            term is assigned the value of (-3 to the power of -k using the
            exponentiation function) divided by (2 times k plus 1)
            sum of terms is added by term value
            k is incremented by 1
            }
    return sum of terms time sthe square root of 12
```

```
Euler's solution algorithm
    initialize k at 1
    infinite loop until the absolute value of a term is less than epsilon{
            term is assigned the value of 1 divided by k to the power of 2
            sum of terms is added by term value
            k is incremented by 1
            }
    return square root of (sum of terms times 6)
```

```
Viete formula algorithm
    initialize k at 1
    initialize a variable as 0
    infinite loop until the absolute value of a term is less than epsilon{
            term is assigned the value of (a with subscript k) divided by 2
            product of terms is multiplied by term value
            k is incremented by 1
            }
    return 2 divided by product of terms
```

```
The Wallis series algorithm
    initialize k at 1
    product of term is initialized with 1
    infinite loop until the absolute value of a term is less than epsilon{
            term is assigned the value of (4 times k to the power of 2)
            divided by ((4 times k to the power of 2) minus 1)
            product of terms is multiplied by term value
            k is incremented by 1
            }
    return product of terms
```

Function Descriptions

- int main(int argc, char **argv)- the main function is the main implementation of the program. This is where the tests are being called, depending on what the user puts in the command argument. The inputs are argc, which is the number of elements in the command argument, and argv, which is an array that stores each element in the command argument. Then it returns zero and prints the output of a function whether it prints an approximation of e or pi.
- double e(void)-This function's purpose is to give the approximation of euler's number. The function has no inputs and returns the approximation of e, using the formula for e, as its output.
- int e terms(void) -This function's purpose is to give the amount of terms that have been iterated in the summation for e(). The function has no inputs and returns the total amount of terms iterated as the output.
- double sqrt newton(double base)-This function's purpose is to give the square root of of a user-assigned number. The function has no inputs and returns the approximation of pi, using newton raphson's method of calculating square roots, as its output.
- int sqrt newton iters(void)-This function's purpose is give the amount of iterations that the sqrt newton() goes through to get the square root. The function has no inputs and returns the total amount of terms iterated as the output.
- double pi bbp(void)-This function's purpose is to give the bbp(bailey borwein plouffe) approximation of pi. The function has no inputs and returns the approximation of pi, using bbp's formula of pi, as its output.
- int pi bbp(void)-This function's purpose is to give the amount of terms iterated in pi bbp(void). The function has no inputs and returns the total amount of terms iterated as the output.
- double pi madhava(void)-This function's purpose is to give the madhava approximation of pi. The function has no inputs and returns the approximation of pi, using the madhava series formula of pi, as its output.

- int pi madhava terms(void)-This function's purpose is to give the amount of terms iterated in pi madhava(). The function has no inputs and returns the total amount of terms iterated as the output.
- double pi euler(void)-This function's purpose is to give the euler approximation of pi. The function has no inputs and returns the approximation of pi, using the euler solution method, as its output.
- int pi euler terms(void)-This function's purpose is to give the amount of terms iterated in pi euler(). The function has no inputs and returns the total amount of terms iterated as the output.
- double pi viete(void)-This function's purpose is to give the viete approximation of pi. The function has no inputs and returns the approximation of pi, using the viete formula of pi, as its output.
- int pi viete factors(void)-This function's purpose is to give the amount of factors iterated in pi viete(). The function has no inputs and returns the total amount of factors iterated as the output.
- double pi wallis(void)-This function's purpose is to give the wallis approximation of pi. The function has no inputs and returns the approximation of pi, using the wallis formula of pi, as its output.
- int pi wallis factors(void)-This function's purpose is to give the amount of factors iterated in pi wallis(). The function has no inputs and returns the total amount of factors iterated as the output.

Results

After implementing my code, it ran as intended. It would respond to the user's command argument with the respective approximations of pi/e, from both the program and math.h, and their difference. It also printed the amount of terms/factors iterated when inputting -s, printed the help message when the user either inputs, -h, -s by itself, or no options. Even though the program works, I think that it can be improved. For example, making functions for repeated code that takes a lot of lines to write or use different, shorter, and more efficient ways of implementing a certain part of the algorithm. Besides that, my codes approximations were the same as the binary file so I think that my approximations are fine.

Numeric results

Under the Error Handling section is a picture of this program running ./mathlib-test -a.

Error Handling

The program alone handles invalid option errors through getopt but I had to create a condition, in which the user inputs no options, by checking if none of the options were checked for in the switch statement I created, for checking what options were inputted, then it would print the help message. Other than that, most of my errors were in e.c with my approximations being short due to overflow of k!. Other than that, I was able to learn from my mistakes, and apply it to the other functions, making it easier to finish with the correct approximations. Also, for terms I forgot to add a static variable reset statement, i.e. term count = 0, causing the terms to add after rerunning the program, and I forgot the word pi before madhava terms() in a print statement.

```
1,17c1
< SYNOPSIS
<    A test harness for the small numerical library.
<
< USAGE
<    ./mathlib-test-x86 -[aebmrvnsh]
<
< OPTIONS
<   -a   Runs all tests.
<   -e   Runs e test.
<   -b   Runs BBP pi test.
<   -m   Runs Madhava pi test.
<   -r   Runs Euler pi test.
<   -v   Runs Viete pi test.
<   -w   Runs Wallis pi test.
<   -n   Runs Newton square root tests.
<   -s   Print verbose statistics.
<   -h   Display program synopsis and usage.
---
> Your input shoud be in the format, ./{filename} {options i.e. -a -r -n}. To run all tests, you'd use the -a option. To run run e approximation test, use -e option. To run B
ailey-Borwein-Plouffe pi approximation test, use -b option. To run Madhava pi approximation test, use -m option. To run Euler sequence pi approximation test, use -r option. T
o run Viete pi approximation test, use -v option. To run Wallis pi approximation test, run -w option. To run Newton-Raphson square root approximation test, run -n. To show al
l of the test's amount of terms, use -s. These options can only run once, meaning that if you put ./mathlib-test -a -e -v, ./mathlib-test -b -m -a, or ./mathlib-test -e -e, i
t would basically ignore the second request to run the test.
```

```
e() = 2.718281828459046, M_E = 2.718281828459045, diff = -0.000000000000000
pi_bbp() = 3.141592653589793, M_PI = 3.141592653589793, diff = 0.000000000000000
madhava() = 3.141592653589800, M_PI = 3.141592653589793, diff = -0.000000000000007
pi_euler() = 3.141592558095903, M_PI = 3.141592653589793, diff = 0.000000095493891
pi_viete() = 3.141592653589789, M_PI = 3.141592653589793, diff = 0.000000000000004
pi_wallis() = 3.141592495717063, M_PI = 3.141592653589793, diff = 0.000000157872730
sqrt_newton(0.00) = 0.000000000000007, sqrt(0.00) = 0.000000000000000, diff = -0.000000000000007
sqrt_newton(0.10) = 0.316227766016838, sqrt(0.10) = 0.316227766016838, diff = 0.000000000000000
sqrt_newton(0.20) = 0.447213595499958, sqrt(0.20) = 0.447213595499958, diff = -0.000000000000000
sqrt_newton(0.30) = 0.547722557505166, sqrt(0.30) = 0.547722557505166, diff = 0.000000000000000
sqrt_newton(0.40) = 0.632455532033676, sqrt(0.40) = 0.632455532033676, diff = 0.000000000000000
sqrt_newton(0.50) = 0.707106781186547, sqrt(0.50) = 0.707106781186548, diff = 0.000000000000000
sqrt_newton(0.60) = 0.774596669241483, sqrt(0.60) = 0.774596669241483, diff = 0.000000000000000
sqrt_newton(0.70) = 0.836660026534076, sqrt(0.70) = 0.836660026534076, diff = 0.000000000000000
sqrt_newton(0.80) = 0.894427190999916, sqrt(0.80) = 0.894427190999916, diff = 0.000000000000000
sqrt_newton(0.90) = 0.948683298050514, sqrt(0.90) = 0.948683298050514, diff = 0.000000000000000
sqrt_newton(1.00) = 1.000000000000000, sqrt(1.00) = 1.000000000000000, diff = -0.000000000000000
sqrt_newton(1.10) = 1.048808848170152, sqrt(1.10) = 1.048808848170151, diff = -0.000000000000000
sqrt_newton(1.20) = 1.095445115010332, sqrt(1.20) = 1.095445115010332, diff = 0.000000000000000
sqrt_newton(1.30) = 1.140175425099138, sqrt(1.30) = 1.140175425099138, diff = 0.000000000000000
sqrt_newton(1.40) = 1.183215956619923, sqrt(1.40) = 1.183215956619923, diff = -0.000000000000000
sqrt_newton(1.50) = 1.224744871391589, sqrt(1.50) = 1.224744871391589, diff = 0.000000000000000
sqrt_newton(1.60) = 1.264911064067352, sqrt(1.60) = 1.264911064067352, diff = -0.000000000000000
sqrt_newton(1.70) = 1.303840481040530, sqrt(1.70) = 1.303840481040530, diff = 0.000000000000000
sqrt_newton(1.80) = 1.341640786499874, sqrt(1.80) = 1.341640786499874, diff = 0.000000000000000
sqrt_newton(1.90) = 1.378404875209022, sqrt(1.90) = 1.378404875209022, diff = 0.000000000000000
sqrt_newton(2.00) = 1.414213562373095, sqrt(2.00) = 1.414213562373095, diff = -0.000000000000000
sqrt_newton(2.10) = 1.449137674618944, sqrt(2.10) = 1.449137674618944, diff = -0.000000000000000
sqrt_newton(2.20) = 1.483239697419133, sqrt(2.20) = 1.483239697419133, diff = 0.000000000000000
sqrt_newton(2.30) = 1.516575088810310, sqrt(2.30) = 1.516575088810310, diff = 0.000000000000000
sqrt_newton(2.40) = 1.549193338482967, sqrt(2.40) = 1.549193338482967, diff = 0.000000000000000
sqrt_newton(2.50) = 1.581138830084190, sqrt(2.50) = 1.581138830084190, diff = 0.000000000000000
sqrt_newton(2.60) = 1.612451549659710, sqrt(2.60) = 1.612451549659710, diff = 0.000000000000000
sqrt_newton(2.70) = 1.643167672515499, sqrt(2.70) = 1.643167672515499, diff = 0.000000000000000
sqrt_newton(2.80) = 1.673320053068152, sqrt(2.80) = 1.673320053068151, diff = -0.000000000000000
```

Here is an image of using diff between the outputs of both binaries showing the only difference being the help messages.