Assignment 2 DESIGN.pdf

Description of Program:

This program approximates the values of π and *e* using different mathematical series' and informs the user on two things: the number of iterations required to produce that result, and the difference between the approximated calculations and their true values. By displaying these values, can the user compare the efficiency of choosing one approximation formula versus the others.

Files to be included in directory "asgn2":

1. Makefile:
   - Supports the compilation of the all .c programs in the "asgn2" directory
2. mathlib.h:
   - Header file containing the global constant "EPSILON" and the absolute() function, which are both used in other programs. This file also links the .c files together so that functions can be called in other .c files, like mathlib-test.c for example.
3. mathlib-test:
   - Executable file that instructs the computer to run mathlib-test.c
4. mathlib-test.c:
   - Source file containing main(). Allows the user to test the following approximation formulas using command-line options.
5. e.c:
   - Approximates the value of *e* using a Taylor series and counts the number of computed terms.
6. bbp.c:
   - Approximates the value of *π* using the Bailey-Borwein-Plouffe formula and counts the number of computed terms.
7. madhava.c:
   - Approximates the value of *π* using the Madhava series and counts the number of computed terms.
8. euler.c:
   - Approximates the value of *π* using a version of Euler's solution and counts the number of computed terms.
9. viete.c:

- Approximates the value of π using an implementation of Viète's formula and counts the number of computed factors.

10. newton.c:
    - Approximates the square root of the number passed to it using Newton's method and counts the number of iterations.
11. README.md:
    - Proper explanation of how to run the program and *Makefile*. It also contains a list of available command-line options and explains their functions.
12. DESIGN.pdf:
    - Explains the thought process behind the program with sufficient detail. Must be in PDF format.
13. WRITEUP.pdf:
    - PDF file created using LaTeX that includes graphs displaying the differences between approximate and actual values and an analysis as to what they mean.

Pseudocode / Structure:

mathlib-test.c:

```
Create getopt variable and set it to 0
While retrieving arguments from command line:
    If user types -a:
        Retrieve all $e$ and $\pi$ approximation values, calculate their differences from actual values of $e$ and $\pi$, and print these results.
    If user types -e:
        Retrieve $e$ approximation value using function in e.c, calculates the difference from actual value of $e$, and prints these results.
    If user types -b:
        Retrieve $\pi$ approximation value using function in bbp.c, calculates the difference from actual value of $\pi$, and prints these results.
    If user types -m:
        Retrieve $\pi$ approximation value using function in madhava.c, calculates the difference from actual value of $\pi$, and prints these results.
    If user types -r:
        Retrieve $\pi$ approximation value using function in euler.c, calculates the difference from actual value of $\pi$, and prints these results.
    If user types -v:
        Retrieve $\pi$ approximation value using function in viete.c, calculates the difference from actual value of $\pi$, and prints these results.
    If user types -n:
        Retrieve $\pi$ approximation value using function in newton.c, calculates the
        difference from actual value of $\pi$, and prints these results.
    If user types -s:
        Set s toggle switch to 1 (on)
    If user types -h:
        Prints a help message explaining how to use the program
```

e.c:

```
Create a static variable count and initialize it to 0

Create a function e() that takes no arguments and returns a double:
    Assign initial sum to 0 and previous factor to 1

    While precision of current iteration > epsilon (1x10^-14):
        If it's the first loop:
            Add 1 to sum
        Else:
            Calculate $k^{th}$ iteration of the euler's value approximation Taylor series and
            add it to sum
        Add 1 to count

    Return calculated sum (e approximation)

Create a function e_terms() that takes no arguments and returns an int:
    Return number of computed terms
```

bbp.c:

```
Create a static variable count and initialize it to 0

Create a function pi_bbp() that takes no arguments and returns a double:
    Assign initial sum to 0 and previous factor to 1

    While precision of current iteration > epsilon:
        Calculate $k^{th}$ iteration of the pi approximation Bailey-Borwein-Plouffe formula
        and add it to sum

        Multiply previous factor by k
        Increase count by 1
    Return calculated sum (pi approximation)

Create a function pi_bbp_terms() that takes no arguments and returns an int:
    Return number of computed terms
```

madhava.c:

```
Create a static variable count and initialize it to 0

Create a function pi_madhava() that takes no arguments and returns a double:
        Assign initial sum to 0 and factor to 1

        While absolute value of the precision of current iteration > epsilon:
                Calculate $k^{th}$ iteration of the pi approximation Madhava series and add it to sum

                Multiply factor by -3
                Increase count by 1

        Return calculated sum (pi approximation)

Create a function pi_madhava_terms() that takes no arguments and returns an int:
        Return number of computed terms
```

euler.c:

```
Create a static variable count and initialize it to 0

Create a function pi_euler() that takes no arguments and returns a double:
        Assign initial sum to 0

        While the precision of current iteration > epsilon:
                Calculate $k^{th}$ iteration of the given version of Euler’s solution series and add it to
                sum

                Increase count by 1

        Return square root of calculated sum (pi approximation)

Create a function pi_euler_terms() that takes no arguments and returns an int:
    Return number of computed terms
```

viete.c:

```
Create a static variable *count* and initialize it to 0

Create a function *pi_viete()* that takes no arguments and returns a double:
    Assign initial product to 1, previous term to 0, and current term to 0

    While absolute value of the precision of current iteration > *epsilon*:
        Calculate k$^{th}$ iteration of the pi approximation Viète’s formula and set it to current term
        Set previous term equal to the current term
        Multiply the product by the current term divided by 2

        Increase *count* by 1

    Return 2 divided by the calculated product (pi approximation)

Create a function *pi_viete_factors()* that takes no arguments and returns an int:
    Return number of computed terms
```

newton.c:

```
Create a static variable *count* and initialize it to 0

Create a function *sqrt_newton()* that takes a double as an argument and returns a double:
    Set two doubles, *z and y*, equal to 0 and 1 respectively

    While the absolute value of the difference between *y and z* is > *epsilon*:
        Assign *y* to *z*
        Divide *x* by *z*, add it to *z*, and assigned half of that result to *y*

        Increase *count* by 1

    Return *y* (calculated square root)
```

Create a function *sqrt_newton_iters()* that takes no arguments and returns an int:

Return number of computed terms

Credit:

- The method and approach used to approximate the square root of a given value was provided by Professor Long. In the final program, this is found solely in the sqrt_newton() function.