Assignment 2 - A Slice of Pi

Teresa Joseph

CSE 13S - Professor Long

Fall 2021 - RD October 7/ FD October 10

Purpose

Assignment 2 is the implementation of several mathematical functions, namely ones that serve to approximate the values e and pi and to calculate the square root of a given float. These functions will be implemented in files called e.c, madhava.c, euler.c, bbp.c, viete.c, and newton.c. Another file called mathlib-test.c will also be made to test these files with command line arguments.

Breakdown of Functions

e.c

- Contains e() and e_terms()
  - e() approximates the value of e using taylor series and track the number of computed terms through a static variable local to the file
  - e_term() returns number of computed terms

madhava.c

- Contains pi_madhava() and pi_madhava_terms()
  - pi_madhava() approximate value of pi using Madhava series and track the number of computed terms with a static variable (like e.c)
  - pi_madhava_terms() returns number of computed terms

euler.c

- Contains pi_euler() and pi_euler_terms()
  - pi_euler() approximate the value of pi using formula from Euler's solution to Basel problem
  - pi_euler_terms() returns number of computed terms

bbp.c

- Contains pi_bbp() and pi_bbp_terms()
  - pi_bbp() approximates value of pi using Bailey-Borwein-Plouffe formula and track the number of computed terms
  - pi_bbp_terms() returns number of computed terms

viete.c

- Contains pi_viete() and pi_viete_factors()
    - pi_viete() approximates the value of pi using Viete's formula and track the number of computed terms
    - pi_viete_factors() returns number of computed terms

newton.c

- Contains sqrt_newton() and sqrt_newton_iters()
    - sqrt_newton() approximates the square root of the argument passed to it using the Newton-Raphson method and and track the number of iterations taken
    - sqrt_newton_iters() returns the number of iterations taken

mathlib-test.c

- Contain the main test harness for implemented math library
- Should support -a (run all tests), -e (run e approximation test), -b (run bbp pi approximation test), -m (run Madhava pi approximation test), -r (run Euler pi approximation test), -v (run Viete pi approximation test), -n (run Newton square root approximation tests), -s (enable printing statistics to see computed terms and factors for each tested function), and -h (display a help message detailing program usage)

Pseudocode

e.c

```
set static counter
set e() function
    first term = 1 (for summation formula)
    second term = 0 (for summation formula)
    factorial counter = 1 (for tracking summation)
    counter for terms = 1
    while absolute of first-second term > epsilon, do summation
        second term = first term
        factorial counter *= counter
        first term = 1/factorial counter (the summation portion)
        increase counter by 1
    return first term (the approximation of e)
define e_terms()
```

```
    return counter
```

madhava.c

```
set static counter
define pi_madhava()
    first term = 1 (for summation formula)
    second term = 0 (for summation formula)
    fraction counter = 0 (for summation formula)
    numerator counter = 0 (for summation formula)
    denominator counter = 0 (for summation formula)
    while counter is less than 2 or absolute of first-second term > epsilon, do summation
        second term = first term
        if counter is 0, make numerator and denominator both 1 (1/1)
        otherwise:
            numerator *= 3
            denominator = 2*counter + 1
            fraction = 1/numerator (inverts numerator)
            first term += fraction/denominator (actual fraction)
        increase counter by 1
    return (first term * sqrt of 12) (the approximation of pi)
define pi_madhava_terms()
    return counter
```

euler.c

```
set static counter
define pi_euler()
    first term = 0 (for summation formula)
    second term = 0 (for tracking summation)
    while counter is less than 2 or absolute of first-second term > epsilon, do summation
        second = first term
        first term += 1/(counter squared)
        increase counter by 1
    take sqrt of 6*first term and return this value (the approximation of pi)
```

```
define pi_euler_terms()
    return counter-1 (actual number of iterations and thus terms)
```

bbp.c

```
set static counter
define pi_bbp()
    first term = 0 (for summation formula)
    second term = 0 (for tracking summation)
    coefficient = 0
    fractional coefficient = 0
    while counter is less than 2 or absolute of first-second term > epsilon, do summation
        second term = first term
        (initialize) fraction = [denominator of bbp formula, using counter as k]
        if counter = 0, then coefficient = fractional coefficient = 1 (1/1)
        otherwise:
            coefficient *= 16
            fractional coefficient = 1/coefficient
        increase counter by 1
        first term += fractional coefficient * fraction
    return first term (approximation of pi)
define pi_bbp_terms()
    return counter
```

viete.c

```
set static counter
define pi_viete()
    first term = 1 (for summation formula)
    second term = 0 (for summation formula)
    while absolute of first-second term > epsilon, do summation
        second term = first term
        numerator = sqrt of numerator + 2
        first term *= numerator/2
        increase counter by 1
    return 2/first term (approximation of pi)
define pi_viete_factors()
    return counter
```

newton.c

```
set static counter
define sqrt_newton(x)
    first term = 1.0
    second term = 0.0
    while absolute of first-second term > epsilon, do summation
        second term = first term
        first term = 0.5 * (second term + x / second term)
        increase counter by 1
    return first term (the approximation of sqrt)
define sqrt_newton_iters(x)
    return counter
```

mathlib-test.c

```
define all strings [aebmrvnsh]
use bool to set all flags as false
use getopt and switch to make cases for each string
    check command line inputs with (and if string exists, make flag true)
        if -a, run all tests (return all)
        if -e, run e approximation test
        if -b, run bbp pi approximation test
        if -m, run Madhava pi approximation test
        if -r, run Euler pi approximation test,
        if -v, run Viete pi approximation test,
        if -n, run Newton_raphson square root approximation tests,
        if -s, print statistics to see computed terms and factors for each tested function
        if -h, display a help message detailing program usage
in each flag, call my functions, call math.h library values, and print differences
    differences = absolute of one value - the other value
```

Notes

- mathlib.h and stdio.h libraries are included in each file
- While loops are implemented in all files except mathlib-test.c
- All while loop conditions involve the absolute value of (first term-second term)
- Types for the variables will generally be int/long int or double

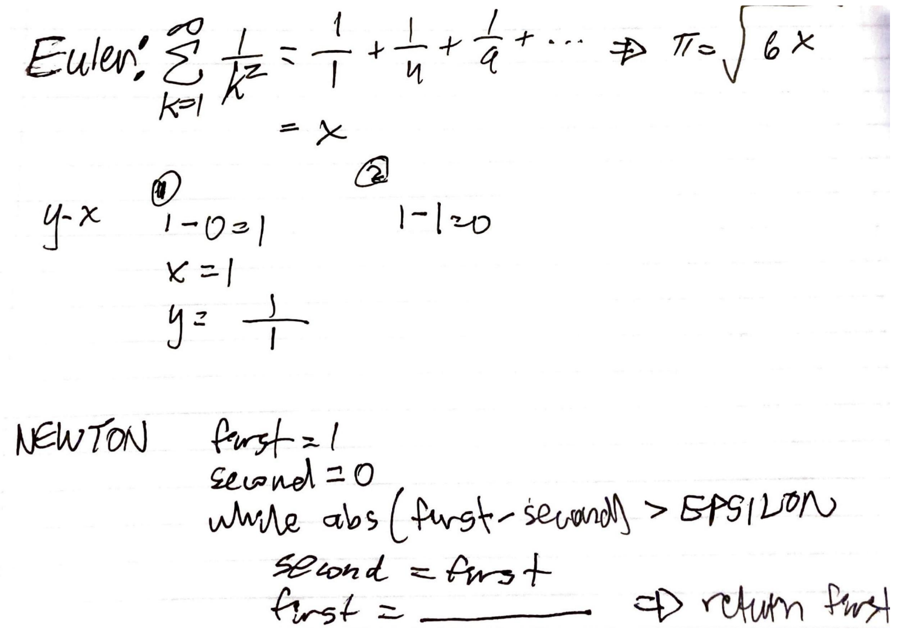
Overall Description

All files except mathlib-test.c will have two functions, one that approximates either e or pi or calculates square root and another that returns the number of terms or iterations needed to reach that approximation. These functions have similar setups: variables will be initialized, while loops will iterate over certain conditions, and returns at the end will provide approximations/the number of iterations. All functions follow a similar pattern, comparing terms to epsilon. The file mathlib-test.c will test my implementations by running them with command line inputs as described in the assignment document.

Goals/Intended Process

- Replicate the described pseudocode for each file in C
- Make code more readable and efficient if possible
- Add sufficient comments and clean up format

Other



Euler: $\sum_{k=1}^{\infty} \frac{1}{k^2} = \frac{1}{1} + \frac{1}{4} + \frac{1}{9} + \cdots \Rightarrow \pi = \sqrt{6x}$

$= x$

y-x ① $1-0=1$ ② $1-1=0$

$x=1$

$y = \frac{1}{1}$

```
NEWTON  first = 1
        second = 0
        while abs(first - second) > EPSILON
            second = first
            first = ________    => return first
```

Helped visualized euler.c and newton.c structure