A Little Slice of π: Design

Lucais Sanderson

29 January 2023

Part I
Initial Design

1 Description of Program

There are 6 source files that contain functions approximating various irrational constants, such as e and π. Specifically, Jacob Bernoulli's Taylor series to approximate e. Then to calculate π I use methods from Madhava, Euler, Bailey-Borwein-Plouffe, and Viete. Additionally, I implement a square root function using the Newton-Raphson method.

To compile, compare, and analyze the results from these functions, I will use another source file that takes command line arguments to determine which tests to run and whether to print the statistics from them. Essentially the interface for all the functions.

2 Pseudocode / Structure:

- e.c

```
function e:
    sum = 0
    count = 1
    previous_term = 1

    loop from k=1 until difference between terms less than epsilon
        current_term = ( x / k ) * previous_term
        add current_term to sum
        set previous_term to current_term
        increment count 1

    return sum

function e_terms:
    return count
```

- madhava.c

  ```
  function pi_madhava:
      sum = 0
      count = 1
      previous_term = 1

      loop from k=1 until difference between terms less than epsilon
          temp = 1
          loop from 0 to k
              temp = temp * 3
          current_term = ( 1 / temp * (2k + 1) ) + previous_term
          previous_term = current_term
          increment count 1

  function pi_madhava_pi:
      return count
  ```

- euler.c

  ```
  function pi_euler:
      sum = 0
      count = 1
      previous_term = 1

      loop for k=2 to absolute(previous_term - current ) < eplison, increment k by 1
          current_term = ( 1 / k*k ) + previous_term
          add current_term to sum
          previous_term = current_term
          count++

      return root(6 * sum)

  function pi_euler_terms:
      return count
  ```

  ‘

- bbp.c

  ```
  function pi_bbp:
      sum = 0
  ```

```
    count = 1
    previous_term

    loop for k=1 to (pi - sum) < epsilon, increment k by 1
        current_term =
            (1/16^k) * ((k(120*k + 151) + 47)
            /
            (k(k(k(512*k + 1024) + 712) + 194) + 15))
            + previous_term
        previous_term = current_term
        increment count by 1

function pi_bbp_terms:
    return count
```

- viete.c

```
function pi_viete:
    count = 1
    previous = sqrt(2) / 2
    # this represents the first term

    loop for k=2, |1 - previous| < epsilon
        temp = sqrt(2)
        loop for i=2 to i>k
            temp = sqrt( 2 + temp )
        current = (temp / 2) * previous
        previous = current
        add 1 to count
    result = 2 / previous # out final estimate

function pi_viete_terms
    return count
```

- newton.c

```
count

sqrt_newton(x):
    z = 0
    y = 0
    while the absolute value of (y- z) > epsilon:
        z = y
        y = 0.5 * (z + x / z)
```

```
        count ++
    return y

sqrt_newton_iters:
    return count
```

- `mathlib-test.c`

```
if ran without arguments or "-h", then show synopsis, usage, and options

"-a" option runs all tests and an option for each indiviudal test
can be ran as well.

"-s" prints verbose statistics.
```

3 Credit

Thus far, I've only used the assignment document to source all my code.

Part II
Final Design

4 Description of Program

There are 6 source files that contain functions approximating various irrational constants, such as e and π. Specifically, Jacob Bernoulli's Taylor series to approximate e. Then to calculate π I use methods from Madhava, Euler, Bailey-Borwein-Plouffe, and Viete. Additionally, I implement a square root function using the Newton-Raphson method.

To compile, compare, and analyze the results from these functions, I will use another source file that takes command line arguments to determine which tests to run and whether to print the statistics from them. Essentially the interface for all the functions.

5 Pseudocode / Structure:

- `e.c`

```
function e:
    sum = 1
    count = 1
    previous_term = 1
```

```
    loop from k=1 until pervious term less than epsilon
        current_term = ( 1 / k ) * previous_term
        add current_term to sum
        set previous_term to current_term
        increment count 1

    return sum

function e_terms:
    return count
```

- madhava.c

```
function pi_madhava:
    sum = 1
    count = 1
    previous_term = 1

    loop from k=1 until difference between terms less than epsilon
        temp = 1
        loop from 0 to k
            temp = temp * 3
        current_term = ( 1 / temp * (2k + 1) ) + previous_term
        previous_term = current_term
        increment count 1

function pi_madhava_pi:
    return count
```

- euler.c

```
function pi_euler:
    sum = 0
    count = 1
    previous_term = 1

    loop for k=2 to absolute(previous_term - current ) < eplison, increment k by 1
        current_term = ( 1 / k*k ) + previous_term
        add current_term to sum
        previous_term = current_term
        count++

    return root(6 * sum)
```

```
function pi_euler_terms:
    return count
```

‘

- bbp.c

```
function pi_bbp:
    sum = 0
    count = 1
    previous_term

    loop for k=1 to (pi - sum) < epsilon, increment k by 1
        current_term =
            (1/16^k) * ((k(120*k + 151) + 47)
            /
            (k(k(k(512*k + 1024) + 712) + 194) + 15))
            + previous_term
        previous_term = current_term
        increment count by 1

function pi_bbp_terms:
    return count
```

- viete.c

```
function pi_viete:
    count = 1
    previous = sqrt(2) / 2
    # this represents the first term

    loop for k=2, |1 - previous| < epsilon
        temp = sqrt(2)
        loop for i=2 to i>k
            temp = sqrt( 2 + temp )
        current = (temp / 2) * previous
        previous = current
        add 1 to count
    result = 2 / previous # out final estimate

function pi_viete_terms
    return count
```

- newton.c

```
count

sqrt_newton(x):
    z = 0
    y = 0
    while the absolute value of (y- z) > epsilon:
        z = y
        y = 0.5 * (z + x / z)
        count ++
    return y

sqrt_newton_iters:
    return count
```

- `mathlib-test.c`

```
get options from argv

flag bool variables for each option so each option
is printed once where necessary

if -h flagged or invalid option, print help screen

if -s flagged with any other option (not h), prints count as well

-a prints all

each individual function may be called respectively
```

6 Credit

Only used the information from the spec document in `resources`.