

# Assignment 2 – Slice of Pi

Yazmyn Sims

CSE 13S – Spring 2023

## Purpose

We will use math formulas to calculate the values of  $e^x$ , square root  $x$  and  $\pi$  without using aids like `#include <math.h>`. Afterwards, we will write a test comparing my functions to those in the math libraries.

## How to Use the Program

All you have to do is go into vim and type `./mathlib-test` followed by the command line/s (Ex: `-e`, `-b`, `-v`, etc) for the code you'd like to test and it will output the results. When the code is run the program will perform the desired mathematical calculation over and over again, getting closer and closer to the exact value. It will return the closest result on screen, the result from the `math.h` library, how many terms were run, and how far off your code was from the code in the math libraries.

This what the output will look like:

```
$ ./mathlib-test -e -b -v -s e() = 2.718281828459046, M_E = 2.718281828459045, diff =  
0.0000000000000000 e terms = 18 pi_bbp() = 3.141592653589793, M_PI = 3.141592653589793, diff =  
0.0000000000000000 pi_bbp() terms = 11 pi_viete() = 3.141592653589775, M_PI =  
3.141592653589793, diff = 0.0000000000000018 pi_viete() terms = 23
```

## Program Design

In the `bbp.c` file we are using Bailey-Borwein-Plouffe's formula to find  $\pi$  and return the number of terms.

In the `e.c` file we care about finding the value of  $e$  using Taylor Series and returning the number of terms.

In the `newton.c` file we're finding the value of square root  $x$  with Newton's method and return the number of terms.

In the `euler.c` file we are using Euler's solution to find  $\pi$  and return the number of terms.

In the `madhava.c` file file we are using the Madhava series to find  $\pi$  and return the number of terms.

In the `viete.c` file we are using the Viete's formula to find  $\pi$  and return the number of factors.

In the `wallis.c` file we are using the Wallis formula to find  $\pi$  and return the number of factors.

The `mathlib-test.c` file provides original `math.h` calculations for us to compare our calculations to the code we've created against the math code already present in `math.h`.

In short, we are writing these formulas for  $\pi$ ,  $e$ , and the square root of  $x$  in the C programming language and outputting the result, the number of terms, and how close our answer was to the

math.h answer. So that the program doesn't run forever, we make sure it stops calculating more values once we've reached EPSILON.

## Data Structures

We're storing values of pi, e, and square root x as well as dif (which compares our result to the math.h result and shows the difference), terms (which shows how many times the program updated the values before reaching epsilon), and factors (which shows how many times the program update the values before reaching epsilon).

You transfer data between different functions by storing the data in a variable or function and placing that variable/function within the function you'd like that data to be transferred to. For example I stored the value of pi within the function pi\_euler() and I was able to transfer that information to the mathlib.c file and into the if statement that would print out the stored result.

### variables used:

- count
- term
- factor
- pi
- exponent
- new\_exponent
- k
- EPSILON
- E
- M\_PI
- M\_E
- diff
- opt
- is\_b
- is\_e
- is\_n
- is\_w
- is\_v
- is\_a
- is\_h
- is\_s
- is\_m
- is\_r
- y
- next\_y
- x
- result

### How are you storing the options passed into the function:

- We stored the command-line options with the getopt() function by pre-defining the possible OPTIONS ahead of time.

## Algorithms

If there was a Sigma, I used a for loop:

Let the starting value be k = 0 or 1 (depending on the algorithm)

Compute some value and add this new value to the previous value

let k increment by 1 each time that value is computed.

Increment the value of count each time to track the number of times the value is computed.

When the term is greater than EPSILON, stop the for loop.

In some cases there were operations that need to be performed after the sigma:

When the for loop is finished, apply operation to the result of the sigma.

Return this new final result.

If there was a product notation symbol I used a for loop:

Let the starting value be  $k = 1$

Compute some value and multiply this new value to the previous value

let  $k$  increment by 1 each time that value is computed.

Increment the value of count each time to track the number of times the value is computed.

When the absolute value of 1-factor is greater than EPSILON, stop the for loop.

In some cases there were operations that need to be performed after the production:

When the for loop is finished, apply operation to the result of the production notion.

Return this new final result.

If I was calculating newtons sqrt root I used a while loop:

While the absolute value of nexty-y is greater than EPSILON

Let  $y$  equal next\_y.

Next\_y then equals .5 times  $y + x/y$

Increment count each time to track the number of terms.

Once the absolute value is greater than EPSILON, end the while loop and return the final next\_y result.

## Function Descriptions

- Input:

No user input perse other than running the code

In code functions have inputs though:

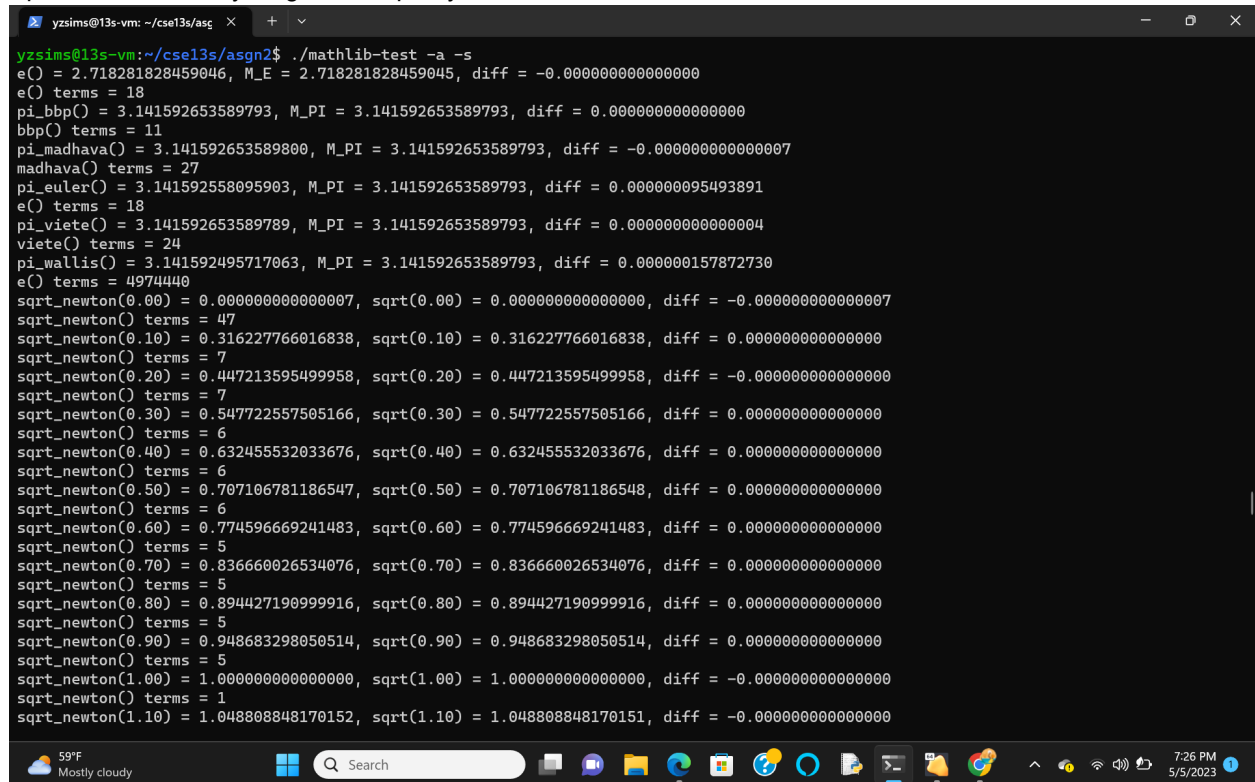
1.  $k$  is input for the for-loop and sometimes for if statements
2. term and factor are inputs for the for-loop that tells it when to stop running
3. OPTIONS is input for the switch case statements
4. is\_h, is\_a, is\_e, etc are inputs for the if statements

- The outputs of every function

1. **pi\_bbp\_terms()** outputs my value of terms
2. **pi\_bbp()** output my value of PI
3. **pi\_euler\_terms()** outputs my value of terms
4. **pi\_euler()** outputs my value of PI
5. **e()** outputs my value of E
6. **e\_terms()** outputs my value of terms
7. **pi\_wallis()** output my value of PI
8. **pi\_wallis\_factors()** outputs my value of factors
9. **pi\_viete()** outputs my value of PI
10. **pi\_viete\_factors()** outputs my value of factors
11. **pi\_madhava()** output my value of PI
12. **pi\_madhava\_terms()** outputs my value of terms
13. **sqrt\_newton(x)** outputs my value of sqrt x
14. **sqrt\_newton\_iters()** outputs my value of terms

## Results

My program runs as it's supposed to although some values are off by a small amount, however I was told that this was to be expected and will be compensated for. That being said, I know that mathlib-test isn't the most efficient but I was in a time crunch and I don't think we're graded on efficiency yet. That being said, I will try to make the code more efficient in the near future. This assignment is being turned in on Friday not because it's late but because I was given an extension. Please let me know if something does not run correctly although I've triple checked everything and I'm pretty confident in it.



```
yzsims@13s-vm: ~/cse13s/asn2$ ./mathlib-test -a -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
bbp() terms = 11
pi_madhava() = 3.141592653589800, M_PI = 3.141592653589793, diff = -0.000000000000007
madhava() terms = 27
pi_euler() = 3.141592558095903, M_PI = 3.141592653589793, diff = 0.000000095493891
e() terms = 18
pi_viete() = 3.141592653589789, M_PI = 3.141592653589793, diff = 0.000000000000004
viete() terms = 24
pi_wallis() = 3.141592495717063, M_PI = 3.141592653589793, diff = 0.000000157872730
e() 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
```

Figure 1: Screenshot of the program running.

## Error Handling

N/A

## References

N/A