A Little Slice of Pi - CSE13S Assignment 2

Nicholas Reis

October 4, 2021

Program Description

This program will implement various formulas to approximate pi including: the Bailey-Borwein-Plouffe formula, Euler's formula, the Madhava series, and Viète's formula. It also includes the implementation of the Taylor series to approximate Euler's number e and a square root approximation using Newton's method. This is all done without the math.h library that implements some of these functions.

Test Harness

Within mathlib-test.c, there will be code that supports command line options for each of the functions written in this assignment. I utilize boolean expressions, getopt(), and a switch statement in order to take in options from the user and then run specific functions from our mathlib.h library.

Bailey-Borwein-Plouffe Formula for Pi

Within bbp.c, there will be code that approximates the value of pi to the first 15 decimal places using the Bailey-Borwein-Plouffe formula. Using the following equation, I was able to write this program.

$$p(n) = \sum_{k=0}^{n} 16^{-k}\left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6}\right).$$

Pseudocode

```
include stdio.h and mathlib.h

create static int variable to count iters
```

```
double subtractions function:
    create double to store first fraction
    create double to store second fraction
    create double to store third fraction
    create double to store fourth fraction
    return first - second - third - fourth

double pi_bbp function:
    set iters to 0
    create double for divider
    create double for term
    create double for pi_approx

    for loop that runs until term is less than EPSILON:
        if loop iteration is 0:
            term = subtractions(0)
            pi_approx = term
            increment iters
        else:
            divider = divider * 16.0
            term = subtractions(loop iteration) / divider
            add term to pi_approx
            increment iters
    return pi_approx

int pi_bbp_terms function:
    return iters
```

The summation from the equation is made into a for loop that runs until the previous term added is less than EPSILON. If the loop iteration is 0, the only number added to the approximation is the value of the subtraction function, since 16ˆ-k is 1 when k is 0. Otherwise, the previous term is added to the outcome of (subtractions(i) / divider), creating the new term that is then added to the approximation.

Euler's Number

Within e.c, there will be code that approximates the value of Euler's number e up to the first 15 decimal places. Using the following formula, I was able to

$$\frac{x^k}{k!} = \frac{x^{k-1}}{(k-1)!} \times \frac{x}{k}.$$

write this program.

Pseudocode

```
include stdio.h and mathlib.h

create static int variable to count iters

double e function:
    set iters to 0
    create double for e_approx
    create double for term
    create double for k

    for loop that runs until term is less than EPSILON:
        if (loop iteration equals 0 or 1):
            k = 1.0
        else:
            k = k + 1
        create double multiplier = 1.0 / k
        term equals term * multiplier
        add term to e_approx
        increment iters
    return e_approx

int e_terms function:
    return iters
```

The program includes a for loop that runs until the previous term added is less than EPSILON. If the loop iteration is 0 or 1, the denominator value is just 1 (because 0! and 1! both equal 1). Otherwise, the multiplier becomes 1/k, with k increasing by 1 each iteration. The previous term is multiplied by this number to create the next term that is then added to the approximation.

Euler's Solution for Pi

Within euler.c, there will be code that approximates the value of pi using Euler's solution. Using the following equation, I was able to write this program.

$$p(n) = \sqrt{6 \sum_{k=1}^{n} \frac{1}{k^2}}$$

Pseudocode

```
include stdio.h and mathlib.h
```

```
create static int variable to count iters

double square function:
    return input * input

double pi_euler function:
    set terms to 1
    create double for e_approx
    create double for previous_term
    create double for k

    for loop that runs until previous_term is less than EPSILON:
        previous_term equals 1 / square(k)
        add term to e_approx
        increment iters
        increment k
    return sqrt_newton(pi_approx * 6)

int pi_euler_terms function:
    return iters
```

The program includes a for loop that runs until the previous term added is less than EPSILON. I made my own square function, as it was useful for this series. To calculate each term, I set previous_term equal to 1/square(k), where k incremented by 1 each iteration. When reaching the end of the function, I made sure to multiply the approximation by 6 and use my sqrt_newton function to square root the solution.

Madhava Series for Pi

Within madhava.c, there will be code that approximates the value of pi using the Madhava series. Using the following equation, I was able to write this program.

$$\sum_{k=0}^{\infty} \frac{(-3)^{-k}}{2k+1} = \sqrt{3}\tan^{-1}\frac{1}{\sqrt{3}} = \frac{\pi}{\sqrt{12}}$$

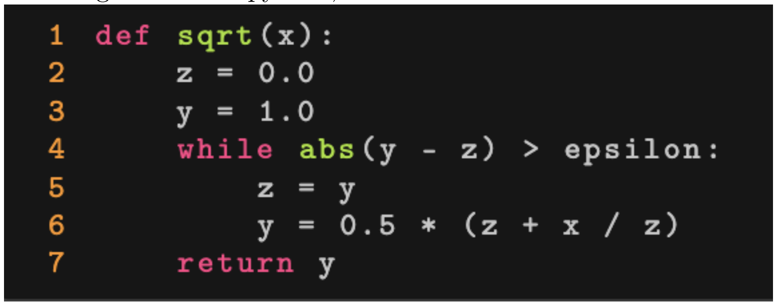
Pseudocode

```
include stdio.h and mathlib.h

create static int variable to count iters

double pi_madhava function:
    set iters to 0
    create double for divider
    create double for term
    create double for pi_approx

    for loop that runs until term is less than EPSILON:
        if loop iteration (i) is 0:
            term = 1.0
            pi_approx = term
            increment iters
        else:
            divider = divider * -3.0
            term = ((1.0/(2.0 * i + 1.0)) * divider) + term
            add term to pi_approx
            increment iters
    return (sqrt_newton(12) * pi_approx)

int pi_madhava_terms function:
    return iters
```

As you may notice, the code for this function is quite similar to the formula used in bbp.c. The main difference with this solution is that the pi_approx must be multiplied by the square root of 12 before returning it, since the outcome of this loop will actually turn out to be pi/sqrt(12). Thus, I utilized my sqrt_newton function to first calculate the value of sqrt(12) before multiplying it by pi_approx and returning the outcome.

Newton's Method for Square Roots

Within newton.c, there will be code that approximates the square root of a number using Newton's method. Professor Long wrote the following code in python, so credit to him for what I wrote in C.



```python
def sqrt(x):
    z = 0.0
    y = 1.0
    while abs(y - z) > epsilon:
        z = y
        y = 0.5 * (z + x / z)
    return y
```

Pseudocode

```
include stdio.h and mathlib.h
```

```
create static int variable to count iterations

double sqrt_newton function:
    create double z
    create double y
    create double x to store input

    while loop that runs until absolute value of (y - z) is less than EPSILON:
        set z equal to y
        y = 0.5 * (z + (x / z))
        increment number of iterations
    return y

int sqrt_newton_iters function:
    return computed iteration count
```

Viète's Formula for Pi

Within viete.c, there will be code that approximates the value of pi using Viète's formula. Using the following equation, I was able to write this program.

$$\frac{2}{\pi} = \frac{\sqrt{2}}{2} \times \frac{\sqrt{2+\sqrt{2}}}{2} \times \frac{\sqrt{2+\sqrt{2+\sqrt{2}}}}{2} \cdots$$

Pseudocode

```
include stdio.h and mathlib.h

create static int variable to count iters

double pi_viete function:
    set iters to 0
    create double for term = sqrt_newton(2.0)
    create double for pi_approx = term / 2.0
    create double for previous_approx
    create double for diff

    for loop that runs until diff is less than EPSILON:
        previous_approx = pi_approx
        previous_term = sqrt_newton(2.0 + term)
        pi_approx = pi_approx * (term /2.0)
        diff = absolute(pi_approx - previous_approx)
    return (2.0 / pi_approx)

int pi_viete_factors function:
    return iters
```

The program was definitely the trickiest one for me to figure out, as it seems to be the most different from each of the other ones. How I approached this equation was by using the difference in values between the previous approximation and the current one, as it was the only part of the solution that will trend towards 0. I made it such that the next term would just be the previous one with 2 added to it, then square rooted. Because dividing by 2 would cause issues here, I only used my previous_term variable to store numerators and divided the two when I multiplied it into the approximation. I then had to return 2 / pi_approx, as the equation does not solve for pi, it solves for 2 / pi, so to get pi isolated, I had to return it in this fashion.

Design Process

- Began with e.c, utilized the equation found in the assignment 2 pdf to structure my program
- Wrote most of the program but during testing realized that I had accidentally made the code utilize factorials (thus causing my output to be different)
- Rewrote the section to only multiply the previous term by 1/(iteration count) to replicate the Taylor series but also not using numbers that would become too large too quickly
- Made final adjustments such as adding the e_terms function and having the for loop run until the previous term added was less than EPSILON
- Began writing newton.c, utilized Professor Long's python code from the assignment 2 pdf to help write the code in C
- Added the sqrt_newton_iters function to the bottom of the file
- Rewatched Eugene's section video on Yuja for help on how to write my mathlib-test.c file
- Wrote the test harness using boolean expressions, along with the getopt() function and a switch case to handle each of the possible OPTIONS
- If statements at the end of the file to test if the booleans are true, then print out the corresponding outputs
- Attended Sloan's in-person section to learn about creating a functional Makefile
- It includes some of the things from the assignment 1 Makefile, but also utilizes: EXEC, SOURCES, OBJECTS, LDFLAGS, and tidy in order to have the Makefile be usable in future assignments
- Began working on bbp.c, decided to use a function named "subtractions" in order to calcuate the equation within the for loop

- Noticed that I did not have to deal with the 16 ˆ -k number, I could just multiply the previous term by 1/16 each iteration
- Began working on madhava.c, found that it was very similar to bbp.c. Pretty straightforward, just had to make sure to multiply the approximation by sqrt(12) to actually get the value of pi (using my sqrt_newton() function).
- Started writing euler.c, which was relatively similar to e.c. Decided to make a square function for better readability.
- Translating the summation to code was pretty simple, just set the next term to 1/square(k), where k starts at 1 and increments by 1 each iteration. Then add the term to the approximation until the last term added is less than EPSILON.
- Had to make sure to multiply approximation by 6 and square root it for the correct value of pi.
- Wanted to get started on Viète's solution, however it took me awhile to first think about the equation and understand how I could translate it into code first. I realized it was quite different from some of the other functions, so I took a different approach to writing the code for it.
- Considering I had not yet tested some of my functions before submitting my first DESIGN.pdf draft, I went back and tested each function using my test harness.
- Most of my intial code was correct and only needed minor modifications to come to the correct output.
- Went through my test harness and updated it with all of the statements that need to print out from the OPTIONS
- Wrote README.md and updated this document