Assignment 2: A little slice of pi

By: Derfel Terciano

Background:

In this assignment, we need to implement different calculations that calculate pi, e (Euler's number), and square roots. We must then implement a test-harness that compares how accurate our calculations are to the math.h library values. Lastly, our test-harness must use command-line options in order to call the different calculations.

The outputs of each calculation must be compared to the corresponding numbers of M_PI and M_E from the math.h library. Lastly, we must calculate the difference between our calculation and the corresponding math.h variable.

Files to submit:

- bbp.c
- e.c
- euler.c
- madhava.c
- mathlib-test.c
- mathlib.h
- newton.c
- viete.c
- Makefile
- README.md
- DESIGN.pdf
- WRITEUP.pdf

Calculating *e*

For calculating e, we can use the Taylor series in order to get a very close approximation of e. Below is an image of how to calculate e using the Taylor series:

$$e = \sum_{k=0}^{\infty} \frac{1}{k!} = 1 + \frac{1}{1} + \frac{1}{2} + \frac{1}{6} + \frac{1}{24} + \frac{1}{120} + \frac{1}{720} + \frac{1}{5040} + \frac{1}{40320} + \frac{1}{362880} + \frac{1}{3628800} + \cdots$$

The main difficulty of calculating *e* in C is that we have to calculate a factorial which can become very big. So in order to combat this, we can use a for loop and just calculate 1/k in multiplicative iterations. By using multiplicative iterations, we can just use the fact that:

$$\frac{1}{k!} = 1 \cdot \frac{1}{2} \cdot \frac{1}{3} \cdot \ldots \cdot \frac{1}{k}$$

Pseudocode for calculating *e*

Using python as my pseudo code, I have came up with the following:

```
factorial_term = 1
k = 1
summation = 0
EPSILON = 1e-14

while term > EPSILON:
    factorial_term = 1;
    for i in range(k):
        factorial_term *= 1/i
    k += 1
    summation += factorial_term
```

This just says to calculate each factorial term and then to add it to a summation variable. We stop calculating when the factorial term is smaller than epsilon

PI

Now, in order to calculate pi, there are many different ways as to how to calculate pi. In this assignment, we will be using the Madhava Series, Euler's method, the Bailey-Borwein-Plouffe formula, and the Viete formula. (NOTE: When looking at the python pseudo code, ** means exponent. In C, I will have to implement my own loop that deals with the exponents.)

The Newton-Raphson method for approximating square roots

Before we begin calculating the different formulas and approaches for pi, we would need to figure out how to implement square roots. Though we can just call the math.h library, we are not allowed to do that in this assignment. As a compromise, we would be implementing the Newton-Raphson method in order to find square roots.

The Newton-Raphson method involves using the inverse of x^2 as an interactive algorithm that approximates the roots of any real-valued functions. This method uses a *Newton iterate* which is shown as:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}.$$

Pseudo code for approximating

```
1 def sqrt(x):
2     z = 0.0
3     y = 1.0
4     while abs(y - z) > epsilon:
5         z = y
6         y = 0.5 * (z + x / z)
7     return y
```

Credit to Professor Long for providing the pseudo code to the class

Approximating PI using The Madhava Series

The Madhava series is given to us as a rapidly converging series. This series can be expressed as the following:

$$p(n) = \sqrt{12} \sum_{k=0}^{n} \frac{(-3)^{-k}}{2k+1}$$

The difficulty with this series is that the terms of the summation go from positive to negative as it approaches zero. This can be seen in the following graph:

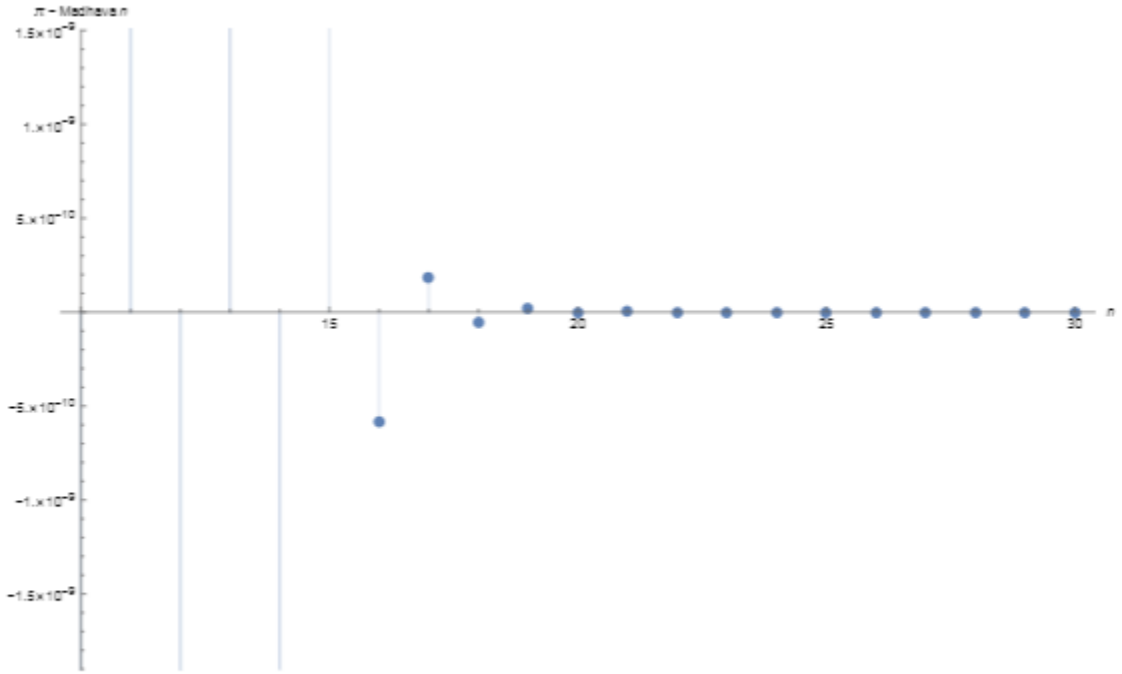

Figure 3: Madhava sequence $\sqrt{12}\sum_{k=0}^{n} \frac{(-3)^{-k}}{2k+1}$

As we can see here, the terms would be bouncing from negative to positive a lot. In order to combat this, we can just use the absolute function in the mathlib.h file. From there we can just compare the absolute value of the current term to epsilon.

Pseudo code for calculating pi using the Madhava series

Using python as my pseudo code, I came up with the following:

```
summation = 0
k = 0
term = 1
EPSILON = 1e-14

while (abs(term) > EPSILON):

    numerator = (-3)**(-k)
    denominator = (2* k) + 1

    term = (numerator / denomina
    summation += term

    k += 1

result = summation * sqrt(12)
```

This pseudo code is saying to calculate (-3)^-k and 2k+1 and then divide them together. Then you would take the result of that fraction and add it to the summation variable. Before returning the summation result, you would need to multiply it to sqrt(12). We can just use Newton's square root method that we implemented earlier.

Euler's Solution

Euler's solution to approximating pi is given to us with the following equation:

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

This equation is rather easy to implement since we don't have to deal with any factorials or exponents. The only difficult part to implement is the square root but we have already implemented it using Newton's method.

Pseudo code for Euler's Solution

Just like previously, I will be using python to show my pseudo code.

```
summation = 0
k = 1
term = 1
EPSILON = 1e-14

while term > EPSILON:
    term = 1 / k**2
    summation += term
    k += 1

result = sqrt(6 * summation)
```

Here, I first determine what 1/(k^2) is. Then I add that term to the summation variable. To get our result, we would take what the summation value was, multiply it by 6 and then square root out the entire result.

The Bailey-Borwein-Plouffe Formula

The Bailey-Borwein-Plouffe Formula (bbp) can be given as the following:

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

When implementing this formula, I prefer to leave the terms unsimplified. Simplifying the terms makes it look messy and becomes hard to implement.

Pseudo code for bbp

I am using python again for my pseudo code.

```
summation = 0
k = 1
term = 1
EPSILON = 1e-14


while term > EPSILON:
    term = (4.0 / (8.0 * k + 1.0)
     - 2.0 / (8 * k + 4.0)
     - 1.0 / (8 * k + 5.0)
     - 1.0 / (8.0 * k + 6.0)) * (16**-k)

    summation += term
    k += 1

result = summation
```

As you can see, we basically just copied the summation terms from the formula. The only hard part of implementing this is typing out the fractions.

Viète's Formula

This formula is given as the following:

$$\frac{2}{\pi} = \prod_{k=1}^{\infty} \frac{a_i}{2}$$

where $a_1 = \sqrt{2}$ and $a_k = \sqrt{2 + a_{k-1}}$ for all $k > 1$.

Now implementing this formula is particularly tricky since it uses an infinite product of nest radicals. If we were to write out each term, we would get a nasty equation that would look like the following:

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

In order to implement this nasty equation, what we can do is to keep track of a_{k-1}. If we keep track of this, then it would become pretty straightforward to implement. Lastly, we need to keep in mind that the formula gives us 2/pi not pi itself. So after some simple algebra, we can solve pi as:

$$\frac{2}{\prod_{k=1}^{\infty} \left(\frac{a_i}{2} \right)}$$

Pseudo code for Viète's Formula

Just like before, I will be using python to show my pseudo code:

```
import math

EPSILON = 1e-14

a = sqrt(2)
term = 1
result = 1
counter = 0

previous_term = 0
cur_term = 0

while(abs(cur_term - previous_term) > EPSILON):
    previous_term = term
    term = a / 2
    cur_term = term
    result *= term
    a = sqrt(2 + a)

result = 2 / result
```

Here, I'm starting *a_i* with sqrt(2) since we were told that *a_1* is sqrt(2). From there we divide *a* by 2 and save whatever that result is. Next, we would save the previous value of *a* then do the following: a_{k + 1} = sqrt(2 + a_k) to get the new value of *a*.

Since this formula only has terms that only get bigger, the only way we can halt the calculation is to take the difference of the previous term and the current term, and compare that to epsilon. As the calculation progresses, the difference between the previous term and the current term gets only smaller.

Command line handling

In the assignment 2 document, Eugene has provided us the template as to how to handle command line inputs. In order for the command line options to call the corresponding calculations, we must use a switch() statement to call them.

Credits

- All images of formulas and graphs were taken from Prof. Long's asgn2.pdf
- The pseudo code for Newton's square root method was given to us from Prof. Long's asgn2.pdf
- The command-line handling part of the assignment was provided to us by TA Eugene.
- The bbp code was provided to us from Prof. Long's 10/4/21 lecture.
- The TA Eugene taught us how to create Makefiles that link multiple files together in the 10/5/21 section.
- Eugene also taught us how to use gnuplot for our write up in the 10/5/21 section as well.