Analysis – A Little Slice of π

Thadeus Ballmer

9 October 2021

1 Introduction

Assignment 2's challenge was to implement a custom math library and test harness to compare results with the **math.h** library. Due to the shortcomings of numerical computation, all irrational solutions must be approximated. The custom library, **mathlib.h**, approximates the values of e through a Taylor series and π through various formulas including: Madhava series, Euler solution, the Bailey-Borwein-Plouffe Formula, and Viete's forumla. This library also implemented a square root approximation using the *Newton-Raphson method*.

2 Analysis

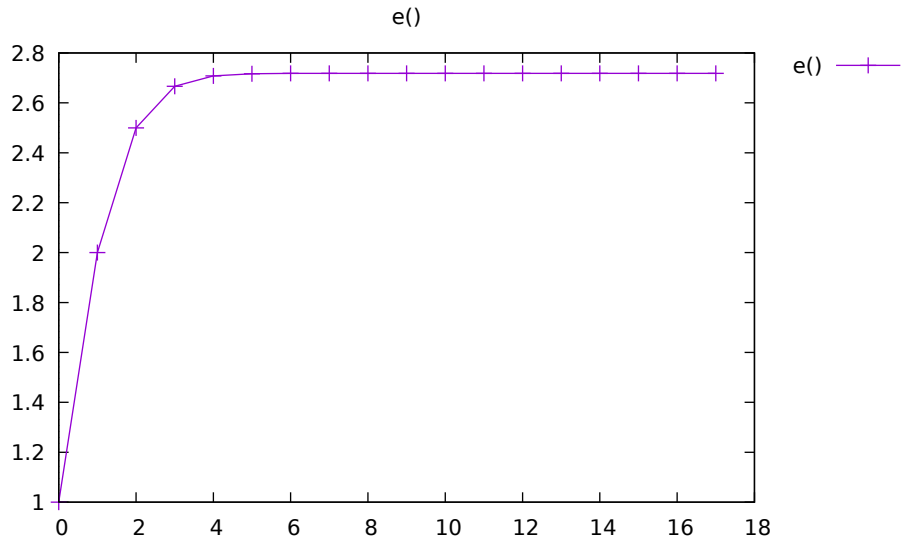
2.1 e()

2.1.1 Code

```c
#include "mathlib.h"

#include <stdio.h>
#include <stdlib.h>

static int num_of_terms = 0;

// e's approximation may be one of the more simple approximations in this assignment,
// but the program still needs to watch out for the first term (0! = 1). Other than
// that, the calculation 1 / k! is easily handled with a few variables.

double e(void) {
    double current_value = 1.0, factorial_value = 1.0;
    double total = 0.0;

    for (double i = 0; EPSILON < current_value; i++) {
        if (i == 0) {
```

```
            factorial_value = 1.0;
            current_value = 1.0;
        } else {
            factorial_value *= i;
            current_value = 1 / factorial_value;
        }
        num_of_terms++;
        total += current_value;
    }
    return total;
}
```

2.1.2 e() Analysis



This graph displays how this approximation takes very little terms in order to converge accurately. In fact, the **mathlib-test** test harness output below proves how this function is accurate at least up to 10^{-15} precision!

```
e() = 2.718281828459046, M_E = 2.718281828459045, diff = 0.000000000000000
e terms = 18
```

The speed and accuracy of this computation can in some part be attributed to its simplicity. With the formula:

$$e = \sum_{k=0}^{\infty} \frac{1}{k!}$$

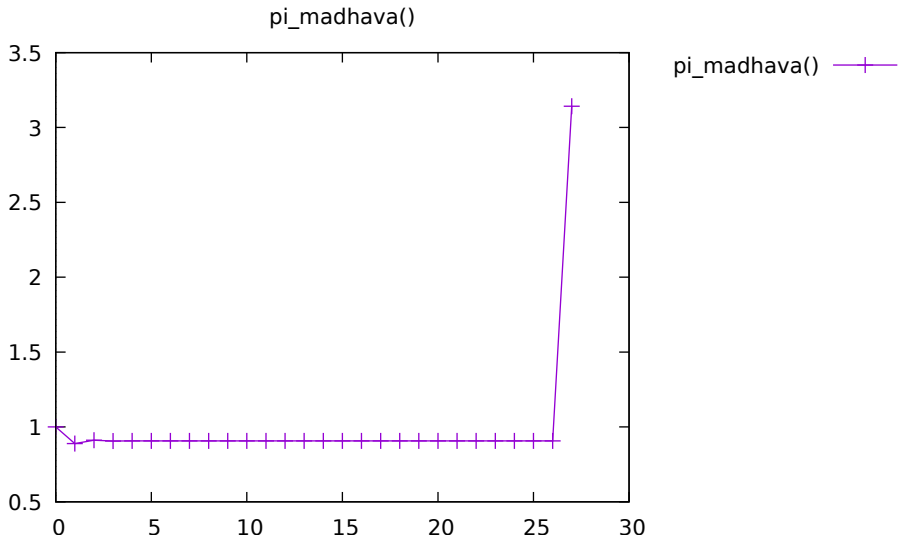
each term only has to compute the product of the previous term and $\frac{1}{k}$—a simple calculation for a computer. In this case, formulaic simplicity leads to computational efficiency and accuracy.

2.2 pi_madhava()

2.2.1 Code

```c
#include "mathlib.h"

#include <stdio.h>
#include <stdlib.h>

static int num_of_terms = 0;

// The madhava solution proved itself to be way more complicated in calculation
// than other formulas. Since -3 has an exponent -i, each new term flips signs.
// This is why absolute() is used on the current_value function. And this
// solution also had to address the first iteration having (-3)^0.

double pi_madhava(void) {
    double current_value, exponent_value, sqrt, total;
    current_value = 1.0;
    total = 0.0;
    sqrt = sqrt_newton(12.0);

    for (int i = 0; EPSILON < absolute(current_value); i++) {
        if (i == 0) {
            exponent_value = 1.0;
            current_value = 1.0;
        } else {
            exponent_value *= (-3);
            current_value = 1 / (exponent_value * (2 * i + 1));
        }
        num_of_terms++;
        total += current_value;
    }
    total *= sqrt;
    return total;
}
```

2.2.2 pi_madhava() analysis



On first impression of this graph, the behavior of pi_madhava() make seem peculiar. However, the pi_madhava() implementation of the Madhava formula relied on this behavior for additional accuracy. The Madhava formula:

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

multiples its series summation by $\sqrt{12}$. Since $\sqrt{12}$ is irrational, it can only be approximated through computational arithmetic. This means that each square root computation diminishes accuracy. This is why pi_madhava() calculates the product of the series and $\sqrt{12}$ only at the end of the calculation. This strategy proved successful; the **mathlib-test** output only calculated a +0.000000000000007 inaccuracy.

2.3 pi_euler()

2.3.1 Code

```c
#include "mathlib.h"

#include <stdio.h>
#include <stdlib.h>

static int num_of_terms = 0;

// In the Euler approximation of pi, the series summation itself is not the
// approximation of pi. The summation needs to be multiplied and square rooted.
```

```c
// The square root function used is sqrt_newton(), which is defined by myself
// in the newton.c file.

double pi_euler(void) {
    double current_value = 1.0;
    double total = 0.0;

    // I made the iterator i a double because I was having problems with C
    // converting the calculation into an integer.
    for (double i = 1.0; EPSILON < current_value; i++) {
        current_value = 1.0 / (i * i);
        current_value *= 6.0;
        total += current_value;
        num_of_terms++;
    }
    total = sqrt_newton(total);
    return total;
}
```

2.3.2 pi_euler() analysis

The Euler solution to approximating π proved less accurate and slower than its counterparts.[1] The formula sheds light on why the computation is problematic:

$$\pi = \sqrt{6 \sum_{k=1}^{\infty} \frac{1}{k^2}}$$

The series is simple—resembling e's approximation—but the summation must be multiplied by 6 and square rooted. The combination of multiplication and a square root approximation hinders this function's performance. Although my pi_euler() converges over twice as slow as the reference pi_euler(), my function is over twice as accurate as the reference in comparison to **math.h**'s π approximation. This discrepancy could be attributed to the square root strategy implemented in pi_madhava() also being implemented here. The loss of speed is made up for with a boost to accuracy. And with even the reference program suffering an inaccuracy of 0.000000095493891, this solution needs all the accuracy it can get.

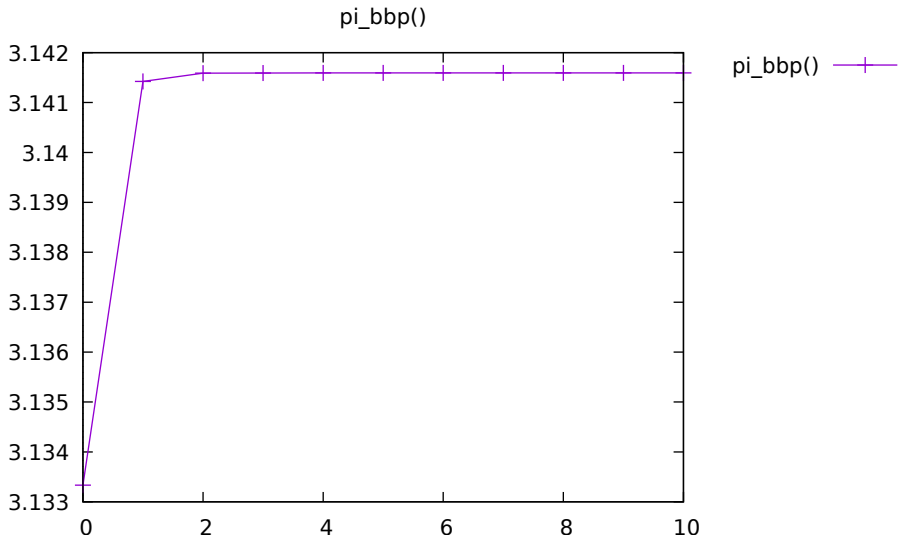
[1] I do not have a graph for this function because it took nearly 25,000,000 terms for the calculation to converge. The 25,000,000 terms overloaded and killed gnuplot. Although I'm sure there is a way to put this much data into gnuplot I have yet to find a solution.

2.4 pi_bbp()

2.4.1 Code

```c
#include "mathlib.h"

#include <stdlib.h>

static int num_of_terms = 0;

// I chose to implement the Bailey-Borwein-Plouffe formula in Horner normal form,
// which is why there appears to be many seemingly random numbers.

double pi_bbp(void) {
    double current_value = 1.0;
    double exponent_value;
    double total = 0.0;
    double fraction_value;

    for (double i = 0.0; EPSILON < current_value; i++) {
        fraction_value = ((i * (120.0 * i + 151.0) + 47.0)
                          / (i * (i * (i * (512.0 * i + 1024.0) + 712.0) + 194.0) + 15.0
        if (i == 0) {
            exponent_value = 1.0;
        } else {
            exponent_value *= (1.0 / 16.0);
        }
        current_value = exponent_value * fraction_value;
        num_of_terms++;
        total += current_value;
    }
    return total;
}
```

2.4.2 pi_bbp() analysis



pi_bbp() in *Homer normal form* is by far the most accurate and efficient π approximation in the **mathlib.h** library. The stunningly simple equation only needs multiplication, addition, and division to approximate π:

$$\pi = \sum_{k=0}^{\infty} 16^{-k}\left(\frac{(k(120k+151)+47)}{k(k(k(512k+1024)+712)+194)+15}\right)$$

By only needing these few operations, computer calculations match this equation perfectly. In fact, pi_bbp() was the only function in the **mathlib.h** library to compute π with 10^{-15} precision. At the same time, this function converged faster than any other approximation of π. When programming computations, the architecture and constraints of binary arithmetic must be kept in mind, and this equation addresses these constraints. After looking through the other series in the assignment document, I doubt even the Ramanujan series would be as accurate as the BBP series because the Ramanujan series multiplies its summation by a square root.

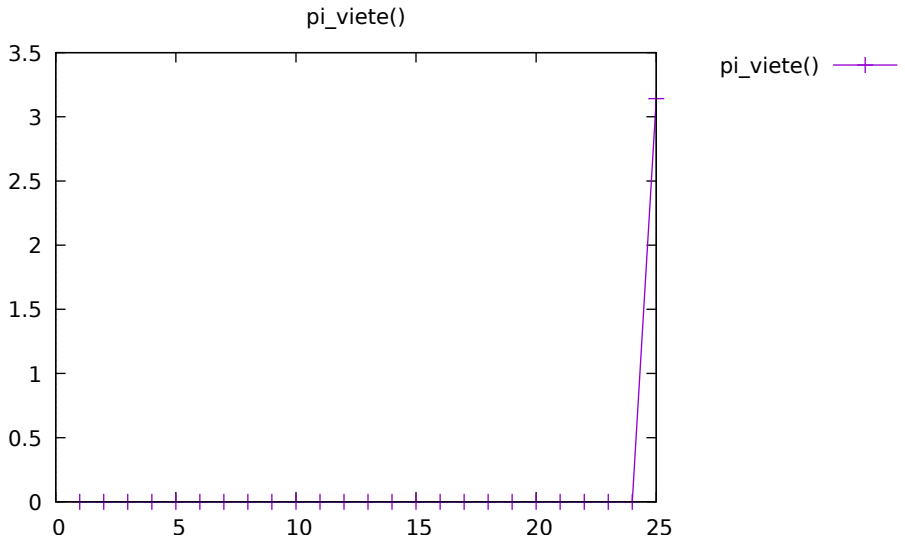
2.5 pi_viete()

2.5.1 Code

```
#include "mathlib.h"

#include <stdlib.h>

static int num_of_factors = 0;

// It made the most sense for me to use recursion to calculate the
```

```
// nested square roots because the recusive solution was so simple.

double rec(double x) {
    if (x == 1.0) {
        return sqrt_newton(2.0);
    } else {
        return sqrt_newton(2.0 + rec(x - 1.0));
    }
}

// Viete's formula is unique from the others because it used a product
// rather than a summation. Since stopping the calculation when the current
// value < EPSILON does not apply here. I found the difference between
// the previous factor and current factor and then compared to EPSILON.

double pi_viete(void) {
    double current_value = 0.0;
    double previous_value = 1.0;
    double total = 0.0;

    for (double i = 1.0; EPSILON < (previous_value - current_value); i++) {
        if (i == 1.0) {
            current_value = 1.0;
        } else {
            previous_value = current_value;
        }
        current_value *= (rec(i) / 2.0);
        num_of_factors++;
    }
    total = 2.0 / current_value;
    return total;
}
```

2.5.2 pi_viete() analysis



pi_viete() is a unique approximation in this library because it approximates π using a product of factors:

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

where $a_1 = \sqrt{2}$ and $a_k = \sqrt{2 + a_{k-1}}$ for all $k > 1$. As the graph and equation show, in order to end with the value of π, 2 must be divided by the total at the end of the product computation. This equation also provided the unique challenge of deciding when to end computation. Every other approximation compared the latest term in a summation to an $\epsilon = 10^{-14}$. In order to achieve the same level of accuracy in a product of factors, the difference between the current and previous products must be compared to ϵ. The end result was satisfactory when compared to **math.h**'s π approximation.

```
pi_viete() = 3.141592653589789, M_PI = 3.141592653589793,
diff = 0.000000000000004
pi_viete() terms = 24
```

Although not the fastest or most efficient solution, especially compared to pi_bbp(), the unique computation could be useful in certain scenarios.

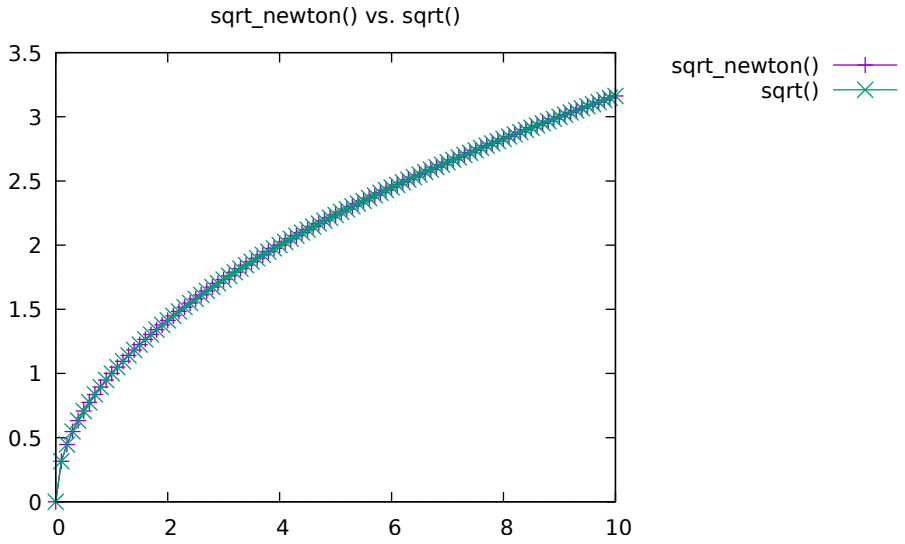
2.6 sqrt_newton()

2.6.1 Code

```
include "mathlib.h"

#include <stdio.h>
#include <stdlib.h>
```

```
static int num_of_iters = 0;

// This program is based on Professor Long's sqrt pseudocode
// found in the assignment 2 pdf. I simply converted his python
// code into C and kept track of the iterations.

double sqrt_newton(double x) {
    num_of_iters = 0;
    double z = 0.0;
    double y = 1.0;
    while (EPSILON < absolute(y - z)) {
        z = y;
        y = 0.5 * (z + x / z);
        num_of_iters++;
    }
    return y;
}
```

2.6.2 sqrt_newton() analysis



The *Newton-Raphson method* of approximating square roots proved computationally accurate and efficient. Only the square root of 0 had a difference greater than 10^{-15} with the **math.h** sqrt() function. And all calculations but 0 converged in 7 terms or less. Although some strange behavior persists like sqrt_newton(9.0) = 2.999999999999997, the sqrt(9.0) also = 2.999999999999997. These calculations are approximations afterall. The success of this computation relied on converting Professor Long's pseudocode into C, who understands how this method functions.

3 Conclusion

This assignment taught me so much about the programming libraries I took for granted. I also learned how arithmetic on a computer is implemented properly. Being able to convert a series discovered hundreds of years ago into a useful computation opened my eyes to how computer science operates. I now appreciate how instrumental computer science is in the development and aid of other sciences.