Analysis - A Little Slice of Pi
James Gu
Fall 2021

1. INTRODUCTION

Computers can only do the simple operations of multiplication, division, addition, and subtraction so functions like integrals need to be computed using its Taylor series expansion. Not only integrals can be represented through these summations but most functions can be written as an infinite sum. In this program the goal is to approximate the value of e using the Taylor series, the value of pi using the Madhava series, Euler's solution to the Basel problem, Bailey-Borwein-Plouffe formula, and Viete's formula, and the square root of an argument using the Newton-Raphson method.

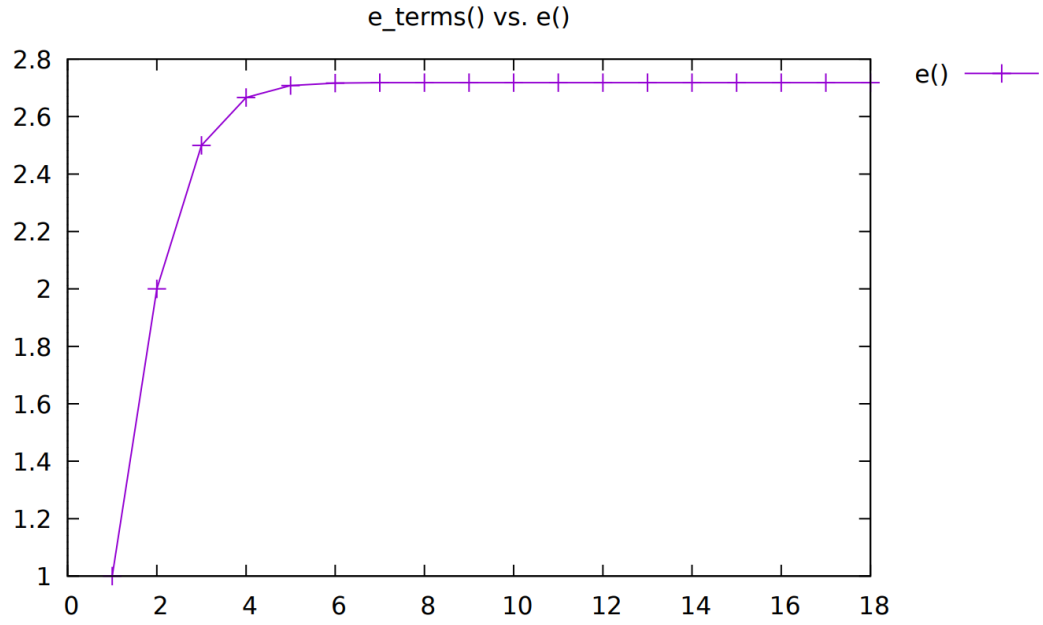
2. ANALYSIS

e.c:

The first program I did was e.c. It contains two functions: pi_euler() and pi_euler_terms(). Essentially this program would calculate e using the Taylor Series:

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

My implementation of this Taylor Series involved a while loop that loops until the difference between the current term being added and the previous term was less than the given epsilon. The results were as follows:



The difference between my implementation and math.h's was 0.000000000000001 which makes sense because each additive term lets the value get closer and closer to e. The limit of the summation is e so it makes sense that the output would be so close to it. So it only took around 18 terms.

bbp.c:

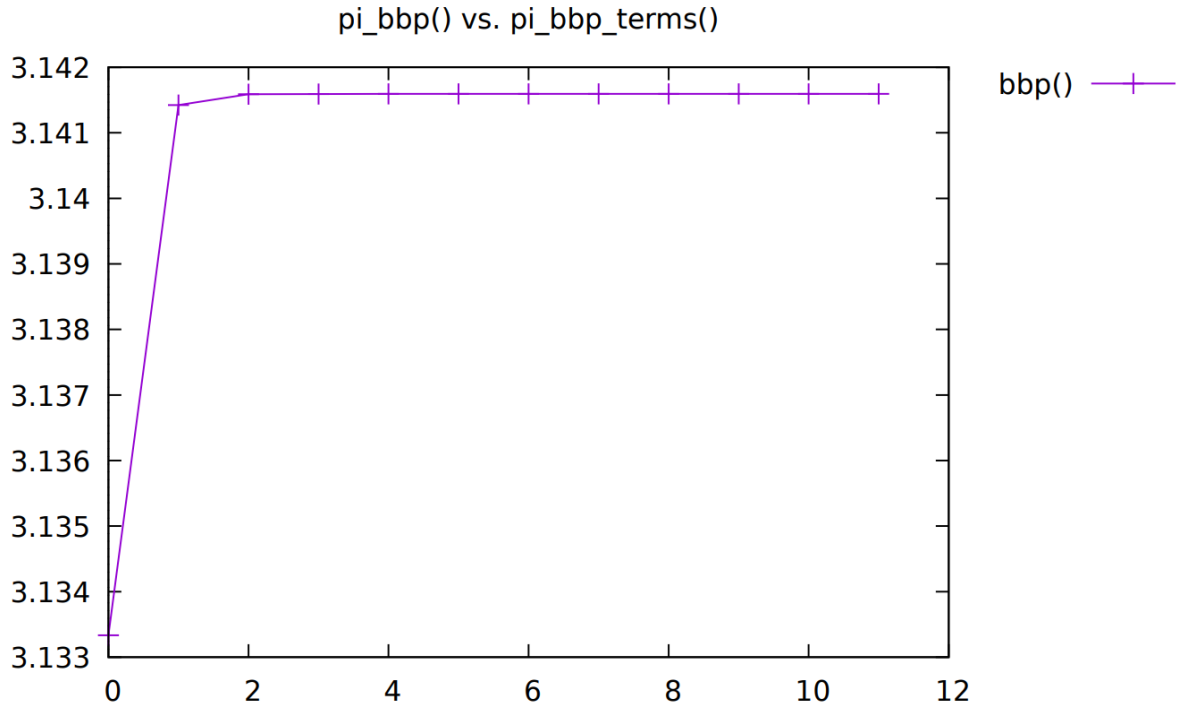
The second program to be implemented was bbp.c which had the functions pi_bbp() and pi_bbp_terms(). This program calculates pi using:

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

And if you desire to reduce it to the least number of multiplications, you can rewrite it in *Horner normal form*:

$$p(n) = \sum_{k=0}^{n} 16^{-k} \times \frac{(k(120k+151)+47)}{k(k(k(512k+1024)+712)+194)+15}.$$

Either the top summation or the bottom summation work so I decided to implement the bottom function with a while loop that added the mess of a fraction every iteration and it worked pretty well at approximating pi:



As the amount or summations increases it gets closer to pi as shown from the jump from 0 to 1 and 1 to 2. Around a dozen terms is all it took for the summation to be close enough to pi that the output difference is 0.000000000000000.

euler.c:

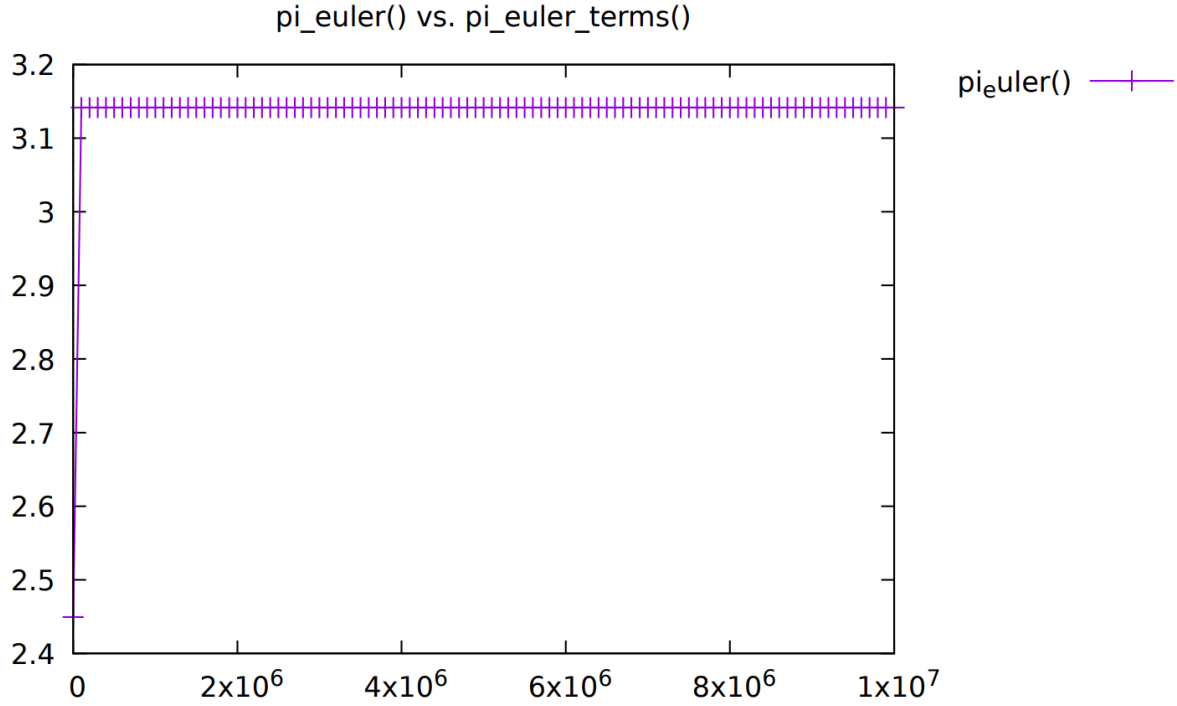
The third program is euler.c which includes the functions pi_euler() and pi_euler_terms(). This program estimates pi using:

$$\sum_{k=1}^{\infty} \frac{1}{k^2} = \frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \cdots = H_{\infty}^{(2)},$$

which again involves harmonic numbers. Euler's solution showed that the solution is $\pi^2/6$, but his method gave us this series:

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

This summation takes a very very long time to converge to pi. My attempt with a while loop that adds 1/k^2 every iteration took me around 2 million terms:



The reason why this function takes so long is because there is such a marginal increase every term yet is still above epsilon that it gets to the point where there's an unimaginable amount of summations. Because of these extremely extremely small increments the difference between the code and actual pi is the greatest for this function with 0.000000095493881 as the difference.

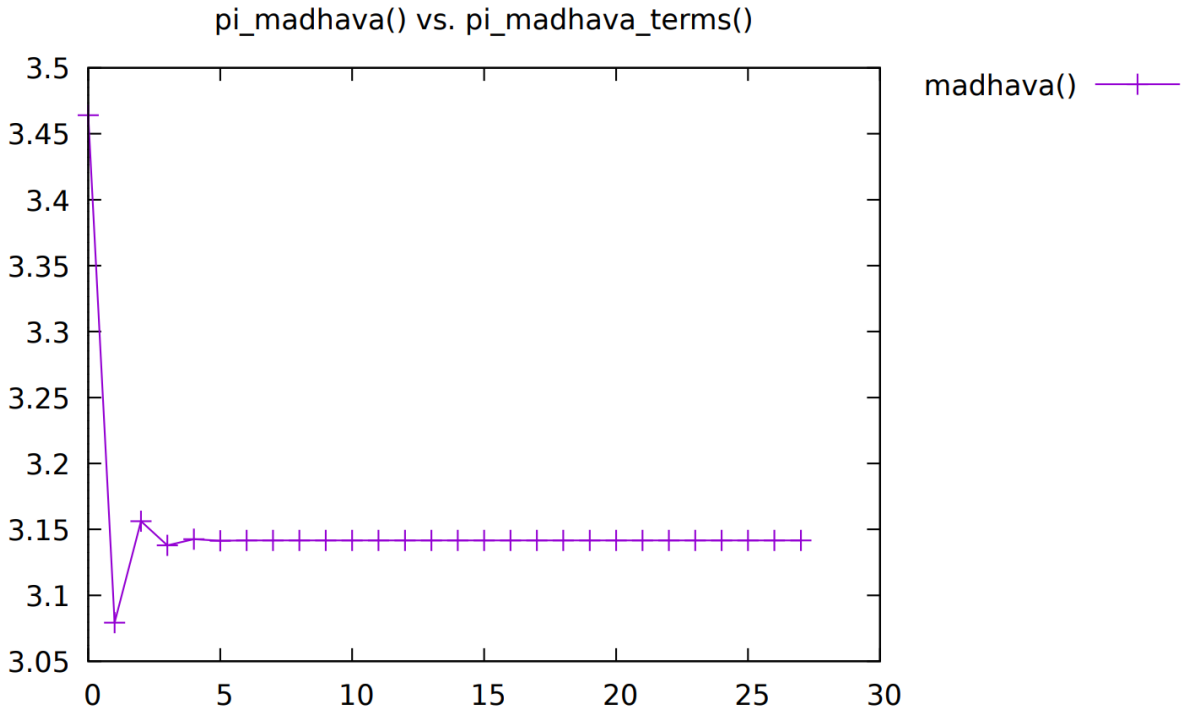
madhava.c

There was also madhava.c which had pi_madhava() and pi_madhava_terms() in it. This program would solve for pi using the following formula:

The Madhava series (Mādhava of Sangamagrāma, c. 1340 – c. 1425) is also related to $\tan^{-1} x$,

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

A little more steps are involved for this one to solve pi. At the end of the implementation it is required to multiply by the square root of 12 to get pi. Like the other functions, this one I used a while loop to solve for. But additionally required another variable for the exponent (-3)^-k. This is what my results got me:



Unlike the other functions the value of pi goes back and forth to converge to the same value which might explain the slightly larger difference compared to e.c of 0.000000000000002 with a little over 25 terms. In a way the other summations brute force finding pi by adding smaller numbers, but this one finds pi with its use of the relation to arctan(x) in its formula.

viete.c

Viete.c used the functions pi_viete() and pi_viete_factors(). The reason why it uses factors and not terms is because the formula involves multiplication:

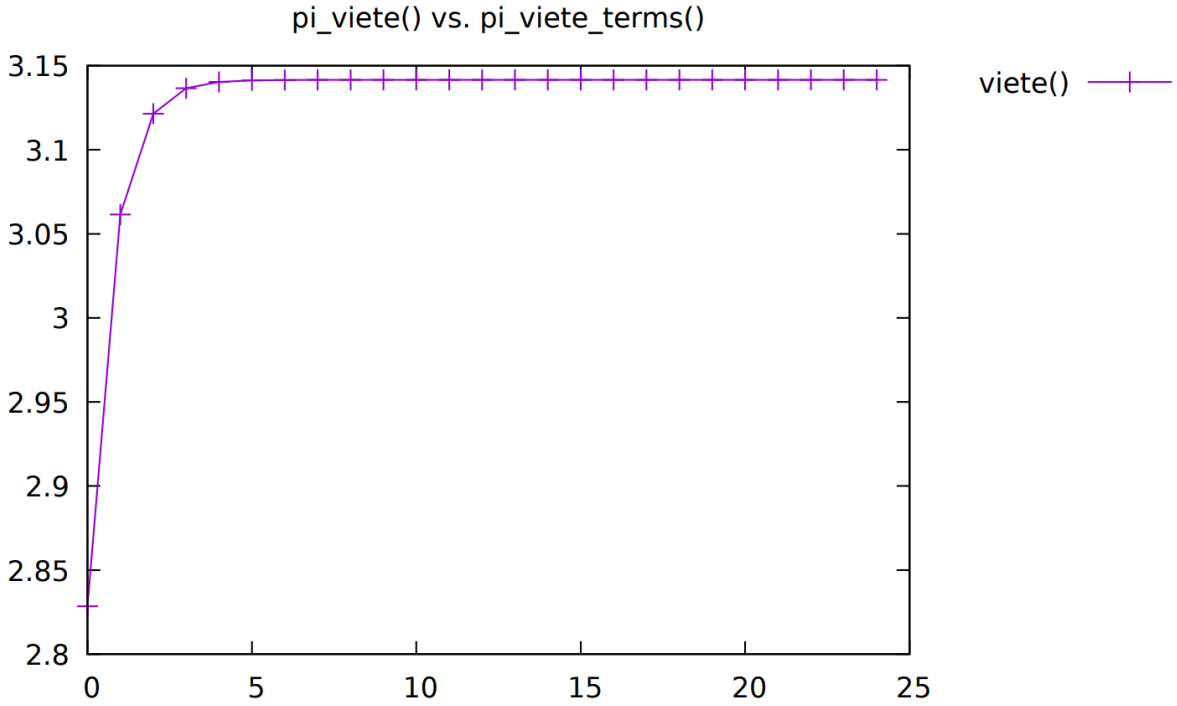
Viète's formula can be written as follows:

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

Or more simply,

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

Viete's implementation involved building up the square root in the formula using the same term over and over again but adding a 2 and square rooting it. I did this with a while loop and instead of adding each current term to the final solution, I multiplied it. These are my results:



About 25 terms is all that was needed to find a number close enough to pi and surprisingly the difference found was 0.000000000000000. This could be because multiplication is innately more efficient than addition so it'll be easier to be more exact.

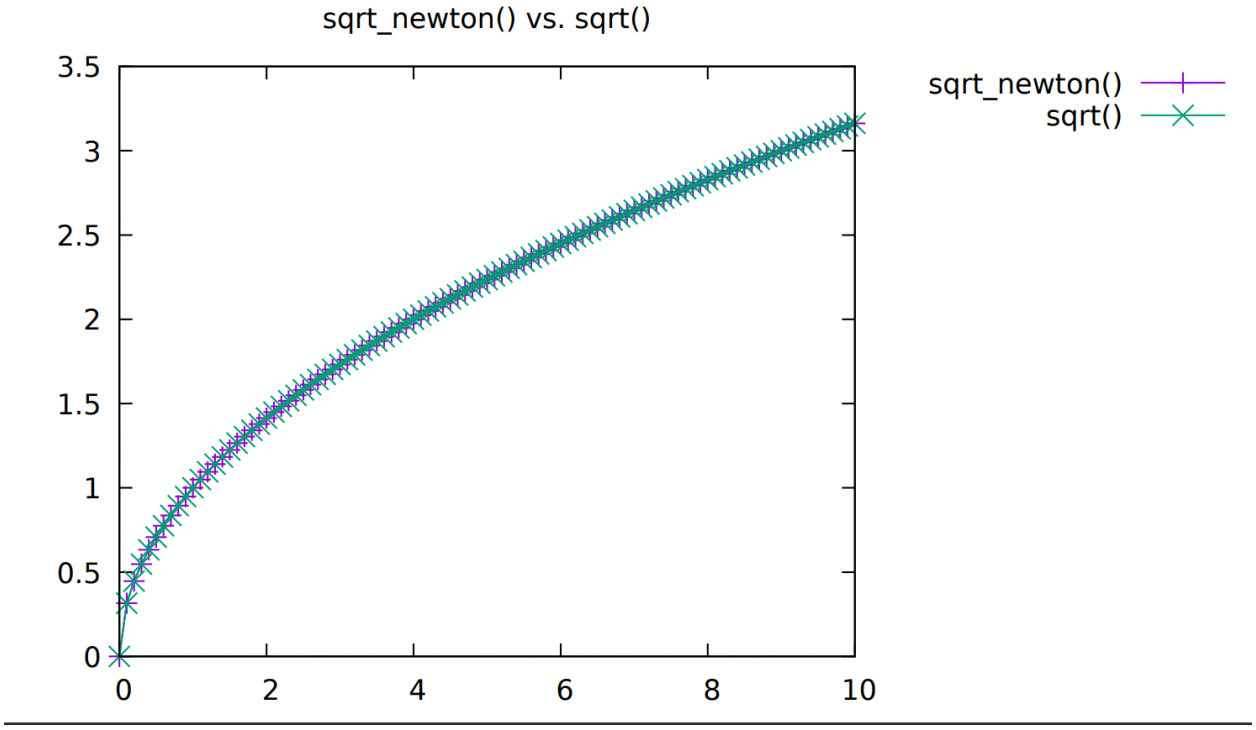
newton.c

Newton.c was given by the professor in the assignment 2 document but it goes as follows:

mations. A *Newton iterate* is defined as:

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

The theory for the implementation is that given a lot of guesses for the square root the program will get the square root of a number. So the initial guess is 1 and then Newton iterates the upcoming guesses closer and closer to the actual square root until it is unintelligible. The comparison is as follows:



With this graph it demonstrates how close the guess gets to the actual square root function with each square root having a difference of 0.000000000000000 and square root 0 having a difference of 0.000000000000007. With how many guesses this iterate takes at around 1703 for the square root of 10, it makes sense the guesses become accurate.