The Calculation of Mathematical Constants in the C Language

Alex Asch

October 2021

1 Abstract

This paper outlines the findings gained by calculating the mathematical constants of π using various methods as well as calculating e and the square root of a given number. The methods for calculating these constants are all through repeated arithmetic operations, and thus are implemented as loops in the C language. As floating point arithmetic is not exact, all of these values are calculated to within a certain degree of accuracy. The way this is ascertained is by taking the change between the previous and current output, and checking if it is larger than a certain value, which will be referred to as Epsilon, and is $\epsilon = 10^{-6}$ for the duration of this paper. The purpose of this is to calculate these constants to a certain degree of accuracy, and see the speed at which these various methods converge to the given value.

2 Introduction

To begin, as outlined in the abstract, the nature of this research is to implement in an iterative fashion in the C language, various methods of calculating mathematical constants. After this calculation, the calculated constant is compared with the given value from the math.h library, which will serve as a reference point for the true value of the number. While this is not truly accurate, as π and e are both irrational, and floating point arithmetic is not truly accurate, we will treat these approximations as the values of π and e. The methods used to calculate π include Viete's method, Madhava's method, Euler's method, and the Bailey-Borwein-Plouffe formula. The square root is calculated using the Newton-Raphson method, and e is calculated using a Taylor polynomial. These methods are shown below. What is noticeable of these methods is that all except the Newton-Raphson method are using repeated multiplication or addition, and thus can easily be evaluated in an iterative fashion. The Newton-Raphson calculates the next term off of the previous term, thus is also similarly implemented. By evaluating these results in this way, it is easy to see the comparatives efficiencies of the various methods, by the number of iterations of each loop. Given these results, there will be an analysis on the relative efficiency of each method, and why that is the case.

3 Materials and Methods

Madhava's Method

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

Euler's Method

$$\sqrt{6\sum_{k=0}^{\infty} \frac{-1}{k^2}} = \pi$$

Bailey-Borwein-Plouffe Formula

$$\sum_{k=0}^{\infty} 16^{-k} * (\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6}) = \pi$$

Bailey-Borwein-Plouffe Formula Horner Normal form

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

Viete's Formula

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

Where a_1 is $\sqrt{2}, and\mathrm{a}_k$ is $\sqrt{2+}\mathrm{a}_{k-1}1$
Calculating E

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

Newton-Raphson square root

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

This section will cover the various methods and materials used to perform the testing. To begin, the above mathematical equations demonstrate the various methods we will be using to calculate π. These equations are then implemented in the C language through looping iteration. This means that each iteration of the above summations will add another term to the summation, and since each summation approaches the given constant, the added term will also approach a given number, that number being zero. For the repeated multiplication, this same behavior is seen, but with the number being 1. Thus we will approximate the above summations in an iterative fashion until the term is within epsilon(defined above) of the constant the term approaches to. The implemented code for some summations initializes at the first term, as terms are calculated using the counter value and previous term. This is the case for Madhava's method and Bailey-Borwein-Plouffe Formula. The materials used for this are a Linux virtual machine to write, compile, and run code in the C language.

4 Results

The two important results to discuss in this case are the number of iterations taken, and the difference between the result approximated and the constant given. For approximating e, the approximation we use calculates e in 18 terms, and calculates e as 2.718281828459046, with a difference of 0 from math.h E for the first 14 digits of e. The difference still displays as all zeros even though the 14th digit of math.h e and the approximated e due to the number in reality being longer and rounding.

For π, each of our methods iterates a different amount of times and with a different margin of error. By far, the least efficient and accurate method is Euler's method, which iterates a total of 10000000 times, and has a difference of 0.00000009549381. Then, next we have the Madhava series, which iterates 27 times, and has a difference of $7 * 10^{-14}$. Then we have Viete's method of repeated multiplication, which iterates 24 times, with a difference of $3 * 10^{-14}$, and most efficient is the Bailey-Borwein-Plouffe method with 11 iterations and a difference of $2 * 10^{-14}$.

In addition, we compare our square root method with the given method for numbers between 0 and 10 with a step of 0.1. Most of these numbers iterate in a similar number of terms, being between 5 and 7, except of 0, which takes 47 terms, and 1, which takes 1 term. The meaning of and reasoning of these results will be discussed in the next section, after the below graphs demonstrating the converging of these functions to the given constant.

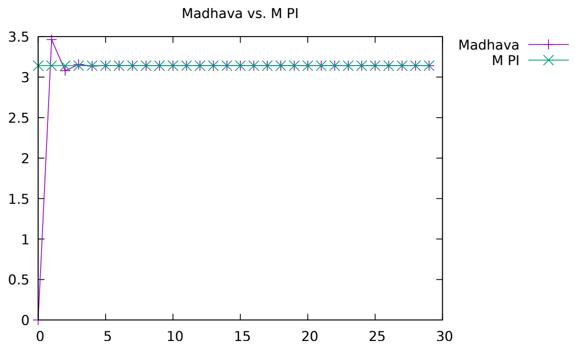

Figure 1: Madhava series convergence

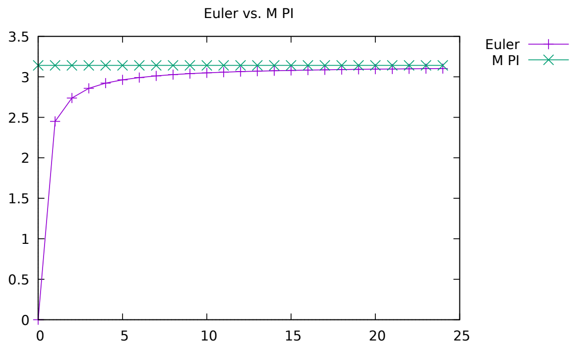

Figure 2: Euler Series convergence, small domain

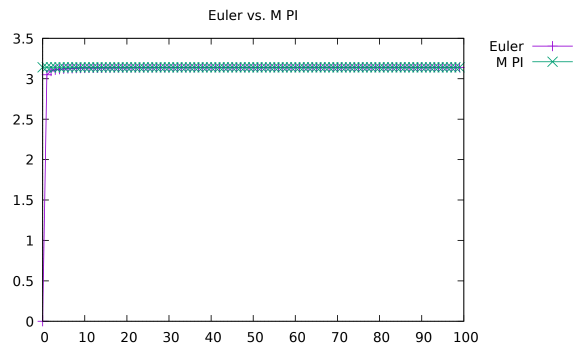

Figure 3: Euler series convergence, large domain

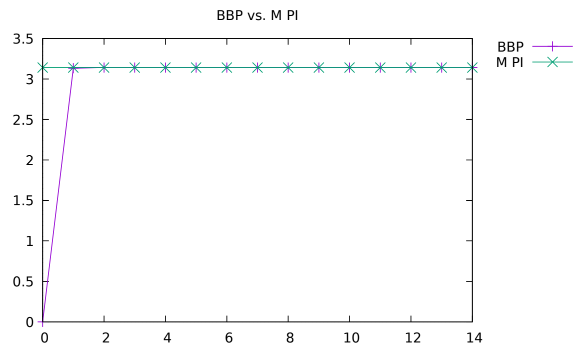

Figure 4: Bailey-Borwein-Plouffe convergence

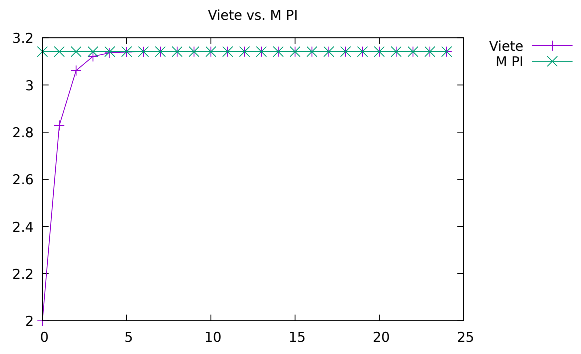

Figure 5: Viete's method convergence

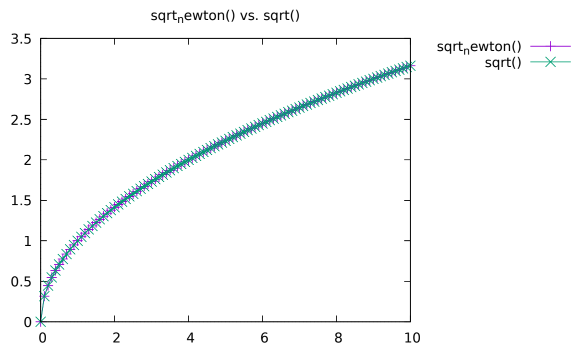

Figure 6: Square root in range of 0 to 10

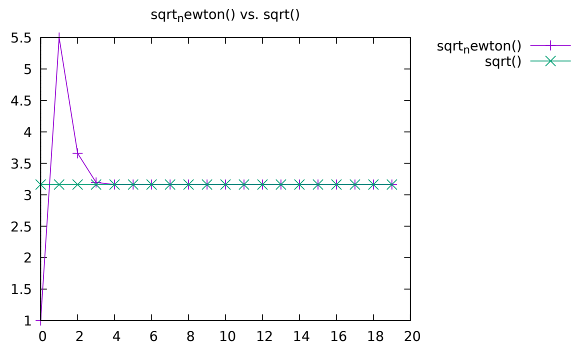

Figure 7: Square root convergence for 10, with terms

5 Discussion

Now that we have the above results, it is important to discuss why these results are as seen. To begin, the approximation for e converges rapidly and approximates the value accurately in 14 terms since the factorial of k, the denominator of the term grows rapidly. This means that the term rapidly approaches 0, and thus converges to below the epsilon value rapidly.

Next, the relative efficiency of the pi methods must by approached. To begin, the slow pace at which Euler's method converges is known as 10000000 terms. This is due to the successive terms being harmonic numbers, with each term being $1/k^2$ which converges very slowly, since each successive term is not a large amount greater to the previous. Since this means that a very large number of iterations must be done, this explains the relative inaccuracy of this method, since the terms fall below the threshold of epsilon before a more accurate approximation can be found.

The next most efficient method is the Madhava method, as can be seen in the graph, the approximation fluctuates between greater than and less than the given constant, due to the exponent of an exponent. This oscillation allows each term to be a larger change, since the method is not purely additive. Since the term is multiplied by an negative exponent, this makes the series converge to 0 much faster than Euler's method.

Next, we have Viete's method. This method iterates 24 times, marginally more efficient than the Madhava method. This method differs from all of the above methods, due to being repeated multiplication instead of a summation. Thus, this method has terms that converge to one instead of zero, since as seen in the above equation, by repeated nesting of the radical, the numerator approaches two and the term approaches one. For the purposes of this implementation, the reciprocal of the repeated multiplication being equal to $\pi/2$ was used. As can be seen in the graph, the method converges smoothly, and does not every exceed π, since it is purely multiplicative.

Finally, the most efficient method is the Bailey-Borwein-Plouffe. This method uses a repeated exponent of a negative, to cause an oscillation of around π, and converges very rapidly. The rapid nature of convergence can be seen in the graph, as the first term at k=0 is equal to 47/15, which is already very close to pi, thus the series need not iterate for as many terms. In addition the formula is a repeated negative exponent, multiplied by the subtraction many fractions, with a small constant numerator and variable denominator. This makes the series converge very rapidly, as a the negative exponent in this case is larger than that in the Madhava series. This series has the closest first term to π with the fastest convergence and thus is the most efficient.

The square root function is displayed in two above graphs, one showing the given Newton-Raphson method compared to the math.h square root function from 0 to 10, and one showing the term by term accuracy for finding the square root of 10. The first graph merely shows the relative accuracy of the two functions. The second graph shows the way the terms converge to the wanted result. This method of the next term being dependant on the previous, makes the function first overshoot the given constant, and then oscillate term by term to accuracy. The wildly different number of iterations if the constant is 0 or 1 can also be explained that way. When the constant is one, the first and second term are the same, as given in the above equation, and thus the method only iterates once, while for 0, since the first and second term are very close to each other, the function iterates 47 times, much more than any other number tested.

6 Acknowledgment

In this section, I would first like to acknowledge Professor Long for firstly, the necessary skills in coding in the C language to implement the above functions, and for the assignment document, which showed simplified forms of the above formulas, and tips for implementing them in C.

In addition, I would like to acknowledge Eugene Chou for the guide to usage of gnuplot in section, thus allowing me to present my findings in graphs.