Comparison of Pi/e Approximator with the Standard Math Library in C

Introduction

This writeup explores the comparison between approximation functions for the constants pi and e written in C, with the standard, general math library that offers these approximations on the fly.

When you think of pi, e, or any constant that continues on indefinitely, the approximate value of these numbers might come natural to you due to memorization, or from school. It is natural to think that an instrument as smart as a computer should also know the general approximation of a constant such as pi. But that is not the case. In a C program, when you call the Pi constant from the standard library $< math.h >$, it does not "know" what pi is. Instead, it will make its own estimation based on a calculated math function and return that to the user to then implement in their program.

We are curious to see if we can replicate this approximation to the highest accuracy, by using multiple pi/e mathematical functions **(these include Euler's Solution, The Madhava Series, The Bailey-Borwein-Plouffe Formula, Viète's Formula, and for e Euler's Taylor Series)**. In this experiment, we have written a C function for every one of these mathematical approximations, halted calculation at step epsilon value of around $1x10^{-14}$ and compared each of their returned values with the value set for pi and e in the standard math library. A short summary of the results, all approximations come within an astounding degree of accuracy compared to the library's constants.

(The mathematical operation $\sqrt{x}$ was also written in its own function to be used as needed in the approximation functions).

Materials and Methods

Listed below are each respectable mathematical series used for our study:

Euler's Taylor Series for e:

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

The Madhava Series:

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

Euler's Solution:

$$\sum_{k=1}^{\infty} \frac{1}{k^2} = H_{\infty}^{(2)} = \frac{\pi^2}{6}$$

The Bailey-Borwein-Plouffe Formula:

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

Viète's Formula:

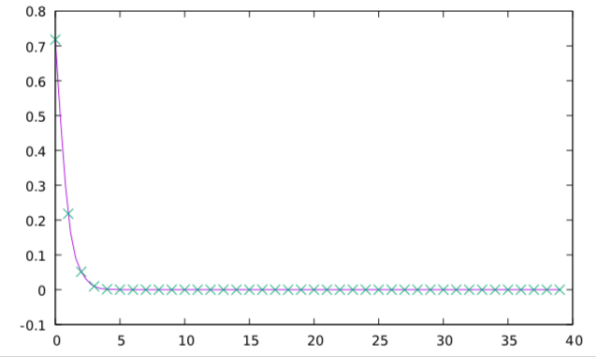
$$\frac{2}{\pi} = \prod_{k=1}^{\infty} \frac{a_i}{2}$$

An implementation of $\sqrt{x}$ and $|x|$ was also used to help our functions.

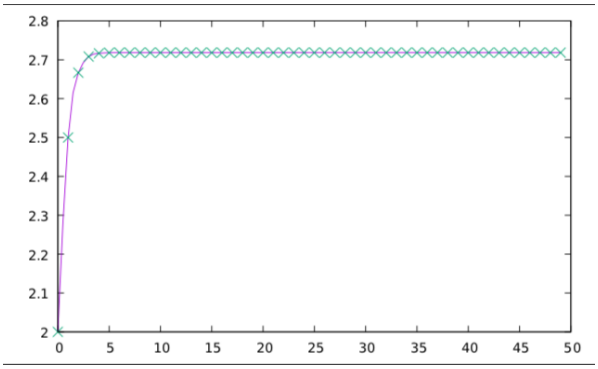
Results

e:

e Converging

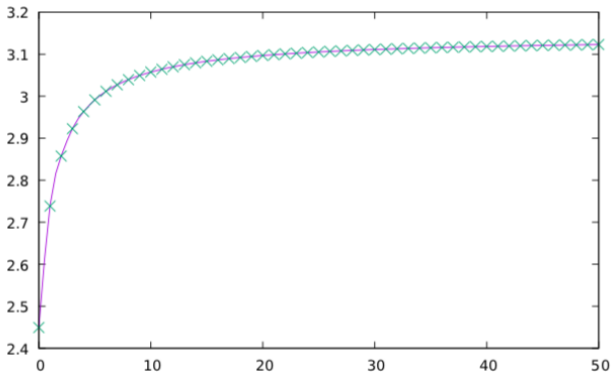


M_E - e difference

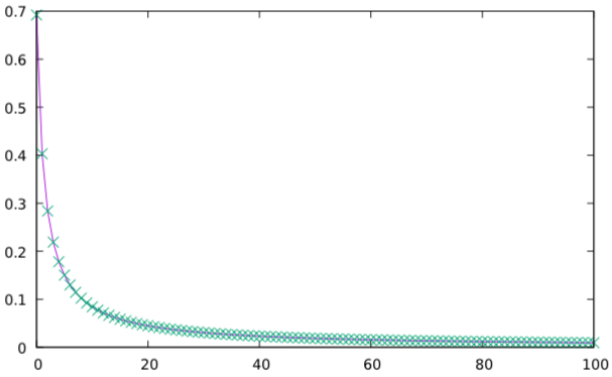


Euler's Solution:

Euler Converging

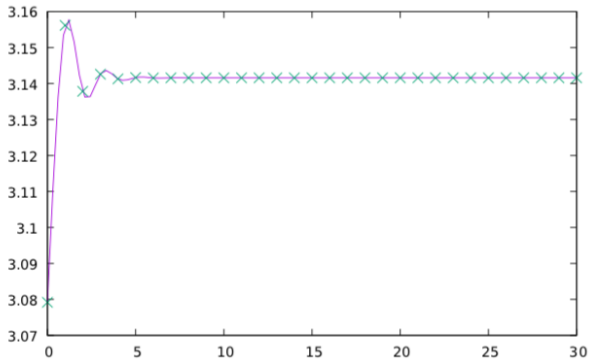


M_PI - Euler Difference

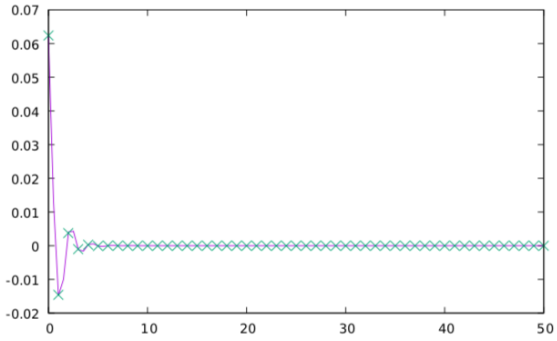


The Madhava Series

Madhava Converging

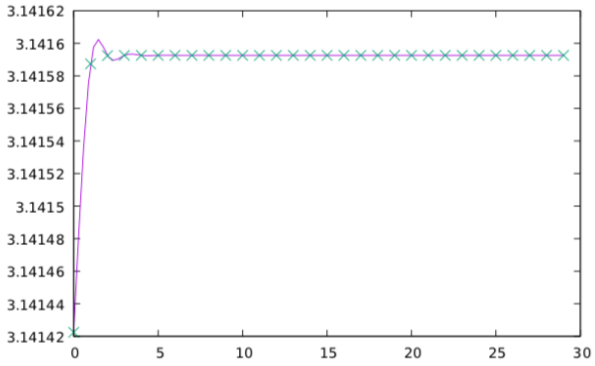


M_PI - Madhava Difference

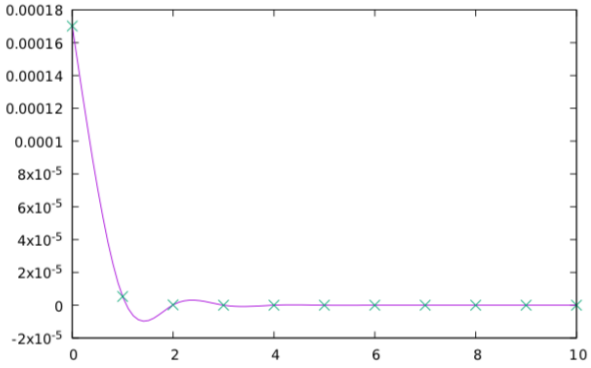


Bailey-Borwein-Plouffe Formula:

BBP Converging

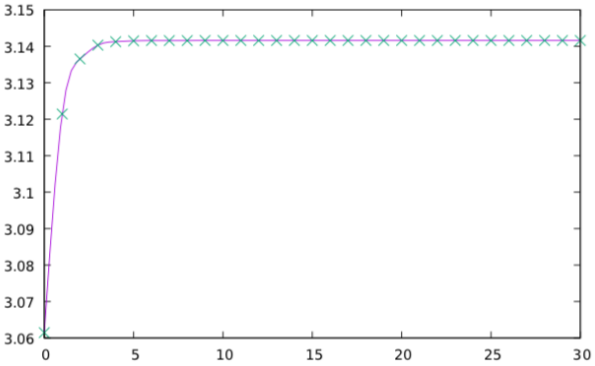


M_PI - BBP Difference

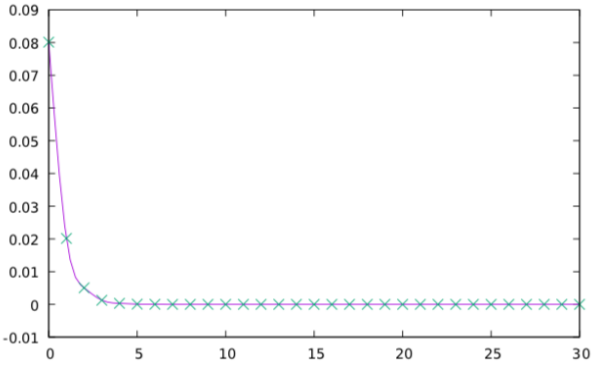


Viète's Formula:

Viète Converging



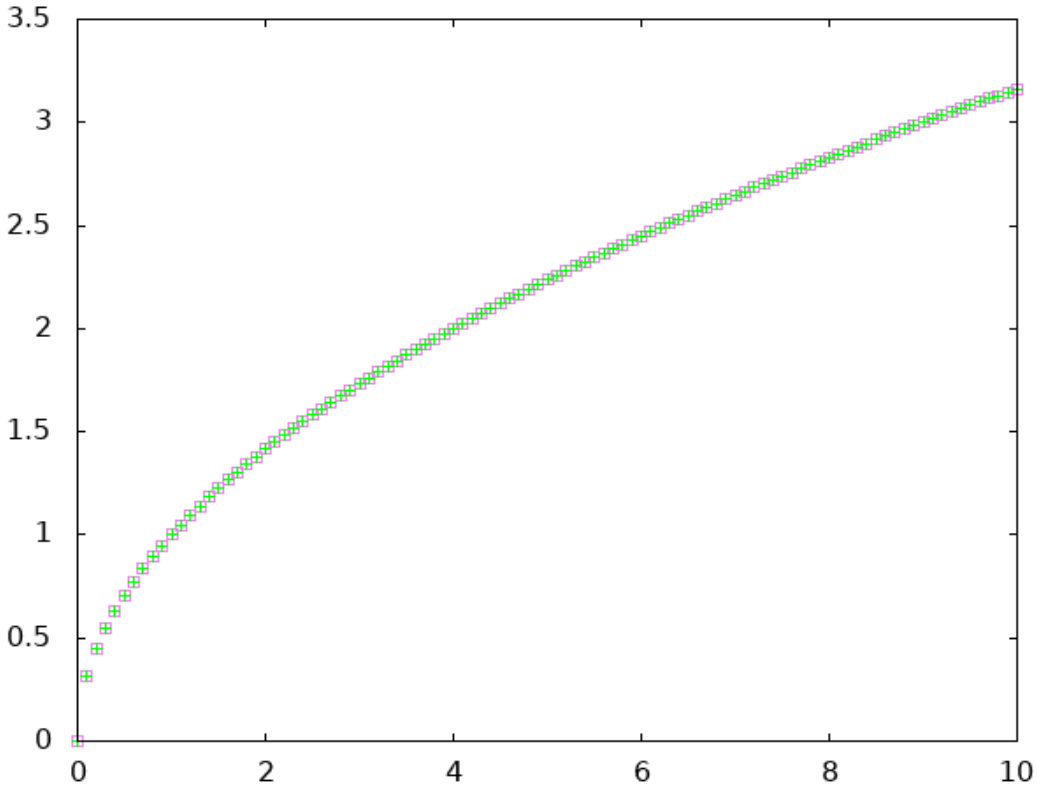
M_PI - Viète Difference



Legend (for all graphs above):
Y-axis: Sum of Series
X-axis: Number of Iterations (terms/factors)

Graphs on the left represent the function converging to their specified constant
Graphs on the right represent the difference between Math library constant with each sum iteration of the respectable function.

Sqrt_newton() vs Sqrt()



Legend:
X-axis: x-value to be squared from 0 to 10 in increments of 0.1
Y-axis: value of calculated operation

Sqrt(x) = violet square
Sqrt_newton(x) = green plus

FUNCTION	APPROXIMATION	MATH LIBRARY	DIFFERENCE
e()	2.718281828459043	2.718281828459045	0.000000000000002
euler()	3.141592558095893	3.141592653589793	0.000000095493900
madhava()	3.141592653589774	3.141592653589793	0.000000000000019
bbp()	3.141592653589791	3.141592653589793	0.000000000000002
viete()	3.141592653589789	3.141592653589793	0.000000000000004

Analysis

Let us take a look first at the graphs on the left, which represent each respective function converging to their constant after a set amount of iterations of their infinite series. An interesting point to be made is that all of these functions converge within decimals of their constant within around 10-30 iterations, which is good from this point of view, however in a program it is best to iterate until the individual step reaches a tiny threshold just in the name of accuracy. Viète's Formula, Euler's Solution, and e seem to be the only functions that converge in a shape similar to e^x minus the focus of the orientation, while BBP and Madhava have some variation before they converge.

The graphs to the right of the page represent the difference between the math library's constant value and the current function value at each iteration. This is calculated by M_PI/M_E - function(current step). As we would've guessed by looking at our converging graphs, the difference between these two also converges early, most of which reach around 0 between 2 and 10 steps. The only outlier here is our Euler's Solution, which takes around 100 iterations to be somewhat close to 0. This is expected, since our results from our program show that our Euler's Solution takes around 10 million steps to get to a step size smaller than EPSILON, and thus ending. However even with 10 million steps, Euler's solution is the least accurate out of all of the approximations, relative to the value of pi from the math library. This reason is still a bit unknown to me, however I can affirm that this slight inaccuracy most likely comes from the actual math formula rather than my code, since all of my functions are written in a similar way.

The last graph that is displayed is the sqrt_newton(x) vs sqrt(x) graph which shows a comparison between the approximation of our square root function and the math library's iteration. According to the graph, the points line up on top of each other approximately, which shows that our function is a correct representation of the square root math operation.

This fact about the total number of steps for Euler's solution did lead to an issue that was not perceivable at first. A close look at the function shows that the denominator of the series is k^2, which when you get into the million steps, can get pretty big. Once the calculation got to around 37 thousand steps, the type double in the code for our denominator was not able to hold it, thus overflowing and turning the sign bit negative, which would terminate the program due to our condition. The fix to this issue was instead of using a type double to store this obscenely huge number, the denominator would be of type uint_32t.