UNIVERSITY OF CALIFORNIA SANTA CRUZ

Analysis - A Little Slice of π

Student name: *Nicholas Reis*

Course: *CSE13S* – Professor: *Darrell Long*
Due date: *October 10th, 2021*

Introduction

In this write up I will be demonstrating my findings pertaining to writing functions that estimate the value of π, Euler's number e, and the square root of a number. All of my work was done in the C programming language, and there are six separate functions I used in total: e(), bbp(), madhava(), viete(), euler(), and newton(). The formulas for each are listed below along with a description of how each of them works.

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

(*a*)

This formula above is the Bailey-Borwein-Plouffe formula for approximating π. Essentially the solution is to sum up each term from k = 0 to any desired number by using the right hand side of the equation. Substitute the value of k into the equation, and the answer will be the given term for that specific iteration.

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

(*b*)

This formula above is the Madhava series that is used for approximating π. Essentially the solution is to sum up each term from k = 0 to any desired number by using the right hand side of the equation. Substitute the value of k into the equation, and the answer will be the given term for that specific iteration. After the desired value of k (iterations) is reached, the output of the summation must be multiplied by $\sqrt{12}$.

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

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

(*c*)

This formula above is the Viète formula for approximating π. This solution is a bit different from the ones above, as instead of a summation, you multiply each term together. Each term is the same as the previous one except a 2 is added to the

numerator and then the numerator is put in a square root. Once the desired value of k (iterations) is reached, make sure to set the approximation equal to the value of 2 divided by the output of the product. This is because the solution from the equation is not actually π, but instead 2 divided by π.

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

This formula above is Euler's solution for approximating π. Essentially the solution is to sum up each term from k = 0 to any desired number by adding the term 1 divided by k^2 to the total. This solution must then be multiplied by 6 and then put in a square root in order to get the approximation of π.

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

$$\frac{x^k}{k!} = \frac{x^{k-1}}{(k-1)!} \times \frac{x}{k}.$$

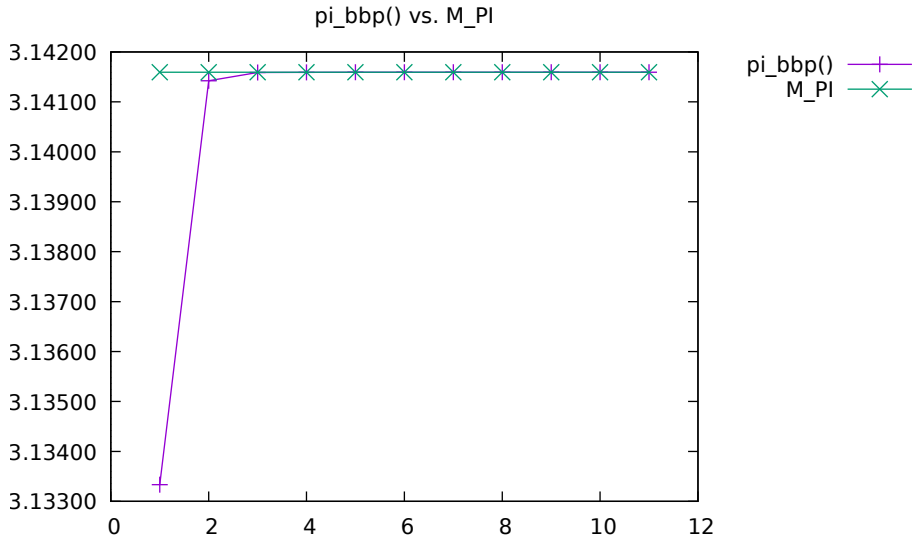
This formula above is the solution for approximating Euler's number e. Essentially the solution is to sum up each term from k = 0 to any desired number by using the series in the above equation. This series adds the term 1 divided by factorial k each iteration, and ends up becoming a very small number quickly. However, as opposed to using factorial, I simply used the bottom equation that states that the next term is the previous term multiplied by 1 divided by the current value of k. This means that I can circumvent using factorials and simply multiply in each term.

Results

Findings and Graph for bbp().

(*a*) This function utilizes the Bailey-Borwein-Plouffe formula to approximate the value of π. I found that this formula was the most efficient overall in calculating π to the first 15 decimal places, as it only took 11 iterations to do so. As you can see in the graph below, it only takes between 2 iterations to reach the point in which it is less than 0.001 away from the actual value of π, and it converges to the point at which you cannot see a difference from the two lines after only 3 iterations.

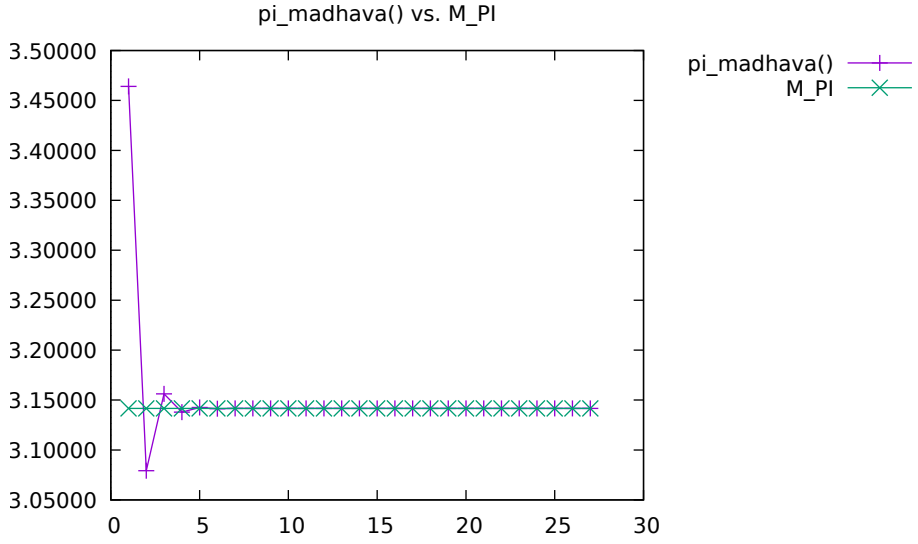
(*b*)



Findings and Graph for madhava().

(*a*) This function utilizes the Madhava series to approximate the value of π. I found that this formula was the third most efficient at calculating π to the first 15 decimal places, as it took 27 iterations to do so. As you can see in the graph below, this series acts quite differently from the one above, as instead of steadily increasing to converge at the value of π, it instead goes over the value, then below it, and repeats in this fashion until it gets closer and closer to the true value.

(*b*)

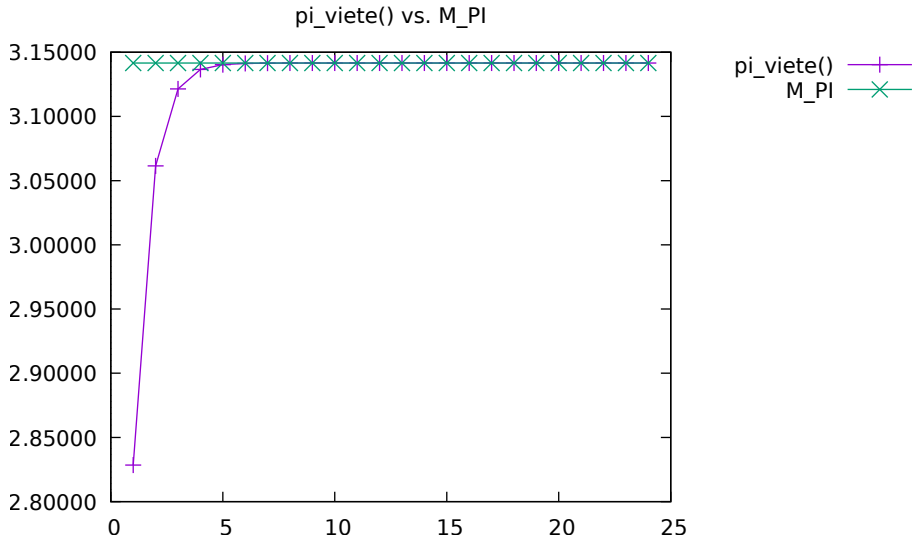


Findings and Graph for viete().

(*a*) This function utilizes Viète's formula to approximate the value of π. I found that this formula was the second most efficient at calculating π to the first 15 decimal

places, as it took 23 iterations to do so. As you can see in the graph below, this series looks similar to the graph of bbp.c above, but takes a few more terms (about 5) before it gets so close to the line that we cannot see much of a change afterwards. This makes sense, as it takes about twice as long to converge to the same value as bbp.c, so although it still converges quite rapidly, it is not as effective.

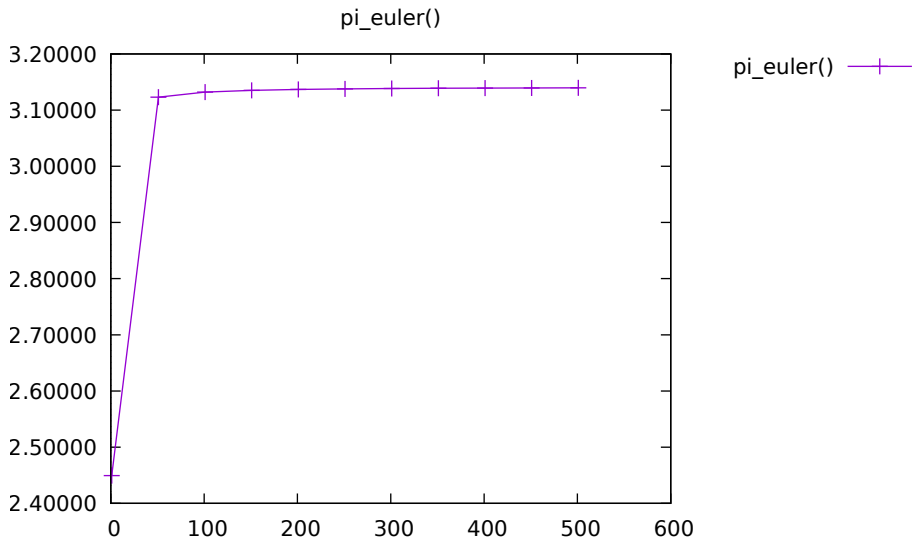
(*b*)



Findings and Graph for euler().

(*a*) Note that for this function and my e() function below I was not able to add the line for M_PI and M_E because my virtual machine installed Gnome for whatever reason and it prevented me from accessing my repository. I am working on fixing it, but I did not have the time to deal with that and finish this write up in time. Regardless, it is still quite obvious to see where the line converges, and you can assume that is where the math.h functions would lie on the graph.

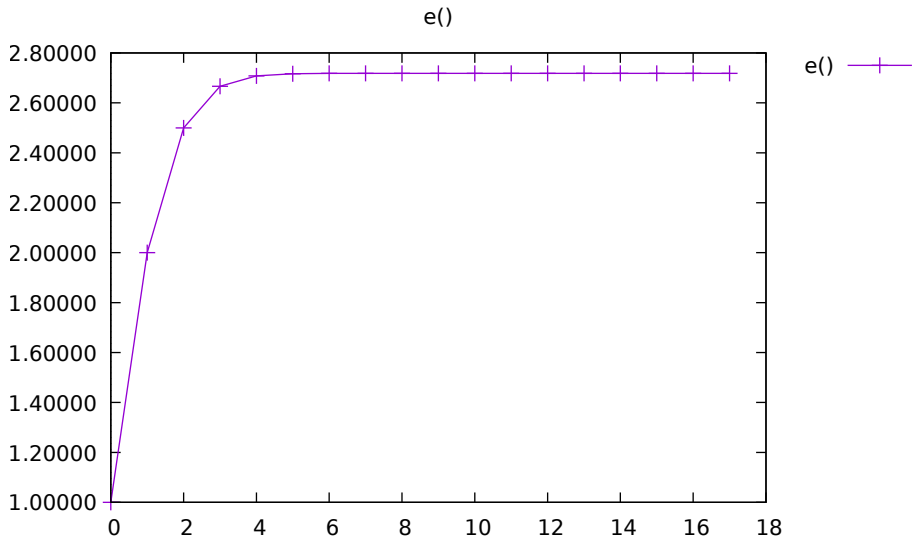
(*b*) This function utilizes Euler's solution for approximating the value of π. I found that this formula was the least efficient at caluculating π, as it could only reach an accuracy of 7 decimal places after it was run 10,000,000 times. As you can see in the graph below, the terms are still gradually increasing even after it is run 100 times, and although it technically converges to the value of π, I imagine it would take billions of iterations to reach the level of precision that the other functions could reach in less than 100 iterations.



(*c*)

Findings and Graph for e().

(*a*) This function utilizes the Taylor series to approximate the value of Euler's number e. I found that it takes about 5 or 6 iterations for the formula to converge very closely to the actual value of e, and gets within 15 decimals precision after only 18 iterations. As you can see in the graph below, it is quite an effective method in calculating e.

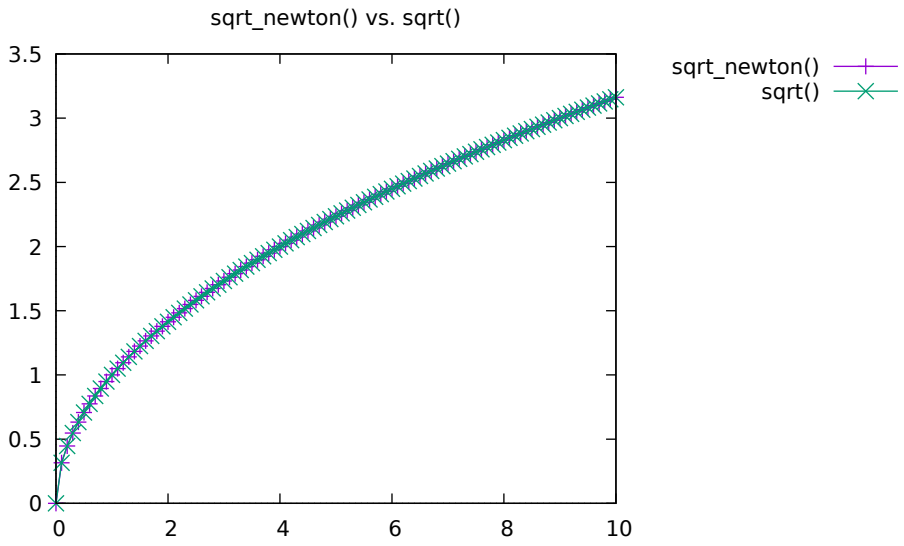


(*b*)

Findings and Graph for sqrt_newton().

(*a*) This function utilizes Newton's method to approximate the square root of a given number. I found that my implementation of the function almost exactly matches the values of the math.h built-in sqrt() function (within 15 decimal places). As you can see in the graph below, every number between 1 and 10 (incremented by 0.1) is in the same location as the math.h functions is, creating the exact same line.

(*b*)



Conculsion

Each of the functions for approximating π do so with great accuracy and efficiency apart from Euler's solution. Although Euler's solution gets the job done, it does so after millions of iterations - ultimately wasting time and your computer's resources to achieve an output with high precision. The other three offer great alternatives to the built-in value of M_PI from the math.h library, as they can each achieve high levels of precision with only a few iterations. The function e() also provides a good alternative for the value of M_E from the math.h library, as it is able to achieve high levels of precision in finding the value of Euler's number e within a small number of iterations. The sqrt_newton() function is almost identical to the built-in sqrt() function from math.h, and thus can be used as an alternative as well.