Write Up Analysis of π, E, and Newton's Square Root methods.

By Derfel Terciano

Calculations being analyzed

- Calculating Euler's number using the Taylor series.
- Calculating the square root of a number using Newton's method.
- Calculating π using the Madhava series.
- Calculating π using Euler's solution.
- Calculating π using the Bailey-Borwein-Plouffe Formula.
- Calculating π using the Viète's Formula.

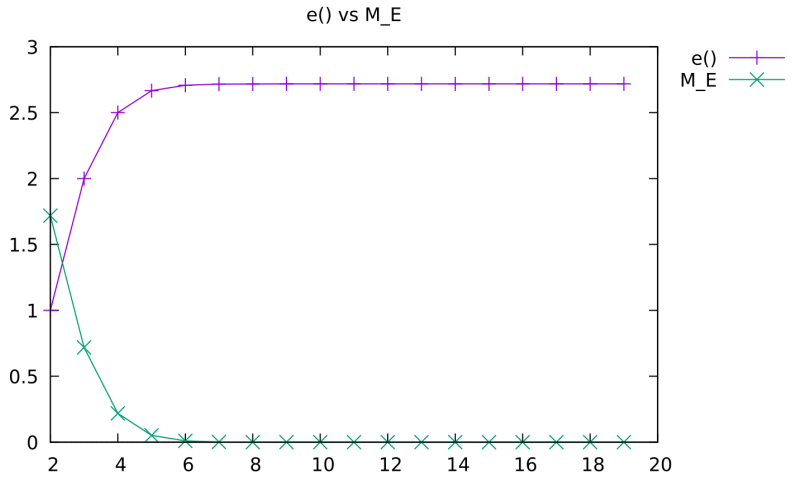
Euler's number using the Taylor series

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

Results from my program

In my program, we can see that the value form e() and M_E (from the math.h library) is completely the same except for that last digit. My program outputs a 6 while M_E outputs a 5. However, if we look at the difference, it says the difference between e() and M_E is 0. The reason why this is happening is due to floating point arithmetic and precision. Since floating point numbers are complicated and don't really represent numbers, there is going to be a lot of round off errors and precision errors in my final result.

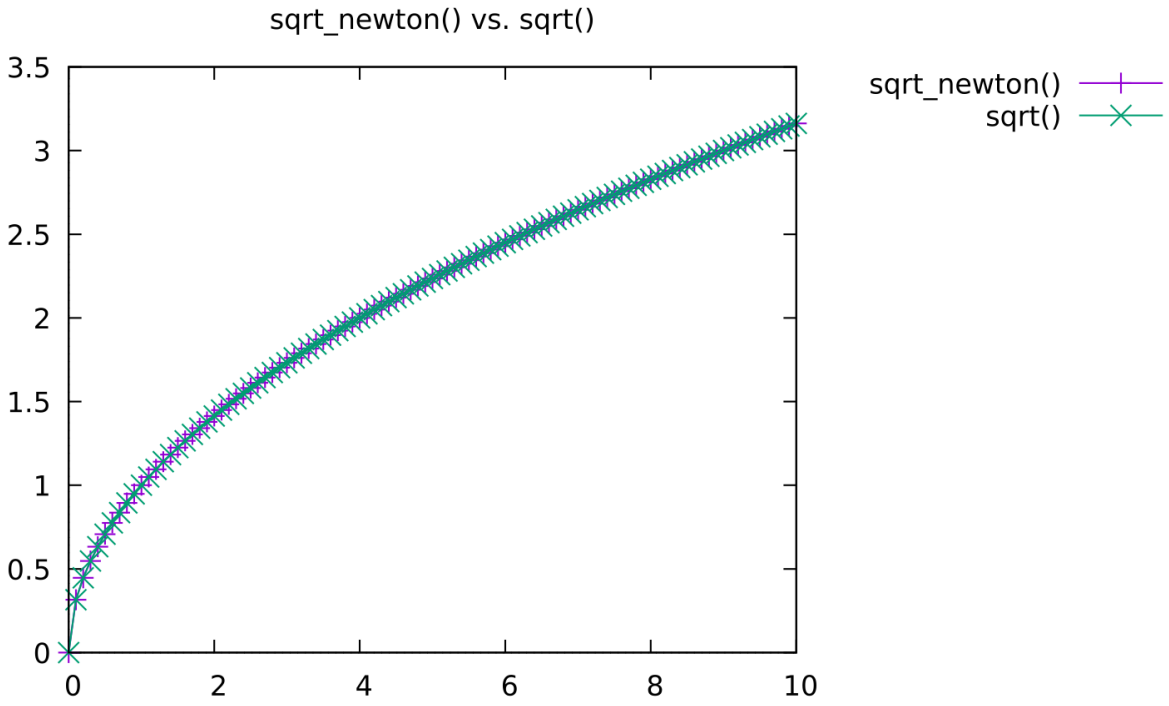
Though we have a floating point round off error, we can still take a look at how my program is calculating e().



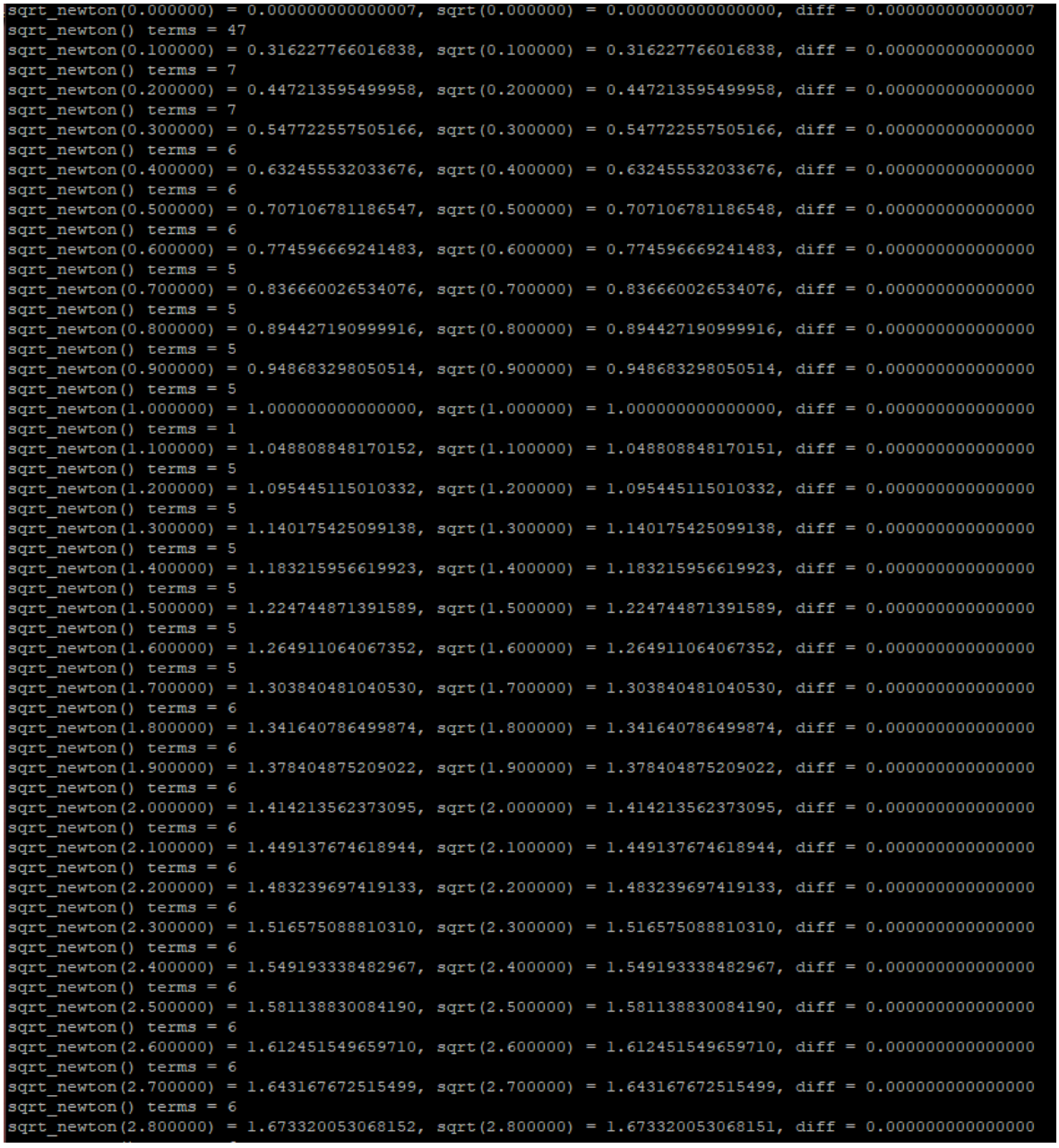
Here, the green line indicates the difference between e() and M_E. We see that as the number of terms increases, the closer we get to Euler's number. Based on the graph, my e() function is accurate enough to pass for Euler's number. As the green line hits zero, the more accurate e() (the purple line) becomes.

The Newton-Raphson approximation for square roots

Let's compare the results from sqrt_newton() and sqrt() using a graph. In this test, we calculated the square root of [0,10] in increments of 0.1.



Based on the graph, it looks like Newton's method is very accurate compared to the sqrt() function from the math.h library. However, if we look at some of the output results, we can see that we are also getting the same floating point errors as my e() function:



If we look especially at sqrt_newton(0), we can see that my algorithm has a problem with this calculation. For some reason, my algorithm is using 47 iterations to find out what sqrt(0) is, which is pretty weird. This is because Newton's method fails when the derivative is 0.

Remember that Newton's iterative is define as:

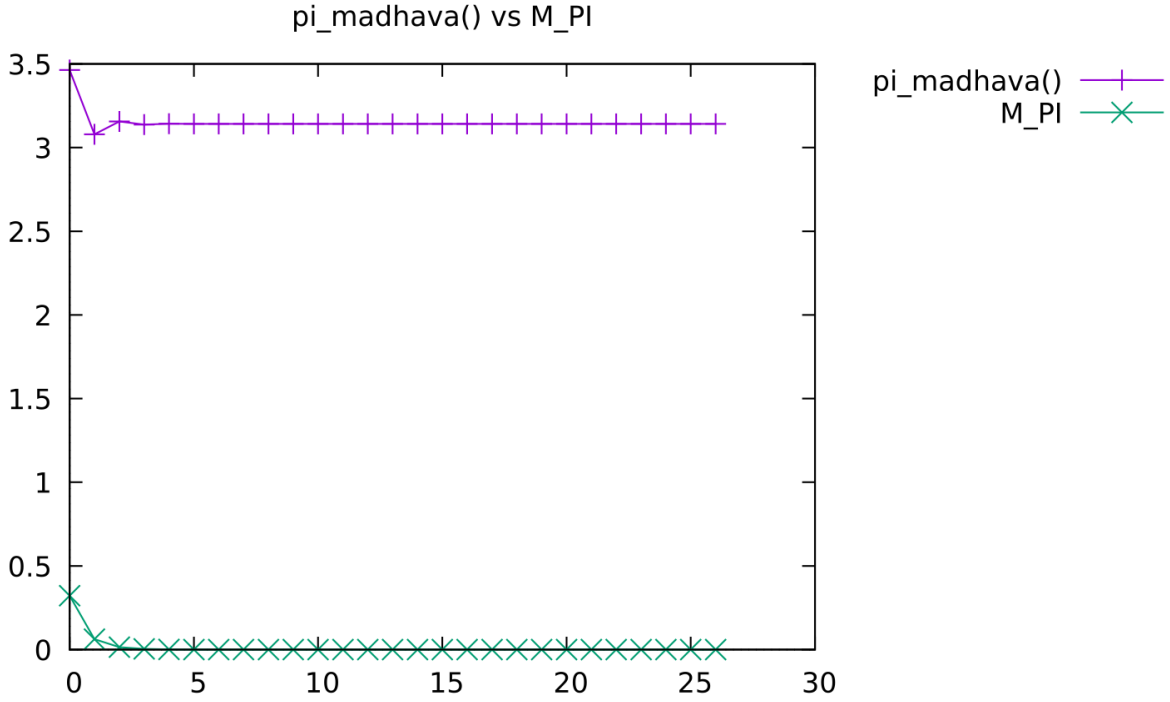
$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}.$$

When the derivative is 0, then we would get a vertical line which is impossible to calculate since there would be a zero in the denominator. That is why sqrt_newton(0) fails.

In addition, we can also see that sqrt_newton(2.8) also has the same floating point error as when we calculated e().

Calculating π using the Madhava series

Let's now compare pi_nadhava() and M_PI (from the math.h library). For my own calculations below, we can see that as more and more terms are calculated, the closer we get to π. This is due to the fact that as more terms are calculated and added, the closer the series converges to π.



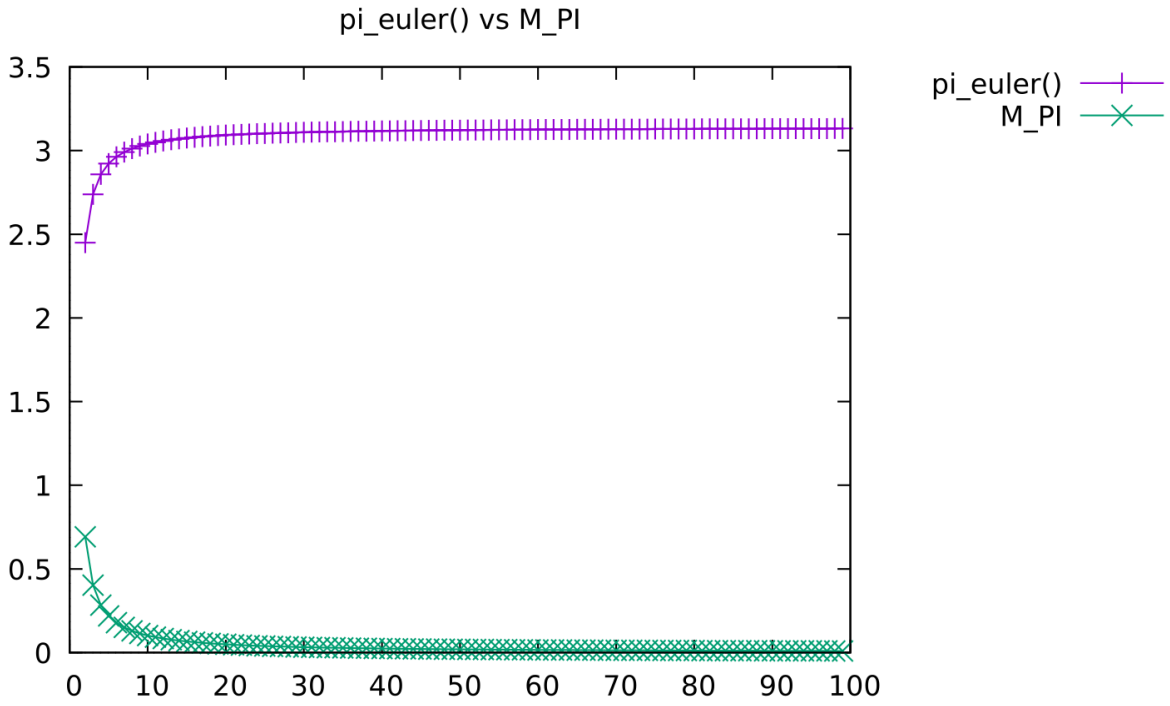
One interesting thing that we can see from the purple line of this graph is how in the second iteration of the calculation the term dips below the actual value of π. Then in the next iteration, the term rises above the actual value of π. The reason why it does is due to the fact that the denominator of madhava series is $(-3)^{-k}$. This part of the denominator switch +/- signs depending on the value of k. If k is an even number, then the denominator becomes positive, if it is an odd number, then the denominator becomes negative. Lastly, let's take a look at my output result of the program:

```
pi_madhava() = 3.141592653589800, M_PI = 3.141592653589793, diff = 0.000000000000007
pi_madhava() terms = 27
```

As we can see, I have a slight difference in my program compared to the M_PI value. This is due to the fact that I calculated less terms than needed. If I had a value of epsilon that was much smaller than what was specified in the assignment, then I would have been able to almost reach a difference of zero. In addition, we also need to keep in mind that float precision in C is recommended to be less than 15 decimal places. Lastly, keep in mind that arithmetic that involves floats can have rounding errors and precision errors as well.

Calculating π using Euler's solution.

Let's compare the result for pi_euler() and M_PI. In my calculations, I only graphed only the first 100 terms. Overall, my code calculated 10,000,000 terms which I think is a very inefficient way to calculate π.



From my calculations, we can see that Euler's solution takes a long time to reach π. Let's take a look at my program output:

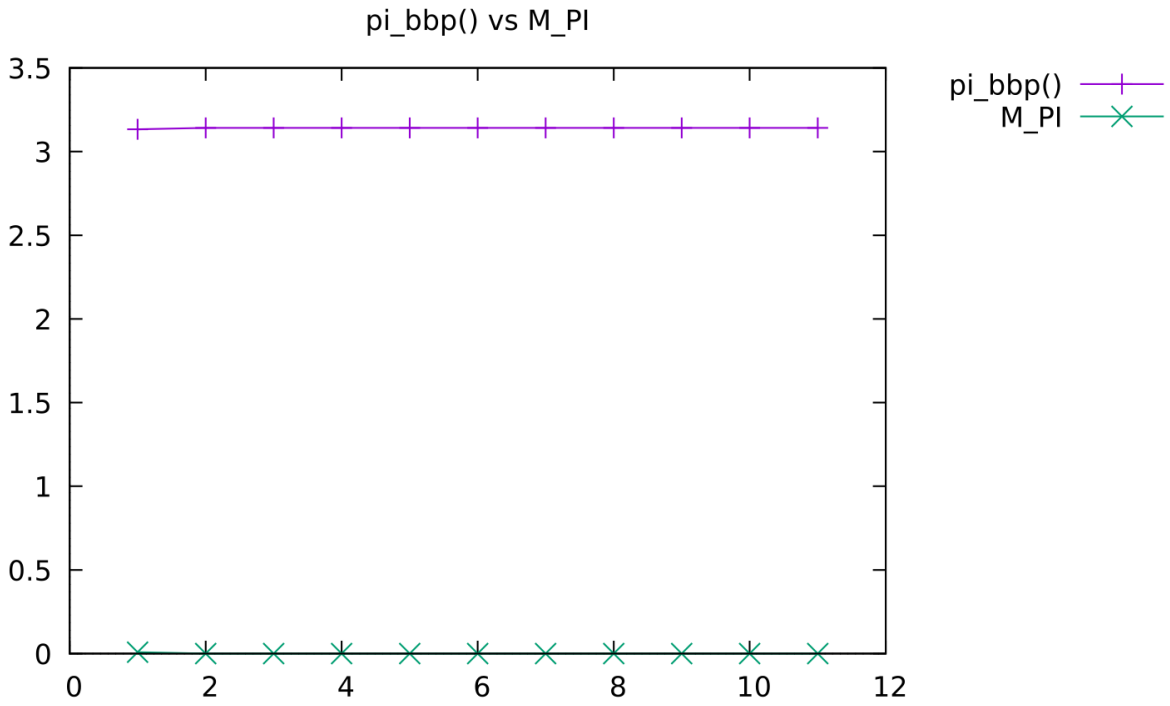
```
pi_euler() = 3.141592558095903, M_PI = 3.141592653589793, diff = 0.000000095493891
pi_euler() terms = 10000000
```

As you can see, we end up with a big difference even though we calculated only 10 million terms before we hit epsilon. In order for this calculation to be very accurate, we would need to calculate past 10 million terms and past the floating point precision in order to get a difference of 0. The reason why this is happening is due to the fact that the nature of Euler's solution takes a long time to

converge. In addition, the formula for Euler's solution does not involve any mathematical tools that can help grow the equation fast. Some tools I am referring to are exponentials, factorials, and such.

Calculating π using the Bailey-Borwein-Plouffe Formula

Let's now compare pi_bbp() to M_PI. From the graph below, we can see that even the first term gets us pretty close to π.



Using the pi_bbp formula, we can see that we reach the epsilon term very quickly. This occurs due to the fact that the main term:

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

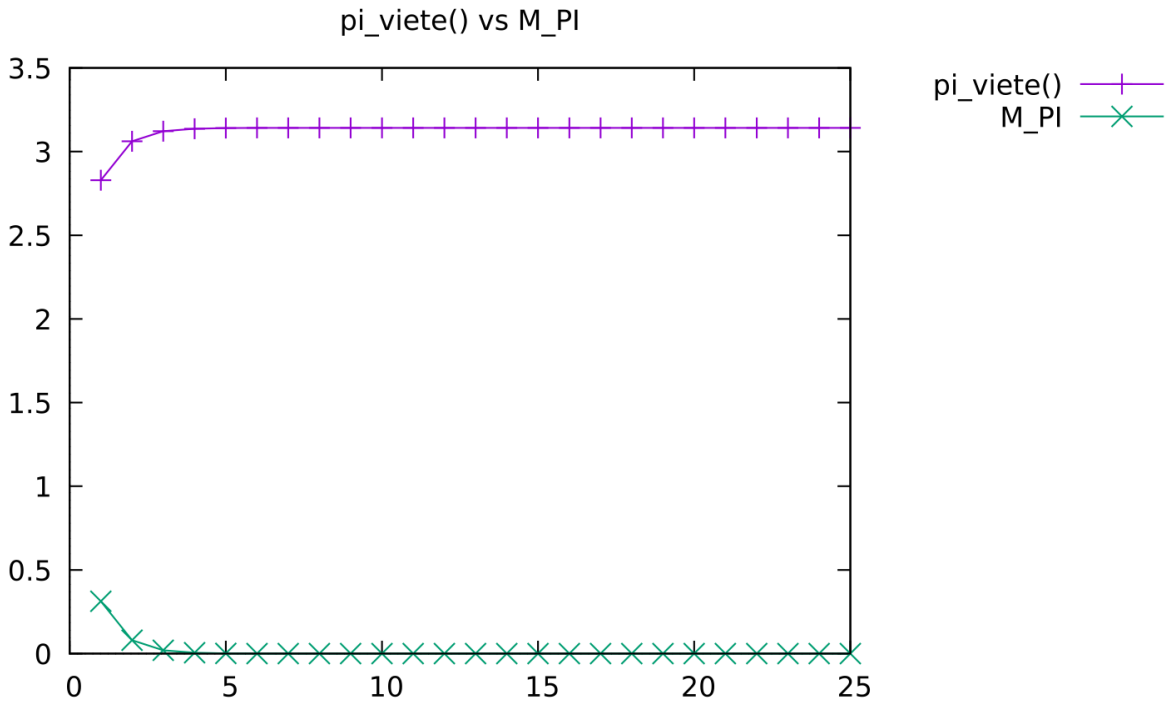
already gets us close to π. As k increases in this equation, the more accurate the formula becomes. Let's take a look at my program output:

```
pi_bbp() = 3.141592653589793, M_PI = 3.141592653589793, diff = 0.000000000000000
pi_bbp() terms = 11
```

As we can see, my calculation matches exactly with M_PI. However, keep in mind that there may be some floating point round off errors and precision errors.

Calculating π using the Viète's Formula

Let's compare my pi_viete() to M_PI. In this calculation, my code has computed an extra two terms instead of 23 terms from the reference program.



As we can see from the graph, the formula goes to π relatively quickly. This is due to the fact that the formula relies on the previous term in order to make the formula work.

```
pi_viete() = 3.1415926535897927, M_PI = 3.141592653589793, diff = 0.000000000000000
pi_viete() terms = 25
```

Looking at my program's output, we can see that there should be a difference between the M_PI and my calculation. However, due to floating point errors, C says that there is no difference between the 2 values. Lastly, the reason my program calculated 2 extra terms is due to the floating point errors. If my program didn't have any floating point errors then the program would calculate the same amount of terms as the reference program. However in this case, it is better to have more terms so that we can have an accurate calculation of pi.

Conclusion

In conclusion, my calculations overall are very accurate to $\boldsymbol{\pi}$, and e. In addition, the sqrt_newton() function is very accurate when compared to the sqrt() function in the math.h library.