Analyzing Methods of Approximating π

Lucais Sanderson

29 January 2023

1 Intro

The keystone program, `mathlib-test.c`, accepts multiple inputs from the user when called. Depending on the options flagged, it will run functions approximating certain irrational constants and/or a square root function calculating various decimal values. Additionally, if specified, the program will print the number of terms/iterations done until it reached a constant precision.

The methods used to calculate π are Madhava's series, Euler's, Bailey-Borwein-Plouffe, and Viete's method.

2 Bernoulli's e Method

The first and only method (within this assignment) that will be approximating e is the one discovered by Jacob Bernoulli. Bernoulli discovered that by taking the limit, $\lim_{n\to\infty}(1+\frac{1}{n})^n$, he could find e. The limit can be described as a convenient sum

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

The factorial term in the expression grows very fast so we can assume the sum approaches e pretty quick as well. (This is reflected in the results shown later.)

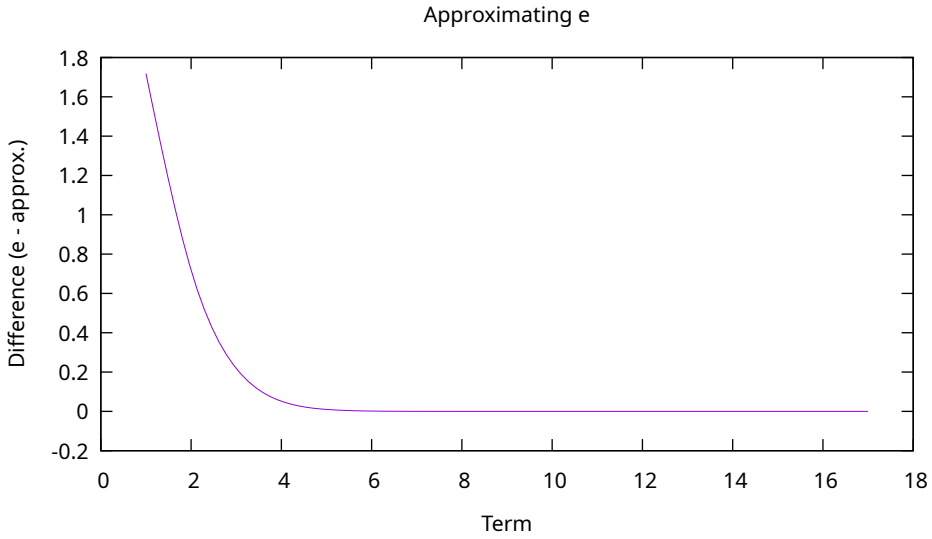
Since the terms grow very fast we needed a way to reduce the number of times we perform multiplications from the factorial term. Due to how factorials are defined we can use the fact that

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

As noted in the spec document, we could use a recursive function to implement this but it turns out to not be necessary as saving the previous term in a variable and using it to compute the next term on the next loop (in a `for` loop for instance) works fine.

RESULTS

Graph of the current value the function is at versus the current term ($i = 1, i = 2, ..., i = n$).



By term 6 the value is indiscernible from the "true" value of e on the graph. From that we can see that Bernoulli's method is a pretty efficient algorithm. Unfortunately, we don't have any other e approximation methods we can compare to. However, the algorithm ends at the 17th term with a value of 2.718281828459046.

For transparency, I'm using 2.718281828459045 for e on the graph. This is the same precision used for the primary `mathlib-test.c` file.

Overall, this algorithm is predictable and these results are expected.

3 Madhava Method

The Madhava method uses a series to approximate π.

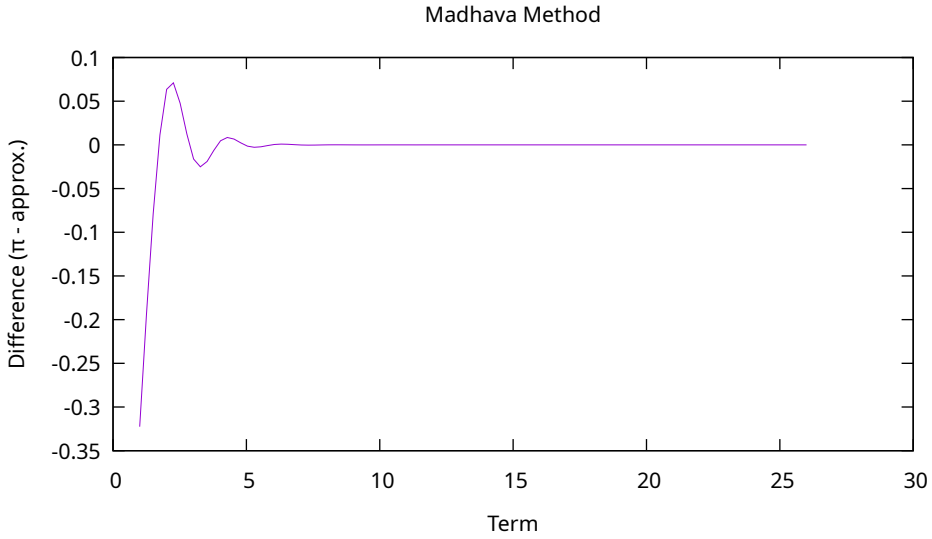
$$\sum_{k=0}^{\infty} \frac{(-3)^{-k}}{2k+1} = \sqrt{3}\tan^{-1}\frac{1}{\sqrt{3}} = \frac{\pi}{\sqrt{12}} \tag{1}$$

The $(-3)^{-k}$ term in the equation makes it an alternating series which is pretty interesting and something we should look for in our analysis.

To adapt the sum to a more program-friendly form, we can alter the bounds so that

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

where $p(n)$ is the π approximation for a given number of terms n. **RESULTS**



As expected, we get a function that oscillates around zero for the first 5 terms. After that, the function quickly converges to 0, meaning the value gets very close to π. In relation to the other methods, this one is around middle of the pack; it uses the most iterations than any other but its difference is only 7×10^{-15}.

4 Euler's Method

Euler discovered the solution to the Basel problem, a precise summation of the reciprocals of the squares of the natural numbers

$$\sum_{k=1}^{\infty} \frac{1}{k^2} = \frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \cdots$$

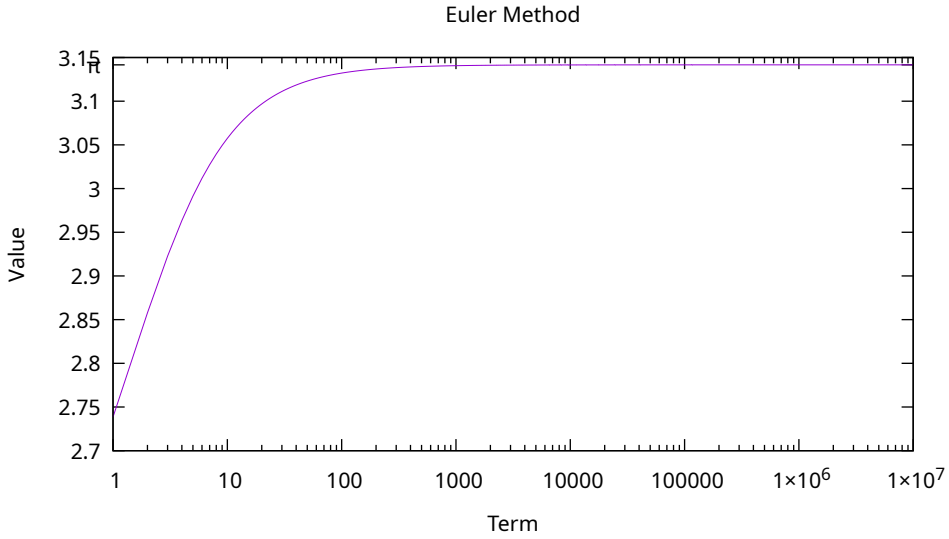
Euler's solution to this problem was $\pi^2/6$. Clearly, from here we can derive π

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

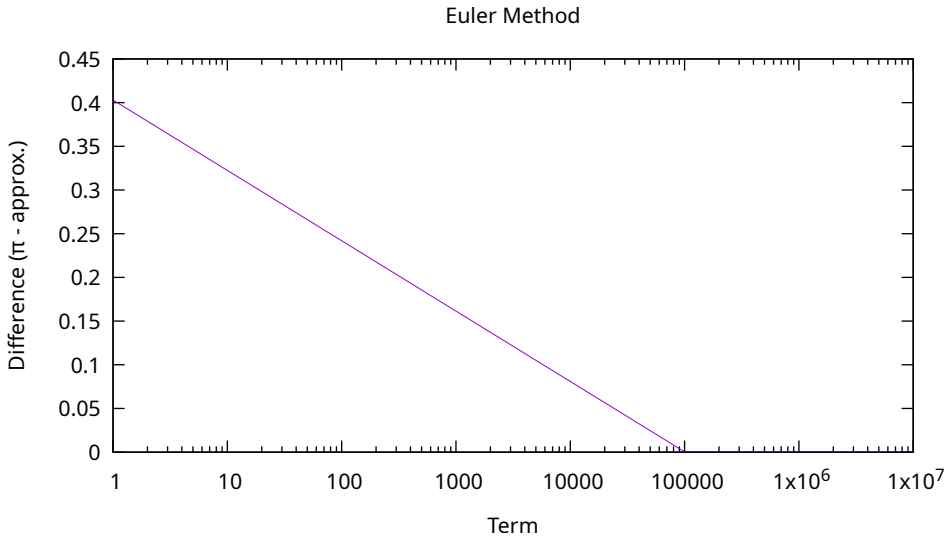
$p(n)$ being the approximation to π.

RESULTS

This graph expresses the actual value at a given term.



In contrast, this graph compares the difference between π and the approximation, and the term.



As it turns out, Euler's method is pretty poor in terms of accuracy relative to all the other methods. Additionally, this algorithm is not efficient, in fact it's very slow. It takes 10 Million terms/iterations to arrive at a term less than a given threshold (ϵ). Even then, the final sum it arrives at is $p(10,000,000) = 3.141592558095903$ which leaves a difference of 9.5493891×10^{-8} with respect to the *pi* value given in `C`'s `math.h`. This is the highest difference than any of the other *pi* methods. A logscale on the x-axis was necessary to effectively demonstrate the amount of terms used.

The second graph detailing the error from π shows a linear relation. This is especially interesting since we'd expect a shape resembling the previous graph. This might be a bug generating the graph but I can't find it if there is.

5 Bailey-Borwein-Plouffe Method

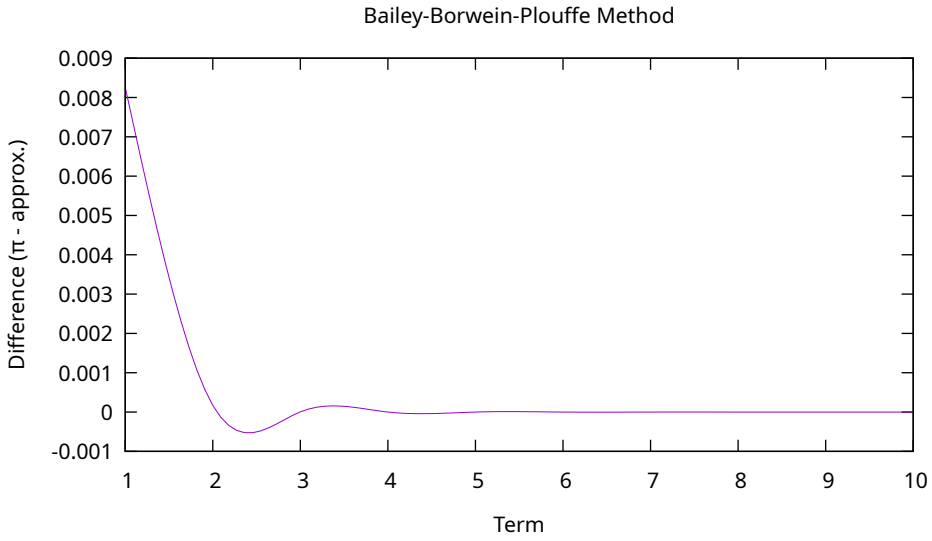
The Bailey-Borwein-Plouffe formula for π was discovered by Simon Plouffe and the article published was co-authored by Peter Borwein and David Bailey. Hence, the Bailey-Borwein-Plouffe (BBP for short) formula. It is given as

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

As pointed out in the spec document, the formula can be reduced to include less multiplications

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

I opted for this version in the final function. **RESULTS**
The difference between π and our approximation.



The function starts off immediately (relatively) close to its target. This tracks since when $n = 0$ the sum turns out to be $\frac{47}{15}$ which is around 3.13333...

In terms of how this method stacks up with the others, it is both the most efficient in how many terms it computes and is the most accurate, The final sum we get after 10 terms is 3.141592653589793. The difference is essentially 0 from the `M_PI` constant with 15 points of precision.

6 Viète's Method

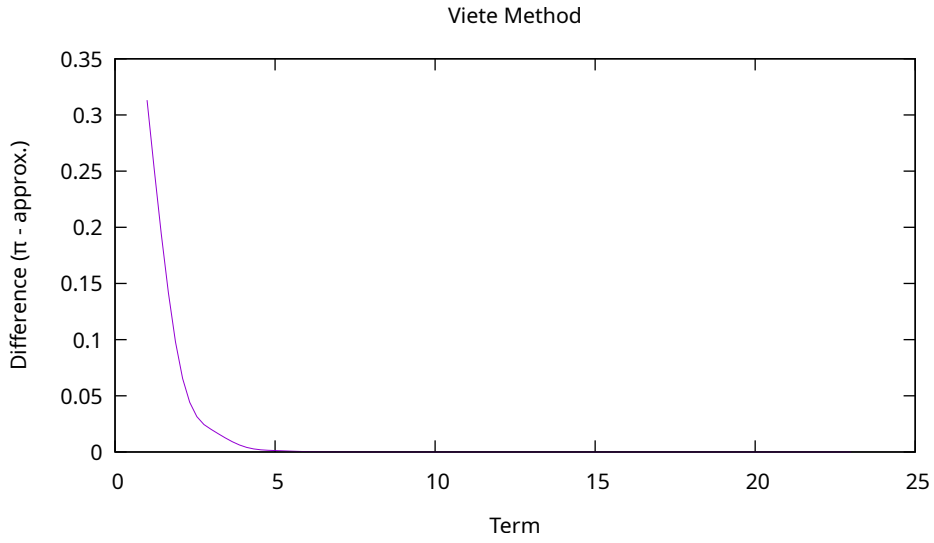
Unique from the other methods, François Viète's formula utilizes infinite products of nested radicals (so cool!). It is written as

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

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

RESULTS

This graph describes the relation between the current term and the difference from the estimate and π.



Viete starts off with pretty poor error, but quickly converges to π. By term 5 the difference is (visually W.R.T. the graph) zero. After 25 terms, the function stops and the final sum returns 3.141592653589789 with an error of 4×10^{-15} W.R.T. `M_PI`. What's interesting is that this error is actually *less* than that of the sample `mathlib-test` file which returns an error of 1.8×10^{-14} for this function. I believe this is due to my function computing one more term than the one in `resources` because of how the looping condition is defined. I compared the `current_term` to ϵ directly and there may be some difference in that implementation that causes this change.

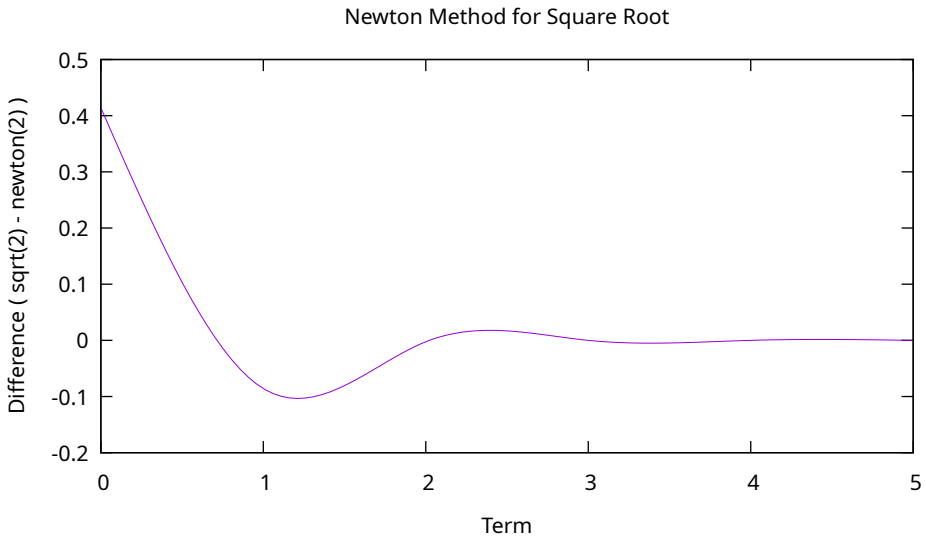
7 Newton's Method to Calculate Square Roots

In a couple of the previously described functions, a square root method is required. A convenient one to implement is the Newton-Raphson method (more simply Newton's method). We compute the inverse of x^2 iteratively with

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

RESULTS

Plots the difference between `<math.h>`*'s* `sqrt(2)` *and our* `sqrt_newton(2)` *function.*

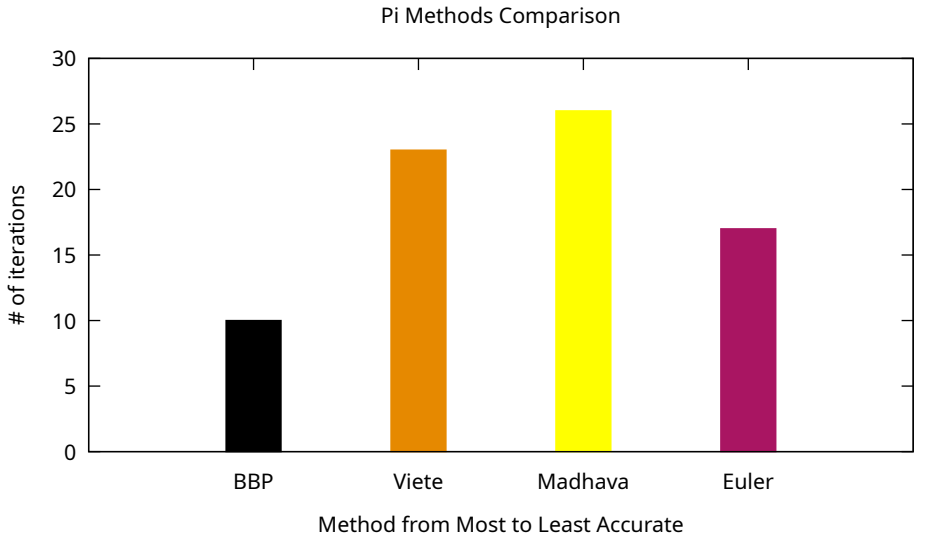


The function shortly oscillates around zero and after 5 terms we arrive at 1.414213562373095 with effectively zero difference between that and `<math.h>`'s $\sqrt{2}$ with 15 points of precision.

8 Wrap-Up

Comparing π Methods

The 4 π methods we analyzed were Bailey-Bowein-Plouffe, Viete, Madhava, and Euler. Below shows their final number of iterations before the respective function terminates, ordered from the most accurate to least accurate.



Overall, as explained earlier, the BBP method is both the most efficient regarding the amount of terms computed *and* is the most accurate to π. Regarding Viete and Madhava, I'd say from *my* results, Viete is overall the better algorithm. However, if we're considering the sample function used from `resources`, the error on Viete is greater and the two methods would be more competitive

with eachother. Euler's method is by far the least efficient and uses the most terms to achieve it.

Last Thoughts

I come away from this assignment understanding how to implement rudimentary arithmetic operations such as exponents and factorials, interpreting algorithms in `C`, and analyzing results from those implementations. Additionally, I'm more comfortable with `Make` files and implementing header files.

Appendices

- ϵ
  - In the context of this assignment, ϵ (Epsilon) is a constant 1×10^{-14}. This is used to limit the number of iterations a given function goes through. In general, the loop ends when the current term is less than ϵ.