Approximations of Mathematical Constants

Alexander Ragland

January 29, 2023

1 Introduction

Often we take for granted the mathematical libraries available to us when programming and never stop to think about how these mathematical libraries work, or possible alternatives. Two mathematical constants often provided by libraries such as these are e, Euler's number and π. These mathematical constants are irrational, yet there are several formulas we can use to approximate their values. Depending on the formula, efficiency varies; one formula might take 10 iterations in order to approximate π within an ϵ precision, and another formula might take 1000 to do so using the same precision. In order to test which formulas are the most efficient in their calculations, we implemented several variations of these formulas and kept track of how many iterations it took to approximate a mathematical constant with an ϵ of 10e-14.

2 Mathlib-test.c

In order to test our functions, we needed a test harness. So, we implemented one that takes a specified list of arguments and sets a corresponding flag to True. After enabling the correct flags, a series of conditionals check for a corresponding flag and carry out the proper functions.

```
    #define OPTIONS "aebmrvnshg"

int main(int argc, char **argv) {
    int opt = 0;
    // flags to enable functions
    bool stats = false;
    bool all = false;
    bool help = false;
    bool ebool = false;
    bool bbpbool = false;
    bool madhavabool = false;
    bool eulerbool = false;
    bool vietebool = false;
    bool newtonbool = false;
    bool graph = false;
    while ((opt = getopt(argc, argv, OPTIONS)) != -1) {

        // argument cases set function flags
```

```
        switch (opt) {
        case 's': stats = true; break;
        case 'a': all = true; break;
        case 'e': ebool = true; break;
        case 'b': bbpbool = true; break;
        case 'm': madhavabool = true; break;
        case 'r': eulerbool = true; break;
        case 'v': vietebool = true; break;
        case 'n': newtonbool = true; break;
        case 'h': help = true; break;
        case 'g': graph = true; break;
        }
    }
```

In the below example, the if statement checks if the "e" argument has been passed and if it has, then it prints the approximation of *e*, the <math.h> library's approximation of *e*, and the difference between the two. It also checks if the "s" argument has been passed and if it has, then it also prints the number of iterations it took to calculate.

```
        if (ebool == true) {
        double e_copy = e(graph);
printf(
"e() = %16.15lf, M_E =  %16.15lf, diff =  %16.15lf\n",
e_copy, M_E, fabs(e_copy - M_E));
        if (stats == true) {
            printf("e() terms = %d\n", e_terms());
        }
    }
```

Similar logic is utilized for the rest of the available functions, with only slight variations. For example, as seen below, the *sqrt_newton()* function is tested by comparing it's approximation the <math.h> approximation on values in the range [0:10).

```
    if (newtonbool == true) {
    for (double nti = 0; nti < 10; nti += .1) {
        double app = sqrt_newton(nti);
        double lapp = sqrt(nti);
        if (graph == true) {
            printf("%16.15lf\t%16.15lf\t%7.6lf\n", lapp, app, nti);
        } else {
        printf(
        "sqrt_newton(%7.6lf) = %16.15lf,
        sqrt(%7.6lf) =  %16.15lf, diff =  %16.15lf\n",
        nti, app, nti, lapp, fabs(lapp - app));
        }
        if (stats == true) {
```

```
            printf("sqrt_newton() terms = %d\n", sqrt_newton_iters());
        }
    }
}
```

3 *Function*.c

Many of the functions are implemented using while or for loops. As an example, Viète formula's function is shown below.

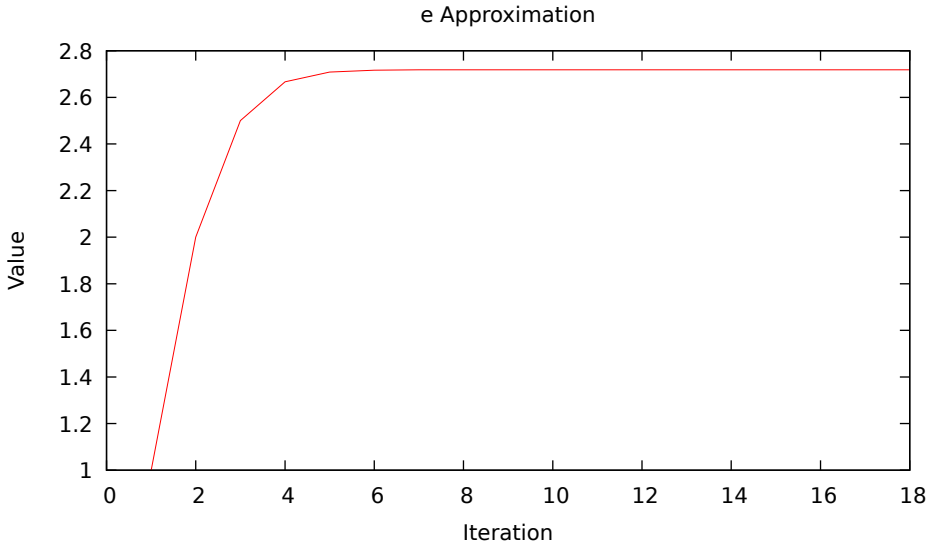
```
#include "mathlib.h"

#include <stdio.h>
#include <stdlib.h>

static int viete_i = 0;

double pi_viete(bool graph) {
    double root = sqrt_newton(2);
    double base = 2;
    while ((2 / root) > 1) {
        base = base * (2 / root);
        root = sqrt_newton(2 + root);
        viete_i++;
        if (graph == true) {
            printf("%16.15lf\t%d\n", base, viete_i);
        }
    }
    return base;
}

int pi_viete_terms(void) {
    return viete_i;
}
```

The value $\sqrt{2}$ is assigned to "root", and *2* is assigned to "base". Each loop iteration calculates the needed terms, then updates the iteration counter *viete_i*.

4 Results

GNUplot was used via a BASH script in order to plot the functions, as well as export those plots as PDF files.

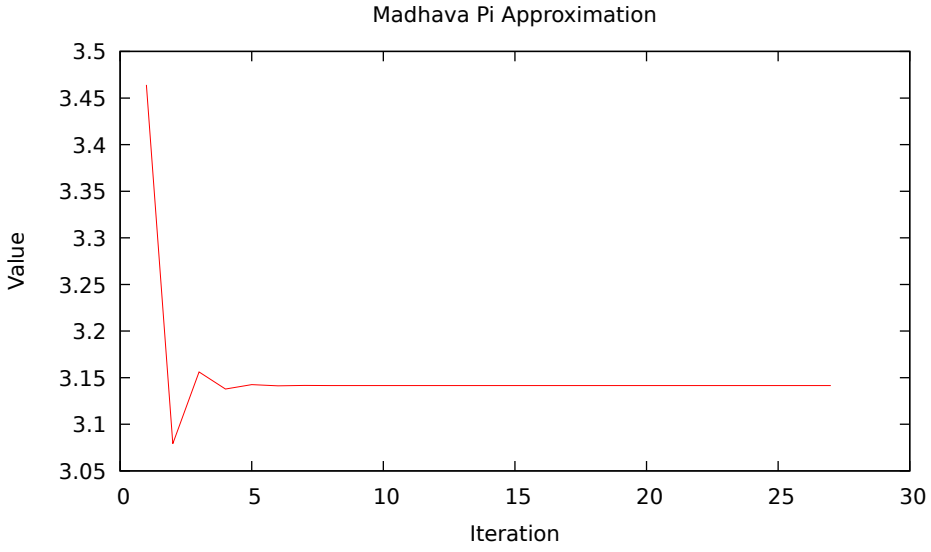
4.1 Euler's Number (*e*)



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

This formula for Euler's number is very efficient. As you can see, by the 6th iteration the approximation is already extremely precise. In the range [0:4], the approximation improves greatly, then tapers off.

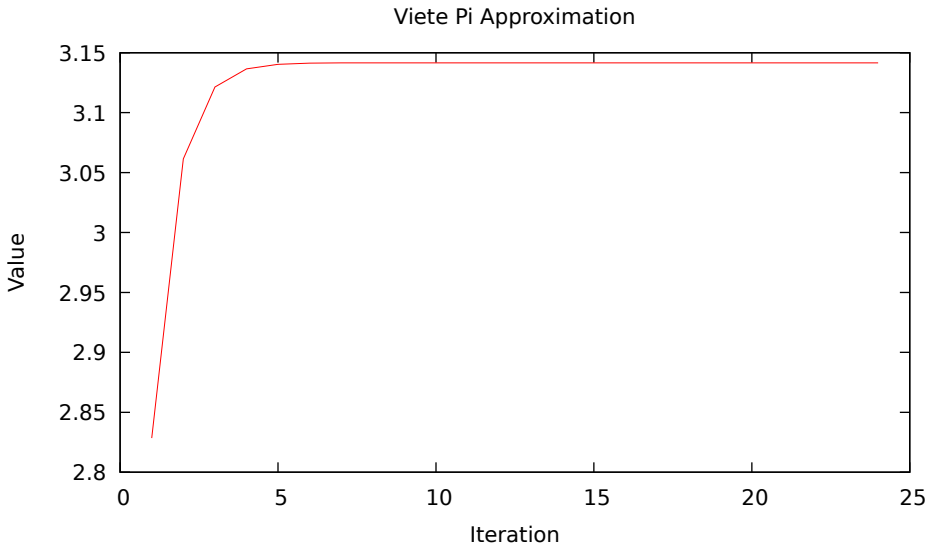
4.2 Madhava Series (π)



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

This formula for π is very efficient. By the 5th iteration, the approximation is very close the the value of pi. Interestingly, the sequence begins too large, then over-shoots and is too small, and finally evens out.

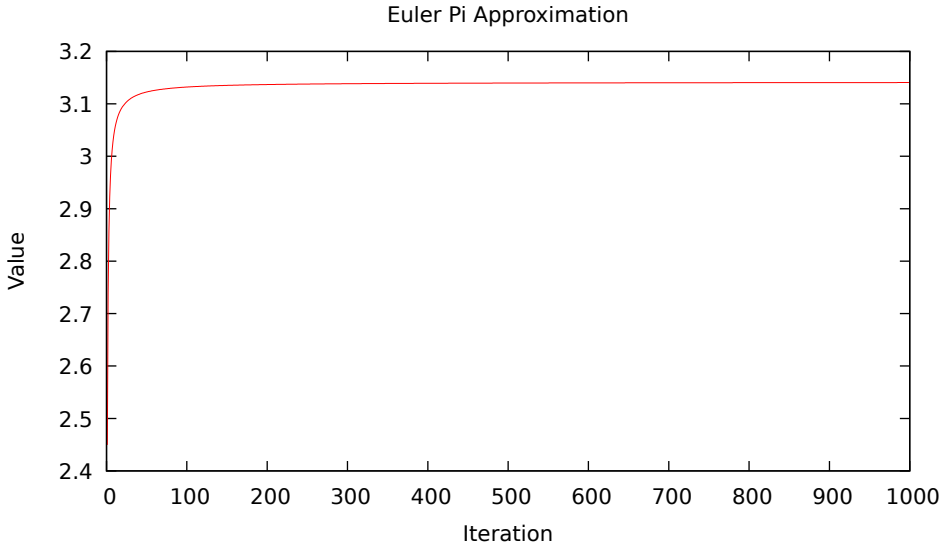
4.3 Viète's Formula (π)



$\frac{2}{\pi} = \Pi_{n=1}^{\infty} a_n$ where $a_1 = \sqrt{2}$ and $a_n = \sqrt{2 + a_{n-1}}$ for all $n > 1$.

Viète's Formula is also extremely efficient for approximating π. By the 5th iteration, the approximation is very close.

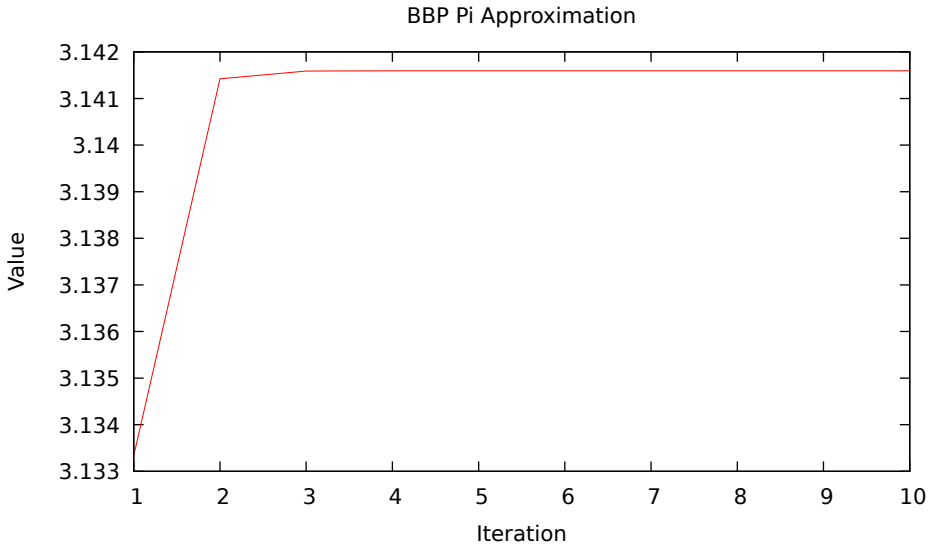
4.4 Euler's Solution (π)



$\pi = \sqrt{6 \sum_{n=1}^{n} \frac{1}{n^2}}$

Euler's Solution is, unfortunately, very inefficient for approximating π. It takes 50 iterations in order to be a rough approximation, and then takes another 100000+ iterations to be as precise as the previous methods.

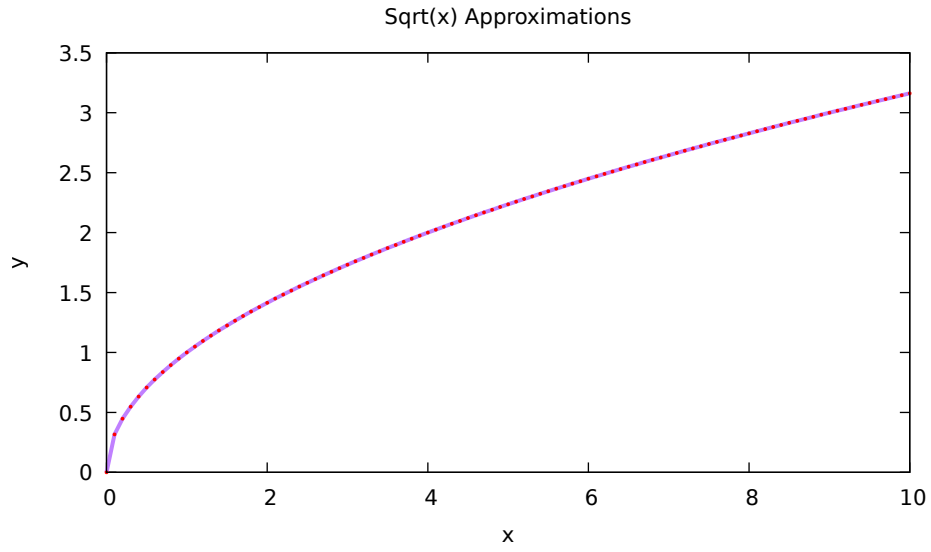
4.5 Bailey-Borwein-Plouffe Formula (π)



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

The Baily-Borwein-Plouffe formula is another extremely efficient calculation method for π. By the 2nd iteration, it is fairly precise, and then continues to get more accurate.

4.6 Newton-Raphson Method ($\sqrt{x}$)



$$f(\sqrt{x}) = x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

The Newton-Raphson Method for calculating square roots proved to be quite accurate. The approximated roots were compared against the <math.h> library's square root function (red dots) and were extremely close over the range [0:10).