Assignment 2
A Little Slice of π

Prof. Darrell D. E. Long
CSE 13S – Winter 2023

Due: January 29th at 11:59 pm

1 Introduction

> *Programming is one of the most difficult branches of applied mathematics; the poorer mathematicians had better remain pure mathematicians.*
>
> —Edsger Dijkstra

As we know, computers are simple machines that carry out a sequence of elementary steps, albeit very quickly. Unless you have a special-purpose processor, a computer can only compute *addition*, *subtraction*, *multiplication*, and *division*. If you think about it, you will see that the functions that might interest you when dealing with real or complex numbers can be built up from those four operations. We use many of these functions in nearly every program that we write, so we ought to understand how they are created.

We cannot expect the computer to solve integrals such as

$$\left[\int_{-\infty}^{\infty} e^{-x^2}\,dx\right]^2 = \pi$$

in order to calculate π, and computing *numerical integrals* is more complicated than we want to attempt in this class. But, suppose that you wanted to calculate that integral, how would you do it? You might use a formula like this:

$$\frac{h}{3}\left[f(x_0) + 2\sum_{j=1}^{\lfloor n/2-1\rfloor} f(x_{2j}) + 4\sum_{j=1}^{\lfloor n/2\rfloor} f(x_{2j-1}) + f(x_n)\right]$$

which is called the *composite Simpson's* $\frac{1}{3}$ *rule*. If you were to compute $\int_{-5}^{+5}$ the error would be just 9.69935×10^{-12} or about 11 digits of precision.

Fortunately, you may recall from your Calculus class, with some conditions a function $f(x)$ can be represented by its Taylor series expansion near some point $f(a)$:

$$f(x) = f(a) + \sum_{k=1}^{\infty} \frac{f^{(k)}(a)}{k!}(x-a)^k.$$

We will make extensive use of infinite series, suitably truncated, to solve many problems of interest. Since we cannot compute an infinite series, we must be content to calculate a finite number of terms. In general, the more terms that we calculate, the more accurate our approximation. Note: when you see Σ or Π, you should generally think of a `for` loop.

If you have forgotten (or have never taken) Calculus, do not despair. Attend a laboratory section for review: the concepts required for this assignment are do not extend beyond derivatives.

2 Fundamental Constants

Transcendental numbers, they transcend the power of algebraic methods.

—Leonhard Euler

We live in a world (especially if you are Platonist) that is described by and perhaps governed by mathematics and mathematical objects. Our world is populated by numbers—fundamental constants—that we know to exist, but which we cannot write exactly using our decimal (or any positional) number system. We are thus presented with a pleasant conundrum: How do we calculate these numbers that need in order to pursue science?

There are many little things that remind us of the wonders of the physical world in which we live. One of the most beautiful things in mathematics, which Richard Feynman called "our jewel," is *Euler's identity*:

$$e^{i\pi} + 1 = 0$$

which unites the most fundamental numbers in a single formula. A simple informal proof will show you why this is so.

The tool we will use for our proof is the *Taylor series* (Brook Taylor, 1685–1731). Taylor showed that you can write most functions as an infinite sum (there are some restrictions). This allows us to evaluate a function $f(x)$ near a point a by writing it like this:

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(a)}{k!}(x-a)^k = f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f^{(3)}(a)}{3!}(x-a)^3 + \cdots$$

where f' is the *first derivative*, f'' is the second, $f^{(3)}$ is the third, and so forth. For example,

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} + \cdots$$

and similarly,

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} + \cdots$$

Now, notice that *sine* has the odd powers of x, while *cosine* has the even powers of x. The reason for this is simple, it is that we are using $a = 0$, and since the $\sin' x = \cos x$ and $\cos' x = -\sin x$ and since $\sin 0 = 0$ and $\cos 0 = 1$ all the trigonometric functions drop out and just leave us with powers of x. We also need to look at the exponential function,

$$e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots$$

If you look carefully at the Taylor series expansion of the exponential function, you will see that it has both the even and odd powers of x, but the signs are all positive, while for *sine* and *cosine* half of the terms are negative.

The imaginary number $i = \sqrt{-1}$ solves the problem for us,

$$e^{ix} = 1 + ix - \frac{x^2}{2!} - \frac{ix^3}{3!} + \frac{x^4}{4!} + \frac{ix^5}{5!} - \frac{x^6}{6!} - \frac{ix^7}{7!} + \frac{x^8}{8!} + \frac{ix^9}{9!} - \frac{x^{10}}{10!} + \cdots$$

and look closely at the even numbered terms. You will see that they are the same as $\cos x$, and if you look at the odd numbered terms, you will see that they are the same as $i \sin x$ (you simply have to factor out the i). Consequently, we see that

$$e^{ix} = \cos x + i \sin x.$$

If you let $x = \pi$, then

$$e^{i\pi} = \cos \pi + i \sin \pi = -1 + i \times 0 = -1.$$

Thus, if $e^{i\pi} = -1$ then $e^{i\pi} + 1 = 0$. If you are like most scientists, you are left with a feeling of awe.

Can we make use of this jewel to calculate a value of π? We will start with $e^{i\pi} + 1 = 0$ and subtract 1 from both sides, and we get $e^{i\pi} = -1$. We take the square root of both sides:

$$\sqrt{e^{i\pi}} = \sqrt{-1}$$

and we get $e^{\frac{i\pi}{2}} = i$. We take the i^{th} root of both sides:

$$\sqrt[i]{e^{\frac{i\pi}{2}}} = \sqrt[i]{i}$$

which simplifies to $e^{\frac{\pi}{2}} = \sqrt[i]{i}$. We are almost finished once we take the natural logarithm of both sides: $\log(e^{\frac{\pi}{2}}) = \log(\sqrt[i]{i})$ yielding $\frac{\pi}{2} = \log(\sqrt[i]{i})$ which simplifies to $\pi = 2\log(\sqrt[i]{i}) = 2\log(i^{-i})$. So ultimately we find that π is the logarithm of the i^{th} root of an imaginary number. We may want to contemplate the words of Benjamin Peirce (1809–1880), a professor of mathematics at Harvard, who said "Gentlemen, that is surely true, it is absolutely paradoxical; we cannot understand it, and we don't know what it means. But we have proved it, and therefore we know it is the truth."

3 Calculating e

> *Let us change our traditional attitude to the construction of programs. Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.*
>
> —Donald Knuth

The number e, also known as *Euler's number* (Leonhard Euler, 1707–1783), is an irrational mathematical constant approximately equal to 2.71828, that appears pervasively in the natural and mathematical worlds. It is the base of the natural logarithm, it is the limit of $\lim_{n\to\infty}(1+\frac{1}{n})^n$ which was discovered by Jacob Bernoulli in his work on the calculation of compound interest. And, of course, it can be expressed as the Taylor series:

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

How many terms must you compute? Fewer than you might expect, since $k!$ grows very fast. You will be determining that experimentally as part of this assignment.

If we are naïve about computing the terms of the series we can quickly get into trouble — the values of $k!$ get large *very quickly*. We can do better if we observe that:

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

At first, that looks like a recursive definition (and in fact, you could write it that way, but it would be wasteful). As we progress through the computation, assume that we know the previous result. We then just have to compute the next term and multiply it by the previous term. At each step we just need to compute $\frac{x}{k}$, starting with $k = 0!$ (remember $0! = 1$) and multiply it by the previous value and add it into the total. It turns into a simple `for` or `while` loop.

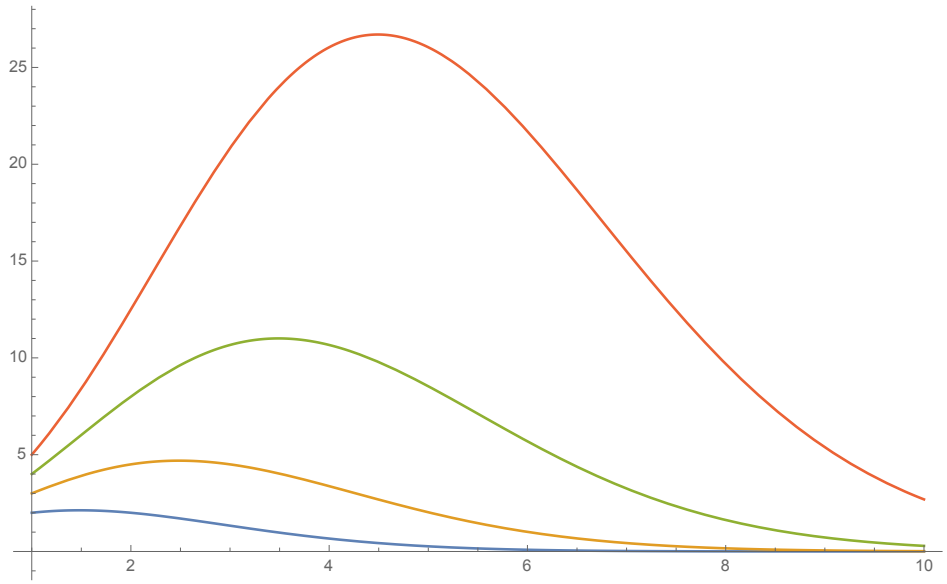

Figure 1: Comparing $\frac{x^k}{k!}$ for $x = 2, 3, 4, 5$.

4 Calculating π

> *The best programs are written so that computing machines can perform them quickly and so that human beings can understand them clearly. A programmer is ideally an essayist who works with traditional aesthetic and literary forms as well as mathematical concepts, to communicate the way that an algorithm works and to convince a reader that the results will be correct.*
>
> —Donald Knuth

Why, you might ask, do we need to calculate π? In practice, we do not: it is available as part of the of the `<math.h>` library as `M_PI`, since we do need it for all manner of scientific and engineering calculations. The area of a circle is πr^2 and the volume of a sphere is $\frac{4}{3}\pi r^3$. The cosmological constant is

$$\Lambda = \frac{8\pi G}{3c^2}\rho.$$

Heisenberg's uncertainty principle is given by

$$\Delta x \Delta p \geq \frac{h}{4\pi}.$$

Einstein's general relativity field equation is

$$R_{\mu\nu} - \frac{1}{2}g_{\mu\nu} + \Lambda g_{\mu\nu} = \frac{8\pi G}{c^4}T_{\mu\nu},$$

and so forth. No matter where you look, π is pervasive in the physical world.

So why do we calculate it? Well, suppose the person who typed up the `<math.h>` file made a mistake. Or, perhaps you need more accuracy than is normally provided. But ultimately, most people do it for the sheer fun of it. Computations such as finding the most digits of π fall into the area of experimental mathematics.

Experimental mathematics is an approach to mathematics in which computation is used to investigate mathematical objects and identify properties and patterns. It has been defined in a discussion by J. Borwein, P. Borwein, R. Girgensohn and S. Parnes as "that branch of mathematics that concerns itself ultimately with the codification

and transmission of insights within the mathematical community through the use of experimental (in either the Galilean, Baconian, Aristotelian or Kantian sense) exploration of conjectures and more informal beliefs and a careful analysis of the data acquired in this pursuit."

In the subsequent sections, we will present a number of functions p of n that are series that can be used to approximate π. In each of these formulæ there is a $p(n)$ where the limit

$$\lim_{n\to\infty} p(n) = \pi.$$

We know, both through experiment (which usually comes first) and mathematical proof, that these do result in π. The question then is: Which of these compute it most efficiently? Efficiency in this case is measured in the *number of terms* (or factors, in the case of Wallis and Viète).

4.1 The Leibniz Formula

The Leibniz formula for π (Gottfried Wilhelm von Leibniz, 1646–1716), given by

$$p(n) = 4\sum_{k=0}^{n}\frac{(-1)^k}{2k+1} = 4\left(1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\frac{1}{9}-\cdots\right) = \left((-1)^n H_{\frac{n}{2}+\frac{1}{4}} - H_{\frac{n}{2}-\frac{1}{4}}\right)+\pi$$

converges *extremely slowly*, since the *error term* involves *harmonic numbers*, and consequently is not reasonable for calculation. It is no more than the Taylor series expansion of $\tan^{-1} x$ with $x = 1$.

4.2 The Madhava Series

The Madhava series (Mādhava of Sangamagrāma, c. 1340 – c. 1425) is also related to $\tan^{-1} x$,

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

but it gives us a more rapidly converging series, given by:

$$p(n) = \sqrt{12}\sum_{k=0}^{n}\frac{(-3)^{-k}}{2k+1} = \sqrt{12}\left[\frac{1}{2}3^{-n-1}\left((-1)^n\Phi\left(-\frac{1}{3},1,n+\frac{3}{2}\right)+\pi 3^{n+\frac{1}{2}}\right)\right].$$

Which leaves us with the problem of calculating $\sqrt{12}$. You may be thinking: "Can't I just use the library?" The answer, of course, is *no*, you will need to compute $\sqrt{12}$ on your own. Do you need to worry about Φ? No, you just need to understand that in the limit, the Φ term (called the *Lerch transcendent*) goes to *zero* and that the remaining term

$$\frac{\pi}{2}3^{-n-1}3^{n+\frac{1}{2}} = \frac{\pi}{2\sqrt{3}} = \frac{\pi}{\sqrt{12}}.$$

4.3 The Wallis Series

John Wallis (1616–1703) was an English clergyman and mathematician who is given partial credit for the development of infinitesimal calculus. He gave us a series that is purely multiplicative (instead of terms, we have factors):

$$p(n) = 2\prod_{k=1}^{n}\frac{4k^2}{4k^2-1} = \frac{\pi\Gamma(n+1)^2}{\Gamma\left(n+\frac{1}{2}\right)\Gamma\left(n+\frac{3}{2}\right)}.$$

It is easy to calculate, but how rapidly does it converge? What is this Γ function? Do we need to compute it? How do we know this converges to π? Let's factor out π, then take the limit and note that

$$\lim_{n\to\infty}\frac{\Gamma(1+n)^2}{\Gamma\left(\frac{1}{2}+n\right)\Gamma\left(\frac{3}{2}+n\right)} = 1.$$

This tells us that as you compute more terms, you get closer to the true value of π.

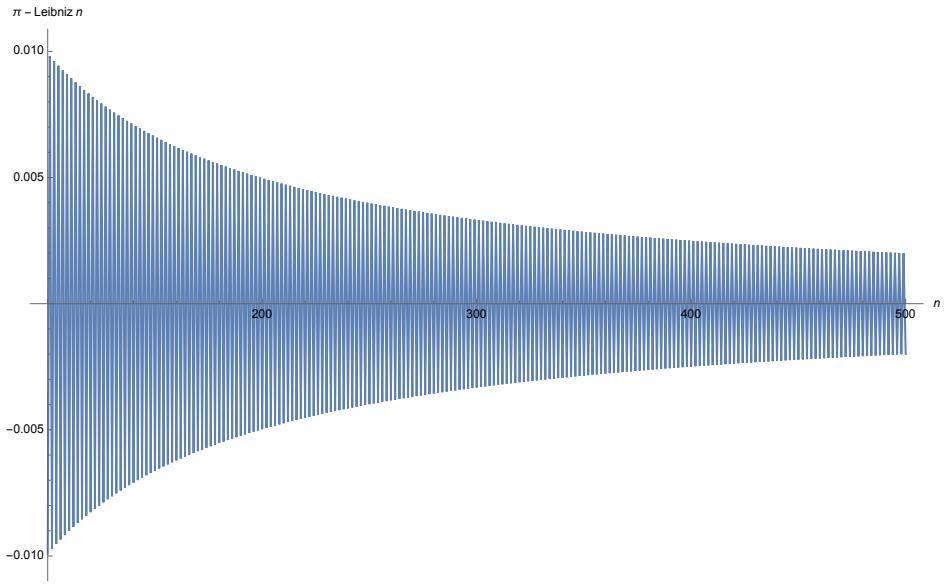

Figure 2: Leibniz sequence $4\sum_{k=0}^{n}\frac{(-1)^k}{2k+1}$

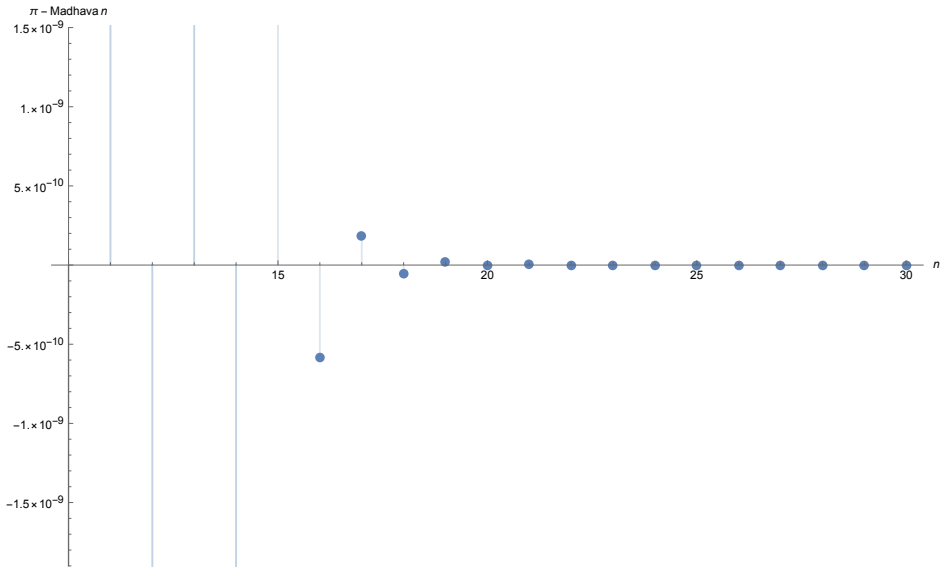

Figure 3: Madhava sequence $\sqrt{12}\sum_{k=0}^{n}\frac{(-3)^{-k}}{2k+1}$

4.4 Euler's Solution

The Basel problem is a problem in mathematical analysis with relevance to number theory, first posed by Pietro Mengoli in 1650 and solved by Leonhard Euler in 1734. The Basel problem asks for the precise summation of the reciprocals of the squares of the natural numbers, *i.e.* the sum of the infinite series

$$\sum_{k=1}^{\infty} \frac{1}{k^2} = \frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \cdots = H_{\infty}^{(2)},$$

which again involves harmonic numbers. Euler's solution showed that the solution is $\pi^2/6$, but his method gave us this series:

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

which also requires us to calculate the square root, of this case an unknown (until we calculate it) number.

4.5 The Bailey-Borwein-Plouffe Formula

The Bailey-Borwein-Plouffe formula (BBP formula) is a formula for π. It was discovered in 1995 by Simon Plouffe and is named after the authors of the article in which it was published, David H. Bailey, Peter Borwein, and Plouffe. The formula that they discovered is remarkably simple:

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

And if you desire to reduce it to the least number of multiplications, you can rewrite it in *Horner normal form*:

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

4.6 Viète's Formula

Named after François Viète, Viète's formula is a infinite product of nested radicals that can be used for calculations of π, though it should be noted that methods found before this specific formula are known to produce greater accuracy.

Viète's formula can be written as follows:

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

Or more simply,

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

where $a_1 = \sqrt{2}$ and $a_k = \sqrt{2 + a_{k-1}}$ for all $k > 1$.

4.7 Fastest Series

Perhaps the most interesting is the series given by Srinivasa Ramanujan (1887–1920)

$$\sum_{k=0}^{\infty} \frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \frac{(4k)!(1101+26390k)}{(k!)^4 396^{4k}}$$

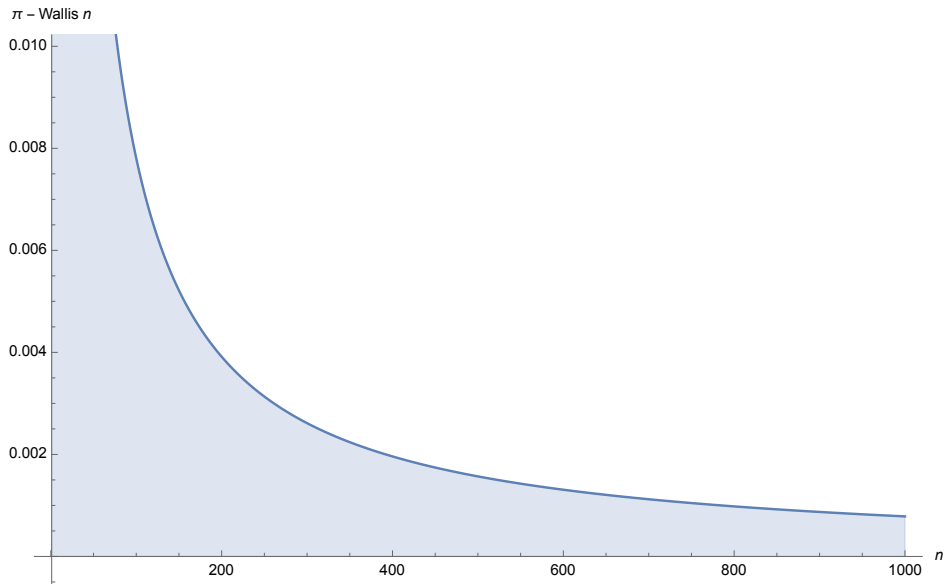

Figure 4: Wallis sequence $2\prod_{k=1}^{n}\frac{4k^2}{4k^2-1}$

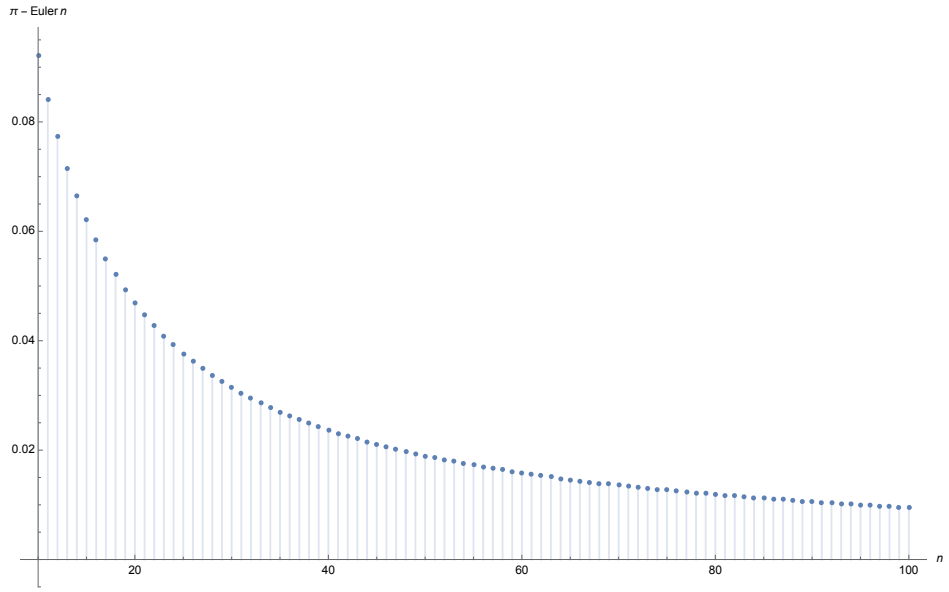

Figure 5: Euler sequence $\sqrt{6\sum_{k=1}^{n}\frac{1}{k^2}}$

which was later extended by the Chudnovsky brothers (whose lives seem to be centered around the calculation of π) to the series:

$$\frac{1}{\pi} = 12\sum_{k=0}^{\infty} \frac{(-1)^k (6k)!(13591409 + 545140134k)}{(3k)!(k!)^3 640320^{3k+3/2}}.$$

If you can find no particular significance in the constants in Ramanujan's formula, then perhaps we should give up any hope of understanding the Chudnovsky's!

Ultimately, what we have learned is this: we cannot know π exactly, but we know relations that involve π that we can approximate using a series. The question then becomes: Which series converges to π most rapidly?

5 The Problem of Irrationality

> *What good your beautiful proof on the transcendence of π: Why investigate such problems, given that irrational numbers do not even exist?*
>
> —Leopold Kronecker

Hippasus of Metapontum (c. 530 – c. 450 BC) was a Pythagorean philosopher, usually credited with the discovery of the *irrational numbers*. According to Prof. Dimotakis, the existence of irrational numbers was known to the Pythagoreans but it was a closely guarded secret. Hippasus was so enthralled by the knowledge that he went to the center of town and expounded the discovery. This so angered his fellow Pythagoreans, that they drowned him in the sea. Pappus of Alexandria merely says that the knowledge of irrational numbers originated in the Pythagorean school, and that the member who first divulged the secret perished by drowning.

> *It is related to Hippasus that he was a Pythagorean, and that, owing to his being the first to publish and describe the sphere from the twelve pentagons, he perished at sea for his impiety, but he received credit for the discovery, though really it all belonged to* him *(for in this way they refer to Pythagoras, and they do not call him by his name).*
>
> —Iamblichus the Syrian

Among irrational numbers are π most commonly known as the ratio the circumference of a circle to its diameter, Euler's number e, the golden ratio $\varphi = \frac{1+\sqrt{5}}{2}$, and, as we know, the square root of two. In fact, one can show that all square roots of natural numbers, other than of perfect squares, are irrational.

The irrational numbers are all the real numbers (denoted by $\mathbb{R}$) which are not rational numbers (denoted by $\mathbb{Q}$) and those are *equinumerous* with the integers (denoted by $\mathbb{Z}$). That is, irrational numbers cannot be expressed as the ratio of two integers. You may also be surprised to learn that $|\mathbb{Q}| = |\mathbb{Z}|$.

When the ratio of lengths of two line segments is an irrational number, the line segments are also described as being *incommensurable*, meaning that they share no *measure* in common, that is, there is no length, no matter how short, that could be used to express the lengths of both of the two segments as integers. Consider the triangle ABC where sides $a = 1$ and $b = 1$ then by the Pythagorean theorem $c^2 = a^2 + b^2$, thus, $c = \sqrt{a^2 + b^2} = \sqrt{1^2 + 1^2} = \sqrt{2}$. And for this, Hippasus lost his life.

The first number to be proved irrational was the $\sqrt{2}$, probably by our old friend Hippasus. Proofs by ancient Greeks were usually geometric in nature, and for many, difficult to follow. Instead, let us use this classical proof. Suppose that $\sqrt{2}$ is rational, which means that there exist p and q such that

$$\sqrt{2} = \frac{p}{q}.$$

We assume that p and q have no common factors, in other words, $\gcd(p, q) = 1$. If there are common factors, then we simply cancel them using the usual method for reducing fractions. We then square both sides of the equation resulting in

$$2 = \frac{p^2}{q^2},$$

which means that $p^2 = 2q^2$, and so p^2 is *even*. That means that p is even, and so p^2 is a multiple of 4. We can divide out 2 and the quotient is still even, including q^2, and so is q. If both p and q are even, then they share 2 as a common factor, contradicting our assumption.

Like all real numbers, irrational numbers can be expressed (approximately) in positional notation, notably as a decimal number. In the case of irrational numbers, the decimal expansion does not terminate, nor end with a repeating sequence. Conversely, a decimal expansion that terminates or repeats must be a rational number. These are provable properties of rational numbers and positional number systems, and are not used as definitions in mathematics.

As a consequence of Georg Cantor's (1845–1918) proof that the real numbers are uncountable and the rationals countable, it follows that almost all real numbers are irrational. In other words, there are more irrational numbers than there are rational numbers (in fact, $|\mathbb{R} - \mathbb{Q}| = |\mathbb{R}|$).

In practice, this means that we can never exactly represent an irrational number, even if we had infinite time and infinite space in which to do so. We must instead be content with approximations, but these must be of sufficient accuracy for us to engage in science, engineering, and technology (and in the case of φ, art and architecture).

Since, aside from perfect squares, all square roots are irrational, we must have a method to approximately compute them. In our case, to compute $\sqrt{x}$, you will use Newton's method, also called the *Newton-Raphson method*, by computing the inverse of x^2. It is an iterative algorithm to approximate roots of real-valued functions, *i.e.*, solving $f(x) = 0$. Starting with an initial guess, each iteration of Newton's method produces successively better approximations. A *Newton iterate* is defined as:

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

Each guess x_{k+1} gives a successive improvement over the previous guess x_k. In essence, we are using the *slope of the line* at the evaluation point to guide the next guess. The function begins with an initial guess $x_0 = 1.0$ that it uses to compute better approximations. `sqrt()` is sufficiently calculated once the value converges, *i.e.* when the difference between consecutive approximations is sufficiently small. In this case, $f(x) = x^2 - y$, so you can see that $f(x) = 0$ when $x = \sqrt{y}$.

```
1 def sqrt(x):
2     z = 0.0
3     y = 1.0
4     while abs(y - z) > epsilon:
5         z = y
6         y = 0.5 * (z + x / z)
7     return y
```

6 Your Task

Your task for this assignment is to implement a small number of mathematical functions (e^x and $\sqrt{x}$), mimicking `<math.h>`, and using them to compute the fundamental constants e and π. You will also need to write a dedicated test harness comparing your implemented functions with that of the math library's, then analyzing and presenting your findings in a writeup. The test harness should be a program named `mathlib-test`. The interface for your math library will be given in `mathlib.h`. You may not modify this file. The following sections will describe the functions that you need to write, and the files that should contain the functions.

You are *strictly forbidden* to use any functions from `<math.h>` in your own math library. You are also forbidden to write a `factorial()` function.

Each of the functions you will write must halt computation using an $\epsilon = 10^{-14}$, which will be defined in `mathlib.h`. For example, consider approximating the value of e. For sufficiently large k, $|x^k| < k!$. As seen in Figure 1, x^k dominates briefly, but is quickly overwhelmed by $k!$, making the ratio rapidly approach zero.

6.1 `e.c`

This file should contain two functions: `e()` and `e_terms()`. The former function will approximate the value of e using the Taylor series presented in §3 and track the number of computed terms by means of a `static` variable local to the file. The latter function will simply return the number of computed terms.

6.2 `madhava.c`

This file should contain two functions: `pi_madhava()` and `pi_madhava_terms()`. The former function will approximate the value of π using the Madhava series presented in §4.2 and track the number of computed terms with a `static` variable, exactly like in `e.c`. The latter function will simply return the number of computed terms.

6.3 `euler.c`

This file should contain two functions: `pi_euler()` and `pi_euler_terms()`. The former function will approximate the value of π using the formula derived from Euler's solution to the Basel problem, as described in §4.4. It should also track the number of computed terms. The latter function will simply return the number of computed terms.

6.4 `bbp.c`

This file should contain two functions: `pi_bbp()` and `pi_bbp_terms()`. The former function will approximate the value of π using the Bailey-Borwein-Plouffe formula presented in §4.5 and track the number of computed terms. The latter function will simply return the number of computed terms.

6.5 `viete.c`

This file should contain two functions: `pi_viete()` and `pi_viete_factors()`. The former function will approximate the value of π using Viète's formula as presented in §4.6 and track the number of computed factors. The latter function will simply return the number of computed factors.

6.6 `newton.c`

This file should contain two functions: `sqrt_newton()` and `sqrt_newton_iters()`. The former function will approximate the the square root of the argument passed to it using the Newton-Raphson method presented in §5. This function should also track the number of iterations taken, which the latter function will return.

6.7 `mathlib-test.c`

This file will contain the main test harness for your implemented math library. It should support the following command-line options:

- `-a:` Runs all tests.
- `-e:` Runs e approximation test.
- `-b:` Runs Bailey-Borwein-Plouffe π approximation test.
- `-m:` Runs Madhava π approximation test.
- `-r:` Runs Euler sequence π approximation test.
- `-v:` Runs Viète π approximation test.
- `-n:` Runs Newton-Raphson square root approximation tests.

- `-s:` Enable printing of statistics to see computed terms and factors for each tested function.
- `-h:` Display a help message detailing program usage.

The expected output for each of the e or π tests should resemble the following:

```
$ ./mathlib-test -e -b -v
e() = 2.718281828459046, M_E = 2.718281828459045, diff = 0.000000000000000
pi_bbp() = 3.141592653589793, M_PI = 3.141592653589793, diff = 0.000000000000000
pi_viete() = 3.141592653589775, M_PI = 3.141592653589793, diff = 0.000000000000018
```

Note that the newline should occur *after* the printed difference. You can refer to the reference program in the resources repostiroy for the example output. With the statistics option enabled, the output should resemble the following:

```
$ ./mathlib-test -e -b -v -s
e() = 2.718281828459046, M_E = 2.718281828459045, diff = 0.000000000000000
e terms = 18
pi_bbp() = 3.141592653589793, M_PI = 3.141592653589793, diff = 0.000000000000000
pi_bbp() terms = 11
pi_viete() = 3.141592653589775, M_PI = 3.141592653589793, diff = 0.000000000000018
pi_viete() terms = 23
```

You will specially test your `sqrt_newton()` function in the range [0, 10) in steps of 0.1. Again, you can refer to the reference program in the resources repository for the example output. Any `double` that is printed should use the following `printf()` format specifier:

```c
double pi = M_PI;
printf("%16.15lf\n", pi); // Newline is included as an example.
```

7 Command-line Options

A few dud universes can really clutter up your basement.

—Neal Stephenson, *In the Beginning... Was the Command Line*

Your test harness will determine which of your implemented functions to run through the use of *command-line option*. In most **C** programs, the `main()` function has two parameters: `int argc` and `char **argv`. A command, such as `./exec arg1 arg2`, is split into an array of strings referred as arguments. The parameter `argv` is this array of strings. The parameter `argc` serves as the argument counter, which is the number of arguments that were supplied. Try the following code, and make sure that you understand it.

Printing out supplied command-line arguments.

```c
#include <stdio.h>

int main(int argc, char **argv) {
    for (int i = 0; i < argc; i += 1) {
        printf("argv[%d] = %s\n", i, argv[i]);
    }
    return 0;
}
```

A command-line option is an argument, usually prefixed with a hyphen, that modifies the behavior of a command or program. They are typically parsed using the `getopt()` function. Do not attempt to parse the command-

line arguments yourself. Instead, use the `getopt()` function. Command-line options must be defined in order for `getopt()` to parse them. These options are defined in a string, where each character in the string corresponds to an option character that can be specified on the command-line. Upon running the executable, `getopt()` should be used to scan through the command-line arguments, checking for option characters in a loop.

Parsing options using `getopt()`.

```c
#include <stdio.h>
#include <unistd.h>

#define OPTIONS "pi:"

int main(int argc, char **argv) {
    int opt = 0;
    while ((opt = getopt(argc, argv, OPTIONS)) != -1) {
        switch (opt) {
        case 'p':
            printf("-p option.\n");
            break;
        case 'i':
            printf("-i option: %s is parameter.\n", optarg);
            break;
        }
    }
    return 0;
}
```

This example program supports two command-line options, `-p` and `-i`. Note that the option character `'i'` in the defined option string `OPTIONS` has a trailing colon. The colon signifies, when the `-i` option is enabled on the command-line, that `getopt()` should search for a option argument following it. An error is thrown by `getopt()` if an argument for a option, or *flag*, requiring one is not supplied.

8 Deliverables

You will need to turn in the following source code and header files:

1. `bbp.c`: This contains the implementation of the Bailey-Borwein-Plouffe formula to approximate π and the function to return the number of computed terms.
2. `e.c`: This contains the implementation of the Taylor series to approximate Euler's number e and the function to return the number of computed terms.
3. `euler.c`: This contains the implementation of Euler's solution used to approximate π and the function to return the number of computed terms.
4. `madhava.c`: This contains the implementation of the Madhava series to approximate π and the function to return the number of computed terms.
5. `mathlib-test.c`: This contains the `main()` function which tests each of your math library functions.
6. `mathlib.h`: This contains the interface for your math library.
7. `newton.c`: This contains the implementation of the square root approximation using Newton's method and the function to return the number of computed iterations.
8. `viete.c`: This contains the implementation of Viète's formula to approximate π and the function to return the number of computed factors.

You may have other source and header files, but *do not make things over complicated.* Any additional source code and header files that you may use must not use global variables. You will also need to turn in the following:

1. `Makefile`:
   - `CC = clang` must be specified.
   - `CFLAGS = -Wall -Wextra -Werror -Wpedantic` must be specified.
   - `make` must build the `mathlib-test` executable, as should `make all` and `make mathlib-test`.
   - `make clean` must remove all files that are compiler generated.
   - `make format` should format all your source code, including the header files.
2. `README.md`: This must use proper Markdown syntax. It must describe how to use your program and `Makefile`. It should also list and explain any command-line options that your program accepts. Any false positives reported by `scan-build` should be documented and explained here as well. Note down any known bugs or errors in this file as well for the graders.
3. `DESIGN.pdf`: This document *must* be a proper PDF. This design document must describe your design and design process for your program with enough detail such that a sufficiently knowledgeable programmer would be able to replicate your implementation. This does not mean copying your entire program in verbatim. You should instead describe how your program works with supporting pseudocode.
4. `WRITEUP.pdf`: This document *must* be a proper PDF. This writeup must include, at least, the following:
   - Graphs displaying the difference between the values reported by your implemented functions and that of the math library's. Use a UNIX tool — not some website — to produce these graphs. `gnuplot` is recommended. Attend section for examples of using `gnuplot` and other UNIX tools. An example script for using `gnuplot` to help plot your graphs will be supplied in the resources repository.
   - Analysis and explanations for any discrepancies and findings that you glean from your testing.

9 Submission

Refer back assignment 0 for the instructions on how to properly submit your assignment through `git`. Remember: *add*, *commit*, and *push*!

Your assignment is turned in *only* after you have pushed and submitted the commit ID you want graded on Canvas. "I forgot to push" and "I forgot to submit my commit ID" are not valid excuses. It is *highly* recommended to commit and push your changes *often*.

10 Supplemental Readings

- *The C Programming Language* by Kernighan & Ritchie
  - Chapter 3 §3.4 – 3.7
  - Chapter 4 §4.1 & 4.2 & 4.5
  - Chapter 7 §7.2
  - Appendix B §B4

Adding features does not necessarily increase functionality – it just makes the manuals thicker.