Section 1: Basic concepts in numerical analysis

February 4, 2019

1 Examples of problems in numerical analysis.

What is numerical analysis? A definition put forth by prof. Nick Trefethen is:

> Numerical analysis is the study of algorithms for solving problem of continuous mathematics.

The phrase continuous refers here to mathematical problems involving real or complex variables[1]. This is opposed to problems in discrete mathematics which are mostly adressed by sister discipline computer science. What is an algorithm, you may ask? Well let us give the following working definition:

> A set of steps and rules that are followed in order to solve a mathematical problem.

That is, if you give me the rules and steps, I can turn off my brain and simply follow the step and rules to extract a solution to some problem.

[1]Indeed, the real numbers are like a continuum, in contrast to integers which are discrete.

Taking a leisurely stroll, let us look at some typical problems addressed in numerical analysis to get a sense of the field.

Solving a linear system.
This is probably the most basic problem adressed in numerical analysis. Solve:

$$Ax = b$$

for x. Many other problems in numerical analysis are eventually reduced to this problem

Evaluation of functions.
At a fundamental level, computer arithmetic only does additions, subtractions, multiplication and division. How does one evaluate special functions such as $\sin(x)$, $\log x$, $\text{erf}(x)$, Bessel functions, Hankel function, Theta function, etc. within certain digits of accuracy? One can aso pose the same question for special constants such as pi, e, or Feigenbaum's number.

Function approximation and polynomial interpolation.
Function approximation and interpolation is an age old problem in numerical analysis and related subject of approximation theory. Function approximation nowadays plays a big role in the machine learnig.

Finding roots of a polynomial.
Finding the roots of a polynomial is probably the best example of why numerical analysisis necessary. Abel and Rufinni has shown that for polynomials of degree greater than 5, there are no algebraic solutions in terms of radicals. A negative theorem in mathematics with important consequences: the roots of polynomials of degree greater than 5 has to obtained through approximative means.

Numerical integration.
Numerical evaluation of integrals.

Solving an ordinary differential equation.
Again here, it it is not possible to obtain a simple algebraic solution if the dynamical system is sufficiently complex. Consider the transition from the 2 body problem to the 3 body problem.

Solving a PDE
Many problems in mathematical physics are modeled in terms PDEs. We would like to solve them.

Optimizating functions of real variables.
A common problem in industry.

2 Finite precision arithmetic.

Problems in numerical analysis intrinsically involve computations with real numbers. Unless the number turns out to be special (i.e. an integer or rational), to describe a real number requires an infinite number digits. In the decimal representation, a real number $x \in \mathbb{R}$ can be repesented by:

$$x = \pm 10^q \cdot 0.a_1a_2a_3a_4 \dots$$

where $q \in \mathbb{Z}$ is some integer and $a_k \in 0, 1, \dots, 9$ is one of the ten digits of the decimal representation. To give an example:

$$\pi = 10^1 \cdot 0.3146 \dots.$$

In a computer architecture[2], one does not have infinite memory to store all these digits, and most truncate the sequence somewhere. Also one cannot let the integer q become arbitrarily large in both the positive and negative direction. This would require again an infinite memory to store the integer. So must bound it by some integer N. Overall, instead of the exact representation, we have the approximate representation:

$$fl(x) = \pm 10^{\hat{q}} \cdot 0.a_1a_2a_3a_4 \dots a_M, \quad \hat{q} = \begin{cases} -N & q < -N \\ N & q > N \\ q & \text{otherwise} \end{cases}$$

The notation fl stands for floating point number. The word used to descibe such approximations. With floating point number, the continuum of real

[2]In computer architectures one of course uses the binary representation, but for the issues that we will be discussing here, it does not matter what representation is being used.

numbers is reduced to something finite. This has conquences. The digits a_1, a_2, a_3, a_4,..., a_M are called the mantissa or significand. The number $\hat{q}$ is called the exponent.

Relative error versus absolute error Let us look at the error introduced by approximating a number with a float. First, consider the case where $-N \leq q \leq N$. We have:

$$|x - fl(x)| = 10^{q-M} \cdot 0.a_{M+1}a_{M+2}\ldots.$$

In the worst case, $a_{M+1} = 9$, $a_{M+2} = 9$, etc. Recalling that $0.9999\ldots = 1$, the upper bound is:

$$|x - fl(x)| \leq 10^{q-M}$$

The appearance of q in the upper bound shows that the error has a dependence on the magnitude of the approximated number itself. This is because we are looking at the *absolute error.* If we consider the relative error, we see that:

$$\frac{|x - fl(x)|}{|x|} \leq 10^{-M}.$$

That is, the error decays with the number of digits in the signficand.

If $q > N$ or $q < -N$, saturation occurs. The absolute erors are respectively (overflow):

$$|x - fl(x)| = 10^q \cdot 0.a_1a_2a_3a_4\ldots - 10^N \leq 10^q - 10^N.$$

and (underflow)

$$|x - fl(x)| = 10^q \cdot 0.a_1a_2a_3a_4\ldots \leq 10^q.$$

A key observation: allowing more digits in the significand means better relative error, allowing larger integers in the exponent pushes the bounds for underflow and overflow.

Calculations with floating point arithmetic. Calculations with floating point arithmetic can (and typically will) amplify errors. You probably have experienced this phenomenon already in a highschool math homework. In some itermediate steps of your calculations, you perform some round-off. But instead taking the exact answer of your intermediate step, you use your rounded version in the subsequent calculations. Finally, it turns that your answer is nothing but close to the answer in the solution manual. You have been a victim of floating point arithmetic!

The analysis of round-off error in floating point arithmetic can go quite in depth. Here we are not going cover all those details, but let us look at an example of what *can* go wrong. Consider the task of subtracting two large, nearly equal positive numbers x and y:

$$x = \pm 10^q \cdot 0.a_1 a_2 \dots a_M a_{M+1} \dots, \qquad y = \pm 10^q \cdot 0.b_1 b_2 \dots b_M b_{M+1} \dots$$

with $q > M$ and $a_k = b_k$ for $k = 1, 2, \dots, M$, but $a_k \neq b_k$ for $k > M$. Subtraction in floating gives

$$fl(fl(x) - fl(y)) = 0$$

But we know that $x - y \neq 0$, and if fact a number of quite large magnitude! This phenomenon is called *catostrophic cancellation.* It can occur in the simplest of problems such as find the roots of quadratic function. Algebraically, the expressions:

$$x_1 = p - \sqrt{p^2 + q} = \frac{q}{p + \sqrt{p^2 + q}}$$

both describe the smallest root of the quadratic $x^2 - 2px - q$. However, the latter one avoids the possibility of catostrophic cancellation. On a computer, for $p = 12345678$ and $q = 1$, we get:

$$x_1 = -4.097819 \cdot 10^{-8},$$

for the first expression and:

$$x_1 = -4.050000 \cdot 10^{-8}.$$

The latter expresion is closer to the true solution.

Instabilities in algorithms due to floating point arithmetic. The approximate calculations induced by floating point arithmetic can cause instabilities which are unforeseen in exact arithmetic. We already encountered such an example when we discussed the two formulas for find the root of a quadratic function. To give a more "large-scale" example, consider the problem of solving a least squares problem for which the solution is well known to be expressed by:

$$\hat{x} = (A^T A)^{-1} A b.$$

As the expression shows, one has to take inverse of the matrix $A^T A$. But this matrix first needs to be evaluated. If we were allowed to do exact arithmetic, we would need twice the number of digits to store each entry of the matrix. However, round off and possible catostrophic cancellations are going to perturb these entries. Hence, instead of inverting $A^T A$, we would be inverting a nearby system $A^T A + E$. The question remains how this would affect further computations. We would be looking into this later in a bit more detail. For now, it should be noted that it has taken computer scientists and numerical analysists quite some effort to make solving least squares problems the standard technology it is today.

3 The computational complexity of an algorithm

One quantity which a numerical analysist must be able to assess is:

> How much work is required to apply some algorithm?

In particular, it is important to answer this question with regard to the scale/size of the problem. Let us illustrate that point. Consider the task of adding n numbers:

$$\sum_{k=1}^{n} a_k = a_1 + a_2 + \ldots + a_n$$

Obviously, n describes here the size of the problem. The work grows here proportionally: n additions are required to evaluate a sum of n numbers. No spectalar analysis indeed. Now consider another problem, evaluating the dot product of two vectors:

$$\sum_{k=1}^{n} a_k b_k.$$

Now we need to do n multiplications and n additions. If we assume for now that multiplications and additions require roughly the same amount of work, it should be clear that we need to do roughly $2n$ elementary floating point operations. There is still proportional growth, but the constant has increased. This is not much a significant change practically.

Now let us look at more demanding problem: evaluating a matrix vector product. Assuming a square matrix:

$$\begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & & \vdots \\ a_{n1} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix} = \begin{bmatrix} \sum_{k=1}^n a_{1k} b_k \\ \vdots \\ \sum_{k=1}^n a_{nk} b_k \end{bmatrix}$$

The problem size is again n, but we see that matrix vector multiplication is equivalent to n dot products. The number of operations are $2n^2$. This is interesting because now the growth is quadratic with respect to problem size. We would say that matrix vector multiplication is computationally more expensive.

The big-O notation. Our previous discussion motivates a very useful notation which is used a lot in numerical analysis (and actually pretty much everywhere in mathematics):

> A function $f(n)$ belongs to the set $\mathcal{O}(g(n))$ if there exists a constants $C, N > 0$ such that:
>
> $$|f(n)| \leq C|g(n)| \quad \text{for all} n > N.$$

This definition allows one to classify functions according to their asymptotic growth properties. That is, if $f(n) \in \mathcal{O}(g(n))$ but $g(n) \notin \mathcal{O}(f(n))$, then $g(n)$ grows faster asymptotically than $f(n)$. This means that no matter how large pick your constant $C > 0$, one can always find a $N > 0$ such that:

$$|g(n)| > C|f(n)| \quad \text{for } n > N.$$

It should be clear that $n \in \mathcal{O}(n^2)$ but $n^2 \notin \mathcal{O}(n)$.

Assesing the performance of an algorithm. The fairest way to quantify the performance of an algorithm to solve some mathematical problem can be loosely posed as follows:

> For given $\epsilon > 0$, how many arithemetic operations do we need to evualate the solution to a problem within ϵ precision?

Using floating point operations (as discussed before) is a way to quantify the number of operations. Later on, we will illustrate this idea of assesing the performance of an algorithm thoroughly by means of an example. For now, we would like to mention that measuring in terms of floating point operations could leave out a lot of details. For example, weighing addition/subtraction equally to multiplication/division is far from truth. Also, numbers with a huge mantissa require more work to evaluate, and that is all being ignored now.

4 Formulating well-posed problems

Given the fact that real computer algorithms are constrained by approximate calculations, it is very important to consider the potential sentivity of the probem to these approximate calculations. A problem which is inherently sensitive to such perturbations is by default a difficult problem to solve numerically. It will require extra elbow grease to show that things do not go kaput under finite precision arithmetic. Often such difficult situations can be avoided entirely by re-formulating the problem in the right way. But sometimes it is also intrinsic to the nature of the underlying physical problem itself. As a numerical analysist, you want to verify for yourself that what you aim to compute through approximations is not a fragile quantity. The best way to illustrate this is to consider the example of solving a linear system:

$$Ax = b.$$

A 2 by 2 example. Consider the following simple system:

$$\begin{bmatrix} 1 & \\ & \frac{1}{\sigma} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad \text{and} \quad \sigma > 0.$$

Now suppose that we perturb the right hand side by:

$$\begin{bmatrix} 1 & \\ & \frac{1}{\sigma} \end{bmatrix} \begin{bmatrix} x_1(\epsilon) \\ x_2(\epsilon) \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ \epsilon \end{bmatrix}.$$

This pertubations can be seen as the consequence of some round-off error that may occur in some algorithm. The important question that one should ask is: how does this affect the solution of the problem? Well, the solution can be derived explitly:

$$\begin{bmatrix} x_1(\epsilon) \\ x_2(\epsilon) \end{bmatrix} = \begin{bmatrix} 1 \\ \sigma\epsilon \end{bmatrix}$$

It should be clear that if $\sigma >> 0$ is an extremely larger, the perturbed solution can be quite signficantly different. Likewise, consider a perturbation to the matrix itself:

$$\begin{bmatrix} 1 & \\ & \frac{1}{\sigma} + \epsilon \end{bmatrix} \begin{bmatrix} x_1(\epsilon) \\ x_2(\epsilon) \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

The solution is:

$$\begin{bmatrix} x_1(\epsilon) \\ x_2(\epsilon) \end{bmatrix} = \begin{bmatrix} 0 \\ \frac{\sigma}{1+\epsilon\sigma} \end{bmatrix}$$

If $\sigma >> 0$ is again large, we have a perturbation or rougly $1/\epsilon$ to the solution which is quite crazy. We are dealing here with a so-called ill-conditioned matrix. Such matrices can be a potential nightmare for numerical analysists.

Vector and matrix norms Our goal is to generalize what we have seen in the simple 2 by 2 example. Let us revise some basic linear algebra notions first. Consider a vector $x = \begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix}^T$. The (Euclidian) norm of a matrix is defined as:

$$\|x\| := \sqrt{x_1^2 + x_2^2 + \ldots + x_n^2} = \sqrt{x^T x}$$

Similarly, we can define the so-called matrix norm:

$$\|A\| := \max_{0 \neq x \in \mathbb{R}^n} \frac{\|Ax\|}{\|x\|} = \max_{\|x\|=1} \|Ax\|$$

As you may know, this describes the worst-case amplication factor of vector x when applied to a matrix A. Let us recall some few important properties of norms which we will be needing later:

1. $\|x\| > 0$
2. $\|cx\| = |c| \, \|x\|$ for some scalar $c \in \mathbb{R}$
3. $\|x+y\| \leq \|x\| + \|y\|$ (the triangle inequality)

and the multplicatice property for matrix norms:

$$\|AB\| \leq \|A\| \, \|B\| \, .$$

The condition number A general linear system with a general perturbation would look as follows:

$$(A + \epsilon F)x(\epsilon) = b + \epsilon f, \quad x(0) = x$$

For small perturbations ϵ, the solution $x(\epsilon)$ depends smoothly on the perturbations if A is nonsingular (can you explain why?). From implicit differentiation, we can show that:

$$\dot{x}(0) = A^{-1}(f - Fx)$$

A taylor series $x(\epsilon) = x + \epsilon \dot{x} + \mathcal{O}(\epsilon^2)$ hence can be expressed as:

$$x(\epsilon) - x(0) = \epsilon A^{-1}(f - Fx) + \mathcal{O}(\epsilon^2).$$

Now applying norms on both sides, we can show that:

$$\begin{aligned}
\|x(\epsilon) - x\| &= ||\epsilon A^{-1}(f - Fx) + \mathcal{O}(\epsilon^2)|| \\
&\leq |\epsilon| \left\|A^{-1}(f - Fx)\right\| + \mathcal{O}(\epsilon^2) \\
&\leq |\epsilon| \left\|A^{-1}\right\| \|(f - Fx)\| + \mathcal{O}(\epsilon^2) \\
&\leq |\epsilon| \left\|A^{-1}\right\| \|f\| + |\epsilon| \, \|Fx\| + \mathcal{O}(\epsilon^2) \\
&\leq |\epsilon| \left\|A^{-1}\right\| \|f\| + |\epsilon| \, \|F\| \, \|x\| + \mathcal{O}(\epsilon^2)
\end{aligned}$$

Notice how we have used all those norm properties of the previous section. To continue, let us divide by $\|x\|$ to obtain:

$$\frac{\|x(\epsilon) - x\|}{\|x\|} \leq |\epsilon| \frac{\|A^{-1}\| \, \|f\|}{\|x\|} + |\epsilon| \, \|F\| + \mathcal{O}(\epsilon^2) \tag{1}$$

Now recognize that:

$$\|b\| \leq \|A\| \, \|x\| \quad \text{or} \quad \frac{1}{\|x\|} \leq \frac{\|A\|}{\|b\|}$$

substitution back into (1) and doing some further manipulations yields:

$$\begin{aligned} \frac{\|x(\epsilon) - x\|}{\|x\|} &\leq |\epsilon| \frac{\|A^{-1}\| \, \|A\| \, \|f\|}{\|b\|} + |\epsilon| \, \|F\| + \mathcal{O}(\epsilon^2) \\ &\leq \left\|A^{-1}\right\| \|A\| \left(|\epsilon| \frac{\|f\|}{\|b\|} + |\epsilon| \frac{\|F\|}{\|A\|} \right) + \mathcal{O}(\epsilon^2) \end{aligned}$$

We now have derived the following revealing result:

Theorem 1. *Consider a linear system $Ax = b$ and suppose we have the following perturbation:*

$$(A + \epsilon F) x(\epsilon) = b + \epsilon f, \quad x(0) = x$$

For sufficiently small ϵ, The relative error of the perturbed solution is bounded by:

$$\frac{\|x(\epsilon) - x\|}{\|x\|} \leq \left\|A^{-1}\right\| \|A\| \left(\rho_b + \rho_A\right) + \mathcal{O}(\epsilon^2)$$

where:

$$\rho_b \leq |\epsilon| \frac{\|f\|}{\|b\|}, \qquad \rho_A = \leq |\epsilon| \frac{\|F\|}{\|A\|}.$$

Interpretation of theorem: The terms ρ_b and A, describe respectively the measure of pertubations in b and A respectively, the quantity

$$\left\|A^{-1}\right\| \|A\|$$

is called the condition number and can be seen as as gain which amplifies this perturbations. If the condition is small, we are more less gauranteed to be safe. If the condition number is large, hell may break lose (or it may as well not, since this is only an upper bound!).

5 Designing efficient algorithms: a cute example

The main goal of a numerical analysist is to design algorithms to solve mathematical problem statements such as those presented in section 1. The focus here is to compute the solution to a problem with the highest accuracy possible (i.e. read here: number of correct digits), while using as few elementary operations as possible. Any significant breakthrough in that regard would be considered a progession. A lot of creativity is often required to make such progressions. This is what makes numerical analysis exciting!

Let us illustrate this by means of an example. Consider the probem of computing a rational approximation to the number π. One way to solve this problem is to use Leibnitz formula:

$$\pi = 4\sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}.$$

Analysis of Leibnitz's formula. Leibnitz formula can be derived as follows. Recall from calculus:

$$\begin{aligned}
\frac{\pi}{4} &= \arctan(1) - \arctan(0) \\
&= \int_0^1 \frac{1}{1+x^2}\mathrm{d}x \\
&= \int_0^1 \frac{1}{1-(-x^2)}\mathrm{d}x \\
&= \int_0^1 \left(\sum_{k=1}^{\infty}(-1)^k x^{2k}\right)\mathrm{d}x
\end{aligned}$$

In the last manipulation, we used the geometric series, which is convergent for the range values considered in the integral. Exchanging the summation with the integral, and then subsequently evaluating the integral would give us the desired result. However, since it is an infinite sum we need to be more careful and do things more carefully. Let us continue.

$$\begin{aligned}
\frac{\pi}{4} &= \int_0^1 \left(\sum_{k=0}^{\infty}(-1)^k x^{2k}\right) \mathrm{d}x \\
&= \int_0^1 \sum_{k=0}^{n}(-1)^k x^{2k} + \sum_{k=n+1}^{\infty} (-1)^k x^{2k}\mathrm{d}x \\
&= \int_0^1 \sum_{k=0}^{n}(-1)^k x^{2k} + (-1)^{n+1}x^{2n+2}\sum_{k=1}^{\infty}(-1)^k x^{2k}\mathrm{d}x \\
&= \int_0^1 \sum_{k=0}^{n}(-1)^k x^{2k} + \frac{(-1)^{n+1}x^{2n+2}}{1+x^2}\mathrm{d}x \\
&= \sum_{k=1}^{n}\int_0^1 (-1)^k x^{2k}\mathrm{d}x + \int_0^1 \frac{(-1)^{n+1}x^{2n+2}}{1+x^2}\mathrm{d}x \\
&= \sum_{k=1}^{n}\frac{(-1)^k}{2k+1} + \int_0^1 \frac{(-1)^{n+1}x^{2n+2}}{1+x^2}\mathrm{d}x
\end{aligned}$$

In other words:

$$\begin{aligned}
\left|\pi - 4\sum_{k=0}^{n}\frac{(-1)^k}{2k+1}\right| &\leq 4\left|\int_0^1 \frac{(-1)^{n+1}x^{2n+2}}{1+x^2}\mathrm{d}x\right| \\
&\leq 4\int_0^1 \left|\frac{(-1)^{n+1}x^{2n+2}}{1+x^2}\right| \mathrm{d}x \\
&\leq 4\int_0^1 \frac{x^{2n+2}}{1+x^2}\mathrm{d}x \\
&\leq 4\int_0^1 x^{2n+2}\mathrm{d}x \\
&\leq \frac{4}{2n+1}
\end{aligned}$$

As $n \to \infty$, the right hand side decays to zero, which shows that we can approximate by simply evaluating the partial sum $\sum_{k=0}^{n} \frac{(-1)^k}{2k+1}$ with more and more terms. There are however two problems with this method of approximating π:

1. The possibility of catastrophic cancellation which may lead to numerical instabilities. Indeed, Leibnitz is an alternating sum:

   $$\pi = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots\right)$$

   and this should raise eyebrows of concern.

2. Even if we were able to do exact arithmetic, the sum converges very slowly. To approximate π within some given $\epsilon > 0$, i.e.

   $$\left|\pi - 4\sum_{k=1}^{n} \frac{(-1)^k}{2k+1}\right| \leq \epsilon,$$

   we need to add $n > 2/\epsilon + \frac{1}{2}$ terms. If we would like to approximate the first d digits correctly, this would mean $n > 2 \cdot 10^d + \frac{1}{2}$. The number of terms needed grows exponentially! Given that evaluating every term in the summation takes roughly $\mathcal{O}(1)$, adding n terms will take $\mathcal{O}(n)$. Overall the complexity of the method is $\mathcal{O}(10^d)$. This is not a very practical way of evaluating π, we can do better!

What follows next is how we can resolve these issues.

Avoiding potential catastrophic cancellation. To avoid possible catastrophic cancellation. We can easily convert the alternating sum into a sum of only positive terms. Simply notice that:

$$\begin{aligned}
\pi &= 4\left(\left(1 - \frac{1}{3}\right) + \left(\frac{1}{5} - \frac{1}{7}\right) + \dots\right) \\
&= 4\sum_{k=0}^{\infty} \frac{1}{4k+1} - \frac{1}{4k+3} \\
&= 4\sum_{k=0}^{\infty} \frac{2}{(4k+1)(4k+3)}.
\end{aligned}$$

That solves the cancellation problem, but we have not really improved the compexity of the algorithm.

Accelerating the algorithm. We can derive a better of way approximating π through cleverly manipulating the Leibnitz sum to obtain a "better series". For that purpose, write[3]:

$$\sum_{k=0}^{n} \frac{(-1)^k}{2k+1} = a_0 - a_1 + a_2 - a_3 + \ldots + a_{n-1} - a_n, \quad a_k = \frac{1}{2k+1}. \tag{2}$$

Perform the manipulation:

$$\sum_{k=0}^{n} \frac{(-1)^k}{2k+1} = \frac{1}{2}a_0 + \frac{1}{2}\left((a_0 - a_1) - (a_1 - a_2) + \ldots - (a_n - a_{n+1})\right) - \frac{1}{2}a_{n+1}$$

We can repeat this step again:

$$\begin{aligned}&(a_0 - a_1) - (a_1 - a_2) + \ldots - (a_n - a_{n+1}) = \\ &((a_0 - 2a_1 + a_2) - (a_1 - 2a_2 + a_3) + \ldots - (a_n - 2a_{n+1} + a_{n+2})) - \frac{1}{2}(a_{n+1} - a_{n+2})\end{aligned}$$

to obtain:

$$\begin{aligned}\sum_{k=0}^{n} \frac{(-1)^k}{2k+1} = &\left(\frac{1}{2}a_0 + \frac{1}{4}\sum_{k=0}^{n}(a_k - a_{k+1}) + \frac{1}{8}\sum_{k=0}^{n}(a_k - 2a_{k+1} + a_{k+2})\right) \\ &- \left(\frac{1}{2}a_n + \frac{1}{4}(a_{n+1} - a_{n+2})\right)\end{aligned}$$

Doing this n times gives us the expression (please convince yourself that this is true!):

$$\sum_{k=0}^{n} \frac{(-1)^k}{2k+1} = \sum_{k=0}^{n} \frac{1}{2^{k+1}} \left(\sum_{j=0}^{k} (-1)^j \binom{k}{j} a_j\right) - \sum_{k=0}^{n-1} \frac{1}{2^{k+1}} \left(\sum_{j=0}^{k} (-1)^j \binom{k}{j} a_{n+1+j}\right).$$

You may be wondering why we are all doing this, but hold on for a second. It can be shown that the first summation term on the right hand side can also be used as a means to approximate π, and this summation converges much faster than the Leibnitz sum! What we will do next is to first show that:

$$\frac{\pi}{4} = \sum_{k=0}^{\infty} \frac{1}{2^{k+1}} \left(\sum_{j=0}^{k} (-1)^j \binom{k}{j} a_j\right). \tag{3}$$

[3]Notice that the partial sum is expressed assuming that n is some odd number

Then we will show that this sum converges more rapidly. Finally, we will do some further shenanigans to obtain a more efficient algorithm to approximate π for a given accuracy.

The expression (3) *is correct.*
To verify (3) we make the following observation:

$$a_n > a_n - a_{n+1} > a_n - 2a_{n+1} + a_{n+2} > \ldots > \sum_{j=0}^{n} (-1)^j \binom{k}{j} a_{n+1+j}.$$

Hence,

$$\begin{aligned}
\left| \sum_{k=0}^{n} \frac{(-1)^k}{2k+1} - \sum_{k=0}^{n} \frac{1}{2^{k+1}} \left(\sum_{j=0}^{k} (-1)^j \binom{k}{j} a_j \right) \right| &\leq \sum_{k=0}^{n-1} \frac{1}{2^{k+1}} \left(\sum_{j=0}^{k} (-1)^j \binom{k}{j} a_{n+1+j} \right) \\
&\leq a_n \sum_{k=0}^{n-1} \frac{1}{2^{k+1}} \left(\sum_{j=0}^{k} (-1)^j \binom{k}{j} a_{n+1+j} \right) \\
&\leq a_n \sum_{k=0}^{\infty} \frac{1}{2^{k+1}} \\
&= a_n = \frac{1}{2n+1}
\end{aligned}$$

As $n \to \infty$ these two sums converge to each other, hence (3) is correct.

The expression (3) *converges more rapidly.*
Now we need to find a way to bound this monster:

$$\left| \pi - \sum_{k=0}^{n} \frac{1}{2^{k+1}} \left(\sum_{j=0}^{k} (-1)^j \binom{k}{j} a_j \right) \right| \leq \sum_{k=n+1}^{\infty} \frac{1}{2^{k+1}} \left(\sum_{j=0}^{k} (-1)^j \binom{k}{j} a_j \right)$$

The trick is to again observe that:

$$a_0 > a_0 - a_1 > a_0 - 2a_1 + a_2 > \ldots > \sum_{j=0}^{k} (-1)^j \binom{k}{j} a_j < \ldots.$$

A look at the definition of a_k in (2) should tell you why. Once this little observation is made, the remaining part becomes easy:

$$\begin{aligned}\left|\pi - \sum_{k=0}^{n}\frac{1}{2^{k+1}}\left(\sum_{j=0}^{k}(-1)^j\binom{k}{j}a_j\right)\right| &\le \sum_{k=n+1}^{\infty}\frac{1}{2^{k+1}}\left(\sum_{j=0}^{k}(-1)^j\binom{k}{j}a_j\right)\\ &\le a_0\sum_{k=n+1}^{\infty}\frac{1}{2^{k+1}}\\ &= \frac{1}{2^n}\end{aligned}$$

This is great. To approximate π within a given ϵ error, we only now need $n > -\log_2 \epsilon$. In terms of correct digits d, we only need $n > d\log_2 10$. It seems that we now have $\mathcal{O}(d)$ algorithm to compute the first d digits of π. But this is far from the truth! We have a nested sum: to evaluate the 2nd term in (3) requires 2 additions, the 3rd term requires 6 additions, the 10th term will require 3628800 operations, etc. Nothing has really been achieved, but we are not done yet!

Our modified sum permits a more efficient algorithm.
We need to find a more efficient way to evaluate the terms in (3). This can be done through further manipulation as follows. That is, by evaluating the terms carefully, it can be observed that:

$$\begin{aligned}a_0 &= 1\\ a_0 - a_1 &= \frac{2}{3}\\ a_0 - 2a_1 + a_2 &= \frac{2\cdot 4}{3\cdot 5}\\ &\vdots\\ \sum_{j=0}^{k}(-1)^j\binom{k}{j}a_j &= \frac{2\cdot 4\cdot\ldots 2k}{3\cdot 5\cdot\ldots\cdot(2k+1)} = \frac{(2k)!}{(2k+1)!}\end{aligned}$$

Hence, plugging it into (3):

$$\frac{\pi}{4} = \sum_{k=0}^{\infty}\frac{1}{2^{k+1}}\frac{(2k)!}{(2k+1)!} = \sum_{k=0}^{\infty}\frac{1}{2}\frac{k!}{(2k+1)!}$$

This is an amazing result because:

$$\frac{k!}{(2k+1)!} = \frac{(k-1)!}{(2(k-1)+1)!}\frac{k}{2k+1}$$

The k-th term can be evaluated by simply scaling $(k-1)$-th term. Every term can be evaluated $\mathcal{O}(1)$ operations. We now have a $\mathcal{O}(d)$ algorithm to compute the first d digits of π.

Can we do even better? There exist even faster and better algorithms to approximate π, and in principle, this whole business of appoximating π has turned into bragging competition amongst mathematicians and computer scientists on who can compute the most number digits of π. Much of numerical analysis can be seen as a game of being the fastest to solve a problem(while being correct of course!). All numerical analysists are in principle like 100m dash runners trying to beat Usain Bolt's record.