Introduction to Numerical Analysis

Based on lectures by
Notes taken by yehelip

Winter 2025

Contents

1 Introduction

This course addresses what of all the math we have learned so far can we compute with the computer

For example, given the task of computing the determinant of a matrix A, we want to find the most efficient algorithm. The most efficient algorithm is the one that uses the least amount of operations.

One way of calculating $\det A$ is using permutations:

$$\det A = \sum_{\gamma \in S_n} \operatorname{sgn}(\gamma) a_{1\sigma(1)} \cdot a_{2\sigma(2)} \cdots a_{n\sigma(n)}.$$

Each permutation costs us n operations of multiplication, and to calculate the sum we perform $n! - 1$ summation operations. Since there are $n!$ permutations in S_n, the total amount of operations of this algorithm is $n \cdot n! + n! - 1$.

An alternative algorithm would be to find the eigendecomposition $A = U^T D U$ where U is an orthogonal matrix, and then calculate the product of the elements on the main diagonal of D (which are the eigenvalues of A). Using this algorithm we can compute $\det A$ in $Cn^3 + n - 1$ operations for some costant C. Although this algorithm is much faster than the previous ones, there are more efficient algorithms still.

Definition 1.1 (Big O notation)**.** Let $(x_n)_{n\geq 1}$ and $(y_n)_{n\geq 1}$ be sequences of real numbers. We say that $x_n = O(y_n)$ if there exist constants C and N such that for all $n > N$

$$|x_n| \leq C|y_n|$$

Here are some examples

$$\frac{n+1}{n^2} = O\left(\frac{1}{n}\right)$$
$$Cn^3 + n - 1 = O(n^3)$$
$$n \cdot n! + n! - 1 = O(n \cdot n!)$$

Using big O notation we can say that direct determinant calculations requires $O(n \dots n!)$ calculations, while using eigendecomposition to calculate it takes $O(n^3)$ operations.

However, the big O notation is still not very satisfying because we still have

$$Cn^3 + n - 1 = O(n \cdot n!).$$

To fix this problem we introduce the Θ notation.

Definition 1.2 (big Θ notation)**.** Let $(x_n)_{n\geq 1}$ and $(y_n)_{n\geq 1}$ be sequences of real numbers. We say that $x_n = \Theta(y_n)$ if there exist constants $0 < c < C$ and N such that for all $n > N$

$$c|y_n| \leq |x_n| \leq C|y_n|.$$

Here are some examples

$$a_k n^k + \cdots + a_1 n + a_0 = \Theta(n^k)$$
$$Cn^3 + n - 1 = \Theta(n^3)$$
$$n \cdot n! + n! - 1 = \Theta(n \cdot n!)$$

Another important method before we move to the next section, is Horner's method. It allows computing a polynomial $p(x) = \sum_{k=0}^{n} a_k x^n$ in $\Theta(n)$ operations instead of $\Theta(n^2)$ operations. It states that

$$\begin{aligned} &a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots + a_n x^n \\ &= a_0 + x\Big(a_1 + x\Big(a_2 + x\left(a_3 + \cdots + x(a_{n-1} + x a_n)\cdots\right)\Big)\Big). \end{aligned}$$

2 Digital Number Representation

We usually use base-10 expansion to represent number. For example, the number $x = 123.45$ means

$$x = 1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0 + 4 \times 10^{-1} + 5 \times 10^{-2}.$$

In this case, we say that x has a finite base-10 expansion. All non-negative real numbers have a (possibly infinite) base-10 expansion.

Theorem 2.1. *Let x be a real number in $[0,1)$, and $b \geq 2$ an integer. Then there exists $L \in \mathbb{Z}$ and a sequence $(c_k)_{k=1}^{\infty}$ where each c_k is in $\{0,1,\dots,b-1\}$, so that*

$$x = \sum_{k=1}^{\infty} c_k b^{-k}.$$

The motivation for this theorem is to prove for example, that computers, who work in base 2, can represent the same numbers we can represent in the way that is more convenient to us like base 10.

Lemma 2.2. *If $b \geq 2$ is an integer, $L \in \mathbb{Z}$ and $x \in [0, b^{L+1})$. Then there exists $c_L \in \{0,1,\dots,b-1\}$ such that $x - c_L b^L \in [0, b^L)$.*

Proof. We notice that

$$\bigcup_{j=0}^{b-1} \Big[j b^L, (j+1) b^L \Big) = [0, b^{L+1}).$$

So there exists $c_L \in \{0,1,\dots,b-1\}$ so that $x \in [c_L b^L, (c_L+1) b^L)$. Then

$$r_L = x - c_L b^L \in [0, b^L).$$

□

We can now go back to prove Theorem 2.1

Proof. By assumption $x \in [0, b^0)$. By Lemma 2.2 with $L = -1$ we now define:

$$r_1 := x - c_1 b^{-1} \in [0, b^{-1}).$$

And continue indefinitely

$$r_S := x - \sum_{k=1}^{S} c_k b^{-k} \in [0, b^{-k}).$$

This implies that $r_S \xrightarrow{S \to \infty} 0$ which means that

$$x = \sum_{k=1}^{\infty} c_k b^{-k}.$$

□

We can now conclude from the theorem that for $x > 0$, we can find L large enough so that $x < b^L$ and so $x b^{-L} < 1$. We can then write

$$x = b^L \sum_{k=1}^{\infty} c_k b^{-k}.$$

It is also possible in base-b for $b \geq 2$ to represent any non-negative number as:

$$x = (-1)^s \times [1.f]_b \times 2^m,$$

for some $s \in \{0,1\}$, $m \geq 0$ and f an infinite sequence of digits.

Remark 2.1. Obsvioulу we can also represent 0 because $0 \in \{0, 1, \ldots, b\}$.

In a computer, we represent integers using 32 bits. We use one bit to represent the sign of the integer, and the other as constants to the powers of two. In this way we can represent every integer $n \in [-(2^{-31} - 1), 2^{31} - 1]$ as such:

$$n = (-1)^{c_31} \sum_{j=0}^{30} .$$

Where $(c_0, c_1, \ldots, c_{31})$ represent the bits. If we add two numbers and get a result larger than $2^31 - 1$, the computer will give us the result Inf.

When we consider rational numbers, we write them as

$$x = (-1)^s \times [1.f]_2 \times 2^m.$$

We use a single bit to encode the sign, eight bits to encode m (also known as the exponent) and the remaining 32 bits to encode f (also known as the normalized mantissa). We call $[1.f]_2$ the mantissa. This method is known as single precision.

In double precision (which is the standard in Matlab) we use 64 bits to store each number. A single bit to encode the sign, 11 bits to encode the exponent and the remaining 52 bits to encode the normalized mantissa. In general we only focus on single precision.

For now, since

Denote $\lfloor x \rceil$ the closest number we can represent using this method. Then the absolute error is

$$\big|\lfloor x \rceil - x\big| \le 2^m 2^{-24}$$

and the relative error is bounded by

$$\frac{\big|\lfloor x \rceil - x\big|}{|x|} \le \frac{2^m 2^{-24}}{[1.f]_2 \times 2^m} \le 2^{-24}.$$

This means that the relative error of rounding is pretty low. In double precision because we have more bits the relative error is bounded by $2^{-52}\tilde{2} \times 10^{-16}$. This number is called *eps* in Matlab.

Definition 2.1 (Relative error)**.** Let $f(x)$ be some approximation of $x \neq 0$, the relative error is defined by

$$\frac{|f(x) - x|}{|x|} < \epsilon$$

if and only if

$$f(x) = x(1 + \delta)$$

for some $\delta \in (-\epsilon, \epsilon)$. We can get this equlity by setting $\delta = \frac{f(x)-x}{x}$.

Proposition 2.3. *Let $x_1, x_2 > 0$ and assume there exists some $f\colon \mathbb{R} \to \mathbb{R}$ such that there exist δ_1, δ_2, δ_3 such that*

$$f(x_1) = (1 + \delta_1)x_1 \text{ and } f(x_2) = (1 + \delta_2)x_2 \text{ and } f(f(x_1) \cdot f(x_2)) = (1 + \delta_3)[f(x_1) \cdot f(x_2)]$$

where $|\delta_1|, |\delta_2|, |\delta_3| < \epsilon$. Then

$$f(f(x_1) \cdot f(x_2)) = (1 + \delta)x_1 x_2$$

For some $|\delta| < 3\epsilon + 3\epsilon^2 + \epsilon^3$.

Proof.

$$f(f(x_1) \cdot f(x_2)) = (1+\delta_3)[f(x_1) \cdot f(x_2)] = (1+\delta_3)(1+\delta_2)(1+\delta_1)x_1x_2$$

and

$$(1+\delta_3)(1+\delta_2)(1+\delta_1) = 1+\delta_1+\delta_2+\delta_3+\delta_1\delta_2+\delta_1\delta_3+\delta_2\delta_3+\delta_1\delta_2\delta_3.$$

Denote

$$\delta = \delta_1+\delta_2+\delta_3+\delta_1\delta_2+\delta_1\delta_3+\delta_2\delta_3+\delta_1\delta_2\delta_3.$$

we get

$$|\delta| \leq 3\epsilon + 3\epsilon^2 + \epsilon^3$$

which completes the proof. □

The above theorem shows that when taking the product of two numbers, the error is bigger than epsilon, but not by much. The situation is similar for addition and division, but not for subtraction.

Consider

$$x_1 = [1.f0]_2 \text{ and } x_2 = [1.f6]$$

we have that $\lfloor x_1 \rfloor - \lfloor x_2 \rfloor = 0$ which implies that relative error is

$$\frac{\Big| \big\lfloor \lfloor x_1 \rfloor - \lfloor x_2 \rfloor \big\rfloor - |x_1 - x_2| \Big|}{|x_1 - x_2|} = 1.$$

This occurs when the distance between x_1 and x_2 is very small compared to x_1.

Proposition 2.4. *Assume that $x_2 > x_1 > 0$ and $\frac{x_2}{x_2 - x_1} < a$ for some a. Let $f \colon \mathbb{R} \to \mathbb{R}$ be a function satisfiying*

$$f(x_1) = (1+\delta_1)x_1 \text{ and } f(x_2) = (1+\delta_2)x_2 \text{ and } f(f(x_1) - f(x_2)) = (1+\delta_3)[f(x_1) - f(x_2)]$$

where $|\delta_1|, |\delta_2|, |\delta_3| < \epsilon$. Then

$$f(f(x_1) - f(x_2)) = (1+\delta)(x_2 - x_1)$$

where $|\delta| < 2\epsilon + 2a\epsilon + \epsilon^2 + 2a\epsilon^2$.

Proof. The proof is simply by calculation □

Because of the propositions we have seen so far, some ways to compute certain expressions are more accurate than others. For example instead of computing

$$x - \sin(x) \quad \text{or} \quad \sqrt{x^2+1} - 1$$

to get a more accurate calculation we can calculate the equivalent expressions

$$-\frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} \cdots \quad \text{and} \quad \frac{x^2}{\sqrt{x^2+1}+1}.$$

As mentioned earlier, in single precision we can not represent numbers bigger than 2^{128} or smaller than 2^{-128}. Instead the computer will return us the values ∞ and $-\infty$ respectively. This is called *overflow*. We also can not represent numbers in the range $(-2^{-128}, 2^{-128})$. Instead the computer will return us the values 0 or -0 if the number was negative. This is called *underflow*.

Definition 2.2 (Softmax function)**.** For a fixed $\alpha \geq 0$, the softmax function $s^\alpha \colon \mathbb{R}^d \to \mathbb{R}^d$ is defined as

$$s_i^\alpha := \frac{e^{\alpha x_i}}{\sum_{j=1}^d e^\alpha x_j}.$$

Remark 2.2. For all $x \in \mathbb{R}^d$ and $\alpha \geq 0$ the entries of $s^\alpha(x)$ are all positive and sum to one.

Remark 2.3. If k is some index such that $x_k > x_i$ for all $i \neq k$, then

$$\lim_{\alpha\infty} s6\alpha(x) = e_k$$

where e_k is the unit vector with 1 in the kth index and 0 otherwise. Thus, for large α softmax gives a strong idication of where the maximum of the vector is located.

Remark 2.4. If $\alpha = 0$ then $s^0(0) = (1/d, 1/d, \dots, 1/d)$ for all x.

This function is intresting because although the entries in the result vectors are all between 0 and 1, the exponentiation can lead to underflow and overflow which will greatly affect its computation.

To avoid overflow when computing softmax we can notice that for any scalar M and vector $1_d = (1, 1, \dots, 1)$ we have that $s^\alpha(x) = s^\alpha(x - M1_d)$. This allows use to choose $M = \max\{x_i\}_i$ and compute $s^\alpha(x - M1_d) = s^\alpha(x)$ without overflow.

2.1 Alternatives for symbolic computation*

The method we have used so far to calculate nubmbers is called numercial computation. Because of the way we represent numbers in (floating points) we always get an error bounded by 2^{-24}. There are some alternatives to numerical calculation.

One alternative for numerical calculations is *symbolic calculations.* This method is used by Mathematica, Maple, Wolfram Alpha and can also be used in Matlab. It has two main behaviours

- When multiplying numbers, the numbers of digits increase to avoid truncation.
- Solutions of equations can be given symbolically like $\sqrt{2}$, π etc.

Using this method we gain more accuracy, but lose speed of computation.

Another alternative is called *interval arithmetics.* Since our calculations are not exact we can represent numbers using intevals. For example, let $x = [1.f]_2$ such that $f = f_1 \circ f_2$. where f_1 is the first 23 digits of f and f_2 is the rest of the digits. In this case, we use the interval $I_x = [[1.f_1]_2, [1.f_1 1]_2]$ to represent x because $x \in I_X$.

Suppose we have two numbers $x = [a_x, b_x]$ and $y = [a_y, b_y]$, when calculating the sum $x+y$ we get $x+y = [a_x + a_y, b_x + b_y]$. In this way we can be certain that $x+y \in I_{x+y} = [a_x + a_y, b_x + b_y]$. We define multiplication in a similar, though more complicated way. When tying to define multiplication for intercal arithmetics, one could use the following proposition.

Proposition 2.5. *The minimum and maximum of the function $f(x,y) = xy$ over the rectangle $[a_x, b_x] \times [a_y, b_y]$ are obtained at the corners.*

3 Solving non-linear equations

From now on, we assume we can store numbers without inaccuracies, and perfrom arithmetics without errors.

3.1 Bisection method

Suppose we wanted to find a solution to the equation

$$\sin(x) = e^x.$$

Then from analysis courses we know to approximate the solution by changing the equaition to standard form

$$f(x) = e^x - \sin(x)$$

and then find an interval like $[a, b] = [-\frac{3\pi}{2}, 1]$ for which $f(a) < 0 < f(b)$. Because then we know that there exists a solution $x \in [a, b]$.

The bisection method allows us to find roots of continuous functions in intervals $[a, b]$ such that $\text{sign} f(a) \neq \text{sign} f(b)$. It works by repeating the following process

(-1) Set $a_0 = a$, $b_0 = b$ and $c_0 = \frac{a_0+b_0}{2}$.

(n) Assume a_n, b_n, c_n are defined. If $f(c_n) = 0$ we have found a root. Otherwise, from the pigeonhole principle there exists $x \in \{a_n, b_n\}$ such that $f(c_n)$ and $f(x)$ form an interval that contains 0. We call this interval $[a_{n+1}, b_{n+1}]$. We set $c_{n+1} = \frac{a_{n+1}+b_{n+1}}{2}$. Treating n as a variable for formality, we increment it by one, and continue to the next step (step (n)).

Setting $|a - b| = L$ we get that in the $|a_n - b_n| = L2^{-n}$. By Cantor's intersection theorem, and the squeeze theorem, it is clear that $c_n \xrightarrow{n\to\infty} x$ where x is a root of $f(x)$.

Remark 3.1. Using the bisection method, we can calculate the number of steps we need to take to obtain x' where $x' \in [x - \epsilon, x + \epsilon]$ for any $\epsilon > 0$.

3.2 Newton's algorithm

Newton's algorithm (also known as the Newton–Raphson algorithm) is an algorithm to compute the root of a function $f : \mathbb{R} \to \mathbb{R}$. We will assume $f \in C^2(\mathbb{R})$ and that f has a root x.

The idea behind the algorithm is to have a first guess of the root x_0, and then set x_{n+1} to be the root of the Taylor expansion up to linear order of f at the point x_n.

This algorithm does not always converge to the root, but if we have a good approximation for the root as x_0 it will always converge to the root very quickly.

Example 3.1. Consider the function $f(x) = e^x - 6$. The solution is $x = \log 6 \approx 1.7918$. We start by a close guess $x_0 = 2$. The iterations are given by

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = x_n - \frac{e^{x_n} - 6}{e^{x_n}}.$$

Applying this process gives

$$x_1 = 1.8120 \text{ and } x_2 = 1.7920 \text{ and } x_3 = 1.7918 \text{ and } x_4 = 1.7918$$

with errors

$$e_1 = 2 \times 10^{-2} \text{ and } e_2 = 2 \times 10^{-4} \text{ and } e_3 = 2 \times 10^{-8} \text{ and } e_4 = 4 \times 10^{-16}.$$

which shows that the process is very useful.

Lemma 3.1. *Let $f \in C^2(\mathbb{R})$ be a function with a root $r \in \mathbb{R}$. Let x_{n+1} be the solution of $L(x \mid x_n) = 0$ which is the root of the linear Taylor approximation of $f(x)$ at x_n. Then there exists $\xi \in (r, x_n)$ such that*

$$f'(x_n)(x_{n+1} - r) = \frac{1}{2} f''(\xi)(x_n - r)^2$$

Proof. By definition we have $L(x_{n+1} \mid x_n) = 0$, and Taylor's expansion around x_n gives us

$$0 = f(r) = L(r \mid x_n) + \frac{1}{2} f''(\xi)(x_n - r)^2$$

so

$$f'(x_n)(x_{n+1} - r) = L(x_{n+1} \mid x_n) - L(r \mid x_n) = L(r \mid x_n) = \frac{1}{2} f''(\xi)(x_n - r)^2$$

which completes the proof. □

Proposition 3.2. *Let $f \in C^2(\mathbb{R})$ be a function with a simple root $r \in \mathbb{R}$. Then there exists $\delta > 0$ and $C > 0$ such that for every Newton iterations starting from $x_0 \in [r - \delta, r + \delta]$, the sequence $|x_n - r|$ converges monotonely to zero and*

$$|x_{n+1} - r| \leq C(x_n - r)^2$$

Remark 3.2. A simple root means that $f'(r) \neq 0$.

Proof. Let x_n be the sequence Newton–Raphson iterates starting from x_0. Since $f'(r) \neq 0$, we can choose some $\delta_0 > 0$ small enough so that $f'(x) \neq 0$ on $U = [r - \delta_0, r + \delta_0]$. Set

$$m = \min_{x \in U} |f'(x)| \text{ and } M = \max_{x \in U} |f''(x)|$$

then using Lemma 3.1 we have that whenever TO BE CONTINUED □

The convergence rate of Newton's method near a minimun is qaudratic. This means that

$$|x_{n+1} - r| \leq C(x_n - r)^2.$$

In comparison, the convergence rate of the bisection method was linear

$$|x_{n+1} - r| \leq \frac{1}{2}|x_n - r|.$$

Definition 3.1 (Convergence rate)**.** Let x_n be a sequence converging to L. We say that x_n has a convergence rate $\alpha > 0$ if there exists $C > 0$ and $N > 0$ such that for all $n < N$ we have

$$|x_{n+1} - L| \leq C|x_n - L|^\alpha$$

Theorem 3.3. *Assume $f \in C^2(\mathbb{R})$ is strictly increasing, strictly convex, and has a root. Then the root is unique, and Newton iterations will converge to it from any starting point.*

Proof. It is clear that the root is unique. From strictly convex, and strictly increasing we have that $f''(x) > 0$ and $f'(x) > 0$ for all x. From Lemma 3.1 we see that $x_n - r > 0$ for all $n \geq 1$. Since f is increasing it follows that $f(x_n) > f(r) = 0$. Since

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

we see that $r < x_{n+1} < x_n$, so x_n is monotonically decreasing and bounded below, thus it converges to some limit L. Since $f \in C^2(\mathbb{R})$ we have that

$$L = \lim_{x \to \infty} x_{n+1} = \lim_{n \to \infty} x_n - \frac{f(x_n)}{f'(x_n)} = L - \frac{f(L)}{f'(L)}$$

which means that $f(L) = 0$ and thus $L = r$ which completes the proof. □

We can also generalize Newton's method to higher dimensions.

Let $f \colon \mathbb{R}^d \to \mathbb{R}^d$ be a function such that $f \in C^2\mathbb{R}^d$ and f has a root $r \in \mathbb{R}^d$. Given a point $x_n \in \mathbb{R}^d$, we can define x_{n+1} as the solution to the linear approximation of f around x_n,

$$L(x \mid x_n) = f(x_n) + D_f(x_n)(x - x_n).$$

The solution for the equation $L(x \mid x_n) = 0$ is

$$x_{n+1} = x_n - D_f^{-1}(x_N) f(x_n)$$

The solution to the equation $L()$