Introduction to Numerical Analysis

Based on lectures by

Notes taken by yehelip

Winter 2025

These notes are not endorsed by the lecturers. I have revised them outside lectures to incorporate supplementary explanations, clarifications, and material for fun. While I have strived for accuracy, any errors or misinterpretations are most likely mine.

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

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

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

$$\begin{aligned} a_k n^k + \cdots + a_1 n + a_0 &= \Theta(n^k) \\ Cn^3 + n - 1 &= \Theta(n^3) \\ n \cdot n! + n! - 1 &= \Theta(n \cdot n!) \end{aligned}$$

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

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

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 $\{0, 1, \ldots, b-1\}$, 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 $c_L \in \{0, 1, \ldots, b-1\}$ such that $x - c_L b^L \in [0, b^L)$.*

Proof. We notice that

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

So there exists $c_L \in \{0, 1, \ldots, b-1\}$ 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 $xb^{-L} < 1$. 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. Obsviouly 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:

$$n = (-1)^{c_3 1} \sum_{j=0}^{30}.$$

Where $(c_0, c_1, \ldots, c_{31})$ represent the bits. If we add two numbers and get a result larger than $2^3 1 - 1$, 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 $\lfloor x \rfloor$ the closest number we can represent using this method. Then the absolute error is

$$\left| \lfloor x \rfloor - x \right| \leq 2^m 2^{-24}$$

and the relative error is bounded by

$$\frac{\left| \lfloor x \rfloor - x \right|}{|x|} \leq \frac{2^m 2^{-24}}{[1.f]_2 \times 2^m} \leq 2^{-24}.$$

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. □