

Introduction to Numerical Analysis

Notes taken by Yeheli Fomberg <yp@campus.technion.ac.il>

Based on lectures by

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	3
2	Digital Number Representation	4
2.1	Representing numbers in the computer	5
2.2	Alternatives for symbolic computation*	8
3	Solving non-linear equations	9
3.1	Bisection method	9
3.2	Newton's algorithm	10
3.3	Generalization to higher dimensions	11
3.4	Newton's method with the contractive mapping theorem	12
3.5	Newton-like method without derivatives*	12
3.6	Polynomials the their roots	12
3.7	Bairstow's method*	12
3.8	Homotopy method*	12
4	Numerical linear algebra	13
4.1	Solving linear equations	13
4.1.1	Guassiam elimination	14
4.2	Scaled row pivoting	15
4.3	Condition number	15
4.4	Pure LU and Cholesky decomposition	16
4.5	The power method	17
4.6	The power method beyond a singlue eigenvalue	18
4.7	Laplacian matrix	19

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:

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

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 constant C . Although this algorithm is much faster than the previous ones, there are more efficient algorithms still.

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

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

Here are some examples

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

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

However, the big $\mathcal{O}$ notation is still not very satisfying because we still have

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

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

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

Example 1.1 (Horner's algorithm). Another important method before we move to the next section, is Horner's method. It allows computing a polynomial $p(x) = \sum_{k=0}^n a_k x^k$ 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 $\{0, 1, \dots, b-1\}$, 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

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

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 Theorem 2.2 with $L = -1$ we now define:

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

And continue indefinitely

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

This implies that $r_S \xrightarrow{S \rightarrow \infty} 0$ 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. Obviously we can also represent 0 because $0 \in \{0, 1, \dots, b\}$.

2.1 Representing numbers in the computer

Definition 2.1 (Integers). 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:

$$n = (-1)^{c_{31}} \sum_{j=0}^{30} c_j 2^j.$$

Where $(c_0, c_1, \dots, c_{31})$ 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*.

Definition 2.2 (Fixed point representation). One (uncommon) way of representing rational numbers with 32 bits is called fixed point representation. In this method we allot the last bit for the sign, and we write the numbers as such

$$x = (-1)^{c_{31}} (c_{30}, \dots, c_{15}, c_{14}, \dots, c_1, c_0)_2 = (-1)^{c_{31}} (c_{30}2^{15} + c_{29}2^{14} + \dots + c_{15} + \dots + c_02^{-15}).$$

Definition 2.3 (Floating point). The common way to represent rational numbers in the computer is by using the floating point format. In this method we write rational numbers 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.

Precise details about the implementation of special number like ± 0 , $\pm \infty$ or NaN are the in solutions.

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.

The largest floating point number in single precision is

$$2^{127} \leq x = (1.11 \dots 1) \times 2^{127} \leq 2^{128}.$$

Definition 2.4 (Machine number). Any number that can be represented in the computer exactly is called a machine number. In most cases we refer to numbers that can be represented using the floating point method.

Denote $\lfloor x \rfloor$ or $\text{fl}(x)$ the closest machine number to x .

Definition 2.5 (Absolute error). The absolute error is defined as the value of $|\text{fl}(x) - x|$ and we can see that in the case of a floating point that

$$|\text{fl}(x) - x| \leq 2^m 2^{-24}$$

Definition 2.6 (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 equality by setting $\delta = \frac{f(x) - x}{x}$.

Remark 2.2. The relative error in the case of a floating point approximation satisfies

$$\frac{|\text{fl}(x) - x|}{|x|} \leq \frac{2^m 2^{-24}}{[1.f]_2 \times 2^m} \leq 2^{-24}.$$

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 $2^{-52} \sim 2 \times 10^{-16}$. This number is called *eps* in Matlab.

Definition 2.7 (Unit roundoff error). The unit roundoff error in a machine, is defined to be the minimal upper bound of the relative error. In single precision, the unit roundoff error is 2^{-24} .

Definition 2.8 (Machine precision, machine epsilon). The machine precision (or machine epsilon) is defined to be the distance between the number 1 and the immediate following machine number. In single precision, the machine epsilon is 2^{-23} . That is because

$$(1.00 \dots 01)_2 = 1 + 2^{-23}.$$

The closest number from below however is

$$(1.11 \dots 11)_2 \times 2^{-1} = 1 - 2^{-24}.$$

In double precision the machine epsilon drops to 2^{-52} .

Proposition 2.3. Let $x_1, x_2 > 0$ and assume there exists some $f: \mathbb{R} \rightarrow \mathbb{R}$ such that there exist $\delta_1, \delta_2, \delta_3$ 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

$$f(f(x_1) \cdot f(x_2)) = (1 + \delta)x_1x_2$$

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. $\square$

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

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

we have that $[x_1] - [x_2] = 0$ which implies that relative error is

$$\frac{|[x_1] - [x_2]|}{|x_1 - x_2|} = 1.$$

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 $f: \mathbb{R} \rightarrow \mathbb{R}$ be a function satisfying

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

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 2.9 (Softmax function). For a fixed $\alpha \geq 0$, the softmax function $s^\alpha: \mathbb{R}^d \rightarrow \mathbb{R}^d$ is defined as

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

Remark 2.3. 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 2.4. If k is some index such that $x_k > x_i$ for all $i \neq k$, then

$$\lim_{\alpha \rightarrow \infty} s^\alpha(x) = e_k$$

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

Remark 2.5. If $\alpha = 0$ then $s^0(0) = (1/d, 1/d, \dots, 1/d)$ for all x .

This function is interesting 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 us to choose $M = \max\{x_i\}_i$ and compute $s^\alpha(x - M1_d) = s^\alpha(x)$ without overflow.

Softmax is useful in machine learning in for *classification tasks*. For example, suppose you have d different kanji characteres registered in some database by order of their JIS code. We want to build a function h that assigns each vector X representing a picture of a single handwritten character a probability vector ¹ $(h_1(X), \dots, h_d(X))$ so that $h_i(X)$ represents the

¹A vector with nonnegative entries that sum up to one.

probability of the picture to describe the i th character. The standard way of doing this is by defining

$$h(X) := s^\alpha A_2 \sigma A_1(X),$$

where A_1, A_2 are matrices, σ is a nonlinear transformation, and s^α is the softmax function.

This is a simple neural network, where the parameter α controls how "sharp" the probability vector $h(X)$ will be.

The real challenge is how to define h well, but this is beyond the scope of this lecture. Instead we can notice that underflow doesn't really affect softmax, and while overflow does lead to complete nonsense, we can make sure it never happens. More on that in the solutions section.

2.2 Alternatives for symbolic computation*

The method we have used so far to calculate numbers is called numerical 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 intervals. 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 $I_x = [[1.f_1]_2, [1.f_11]_2]$ 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 trying to define multiplication for interval 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 perform 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 equation 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 $|a_n - b_n| = L2^{-n}$. By Cantor's intersection theorem, and the squeeze theorem, it is clear that $c_n \xrightarrow{n \rightarrow \infty} x$ 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 $\epsilon > 0$.

Here is pseudocode for the bisection method, f represents a continuous function, a , b represent the limits of the interval $[a, b]$ where the root is. Tolerance is the word used for the accuracy we need, represented by ϵ . We also need to have a maximum amount of iterations denoted $N_{\max}$.

Input: f , a , b , ϵ , $N_{\max}$

Output: Approximate root c or failure to converge

if $f(a) \cdot f(b) \geq 0$ **then**

 Print "Error: No root in the interval $[a, b]$ ";

return

end

for $k \leftarrow 1$ **to** $N_{\max}$ **do**

$c \leftarrow \frac{a+b}{2}$;

if $f(c) = 0$ **or** $\frac{b-a}{2} \leq \epsilon$ **then**

return c ;

end

if $f(a) \cdot f(c) < 0$ **then**

$b \leftarrow c$;

end

else

$a \leftarrow c$;

end

end

Print "Failed to converge after $N_{\max}$ iterations";

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 $f: \mathbb{R} \rightarrow \mathbb{R}$ so that $f \in C^2(\mathbb{R})$ and 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 .

Remark 3.2. 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. We will see what “good” exactly means later.

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 $L(x | x_n) = 0$ (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 $L(x_{n+1} | x_n) = 0$, and Taylor's expansion around x_n gives us

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

so

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

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 any 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 3.3. 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 Theorem 3.1 we have that whenever

to be added □

Remark 3.4. The convergence rate of Newton's method near a minimum is quadratic. 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 Theorem 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

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

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

3.3 Generalization to higher dimensions

Let $f: \mathbb{R}^d \rightarrow \mathbb{R}^d$ 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 ,

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

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

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

Given that D_f is non-singular, we get a quadratic convergence rate.

Remark 3.5. While the bisection method allowed linear convergence under very minimal convergence, Newton's method allows (at least ²) quadratic convergence, and it can also be generalized to higher dimensions.

²See solutions for cubic convergence of Newton's method in the solutions section

3.4 Newton's method with the contractive mapping theorem

The iterations in Newton's method are given by

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

Define a function F as follows

$$F(x) = x - \frac{f(x)}{f'(x)}.$$

The function is well defined for points where the derivative of f doesn't vanish, and also satisfies

$$x_{n+1} = F(x_n) \quad \text{and} \quad F(r) = r \iff f(r) = 0.$$

Algorithms that compute sequences like $x_{n+1} = F(x_n)$ aim to find a fixed point of F at the limit. These algorithms are called fixed point iterations. One of the most helpful theorems when dealing with these algorithms, is the contractive mapping theorem.

Theorem 3.4 (Contractive mapping theorem). *Let (X, d_X) be a complete metric space and let $F: X \rightarrow X$ be a contraction with contraction factor λ . Then F has a unique fixed point $r \in X$. Moreover, every sequence $\{x_n\}_{n=1}^\infty \subseteq X$ of the form*

$$x_{n+1} = F(x_n)$$

converges to r at least linearly. Note that when working with an interval, the image of $[a, b]$ must be contained in $[a, b]$.

Proof. To be added. □

Example 3.2 (Gradient descent). Another fixed point iteration method is the gradient descent method. Assume that $f: \mathbb{R} \rightarrow \mathbb{R}$ is an element of $C^2(\mathbb{R})$, and let r be a strict minimum ($f'(r) = 0$ and $f''(r) > 0$). The iterations in the gradient descent method are defined as such

$$F(x_n) = x_n - \alpha f'(x_n)$$

where $\alpha > 0$ is called the *step-size*.

Remark 3.6. Each step we “descend” in the direction of the derivative.

Remark 3.7. This method also works in higher dimensions by changing the iterations to

$$F(x_n) = x_n - \alpha \nabla f(x_n).$$

Example 3.3. Assume we want to minimize the function $f(x) = x^2$. The iterations given by the gradient descent method is

$$x_{n+1} = x_n - \alpha 2x_n.$$

We need to understand for which values these iterations converge.

Proposition 3.5. *Let r be a strict minimum of $f \in C^2(\mathbb{R})$. Then for all small enough $\alpha > 0$ there exists $\delta > 0$ such that the gradient descent iterations converge linearly to the minimum r for any initial $x_0 \in (r - \delta, r + \delta)$.*

Proof. □

3.5 Newton-like method without derivatives*

3.6 Polynomials and their roots

3.7 Bairstow's method*

3.8 Homotopy method*

4 Numerical linear algebra

4.1 Solving linear equations

In this subsection, we will focus on solving a system of linear equations such as

$$Ax = b$$

where $A \in \mathbb{R}^{n \times n}$ is non-singular.

- When A is diagonal we can find a solution in $\mathcal{O}(n)$ operations.
- When A is upper triangular or lower triangular we can find a solution in $\mathcal{O}(n^2)$ operations.

Moreover, if $A = LU$ where L is a lower triangular matrix, and U is an upper triangular matrix we can solve $Ax = b$ by first solving $Ux = z$ then solving $Lz = b$. Since both of the solutions can be found at complexity $\mathcal{O}(n^2)$, the original system $Ax = b$ can be solved in $\mathcal{O}(n^2)$ operations as well.

Definition 4.1 (Permutation matrix). A permutation $\sigma: \{1, \dots, n\} \rightarrow \{1, \dots, n\}$ induces a linear map $\hat{\sigma}: \mathbb{R}^n \rightarrow \mathbb{R}^n$ by

$$[\hat{\sigma}(v)]_{\sigma(i)} = v_i$$

for all $1 \leq i \leq n$. The matrix representing the linear operator $\hat{\sigma}$ is the matrix P_σ defined by

$$P_{i,j} := \begin{cases} 1, & i = \sigma(j) \\ 0, & \text{otherwise} \end{cases}.$$

Remark 4.1. When a permutation matrix P_σ multiplies the matrix A from the left, the product $P_\sigma A$ is given by

$$\begin{pmatrix} - & R_{\sigma(1)} & - \\ & \vdots & \\ - & R_{\sigma(n)} & - \end{pmatrix}$$

where R_i is the i th row of A .

Remark 4.2. Every permutation matrix P is orthogonal, with its inverse equal to its transpose. Moreover, permutation matrices can be characterized as the orthogonal matrices whose entries are all non-negative.

Remark 4.3. We denote $L^{(r,s,\lambda)}$ the matrix that operates on a matrix A by adding the r th row of A times λ to the s th row of A . Formally,

$$L^{(r,s,\lambda)} = I_n + \lambda E^{sl}$$

where $E_{sl}^{sl} = 1$ and otherwise $E_{ij} = 0$.

Lemma 4.1. Let P_σ be a permutation matrix which swaps the rows t and u and leaves the rest of the rows fixed. Let $L^{r,s,\lambda}$ be with $r > s$, and assume that $r < \min\{u, t\}$. Then

$$PL^{r,s,\lambda} = L^{r,\sigma(s),\lambda}P$$

and $\sigma(s) > r$.

Proof. To be added □

4.1.1 Guassiam elimination

The process of Guassiam elimination on A can be given by first multiplying $A_0 = A$ by a permutation matrix such that the element in $(P_0 A_0)_{11} \neq 0$ and then multiplying $P_0 A_0$ on the left by $n - 1$ lower triangular matrices L^{1,j,λ_j} and get a matrix

$$A_1 = L^{1,n,\lambda_n} \dots L^{1,2,\lambda_2} P_0 A_0.$$

Denote

$$L^k := L^{k,n,\lambda_n} \dots L^{1,k+1,\lambda_{k+1}}$$

and then by the end of the elimination we have

$$U = A_{n-1} = L_{n-1} P_{n-2} L^{n-2} P_{n-3} \dots L^1 P_0 A.$$

where U is an upper triangular matrix since it is after the elimination.

Theorem 4.2 (*LU decomposition*). *Let $A \in \mathbb{R}^{n \times n}$ be a non-singular matrix, let $P_0, \dots, P_{n-2}$ be the permutation matrices from above, let $L^{n-1}, \dots, L^1$ be the lower triangular matrices from above, and let U be the upper triangular matrix obtained from applying Gaussian elimination on A . Then*

$$PA = LU$$

where L is some triangular matrix, and P is the permutation matrix $P = P_{n-2} \dots P_0$.

Proof. Each P_k is a permutation matrix P_{σ_k} that permutes two rows u and t , and each L^r for $r \leq k$ is a product of matrices of the form $L^{r,s,\lambda}$. According to Theorem 4.1 these types of matrices commute (with a slight modification) and thus

$$\begin{aligned} P_k L^r &= P_k L^{r,n,\lambda_n} \dots L^{1,r+1,\lambda_{r+1}} \\ &= L^{r,\sigma_k(n),\lambda_n} \dots L^{1,\sigma_k(r+1),\lambda_{r+1}} P_k \\ &= L^{r,n,\lambda_n} \dots L^{1,\sigma_k(r+1),\lambda_{r+1}} P_k \end{aligned}$$

□

Remark 4.4. The complexity to find the LU decomposition of a matrix is $\mathcal{O}(n^3)$, but this gives us $A = P^T LU$, then we can solve $Ax = P^T LUx = b$ in complexity $\mathcal{O}(n^2)$.

This might seem useless at first, because solving $Ax = b$ is still $\mathcal{O}(n^3 + n^2) = \mathcal{O}(n^3)$. However, if A is fixed and we want to find the solutions $Ax = b_i$ for $b_1, \dots, b_m$, the complexity of solving the equations without LU decomposition will be $\mathcal{O}(mn^3)$, and with LU decomposition $\mathcal{O}(n^3 + mn^2)$ which is faster in practice.

Example 4.1. We can use LU decomposition to find the inverse of A by solving the equations

$$Ac_j = e_j \quad j = 1, \dots, n$$

in $\mathcal{O}(n^3 + mn^2)$ and then

$$A^{-1} = \begin{pmatrix} | & | & | \\ c_1 & \dots & c_n \\ | & | & | \end{pmatrix}.$$

Example 4.2. We can use LU decomposition to compute the determinant of A with

$$\det(A) = (\det(P))^{-1} \det(L) \det(U).$$

This is very efficient because we can calculate the determinants of permutations matrices and triangular matrices in linear time.

4.2 Scaled row pivoting

Consider the equation

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}.$$

The solution is $(x, y) = (1, 1)$. However, if we modify the equation slightly with a very small $\epsilon > 0$

$$\begin{pmatrix} \epsilon & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

After applying Gaussian elimination we get

$$\begin{pmatrix} \epsilon & 1 \\ 1 & 1 - \epsilon^{-1} \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1 \\ 2 - \epsilon^{-1} \end{pmatrix}$$

and the solutions will be

$$\begin{aligned} x &= \epsilon^{-1} \left(1 - \frac{2 - \epsilon^{-1}}{1 - \epsilon^{-1}} \right) = -\frac{\epsilon^{-1}}{1 - \epsilon^{-1}} \approx 1 \\ y &= \frac{2 - \epsilon^{-1}}{1 - \epsilon^{-1}} \approx 1. \end{aligned}$$

However, when ϵ is small enough, we have that

$$\text{fl}(1 - \epsilon^{-1}) = \text{fl}(2 - \epsilon^{-1}) \implies x = 0$$

which then gives that the solution is $(x, y) = (0, 1)$ which is far from the correct solution. This shows us that when performing Gaussian elimination we need to have scaled row pivoting. With scaled row pivoting, for each row we set

$$s_i := \max \{|a_{i2}, \dots, |a_{in}||\}$$

and then choose the first row to be the one for which $|a_{i1}|/s_i$ is maximal in all the steps.

Using this approach in our previous example gives

$$\begin{pmatrix} 1 & 1 \\ \epsilon & 1 \end{pmatrix} \begin{pmatrix} y \\ x \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \end{pmatrix} \implies \begin{pmatrix} 1 & 1 \\ 0 & 1 - \epsilon \end{pmatrix} \begin{pmatrix} y \\ x \end{pmatrix} = \begin{pmatrix} 2 \\ 1 - 2\epsilon \end{pmatrix}$$

and then even for a very small $\epsilon > 0$ we have that $\text{fl}(1 - \epsilon) = 1$ and $\text{fl}(1 - 2\epsilon) = 1$, which leads to $(x, y) = (1, 1)$ which is very close to the desired solution.

4.3 Condition number

Question. We know that we can solve $Ax = b$ when $\det(A) \neq 0$ but what happens when $\det(A)$ is very small?

Answer. The solution will be very unstable.

Definition 4.2 (Condition number). The condition number of a non-singular matrix A is defined as

$$\kappa(A) := \|A\| \cdot \|A^{-1}\|.$$

Lemma 4.3. Let A be a non-singular matrix. Then $\kappa(A) > 1$.

Proof.

$$1 = \|AA^{-1}\| \leq \|A\| \cdot \|A^{-1}\| = \kappa(A).$$

□

Remark 4.5. Since we are solving equations numerically we only get approximate solutions $\hat{x} \approx x$. Denote $\hat{b} = A\hat{x}$.

Theorem 4.4. Let $A \in \mathbb{R}^{n \times n}$ be non-singular, and let $x, \hat{x}, b, \hat{b}$ be nonzero vectors in $\mathbb{R}^n$ such that

$$Ax = b \quad \text{and} \quad A\hat{x} = \hat{b}$$

then

$$\kappa^{-1}(A) \frac{\|b - \hat{b}\|}{\|b\|} \leq \frac{\|x - \hat{x}\|}{\|x\|} \leq \kappa(A) \frac{\|b - \hat{b}\|}{\|b\|}.$$

Proof. We have that

$$\begin{aligned} \|x - \hat{x}\| &= \|A^{-1}(b - \hat{b})\| \leq \|A^{-1}\| \|b - \hat{b}\| \\ &= \|A^{-1}\| \|b - \hat{b}\| \|Ax\| \|b\|^{-1} \leq \kappa(A) \|b - \hat{b}\| \|x\| \|b\|^{-1}. \end{aligned}$$

Similarly since $x = A^{-1}b$ and $\hat{x} = A^{-1}\hat{b}$ we get

$$\frac{\|b - \hat{b}\|}{\|b\|} \leq \kappa(A^{-1}) \frac{\|x - \hat{x}\|}{\|x\|} = \kappa(A) \frac{\|x - \hat{x}\|}{\|x\|}$$

□

Theorem 4.5. Let $A \in \mathbb{R}^{n \times n}$ be a non-singular matrix with an orthonormal basis $\mathcal{B} = \{v_i\}_{i=1}^n$ and eigenvalues $\Lambda = \{\lambda_i\}_{i=1}^n$. Then

$$\kappa(A) = \frac{\min \Lambda}{\max \Lambda}.$$

Proof. Since each vector can be decomposed to a linear combination of the basis vectors, we get that $\|A\| = \|A\|_{\text{op}} = \max \Lambda$. Similarly, since the eigenvalues of A^{-1} are λ_i^{-1} we get that $\|A^{-1}\| = 1/\min \Lambda$. It follows that

$$\kappa(A) = \frac{\min \Lambda}{\max \Lambda}$$

which completes the proof. □

4.4 Pure LU and Cholesky decomposition

There are some types of matrices such that we do not need any pivoting.

Theorem 4.6. Let $A \in \mathbb{R}^{n \times n}$ be a matrix whose principle minors $A_k \in \mathbb{R}^{k \times k}$ are all non-singular. Then there exist a lower triangular matrix L and an upper triangular matrix U such that $A = LU$ and the diagonal entries of L are 1.

Proof. To be added □

4.5 The power method

Example 4.3. Consider the matrix

$$A = \begin{pmatrix} -2 & 0 \\ 0 & 1 \end{pmatrix}.$$

It clearly has the eigenvalues $\lambda_1 = -2$ and $\lambda_2 = 1$. Given an arbitrary vector $x = (x_1, x_2)$ we get

$$A^k x = \begin{pmatrix} (-2)^k x_1 \\ x_2 \end{pmatrix}$$

which in a sense converges to the eigenvector corresponding to λ_1 (the vector e_1).

In order to see this more clearly, we first need to normalize the vector by defining recursively

$$x^{(k)} = \frac{A^k x}{\|A^k x\|}.$$

We see that $x^{(2k)}$ converges to e_1 , and $x^{(2k+1)}$ converges to $-e_1$, which are scalar multiples of each other. Either way we have that

$$\|Ax^{(k)} - \lambda_1 x^{(k)}\| \xrightarrow{k \rightarrow \infty} 0.$$

Let v_1 be the eigenvector of λ_1 . Since for any vector $w \in \mathbb{R}^2$ such that $\langle v_1, w \rangle \neq 0$ we get

$$\frac{\langle Av_1, w \rangle}{\langle v_1, w \rangle} = \lambda_1$$

we can also approximate it by

$$\frac{\langle Ax^{(k)}, w \rangle}{\langle x^{(k)}, w \rangle} \xrightarrow{k \rightarrow \infty} \lambda_1$$

What we have seen in the previous example is called the power method. It states that given $A \in \mathbb{C}^{n \times n}$ and $x^{(0)}, w \in \mathbb{C}^n$, if we recursively define

$$x^{(k)} = \frac{Ax^{(k-1)}}{\|Ax^{(k-1)}\|}$$

$$\lambda^{(k)} = \frac{\langle Ax^{(k)}, w \rangle}{\langle x^{(k)}, w \rangle}$$

the sequence $x^{(k)}$ will converge to the “first” eigenvector under some conditions.

Remark 4.6. This means that $\lambda^{(k)}$ will necessarily converge to the corresponding eigenvalue.

Theorem 4.7. Let $n \geq 2$ be a natural number, let $A \in \mathbb{C}^{n \times n}$, let $w^{(0)}, w \in \mathbb{C}^n$, and assume that

- (1) A is diagonalizable.
- (2) A must have a dominant eigenvalue which we will denote λ_1 .
- (3) When $x^{(0)}$ is expressed as a linear combination

$$x^{(0)} = a_1 u_1 + a_2 u_2 + \cdots + a_n u_n$$

the coefficient a_1 is not zero.

$$(4) \langle w, u_1 \rangle \neq 0.$$

Then $x^{(k)}$ is well defined for all $k \geq 1$, and $\lambda^{(k)}$ is well defined for all sufficiently large k , and

$$\begin{aligned} \lambda^{(k)} &\xrightarrow{k \rightarrow \infty} \lambda_1 \\ \|Ax^{(k)} - \lambda_1 x^{(k)}\| &\xrightarrow{k \rightarrow \infty} 0. \end{aligned}$$

Remark 4.7. A dominant eigenvalue is an eigenvalue λ_1 such that for all $k \geq 2$ we have $|\lambda_1| > |\lambda_k|$. This is also the most restrictive condition, the rest of the conditions are usually satisfied. If the first two eigenvalues have similar magnitude the convergence will be very slow.

Proof. □

Remark 4.8. In linear algebra we saw that for two column vectors $x, y \in \mathbb{R}^{n \times 1}$ we have that the standard inner product is

$$\langle x, y \rangle = y^T x,$$

but we also find the *outer product* (or tensor product) useful, which is defined as

$$x \otimes y = xy^T \in \mathbb{R}^{n \times n}.$$

This is a linear transformation, such that

$$(x \otimes y)z = \langle y, z \rangle x.$$

In particular if v_1, v_2 are orthogonal, and of norm one,

$$(v_1 v_1^T) v_1 = v_1 \quad \text{and} \quad (v_1 v_1^T) v_2 = 0.$$

4.6 The power method beyond a single eigenvalue

Suppose we have a symmetric real matrix with eigenvalues such that

$$|\lambda_1| > |\lambda_2| > |\lambda_3| \geq \dots \geq |\lambda_n|.$$

After finding the first eigenvalue and eigenvector (λ_1, u_1) using the power method, we define $\tilde{A} = A - \lambda_1 u_1 u_1^T$. This gives that

$$(A - \lambda_1 u_1 u_1^T) u_j = \lambda_j u_j - \lambda_1 u_1 u_1^T u_j = \begin{cases} 0, & j = 1 \\ \lambda_j u_j, & j \neq 1 \end{cases}.$$

This way the dominant eigenvalue of $\tilde{A}$ is λ_2 , and in this way we can theoretically find all eigenvalues of A .

Remark 4.9. To find all eigenvalues of A we need to slightly modify the conditions so that

$$|\lambda_1| > |\lambda_2| > |\lambda_3| > \dots > |\lambda_n|.$$

Remark 4.10. In practice, the power method is used when we need to find only a few eigenvectors, in order to find all of them other method, such as the QR decomposition method are preferred.

Remark 4.11. In order to find the smallest eigenvalue, we need to assume that there exists a dominant eigenvalue, and a unique minimal eigenvalue, which are all positive. Then after finding λ_1 , we can define $\tilde{A} = \lambda_1 I_n - A$, which has the eigenvalues $\{\lambda_1 - \lambda_i\}_{i=1}^n$. The dominant eigenvalue is $\lambda_1 - \lambda_n$, so we can find it by using the power method on $\tilde{A}$. Since we already know λ_1 , this means we can find λ_n .

4.7 Laplacian matrix

Definition 4.3 (Adjacency matrix). Let $G = (V, E)$ be a simple graph. We define its adjacency matrix A by

$$(A)_{ij} = \begin{cases} 1, & \{i, j\} \in E \\ 0, & \{i, j\} \notin E \end{cases}.$$

Remark 4.12. The degree d_i of the i th vertex is given by

$$d_i = \sum_{j=1}^n A_{ij}.$$

Definition 4.4 (Laplacian matrix). Let $G = (V, E)$ be a graph, let A be its adjacency matrix, and let $\{d_i\}_{i=1}^n$ be the degrees of the vertices of G . The Laplacian matrix of G is

$$L = \text{diag}(d_1, \dots, d_n) - A.$$