Mathematics, Computational Methods and Signal Processing

A script for students of the TUM Department of Chemistry

Ilgis Ibragimov & Elena Ibragimova & AI*

September 2026
DOI: 10.5281/zenodo.23002397

Dedicated to our daughters Maria and Alevtina.
May mathematics become for you not a collection of formulas,
but the language in which the Universe is written.

*Editing and Translation Assistant

Contents

1 Preface

Dear daughters,

We so wanted everything in your life's path to be clear and simple. We know that mathematics is often taught by deliberately confusing the learner with terminology and complicated formulas — so that the forest cannot be seen for the trees.

This script is our attempt to give an overview of the mathematical and computational ideas that are especially useful for a modern chemist working with experimental data, modelling and computations, so that you have a *clear picture* of the modern level of mathematical knowledge. Not all knowledge — that is impossible in one book — but precisely that which is used in chemistry, biochemistry, spectroscopy, molecular modelling and data processing.

We have travelled a long way together:

- From the very foundations — sets of numbers, complex numbers, vectors and matrices,
- Through linear algebra — SVD, conditioning, methods for solving systems,
- To nonlinear problems — optimisation, gradients, Newton's method,
- Through signal processing — Fourier, Prony, compressed sensing,
- To numerical methods — integration, finite elements, basis functions,
- And finally — to machine learning, neural networks and modern computing hardware.

All of this is not just a collection of disconnected topics. It is a **single language** in which modern science speaks. And if you master this language, doors will open before you that you do not even suspect exist today.

1.1 How to read this book

Although the text is written to be as easy to read as possible, without detailed mathematical proofs, some places may be formulated in too abstruse a way. That is normal — mathematics does not come all at once.

Therefore the source of this book in LaTeX is attached. If something is not quite clear:

1. Find the corresponding text in the source,
2. Copy it into some chat (Qwen, DeepSeek, Kimi, ChatGPT, GROK, Gemini),
3. Ask it to explain this more clearly, or ask the question you do not understand.

You can also use these chats to translate the whole text into English or German — simply ask them to translate while preserving the LaTeX markup, and compile the result with the command `xelatex NumMathForChemists.tex` into a PDF file.

Tip

Do not try to read everything in one sitting. Mathematics is like music: you have to "play" it, that is, solve problems, try code, experiment. If some chapter seems difficult — come back to it in a week, and you will be surprised how much easier it has become.

2 Introduction: Notation and the Nature of Numbers

2.1 The hierarchy of number sets

Before we start talking about complicated things, let us agree on the language. In mathematics we group numbers into sets that have certain properties. We shall use the following standard notation:

- $\mathbb{N} = \{1, 2, 3, \dots\}$ — **natural numbers**. Used for counting.
- $\mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$ — **integers**. They add zero and negative numbers, allowing us to describe debt and assets, temperatures below zero, and so on.
- $\mathbb{Q}$ — **rational numbers**. Any number that can be represented as a fraction $\frac{p}{q}$, where $p \in \mathbb{Z}$, $q \in \mathbb{Z} \setminus \{0\}$.
- $\mathbb{R}$ — **real numbers**. They include all rational numbers, as well as irrational numbers (for example, $\sqrt{2}, \pi, e$), which cannot be represented as an ordinary fraction. They fill the entire number line continuously.
- $\mathbb{C}$ — **complex numbers**. We shall discuss them in detail below.

Nesting of sets: $\mathbb{N} \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R} \subset \mathbb{C}$.

2.2 Numbers in a computer

We are used to numbers in mathematics being infinitely precise. But when we move to **computational mathematics** and programming, the computer is forced to store numbers in memory cells of fixed size (in bits and bytes).

- **Integer types** (int8, int16, int32, int64) store exact values from $\mathbb{Z}$, but are limited in range (for example, int32 can store numbers from -2^{31} to $2^{31} - 1$).
- **Real types** (float32, float64 / double) store numbers from $\mathbb{R}$ in exponential format (mantissa and exponent). Because of this, rounding errors arise: the computer's $0.1 + 0.2$ is not always exactly equal to 0.3. Understanding this is critically important for numerical methods!

3 When one number is not enough: Complex numbers

Our world is so complex that not everything can be described by a single number. In physics and chemistry we often have to work with entities that require *two* numbers for their description (for example, amplitude and phase of a wave, or the real and imaginary parts of a wave function).

3.1 Where do they come from?

The simplest and most historical way to get acquainted with "number-pairs" is to try to solve the equation:

$$x^2 + 1 = 0 \quad \Rightarrow \quad x^2 = -1$$

In the set of real numbers $\mathbb{R}$, the square of any number is non-negative. There is no root of -1. But mathematicians (and physicists after them) said: "What if we simply *invent* a number which, when multiplied by itself, gives -1?".

Thus the **imaginary unit** i appeared. We do not "extract the root" of -1, we *define* a new entity:

$$\mathrm{i}^2 = -1$$

Important remark: the rule $\sqrt{a}\sqrt{b} = \sqrt{ab}$ *works only for* $a, b \geq 0$. *Therefore we do not write* $\sqrt{-1}\sqrt{-1} = \sqrt{(-1)(-1)} = 1$. *We simply accept* $\mathrm{i}^2 = -1$ *as a fact.*

3.2 Algebraic form and the complex plane

Any complex number $z \in \mathbb{C}$ can be written as:

$$z = x + \mathrm{i}y$$

where $x = \mathrm{Re}(z)$ is the **real part**, and $y = \mathrm{Im}(z)$ is the **imaginary part**. Both parts $x, y \in \mathbb{R}$.

Geometrically, a complex number is a point on the **complex plane**.

- Along the horizontal axis (the Re axis) the real part x is plotted.
- Along the vertical axis (the Im axis) the imaginary part y is plotted.

A complex number can also be viewed as a **vector** going from the origin $(0, 0)$ to the point (x, y).

3.3 Arithmetic of complex numbers

Addition is intuitively clear. If we have two numbers $z_1 = x_1 + \mathrm{i}y_1$ and $z_2 = x_2 + \mathrm{i}y_2$, we simply add their real and imaginary parts separately:

$$z_1 + z_2 = (x_1 + x_2) + \mathrm{i}(y_1 + y_2)$$

Geometrically this works as the **triangle rule** for vectors: we draw two vectors from the origin, move the start of the second vector to the end of the first, and the sum is the vector from the start of the first to the end of the second.

Multiplication in algebraic form looks a little more complicated. We expand the brackets as in ordinary algebra, remembering that $\mathrm{i}^2 = -1$:

$$\begin{aligned} z_1 \cdot z_2 &= (x_1 + \mathrm{i}y_1)(x_2 + \mathrm{i}y_2) \\ &= x_1x_2 + \mathrm{i}x_1y_2 + \mathrm{i}y_1x_2 + \mathrm{i}^2y_1y_2 \\ &= (x_1x_2 - y_1y_2) + \mathrm{i}(x_1y_2 + y_1x_2) \end{aligned}$$

But to understand the *true magic* of complex multiplication, we need to move to another form of notation.

3.4 Trigonometric and exponential forms

Instead of the coordinates (x, y), a point on the plane can be specified using polar coordinates: the distance from the origin (the modulus) r and the angle φ relative to the positive direction of the Re axis.

The connection with the algebraic form is obvious from trigonometry:

$$x = r\cos\varphi, \quad y = r\sin\varphi$$

Then the complex number takes the **trigonometric form**:

$$z = r(\cos\varphi + \mathrm{i}\sin\varphi)$$

3.4.1 Euler's formula and the exponential form

Here the greatest formula of mathematics enters the stage — **Euler's formula**:

$$e^{\mathrm{i}\varphi} = \cos\varphi + \mathrm{i}\sin\varphi$$

Thanks to it, any complex number can be written in the incredibly convenient **exponential form**:

$$z = re^{\mathrm{i}\varphi}$$

3.4.2 The geometric meaning of multiplication

Let us see what happens when two numbers are multiplied in exponential form:

$$z_1 = r_1 e^{\mathrm{i}\varphi_1}, \quad z_2 = r_2 e^{\mathrm{i}\varphi_2}$$

$$z_1 \cdot z_2 = (r_1 e^{\mathrm{i}\varphi_1}) \cdot (r_2 e^{\mathrm{i}\varphi_2}) = (r_1 r_2) e^{\mathrm{i}(\varphi_1+\varphi_2)}$$

The conclusion to remember forever: Multiplication of complex numbers is the **multiplication of their lengths (moduli)** and the **addition of their angles (arguments)**! Geometrically: multiplication by a complex number is a rotation of the vector by the angle φ and its stretching/compression by a factor of r.

3.5 Complex exponentials and logarithms

3.5.1 Properties of the exponential

Since $e^{\mathrm{i}\varphi}$ behaves like an ordinary exponential, all the standard rules work for it:

$$e^{z_1} \cdot e^{z_2} = e^{z_1+z_2}, \quad \frac{e^{z_1}}{e^{z_2}} = e^{z_1-z_2}, \quad (e^z)^n = e^{nz}$$

For a complex number $z = x + \mathrm{i}y$ the exponential splits into real and imaginary parts:

$$e^z = e^{x+\mathrm{i}y} = e^x \cdot e^{\mathrm{i}y} = e^x(\cos y + \mathrm{i}\sin y)$$

The modulus of this number is e^x, and the argument is y.

3.5.2 The complex logarithm

The logarithm is the operation inverse to the exponential. If $e^w = z$, then $w = \ln z$. Let $z = re^{\mathrm{i}\varphi}$ and $w = u + \mathrm{i}v$. Then:

$$e^{u+\mathrm{i}v} = re^{\mathrm{i}\varphi} \quad \Rightarrow \quad e^u e^{\mathrm{i}v} = re^{\mathrm{i}\varphi}$$

Equating moduli and arguments, we get:

$$e^u = r \quad \Rightarrow \quad u = \ln r$$

$$v = \varphi + 2\pi k, \quad k \in \mathbb{Z}$$

Thus the **complex logarithm** is multivalued:

$$\ln z = \ln|z| + \mathrm{i}(\arg z + 2\pi k)$$

The principal value of the logarithm (at $k = 0$ and $\arg z \in (-\pi, \pi]$) is denoted $\operatorname{Ln} z$. The multivaluedness arises because a rotation by 2π returns us to the same point on the complex plane.

3.6 Cheat sheet: Trigonometry and hyperbolic functions

Hyperbolic functions often frighten chemistry students, but in fact they are close relatives of ordinary sines and cosines, simply "living" in the complex plane.

3.6.1 Definitions via the exponential

$$\cos z = \frac{e^{\mathrm{i}z} + e^{-\mathrm{i}z}}{2}, \qquad \sin z = \frac{e^{\mathrm{i}z} - e^{-\mathrm{i}z}}{2\mathrm{i}}$$

$$\cosh z = \frac{e^{z} + e^{-z}}{2}, \qquad \sinh z = \frac{e^{z} - e^{-z}}{2}$$

3.6.2 Connection between trigonometric and hyperbolic functions

If we substitute $\mathrm{i}z$ for z in the definitions, we see an amazing symmetry (Osborn's rules):

$$\cos(\mathrm{i}z) = \cosh z, \qquad \cosh(\mathrm{i}z) = \cos z$$
$$\sin(\mathrm{i}z) = \mathrm{i}\sinh z, \qquad \sinh(\mathrm{i}z) = \mathrm{i}\sin z$$

Mnemonic: When passing from trigonometry to hyperbolics the argument is multiplied by i, and imaginary units may appear in front of the sine/cosine. Hyperbolic functions describe not oscillations (like the sine) but exponential growth/decay (like e^x), which is critically important for damped processes.

3.7 Why is this needed in real chemistry and physics?

It may seem that imaginary numbers are just a beautiful mathematical abstraction. But our world is arranged such that *without* the imaginary space we could not describe real things.

3.7.1 Quantum mechanics and orbitals

The Schrödinger equation, which describes the behaviour of electrons in atoms, contains the imaginary unit i explicitly:

$$\mathrm{i}\hbar\frac{\partial\Psi}{\partial t} = \hat{H}\Psi$$

The wave function Ψ is complex-valued. For the ground state of the hydrogen atom (the 1s orbital) the wave function is real, and it can be drawn in real space. But as soon as the electron goes into an excited state (for example, a 2p orbital with magnetic quantum number $m \neq 0$), its wave function contains the phase factor $e^{\mathrm{i}m\varphi}$. Without complex numbers it is simply impossible to describe the angular momentum of the electron and the fine structure of spectra.

3.7.2 Nuclear Magnetic Resonance (NMR) and Fourier analysis

You will work a lot with NMR spectroscopy. When you place a sample in a magnetic field and apply a radio-frequency pulse, the nuclei begin to precess. The detector registers a signal decaying in time — this is called the **FID** (Free Induction Decay).

The signal from one type of nucleus looks like a damped oscillation:

$$S(t) = Ae^{-t/T_2}\cos(\omega t)$$

If the sample contains many different nuclei, the signal turns into a mess of many superimposed cosines. How can we find out which frequencies ω are hidden there?

Here the **Fourier transform** comes to the rescue. But working with cosines is difficult. It is much easier to pass to complex exponentials using Euler's formula:

$$\cos(\omega t) = \frac{e^{\mathrm{i}\omega t} + e^{-\mathrm{i}\omega t}}{2}$$

In NMR spectrometers quadrature detection is used, which allows measuring a complex signal directly:

$$S_{complex}(t) = Ae^{-t/T_2}e^{\mathrm{i}\omega t}$$

When we apply the fast Fourier transform (FFT) to this signal, time t passes into frequency ω. A long oscillating function in the time domain turns into a **narrow peak (a Lorentzian)** in the frequency domain! Each type of nucleus corresponds to its own frequency ω, and in the resulting NMR spectrum we see individual peaks. All modern signal processing (from MRI in medicine to audio MP3) works precisely thanks to the magic of complex numbers and Euler exponentials.

4 When one measurement is not enough: Vectors and norms

4.1 From the plane to N-dimensional space

Well, if we have complex numbers (essentially pairs of numbers), then surely there is something more complicated? Quaternions? Octonions?

Yes, indeed, mathematicians have generalised numbers from the plane to higher dimensions. But beyond complex numbers and quaternions (which, by the way, have taken root excellently in robotics and 3D graphics for describing rotations in our three-dimensional space), special "types of numbers" are practically not used. Instead, people simply group numbers into sets and call them **N-dimensional vectors**.

This is simply N-dimensional space. N can be two (then it is a plane), three (our ordinary physical space) or very, very large. We shall denote such vectors in bold, for example $\mathbf{v} \in \mathbb{R}^N$. This is a vector:

$$\mathbf{v} = (v_1, v_2, \ldots, v_N)^T$$

Here T means transposition, so as to write a column in one line of text. We are free to invent what to do with this space, but the first thing we need is to understand how to measure the "size" or "length" of such a vector.

4.2 Why do we need vectors? Examples from chemistry

At first glance, a vector is just a set of numbers. But where did the numbers come from? Suppose this is data from a chromatograph or an NMR spectrometer.

Imagine we want to find out which is the "brightest" peak here. That is simply the maximum among these numbers, right? And now let us take a chromatogram of some complex dirt and a chromatogram of a single pure substance. We inject both into the chromatograph in the same molar volume (let us agree that the detector senses all molecules equally).

- In the case of the pure substance we get one beautiful, very sharp and tall peak.
- In the case of the dirt we get a mountain of peaks, perhaps even merged into one continuous gentle hump.

But here is what is interesting: the **area** (integral intensity) of both chromatograms will be the same! The peak of the pure substance will be very tall compared to the mountain, but the sum of all responses is equal to the amount of substance.

From this problem three different ways naturally arise to estimate our data vector:

1. **Maximum.** We care about the maximum concentration (or maximum absorption, if the peak "looks" downwards). We take the largest value.

2. **Sum.** We care about the total amount of substance. We add up all the values (usually in absolute value, since the baseline may not be ideal).

3. **Root of the sum of squares.** And why is this needed? Imagine that we describe a molecule not by a chromatogram but by three parameters: *polarity, molar mass, boiling point*. This is a vector in $\mathbb{R}^3$. To understand how *similar* two molecules are to each other (for example, for drug design), we cannot simply add the differences of the parameters or take the maximum difference. We need precisely the **Euclidean distance** — the shortest path in this "chemical space". Or another example: in physics the quadratic norm of a vector (a spectrum) is proportional to the **total energy** of that signal.

4.3 The mathematical language of norms

All these three intuitive notions are called **norms of vectors** in mathematics and are denoted by double vertical bars $\|\mathbf{v}\|$.

- **Maximum norm (L_∞):**
$$\|\mathbf{v}\|_\infty = \max_{1\le k\le N} |v_k|$$
- **Manhattan distance / sum of absolute values (L_1):**
$$\|\mathbf{v}\|_1 = \sum_{k=1}^{N} |v_k|$$
- **Euclidean norm / length of the vector (L_2):**
$$\|\mathbf{v}\|_2 = \sqrt{\sum_{k=1}^{N} |v_k|^2}$$

But mathematicians like order, so they combined all this into one general formula for the L_p**-norm**:

$$\|\mathbf{v}\|_p = \left(\sum_{k=1}^{N} |v_k|^p\right)^{1/p}$$

In this formula L_1 is the case $p = 1$. L_2 is $p = 2$. And what will p be for the maximum? Yes, exactly, $p \to \infty$. If we take the limit of this formula as $p \to \infty$, we rigorously obtain precisely the maximum element of the vector.

4.4 The continuous case: Functions as vectors

Moreover, our vector can be entirely continuous. Then it is in fact a **function** $f(x)$. For now let us just remember that a function can be viewed as an "infinite-dimensional vector" whose values are given at each point x. And for functions the norms work in exactly the same way, only the sum is replaced by an integral:

$$\|f\|_p = \left(\int |f(x)|^p \, dx\right)^{1/p}$$

We shall sometimes operate with an array (a vector), sometimes with a function, and later, in the course on signal processing, we shall see that these are two sides of the same coin.

4.5 Minkowski's inequality

There is one critically important point to remember. We shall often compare $\|\mathbf{x} + \mathbf{y}\|$ with $\|\mathbf{x}\|$ and $\|\mathbf{y}\|$. It is intuitively clear that if we go from point A to point B and then from B to C, the total path will be no shorter than if we went directly from A to C.

This geometric property (the length of a side of a triangle is not greater than the sum of the other two) for any L_p-norms is formalised in **Minkowski's inequality**:

$$\|\mathbf{x} + \mathbf{y}\|_p \le \|\mathbf{x}\|_p + \|\mathbf{y}\|_p$$

How to use this? It allows us to estimate errors. If $\mathbf{x}$ is the true signal and $\mathbf{y}$ is noise (measurement error), then Minkowski's inequality guarantees that the norm (size) of our measured signal $(\mathbf{x} + \mathbf{y})$ will not exceed the sum of the norm of the true signal and the norm of the noise. *(A rigorous and beautiful proof of this inequality can be found in Wikipedia, so as not to overload this script).*

5 Two dimensions and more: Matrices and tensors

5.1 From vectors to matrices

Since we have a vector $\mathbf{a} \in \mathbb{R}^N$ (a one-dimensional entity, the discrete analogue of the function $f(x)$), it is logical to ask: what if we have a function of two variables $f(x, y)$? What is its discrete analogue? And for $f(x, y, z)$?

Yes, two-dimensional objects in discrete space are **matrices**. And as with complex numbers, everything here is very, very well studied. And everything that has three or more dimensions is usually called **tensors**.

By the way, if you ever come across old scientific literature on chemometrics or data analysis, you may be surprised by the zoo of names for tensors. They are called *multilinear, multidimensional, procrustes, multimodal, three-way, multi-way arrays*. Do not be frightened: under all these beautiful words there is simply a multidimensional array of numbers.

For now let us stop at two-dimensional objects: the function $f(x, y)$ and the matrix $\mathbf{A} \in \mathbb{R}^{N \times M}$.

5.2 Complex vectors and matrices

Why have I so far written only $\mathbb{R}$? Both a vector and a matrix can, of course, be complex! We can consider the spaces $\mathbb{C}^N$ and $\mathbb{C}^{N \times M}$. Here everything is simple: instead of one real number in each cell of the vector or matrix there is a complex pair.

But how do we compute the norm of a complex number? In exactly the same way! The modulus of a complex number $z = x + \mathrm{i}y$ is computed via the conjugate number $\bar{z} = x - \mathrm{i}y$:

$$|z| = \sqrt{x^2 + y^2} = \sqrt{z \cdot \bar{z}}$$

Accordingly, the L_2-norm of a complex vector $\mathbf{z} \in \mathbb{C}^N$ looks like this:

$$\|\mathbf{z}\|_2 = \sqrt{\sum_{k=1}^{N} |z_k|^2} = \sqrt{\sum_{k=1}^{N} z_k \bar{z}_k}$$

In matrix notation this is written via the Hermitian conjugate (transpose + complex conjugate) $\mathbf{z}^H$:

$$\|\mathbf{z}\|_2 = \sqrt{\mathbf{z}^H \mathbf{z}}$$

5.3 Norms of matrices

And for matrices we can also define norms! But there is an important nuance here.

Most simple norms for matrices are computed as follows: we mentally "cut" the matrix into rows, arrange all its $N \times M$ cells into one long column vector, and take the norm of that vector. The most popular of these norms is the **Frobenius norm** (or Euclidean norm for matrices), which is the analogue of the L_2-norm:

$$\|\mathbf{A}\|_F = \sqrt{\sum_{i=1}^{N} \sum_{j=1}^{M} |a_{ij}|^2}$$

For it, as for the ordinary L_2, there are many important and beautiful properties which we shall soon consider.

However, besides such "element-wise" norms, in linear algebra there are special **operator (induced) norms**. They answer the question: "How strongly can a matrix stretch a vector if we multiply it by it?". But that is a story for the next chapter, where we shall deal closely with linear operators.

6 Operators: when mathematics begins to act

6.1 What is an operator?

We have come to a very important concept which is encountered everywhere in mathematics and in chemistry. This is the **operator**.

An operator is some *action* on an object. But here it immediately becomes somehow confusing: an action on what? On a number? On a vector? On a function? Let us consider several examples together, and everything will fall into place.

6.2 A matrix as an operator

Suppose we have a square matrix $\mathbf{A} \in \mathbb{C}^{N\times N}$ (I shall henceforth almost always use $\mathbb{C}$, since $\mathbb{R}$ is merely a subset of $\mathbb{C}$, and everything that works for real numbers works for complex ones too).

Consider the action **Ab**, that is, multiplication of a matrix by a vector. The result is again a vector from $\mathbb{C}^N$. The matrix *transforms* one vector into another. This is the simplest example of a linear operator.

6.3 Analogy with a functional operator

And now — the most interesting thing. The analogue of a matrix operator in the world of functions is the **integral operator**:

$$(\hat{K}g)(x) = \int_a^b K(x,y)\, g(y)\, \mathrm{d}y \tag{1}$$

Here $K(x,y)$ is the *kernel* of the operator (the analogue of the matrix **A**), $g(y)$ is the input function (the analogue of the vector **b**), and the result is a new function of x.

Where, interestingly, does such an abstraction occur in real life? It turns out, everywhere! For example, in quantum mechanics the Hamilton operator $\hat{H}$ acts on the wave function Ψ, and this is precisely an integro-differential operator. In spectroscopy the response of an instrument to an input signal is often described by a convolution integral — that too is an operator.

6.4 Back to school: a system of equations

But let us start with the simplest thing. At school we already solved systems of equations, for example:

$$\begin{cases} 2x - 3y = -4 \\ 3x + 2y = 9 \end{cases}$$

You are probably thinking: "So what? We could solve that by substitution or by elimination". Yes, we could. But let us rewrite it in matrix form $\mathbf{Ax} = \mathbf{b}$:

$$\underbrace{\begin{pmatrix} 2 & -3 \\ 3 & 2 \end{pmatrix}}_{\mathbf{A}} \underbrace{\begin{pmatrix} x \\ y \end{pmatrix}}_{\mathbf{x}} = \underbrace{\begin{pmatrix} -4 \\ 9 \end{pmatrix}}_{\mathbf{b}}$$

You may reasonably ask me: "Why complicate it like that?". And I answer: no, we are not complicating — we are *putting in order* our knowledge. We are isolating the *structure* of the problem. And now we can say: if we have a **linear operator A**, then it acts on the vector **x** and the result is the vector **b**.

6.5 Reminder: basic operations

Before going further, let us fix explicitly the operations we shall constantly use.

A vector as a column matrix. A vector $\mathbf{x} \in \mathbb{C}^N$ is in fact a matrix of size $N \times 1$. Therefore all the rules of matrix multiplication apply automatically to vectors too.

Transposition. The operation $\mathbf{A}^T$ swaps rows and columns: $(\mathbf{A}^T)_{ij} = A_{ji}$. For complex matrices the **Hermitian conjugate** $\mathbf{A}^H = \overline{\mathbf{A}^T}$ (transposition + complex conjugation of all elements) is more often used.

Scalar product. For two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{C}^N$ the scalar product is defined as:

$$\langle \mathbf{x}, \mathbf{y} \rangle = \mathbf{x}^H \mathbf{y} = \sum_{k=1}^{N} \bar{x}_k y_k$$

For real vectors this is simply $\mathbf{x}^T \mathbf{y} = \sum x_k y_k$.

The most important property: the scalar product of a vector with itself equals the **square of its Euclidean norm**:

$$\langle \mathbf{x}, \mathbf{x} \rangle = \|\mathbf{x}\|_2^2 = \sum_{k=1}^{N} |x_k|^2$$

This is the bridge between algebra and geometry: the length of a vector is the root of its "scalar square".

Multiplication of a matrix by a vector. If $\mathbf{A} \in \mathbb{C}^{M \times N}$ and $\mathbf{x} \in \mathbb{C}^N$, then the result $\mathbf{y} = \mathbf{A}\mathbf{x} \in \mathbb{C}^M$ is computed by the rule:

$$y_i = \sum_{j=1}^{N} A_{ij} x_j$$

That is, the i-th component of the result is the scalar product of the i-th row of the matrix with the vector $\mathbf{x}$.

Multiplication of matrices. If $\mathbf{A} \in \mathbb{C}^{M \times K}$ and $\mathbf{B} \in \mathbb{C}^{K \times N}$, then the product $\mathbf{C} = \mathbf{A}\mathbf{B} \in \mathbb{C}^{M \times N}$:

$$C_{ij} = \sum_{k=1}^{K} A_{ik} B_{kj}$$

Important: matrix multiplication is, generally speaking, **not commutative** — $\mathbf{A}\mathbf{B} \neq \mathbf{B}\mathbf{A}$. This is one of the most important differences from the multiplication of ordinary numbers!

All these operations — scalar product, matrix-vector multiplication, matrix-matrix multiplication — have their complete analogues in the world of functions and integral operators. Only everywhere where we had a sum $\sum$, an integral $\int$ appears, and transposition is replaced by a more complicated conjugation operator. But that is already a matter of technique.

6.6 The four main problems of linear algebra

In fact, as long as our operators are linear (that is, they can be written in the form (1) or $\mathbf{A}\mathbf{x}$), the whole world of possible problems revolves around a few typical questions. Let us list them.

Problem 1. Compute a scalar product. Given two vectors — find the number $\langle \mathbf{x}, \mathbf{y} \rangle$. This is the basic operation from which, like bricks, everything else is built.

Problem 2. Compute the action of an operator. Given $\mathbf{A}$ and $\mathbf{x}$ — find $\mathbf{A}\mathbf{x}$. Or given $\mathbf{A}$ and $\mathbf{B}$ — find $\mathbf{A}\mathbf{B}$. This is a direct computation.

Problem 3. Solve a linear system of equations. Given $\mathbf{A}$ and $\mathbf{b}$ — find $\mathbf{x}$ such that $\mathbf{A}\mathbf{x} = \mathbf{b}$. This is the *direct* problem: the operator and the result are known, we must find what it acted on.

Problem 4. Minimise the residual (least squares method). But what if the system $\mathbf{A}\mathbf{x} = \mathbf{b}$ has no exact solution? (For example, there are more equations than unknowns — an overdetermined system, which is typical for processing experimental data.) Then we look for $\mathbf{x}$ that *minimises* the residual:

$$\min_{\mathbf{x}} \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_p$$

Almost always $p = 2$ is used as the norm — this is the famous **least squares method (LS)**. It has a deep geometric interpretation: we project the vector $\mathbf{b}$ onto the subspace spanned by the columns of the matrix $\mathbf{A}$.

6.7 And what if the right-hand side is zero?

And now — a cunning turn. Imagine that $\mathbf{b} = \mathbf{0}$. Then the problem $\min_{\mathbf{x}} \|\mathbf{A}\mathbf{x}\|_2$ has the trivial solution $\mathbf{x} = \mathbf{0}$. But that is uninteresting!

Let us impose an additional condition: we look for a **nonzero** solution, for example with the constraint $\|\mathbf{x}\|_2 = 1$. That is, we look for the direction in which the matrix $\mathbf{A}$ "compresses" space most strongly (or, conversely, stretches it most strongly — that is already a question of sign).

And here **eigenvalues and eigenvectors** enter the stage. We look for such special vectors $\mathbf{v}$ and numbers λ for which the action of the matrix reduces simply to stretching:

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$$

Geometrically: an eigenvector is a direction which the matrix *does not rotate*, but only stretches or compresses by a factor of λ.

The problem of minimising $\|\mathbf{A}\mathbf{x}\|_2$ subject to $\|\mathbf{x}\|_2 = 1$ is solved precisely via the eigenvalues of $\mathbf{A}^H\mathbf{A}$ or the singular values of $\mathbf{A}$: the minimum is attained on the eigenvector of the matrix $\mathbf{A}^H\mathbf{A}$ corresponding to the *smallest in modulus* eigenvalue. And the maximum — on the vector corresponding to the largest.

6.8 And if the operator itself is unknown?

You may ask: "And what if the operator is unknown to us, and we know only the input functions and the results?"

Here the answer is somewhat ambiguous. Think: in matrix form we have only N input parameters (the vector $\mathbf{x}$), and we want to find N^2 parameters of the matrix $\mathbf{A}$. Nature is rarely arranged such that a small number of inputs determines a larger number of parameters well. This is a classical **underdetermined** problem.

But here too there is a solution! If we say that the same matrix $\mathbf{A}$ acts on several different vectors $\mathbf{x}_1, \ldots, \mathbf{x}_K$ with known results $\mathbf{b}_1, \ldots, \mathbf{b}_K$, then we can formulate the problem as:

$$\forall k = 1, \ldots, K: \quad \mathbf{A}\mathbf{x}_k = \mathbf{b}_k, \quad \text{where } \mathbf{A} \text{ is unknown.} \tag{2}$$

If we collect the vectors into matrices $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_K)$ and $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_K)$, the problem is rewritten as:

$$\mathbf{A}\mathbf{X} \approx \mathbf{B}$$

And this, attention, is **the same minimisation problem**, only now we minimise not over $\mathbf{x}$ but over $\mathbf{A}$:

$$\min_{\mathbf{A}} \|\mathbf{A}\mathbf{X} - \mathbf{B}\|_F$$

(Here $\|\cdot\|_F$ is the Frobenius norm for matrices, which we discussed in the previous chapter.)

Let us transpose for convenience: $\mathbf{X}^T\mathbf{A}^T \approx \mathbf{B}^T$. And we have again obtained a problem of the form "minimise $\|\mathbf{M}\mathbf{z} - \mathbf{c}\|_2$", only for each column of $\mathbf{A}^T$ separately. This is classical **linear regression** — the basis of chemometrics, QSAR, instrument calibration and much else.

6.9 Singular value decomposition: a bridge into big worlds

And here one of the most beautiful constructions in all of linear algebra enters the stage — the **singular value decomposition (SVD)**.

Any matrix $\mathbf{A} \in \mathbb{C}^{M\times N}$ can be represented as:

$$\mathbf{A} = \mathbf{U}\square\mathbf{V}^H$$

where $\mathbf{U} \in \mathbb{C}^{M\times M}$ and $\mathbf{V} \in \mathbb{C}^{N\times N}$ are unitary matrices (their columns are orthonormal vectors), and $\square$ is a diagonal matrix with non-negative numbers $\sigma_1 \geq \sigma_2 \geq \cdots \geq 0$ on the diagonal. These numbers are called **singular values**.

The geometric meaning of SVD is astonishing: any matrix is simply a rotation ($\mathbf{V}^H$), then a stretching along the axes (□), then another rotation (**U**). That is all! Matrices can do nothing more.

SVD is a universal tool. Through it one solves:

- least squares problems,
- finding the pseudoinverse matrix,
- data compression and principal component extraction (PCA — Principal Component Analysis, the basis of chemometrics!),
- regularisation of ill-conditioned problems.

There is an important point here: unitary matrices have one very important property, namely that always $\mathbf{V}^H\mathbf{V} = \mathbf{I}$, where **I** is the identity matrix, that is, one with ones on the main diagonal and everything else filled with zeros.

6.10 A bridge into functional analysis: Fredholm theory

And now — the most interesting thing. Do you remember the integral operator (2)? Well, there is an analogue of SVD for it too! It is called the **singular value decomposition of a compact operator** or, in a more classical formulation, **Fredholm theory**.

The essence is as follows: the kernel of the integral operator $K(x, y)$ can be expanded in a series in “singular functions” $u_k(x)$ and $v_k(y)$:

$$K(x, y) = \sum_{k=1}^{\infty} \sigma_k \, u_k(x)\overline{v_k(y)}$$

where σ_k are the singular values (decreasing to zero), and u_k, v_k are orthonormal systems of functions.

This is exactly the analogue of SVD for matrices, only in an infinite-dimensional space! And all the ideas we understood on matrices carry over here almost word for word.

But let us not overload our minds with Fredholm and functional analysis right now. The good news is that **most of what we need in chemistry and signal processing can be done at the matrix level**. And where it no longer works (for example, in quantum mechanics or in the rigorous theory of integral equations), we simply remember that “somewhere out there is Fredholm”, and if necessary ask AI for details — it will gladly tell us about Fredholm theory, about Hilbert–Schmidt kernels and about the spectra of compact operators.

The main thing is to understand the *structure*. And the structure is the same everywhere: operator, space, action, expansion in “basis directions”.

7 Conditioning: when a matrix “wants” to deceive us

7.1 When a solution exists, but seems not to

So, let us have a square matrix **A** and a linear system $\mathbf{Ax} = \mathbf{b}$. What can we say about it?

We remember from the school course that sometimes a system of equations is such that, when we start substituting, either there are no solutions, or we end up with two or more identical equations — and then there are infinitely many solutions.

In chemistry, and indeed in any industrial problem, the initial data for matrix equations often come from measurements with instruments. And here we may be unlucky in three ways at once:

1. Our system will be degenerate (no unique solution).
2. Our system will have many solutions.

3. **The most insidious case:** solving in machine arithmetic, we find ourselves *very close* to the bad case, but do not notice it. And we get an answer that looks reasonable, but in fact is complete nonsense.

When can this happen? Let us sort it out with a vivid example.

7.2 An example from chromatography: the trap of the large and the small

Suppose we have two chromatograms in which two substances were recorded against the background of a solvent. The solvent peak came out very broad and “crept” onto the peaks of the substances sought, while the responses of the substances sought turned out to be very weak.

In fact, in the peak of the substances sought we have:

- First chromatogram: $(a + b_1)$, where a is a huge response from the solvent, and b_1 is a very small number from the substance sought.
- Second chromatogram: $(a+b_2)$, but since the temperature became a little higher, this measurement is slightly inaccurate, say, it increased by some value ε *relative to the solvent peak*, that is $(1 + \varepsilon)(a + b_2)$.

If we want to subtract the second from the first to compute $b_1 - b_2$, then instead we get:

$$(a + b_1) - (1 + \varepsilon)(a + b_2) = b_1 - b_2 - \varepsilon(a + b_2) \approx b_1 - b_2 - \varepsilon a$$

That is, if εa is comparable in order of magnitude to $b_1 - b_2$, we can get not just a large error, but also the **opposite sign**!

This is the essence of poor conditioning: when one number contains simultaneously a very large and a very small component, and we try to extract the small one by subtraction.

Rule for subtracting close numbers

Never subtract two numbers of similar magnitude if you need relative accuracy. Loss of significant digits is not a bug of the computer, it is a fundamental property of arithmetic.

7.3 How SVD reveals the mechanism of the catastrophe

The same thing happens when solving systems of equations. And we have a nice way to *estimate in advance* what to do.

Suppose we have a system $\mathbf{A}\mathbf{x} = \mathbf{b}$, where we seek $\mathbf{x}$ and everything else is given. Recall the singular value decomposition $\mathbf{A} = \mathbf{U}\square\mathbf{V}^H$. Then the solution can be rewritten in several stages:

$$\begin{aligned}
\mathbf{U}\square\mathbf{V}^H\mathbf{x} &= \mathbf{b} \\
\square\mathbf{V}^H\mathbf{x} &= \mathbf{U}^H\mathbf{b} \equiv \mathbf{t}_1 \\
\mathbf{V}^H\mathbf{x} &= \square^{-1}\mathbf{t}_1 \equiv \mathbf{t}_2 \\
\mathbf{x} &= \mathbf{V}\mathbf{t}_2
\end{aligned}$$

Here we use the property that solving a system with unitary matrices $\mathbf{U}, \mathbf{V}$ is very simple — it suffices to multiply by $\mathbf{U}^H$ or $\mathbf{V}^H$, since $\mathbf{U}^H\mathbf{U} = \mathbf{I}$. And solving a system with a diagonal matrix is altogether trivial.

Let us draw how this looks for a 3×3 matrix:

$$\begin{pmatrix} \sigma_1 & 0 & 0 \\ 0 & \sigma_2 & 0 \\ 0 & 0 & \sigma_3 \end{pmatrix} \begin{pmatrix} t_{2,1} \\ t_{2,2} \\ t_{2,3} \end{pmatrix} = \begin{pmatrix} t_{1,1} \\ t_{1,2} \\ t_{1,3} \end{pmatrix} \quad \Rightarrow \quad \begin{cases} \sigma_1 t_{2,1} = t_{1,1} \\ \sigma_2 t_{2,2} = t_{1,2} \\ \sigma_3 t_{2,3} = t_{1,3} \end{cases} \quad \Rightarrow \quad t_{2,k} = \frac{t_{1,k}}{\sigma_k}$$

Do you see? Each equation is simply a division of one component by the corresponding singular value.

And here is the most important thing. Imagine that we have computed the singular values of the original matrix and noticed that the largest singular value σ_1 is very, very much larger than the smallest σ_N.

Then at the step $\mathbf{t}_2 = \square^{-1}\mathbf{t}_1$ the following happens:

- In the vector $\mathbf{t}_1$ the measurement error is distributed more or less uniformly over all components.
- After multiplication by $\square^{-1}$ the component corresponding to the *small* σ_N grows by a factor of σ_1/σ_N!
- And if σ_N is altogether zero? Then we divide by zero — and the solution does not exist.

It is precisely because of this that people decided to denote the ratio σ_1/σ_N as the **condition number** of the matrix:

$$\mathrm{cond}(\mathbf{A}) = \frac{\sigma_{\max}}{\sigma_{\min}}$$

In fact, this is a characteristic of *how much we can spoil the solution with such a matrix*. If $\mathrm{cond}(\mathbf{A}) \approx 10^k$, then when solving the system we lose approximately k significant digits of accuracy. For double precision (16 significant digits) this means that at $\mathrm{cond}(\mathbf{A}) \approx 10^{16}$ the answer will consist entirely of noise.

Important remark

The condition number is a characteristic of the *matrix*, not of the algorithm. No algorithm, however ingenious, can solve an ill-conditioned system more accurately than $\mathrm{cond}(\mathbf{A})$ permits. This is a fundamental limitation of the problem, not of our computational methods.

At the same time, conditioning *does not affect* the computation of eigenvectors of symmetric/Hermitian matrices or the finding of singular vectors — these problems are, as a rule, much more stable.

7.4 Where does all this occur in chemistry and beyond?

We have had a lot of dry theory, and it would be good to recall where all this is applied in chemistry.

Quantum chemistry: the Schrödinger equation. Yes, the Schrödinger wave equation and all its approximations — the Hartree–Fock equations, Density Functional Theory (DFT) — all these are eigenvalue problems: finding vectors $\mathbf{x}$ and values λ satisfying the equation $\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$. The matrices here — Fock matrices or Hamiltonian matrices — often have dimensions of thousands and tens of thousands, and their conditioning directly determines whether we can obtain a physically meaningful answer at all.

Mass spectrometry and spectroscopy. Finding a recorded spectrum from a database of responses in a mass spectrometer — these are algorithms based on solving linear systems. If you have a mixture of substances and want to understand what it consists of, you solve the system $\mathbf{A}\mathbf{x} = \mathbf{b}$, where $\mathbf{A}$ is the matrix of reference spectra, $\mathbf{b}$ is the measured spectrum of the mixture, and $\mathbf{x}$ are the concentrations of the components. And if the spectra of the components are similar — the matrix is ill-conditioned, and the concentrations will “jump”.

Cryo-electron microscopy. Reconstruction of the 3D structure of proteins from thousands of 2D projections is a gigantic sparse system of equations. Conditioning there is a key factor determining the resolution of the resulting structure.

NMR spectroscopy. Deconvolution (inverse convolution) is a classical ill-conditioned problem. That is precisely why NMR uses Tikhonov regularisation and the Fourier transform — they turn an ill-conditioned problem into a diagonal one.

Calibration of analytical instruments. PLS regression (Partial Least Squares) is, in essence, SVD with regularisation. The conditioning of the matrix of spectra directly affects the accuracy of concentration prediction.

Internet search (PageRank). Even the very first Google search was arranged so that all previously found documents were classified into a huge matrix, and its singular value decomposition was computed (or, equivalently, the principal eigenvector of the transition matrix was found), which helped to find the desired document instantly by a matching combination of words. In modern AI, SVD is used very, very often — from compressing neural network weights to dimensionality reduction in recommender systems.

Signal and image processing. Deconvolution of blurred images in microscopy, noise suppression in ECG/EEG, compression of audio (MP3) and video — all these are problems in which conditioning plays a key role.

Once upon a time these three problems — solving linear systems, finding eigenvalues and SVD — were a stumbling block in solving large industrial problems. There were even firms that developed *only* such solvers — and nothing else. True, that was back in the 1990s.

7.5 How much does it cost? Computational complexity

Roughly speaking, if we have a square $N \times N$ matrix and we either solve a linear system with it, or find its eigenvalues or singular values and vectors, then we must spend approximately $\mathcal{O}(N^3)$ arithmetic operations, if we do not specially use the structure or properties of this matrix.

This is the “price” of the full problem. But if the matrix has special properties, the price can be greatly reduced — sometimes to $\mathcal{O}(N)$ or $\mathcal{O}(N \log N)$. And that is discussed below.

7.6 The connection between singular value decomposition and eigenvalues

And now — one of the most beautiful facts in all of linear algebra. Suppose we have the singular value decomposition:

$$\mathbf{A} = \mathbf{U}\square\mathbf{V}^H$$

Let us see what happens if we multiply $\mathbf{A}$ by its Hermitian conjugate $\mathbf{A}^H$:

$$\begin{aligned}\mathbf{A}\mathbf{A}^H &= (\mathbf{U}\square\mathbf{V}^H)(\mathbf{V}\square\mathbf{U}^H) \\ &= \mathbf{U}\square(\mathbf{V}^H\mathbf{V})\square\mathbf{U}^H \\ &= \mathbf{U}\square^2\mathbf{U}^H\end{aligned}$$

Here we used that $\mathbf{V}^H\mathbf{V} = \mathbf{I}$ (unitarity of $\mathbf{V}$), and $\square^T = \square$ (diagonal matrix).

Similarly:

$$\begin{aligned}\mathbf{A}^H\mathbf{A} &= (\mathbf{V}\square\mathbf{U}^H)(\mathbf{U}\square\mathbf{V}^H) \\ &= \mathbf{V}\square(\mathbf{U}^H\mathbf{U})\square\mathbf{V}^H \\ &= \mathbf{V}\square^2\mathbf{V}^H\end{aligned}$$

What does this mean?

- The singular vectors of the matrix $\mathbf{A}$ are *exactly* the eigenvectors of the matrices $\mathbf{A}\mathbf{A}^H$ and $\mathbf{A}^H\mathbf{A}$.
- The singular values σ_k of the matrix $\mathbf{A}$ are the square roots of the eigenvalues λ_k of the matrices $\mathbf{A}\mathbf{A}^H$ or $\mathbf{A}^H\mathbf{A}$:

$$\sigma_k = \sqrt{\lambda_k}$$

This is a deep connection between two seemingly different problems: singular value decomposition and eigenvalue finding.

Caution: the square of the condition number!

But there is one insidious detail. The condition number of the matrices $\mathbf{A}\mathbf{A}^H$ and $\mathbf{A}^H\mathbf{A}$ is the **square** of the condition number of the original matrix:

$$\text{cond}(\mathbf{A}\mathbf{A}^H) = \text{cond}(\mathbf{A}^H\mathbf{A}) = \text{cond}(\mathbf{A})^2$$

If $\text{cond}(\mathbf{A}) = 10^8$, then $\text{cond}(\mathbf{A}^H\mathbf{A}) = 10^{16}$ — and we will lose all 16 significant digits of accuracy!
Therefore, passing from the singular value decomposition problem to the eigenvalue problem for $\mathbf{A}^H\mathbf{A}$ must be done **with extreme caution**. In modern libraries (LAPACK) SVD is computed directly, without explicitly forming $\mathbf{A}^H\mathbf{A}$ — precisely in order to avoid this catastrophe.

8 Classification of matrices: who is who

Now that it is clear that practically everything is based on these "simple" mathematical problems, let us systematise a little our knowledge of such solutions. After all, we always have a matrix, and much depends on the properties of this matrix.

8.1 By shape

Type	Size	Description
Square	$N \times N$	Equal number of rows and columns. Only for such matrices are the determinant, eigenvalues and inverse matrix defined.
"Tall"	$M \times N,\ M > N$	More equations than unknowns. Usually has no exact solution — we solve via least squares.
"Wide"	$M \times N,\ M < N$	More unknowns than equations. Infinitely many solutions — we seek the one of minimal norm.

8.2 By symmetry and structure of elements

Type	Definition and properties	Complexity
Symmetric real	$\mathbf{A} = \mathbf{A}^T$, elements $\in \mathbb{R}$. Eigenvalues **real**, eigenvectors — orthogonal.	$\mathcal{O}(N^3/3)$
Hermitian complex	$\mathbf{A} = \mathbf{A}^H$, elements $\in \mathbb{C}$. Eigenvalues **real**, eigenvectors — orthonormal.	$\mathcal{O}(4N^3/3)$
Skew-symmetric	$\mathbf{A} = -\mathbf{A}^T$. Eigenvalues — purely imaginary or zero.	$\mathcal{O}(N^3/3)$
Orthogonal / Unitary	$\mathbf{A}^T\mathbf{A} = \mathbf{I}$ / $\mathbf{A}^H\mathbf{A} = \mathbf{I}$. Preserves lengths and angles. Eigenvalues equal 1 in modulus.	$\mathcal{O}(N^2)$ for multiplication
Identity	$\mathbf{A} = \mathbf{I}$. Diagonal of ones, the rest — zeros.	$\mathcal{O}(N)$
Diagonal	$A_{ij} = 0$ for $i \neq j$. Stored as a vector of N numbers.	$\mathcal{O}(N)$

8.3 By positive definiteness

This is a subclass of symmetric/Hermitian matrices, and it is critically important for optimisation and statistics.

Type	Definition and properties
Positive definite ($\mathbf{A} \succ 0$)	$\forall \mathbf{x} \neq 0: \ \mathbf{x}^H \mathbf{A} \mathbf{x} > 0$. All eigenvalues strictly positive. Conditioning — the ratio of the largest and smallest eigenvalues. Solved by the Cholesky method in $\mathcal{O}(N^3/3)$.
Positive semi-definite ($\mathbf{A} \succeq 0$)	$\forall \mathbf{x}: \ \mathbf{x}^H \mathbf{A} \mathbf{x} \geq 0$. All eigenvalues ≥ 0. May be degenerate.

8.4 By the structure of the arrangement of nonzero elements

Type	Definition and properties	Complexity
Banded	Nonzero elements only near the main diagonal: $A_{ij} = 0$ for $\lvert i - j \rvert > k$. Stored as $N \times (2k + 1)$.	$\mathcal{O}(Nk^2)$
Toeplitz	A_{ij} depends only on $i - j$. Each diagonal is a constant. Arises in problems with stationary processes.	$\mathcal{O}(N^2)$, via FFT — $\mathcal{O}(N \log N)$
Circulant	Toeplitz + periodicity: each row is a cyclic shift of the previous one. Diagonalised by the Fourier transform.	$\mathcal{O}(N \log N)$ via FFT
Block	Consists of blocks, each of which is a matrix. Allows recursive algorithms.	Depends on the block structure
Sparse	Most elements are zeros. Stored in special formats (CSR, CSC). The basis of large computations.	$\mathcal{O}(\text{nnz})$, where nnz is the number of nonzeros

8.5 Degeneracy

A **degenerate (singular) matrix** is a matrix whose determinant is zero, or, equivalently, which has at least one singular value equal to zero. For such a matrix:

- there is no inverse matrix,
- the system $\mathbf{A}\mathbf{x} = \mathbf{b}$ either has no solutions or has infinitely many,
- $\text{rank}(\mathbf{A}) < N$.

In practice matrices are almost never *exactly* degenerate — but they can be *almost* degenerate, that is, with a very small σ_{min}. And this is a much more insidious case, because formally a solution exists, but it is entirely determined by noise in the data.

8.6 Additional important types of matrices

Type	Definition and properties	Complexity
Upper triangular	$A_{ij} = 0$ for $i > j$. All nonzero elements above or on the main diagonal. Solving the system — back substitution.	$\mathcal{O}(N^2)$
Lower triangular	$A_{ij} = 0$ for $i < j$. All nonzero elements below or on the main diagonal. Solving the system — forward substitution.	$\mathcal{O}(N^2)$
Permutation matrix	In each row and each column exactly one unit, the rest — zeros. Multiplication by such a matrix is a permutation of rows/columns. $\mathbf{P}^T = \mathbf{P}^{-1}$.	$\mathcal{O}(N)$

Permutation matrices are not just an abstraction. They are critically important for the numerical stability of algorithms (for example, in LU decomposition with pivoting), and we shall meet them again.

8.7 Summary table: what to solve and how

Problem	General case	Symmetric	Sparse
Linear system $\mathbf{Ax} = \mathbf{b}$	LU, $\mathcal{O}(N^3)$	Cholesky, $\mathcal{O}(N^3/3)$	Iterative, $\mathcal{O}(\text{nnz} \cdot k)$
Least squares $\min \|\mathbf{Ax} - \mathbf{b}\|_2$	QR, $\mathcal{O}(MN^2)$	—	LSQR, iterative
Eigenvalues	$\mathcal{O}(N^3)$	$\mathcal{O}(N^3/3)$, all real	Lanczos, $\mathcal{O}(\text{nnz} \cdot k)$
SVD	$\mathcal{O}(MN^2)$	Via eigenvalues of $\mathbf{A}^T\mathbf{A}$	Truncated SVD, iterative

Here k is the number of iterations, which depends on the desired accuracy and the conditioning.

Main conclusion

Before solving a problem, *look at the matrix*. Its properties — symmetry, sparsity, banded structure — can reduce the complexity from cubic to linear. And its conditioning will tell you whether the answer obtained is worth trusting at all.

9 How linear algebra problems are solved: from theory to practice

9.1 There is no single solution for all cases

So how are these linear algebra problems solved? Surely there is one, or perhaps a couple of the simplest and most reliable solutions?

Unfortunately, even now there is no single solution for all cases of life, and none is in sight. The author of this script once participated in writing more than a hundred different such algorithms. Yes, there are very many such algorithms, and one must at least briefly, even without ever implementing it, understand when and what can be used.

9.2 Classification of solution methods

Solution methods can be classified roughly as follows:

9.2.1 Direct decomposition methods

LU decomposition: $\mathbf{A} = \mathbf{LU}$, where $\mathbf{L}$ is lower triangular and $\mathbf{U}$ is upper triangular. With such a factorisation, solving the system $\mathbf{Ax} = \mathbf{b}$ reduces to two simple substitutions:

1. Solve $\mathbf{Ly} = \mathbf{b}$ (forward substitution, $\mathcal{O}(N^2)$)
2. Solve $\mathbf{Ux} = \mathbf{y}$ (back substitution, $\mathcal{O}(N^2)$)

But in general $\text{cond}(\mathbf{L}) \cdot \text{cond}(\mathbf{U}) \geq \text{cond}(\mathbf{A})$, and if no permutations are made, the product of condition numbers may become substantially larger than $\text{cond}(\mathbf{A})$. That is, we can spoil our original problem!

Therefore in practice one almost always uses **LUP decomposition**: $\mathbf{PA} = \mathbf{LU}$, where $\mathbf{P}$ is a permutation matrix. The permutations are chosen so that the diagonal of $\mathbf{U}$ contains the largest possible elements (pivoting). This guarantees numerical stability.

For special matrices:

- If $\mathbf{A}$ is banded, then $\mathbf{L}$ and $\mathbf{U}$ do not go beyond the band. Complexity $\mathcal{O}(Nk^2)$, where k is the band width.
- If $\mathbf{A}$ is sparse, then $\mathbf{L}$ and $\mathbf{U}$ may contain substantially more nonzero elements (fill-in), and this can be a big problem.

- For a symmetric matrix — $\mathbf{A} = \mathbf{L}\mathbf{D}\mathbf{L}^H$ or $\mathbf{P}\mathbf{A}\mathbf{P}^H = \mathbf{L}\mathbf{D}\mathbf{L}^H$.
- For a positive definite one — $\mathbf{A} = \mathbf{L}\mathbf{L}^H$, the so-called **Cholesky method**. But the problem of growth of the condition number exists even for positive definite matrices.

QR decomposition: $\mathbf{A} = \mathbf{Q}\mathbf{R}$, where $\mathbf{Q}$ is unitary ($\mathbf{Q}^H\mathbf{Q} = \mathbf{I}$) and $\mathbf{R}$ is upper triangular.

This method **guarantees no increase in the condition number** of the solution, since unitary transformations preserve the lengths of vectors. It is good for dense matrices without structure.

But if the original matrix is sparse, then $\mathbf{Q}$ will almost always be dense (only for very special cases and methods), which strongly limits the applicability of this method to solving huge problems, when the matrix itself fits in memory but its dense version N^2 does not.

Schur decomposition: $\mathbf{A} = \mathbf{Q}\mathbf{G}\mathbf{Q}^H$, where $\mathbf{G}$ is upper triangular (for real matrices — upper quasi-triangular with 2×2 blocks on the diagonal).

It is used for finding eigenvalues and eigenvectors, which are found by solving the eigenvalue problem for the already triangular matrix $\mathbf{G}$ (and for a triangular matrix the eigenvalues are simply the diagonal elements!).

9.2.2 Iterative methods

These are methods for solving a linear system or an eigenvalue problem when we can efficiently multiply by the matrix because of its structure, and we build the solution iteratively.

There are many methods here, but it is important to note: **the convergence of iterative methods depends on the condition number**. The larger it is, the more slowly they converge. (There are exceptions: if the matrix has only a few very large and very small singular values, then convergence can be substantially faster.)

All these methods can also be divided into subclasses:

1. **Special methods for positive definite matrices** with small additional memory costs and a reasonable number of iterations. The best known is the **conjugate gradient method (CG)**.
2. **Reduction of a nonsymmetric problem to a symmetric one:** instead of solving $\mathbf{A}\mathbf{x} = \mathbf{b}$, solve $\mathbf{A}^H\mathbf{A}\mathbf{x} = \mathbf{A}^H\mathbf{b}$. If the condition number $\mathrm{cond}(\mathbf{A}^H\mathbf{A})$ is not so enormous compared to $\mathrm{cond}(\mathbf{A})$, then this is reasonable. But remember: $\mathrm{cond}(\mathbf{A}^H\mathbf{A}) = \mathrm{cond}(\mathbf{A})^2$, so this is a double-edged sword.
3. **Preconditioners:** special matrices $\mathbf{P} \approx \mathbf{A}^{-1}$ such that instead of solving $\mathbf{A}\mathbf{x} = \mathbf{b}$ we solve $\mathbf{P}\mathbf{A}\mathbf{x} = \mathbf{P}\mathbf{b}$, assuming that $\mathrm{cond}(\mathbf{P}\mathbf{A}) \ll \mathrm{cond}(\mathbf{A})$, which greatly accelerates the solution by such iterative methods.

   By the way, often for large sparse matrices preconditioners are constructed as an **incomplete LU decomposition**: that is, for the original matrix one constructs $\mathbf{A} \approx \mathbf{L}\mathbf{U}$, but tries to "throw out" new nonzero elements when constructing $\mathbf{L}\mathbf{U}$. In that case, applying this matrix as something similar to the original is still possible, so by solving with it we precondition the original problem and improve convergence.

9.3 Two worlds: dense and sparse problems

In fact, most solution algorithms can be divided into two classes:

1. When we are prepared to spend $\mathcal{O}(N^3)$ arithmetic operations and we have $\mathcal{O}(N^2)$ memory.
2. When we need to squeeze, and the matrix is so huge and has some special structure that we can multiply by it, but taking it in full form is very difficult.

In the first case — these are **decomposition methods** (LU, QR, Cholesky). In the second — these are **iterative methods**.

Moreover, because of the specific structure of modern processors and memory, iterative methods have somewhat worse performance than full decomposition methods. Therefore applying iterative methods to very small matrices is practically never justified.

9.4 Examples from quantum chemistry: DFT

Example 1: A small system. A small system of a few electrons by the DFT method, and the size of the Hamiltonian is about 1000×1000. We fit in memory (that is only ~ 8 MB for double precision). We can find all eigenvalues and the set of eigenvectors we are interested in, and for this the obvious choice is a direct method (for example, Schur decomposition or the QR algorithm).

Example 2: A large chemical structure. A rather large chemical structure by the DFT method, in which there are about a thousand electron pairs in the outer orbitals. For this we have constructed the Hamiltonian of the Schrödinger equation, and we must find one eigenvector for each electron pair. And the Hamiltonian is given such that we have taken about a hundred basis functions for each orbital, that is, the dimension of this Hamiltonian is about $100\,000 \times 100\,000$.

This seems not so much, but the matrix of such a Hamiltonian alone already occupies about **80 GB** in RAM, and the solution itself will take almost a gigabyte more. Here an iterative solution is much more obvious (for example, the Lanczos method or Davidson), which finds only the few eigenvectors needed, without working with the full matrix.

9.5 Do not reinvent the wheel: libraries

Nowadays in many programming languages there are well and for years polished libraries for solving such systems of equations, finding singular values and vectors, and eigenvectors and eigenvalues. It is enough to look in the documentation for **NumPy** for Python, and in **LAPACK/BLAS** for C/C++ — and everything will immediately become clear.

Moreover, the code is very well optimised for modern computers and rather complicated. For example, the modern version of SVD in LAPACK contains about **half a million lines of code**, and repeating it or even doing better is really an almost impossible task.

Practical advice

Never write your own solvers for linear algebra, unless it is a study task. Use:

- **Python:** NumPy (`numpy.linalg`), SciPy (`scipy.linalg`)
- **C/C++:** LAPACK, BLAS, Eigen
- **Fortran:** LAPACK (this is its native element)
- **MATLAB:** built-in functions (`eig`, `svd`, `lu`, `qr`)

These libraries have gone through decades of optimisation and testing. They know about the processor cache, about SIMD instructions, about multithreading — all that you cannot repeat by hand.

10 Nonlinear operators: when the world ceases to be straight

10.1 What is a nonlinear operator?

Until now we have spoken of linear operators — those that can be written as $\mathbf{Ax}$ or $\int K(x,y)g(y)\mathrm{d}y$. But the real world is nonlinear.

A **nonlinear operator** is a mapping $\mathcal{F}$ that acts on a function or vector $\mathbf{x}$ but which *does not* possess the properties of additivity and homogeneity:

$$\mathcal{F}(\alpha\mathbf{x} + \beta\mathbf{y}) \neq \alpha\mathcal{F}(\mathbf{x}) + \beta\mathcal{F}(\mathbf{y})$$

Roughly speaking, let us now define that we have some function $f(\mathbf{x})$ that depends on one or several parameters. And we want either:

- To solve the equation $f(\mathbf{x}) = 0$ (a system of nonlinear equations),
- To find the minimum $\min_{\mathbf{x}} f(\mathbf{x})$ (an optimisation problem).

10.2 An example from chromatography: the plate model

A classical example is the system of equations of balance and dissociation constants on each chromatographic plate.

We want to model a chromatograph, or rather its column. Separation occurs in it, and the mobile phase, moving along the column, at each step in fact creates such an equilibrium.

You may say — oh, there are only some 4–5 equations here, and just as many unknowns! Yes, I agree, not many. But:

1. There are many plates (say a thousand) — that is one.
2. We split the time of passage of the substance along the column into many time steps — that is two.

That is, at each time step we have a thousand times the same system of equations with different parameters of the input concentrations. And there will be substantially more such time steps than the number of plates, since the substance is after all retained by the column.

That is, we must solve something like **ten million times** a system of equations of only 5 equations with 5 unknowns. But if we fail to solve it even once — we cannot obtain any further solution.

Unfortunately, such a system is not linear. The balance equations are linear, but the equations relating the dissociation constants contain products and ratios of the unknowns with respect to each other. And, surprisingly, such a system can sometimes indeed be solved very poorly.

10.3 Classification of nonlinear problems

Before speaking of methods, let us classify the problems themselves:

Property	Description
Smoothness:	
Smooth	The function has continuous derivatives (at least the first, and preferably the second). Example: $f(x) = x^2 + \sin x$.
Almost smooth	The function is smooth almost everywhere, but there are break points or discontinuities of derivatives. Example: $f(x) = \lvert x \rvert$.
Discontinuous	The function has discontinuities or is very noisy. Example: experimental data with artefacts.
Number of minima:	
Unimodal	There is only one global minimum (or maximum). Example: convex functions.
Multimodal	There are several local minima, and the task is to find the global one. Example: potentials of complex molecules.

10.4 Methods for solving nonlinear problems

Now let us systematise the solution methods. Each method requires something, depends on something, and each has its strengths and weaknesses.

10.4.1 Monte Carlo methods

Idea: Random search. We generate random points in the parameter space, evaluate the function at them, and choose the best.

Requirements: None. It can work even with discontinuous functions.

Pros:

- Does not get stuck in local minima (with a sufficient number of iterations),
- Simple to implement,
- Parallelises perfectly.

Cons:

- Very slow convergence: the error decreases as $\mathcal{O}(1/\sqrt{N})$, where N is the number of iterations,
- Does not use information about the structure of the function.

When to use: When the function is very noisy, discontinuous, or when one simply needs to find "some" solution to initialise other methods.

10.4.2 Gradient methods (Gradient Descent)

Idea: Move in the direction opposite to the gradient:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \nabla f(\mathbf{x}_k)$$

where α_k is the learning rate.

Requirements: The function must be differentiable (smooth).

Pros:

- Simple to implement,
- Guaranteed convergence to a local minimum (with a correct choice of α),
- Works well in high dimensions.

Cons:

- Gets stuck in local minima,
- Can converge very slowly in "ravines" (when the eigenvalues of the Hessian differ greatly),
- Requires choosing the step α (too large — diverges, too small — slow).

10.4.3 Conjugate direction method (Conjugate Gradient for optimisation)

Idea: Construct search directions so that they are "conjugate" with respect to the Hessian of the function. This allows finding the minimum of a quadratic function in N steps (where N is the dimension).

Important remark: The conjugate gradient method for solving linear systems iteratively is *only named* the same as the conjugate gradient method for nonlinear minimisation. In fact, it is the same algorithm, simply applied to different problems:

- For linear systems $\mathbf{Ax} = \mathbf{b}$ — this is the minimisation of the quadratic function $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{Ax} - \mathbf{b}^T\mathbf{x}$,
- For nonlinear optimisation — this is a generalisation of the same idea to arbitrary functions.

Requirements: A smooth function, preferably with a continuous gradient.
Pros:

- Converges faster than ordinary gradient descent,
- Does not require storing the Hessian (unlike Newton's method),
- Memory — $\mathcal{O}(N)$.

Cons:

- Gets stuck in local minima,
- For non-quadratic functions a "reset" of directions is required.

10.4.4 Newton's methods

Idea: Use not only the gradient but also the second derivatives (the Hessian). We expand the function in a Taylor series up to second order:

$$f(\mathbf{x}+\mathbf{h}) \approx f(\mathbf{x}) + \nabla f(\mathbf{x})^T \mathbf{h} + \frac{1}{2}\mathbf{h}^T \mathbf{H}(\mathbf{x})\mathbf{h}$$

and find the minimum of this quadratic approximation. We obtain the iteration:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{H}^{-1}(\mathbf{x}_k)\nabla f(\mathbf{x}_k)$$

where $\mathbf{H}$ is the matrix of second derivatives (the Hessian).
Requirements: The function must be twice differentiable. The Hessian must be computable.
Pros:

- **Quadratic convergence:** if we start close to the solution, the number of correct digits doubles at each iteration,
- Not sensitive to "ravines" (unlike gradient descent).

Cons:

- Requires computing the Hessian — $\mathcal{O}(N^2)$ elements,
- Requires solving a linear system with the Hessian — $\mathcal{O}(N^3)$ operations,
- May diverge if we start far from the solution,
- The Hessian may be degenerate or not positive definite.

10.4.5 The Newton–Raphson method and block Newton for quantum mechanics

The **Newton–Raphson method** is a variant of Newton's method for solving systems of nonlinear equations $\mathbf{F}(\mathbf{x}) = \mathbf{0}$:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{J}^{-1}(\mathbf{x}_k)\mathbf{F}(\mathbf{x}_k)$$

where $\mathbf{J}$ is the Jacobian matrix (the matrix of first derivatives $\partial F_i/\partial x_j$).

Block Newton for quantum mechanics: In quantum chemistry problems (for example, in the Hartree–Fock method or DFT) a system of equations often arises that can be split into blocks:

- Equations for the orbitals (expansion coefficients),
- Equations for the density,

- Equations for the energy.

The block Newton method takes this structure into account: the Hessian is represented in block form, and each block is processed separately. This allows:

- Efficient use of the structure of the problem,
- Parallelisation of computations,
- Applying different methods to different blocks (for example, Newton for one block, gradient descent for another).

10.4.6 BFGS and L-BFGS methods

BFGS (Broyden–Fletcher–Goldfarb–Shanno) is a quasi-Newton method. The idea: not to compute the Hessian explicitly, but to approximate it at each iteration using information about the change of the gradient.

Requirements: A smooth function with a continuous gradient.

Pros:

- Convergence almost like Newton's, but without computing the Hessian,
- Guaranteed positive definiteness of the Hessian approximation,
- Works well in practice — one of the most popular methods.

Cons:

- Requires storing an $N \times N$ matrix (the Hessian approximation) — $\mathcal{O}(N^2)$ memory.

L-BFGS (Limited-memory BFGS) is a modification of BFGS for large problems. The idea: not to store the full matrix, but to store only the last m pairs of vectors $(\mathbf{s}_k, \mathbf{y}_k)$, where:

$$\mathbf{s}_k = \mathbf{x}_{k+1} - \mathbf{x}_k, \quad \mathbf{y}_k = \nabla f_{k+1} - \nabla f_k$$

Pros:

- Memory — $\mathcal{O}(mN)$, where $m \sim 5$–20 (does not depend on N^2!),
- Ideal for large problems ($N > 1000$),
- Very popular in machine learning and quantum chemistry.

Cons:

- Converges slightly more slowly than full BFGS,
- Requires tuning the parameter m.

10.4.7 Simplex methods (Nelder–Mead)

Idea: We construct a simplex (in N-dimensional space these are $N + 1$ points), evaluate the function at the vertices, and iteratively reflect, contract or expand the simplex, moving towards the minimum.

Requirements: None! The function can be nonsmooth, noisy, even discontinuous.

Pros:

- Does not require gradients,
- Simple to implement,

- Works well for small dimensions ($N < 10$).

Cons:

- Very slow for large N,
- May get stuck,
- No theoretical guarantees of convergence.

When to use: When the function is very noisy or when N is small and one is too lazy to derive gradients.

10.4.8 Simulated Annealing

Idea: Inspired by the physical process of annealing of metals. We start with a high "temperature" T, which allows the algorithm to "jump over" local minima. We gradually lower T, and the algorithm "freezes" in the global minimum.

At each step:

1. We propose a random change $\mathbf{x} \to \mathbf{x}'$,
2. We compute $\Delta f = f(\mathbf{x}') - f(\mathbf{x})$,
3. If $\Delta f < 0$ — we accept the change,
4. If $\Delta f > 0$ — we accept with probability $P = \exp(-\Delta f / T)$.

Requirements: None.
Pros:

- Can find the global minimum (with a correct "annealing schedule"),
- Does not get stuck in local minima in the early stages,
- Simple to implement.

Cons:

- Very slow,
- Requires tuning the schedule $T(t)$,
- No guarantees of convergence in reasonable time.

When to use: When the problem is multimodal (many local minima) and the global one must be found.

10.5 Computing gradients

All gradient methods require computing $\nabla f(\mathbf{x})$. How is this done?

10.5.1 Finite differences

Idea: We approximate the derivative:

$$\frac{\partial f}{\partial x_j} \approx \frac{f(\mathbf{x} + h\mathbf{e}_j) - f(\mathbf{x})}{h}$$

where $\mathbf{e}_j$ is the j-th basis vector.

Problems:

1. **Expensive:** To compute the gradient in N-dimensional space we need $N + 1$ evaluations of the function (or $2N$ for the central difference).
2. **Unstable:** If h is too large — a large approximation error. If h is too small — loss of accuracy due to subtracting close numbers. The optimal $h \sim \sqrt{\epsilon_{\text{mach}}}$, where ϵ_{mach} is the machine precision.
3. **Noise:** If the function is computed with noise (for example, experimental data), then finite differences amplify the noise.

10.5.2 Automatic differentiation (AD): analytic computation of the gradient

And now — magic! It turns out that one can compute the gradient of a function *analytically*, using automatic differentiation, and do it in only **a few times** the cost of computing the function itself!

Idea: Any function $f(\mathbf{x})$ is a composition of elementary operations ($+$, $-$, $\times$, $\div$, sin, cos, exp, log, etc.). We can:

1. Represent the computation of the function as a **computational graph**,
2. Apply the **chain rule** in reverse order (reverse mode).

Result: The gradient is computed at $\mathcal{O}(1)\times$ the cost of the function (in practice — 3–5 times more expensive, but not N times!).

How it works in a nutshell:

1. Forward pass: we compute $f(\mathbf{x})$, saving all intermediate results.
2. Reverse pass: we go through the graph in the reverse direction, applying the chain rule:

$$\frac{\partial f}{\partial x_j} = \sum_{\text{paths}} \frac{\partial f}{\partial u_1} \frac{\partial u_1}{\partial u_2} \cdots \frac{\partial u_k}{\partial x_j}$$

Where it is used:

- **PyTorch, TensorFlow:** The basis of all modern neural networks is precisely reverse-mode automatic differentiation.
- **Quantum chemistry:** Programs like PySCF, Psi4 use AD to compute energy gradients with respect to nuclear coordinates.
- **Optimisation:** All modern libraries (SciPy, JAX) use AD.

Main conclusion

Never compute gradients by finite differences if automatic differentiation can be used! It is faster, more accurate and more stable.

10.6 A live example: force fields and conformers

Another live example from chemistry is molecular modelling. For modelling small molecules one often uses a representation in which the total energy is written as a sum of simple interactions:

1. **Van der Waals interactions** between non-bonded atoms (Lennard-Jones potential):

$$E_{\text{vdW}} = \sum_{i<j} 4\varepsilon_{ij} \left[\left(\frac{\sigma_{ij}}{r_{ij}} \right)^{12} - \left(\frac{\sigma_{ij}}{r_{ij}} \right)^{6} \right]$$

2. **Harmonic bonds** between bonded atoms:

$$E_{\text{bond}} = \sum_{\text{bonds}} \frac{1}{2} k_b (r - r_0)^2$$

3. **Angle interactions** between three consecutive atoms:

$$E_{\text{angle}} = \sum_{\text{angles}} \frac{1}{2} k_\theta (\theta - \theta_0)^2$$

4. **Torsional (dihedral) interactions** between four consecutive atoms:

$$E_{\text{torsion}} = \sum_{\text{dihedrals}} \frac{V_n}{2} [1 + \cos(n\phi - \gamma)]$$

There will not be many of them, but about the square of the number of atoms (for van der Waals). Each such function is some simple formula: sometimes with sines, sometimes with exponentials, sometimes with powers. And in total a molecule has $3N$ degrees of freedom, since each atom has 3 spatial coordinates.

10.6.1 The conformer problem

It would seem the task is clear: find the minimum of this function $E(\mathbf{x})$, where $\mathbf{x} \in \mathbb{R}^{3N}$ are the coordinates of all atoms. But here the most interesting begins.

The energy function of a molecule is a **multimodal landscape** with a huge number of local minima. Each local minimum corresponds to its own **conformation** (conformer) of the molecule — its own way of twisting the molecule in space.

For example, for a protein of 100 amino acids the number of possible conformers can reach 10^{100} — this is the famous **Levinthal paradox**. And the global minimum (the native structure) is only one of 10^{100} possibilities.

10.6.2 Why simple minimisation does not work

If we take a random initial conformation and run ordinary gradient descent or BFGS, we will very quickly (within a few hundred iterations) converge to the *nearest* local minimum. But this will almost certainly *not* be the global minimum — that is, not the conformation the molecule adopts in reality.

10.6.3 Solution: a combination of methods

Therefore in practice a combination of methods is used:

1. **Simulated Annealing:** We start with a high “temperature”, which allows the algorithm to jump over energy barriers and explore different regions of conformational space. We gradually lower the temperature, “freezing” the system in a low-energy region.

2. **Local minimisation (BFGS):** After simulated annealing has found a "good" region, we run a fast local method (BFGS or the conjugate gradient method) for precise convergence to a local minimum.

3. **Repetition:** We run this combination many times with different initial conditions and choose the conformer with the lowest energy.

Popular programs

This approach is implemented in popular molecular dynamics programs:

- **AMBER, GROMACS, NAMD:** Use force fields (AMBER, CHARMM, OPLS) and methods like simulated annealing + local minimisation.
- **Rosetta:** For protein structure prediction it uses a fragment approach + Monte Carlo + local minimisation.

11 The fast Fourier transform: when $O(N^2)$ turns into $O(N \log N)$

11.1 The pattern search problem

Sometimes we need to find a certain fragment in a huge sequence. If it is a very short fragment, then a simple enumeration of the whole sequence and comparison with this short fragment is a rather good strategy. This is often done when searching for short fragments in protein structures.

But sometimes the dimensions of what must be compared are almost equal. For example, we have some periodic crystal structure, and we know that there is some deviation in it, and we even understand that the deviation is only partially similar to what we want to compare with.

Formally: the initial data are v_i, $i = 1, \ldots, N$, and what must be compared is w_j, $j = 1, \ldots, M$, where $M < N$.

We can explicitly compute in a loop:

$$\forall k = 0, \ldots, N - M : \quad s_k = \sum_{j=1}^{M} w_j v_{j+k}$$

and find the maximum over s.

But the computational complexity of such a search will be quadratic in N: if $M \simeq N/2$, then we must perform $N^2/2$ arithmetic operations. For $N = 10^6$ this is 5×10^{11} operations — too many!

11.2 Matrix formulation

Here people came up with a way to find such s_k much faster. Let us represent the computation of these s in matrix form. Suppose we have the matrix:

$$\mathbf{W} = \begin{pmatrix} w_1 & w_2 & \cdots & w_M & 0 & \cdots & 0 \\ 0 & w_1 & w_2 & \cdots & w_M & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & & \ddots & 0 \\ 0 & \cdots & 0 & w_1 & w_2 & \cdots & w_M \end{pmatrix}$$

of size $(N - M + 1) \times N$.

If we multiply it by our vector $\mathbf{v}$, we obtain precisely our coveted s_k:

$$\mathbf{s} = \mathbf{W}\mathbf{v}$$

But how do we do this quickly?

11.3 Circulant matrix

If we supplement this matrix from below with another $M-1$ rows of the form:

$$\begin{aligned} &w_M, 0, \ldots, 0, w_1, \ldots, w_{M-1} \\ &w_{M-1}, w_M, 0, \ldots, 0, w_1, \ldots, w_{M-2} \\ &\vdots \\ &w_2, \ldots, w_M, 0, \ldots, 0, w_1 \end{aligned}$$

then after multiplication we obtain the same vector $\mathbf{s}$, but at the end $M-1$ more numbers will be appended to it, which we can discard.

But this extended matrix is a **circulant matrix C**! It has a remarkable property: each successive row is obtained by a cyclic shift of the previous one.

11.4 Diagonalisation of a circulant matrix

A circulant matrix has an interesting representation via the Fourier matrix:

$$\mathbf{C} = \frac{1}{N}\mathbf{F}^H \operatorname{diag}(\mathbf{Fc})\mathbf{F}$$

where:

- $\mathbf{c}$ is the first column of the original matrix $\mathbf{C}$,
- $\mathbf{F}$ is the discrete Fourier transform (DFT) matrix,
- $\mathbf{F}^H$ is the Hermitian conjugate (complex conjugate and transposed) matrix.

The Fourier matrix is defined as:

$$\mathbf{F} = \{f_{jk}\}_{j,k=0}^{N-1}, \quad f_{jk} = e^{-2\pi ijk/N}$$

11.5 Fast multiplication via FFT

Now the most interesting thing. If we can quickly multiply the Fourier matrix by a vector, then we can compute this $\mathbf{s}$ faster:

$$\mathbf{s} = \mathbf{Cv} = \frac{1}{N}\mathbf{F}^H \operatorname{diag}(\mathbf{Fc})\mathbf{Fv}$$

This computation consists of three steps:

1. $\mathbf{a} = \mathbf{Fv}$ — forward DFT of the vector $\mathbf{v}$,
2. $\mathbf{b} = \operatorname{diag}(\mathbf{Fc})\mathbf{a}$ — element-wise multiplication (this is $\mathcal{O}(N)$),
3. $\mathbf{s} = \frac{1}{N}\mathbf{F}^H\mathbf{b}$ — inverse DFT.

11.6 Decomposition of the Fourier matrix

How do we quickly multiply $\mathbf{F}$ by a vector? It turns out that $\mathbf{F}$ can be represented as a product of special matrices:

1. One permutation matrix (bit-reversal permutation),
2. Several pairs of matrices:
   - Block matrices of the form $\begin{pmatrix} \mathbf{I} & \mathbf{I} \\ \mathbf{I} & -\mathbf{I} \end{pmatrix}$,

- Diagonal matrices with elements $e^{-2\pi ik/N}$ (the so-called "twiddle factors").

Let us write out these matrices explicitly for $N = 8$:

Permutation matrix (bit-reversal):

$$\mathbf{P} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

(rows permuted according to bit reversal: $0, 4, 2, 6, 1, 5, 3, 7$)

Block matrix (first level):

$$\mathbf{B}_1 = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 \end{pmatrix}$$

Diagonal matrix (twiddle factors, first level):

$$\mathbf{D}_1 = \operatorname{diag}\left(1, 1, 1, 1, 1, e^{-2\pi i/8}, e^{-4\pi i/8}, e^{-6\pi i/8}\right)$$

And so on for $\log_2 N$ levels.

Then each such multiplication will require only N and $2N$ arithmetic operations, and the total number of such steps will be $\log_2 N$.

Total: multiplying the Fourier matrix by a vector can be done in $\mathcal{O}(N \log_2 N)$ arithmetic operations!

And hence all s_k can be computed in the same $\mathcal{O}(N \log_2 N)$, and not in $\mathcal{O}(N^2)$.

For $N = 10^6$:

- Naive algorithm: 10^{12} operations,
- FFT: 2×10^7 operations (50 000 times faster!).

11.7 The fast Fourier transform (FFT)

This is the famous **fast Fourier transform** (FFT), discovered by Cooley and Tukey in 1965 (although Gauss knew it already in 1805!).

It is one of the most in-demand algorithms in modern chemistry.

11.7.1 Application in NMR

With its help, for example, the initial FID (Free Induction Decay) from NMR is transformed into the spectral domain.

If one looks at each row of the Fourier matrix, one can see an oscillating function in it:

- The closer the row is to the centre of the matrix, the greater the oscillation,
- The first row has none at all — it is simply a constant,

- The very bottom (or second) row has an oscillation period equal to the length of the row.

That is precisely why, having multiplied the Fourier matrix by the FID, we obtain maxima where the resonant frequencies are: it is simply that such a row coincided in oscillation frequency with the resonant frequency.

Why FFT is so important for NMR

Modern FIDs are very long — they are often digitised into hundreds of thousands or even millions of numbers. If the fast Fourier transform did not exist, multiplication by such a matrix would last on modern computers even longer than the NMR spectra acquisition itself — that is, really minutes and tens of minutes even on modern workstations.
Thanks to FFT the transformation takes **fractions of a second**.

11.7.2 Other applications of FFT in chemistry

- **Cryo-electron microscopy:** Reconstruction of 3D structures from 2D projections.
- **X-ray structure analysis:** Transformation of the diffraction pattern into electron density.
- **Molecular dynamics:** Computation of electrostatic interactions via the PPPM method (Particle-Particle Particle-Mesh).
- **Signal processing:** Noise filtering, data compression.
- **Chemometrics:** Fast convolution and correlation of spectra.

Summary

FFT is one of those algorithms that changed the world. Without it there would be no modern spectroscopy, signal processing, audio and video compression, and much else. It is a must-know for any chemist working with data.

12 When FFT does not see the obvious: spectral leakage and the Prony method

12.1 The paradox: the eye sees a sinusoid, but FFT does not

Although FFT is a must-have algorithm almost everywhere in experimental chemistry, it too has its own features that one must definitely pay attention to.

Let us consider an example. We have recorded a spectrum, the signal oscillates, and this is directly visible to the eye in the graph. But we managed to record only slightly more than one period. When we applied FFT, we obtained something very strange: the peak seems to be there, but also seems not to be — it turned out somehow very smeared. And if we had a lot of other noise, then we obtained in another part of the spectrum something very similar, and yes, accidentally the noise turned out “brighter” than the signal, and our program produced a completely wrong result.

But we see this sinusoid with our eyes! Why does FFT not see it?

12.2 Spectral leakage

FFT is good then, and *only then*, when we have applied it to a signal in which **an exact integer number of periods** of this signal fits.

Why? Because FFT implicitly assumes that our finite signal *is periodically continued* beyond the measurement window. If exactly k full periods fit in the window, then the periodic continuation will be smooth, and FFT will give one clear peak.

But if the signal contains, say, 1.4 periods, then in the periodic continuation a **discontinuity** arises at the boundary of the window. FFT is forced to approximate this discontinuity by a multitude of harmonics — and energy "leaks" from the main peak into all other frequencies. This phenomenon is called **spectral leakage**.

Mathematically: if the true frequency of the signal falls *between* two neighbouring frequency bins of the FFT, then instead of one delta function we obtain a smeared "hump" (a function of the form $\sin(x)/x$, the so-called *sinc*).

Rule of FFT

FFT sees only those frequencies that fit into the measurement window an integer number of times. Everything else is leakage.

12.3 Zero-padding: a simple but not ideal solution

What should we do then?

The simplest option is to add many zeros at the end of the signal (zero-padding) and hope that on the whole we will be substantially luckier. After all, if, for example, we add zeros so that the original data are stretched 5 times, then we will definitely get 7 periods (instead of 1.4). And if we increase, say, 7 times, then there will be 9.8 periods — also very close to an integer.

People often add a different number of zeros, and then try to combine the resulting spectra, guessing which peaks turned out more accurate in the corresponding extended spectra. This really helps!

Important nuance of zero-padding

Zero-padding *does not increase the real frequency resolution* — it only interpolates the spectrum, making it smoother. The resolution is determined by the *length of the original signal*, and not by the length of the zero-padded one. But in practice zero-padding helps to "hit" an integer number of periods and reduce leakage.

12.4 Frequency drift: when the signal "drifts away"

But there is another problem that zero-padding cannot cope with.

Sometimes we measure some spectrum for a long and tedious time, and it "slightly" took and drifted. That is, at the beginning we had frequency ω, and towards the end $\omega + \varepsilon$, and this small correction is enough for us to completely miss with the FFT and obtain instead of a clear single peak something very smeared.

But after all, one can see with the eye — there is the sinusoid, both here and there, both at the beginning and at the end! Only FFT again does not see it.

The reason is that FFT assumes **stationarity** of the signal — that is, that the frequencies do not change with time. If the frequency drifts, then no harmonic of the FFT can approximate the whole signal well.

What should we do?

12.5 The Prony method: the idea of shifts

Several centuries ago Gaspard de Prony (1795) answered this question for us. More precisely, he invented how to solve his own problem (expansion of a signal into a sum of exponentials), but it precisely solves our problem too — when the spectrum drifts slightly in time.

Idea: Take the signal and store it in a vector $\mathbf{s} = (s_0, s_1, \ldots, s_{N-1})^T$. Then shift this vector down by one sample, then again, and put it side by side again. Do this L times.

What do we have here?

If we had a periodic signal with some unknown sinusoid $\sin(\omega t)$, then after a shift by p samples we obtain $\sin(\omega t + \omega p)$. Recalling the addition formulas from Chapter 1:

$$\sin(\omega t + \omega p) = \sin(\omega t)\cos(\omega p) + \cos(\omega t)\sin(\omega p)$$

Since $\cos(\omega p)$ and $\sin(\omega p)$ are *constants* for the whole vector (they do not depend on t), the set of such shifted vectors will have only **as many nonzero singular values as twice the number of distinct oscillating harmonics**.

Moreover, if some frequency has "drifted" a little with time, this *does not affect at all* such an approximation, but completely destroys the FFT result, as we noted above.

12.6 Prony method, variant 1: linear prediction

Let our signal be modelled as a sum of K complex exponentials:

$$s_n = \sum_{m=1}^{K} c_m z_m^n, \quad n = 0, 1, \ldots, N-1$$

where $z_m = e^{(\alpha_m + \mathrm{i}\omega_m)\Delta t}$ are complex "frequencies" (containing both the damping α_m and the oscillation ω_m), and c_m are complex amplitudes.

Key fact: If the signal consists of K exponentials and there is no noise, then it satisfies a **linear recurrence relation** of order K:

$$s_n + a_1 s_{n-1} + a_2 s_{n-2} + \cdots + a_K s_{n-K} = 0 \quad \forall n \geq K$$

This means that the $(n+1)$-th sample can be *exactly predicted* from K previous ones!

The coefficients $a_1, \ldots, a_K$ are the coefficients of the characteristic polynomial whose roots are the z_m:

$$P(z) = z^K + a_1 z^{K-1} + a_2 z^{K-2} + \cdots + a_K = \prod_{m=1}^{K} (z - z_m)$$

How to find the coefficients? Let us write the recurrence relation for all available samples in matrix form:

$$\underbrace{\begin{pmatrix} s_{K-1} & s_{K-2} & \cdots & s_0 \\ s_K & s_{K-1} & \cdots & s_1 \\ \vdots & \vdots & \ddots & \vdots \\ s_{N-2} & s_{N-3} & \cdots & s_{N-K-1} \end{pmatrix}}_{\mathbf{S}\ (N-K)\times K} \underbrace{\begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_K \end{pmatrix}}_{\mathbf{a}} = -\underbrace{\begin{pmatrix} s_K \\ s_{K+1} \\ \vdots \\ s_{N-1} \end{pmatrix}}_{\mathbf{s}_{\text{future}}}$$

This is an overdetermined system (if $N > 2K$), and we solve it by the least squares method:

$$\mathbf{a} = -(\mathbf{S}^H \mathbf{S})^{-1} \mathbf{S}^H \mathbf{s}_{\text{future}}$$

After finding $\mathbf{a}$ we look for the roots of the polynomial $P(z)$ — these are our z_m, from which the frequencies ω_m and dampings α_m are extracted.

12.7 Prony method, variant 2: SVD of the shift matrix

The second variant is more robust to noise and more elegant.

Step 1: Construct the Hankel matrix from the signal shifts.

$$\mathbf{H} = \begin{pmatrix} s_0 & s_1 & \cdots & s_{L-1} \\ s_1 & s_2 & \cdots & s_L \\ \vdots & \vdots & \ddots & \vdots \\ s_{N-L} & s_{N-L+1} & \cdots & s_{N-1} \end{pmatrix}$$

of size $(N - L + 1) \times L$, where $L > K$ (we choose L definitely larger than the expected number of harmonics).

Step 2: Do SVD.

$$\mathbf{H} = \mathbf{U}\square\mathbf{V}^H$$

If the signal consists of K exponentials and there is no noise, then $\text{rank}(\mathbf{H}) = K$, and the last $L - K$ singular values are zero:

$$\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_K > 0, \quad \sigma_{K+1} = \cdots = \sigma_L = 0$$

In the presence of noise the small singular values are not zero, but they are *substantially smaller* than the first K. We discard them (this is called **truncated SVD** or **regularisation**).

Step 3: Extract the polynomial from the null space.

The singular vector $\mathbf{v}_{\text{min}}$ corresponding to the *smallest* singular value (the last column of $\mathbf{V}$) lies in the (approximate) null space of the matrix $\mathbf{H}$. This means:

$$\mathbf{H}\mathbf{v}_{\text{min}} \approx \mathbf{0}$$

If we write the components $\mathbf{v}_{\text{min}} = (v_0, v_1, \ldots, v_{L-1})^T$, this relation is equivalent to:

$$\sum_{j=0}^{L-1} v_j s_{n+j} \approx 0 \quad \forall n$$

That is, $\mathbf{v}_{\text{min}}$ contains the coefficients of the polynomial:

$$Q(z) = v_0 + v_1 z + v_2 z^2 + \cdots + v_{L-1} z^{L-1}$$

Step 4: Find the roots of the polynomial.

The roots of $Q(z)$ contain K "signal" roots $z_m = e^{(\alpha_m + \mathrm{i}\omega_m)\Delta t}$ (lying inside or on the unit circle) and $L - K$ "noise" roots (scattered randomly).

From the signal roots we extract:

$$\omega_m = \frac{\arg(z_m)}{\Delta t}, \quad \alpha_m = \frac{\ln|z_m|}{\Delta t}$$

Why is the SVD variant better?

The SVD variant of the Prony method is more robust to noise, because:

1. Truncation of small singular values automatically filters the noise.
2. We do not solve the overdetermined system directly (which may be ill-conditioned), but use an orthogonal decomposition.
3. The number of harmonics K is determined automatically from the "jump" in the spectrum of singular values.

12.8 Limitations of the Prony method

But Prony is not a panacea.

As soon as a component that is well approximated both by the function itself and by its shifts (for example, white noise or a very broadband signal) appears in the original signal, we immediately obtain a "root" for it, and then it turns out to be **spurious**.

Other limitations:

- The method assumes that the signal is a sum of *exponentials* (including sinusoids as a special case). If the signal has a different structure (for example, rectangular pulses), the method will work poorly.
- The number of harmonics K must be known or guessed in advance (although SVD helps with this).
- Finding the roots of a high-degree polynomial ($L > 50$) may itself be numerically unstable.
- The method is sensitive to strong noise: at a low signal-to-noise ratio spurious roots can "masquerade" as genuine ones.

At the same time this method is very actively and quite long used in NMR spectroscopy and complements FFT well. In practice a **hybrid approach** is often used:

1. FFT for a quick overview of the spectrum and determination of the number of peaks,
2. The Prony method (or its modern variants: MUSIC, ESPRIT, Matrix Pencil) for precise determination of frequencies, dampings and amplitudes of individual peaks.

Summary

FFT is a powerful but "blind" tool: it sees only what fits into its frequency grid. The Prony method "sees" frequencies between bins and is robust to drift, but requires more computations and caution. In real NMR spectroscopy they work in tandem.

13 Least squares, residuals and Tikhonov regularisation

13.1 Statement of the problem

So, we often solve problems that we call least squares problems. Let us consider some of them to understand more precisely how this can be solved.

13.2 An example from medicine: computed tomography (CT)

Let us take an example from an adjacent field — from medicine. Suppose we have an X-ray CT scanner. We have placed the patient (or whatever we are investigating) immobile on some platform, and around him on opposite sides we place a transmitter and receivers of X-rays. We perform measurements of how strongly the radiation has been absorbed by the body under investigation, and we place these receivers and transmitters at various angles.

Usually the patient lies on a couch, and the receiver and transmitters are located on the surface of a ring. This ring is positioned so that its axis is approximately along the couch, and the ring itself both rotates about its axis and moves along the couch.

What is happening here?

Soft tissues hardly absorb X-rays, while bones absorb quite strongly.

We can mentally divide the whole space where the patient is located into small (usually equal) cubes — **voxels** (volume elements). If we know exactly the position of the transmitter and receiver, we can draw a line (the X-ray beam) that intersects our cubes.

Let each such cube have one unknown — the intensity of absorption of X-rays. We number all these cubes and write the intensity of their absorption as an unknown vector $\mathbf{x} = (x_1, \ldots, x_N)^T$.

Then one mutual arrangement of transmitter and receiver gives us a rather sparse set of coefficients — the lengths of passage of the beam through the corresponding cube. Let all this be written in a row of the matrix $\mathbf{A}$, and the number of columns in it equals N (the number of voxels), and the number of rows equals the number of measurements performed (let it be M).

But the measured values themselves (we may have, for example, one X-ray source and many receivers; then each mutual arrangement of receiver and source is one row) will be written in the vector $\mathbf{b}$, also of length M.

Then we can formulate the least squares problem as:

$$\min_{\mathbf{x}} \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2$$

that is, we want each row of the matrix $\mathbf{A}$, multiplied by $\mathbf{x}$ (the total absorption intensity), to be equal to what we measure.

13.3 Normal equations

We remember that such a problem can be solved, but let us look at it carefully.

The residual function:

$$E = \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2^2 = (\mathbf{A}\mathbf{x} - \mathbf{b})^H(\mathbf{A}\mathbf{x} - \mathbf{b}) = \mathbf{x}^H\mathbf{A}^H\mathbf{A}\mathbf{x} - 2\mathbf{x}^H\mathbf{A}^H\mathbf{b} + \|\mathbf{b}\|_2^2$$

If we find the derivative of E with respect to all x_j and equate it to zero (at the minimum), we obtain the system of equations:

$$2\mathbf{A}^H\mathbf{A}\mathbf{x} - 2\mathbf{A}^H\mathbf{b} = 0 \quad \Rightarrow \quad (\mathbf{A}^H\mathbf{A})\mathbf{x} = \mathbf{A}^H\mathbf{b}$$

These are the so-called **normal equations**.

Here everything seems fine: we obtain a positive semi-definite matrix $\mathbf{A}^H\mathbf{A}$ and some right-hand side, with which we can solve this problem.

13.4 The degeneracy problem

But what happens if we have constructed such cubes, but we had no experiment in which the X-ray beam passed through, for example, the i-th cube?

Obviously, then in the original matrix $\mathbf{A}$ the corresponding i-th column will contain only zeros, and the matrix $\mathbf{A}^H\mathbf{A}$ will have zeros both in the i-th column and in the i-th row, that is, it will guaranteed have one zero singular value.

Suppose this cube was somewhere in space, and there is no part of the body under investigation in it, that is — fortunately for us — we do not need this cube. But such a formulation *will spoil* the solution: we simply cannot solve this problem, since the matrix $\mathbf{A}^H\mathbf{A}$ will be degenerate.

Moreover, we cannot guarantee that even if $\mathbf{A}^H\mathbf{A}$ has no such zero columns and rows, its conditioning will be good. That is, instead of a solution we may obtain completely wrong numbers.

13.5 Solution via SVD and the pseudoinverse matrix

How should we act in this case?

Let us return to the singular value decomposition of the matrix $\mathbf{A} = \mathbf{U}\square\mathbf{V}^H$. We can notice that:

- If $\square$ has zeros on the diagonal, they relate to those cubes through which the X-ray beam did not pass.
- If there is something very small, it relates to those experiments where the X-ray beam only a few times and, possibly, only touching, hit some cube or set of cubes.

That is, we should refuse to take such data into account, but they are very difficult to filter out already at the stage of forming the matrix $\mathbf{A}$.

But since these data are uninformative, let us simply "throw them out"! That is, let us perform this SVD and zero out the small diagonal values in $\square$, and reconstruct $\mathbf{A}$.

This, of course, is good, but we still do not know how to solve such a problem, since $\mathbf{A}^H\mathbf{A}$ will also contain zero singular values.

But if for a square non-degenerate system $\mathbf{Ax} = \mathbf{b}$ one can represent the solution as:

$$\mathbf{A} = \mathbf{U}\square\mathbf{V}^H \quad \Rightarrow \quad \mathbf{x} = \mathbf{V}\square^{-1}\mathbf{U}^H\mathbf{b}$$

then our least squares problem too we could probably solve by analogy. True, where $\square$ has zeros, in $\square^{-1}$ we will not invert them, but write a zero there too.

Then it will be incorrect to call it inverse, and we shall call it **pseudoinverse**, denoting it as $\square^+$ (or $\mathbf{A}^+$ for the whole matrix).

13.6 The size problem: when SVD does not fit in memory

All would be fine, except that in real problems the matrix $\mathbf{A}$ can be huge. Often its elements are computed analytically, and there is no need to store them. And the dense matrix $\mathbf{U}$, whose dimension equals that of $\mathbf{A}$, usually does not fit in memory.

For example, for an ordinary CT scan cubes of size about 1 mm are used. In a second about 100–500 measurements occur on 100–500 receivers, and the whole measurement lasts about a minute. That is, N and M can easily be of the order of 10–100 million, and the full singular matrix will require 100 petabytes of memory, which is unreasonably much.

13.7 Tikhonov regularisation

One can notice that if to the degenerate matrix $\mathbf{A}^H\mathbf{A}$ we add the identity $\lambda\mathbf{I}$, such that λ is in the range of those very small singular values that we zeroed out, then such an addition can be written as:

$$\begin{aligned}\mathbf{A}^H\mathbf{A} + \lambda\mathbf{I} &= \mathbf{V}\square\mathbf{U}^H\mathbf{U}\square\mathbf{V}^H + \lambda\mathbf{V}\mathbf{V}^H \\ &= \mathbf{V}(\square^2 + \lambda\mathbf{I})\mathbf{V}^H\end{aligned}$$

And we see that we add this lambda to each singular value, "turning" a degenerate or ill-conditioned matrix into a well-conditioned one.

Moreover, for small λ (of the order of the small singular values) we almost do not change the large singular values, but the small ones we simply replace by λ.

Then, solving the system with such a matrix, we simply do not "spoil" the large singular values and substantially reduce the response of the small singular values.

So, the conditioning of the matrix has become substantially smaller, and we do not "spoil" the solution with the matrix. Knowing that the matrix is sparse, we can apply some iterative methods (even the conjugate gradient method) and converge very quickly.

In fact, such a λ is the so-called **Tikhonov regulariser**, since instead of the original problem we can solve the following:

$$\min_{\mathbf{x}} \|\mathbf{Ax} - \mathbf{b}\|_2^2 + \lambda\|\mathbf{x}\|_2^2$$

in fact requiring that both the residual and the norm of the solution be minimised simultaneously, and their mutual proportion is regulated precisely by the value of this λ.

There is much beautiful theory here, once derived in the 1970s by Tikhonov and his followers, but the essence remains simple: such an additional "regulariser" helps to solve **ill-posed** problems.

13.8 Iterative reduction of λ

Moreover, often at first λ is set rather large, then the conjugate gradient converges very quickly (the conditioning of the matrix will be very small), and then λ is reduced, each time using the previous solution as the initial value.

This allows:

- Quickly obtaining a "rough" solution with a large λ,
- Gradually refining it by reducing λ,
- Avoiding problems with conditioning in the early stages.

Summary

Tikhonov regularisation is a powerful tool for solving ill-conditioned least squares problems. It adds a penalty for the norm of the solution, which stabilises the solution and allows using iterative methods even for degenerate systems.

14 Basis functions: from finite differences to splines

14.1 The idea: approximate the unknown through the known

Until now we have worked with discrete vectors and matrices. But real problems — differential equations, integrals, wave functions — live in continuous space. How do we reduce an infinite-dimensional problem to a finite-dimensional one?

Answer: **basis functions**. We represent the unknown function $f(x)$ as a linear combination of known basis functions $\phi_k(x)$:

$$f(x) \approx \sum_{k=1}^{N} c_k \phi_k(x)$$

and look for the coefficients c_k. This turns the problem of finding a function into the problem of finding a vector $\mathbf{c} \in \mathbb{R}^N$.

14.2 The Ritz method (variational method)

One of the most general approaches is the **Ritz method**. Suppose we have a functional $E[f]$ that we want to minimise (for example, energy in quantum mechanics). We substitute the expansion:

$$f(x) \approx \sum_{k=1}^{N} c_k \phi_k(x)$$

and obtain a function of the coefficients:

$$E(\mathbf{c}) = E\left[\sum_{k=1}^{N} c_k \phi_k\right]$$

Now we minimise $E(\mathbf{c})$ over $\mathbf{c}$ — this is already an ordinary optimisation problem in $\mathbb{R}^N$.

14.3 Finite differences (Finite Difference Method, FDM)

The simplest choice of basis functions is **local constants** or **linear functions** on a uniform grid.

14.3.1 An example from chemistry: diffusion

Suppose we have the diffusion equation for the concentration of a substance $C(x,t)$:

$$\frac{\partial C}{\partial t} = D\frac{\partial^2 C}{\partial x^2}$$

We discretise space with step h: $x_j = jh$, and time with step Δt: $t_n = n\Delta t$.

We approximate the second derivative by the central difference:

$$\left.\frac{\partial^2 C}{\partial x^2}\right|_{x_j} \approx \frac{C_{j+1} - 2C_j + C_{j-1}}{h^2}$$

where $C_j^n \approx C(x_j, t_n)$.

Then the Euler scheme in time:

$$\frac{C_j^{n+1} - C_j^n}{\Delta t} = D\frac{C_{j+1}^n - 2C_j^n + C_{j-1}^n}{h^2}$$

This gives an explicit recurrence formula:

$$C_j^{n+1} = C_j^n + \frac{D\Delta t}{h^2}(C_{j+1}^n - 2C_j^n + C_{j-1}^n)$$

Stability

The explicit scheme is stable only for $\frac{D\Delta t}{h^2} \le \frac{1}{2}$. If the time step is too large — the solution "falls apart".

14.4 Finite elements (Finite Element Method, FEM)

A more complicated but more flexible approach is **finite elements**. We divide the domain into small elements (triangles, tetrahedra) and on each element approximate the function by **local polynomials** (usually linear or quadratic).

The basis functions are **piecewise linear "hats"** (hat functions), each of which equals 1 at one node and 0 at all others.

Advantages of FEM:

- Can work with complex geometries,
- Can adapt the grid (refine where the solution changes rapidly),
- Well theoretically grounded.

14.5 Basis functions in quantum chemistry: Gaussian orbitals

And now — the most interesting for chemists. In quantum chemistry (the Hartree–Fock method, DFT) we solve the Schrödinger equation:

$$\hat{H}\Psi = E\Psi$$

where $\hat{H}$ is the Hamilton operator, Ψ is the wave function.

We represent the molecular orbitals $\psi_i(\mathbf{r})$ as a linear combination of **atomic orbitals** (LCAO — Linear Combination of Atomic Orbitals):

$$\psi_i(\mathbf{r}) = \sum_{\mu=1}^{K} c_{\mu i}\phi_\mu(\mathbf{r})$$

where $\phi_\mu(\mathbf{r})$ are basis functions centred on the atoms.

14.5.1 Gaussian functions (Gaussian Type Orbitals, GTO)

In most modern programs (Gaussian, ORCA, Q-Chem) **Gaussian orbitals** are used as basis functions:

$$\phi_\mu(\mathbf{r}) = (x - X_A)^l (y - Y_A)^m (z - Z_A)^n \exp(-\alpha |\mathbf{r} - \mathbf{R}_A|^2)$$

where:

- $\mathbf{R}_A = (X_A, Y_A, Z_A)$ are the coordinates of atom A,
- l, m, n are the angular quantum numbers (s, p, d, f orbitals),
- α is the exponent (controls the "size" of the orbital).

Why Gaussian functions?

- **The product of Gaussians is again a Gaussian:** This is critically important for computing multi-centre integrals (electron-electron repulsion),
- **Analytic integrals:** All integrals are computed analytically,
- **Contractions:** Real atomic orbitals are approximated by a sum of several Gaussians (contractions), which improves accuracy.

14.5.2 The Ritz method in quantum chemistry

We substitute the LCAO expansion into the Hartree–Fock or DFT energy functional and minimise over the coefficients $c_{\mu i}$. This leads to the eigenvalue problem:

$$\mathbf{F}\mathbf{c}_i = \varepsilon_i \mathbf{S}\mathbf{c}_i$$

where:

- **F** is the Fock matrix (or the Kohn–Sham matrix in DFT),
- **S** is the overlap matrix ($S_{\mu\nu} = \langle \phi_\mu | \phi_\nu \rangle$),
- ε_i are the orbital energies,
- $\mathbf{c}_i$ are the expansion coefficients of the i-th molecular orbital.

This is a generalised eigenvalue problem, and it is solved iteratively (by the self-consistent field method, SCF).

Basis sets

Popular basis sets in quantum chemistry:

- **STO-3G:** Minimal basis (3 Gaussians per atomic orbital),
- **6-31G*:** Double-zeta quality with polarisation functions,
- **cc-pVTZ:** Correlation-consistent triple zeta (high accuracy).

The larger the basis — the more accurate the result, but the more expensive the computations ($\mathcal{O}(N^4)$ for Hartree–Fock, where N is the number of basis functions).

14.6 Splines: smooth piecewise polynomial functions

And now — about splines. These are basis functions widely used in signal processing, computer graphics and numerical methods.

14.6.1 What is a spline?

A **spline** is a piecewise polynomial function that is **smooth** (has continuous derivatives up to some order) at the stitching nodes.

The most popular is the **cubic spline** (degree 3, smoothness C^2 — the function, the first and the second derivatives are continuous).

14.6.2 B-splines (Basis Splines)

B-splines are a special set of basis splines with **compact support**. Each B-spline is nonzero only on a small interval.

Advantages:

- **Locality:** Changing one coefficient affects only a small region,
- **Numerical stability:** The matrices are well conditioned,
- **Recursive definition:** B-splines are constructed recursively (de Boor's formula).

14.6.3 Two-dimensional splines

B-splines are easily generalised to the two-dimensional case via the **tensor product**:

$$B_{ij}(x, y) = B_i(x) \cdot B_j(y)$$

This is used in:

- Image processing (smoothing, interpolation),
- Finite elements (higher orders),
- Computer graphics (surfaces).

14.6.4 Bézier splines

Bézier splines are parametric curves defined by **control points**. The curve does not pass through the control points, but is "attracted" to them.

The formula of a Bézier curve of degree n:

$$\mathbf{B}(t) = \sum_{i=0}^{n} \binom{n}{i} (1-t)^{n-i} t^i \mathbf{P}_i, \quad t \in [0, 1]$$

where $\mathbf{P}_i$ are the control points, and $\binom{n}{i}(1-t)^{n-i}t^i$ are the **Bernstein polynomials**.

Applications:

- Computer graphics (vector graphics, TrueType fonts),
- CAD systems (AutoCAD, SolidWorks),
- Animation and trajectories.

14.6.5 Connection with finite elements

Interestingly, B-splines with compact support **smoothly flow** into basis finite elements. If we take B-splines of degree 0 — these are piecewise constant functions (as in the simplest finite elements). Degree 1 — piecewise linear "hats". Degree 2 and higher — smoother bases.

This allows constructing **isoparametric finite elements** of higher orders, which give a more accurate approximation with fewer nodes.

Summary

Basis functions are the bridge between the continuous world of differential equations and the discrete world of computers. Finite differences are the simplest, finite elements are the most flexible, Gaussian orbitals are specialised for quantum chemistry, and splines are for smooth interpolation and graphics. Understanding their properties is the key to choosing the right method for your problem.

15 Integration: from analytics to Monte Carlo

15.1 Why do we need integration?

The integral is one of the most fundamental operations in mathematics and physics. We constantly compute:

- Areas and volumes,
- Mathematical expectations and variances,
- Normalisation constants in quantum mechanics,
- Multidimensional integrals in statistical physics,
- Convolutions in signal processing.

And if at school we learned to take integrals analytically, then in real life — especially in chemistry and physics — analytics often ends very quickly.

15.2 Analytic integration: when it works

Let us recall the basics. If we have a function $f(x)$ given analytically, and we know its antiderivative $F(x)$, then:

$$\int_a^b f(x)\,dx = F(b) - F(a)$$

For example:

$$\int_0^1 x^2\,dx = \frac{x^3}{3}\Big|_0^1 = \frac{1}{3}$$

Beautiful, exact, fast. But:

- Not all functions have an elementary antiderivative (for example, e^{-x^2} — the error integral),
- Not all functions are given analytically — often we have only a set of points (experimental data),
- In multidimensional cases analytics is almost always powerless.

15.3 Numerical integration in 1D: Simpson's method

What should we do if the integral cannot be taken analytically? Answer: approximate the function by something simple and integrate that simple thing.

In the previous chapter we learned to construct cubic splines. And here — we also construct polynomials, and integrate with such pieces.

15.3.1 Rectangle method

The simplest approach: divide the interval $[a, b]$ into N equal parts of width $h = (b-a)/N$, and on each interval approximate the function by a constant (the value at the midpoint):

$$\int_a^b f(x)\,dx \approx h \sum_{k=0}^{N-1} f\left(a + \left(k + \frac{1}{2}\right) h\right)$$

The error is $\mathcal{O}(h^2)$.

15.3.2 Trapezoidal method

A little better: we approximate the function by a linear polynomial on each interval:

$$\int_a^b f(x)\,dx \approx \frac{h}{2}\left[f(a) + 2\sum_{k=1}^{N-1} f(a+kh) + f(b)\right]$$

The error is $\mathcal{O}(h^2)$.

15.3.3 Simpson's method

Even better: we approximate the function by a **quadratic polynomial** on each pair of intervals:

$$\int_a^b f(x)\,dx \approx \frac{h}{3}\left[f(a) + 4\sum_{k=1,3,5,\ldots}^{N-1} f(a+kh) + 2\sum_{k=2,4,6,\ldots}^{N-2} f(a+kh) + f(b)\right]$$

where N is an even number.

The error is $\mathcal{O}(h^4)$! This is already very good for smooth functions.

Why is Simpson so good?

Simpson's method is, in essence, the integration of piecewise quadratic splines. If the function is smooth (has continuous derivatives up to 4th order), then the error decreases as h^4, which is much faster than for trapezoids.
For very smooth functions there are even more accurate methods — Gaussian quadratures, which use optimally chosen nodes and weights.

15.4 The curse of dimensionality

But what happens if we integrate a 2-, 3-, 4-, 10-dimensional function?

Then we need N^{10} integration points, where N is at least 100... That is somehow a lot!

Let us compute. For a 10-dimensional integral with $N = 100$ points along each axis:

$$100^{10} = 10^{20} \text{ points}$$

Even if each point is computed in 1 nanosecond, this will take:

$$10^{20} \text{ ns} = 10^{11} \text{ s} \approx 3000 \text{ years}$$

This is the **curse of dimensionality**.

15.4.1 Where does this occur in chemistry?

- **Quantum chemistry:** Computation of many-electron integrals (electron-electron repulsion) — these are 6-dimensional integrals (3 coordinates per electron).
- **Statistical mechanics:** The partition function is an integral over the phase space of all particles. For N particles — this is a $6N$-dimensional integral (3 coordinates + 3 momenta per particle).
- **Molecular dynamics:** Averaging over configurations — multidimensional integration.
- **Machine learning:** Bayesian inference — integration over the space of model parameters.

15.5 The Monte Carlo method for integration

And here the Monte Carlo method enters the stage — one of the most elegant and surprising algorithms in computational mathematics.

15.5.1 The idea in a nutshell

Imagine that we want to compute the integral:

$$I = \int_{[a,b]^d} f(\mathbf{x})\, d\mathbf{x}$$

where d is the dimension (can be very large).

The Monte Carlo method says: "Let us simply throw N random points $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N$ uniformly into the integration domain, evaluate the function at them and average".

The formula:

$$I \approx \frac{V}{N} \sum_{k=1}^{N} f(\mathbf{x}_k)$$

where $V = (b-a)^d$ is the volume of the integration domain, and $\mathbf{x}_k$ are random points uniformly distributed in $[a,b]^d$.

15.5.2 Why does this work?

This is simply the law of large numbers. The mathematical expectation of the random variable $f(\mathbf{X})$, where $\mathbf{X}$ is uniformly distributed in $[a,b]^d$, equals:

$$\mathbb{E}[f(\mathbf{X})] = \frac{1}{V} \int_{[a,b]^d} f(\mathbf{x})\, d\mathbf{x} = \frac{I}{V}$$

By the law of large numbers, the mean value $\frac{1}{N}\sum f(\mathbf{x}_k)$ converges to $\mathbb{E}[f(\mathbf{X})]$ as $N \to \infty$.

15.5.3 Rate of convergence

And here — magic! The error of the Monte Carlo method decreases as:

$$\text{Error} \sim \frac{\sigma}{\sqrt{N}}$$

where σ is the standard deviation of the function $f(\mathbf{x})$ in the integration domain.

Important: this rate of convergence *does not depend on the dimension* d!

For comparison:

- Simpson's method in 1D: error $\sim h^4 \sim N^{-4}$,

- Simpson's method in the d-dimensional case: error $\sim N^{-4/d}$ (catastrophically slow for large d),
- Monte Carlo method in any dimension: error $\sim N^{-1/2}$.

Yes, Monte Carlo converges more slowly than Simpson in 1D. But in the 10-dimensional case:

- Simpson: $N^{-4/10} = N^{-0.4}$,
- Monte Carlo: $N^{-0.5}$.

Monte Carlo is *faster*!

The price of Monte Carlo

The Monte Carlo method converges as $N^{-1/2}$ — this means that to increase accuracy 10 times, 100 times more points are needed. This is slow by absolute standards, but it is the *only* method that works in high dimensions.

15.6 An example from quantum chemistry: computation of orbitals

Let us see how the Monte Carlo method is applied in real quantum chemistry.

15.6.1 Problem: normalisation of the wave function

In quantum mechanics the wave function $\Psi(\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_N)$ describes the state of N electrons. It must be normalised:

$$\int |\Psi(\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_N)|^2 \, d\mathbf{r}_1 \, d\mathbf{r}_2 \cdots d\mathbf{r}_N = 1$$

This is a $3N$-dimensional integral! For a water molecule ($N = 10$ electrons) — this is a 30-dimensional integral.

15.6.2 Application of Monte Carlo

We generate M random configurations of electrons:

$$\{\mathbf{r}_1^{(k)}, \mathbf{r}_2^{(k)}, \ldots, \mathbf{r}_{10}^{(k)}\}, \quad k = 1, \ldots, M$$

where each coordinate $\mathbf{r}_i^{(k)} = (x_i^{(k)}, y_i^{(k)}, z_i^{(k)})$ is chosen from some distribution (for example, Gaussian).

We compute $|\Psi|^2$ in each configuration and average:

$$\int |\Psi|^2 \approx \frac{V^{30}}{M} \sum_{k=1}^{M} |\Psi(\mathbf{r}_1^{(k)}, \ldots, \mathbf{r}_{10}^{(k)})|^2$$

15.6.3 Variational Monte Carlo (VMC)

But that is not all! In quantum chemistry there is the **Variational Monte Carlo (VMC)** method, which uses Monte Carlo for the *minimisation* of energy.

We choose a trial wave function $\Psi_T(\mathbf{r}_1, \ldots, \mathbf{r}_N; \alpha)$ with parameters α and compute the energy:

$$E(\alpha) = \frac{\int \Psi_T^* \hat{H} \Psi_T \, d\mathbf{r}_1 \cdots d\mathbf{r}_N}{\int |\Psi_T|^2 \, d\mathbf{r}_1 \cdots d\mathbf{r}_N}$$

Both integrals are $3N$-dimensional, and we compute them by Monte Carlo. Then we minimise $E(\alpha)$ over the parameters α — and obtain an approximate wave function.

15.6.4 Quantum Monte Carlo (QMC)

There are even more advanced methods — **Diffusion Monte Carlo (DMC)**, which solve the Schrödinger equation directly, modelling the "diffusion" of electrons in imaginary time. These methods give *almost exact* solutions for small molecules and are used as a benchmark for testing other methods.

Where is Monte Carlo used in chemistry?

- **Quantum Monte Carlo (QMC):** Exact solution of the Schrödinger equation for small systems,
- **Monte Carlo molecular dynamics:** Sampling of configurations in statistical mechanics,
- **Integration in DFT:** Numerical integration of the exchange-correlation functional,
- **Bayesian optimisation:** Search for the global minimum of the energy of a molecule.

15.7 Improvements of the Monte Carlo method

The basic Monte Carlo method is simply a uniform random walk. But there are many improvements:

15.7.1 Importance sampling

Instead of a uniform distribution we use a distribution proportional to $|f(\mathbf{x})|$. Then the points will more often fall into regions where the function is large, and the error will decrease.

15.7.2 The Metropolis method (Metropolis-Hastings)

For complicated multidimensional integrals we use Markov chains: each next point depends on the previous one. This allows efficient exploration of the space, even if it is very large.

15.7.3 Quasi-Monte Carlo

Instead of random points we use **deterministic** low-discrepancy sequences (Sobol, Halton sequences). They cover the space "more uniformly" than random points and converge faster: error $\sim N^{-1}(\log N)^d$.

Summary

The Monte Carlo method is the only practical way to compute multidimensional integrals. Its rate of convergence does not depend on the dimension, which makes it indispensable in quantum chemistry, statistical physics and machine learning. Although it converges more slowly than deterministic methods in low dimensions, in high dimensions it simply has no competition.

16 Statistics: how to separate information from noise

Until now we have mostly considered mathematical problems in which the numbers are assumed to be known. But experimental chemistry is arranged differently. If we measure the concentration of a substance, the position of a spectral line or the intensity of a signal, the instrument never reports the "true" value with infinite precision. Every measurement differs slightly from the others. Therefore, instead of a single number, we have to deal with a **random variable**.

Suppose we measure the same quantity several times. We obtain: $(x_1, x_2, \ldots, x_N) = \mathbf{x}$, where $\mathbf{x} = \mu + \varepsilon$, and

- μ is the unknown true or mean value;

- ε is the random measurement error.

This simple equation is one of the fundamental ideas of statistics.
Statistics begins where we understand: *we observe data, but we are interested in hidden quantities.*

16.1 The mean

The simplest estimate of μ is the arithmetic mean:

$$\mu = \frac{1}{N}\sum_{i=1}^{N} x_i.$$

Why exactly this? If the errors have a mean value near zero, then the arithmetic mean minimises the sum of squared deviations: $\min_\mu \sum_{i=1}^{N} |x_i - \mu|^2$, and the solution is given precisely by the formula above. By the way, this already explains why repeating an experiment usually increases the accuracy of the result.

16.2 Variance and standard deviation

The mean tells us where the centre of the data is, but says nothing about how strongly the results are scattered. For this one introduces the variance — a measure of how strongly our measurements are "scattered" around the true value (to which the mean tends):

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N} |x_i - \mu|^2$$

Yes, we have just minimised it, have we not? Correct, but the value of the minimum will be equal to zero if and only if all the values x_i were the same; otherwise — this value is precisely the variance. And in order to measure it in the same units, we simply recall that the second norm contains a root, and write

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N} |x_i - \mu|^2}$$

That is, the variance is, in essence, the **square of the length of the vector of deviations from the mean**. Statistics again turns into linear algebra.

16.3 The normal distribution

In many physical and chemical measurements random errors are approximately described by the normal distribution:

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right).$$

It is not necessary to memorise this formula. It is far more important to understand the meaning of the two parameters: μ and σ. The first determines the position of the centre of the distribution, the second — its width, that is, the larger σ, the stronger the scatter of the measurements.

Intuition

- The mean answers the question: *"Where is the result?"*
- The standard deviation answers the question: *"How strongly are the results scattered?"*

16.4 Why does the mean become more accurate when the experiment is repeated?

Suppose we have made N independent measurements. For independent errors the variance of the mean is $\frac{\sigma^2}{N}$, consequently, **the error of the mean decreases as** $\frac{\sigma}{\sqrt{N}}$. Therefore increasing the number of measurements increases accuracy, but not linearly (as in Monte Carlo!).

> To reduce the random error by a factor of 10, in the idealised case approximately 100 times more independent measurements are needed.

16.5 Random error and systematic error

Here a very important distinction must be made. If the instrument gives $x = \mu + \varepsilon$, where ε fluctuates randomly around zero, repeated measurements can reduce the influence of this error. But let us imagine that the instrument systematically overestimates the result: $x = \mu + b + \varepsilon$, where b is a constant offset. Then averaging gives $\bar{x} \approx \mu + b$. And no matter how many times we repeat the experiment, the offset b will not disappear.

Important

Repeating measurements reduces random error, but does not eliminate systematic error.
Statistics cannot correct a wrongly calibrated instrument.

This is one of the reasons why calibration, control samples and independent measurement methods are so important in chemistry.

16.6 Two quantities can be related

Suppose two quantities are measured simultaneously: x_i and y_i. For example:

- the concentration of a substance and the intensity of a signal;
- temperature and reaction rate;
- pressure and volume;
- two spectral characteristics.

We may be interested in the question: *do x and y change in a coordinated way?*

For this one uses the covariance: $\mathrm{Cov}(x, y) = \frac{1}{N}(x - \mu_x)^T(y - \mu_y)$

If large values of x usually correspond to large values of y, the covariance is positive. If large values of x correspond to small y, it is negative. If there is no linear relation, the covariance may be close to zero.

16.7 Correlation

The covariance depends on the units of measurement. Therefore one often uses the normalised quantity: $\rho_{xy} = \frac{\mathrm{Cov}(x, y)}{\sigma_x \sigma_y}$. For it $-1 \leq \rho_{xy} \leq 1$. A value near 1 means a strong positive linear relation, a value near -1 — a strong negative one, and a value near 0 means the absence of a pronounced linear correlation.

But here one very important thing must be remembered:

Correlation does not mean causation

If two quantities are correlated, this does not yet mean that one causes the other.
Correlation speaks of a statistical relation between the data. To establish a causal relation, additional physical, chemical or experimental arguments are needed.

16.8 The covariance matrix

If we have not two but p measured quantities, $x_1, x_2, \ldots, x_p$, then the covariances of all pairs can be collected into a single matrix:

$$\Sigma = \begin{pmatrix} \mathrm{Cov}(x_1, x_1) & \mathrm{Cov}(x_1, x_2) & \cdots \\ \mathrm{Cov}(x_2, x_1) & \mathrm{Cov}(x_2, x_2) & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix}.$$

This matrix is symmetric: $\Sigma = \Sigma^T$, and it does not merely store statistical information. It is a mathematical object of linear algebra. That is precisely why eigenvalues and eigenvectors, which we have already encountered before, again arise in statistics.

16.9 PCA: statistics meets linear algebra

Let us imagine that we have N chemical samples, for each of which p characteristics have been measured. We obtain a data matrix: $X \in \mathbb{R}^{N \times p}$. Some characteristics may be strongly related to each other. For example, if two measured quantities almost always change together, the information about them is partially redundant. It would be desirable to find new coordinates in which:

- the first coordinate contains the maximum possible variation of the data;
- the second — the maximum possible remaining variation;
- the third — the next, and so on.

This is the main idea of **Principal Component Analysis** (PCA). Mathematically PCA is closely related to the eigenvectors of the covariance matrix: $\Sigma v_i = \lambda_i v_i$. The eigenvectors v_i define new directions, and the eigenvalues λ_i show how much of the variation of the data falls on the corresponding direction.

If the first few eigenvalues are significantly larger than the rest,

$$\lambda_1, \lambda_2, \ldots, \lambda_k \gg \lambda_{k+1}, \ldots, \lambda_p,$$

then most of the information can be represented by just a few coordinates.

For example,

$$5000 \text{ measured parameters} \quad \longrightarrow \quad 20 \text{ principal components.}$$

This does not mean that we have "thrown away" chemical information. We have found a more compact mathematical representation of the data.

And here again the SVD appears, which we have already encountered:

$$X = U \Sigma V^T.$$

PCA and SVD turn out to be two sides of the same linear-algebraic idea.

16.10 Regression: when we want to build a model

Suppose we have measured the concentration c_i and the corresponding signal y_i. We assume a linear model:

$$y_i = ac_i + b + \varepsilon_i.$$

Since the measurements contain errors, the points usually do not lie exactly on a single straight line. Therefore we look for a and b that minimise the sum of squared errors:

$$\min_{a,b} \sum_{i=1}^{N} (y_i - ac_i - b)^2 .$$

This is the **least squares method**. Thus regression is not something completely new. We already know this mathematical construction:

$$\boxed{\text{data} \to \text{model} \to \text{residual} \to \text{minimisation}}$$

That is precisely why linear algebra and optimisation are so important for statistics.

16.11 Overfitting

Now a more complicated problem arises. If there is little data, we can choose a very complicated function that will pass practically through every experimental point. For example, instead of a straight line one can use a polynomial of high degree:

$$y = a_0 + a_1 x + a_2 x^2 + \cdots + a_k x^k.$$

For a sufficiently large k one can describe the available measurements almost perfectly. But this does not necessarily mean that we have found the correct physical law. We may simply have fitted the noise. This is called **overfitting**.

That is precisely why in statistics and machine learning we are interested not only in the question: *"How well does the model describe the known data?"* but also: *"How well does it work on new data?"*

16.12 Training, validation and testing

If we build a model from the available data, it is useful to split the data into parts:

$$\boxed{\text{training} \quad + \quad \text{validation} \quad + \quad \text{test}}$$

On the training set the model is trained. The validation set is used to choose the parameters of the model. The test set must remain independent and is used for the final evaluation of the model's ability to work on new data. This idea will be especially important when we reach machine learning.

16.13 What statistics is really trying to do

One can say that statistics deals not so much with "counting averages" as with a more general task: **to extract reliable information from imperfect data.** We observe:

$$\text{data} = \text{structure} + \text{noise}.$$

Our task is to reconstruct from the data the structure that interests us and at the same time to estimate how much we can trust it. In the simplest case this looks like this:

$$x_1, x_2, \ldots, x_N \longrightarrow \bar{x} \pm \text{uncertainty}.$$

In a more complicated case:

$$X \longrightarrow \text{PCA} \longrightarrow \text{low-dimensional representation.}$$

And even more complicated:

$$X \longrightarrow \text{model} \longrightarrow \text{prediction} \longrightarrow \text{validation on new data.}$$

It is precisely from here that modern machine learning naturally grows.

What is important to remember

1. A real measurement contains random and, possibly, systematic error.
2. The mean describes the central value of the data.
3. The standard deviation describes their scatter.
4. The random error of the mean decreases as $1/\sqrt{N}$.
5. Systematic error is not eliminated by averaging.
6. Covariance describes the joint variation of quantities.
7. PCA looks for the most informative directions in multidimensional data.
8. Regression builds a mathematical model from measurements.
9. A good model must work not only on known data but also on new data.

16.14 And again linear algebra

And, perhaps, the most pleasant observation for us is that statistics has turned out to be not such a foreign subject after all.

We started with measurements: $x_1, x_2, \ldots, x_N$,
moved to vectors: $\mathbf{x}$,
then to norms: $\|\mathbf{x}\|_2$,
to covariance matrices: Σ,
to eigenvalues: $\Sigma v = \lambda v$,
to SVD: $X = U\Sigma V^T$,
and finally to optimisation: $\min \|Ax - b\|_2^2$.

That is, statistics does not destroy our mathematical picture. It shows how the very same mathematics begins to work with **real, imperfect and noisy data**. And this is precisely the situation that a modern chemist encounters almost every day.

17 From SVD to compressed sensing: when there is less data than needed

17.1 A practical problem: separation of spectra

Let us consider a practical problem. A mass of colourful organic matter was brought to our laboratory, and the only thing at hand was a VIS/IR spectrometer for fluorescence spectra. We were told that the samples contain several reaction products with presumably differing spectra, and we were asked to determine both the concentrations and the spectra of the pure substances.

We chose a not very strong UV source so that the fluorescence was very bright and gave different spectra for different substances.

This seems to be a problem for SVD! After all, if we put each such spectrum in its own column of the matrix **A**, assume that we hypothetically have a matrix of pure spectra of these substances **P**, and for each sample there are coefficients of concentration of the substances themselves **Q**, then:

$$\mathbf{A} = \mathbf{PQ}$$

and we can obtain **P** as the left singular vectors of the matrix **A**, and **Q** as the right singular vectors multiplied on the left by the diagonal.

We can always also write the singular value decomposition as:

$$\forall i, j: \quad a_{ij} = \sum_{r=1}^{R} d_r u_{ir} v_{rj}$$

Moreover, if we had a single substance ($R = 1$), this is exactly how it would be.

17.2 But something went wrong...

Instead of beautiful pure spectra with positive values, we obtained some strange graphs in **P** — with negative values, oscillations, "ghost" peaks.

In fact — this is exactly how it should have been, the spectra *cannot* be obtained this way. The spectra themselves are positively defined graphs, so the scalar product of any pair, although it can be very close to zero, is not necessarily so. In the singular value decomposition we require that all columns of **U** be **orthogonal** to each other. And real spectra of substances are not orthogonal! They are simply "similar" or "not similar".

We need something else to pull these spectra apart so that our mathematics sees them separately.

17.3 The magic of the third dimension: Kruskal's theorem

If we set up the experiment so that we obtain 3D data instead of 2D as in the example above, then it can be proved mathematically that a **non-orthogonal** decomposition also exists.

For example, if we can record fluorescence spectra at several different excitation wavelengths. Then we will already have three-dimensional data:

$$\forall i, j, k: \quad a_{ijk} = \sum_{r=1}^{R} \alpha_r \, b_{ir} \, c_{jr} \, d_{kr}$$

and, by **Kruskal's theorem** (Kruskal, 1977), we have no requirements on the orthogonality of each of these matrices!

17.3.1 What Kruskal's theorem says

Kruskal's theorem is one of the cornerstones of multidimensional data analysis. It gives conditions for the **uniqueness** of a tensor decomposition (the so-called CANDECOMP/PARAFAC decomposition).

For a third-order tensor $\mathcal{A} = \sum_{r=1}^{R} \mathbf{a}_r \circ \mathbf{b}_r \circ \mathbf{c}_r$ (where $\circ$ is the outer product of vectors), the decomposition is **unique** up to permutation and scaling of components if:

$$k_A + k_B + k_C \geq 2R + 2$$

where k_A is the **k-rank** (Kruskal rank) of the matrix **A**, that is, the maximum number k such that any subset of k columns of the matrix **A** is linearly independent.

Why is this so important?

In ordinary SVD we have *infinitely many* decompositions $\mathbf{A} = \mathbf{U}\square\mathbf{V}^H$ — any orthogonal transformation inside gives a new valid decomposition. Therefore SVD cannot isolate the "physical" components — only mathematically convenient (orthogonal) ones.
Kruskal's theorem says: for tensors of the 3rd and higher order, when the condition on the k-ranks is satisfied, the decomposition is *unique*! This means that we can recover precisely those physical components that were in the data — without requiring orthogonality.

17.3.2 Solution methods: PARAFAC and ALS

In practice the tensor decomposition is sought by iterative methods. The most popular is **ALS** (Alternating Least Squares):

1. We fix **B** and **C** and solve the least squares problem for **A**,
2. We fix **A** and **C** and solve for **B**,
3. We fix **A** and **B** and solve for **C**,
4. We repeat until convergence.

This is a classical nonlinear minimisation problem (we discussed it in the chapter on nonlinear operators), and it may get stuck in local minima. Therefore in practice ALS is run many times with different initial approximations.

Applicability in chemistry

PARAFAC/CANDECOMP is widely used in:

- Fluorescence spectroscopy (EEM — Excitation-Emission Matrix),
- Chromatography-mass spectrometry,
- NMR spectroscopy,
- Analysis of metabolic data (metabolomics).

17.4 Multidimensional NMR spectra

And now — the most interesting for structural chemists. In NMR spectroscopy of proteins one can construct very multidimensional spectra: 2D, 3D, 4D and even 5D.

Each dimension is its own frequency (the chemical shift of a certain nucleus: ^{1}H, ^{13}C, ^{15}N). Cross-peaks in such spectra show which nuclei are close to each other in space, which allows reconstructing the 3D structure of the protein.

But here is the problem: if we want to record a 4D spectrum with resolution $1024 \times 256 \times 256 \times 64$ points, then the total number of points is $\sim 4 \times 10^9$. And each measurement takes time (to wait for relaxation), and in total this is **weeks** of spectrometer operation!

There was a time when, in order to determine all cross-peaks of ubiquitin (a peptide of 76 amino acids), one had to record a spectrum for about **three weeks**. The protein could degrade during that time!

17.5 Sparse sampling: less means more

But one can record only randomly sparse one-dimensional spectra — and even not 10%, and not 1%, but really very, very few, and apply the same decomposition, only for sparse data, and obtain equally reliable results!

Applying such methods 20 years ago, the author with his colleagues accelerated the recording of such a multidimensional spectrum without loss of information from three weeks to **15 minutes**.

The idea is simple: if we know that the spectrum is *sparse* in some basis (that is, consists of a small number of peaks), then we do not need to measure all points. It is enough to measure a random subset, and then *recover* the missing points through optimisation.

17.6 Compressed Sensing: a revolution in measurements

This brings us to one of the most beautiful ideas in modern mathematics and signal processing — **Compressed Sensing** (compressive sampling).

17.6.1 Classical theory: Nyquist–Shannon

The traditional theory of discretisation (Nyquist–Shannon) says: to reconstruct a signal with maximum frequency $f_{\max}$, one must measure it at a frequency of at least $2f_{\max}$.

For a 4D NMR spectrum with resolution $1024 \times 256 \times 256 \times 64$ this means that we are *obliged* to measure all 4×10^9 points. Fewer — impossible, otherwise there will be aliasing (frequency folding).

17.6.2 The revolution: compressed sensing

But in 2004–2006 Donoho, Candès, Tao and other mathematicians showed: if a signal is **sparse** in some basis, then it can be reconstructed from a **substantially smaller** number of measurements!

Formally: let the signal $\mathbf{x} \in \mathbb{R}^N$ have a sparse representation $\mathbf{x} = \square\mathbf{s}$, where $\mathbf{s}$ has only $K \ll N$ nonzero elements. Then we can measure $\mathbf{y} = \square\mathbf{x}$, where $\square \in \mathbb{R}^{M \times N}$ is the measurement matrix with $M \sim K \log(N/K) \ll N$, and reconstruct $\mathbf{x}$ by solving the problem:

$$\min_{\mathbf{s}} \|\mathbf{s}\|_1 \quad \text{subject to} \quad \mathbf{y} = \square\square\mathbf{s}$$

Why the L1 norm and not L0?
The idea is that we want to find the *sparsest* solution (minimise $\|\mathbf{s}\|_0$ — the number of nonzero elements). But minimising the L0 norm is an NP-hard problem (enumeration of all combinations). The magic of compressed sensing is that under certain conditions on the matrix $\square$ (the so-called Restricted Isometry Property, RIP), minimising the L1 norm gives *exactly the same result* as minimising L0! And L1 minimisation is a convex problem, which is solved efficiently (it is linear programming).

17.6.3 Conditions of applicability

For successful reconstruction two conditions are needed:

1. **Sparsity:** The signal must be sparse in some basis (for example, an NMR spectrum is a set of delta functions, that is, very sparse in the frequency domain).

2. **Incoherence:** The measurement matrix $\square$ must be "incoherent" with the basis $\square$. In practice this means that the measurement points must be chosen **randomly** (or pseudorandomly).

17.7 Examples of Compressed Sensing in chemistry and beyond

17.7.1 Fast MRI (Magnetic Resonance Imaging)

One of the best-known applications is the acceleration of MRI in medicine. Classical MRI requires lengthy scanning (k-space is filled line by line). With compressed sensing one can measure only **random** lines of k-space and reconstruct the full image via L1 minimisation. This reduces the scanning time by 5–10 times — critical for patients who cannot lie still for long.

17.7.2 Sparse NMR spectroscopy

In multidimensional NMR of proteins compressed sensing allows:

- Measuring only a random subset of points in the multidimensional k-space,
- Reconstructing the full spectrum via L1 minimisation,
- Reducing the experiment time from weeks to hours or minutes.

This is precisely what the author of the script did 20 years ago, and what has now become the standard in modern NMR spectrometers (sparse sampling, Poisson-gap sampling, etc.).

17.7.3 Single-Pixel Camera (Rice camera)

An amazing example: a camera that has **no pixel matrix**, but only a single detector. The image is reconstructed via compressed measurements using a DMD (Digital Micromirror Device). This works because real images are sparse in the wavelet basis.

17.7.4 Mass spectrometry

In mass spectrometry compressed sensing is used to accelerate FT-ICR (Fourier Transform Ion Cyclotron Resonance) and Orbitrap — one can measure the FID for a shorter time and still obtain high resolution.

17.7.5 Data compression

JPEG uses the discrete cosine transform (a close relative of Fourier), and images are sparse in this basis — that is why JPEG compresses so well. Compressed sensing is the next step: we do not just compress already measured data, but immediately measure less.

17.8 Connection with previous chapters

Let us see how compressed sensing connects everything we have covered:

- **Linear algebra:** We solve an overdetermined system □□**s** = **y** via L1 norm minimisation.
- **Conditioning:** The matrix □ must be well conditioned and satisfy the Restricted Isometry Property (RIP).
- **SVD and tensors:** For multidimensional data we use tensor decompositions (PARAFAC), which give uniqueness without orthogonality.
- **Nonlinear optimisation:** L1 minimisation is a convex problem, but not smooth (there is no derivative at zero). We use methods like ISTA (Iterative Shrinkage-Thresholding Algorithm) or ADMM.
- **FFT:** The operator □ is often the Fourier transform, and we use FFT for fast multiplication.

Main conclusion

Compressed sensing is not just another algorithm. It is a *philosophy of measurement*: if we know that the signal is simple (sparse), then we can measure it substantially less than classical theory requires. It has revolutionised MRI, NMR spectroscopy, mass spectrometry and many other fields. And all this is based on beautiful mathematics: convex optimisation, probability theory (random matrices) and linear algebra.

18 Information compression: from archivers to JPEG

18.1 Two worlds of compression

Information compression is one of the most practical areas of applied mathematics. We constantly compress and decompress data: archives, pictures, video, music. But not all compression is the same.

There are two fundamentally different approaches:

- **Lossless compression:** We can reconstruct the original data *bit for bit*. Examples: ZIP, PNG, FLAC.
- **Lossy compression:** We sacrifice part of the information for the sake of greater compression. Examples: JPEG, MP3, MP4.

18.2 Shannon's information entropy

Before speaking of methods, let us understand *how much* data can be compressed at all.

Claude Shannon in 1948 introduced the concept of **information entropy**. If we have an alphabet of n symbols, and symbol i occurs with probability p_i, then the entropy (the average amount of information per symbol) is:

$$H = -\sum_{i=1}^{n} p_i \log_2 p_i \quad \text{[bits]}$$

This is the **theoretical limit** of lossless compression. We cannot compress data more strongly than H bits per symbol.

Example

In English text the letter "e" occurs most often ($p \approx 0.13$), and "z" — very rarely ($p \approx 0.001$). If we encoded all letters equally (8 bits per symbol), we would spend too many bits on rare letters. The entropy of English text is about 4.2 bits per symbol, so theoretically we can compress it about 2 times.

18.3 Lossless compression: Huffman coding

The idea is simple: we notice that some symbols and sequences of 2–3 symbols occur more often, we set up a table of such symbols, and the more often a symbol or sequence occurs, the fewer bits we use to encode it.

18.3.1 The Huffman algorithm

1. We count the frequencies of all symbols in the text.
2. We build a binary tree: we merge the two rarest symbols into a node with the total frequency, and repeat until we obtain one tree.

3. The code of a symbol is the path from the root to the leaf (0 — left, 1 — right).

Frequent symbols end up closer to the root — their codes are shorter. Rare ones — farther, codes longer.

Example

Let us have the symbols A (frequency 0.5), B (0.25), C (0.125), D (0.125). The Huffman tree will give the codes:

- A: 0 (1 bit)
- B: 10 (2 bits)
- C: 110 (3 bits)
- D: 111 (3 bits)

Average length: $0.5 \times 1 + 0.25 \times 2 + 0.125 \times 3 + 0.125 \times 3 = 1.75$ bits per symbol — close to the entropy!

18.3.2 Dictionary methods: LZ77, LZ78, LZW

Algorithms of the LZ family (Lempel–Ziv) go further: they look not only for frequent symbols but also for **repeating sequences**.

Idea: if we have already seen the string "abracadabra", then when it appears again we do not write it anew, but refer to the previous occurrence: "(back 11, length 11)".

This is the basis of the ZIP, GZIP, PNG formats.

18.4 A chemical example: searching protein sequences

And now — a live example from biology. We have a database of protein sequences (millions of amino acids), and we need to find whether it contains homologues (similar proteins) to our new protein.

Direct comparison "our protein vs every protein in the database" is too slow. We need a smart algorithm for compression and search.

18.4.1 BLAST (Basic Local Alignment Search Tool)

BLAST is one of the most cited algorithms in biology. The idea:

1. We split our protein into short "words" (k-mers, usually $k = 3$ for proteins).
2. For each word we look for similar words in the database (with small mutations).
3. We extend the matches in both directions until the similarity falls below a threshold.

This is, in essence, a dictionary compression method + fast search by a hash table. BLAST allows finding homologues in seconds, although full alignment would take hours.

Why is this important?

BLAST and its variants (PSI-BLAST, BLASTP, BLASTN) are the basis of modern bioinformatics. Without them there would be no genome decoding, no drug development, no evolutionary studies.

18.5 Lossy compression: JPEG and low-rank approximation

Now — lossy compression. The idea: we sacrifice part of the information that the human eye (or ear) will not notice anyway.

18.5.1 JPEG via the discrete cosine transform (DCT)

JPEG works like this:

1. We split the picture into 8×8 pixel blocks.
2. For each block we apply the **discrete cosine transform** (DCT) — a close relative of Fourier.
3. DCT decomposes the block into 64 frequency components (from low frequencies — the general background, to high frequencies — fine details).
4. We quantise the coefficients: divide by a quantisation matrix and round. High frequencies (fine details) are quantised more coarsely — we "lose" them.
5. The remaining nonzero coefficients are compressed losslessly (Huffman).

18.5.2 Low-rank approximation via SVD

An alternative view of image compression is via SVD. Let us have the image matrix $\mathbf{A} \in \mathbb{R}^{M \times N}$. SVD:

$$\mathbf{A} = \mathbf{U} \square \mathbf{V}^H = \sum_{k=1}^{\min(M,N)} \sigma_k \mathbf{u}_k \mathbf{v}_k^H$$

We can approximate $\mathbf{A}$ by the first r singular values:

$$\mathbf{A}_r = \sum_{k=1}^{r} \sigma_k \mathbf{u}_k \mathbf{v}_k^H$$

This is the **low-rank approximation** of rank r. By the Eckart–Young theorem, this is the *best* approximation of rank r in the sense of the L2 norm.

Compression ratio

The original matrix: $M \times N$ numbers. Approximation of rank r: $r(M + N + 1)$ numbers. If $r \ll \min(M, N)$, the compression is enormous!
For example, for a 1000×1000 picture and $r = 50$: compression by $1000 \times 1000/(50 \times 2001) \approx 10$ times.

18.6 Wavelets: local frequencies

Both DCT and SVD are global transformations: they decompose the whole picture into frequencies. But pictures have *local* features: edges of objects, textures, points.

Wavelets are basis functions that are localised both in space and in frequency. They allow analysing a signal at different scales.

18.6.1 The wavelet transform

Instead of sinusoids (as in Fourier) we use "little waves" (wavelets) — functions that decay quickly. The most popular is the **Haar wavelet**:

$$\psi(t) = \begin{cases} 1, & 0 \le t < 0.5 \\ -1, & 0.5 \le t < 1 \\ 0, & \text{otherwise} \end{cases}$$

The wavelet transform decomposes a signal by scales (frequencies) and positions. This allows:

- Compressing pictures better than JPEG (the JPEG2000 format uses wavelets),
- Detecting edges (edges of objects),
- Removing noise while preserving important details.

Summary

Information compression is a balance between mathematical theory (entropy, SVD, wavelets) and psychophysics (what the eye sees, what the ear hears). From archivers to JPEG — everywhere one idea: find structure in the data and use it for a compact representation.

19 Image comparison: from convolution to neural networks

19.1 The problem: find an object in a picture

How do we compare two pictures and understand that they contain the same object?

We have already compared two sequences by constructing a circulant matrix (the chapter on FFT). Probably pictures can be compared in the same way? Let them be 2D. But not everything here is as good as it seems.

If one picture is simply **shifted** relative to the other along one or both axes — yes, everything will work. We can use 2D convolution (via FFT) and find the maximum — this will be the position of the object.

But if there is a **rotation**? Or a **stretch**? Or a **change of illumination**? Simple convolution will no longer help.

19.2 Edge detection

The first step to understanding the content of a picture is the detection of **edges**. Edges are places where the pixel intensity changes sharply.

19.2.1 The Sobel operator

The simplest way is to compute the intensity gradient. The Sobel operator uses two 3×3 masks to compute derivatives with respect to x and y:

$$G_x = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} * I, \quad G_y = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} * I$$

where $*$ is convolution, I is the image.

Gradient modulus: $G = \sqrt{G_x^2 + G_y^2}$. Where G is large — there is an edge.

19.2.2 Canny edge detector

A more advanced algorithm:

1. We smooth the picture with a Gaussian filter (remove noise).
2. We compute the gradient (as with Sobel).
3. We suppress non-maxima (keep only the strongest edges).
4. We apply hysteresis: if an edge is above the upper threshold — keep, below the lower — remove, between — keep if connected to a strong edge.

The result — thin, connected lines of edges.

19.3 Keypoints: key points of an image

Edges are good, but we need something more "point-like" to compare pictures. For this one looks for **keypoints** (interest points) — places that are easy to find and that are stable under transformations.

19.3.1 Harris corner detector

Idea: we look for corners — places where an edge changes direction. For each pixel we compute the matrix of second moments of the gradient:

$$\mathbf{M} = \sum_{x,y} w(x,y) \begin{pmatrix} G_x^2 & G_x G_y \\ G_x G_y & G_y^2 \end{pmatrix}$$

where $w(x,y)$ is a window (for example, Gaussian).

If both eigenvalues of $\mathbf{M}$ are large — this is a corner. If one is large, the other small — this is an edge. If both are small — this is a flat region.

19.3.2 SIFT (Scale-Invariant Feature Transform)

SIFT is one of the most popular algorithms (Lowe, 1999). It finds keypoints that are stable under:

- Scale (the object may be closer or farther),
- Rotation,
- Change of illumination,
- Small changes of viewpoint.

Algorithm:

1. We build a **scale pyramid**: we blur the picture with a Gaussian filter with different σ and reduce the size.
2. We look for **extrema** in the difference of Gaussians (DoG — Difference of Gaussians) — this is an approximation of the Laplacian.
3. For each keypoint we compute a **descriptor**: a histogram of gradients in a 16×16 pixel neighbourhood. This is a vector of 128 numbers.
4. We compare the descriptors between pictures (Euclidean distance).

19.3.3 SURF, ORB, AKAZE

There are many improvements of SIFT:

- **SURF** (Speeded-Up Robust Features) — faster, uses integral images.
- **ORB** (Oriented FAST and Rotated BRIEF) — very fast, patent-free.
- **AKAZE** — uses nonlinear diffusion scales.

19.4 Finding the transformation: RANSAC

We have found keypoints in two pictures and matched them (feature matching). But not all matches are correct — there are outliers.

How do we find the transformation (rotation, scale, shift) that maps one picture into the other, if we have outliers?

19.4.1 RANSAC (Random Sample Consensus)

RANSAC is an elegant algorithm for robust parameter estimation:

1. We randomly choose the minimum number of points (for an affine transformation — 3 pairs of points).
2. From these points we compute the parameters of the transformation.
3. We count how many *of all* points agree with this transformation (inliers) — distance less than a threshold.
4. We repeat N times, choose the transformation with the maximum number of inliers.
5. Finally we refine the parameters over all inliers (least squares method).

Why does RANSAC work?

If the fraction of outliers is not too large (say, $< 50\%$), then the probability of randomly choosing only inliers is nonzero. Repeating enough times, we will almost certainly find the "right" transformation.

19.5 Particle swarm methods and optimisation

Sometimes we have no explicit keypoints, but we know that one picture is a transformed version of the other. How do we find the parameters of the transformation?

This is an optimisation problem: we maximise a **similarity metric** (for example, mutual information) over the parameters of the transformation.

19.5.1 Particle Swarm Optimization (PSO)

The particle swarm method is a heuristic optimisation method inspired by the behaviour of a flock of birds or a school of fish:

1. We initialise a "swarm" of N particles — each particle is a set of transformation parameters (for example, rotation angle, scale, shifts along x and y).
2. Each particle has a "velocity" and a "position".

3. At each step the particle "remembers" its best position (personal best) and knows the best position of the whole swarm (global best).

4. The velocity of the particle is updated as a weighted sum: inertia (continue moving in the same direction), cognitive component (attraction to personal best) and social component (attraction to global best).

5. We repeat until convergence.

PSO is especially useful when:

- The similarity function is nonsmooth or has many local maxima,
- The parameter space is large (for example, a 3D transformation with 12 parameters),
- We cannot compute the gradient (for example, mutual information is not differentiable).

19.6 Neural networks: from hand-crafted features to learned

All the methods we discussed above (Sobel, Harris, SIFT) are **hand-crafted features**. We ourselves invent what an "edge", a "corner", an "interesting point" is, and how to build a descriptor from them.

But what if we let the computer *itself* learn to find features?

19.6.1 Convolutional neural networks (CNN)

Convolutional Neural Networks are a special type of neural networks created for working with images. The idea:

1. **Convolutional layers:** We apply a set of trainable filters (like Sobel masks, but the parameters are learned from data). Each filter extracts its own feature — from simple edges in the first layers to complex textures and parts of objects in the deep layers.

2. **Pooling layers:** We reduce the size of the feature map (max pooling — take the maximum in a 2×2 window). This gives invariance to small shifts.

3. **Fully connected layers:** At the end — an ordinary neural network that classifies or regresses.

19.6.2 Hierarchy of features

What is surprising is that CNNs themselves learn the same hierarchy that we constructed by hand:

- **First layers:** Simple edges, gradients (like Sobel),
- **Middle layers:** Textures, corners, simple shapes (like SIFT descriptors),
- **Deep layers:** Parts of objects (eyes, wheels, leaves),
- **Last layers:** Whole objects (face, car, tree).

This is **feature learning** instead of hand-crafting.

19.7 Pretrained models and transfer learning

Training a CNN from scratch requires millions of labelled images and days of computation on a GPU. But there is a trick — **transfer learning**.

Idea: we take a network pretrained on a huge database (for example, ImageNet — 1.4 million images, 1000 classes) and use it as a **feature extractor**.

19.7.1 Popular architectures

- **VGG (2014):** Simple, deep (16–19 layers), good features.
- **ResNet (2015):** Very deep (up to 152 layers) with "skip connections" — solves the problem of vanishing gradients.
- **EfficientNet (2019):** Optimised in all parameters (accuracy, speed, size).
- **Vision Transformer (ViT, 2020):** Uses the transformer architecture (as in NLP) for images.

19.7.2 How to use a pretrained model

1. **Feature extraction:** We take a pretrained network, cut off the last classification layer, and use the penultimate layer as a feature extractor. We obtain a vector of, say, 2048 numbers for each image. We compare the vectors (cosine distance).
2. **Fine-tuning:** We take a pretrained network and fine-tune it on our data (for example, on microscopic images of cells). The first layers are "frozen" (they already work well), the last ones are trained.
3. **Siamese networks:** Two identical networks with shared weights, trained so that similar images have close feature vectors, and different ones — distant.

19.8 Applications in chemistry and biology

19.8.1 Cryo-electron microscopy (cryo-EM)

In cryo-EM we obtain thousands of 2D projections of protein molecules in random orientations. The task:

1. Find all molecules on the micrographs (particle picking) — this is an object detection task,
2. Determine the orientation of each projection — this is an image comparison task,
3. Reconstruct the 3D structure — this is an inverse tomography problem.

Modern programs (RELION, cryoSPARC) use CNNs for particle picking and classification. This has allowed achieving a "resolution revolution" — determining protein structures with atomic resolution.

19.8.2 Microscopy and cell analysis

In biological research one needs to:

- Count cells in microscopic images,
- Classify cell types,
- Track cell movement over time,
- Find anomalies (cancer cells).

CNN (especially U-Net for segmentation) is the standard in the field.

19.8.3 Chemometrics and drug discovery

In drug discovery one needs to compare molecular structures. But molecules are not pictures! However, they can be represented as 2D images (molecular graphs drawn on a grid) and CNN can be used to predict activity.

19.9 Connection with previous chapters

Let us see how image comparison connects everything we have covered:

- **Linear algebra:** Convolution is multiplication of a matrix by a vector (or tensor). CNN is a sequence of linear operations with nonlinearities.
- **SVD and low-rank approximation:** CNN can be compressed via SVD (low-rank factorisation of convolutional kernels).
- **FFT:** Convolution via FFT is $\mathcal{O}(N \log N)$ instead of $\mathcal{O}(N^2)$. In CNN this is critical for speed.
- **Nonlinear optimisation:** Training a CNN is the minimisation of a loss function (cross-entropy, MSE) via backpropagation + SGD/Adam (variants of gradient descent).
- **Compressed sensing:** In MRI and cryo-EM we use compressed measurements + CNN for reconstruction.

Main conclusion

Image comparison is from simple to complex:

1. Convolution (for shifts) — via FFT,
2. Keypoints (SIFT) + RANSAC (for rotations and scale),
3. Optimisation (PSO) for complex transformations,
4. Neural networks (CNN) — for semantic similarity.

Each method is a compromise between universality, speed and accuracy. And all of them are based on the mathematics we have covered: linear algebra, optimisation, Fourier analysis, probability theory.

20 Machine learning: from the perceptron to AlphaFold

20.1 What is machine learning?

Machine Learning (ML) is a branch of artificial intelligence where the computer *itself learns* to solve problems based on data, rather than by rigidly prescribed rules.

Instead of writing a program "if temperature > 100, then water boils", we give the computer thousands of examples "temperature — state of water" and ask it to find the pattern.

20.1.1 Three types of learning

- **Supervised learning:** There are labelled data (input → output). The task is to learn to predict the output for new inputs. Examples: image classification, prediction of molecular properties.
- **Unsupervised learning:** Data without labels. The task is to find structure, clusters, hidden patterns. Examples: clustering of spectra, PCA.
- **Reinforcement learning:** An agent interacts with the environment and receives rewards/penalties. The task is to learn a strategy that maximises the reward. Examples: playing chess, controlling a robot.

20.2 From the perceptron to neural networks

20.2.1 The perceptron (1958, Rosenblatt)

The simplest "neural network" is the perceptron. It computes:

$$y = \sigma\left(\sum_{i=1}^{n} w_i x_i + b\right)$$

where x_i are the inputs, w_i are the weights, b is the bias, σ is the activation function (for example, a step or a sigmoid).

The perceptron can solve only **linearly separable** problems (for example, logical OR). For XOR (exclusive OR) one perceptron is not enough — a multilayer network is needed.

20.2.2 Multilayer perceptron (MLP)

A network of several layers: an input layer, one or more hidden layers, an output layer. Each neuron is connected to all neurons of the next layer.

Formula for the l-th layer:

$$\mathbf{h}^{(l)} = \sigma\left(\mathbf{W}^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}\right)$$

where $\mathbf{W}^{(l)}$ is the weight matrix, $\mathbf{b}^{(l)}$ is the bias vector, σ is a nonlinear activation function (ReLU, tanh, sigmoid).

20.2.3 Training: backpropagation

How to train a neural network? Minimise the loss function L (for example, MSE for regression or cross-entropy for classification) via gradient descent.

Backpropagation is an efficient algorithm for computing gradients via the chain rule. We go from the output to the input, computing $\partial L/\partial w_{ij}$ for each weight, and update the weights:

$$w_{ij} \leftarrow w_{ij} - \eta \frac{\partial L}{\partial w_{ij}}$$

where η is the learning rate.

20.3 Convolutional neural networks (CNN)

For images MLP is inefficient: too many parameters (each pixel is connected to all neurons).

CNN use **convolutional layers**: a small filter (for example, 3×3) slides over the image and computes a convolution. This gives:

- **Locality of connections:** Each neuron looks only at a small region,
- **Weight sharing:** The same filter is applied to the whole image,
- **Shift invariance:** The object will be found anywhere.

Popular architectures: LeNet (1998), AlexNet (2012), VGG (2014), ResNet (2015), EfficientNet (2019).

20.4 Recurrent neural networks (RNN)

For sequences (text, time series) we need networks with **memory**. RNN pass the hidden state from one step to another:

$$\mathbf{h}_t = \sigma\left(\mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{W}_x \mathbf{x}_t + \mathbf{b}\right)$$

The problem of ordinary RNN is **vanishing gradients**: the network learns poorly on long sequences. Solutions:

- **LSTM** (Long Short-Term Memory, 1997): Adds "gates" that control the flow of information.
- **GRU** (Gated Recurrent Unit, 2014): A simplified version of LSTM.

20.5 The revolution: transformers and attention

In 2017 the paper "Attention Is All You Need" (Vaswani et al.) was published, which turned NLP and much else upside down.

20.5.1 The attention mechanism

Idea: instead of compressing the whole sequence into one vector (as in RNN), we allow each element to "look" at all other elements and weight them by importance.

For an input sequence $\mathbf{x}_1, \ldots, \mathbf{x}_n$ we compute:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V}$$

where **Q** (query), **K** (key), **V** (value) are linear projections of the inputs.

20.5.2 Transformer

The Transformer architecture completely abandons recurrence and convolutions. Instead, only the attention mechanism (self-attention) and fully connected layers are used.

Key components:

- **Multi-Head Attention:** Several parallel attention mechanisms, each of which learns to focus on different aspects of the data.
- **Positional Encoding:** Since the Transformer has no recurrence, it does not know the order of elements. We add positional encodings (sines/cosines or trainable vectors).
- **Layer Normalization:** Normalisation of activations for training stability.
- **Residual Connections:** "Skip connections" to combat vanishing gradients (as in ResNet).

Transformer became the basis for:

- **BERT** (2018): Bidirectional Encoder Representations from Transformers — pretraining on masked words.
- **GPT** (2018–2023): Generative Pre-trained Transformer — autoregressive text generation. GPT-3 (175 billion parameters), GPT-4 (multimodal).
- **T5, BART**: Encoder-decoder architectures for translation, summarisation.

20.6 Machine learning in chemistry: evolution

20.6.1 The QSAR era (1990s – 2010s)

QSAR (Quantitative Structure-Activity Relationship) is a classical approach: we compute **molecular descriptors** (numerical characteristics of a molecule) and build a regression/classification.

Descriptors:

- Physicochemical: molar mass, logP (lipophilicity), polar surface area,
- Topological: connectivity indices, shapes of the molecular graph,
- Electronic: atomic charges, orbital energies,
- 3D descriptors: moments of inertia, radii of gyration.

Methods: PLS (Partial Least Squares), Random Forest, SVM (Support Vector Machines).

Example

Predicting the toxicity of a molecule: we compute 200 descriptors, collect a database of 10 000 molecules with known toxicity, train a Random Forest. Accuracy — 80–85%.

20.6.2 Graph neural networks (2015 – present)

A molecule is a **graph**: atoms are nodes, bonds are edges. Graph Neural Networks (GNN) work directly with graphs.

Message Passing Neural Networks (MPNN):

1. Each atom has an initial representation (a feature vector: atom type, charge, hybridisation).
2. At each step atoms "exchange messages" with neighbours through bonds.
3. After K steps each atom "knows" about its neighbourhood of radius K bonds.
4. The final representation of the molecule is an aggregation of all atomic representations.

Popular architectures:

- **GCN** (Graph Convolutional Network): An analogue of convolution for graphs.
- **GAT** (Graph Attention Network): Attention between atoms.
- **SchNet, DimeNet, SphereNet**: Take into account 3D coordinates of atoms and angles.

Why are GNN so good for chemistry?

- **Permutation invariance:** The order of atoms does not matter,
- **Structure awareness:** GNN see bonds and geometry,
- **Interpretability:** One can look at which atoms/bonds are important for the prediction.

20.6.3 Prediction of molecular properties

Modern must-have models for a chemist:

Task	Model	Accuracy
Molecular energy	SchNet, DimeNet++	MAE $\sim$1 kcal/mol
Solvation	SolTranX	MAE $\sim$0.5 kcal/mol
pKa	ChemProp, GNN	MAE $\sim$0.3
LogP	MolCLR, GNN	MAE $\sim$0.2
Toxicity	GraphCL, GNN	AUC $\sim$0.9
Drug-likeness	MolBERT	AUC $\sim$0.85

20.7 The AlphaFold revolution: protein structure prediction

20.7.1 The problem

Predicting the 3D structure of a protein from its amino acid sequence is one of the grand challenges of biology. Experimental methods (X-ray, cryo-EM, NMR) are expensive and slow.

20.7.2 AlphaFold (2020, DeepMind)

AlphaFold 2 is a breakthrough that solved the problem of protein structure prediction with atomic accuracy.

Architecture:

1. **Evoformer:** A Transformer that processes:
   - Multiple sequence alignment (MSA) — evolutionary information,
   - Pair representation — pairwise distances between residues.
2. **Structure Module:** Iteratively builds 3D coordinates of atoms, minimising the loss function.
3. **Confidence Estimation:** Predicts pLDDT (per-residue confidence) — how confident the model is about each residue.

Results on CASP14 (Critical Assessment of Structure Prediction):

- Average GDT_TS (Global Distance Test) — 92.4 (experimental accuracy — $\sim$90),
- For 2/3 of proteins — accuracy < 1 Å (atomic accuracy).

20.7.3 AlphaFold 3 (2024)

Extension to:

- Protein–ligand complexes,
- DNA/RNA structures,
- Post-translational modifications,
- Ionic interactions.

20.8 A foundation model for conformers

20.8.1 The conformational space problem

A molecule is not a static structure. It constantly "breathes", rotates around bonds, changes conformation. For drug design it is critically important to know not one structure, but an **ensemble** of low-energy conformers.

Classical methods:

- **Molecular Dynamics (MD):** We model the motion of atoms in time, but it is slow (nanoseconds — microseconds).
- **Monte Carlo:** Random walk over conformational space, but inefficient for large molecules.
- **Systematic/Rotamer search:** Enumeration of all combinations of torsion angles, but exponential growth.

20.8.2 ML approach: conformer prediction

Modern models learn to predict the distribution of conformers directly from data.

GeoLDM (Geometric Latent Diffusion Models, 2023):

1. **Encoder:** Encodes a 3D conformation into a latent space.
2. **Diffusion Process:** Adds noise to the latent vectors (as in DALL-E, Stable Diffusion).
3. **Decoder:** Generates new conformations from noise through the reverse diffusion process.
4. **Energy Model:** Filters the generated conformations by energy.

ConfGF (Conformation Graph Factor, 2022):

- Uses GNN to predict 3D coordinates,
- Generates an ensemble of conformers via sampling,
- Takes into account the Boltzmann distribution (low-energy conformers are more probable).

20.8.3 Foundation model: Uni-Mol (2023)

Uni-Mol is a "foundation model" for molecules, pretrained on millions of conformations.

Architecture:

1. **3D Transformer:** Works with atomic coordinates and features.
2. **Pre-training tasks:**
   - Prediction of distances between atoms,
   - Prediction of angles and dihedral angles,
   - Masked atom prediction (as in BERT),
   - Contrastive learning (similar conformers — close).
3. **Fine-tuning:** Fine-tuning on specific tasks (energy, properties, conformers).

Results:

- Energy prediction: MAE $\sim$0.5 kcal/mol (better than DFT for some classes),
- Conformer generation: RMSD < 0.5 Å for 90% of molecules,
- Property prediction: state-of-the-art on most benchmarks.

20.9 Must-have tools for the modern chemist

20.9.1 Software

Tool	Purpose
RDKit	Chemoinformatics: descriptors, fingerprints, similarity
DeepChem	ML for drug discovery, GNN
PyTorch Geometric	Graph neural networks
OpenMM	Molecular dynamics on GPU
ASE (Atomic Simulation Environment)	Quantum chemistry + ML
SchNetPack	SchNet and other GNN for chemistry
AlphaFold (ColabFold)	Protein structure prediction
Uni-Mol	Foundation model for molecules

20.9.2 Typical workflow

1. **Data collection:** PubChem, ChEMBL, PDB (Protein Data Bank).
2. **Preprocessing:** RDKit for descriptors, generation of conformers.
3. **Modelling:** GNN for property prediction, AlphaFold for structure.
4. **Validation:** Cross-validation, external test set, experimental verification.
5. **Interpretation:** Attention weights, SHAP values for understanding predictions.

20.10 Connection with previous chapters

Let us see how ML connects everything we have covered:

- **Linear algebra:** Neural networks are a sequence of matrix multiplications $\mathbf{W}\mathbf{x} + \mathbf{b}$. Backpropagation is the chain rule for matrices.
- **SVD and tensors:** Model compression via low-rank factorisation. Tensor decompositions for efficient computations.
- **FFT:** Convolutional layers use FFT for acceleration. Attention via FFT (Linear Attention).
- **Nonlinear optimisation:** Training neural networks is the minimisation of a loss via SGD, Adam (adaptive methods), L-BFGS (for fine-tuning).
- **Compressed sensing:** Sparse training, pruning of neural networks.
- **Graphs and tensors:** GNN work with molecular graphs. 3D models use tensors of coordinates.
- **Monte Carlo:** Diffusion models are stochastic processes. Variational inference — a Bayesian approach.

Main conclusion

Machine learning is not a replacement for classical chemistry, but a **powerful tool** that:

- Accelerates computations thousands of times (property prediction vs DFT),
- Opens new possibilities (protein structure prediction, molecule generation),
- Requires understanding of mathematics (linear algebra, optimisation, probability theory).

The modern chemist must know:

- The basics of ML (neural networks, GNN, transformers),
- The tools (RDKit, PyTorch, DeepChem),
- When ML works and when it does not (physical limitations, interpretability).

21 Modern computing hardware: from the transistor to the supercomputer

21.1 Numbers worth knowing

A modern computer — a workstation — is:

- 128–1024 GB of RAM,
- ∼500 Gflop/s of peak performance (billions of floating-point operations per second),
- Several processors (from 4 to 128 cores).

But here is a paradox: the speed of memory access is **hundreds of times** slower than the speed of arithmetic operations. And if access is random (not sequential read-write), then another **hundred times** slower.

Why is that? Let us sort it out.

21.2 How a processor is arranged

Everyone says that a processor has "many transistors". But what do they do?

In fact, a processor is an entity that accesses **instructions** and accesses **data**. Both take up space in memory.

21.2.1 Processor commands

The processor reads commands, which usually allow:

- Accessing data in memory and placing them in **registers** — a temporary, very, very fast memory,
- Performing an operation: addition, subtraction, multiplication, comparison,
- Making a **jump**: by default the processor executes command after command, but if it encounters a jump instruction, it can leap forward or backward.

Jumps are:

- **Unconditional:** The jump always happens,
- **Conditional:** If we have previously compared numbers and have the answer "yes" or "no", then the jump happens only if the condition is satisfied.

21.2.2 SIMD: one command — many data

A modern processor command can work with several numbers at once. For example, one AVX-512 instruction can perform 16 pairs of additions simultaneously!

This is called **SIMD** (Single Instruction, Multiple Data) — one instruction, many data. If the data are already on the processor itself, this is quite easy to do.

Where did SIMD come from?

People noticed that one often has to perform identical operations: for example, transform pixels identically to draw a picture, process array elements identically. Why execute the same instruction 16 times if one can do it once — but for 16 numbers at once?

But to perform such operations with *different* data with each command, we would need a very powerful system for pumping data from memory to the processor and back.

21.2.3 The memory wall problem

Unfortunately, such a system would occupy hundreds and thousands of times more transistors than are needed for an arithmetic block.

And people noticed that in programs we often work with some small set of numbers, and then switch to another set, and so on. Making a small block of memory with fast access is not so costly in terms of the number of transistors. The only trouble is that forcing the programmer to drag data back and forth into this block from RAM and back would be difficult.

Therefore people automated this — and the **processor cache** appeared. It is fast memory, there is not much of it, but the processor itself "remembers" what you are using now, and for some time keeps these data in its cache, expecting that later they may be needed again.

21.3 Memory hierarchy

A modern computer is a multilayer memory structure:

Level	Size	Speed	Description
Registers	32–128 × 64 bits	∼0.3 ns	Cells inside the processor that it works with directly
L1 cache	∼32–64 KB	∼1 ns	Separate for instructions and data, own for each core
L2 cache	∼256 KB – 1 MB	∼3–5 ns	Own for each core or shared by a pair of cores
L3 cache	∼8–64 MB	∼10–20 ns	Shared by all cores of the processor
RAM (DRAM)	128–1024 GB	∼50–100 ns	Main memory
Disk (SSD)	1–10 TB	∼10–100 μs	Long-term storage

Let us compare how much a modern workstation can on average do in one nanosecond:

- **Arithmetic:** All its processors, taking into account long SIMD instructions, can perform about **1000 floating-point operations**.
- **L1 cache:** One can read and write 2–3 numbers on each processor, in total ∼200 numbers.
- **L2/L3 cache:** This quantity falls by tens of times.
- **RAM:** Only a few reads and writes per nanosecond, and only if this happens in large blocks of about 512 bytes approximately sequentially.
- **Random access:** If each time we address different blocks, the speed falls by another tens of times — several nanoseconds may be needed per number.

Memory Wall

Random access to memory is **tens of thousands of times** slower than the real computational power of a modern processor!
This is the main bottleneck of modern computations. That is precisely why:

- Matrix algorithms work faster if the data are located "densely" in memory,
- Sparse matrices are slow (many random accesses),
- Cache-oriented algorithms are a separate science.

21.4 Graphics processing units (GPU)

But that was not enough for people. When they drew 3D objects of computer graphics on the screen, they noticed that all this can be done almost entirely **in parallel**.

Roughly: if we had 1024 processors, we would split the whole screen into a grid of 32×32 squares, and each processor would draw in its own square. Thus graphics cards appeared — they have a host of small specialised processors that work fully in parallel.

21.4.1 From graphics to computations

In the late 1990s such graphics cards were used to accelerate the display of graphics. Around 2005 people began to produce such cards so that small pieces of code could be executed on thousands of such small processors — thus appeared **CUDA** (NVIDIA) and **OpenCL** (an open standard).

By 2010 all numerical methods began to migrate to graphics cards. Gradually, to single and double floating-point arithmetic, people added the so-called **half** (FP16) and **quarter** (FP8, INT8) floating points — because with them it was faster to perform computations and they occupied less memory.

In modern NVIDIA graphics cards (H100, B200) it is possible to perform about **a million operations** with such small objects in the same one nanosecond — and this has allowed using such graphics cards for modern artificial intelligence systems.

21.4.2 GPU architecture

A modern GPU (for example, NVIDIA H100) is:

- **Streaming Multiprocessors (SM):** $\sim$132 multiprocessors,
- **CUDA cores:** $\sim$16 896 cores for FP32 arithmetic,
- **Tensor cores:** $\sim$528 specialised cores for matrix operations (FP16, FP8, INT8),
- **Memory:** 80 GB HBM3 (High Bandwidth Memory) with a bandwidth of $\sim$3 TB/s.

21.4.3 Threads and warps

A GPU works with **threads**. Thousands of threads execute in parallel, but they are organised into groups:

- **Warp:** A group of 32 threads that execute **synchronously** — all 32 threads execute the same instruction at the same moment in time.
- **Block:** A group of warps (up to 1024 threads) that can exchange data through a shared **shared memory**.
- **Grid:** All blocks launched on the GPU.

Rule of GPU efficiency

For a GPU to work efficiently, the programs on each multiprocessor for each thread must **not diverge**. This means:

- All 32 threads in a warp must execute *the same* instruction (without conditional jumps that split the threads — "warp divergence"),
- All threads must access *sequential* memory addresses (coalesced memory access),
- There must be enough threads to "hide" the memory latencies.

If these conditions are not met — the GPU works several times more slowly.

21.5 Parallel computing: Flynn's taxonomy

In 1966 Michael Flynn proposed a classification of parallel architectures by instruction and data streams:

Type	Description	Example
SISD	Single Instruction, Single Data. One instruction stream, one data stream. Classical sequential processor.	Old CPUs
SIMD	Single Instruction, Multiple Data. One instruction applied to many data.	Vector instructions (AVX), GPU
MISD	Multiple Instruction, Single Data. Different instructions on the same data. Rarely encountered.	Some specialised architectures
MIMD	Multiple Instruction, Multiple Data. Many processors, each executing its own instructions on its own data.	Modern multiprocessors, clusters

21.5.1 Shared and distributed memory

In MIMD systems there are two approaches:

- **Shared memory:** All processors have access to a single memory. Example: a multi-core CPU. Easy to program (all see the same data), but difficult to scale (memory access conflicts).
- **Distributed memory:** Each processor has its own local memory, data exchange — through a network. Example: clusters, supercomputers. More difficult to program (one must explicitly transfer data — MPI), but it scales to millions of cores.

21.6 TOP500: the most powerful supercomputers in the world

The TOP500 list is a ranking of the 500 most powerful supercomputers in the world, updated twice a year (June and November). Performance is measured in the LINPACK benchmark — solving a large system of linear equations.

21.6.1 Modern leaders (2024–2025)

Name	Rank	Performance	Architecture
Frontier	#1	∼1.2 Eflop/s	AMD EPYC + AMD Instinct MI250X
Aurora	#2	∼1 Eflop/s	Intel Xeon + Intel Max PVC
Eagle	#3	∼561 Pflop/s	AMD EPYC + NVIDIA H100
Fugaku	#4	∼442 Pflop/s	Fujitsu A64FX (ARM)
LUMI	#5	∼231 Pflop/s	AMD EPYC + AMD MI250X

Scale

1 Eflop/s = 10^{18} operations per second. Frontier is ∼1.2 million billion operations per second. For comparison: your laptop is ∼0.1 Tflop/s = 10^{11} operations per second. The difference is 10 million times!

21.6.2 How a modern supercomputer is arranged

Typical architecture:

1. **Nodes:** ∼10 000–100 000 servers, each with 2–4 CPUs + 4–8 GPUs,
2. **Network:** High-speed interconnect (InfiniBand, Slingshot) with a bandwidth of 200–400 Gbit/s per node,
3. **Storage:** A parallel file system (Lustre, GPFS) with a bandwidth of ∼1–10 TB/s,
4. **Cooling:** Liquid cooling (water or even two-phase), since the power is 20–50 MW.

21.7 Digitisers: a bridge between the analogue and digital worlds

But the processor — after all, it must get data for computation from somewhere. And here **analog-to-digital converters** (ADC) enter the stage.

They are determined by two parameters:

- **Accuracy (bit depth):** How many bits are used to encode one sample,
- **Speed (sampling rate):** How many samples per second.

21.7.1 The trade-off: accuracy vs speed

There is a fundamental trade-off: the higher the accuracy, the lower the speed.

Bit depth	Speed	Minimum bit	Application
12 bit	∼10 Gsamp/s	∼500 μV	Oscilloscopes, fast imaging
16 bit	∼1 Gsamp/s	∼10–50 μV	Audio, NMR, precise measurements
24 bit	∼4 Msamp/s	∼100–500 nV	High-resolution audio, seismography

Why are nanovolts not measured directly?

24 bits give a range of tens and hundreds of nanovolts. But nobody measures nanovolts directly — radar interference from everything around is about tens of microvolts! That is, one usually measures *currents* there, not volts — or uses special shielded chambers.

21.8 The physics of switching: from the transistor to megavolts

In a modern processor everything works by turning transistors on and off at about 0.7 V. The speed of such switching is tens of gigahertz. But the currents there are very small (microamperes per transistor).

21.8.1 The trade-off: voltage vs speed

If the voltage is increased, the switching speed begins to fall:

Voltage	Switching time	Technology
0.7 V	∼0.05 ns (20 GHz)	Modern CMOS transistors
40 V	∼3–5 ns	Power MOSFETs (ordinary ratings)
1000 V	∼3–5 ns	Power MOSFETs (top ratings)
4.5–5 kV	∼20–30 ns	IGBT, high-voltage MOSFETs
>10 kV	∼100 ns – 1 μs	A single semiconductor can no longer cope
1 MV	∼100–500 μs	Tubes, semiconductor assemblies, multipliers

Although modern CMOS transistors switch at frequencies around 20 GHz, to execute one processor cycle one must have a margin of several such switchings, therefore the clock frequency of processors is in the range of about 3-4 GHz.

21.8.2 The limit of voltage rise

Above ∼5 kV a single semiconductor can no longer cope — one must switch to:

- **Good old tubes** (!) — vacuum triodes, thyratrons,
- **Semiconductor assemblies** — series connection of several transistors,
- **Voltage multipliers** — cascade circuits with tubes or spark gaps.

Creating even 1 megavolt is not very difficult (Marx generator, Cockcroft–Walton cascades). Another matter is to turn this megavolt on and off in a nanosecond. That really requires hundreds of microseconds.

Fundamental limit

In fact $\sim 10^{12}$ V/s is the modern limit of voltage rise practically over the entire range of possible voltages.
This is a physical limitation associated with:

- The speed of motion of charge carriers in semiconductors (limit $\sim 10^7$ cm/s),
- Parasitic capacitances and inductances,
- The speed of propagation of electromagnetic waves in a medium.

21.9 Connection with previous chapters

Let us see how everything we have covered is connected with the “hardware”:

- **Linear algebra:** Matrix operations are the basis of all computations. GPUs are optimised precisely for them (Tensor Cores).
- **FFT:** The fast Fourier transform is $\mathcal{O}(N \log N)$ operations, and it is critical for NMR, signal processing, data compression. GPUs accelerate FFT by tens of times.
- **Iterative methods:** The conjugate gradient method, GMRES — these are the basis for solving large sparse systems. They work on supercomputers with distributed memory (MPI + OpenMP + CUDA).
- **Machine learning:** Training neural networks is billions of matrix multiplications. Without GPUs and Tensor Cores modern models (GPT-4, AlphaFold) would be impossible.

- **Compressed sensing:** L1 minimisation is an iterative method that requires thousands of iterations. GPUs accelerate it 100–1000 times.
- **Digitisers:** ADC is the bridge between the analogue world (spectra, signals) and the digital one (processor). The accuracy of the ADC determines what mathematics we can apply.

Main conclusion

Modern computing hardware is:

- **Memory hierarchy:** Registers → L1 → L2 → L3 → RAM → disk. Each step is 10–100 times slower, but 10–1000 times larger.
- **Parallelism:** SIMD (vector instructions), multi-core (CPU), massive parallelism (GPU), clusters (supercomputers).
- **Specialisation:** Tensor Cores for AI, FPGA for specific tasks, ASIC for mining.
- **Physical limitations:** Memory wall, power wall, speed of light — all this limits the growth of performance.

For a chemist this means:

- Understanding where the “bottleneck” is (memory or computations),
- Being able to write code that efficiently uses the cache and GPU,
- Knowing when a problem is solved on a workstation, and when a supercomputer is needed.

22 Afterword: How to use this knowledge

Previously, after reading a similar book, when you encountered a real problem, there arose a need to implement a solution. For this one had to be quite good at one or several programming languages and understand how to use third-party software systems and packages.

In the modern age of artificial intelligence development all this can be done with the support of AI systems — and it will be significantly more effective. But for this one must understand a little the key applications of programming languages and the so-called **workflow** — how program development happens.

22.1 Two worlds of programming languages

Although there are a huge number of programming languages, they can be divided into two fundamentally different classes:

Compiled	**Interpreted / bytecode**
C, C++, CUDA, Fortran, Rust	Python, JavaScript, Java, C#, R
Code is compiled into machine instructions for a specific processor	Code is interpreted on the fly or compiled into bytecode
Maximum performance, direct access to the “hardware”	Flexibility, cross-platform, fast development
When to use: — Heavy computations (DFT, MD, ML) — GPU acceleration (CUDA) — Embedded systems	**When to use:** — Prototyping, data analysis — Visualisation, reporting — Web interfaces, scripts

22.1.1 Why interpreted languages are convenient

Interpreted languages allow achieving greater flexibility when moving from one platform to another. For example, we open the same web page on a desktop, on a smartphone and on a tablet — and see the same content. This content is usually generated using the interpreted language JavaScript, and if we had to send our code for each of the processors every time, the server would have to have all variants of such code in advance. This is sometimes even impossible to foresee — since even different Android smartphones may have different processor architectures.

22.1.2 Typical chemist’s workflow

In order to quickly display some results, it is often easier to use interpreted programming languages (Python — the absolute standard in science). But when the task is complex and requires many computational resources, it becomes necessary to use compiled programming languages.

A typical workflow looks like this:

1. **Prototype in Python:** Quickly check the idea, visualise the data, understand whether the algorithm works.
2. **Optimisation:** If the code works but is slow — rewrite the “bottlenecks” in C++/CUDA or use ready-made libraries (NumPy, SciPy, PyTorch).
3. **Production:** If the task is solved regularly — package it, add tests, documentation.

22.2 Working with AI assistants

Nowadays practically any algorithm can be implemented by writing a correct prompt with the help of chats. But it is worth noting: each chat is almost like a person. And if you give it a task that is not fully thought through, it may also muddle things, or, not understanding, program something not quite right.

22.2.1 Golden rules of working with AI

1. **Break the task into small blocks.** Do not ask “write a program for solving the Schrödinger equation”. Ask: “write a function that computes the Fock matrix for a given basis”.
2. **Ask for tests.** For each block, ask the chat to write test verification programs, run such tests and make sure everything is in order.
3. **Combine gradually.** Only after each block works, combine them to obtain the final result.

4. **Demand explanations.** If the chat has written something you do not understand — ask it to explain. If the explanation is unclear — ask for something simpler. This is your script, you must understand every line.

5. **Check on known examples.** If you are solving a system of equations — check on a system for which you know the answer. If you are doing a Fourier transform — check on a sinusoid with a known frequency.

Typical mistakes

- **Blind trust:** AI may write code that "looks right" but contains subtle errors (for example, incorrect normalisation, wrong array bounds, memory leaks).
- **Ignoring context:** AI does not know your specific task — you must explain it in detail, with examples, with constraints.
- **Absence of tests:** Code without tests is code that will break at the most inopportune moment.

22.3 Must-have tools for a chemist

Here is the minimum set of tools you should know:

Language	Application
Python	The absolute standard: data analysis, ML, visualisation, scripts
R	Statistical analysis, bioinformatics
MATLAB	Engineering computations, signal processing (an alternative to Python)
C++/CUDA	High-performance computing, GPU
Bash/Shell	Automation, working with clusters

Library	Purpose
NumPy, SciPy	Linear algebra, optimisation, integration
Pandas	Working with tabular data
Matplotlib, Seaborn	Visualisation
PyTorch, TensorFlow	Machine learning, neural networks
RDKit	Chemoinformatics, molecules
ASE, PySCF	Quantum chemistry
OpenMM, GROMACS	Molecular dynamics

22.4 Final advice

Mathematics is not a set of formulas to be memorised. It is a **way of thinking**. It is the ability to see structure in chaos, to find patterns, to build models, to test hypotheses.

When you encounter a real problem — whether it is protein structure prediction, NMR spectrum analysis, synthesis optimisation or something completely new — remember this script. Remember that:

- Any problem is either a system of equations, or an optimisation problem, or an approximation problem,
- For any problem there are proven mathematical methods,
- Modern tools (Python, GPU, AI) allow solving problems that were impossible 20 years ago,
- The main thing is to understand the essence of the problem, and the rest is technique.

We believe in you. You will succeed. And if someday this script helps you solve a problem that changes the world — we will be incredibly proud.

Last word

Do not be afraid to make mistakes. Do not be afraid to ask. Do not be afraid to experiment. Mathematics is not an exam, it is an adventure. And you are at the beginning of the most interesting path.
Good luck!

With love,
Papa and Mama

September 2026