Математика, Вычислительные методы и Обработка сигналов

Скрипт для студентов химического факультета TUM

Ильгиз Ибрагимов & Елена Ибрагимова & ИИ*

Сентябрь 2026
DOI: 10.5281/zenodo.23002397

Посвящается нашим дочерям Марие и Алевтине.
Пусть математика станет для вас не набором формул,
а языком, на котором написана Вселенная.

*ассистент по правкам и переводу

Содержание

1 Предисловие

Дорогие наши дочери!

Нам так хотелось, чтобы в вашем жизненном пути всё было понятно и просто. Мы знаем, что часто математику объясняют, специально запутывая человека терминами и сложными формулами — так, что за деревьями не видно леса.

Этот скрипт — наша попытка дать обзор математических и вычислительных идей, особенно полезных современному химику, который работает с экспериментальными данными, моделированием и вычислениями. Наша цель — чтобы у тебя была *понятная картина* современного уровня математических знаний. Не всех знаний — это невозможно в одной книге — а именно тех, которые применяются в химии, биохимии, спектроскопии, молекулярном моделировании и обработке данных.

Мы прошли вместе длинный путь:

- От самых основ — множеств чисел, комплексных чисел, векторов и матриц,
- Через линейную алгебру — SVD, обусловленность, методы решения систем,
- К нелинейным задачам — оптимизация, градиенты, метод Ньютона,
- Через обработку сигналов — Фурье, Прони, сжатые измерения,
- К численным методам — интегрирование, конечные элементы, базисные функции,
- И наконец — к машинному обучению, нейросетям и современной вычислительной технике.

Всё это — не просто набор разрозненных тем. Это — **единый язык**, на котором говорит современная наука. И если вы овладеете этим языком — перед вами откроются двери, о которых вы сейчас даже не подозреваете.

1.1 Как читать эту книгу

Хотя текст написан так, чтобы сделать его максимально просто читаемым, без детальных математических доказательств, некоторые места могут быть слишком заумно сформулированными. Это нормально — математика не даётся сразу.

Поэтому к тексту прилагается исходник этой книги в LaTeX. Если что-то будет не сильно понятно:

1. Найдите соответствующий текст в исходнике,
2. Скопируйте его в какой-нибудь чат (Qwen, DeepSeek, Kimi, ChatGPT, GROK, Gemini),
3. Попросите его понятнее рассказать про это, или задайте вопрос, который вы тут не понимаете.

Также с помощью этих чатов можно перевести весь текст на английский или немецкий — для этого надо просто попросить перевести с сохранением LaTeX-разметки и полученный текст скомпилировать командой `xelatex NumMathForChemists.tex` в PDF-файл.

Совет

Не пытайтесь прочитать всё за один присест. Математика — это как музыка: её надо “играть”, то есть решать задачи, пробовать код, экспериментировать. Если какая-то глава кажется сложной — вернитесь к ней через неделю, и вы удивитесь, насколько проще она станет.

2 Введение: Обозначения и природа чисел

2.1 Иерархия числовых множеств

Прежде чем мы начнем говорить о сложных вещах, давай договоримся о языке. В математике мы группируем числа в множества, обладающие определенными свойствами. Мы будем использовать следующие стандартные обозначения:

- $\mathbb{N} = \{1, 2, 3, \dots\}$ — **натуральные числа**. Используются для счета.
- $\mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$ — **целые числа**. Добавляют ноль и отрицательные числа, позволяя описывать долг и имущество, температуру ниже нуля и т.д.
- $\mathbb{Q}$ — **рациональные числа**. Любое число, которое можно представить в виде дроби $\frac{p}{q}$, где $p \in \mathbb{Z}$, $q \in \mathbb{Z} \setminus \{0\}$.
- $\mathbb{R}$ — **действительные (вещественные) числа**. Включают в себя все рациональные, а также иррациональные числа (например, $\sqrt{2}, \pi, e$), которые нельзя представить в виде обычной дроби. Они непрерывно заполняют всю числовую прямую.
- $\mathbb{C}$ — **комплексные числа**. О них мы подробно поговорим ниже.

Вложенность множеств: $\mathbb{N} \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R} \subset \mathbb{C}$.

2.2 Числа в компьютере

Мы привыкли, что в математике числа бесконечно точны. Но когда мы переходим к **вычислительной математике** и программированию, компьютер вынужден хранить числа в ячейках памяти фиксированного размера (в битах и байтах).

- **Целые типы** (int8, int16, int32, int64) хранят точные значения из $\mathbb{Z}$, но ограничены диапазоном (например, int32 может хранить числа от -2^{31} до $2^{31} - 1$).
- **Вещественные типы** (float32, float64 / double) хранят числа из $\mathbb{R}$ в экспоненциальном формате (мантисса и порядок). Из-за этого возникают ошибки округления: компьютерное $0.1 + 0.2$ не всегда в точности равно 0.3. Понимание этого критически важно для численных методов!

3 Когда одного числа мало: Комплексные числа

Наш мир настолько сложен, что не всегда все можно описать одним числом. В физике и химии нам часто приходится работать с сущностями, которые требуют *двух* чисел для своего описания (например, амплитуда и фаза волны, или действительная и мнимая часть волновой функции).

3.1 Откуда они берутся?

Самый простой и исторический способ познакомиться с “числами-парами” — это попытаться решить уравнение:

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

В множестве действительных чисел $\mathbb{R}$ квадрат любого числа неотрицателен. Корня из -1 не существует. Но математики (и физики вслед за ними) сказали: “А что, если мы просто *придумаем* такое число, которое при умножении само на себя дает -1?”.

Так появилась **мнимая единица** i. Мы не “извлекаем корень” из -1, мы *определяем* новую сущность:

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

Важное замечание: правило $\sqrt{a}\sqrt{b} = \sqrt{ab}$ *работает только для* $a, b \geq 0$. *Поэтому мы не пишем* $\sqrt{-1}\sqrt{-1} = \sqrt{(-1)(-1)} = 1$. *Мы просто принимаем* $\mathrm{i}^2 = -1$ *как факт.*

3.2 Алгебраическая форма и комплексная плоскость

Любое комплексное число $z \in \mathbb{C}$ можно записать в виде:

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

где $x = \mathrm{Re}(z)$ — **действительная часть**, а $y = \mathrm{Im}(z)$ — **мнимая часть**. Обе части $x, y \in \mathbb{R}$.

Геометрически комплексное число — это точка на **комплексной плоскости**.

- По горизонтальной оси (ось Re) откладывается действительная часть x.
- По вертикальной оси (ось Im) откладывается мнимая часть y.

Комплексное число также можно рассматривать как **вектор**, идущий из начала координат $(0, 0)$ в точку (x, y).

3.3 Арифметика комплексных чисел

Сложение интуитивно понятно. Если у нас есть два числа $z_1 = x_1 + \mathrm{i}y_1$ и $z_2 = x_2 + \mathrm{i}y_2$, то мы просто складываем их действительные и мнимые части отдельно:

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

Геометрически это работает как **правило треугольника** для векторов: мы строим два вектора от начала координат, переносим начало второго вектора в конец первого, и сумма — это вектор от начала первого до конца второго.

Умножение в алгебраической форме выглядит чуть сложнее. Мы раскрываем скобки как в обычной алгебре, помня, что $\mathrm{i}^2 = -1$:

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

Но чтобы понять *истинную магию* умножения комплексных чисел, нам нужно перейти к другой форме записи.

3.4 Тригонометрическая и экспоненциальная формы

Вместо координат (x, y) точку на плоскости можно задать с помощью полярных координат: расстояния от начала координат (модуля) r и угла φ относительно положительного направления оси Re.

Связь с алгебраической формой очевидна из тригонометрии:

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

Тогда комплексное число принимает **тригонометрическую форму**:

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

3.4.1 Формула Эйлера и экспоненциальная форма

Здесь на сцену выходит величайшая формула математики — **формула Эйлера**:

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

Благодаря ей, любое комплексное число можно записать в невероятно удобной **экспоненциальной форме**:

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

3.4.2 Геометрический смысл умножения

Давай посмотрим, что происходит при умножении двух чисел в экспоненциальной форме:

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

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

Вывод, который нужно запомнить навсегда: Умножение комплексных чисел — это **умножение их длин (модулей)** и **сложение их углов (аргументов)**! Геометрически: умножение на комплексное число — это поворот вектора на угол φ и его растяжение/сжатие в r раз.

3.5 Комплексные экспоненты и логарифмы

3.5.1 Свойства экспоненты

Поскольку $e^{\mathrm{i}\varphi}$ ведет себя как обычная экспонента, для нее работают все стандартные правила:

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

Для комплексного числа $z = x + \mathrm{i}y$ экспонента раскладывается на действительную и мнимую части:

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

Модуль этого числа равен e^x, а аргумент — y.

3.5.2 Комплексный логарифм

Логарифм — это операция, обратная экспоненте. Если $e^w = z$, то $w = \ln z$. Пусть $z = re^{\mathrm{i}\varphi}$ и $w = u + \mathrm{i}v$. Тогда:

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

Приравнивая модули и аргументы, получаем:

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

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

Таким образом, **комплексный логарифм** многозначен:

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

Главное значение логарифма (при $k = 0$ и $\arg z \in (-\pi, \pi]$) обозначается как $\mathrm{Ln}\, z$. Многозначность возникает из-за того, что поворот на 2π возвращает нас в ту же точку на комплексной плоскости.

3.6 Шпаргалка: Тригонометрия и Гиперболические функции

Гиперболические функции часто пугают студентов-химиков, но на самом деле они — близкие родственники обычных синусов и косинусов, просто “живущие” в комплексной плоскости.

3.6.1 Определения через экспоненту

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

3.6.2 Связь тригонометрических и гиперболических функций

Если мы подставим $\mathrm{i}z$ вместо z в определения, то увидим удивительную симметрию (формулы Осборна):

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

Запоминалка: При переходе от тригонометрии к гиперболике аргумент умножается на i, а перед синусом/косинусом могут появляться мнимые единицы. Гиперболические функции описывают не колебания (как синус), а экспоненциальный рост/спад (как e^x), что критически важно для затухающих процессов.

3.7 Зачем это нужно в реальной химии и физике?

Может показаться, что мнимые числа — это просто красивая математическая абстракция. Но наш мир устроен так, что *без* мнимого пространства мы не смогли бы описать реальные вещи.

3.7.1 Квантовая механика и орбитали

Уравнение Шрёдингера, которое описывает поведение электронов в атомах, содержит мнимую единицу i в явном виде:

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

Волновая функция Ψ — комплекснозначная. Для основного состояния атома водорода (1s-орбиталь) волновая функция действительная, и ее можно нарисовать в реальном пространстве. Но как только электрон переходит в возбужденное состояние (например, 2p-орбиталь с магнитным квантовым числом $m \neq 0$), его волновая функция содержит фазовый множитель $e^{\mathrm{i}m\varphi}$. Без комплексных чисел описать угловой момент электрона и тонкую структуру спектров просто невозможно.

3.7.2 Ядерный Магнитный Резонанс (ЯМР) и Фурье-анализ

Вы будете много работать с ЯМР-спектроскопией. Когда вы помещаете образец в магнитное поле и даете радиочастотный импульс, ядра начинают прецессировать. Детектор регистрирует сигнал, затухающий со временем — это называется **FID** (Free Induction Decay).

Сигнал от одного типа ядер выглядит как затухающая осцилляция:

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

Если в образце много разных ядер, сигнал превращается в кашу из множества наложенных друг на друга косинусов. Как понять, какие частоты ω там спрятаны?

Здесь на помощь приходит **Преобразование Фурье**. Но работать с косинусами сложно. Гораздо проще перейти к комплексным экспонентам, используя формулу Эйлера:

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

В ЯМР-спектрометрах используют квадратурное детектирование, которое позволяет измерять сразу комплексный сигнал:

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

Когда мы применяем к этому сигналу быстрое преобразование Фурье (БПФ), время t переходит в частоту ω. Длинная осциллирующая функция во временной области превращается в **узкий пик (Лоренцеву кривую)** в частотной области! Каждому типу ядер соответствует своя частота ω, и на итоговом спектре ЯМР мы видим отдельные пики. Вся современная обработка сигналов (от МРТ в медицине до аудио MP3) работает именно благодаря магии комплексных чисел и экспонент Эйлера.

4 Когда одного измерения мало: Векторы и Нормы

4.1 От плоскости к N-мерному пространству

Что же, если у нас есть комплексные числа (по сути, пары чисел), то, наверное, есть и что-то более сложное? Кватернионы? Октонионы?

Да, действительно, математики обобщили числа с плоскости на большие размерности. Но за пределами комплексных чисел и кватернионов (которые, кстати, отлично прижились в робототехнике и 3D-графике для описания вращений в нашем трехмерном пространстве) специальные "типы чисел" практически не используются. Вместо этого люди просто группируют числа в наборы и называют их **N-мерными векторами**.

Это просто N-мерное пространство. N может быть равно двум (тогда это плоскость), трем (наше обычное физическое пространство) или очень-очень большим. Такие векторы мы будем обозначать жирным шрифтом, например $\mathbf{v} \in \mathbb{R}^N$. Это вектор:

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

Здесь T означает транспонирование, чтобы записать столбец в одну строку текста. Мы сами вольны придумывать, что делать с этим пространством, но первое, что нам нужно — это понять, как измерять "размер" или "длину" такого вектора.

4.2 Зачем нам векторы? Примеры из химии

На первый взгляд, вектор — это просто набор чисел. Но числа-то откуда-то пришли! Допустим, это данные с хроматографа или ЯМР-спектрометра.

Представь, что мы хотим узнать, какой же тут самый "яркий" пик. Так это же просто максимум среди этих чисел, верно? А теперь давай возьмем хроматограмму какой-нибудь сложной грязи и хроматограмму одного чистого вещества. Мы вколем в хроматограф и то, и другое в одинаковом молярном объеме (пусть условимся, что детектор одинаково чувствует все молекулы).

- В случае чистого вещества мы получим один красивый, очень острый и высокий пик.
- В случае грязи мы получим гору пиков, может быть, даже слившихся в один сплошной пологий горб.

Но вот что интересно: **площадь** (интегральная интенсивность) обеих хроматограмм будет одинакова! Сам пик чистого вещества будет очень высоким по сравнению с горой, но сумма всех откликов равна количеству вещества.

Из этой задачи естественно возникают три разных способа оценить наш вектор данных:

1. **Максимум.** Нам важна максимальная концентрация (или максимальное поглощение, если пик "смотрит" вниз). Мы берем самое большое значение.

2. **Сумма.** Нам важно общее количество вещества. Мы складываем все значения (обычно по модулю, так как базовая линия может быть не идеальной).

3. **Корень из суммы квадратов.** А это зачем? Представь, что мы описываем молекулу не хроматограммой, а тремя параметрами: *полярность, молярная масса, температура кипения*. Это вектор в $\mathbb{R}^3$. Чтобы понять, насколько две молекулы *похожи* друг на друга (например, для дизайна лекарств), мы не можем просто сложить разности параметров или взять максимальную разность. Нам нужно именно **евклидово расстояние** — кратчайший путь в этом "химическом пространстве". Или другой пример: в физике квадратичная норма вектора (спектра) пропорциональна **полной энергии** этого сигнала.

4.3 Математический язык норм

Все эти три интуитивные понятия в математике называются **нормами векторов** и обозначаются двойными вертикальными чертами $\|\mathbf{v}\|$.

- **Максимальная норма (L_∞):**
$$\|\mathbf{v}\|_\infty = \max_{1\le k\le N} |v_k|$$

- **Манхэттенское расстояние / Сумма модулей (L_1):**
$$\|\mathbf{v}\|_1 = \sum_{k=1}^{N} |v_k|$$

- **Евклидова норма / Длина вектора (L_2):**
$$\|\mathbf{v}\|_2 = \sqrt{\sum_{k=1}^{N} |v_k|^2}$$

Но математики любят порядок, поэтому они объединили всё это в одну общую формулу для L_p**-нормы**:

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

В этой формуле L_1 — это случай, когда $p = 1$. L_2 — когда $p = 2$. А что же будет в качестве p для максимума? Да всё верно, $p \to \infty$. Если взять предел этой формулы при $p \to \infty$, то математически строго получится именно максимальный элемент вектора.

4.4 Непрерывный случай: Функции как векторы

Более того, наш вектор может быть совсем непрерывным. Тогда это на самом деле — **функция** $f(x)$. Сейчас мы только запомним, что функцию можно рассматривать как "бесконечномерный вектор", значения которого заданы в каждой точке x. И для функций нормы работают точно так же, только сумму мы заменяем на интеграл:

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

Мы будем иногда оперировать то массивом (вектором), то функцией, а позже, в курсе обработки сигналов, увидим, что это две стороны одной медали.

4.5 Неравенство Минковского

Есть один критически важный момент, про который нужно помнить. Мы часто будем сравнивать $\|\mathbf{x}+\mathbf{y}\|$ с $\|\mathbf{x}\|$ и $\|\mathbf{y}\|$. Интуитивно понятно, что если мы пройдем из точки A в точку B, а потом из B в C, то общий путь будет не меньше, чем если бы мы пошли напрямую из A в C.

Это геометрическое свойство (длина стороны треугольника не больше суммы двух других) для любых L_p-норм формализуется в **неравенстве Минковского**:

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

Как это использовать? Это позволяет нам оценивать ошибки. Если $\mathbf{x}$ — это истинный сигнал, а $\mathbf{y}$ — это шум (ошибка измерения), то неравенство Минковского гарантирует, что норма (размер) нашего измеренного сигнала $(\mathbf{x}+\mathbf{y})$ не превысит сумму нормы истинного сигнала и нормы шума. *Строгое и красивое доказательство этого неравенства можно посмотреть в Википедии, чтобы не перегружать этот скрипт.*

5 Два измерения и больше: Матрицы и Тензоры

5.1 От векторов к матрицам

Раз у нас есть вектор $\mathbf{a} \in \mathbb{R}^N$ (одномерная сущность, дискретный аналог функции $f(x)$), то логично спросить: а что если у нас функция двух переменных $f(x,y)$? Каков её дискретный аналог? А для $f(x,y,z)$?

Да, двухмерные объекты в дискретном пространстве — это **матрицы**. И как и в случае с комплексными числами, всё тут очень-очень хорошо изучено. А всё, что имеет три и более измерений, обычно называется **тензорами**.

Кстати, если ты когда-нибудь столкнешься со старой научной литературой по хемометрике или анализу данных, ты можешь удивиться зоопарку названий для тензоров. Их там называют *multilinear, multidimensional, procrustes, multimodal, three-way, multi-way arrays*. Не пугайся, под всеми этими красивыми словами скрывается просто многомерный массив чисел.

Пока давай остановимся на двумерных объектах: функция $f(x,y)$ и матрица $\mathbf{A} \in \mathbb{R}^{N\times M}$.

5.2 Комплексные векторы и матрицы

Почему я до сих пор писал только $\mathbb{R}$? И вектор, и матрица, могут быть, конечно же, комплексными! Мы можем рассмотреть пространства $\mathbb{C}^N$ и $\mathbb{C}^{N\times M}$. Тут всё просто: вместо одного действительного числа в каждой ячейке вектора или матрицы стоит комплексная пара.

Но как считать норму для комплексного числа? Да всё то же самое! Модуль комплексного числа $z = x + \mathrm{i}y$ вычисляется через сопряженное число $\bar{z} = x - \mathrm{i}y$:

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

Соответственно, L_2-норма комплексного вектора $\mathbf{z} \in \mathbb{C}^N$ будет выглядеть так:

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

В матричной нотации это записывается через эрмитово сопряжение (транспонирование + комплексное сопряжение) $\mathbf{z}^H$:

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

5.3 Нормы матриц

А для матриц мы тоже можем задать нормы! Но тут есть важный нюанс.

Большинство простых норм для матриц считаются так: мы мысленно “разрезаем” матрицу на строки, выстраиваем все её $N \times M$ ячеек в один длинный вектор-столбец и берем норму уже этого вектора. Самая популярная из таких норм — **норма Фробениуса** (или евклидова норма для матриц), которая является аналогом L_2-нормы:

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

Для неё, как и для обычной L_2, есть масса важных и красивых свойств, которые мы скоро рассмотрим.

Однако, кроме таких “поэлементных” норм, в линейной алгебре есть специальные **операторные (согласованные) нормы**. Они отвечают на вопрос: “Насколько сильно матрица может растянуть вектор, если мы умножим её на него?”. Но это уже история для следующей главы, где мы вплотную займемся линейными операторами.

6 Операторы: когда математика начинает действовать

6.1 Что такое оператор?

Мы подошли к одной очень важной концепции, которая сплошь и рядом встречается и в математике, и в химии. Это — **оператор**.

Оператор — это некоторое *действие* над объектом. Но тут сразу становится как-то запутанно: действие над чем? Над числом? Над вектором? Над функцией? Давай вместе рассмотрим несколько примеров, и всё встанет на свои места.

6.2 Матрица как оператор

Пусть у нас есть квадратная матрица $\mathbf{A} \in \mathbb{C}^{N\times N}$ (я буду далее почти всегда использовать $\mathbb{C}$, так как $\mathbb{R}$ — это всего лишь подмножество $\mathbb{C}$, и всё, что работает для действительных чисел, работает и для комплексных).

Рассмотрим действие $\mathbf{Ab}$, то есть умножение матрицы на вектор. Результат — снова вектор из $\mathbb{C}^N$. Матрица *преобразует* один вектор в другой. Это и есть простейший пример линейного оператора.

6.3 Аналогия с функциональным оператором

А теперь — самое интересное. Аналогом матричного оператора в мире функций является **интегральный оператор**:

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

Здесь $K(x,y)$ — это *ядро* оператора (аналог матрицы $\mathbf{A}$), $g(y)$ — входная функция (аналог вектора $\mathbf{b}$), а результат — новая функция от x.

Интересно, где же такая абстракция встречается в реальной жизни? Оказывается, сплошь и рядом! Например, в квантовой механике оператор Гамильтона $\hat{H}$ действует на волновую функцию Ψ, и это как раз интегрально-дифференциальный оператор. В спектроскопии отклик прибора на входной сигнал часто описывается интегралом свёртки — это тоже оператор.

6.4 Возвращаемся в школу: система уравнений

Но давай начнем с самого простого. Мы еще в школе решали системы уравнений, например:

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

Ты, наверное, думаешь: “Ну и что? Мы же умели это решать подстановкой или методом сложения”. Да, умели. Но давай перепишем это в матричном виде $\mathbf{Ax} = \mathbf{b}$:

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

Ты резонно спросишь меня: “Зачем так усложнять?”. А я отвечу: нет, мы не усложняем — мы *приводим в порядок* наши знания. Мы выделяем *структуру* задачи. И теперь мы можем сказать: если у нас есть **линейный оператор A**, то он действует на вектор **x** и результатом является вектор **b**.

6.5 Напоминание: базовые операции

Прежде чем идти дальше, давай зафиксируем в явном виде те операции, которыми мы будем постоянно пользоваться.

Вектор как матрица-столбец. Вектор $\mathbf{x} \in \mathbb{C}^N$ — это на самом деле матрица размера $N \times 1$. Поэтому все правила матричного умножения автоматически применяются и к векторам.

Транспонирование. Операция $\mathbf{A}^T$ меняет строки и столбцы местами: $(\mathbf{A}^T)_{ij} = A_{ji}$. Для комплексных матриц чаще используют **эрмитово сопряжение** $\mathbf{A}^H = \overline{\mathbf{A}^T}$ (транспонирование + комплексное сопряжение всех элементов).

Скалярное произведение. Для двух векторов $\mathbf{x}, \mathbf{y} \in \mathbb{C}^N$ скалярное произведение определяется как:

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

Для действительных векторов это просто $\mathbf{x}^T \mathbf{y} = \sum x_k y_k$.

Важнейшее свойство: скалярное произведение вектора самого на себя равно **квадрату его евклидовой нормы**:

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

Это мостик между алгеброй и геометрией: длина вектора — это корень из его “скалярного квадрата”.

Умножение матрицы на вектор. Если $\mathbf{A} \in \mathbb{C}^{M \times N}$ и $\mathbf{x} \in \mathbb{C}^N$, то результат $\mathbf{y} = \mathbf{Ax} \in \mathbb{C}^M$ вычисляется по правилу:

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

То есть i-я компонента результата — это скалярное произведение i-й строки матрицы на вектор $\mathbf{x}$.

Умножение матриц. Если $\mathbf{A} \in \mathbb{C}^{M \times K}$ и $\mathbf{B} \in \mathbb{C}^{K \times N}$, то произведение $\mathbf{C} = \mathbf{AB} \in \mathbb{C}^{M \times N}$:

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

Важно: умножение матриц, вообще говоря, **не коммутативно** — $\mathbf{AB} \neq \mathbf{BA}$. Это одно из самых важных отличий от умножения обычных чисел!

Все эти операции — скалярное произведение, умножение матрицы на вектор, умножение матрицы на матрицу — имеют свои полные аналоги в мире функций и интегральных операторов. Только везде, где у нас была сумма $\sum$, появляется интеграл $\int$, а транспонирование заменяется на более сложный оператор сопряжения. Но это — уже дело техники.

6.6 Четыре главные задачи линейной алгебры

На самом деле, пока наши операторы линейные (то есть их можно записать в виде (1) или $\mathbf{Ax}$), весь мир возможных задач крутится вокруг нескольких типовых вопросов. Давай их перечислим.

Задача 1. Посчитать скалярное произведение. Даны два вектора — найти число $\langle \mathbf{x}, \mathbf{y} \rangle$. Это базовая операция, из которой, как из кирпичиков, строится всё остальное.

Задача 2. Посчитать действие оператора. Даны $\mathbf{A}$ и $\mathbf{x}$ — найти $\mathbf{Ax}$. Либо даны $\mathbf{A}$ и $\mathbf{B}$ — найти $\mathbf{AB}$. Это прямое вычисление.

Задача 3. Решить линейную систему уравнений. Даны $\mathbf{A}$ и $\mathbf{b}$ — найти $\mathbf{x}$ такой, что $\mathbf{Ax} = \mathbf{b}$. Это *прямая* задача: известен оператор и результат, надо найти, на что он действовал.

Задача 4. Минимизировать невязку (метод наименьших квадратов). А что, если система $\mathbf{Ax} = \mathbf{b}$ не имеет точного решения? (Например, уравнений больше, чем неизвестных — переопределенная система, что типично для обработки экспериментальных данных). Тогда мы ищем такое $\mathbf{x}$, которое *минимизирует* невязку:

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

Почти всегда в качестве нормы используется $p = 2$ — это и есть знаменитый **метод наименьших квадратов (МНК)**. Он имеет глубокую геометрическую интерпретацию: мы проецируем вектор $\mathbf{b}$ на подпространство, натянутое на столбцы матрицы $\mathbf{A}$.

6.7 А что, если правая часть — ноль?

А теперь — хитрый поворот. Представим, что $\mathbf{b} = \mathbf{0}$. Тогда задача $\min_{\mathbf{x}} \|\mathbf{Ax}\|_2$ имеет тривиальное решение $\mathbf{x} = \mathbf{0}$. Но это же неинтересно!

Давай поставим дополнительное условие: мы ищем **ненулевое** решение, например, с ограничением $\|\mathbf{x}\|_2 = 1$. То есть мы ищем такое направление, в котором матрица $\mathbf{A}$ сильнее всего “сжимает” пространство (или, наоборот, сильнее всего растягивает — это уже вопрос знака).

И вот тут на сцену выходят **собственные значения и собственные векторы**. Мы ищем такие специальные векторы $\mathbf{v}$ и числа λ, при которых действие матрицы сводится просто к растяжению:

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

Геометрически: собственный вектор — это такое направление, которое матрица *не поворачивает*, а только растягивает или сжимает в λ раз.

Задача минимизации $\|\mathbf{Ax}\|_2$ при $\|\mathbf{x}\|_2 = 1$ решается именно через собственные значения $\mathbf{A}^H\mathbf{A}$ или сингулярные значения $\mathbf{A}$: минимум достигается на собственном векторе матрицы $\mathbf{A}^H\mathbf{A}$, соответствующем *минимальному по модулю* собственному значению. А максимум — на векторе, соответствующем максимальному.

6.8 А если неизвестен сам оператор?

Ты спросишь: “А что, если нам оператор неизвестен, а мы знаем только входные функции и результаты?”

Тут ответ немного неоднозначный. Подумай: в матричном виде у нас есть только N входных параметров (вектор $\mathbf{x}$), а мы хотим найти N^2 параметров матрицы $\mathbf{A}$. Природа редко устроена так, что малое число входов хорошо определяет большее число параметров. Это классическая **недоопределенная** задача.

Но и тут решение есть! Если мы скажем, что одна и та же матрица $\mathbf{A}$ действует на несколько разных векторов $\mathbf{x}_1, \ldots, \mathbf{x}_K$ с известными результатами $\mathbf{b}_1, \ldots, \mathbf{b}_K$, то мы можем сформулировать задачу так:

$$\forall k = 1, \ldots, K: \quad \mathbf{A}\mathbf{x}_k = \mathbf{b}_k, \quad \text{причём } \mathbf{A} \text{ — неизвестна.} \tag{2}$$

Если собрать векторы в матрицы $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_K)$ и $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_K)$, то задача переписывается как:

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

А это, внимание, **та же самая задача минимизации**, только теперь мы минимизируем не по $\mathbf{x}$, а по $\mathbf{A}$:

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

(Здесь $\|\cdot\|_F$ — норма Фробениуса для матриц, о которой мы говорили в прошлой главе.)

Транспонируем для удобства: $\mathbf{X}^T\mathbf{A}^T \approx \mathbf{B}^T$. И мы снова получили задачу вида "минимизировать $\|\mathbf{M}\mathbf{z} - \mathbf{c}\|_2$", только для каждого столбца $\mathbf{A}^T$ отдельно. Это и есть классическая **линейная регрессия** — основа хемометрики, QSAR, калибровки приборов и многого другого.

6.9 Сингулярное разложение: мостик в большие миры

И вот тут на сцену выходит одна из самых красивых конструкций во всей линейной алгебре — **сингулярное разложение (SVD, Singular Value Decomposition)**.

Любую матрицу $\mathbf{A} \in \mathbb{C}^{M\times N}$ можно представить в виде:

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

где $\mathbf{U} \in \mathbb{C}^{M\times M}$ и $\mathbf{V} \in \mathbb{C}^{N\times N}$ — унитарные матрицы (их столбцы — ортонормированные векторы), а $\square$ — диагональная матрица с неотрицательными числами $\sigma_1 \geq \sigma_2 \geq \cdots \geq 0$ на диагонали. Эти числа называются **сингулярными значениями**.

Геометрический смысл SVD потрясающ: любая матрица — это просто поворот ($\mathbf{V}^H$), потом растяжение вдоль осей ($\square$), потом еще один поворот ($\mathbf{U}$). Всё! Больше матрицы ничего делать не умеют.

SVD — это универсальный инструмент. Через него решаются:

- задачи наименьших квадратов,
- поиск псевдообратной матрицы,
- сжатие данных и выделение главных компонент (PCA — Principal Component Analysis, основа хемометрики!),
- регуляризация плохо обусловленных задач.

Здесь есть важный момент, унитарные матрицы, имеют одно очень важное свойство, что всегда $\mathbf{V}^H\mathbf{V} = \mathbf{I}$, где $\mathbf{I}$ — единичная матрица, то есть такая, у которой на главной диагонали стоят единички, а все остальное заполнено нулями.

6.10 Мостик в функциональный анализ: теория Фредгольма

А теперь — самое интересное. Помнишь интегральный оператор (2)? Так вот, для него тоже есть аналог SVD! Это называется **сингулярное разложение компактного оператора** или, в более классической формулировке, **теория Фредгольма**.

Суть в следующем: ядро интегрального оператора $K(x, y)$ можно разложить в ряд по "сингулярным функциям" $u_k(x)$ и $v_k(y)$:

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

где σ_k — сингулярные значения (убывающие к нулю), а u_k, v_k — ортонормированные системы функций.

Это в точности аналог SVD для матриц, только в бесконечномерном пространстве! И все идеи, которые мы поняли на матрицах, переносятся сюда почти дословно.

Но не будем перегружать сознание Фредгольмом и функциональным анализом прямо сейчас. Хорошая новость в том, что **большую часть всего, что нам нужно в химии и обработке сигналов, можно сделать на матричном уровне**. А там, где уже не получается (например, в квантовой механике или в строгой теории интегральных уравнений), мы просто вспомним, что “где-то там есть Фредгольм”, и при необходимости спросим детали у ИИ — он с радостью расскажет про теорию Фредгольма, про ядра Гильберта–Шмидта и про спектры компактных операторов.

Главное — понять *структуру*. А структура везде одна и та же: оператор, пространство, действие, разложение по “базисным направлениям”.

7 Обусловленность: когда матрица “хочет” нас обмануть

7.1 Когда решение есть, но его как будто нет

Итак, пусть у нас есть квадратная матрица $\mathbf{A}$ и линейная система $\mathbf{Ax} = \mathbf{b}$. Что мы можем про неё сказать?

Мы помним из школьного курса, что иногда система уравнений бывает такой, что когда мы начинаем подстановку, вдруг у нас либо нет решений, либо мы получаем в конце два и более одинаковых уравнения — и тогда решений получается бесконечно много.

В химии, да и в любой индустриальной задаче, часто исходные данные для матричных уравнений приходят из измерений с приборов. И тут может не повезти сразу тремя способами:

1. Наша система будет вырожденной (нет единственного решения).
2. Наша система будет иметь множество решений.
3. **Самый коварный случай:** мы, решая в машинной арифметике, окажемся *очень близко* к плохому случаю, но этого не заметим. И получим ответ, который выглядит разумно, но на самом деле — полная ерунда.

Когда же это может произойти? Давай разберёмся на живом примере.

7.2 Пример из хроматографии: ловушка большого и малого

Пусть у нас есть две хроматограммы, в которых были сняты два вещества на фоне растворителя. Пик растворителя вышел очень широким и “заполз” на пики искомых веществ, а отклики искомых веществ оказались очень слабенькими.

Фактически, в пике искомых веществ мы имеем:

- Первая хроматограмма: $(a + b_1)$, где a — огромный отклик от растворителя, а b_1 — очень маленькое число от искомого вещества.
- Вторая хроматограмма: $(a + b_2)$, но так как температура чуть-чуть стала выше, то это измерение слегка неточно, скажем, повысилось на какое-то значение ε *относительно пика растворителя*, то есть $(1 + \varepsilon)(a + b_2)$.

Если мы захотим вычесть из первого второе, чтобы посчитать, чему равно $b_1 - b_2$, то вместо этого мы получим:

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

То есть если εa сравнимо с $b_1 - b_2$ по порядку, мы можем получить не просто большую ошибку, но и **противоположный знак**!

Это и есть суть плохой обусловленности: когда в одном числе “живут” одновременно очень большая и очень маленькая компоненты, и мы пытаемся извлечь маленькую через вычитание.

Правило вычитания близких чисел

Никогда не вычитай два близких по величине числа, если нужна относительная точность. Потеря значащих цифр — это не баг компьютера, это фундаментальное свойство арифметики.

7.3 Как SVD раскрывает механизм катастрофы

То же самое происходит и при решении систем уравнений. И у нас есть классный способ *заранее* оценить, что же нам делать.

Пусть у нас есть система $\mathbf{Ax} = \mathbf{b}$, где мы ищем $\mathbf{x}$, а всё остальное дано. Вспомним сингулярное разложение $\mathbf{A} = \mathbf{U}\square\mathbf{V}^H$. Тогда решение можно переписать в несколько стадий:

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

Здесь мы используем свойства того, что решать систему с унитарными матрицами $\mathbf{U}, \mathbf{V}$ очень просто — достаточно домножить на $\mathbf{U}^H$ или $\mathbf{V}^H$, так как $\mathbf{U}^H\mathbf{U} = \mathbf{I}$. А решать систему с диагональной матрицей — это вообще тривиально.

Давай нарисуем, как это выглядит для матрицы 3×3:

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

Видите? Каждое уравнение — это просто деление одной компоненты на соответствующее сингулярное число.

И вот тут — самое важное. Представим, что мы посчитали сингулярные числа исходной матрицы и заметили, что самое большое сингулярное число σ_1 очень-очень сильно больше самого маленького σ_N.

Тогда на шаге $\mathbf{t}_2 = \square^{-1}\mathbf{t}_1$ происходит следующее:

- В векторе $\mathbf{t}_1$ ошибка измерений распределена более-менее равномерно по всем компонентам.
- После умножения на $\square^{-1}$ компонента, соответствующая *маленькому* σ_N, возрастает в σ_1/σ_N раз!
- А если σ_N вообще равен нулю? Тогда мы делим на ноль — и решение не существует.

Именно из-за этого люди решили обозначать отношение σ_1/σ_N как **число обусловленности** матрицы:

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

Фактически, это — характеристика того, *насколько мы можем испортить решение такой матрицей*. Если $\text{cond}(\mathbf{A}) \approx 10^k$, то при решении системы мы теряем примерно k значащих цифр точности. Для double precision (16 значащих цифр) это означает, что при $\text{cond}(\mathbf{A}) \approx 10^{16}$ ответ будет полностью состоять из шума.

Важное замечание

Число обусловленности — это характеристика *матрицы*, а не алгоритма. Никакой, даже самый гениальный алгоритм не может решить плохо обусловленную систему точнее, чем позволяет $\mathrm{cond}(\mathbf{A})$. Это фундаментальное ограничение задачи, а не наших вычислительных методов.

При этом обусловленность *не влияет* на вычисление собственных векторов симметричных/эрмитовых матриц или на поиск сингулярных векторов — эти задачи, как правило, гораздо устойчивее.

7.4 Где это всё встречается в химии и за её пределами?

Что-то у нас было много сухой теории, и надо бы вспомнить, где же это всё в химии применяется.

Квантовая химия: уравнение Шрёдингера. Да, волновое уравнение Шрёдингера и все его приближения — уравнения Хартри–Фока, Density Functional Theory (DFT) — всё это задачи на собственные значения: поиск таких векторов $\mathbf{x}$ и значений λ, которые удовлетворяют уравнению $\mathbf{Ax} = \lambda\mathbf{x}$. Матрицы здесь — матрицы Фока или матрицы Гамильтониана — часто имеют размерность тысяч и десятков тысяч, и их обусловленность напрямую определяет, сможем ли мы вообще получить физически осмысленный ответ.

Масс-спектрометрия и спектроскопия. Поиск снятого спектра по базе данных откликов в масс-спектрометре — это алгоритмы, которые базируются на решении линейных систем. Если у вас есть смесь веществ, и вы хотите понять, из чего она состоит, вы решаете систему $\mathbf{Ax} = \mathbf{b}$, где $\mathbf{A}$ — матрица эталонных спектров, $\mathbf{b}$ — измеренный спектр смеси, а $\mathbf{x}$ — концентрации компонентов. И если спектры компонентов похожи — матрица плохо обусловлена, и концентрации будут “прыгать”.

Крио-электронная микроскопия. Реконструкция 3D-структуры белков из тысяч 2D-проекций — это гигантская разреженная система уравнений. Обусловленность там — ключевой фактор, определяющий разрешение итоговой структуры.

ЯМР-спектроскопия. Обращение свёртки (деконволюция) — это классическая плохо обусловленная задача. Именно поэтому в ЯМР используют регуляризацию Тихонова и преобразование Фурье — они превращают плохо обусловленную задачу в диагональную.

Калибровка аналитических приборов. PLS-регрессия (Partial Least Squares) — это, по сути, SVD с регуляризацией. Обусловленность матрицы спектров напрямую влияет на точность предсказания концентраций.

Поиск в интернете (PageRank). Даже очень-очень старый Google-поиск был устроен на том, что все заранее найденные документы классифицировались в огромную матрицу, и для неё считалось сингулярное разложение (или, что эквивалентно, ищется главный собственный вектор матрицы переходов), что помогало моментально найти искомый документ по совпадающей комбинации слов. В современном ИИ SVD используется очень-очень часто — от сжатия весов нейросетей до понижения размерности в рекомендательных системах.

Обработка сигналов и изображений. Деконволюция размытых изображений в микроскопии, подавление шума в ЭКГ/ЭЭГ, сжатие аудио (MP3) и видео — всё это задачи, в которых обусловленность играет ключевую роль.

Когда-то эти три задачи — решение линейных систем, поиск собственных значений и SVD — были камнем преткновения при решении больших индустриальных задач. Даже были фирмы, которые *только* разрабатывали такие солверы — и ничегошеньки больше. Правда, это было ещё в 90-ые годы прошлого века.

7.5 Сколько это стоит? Вычислительная сложность

Грубо говоря, если у нас есть квадратная матрица $N \times N$ и мы решаем либо линейную систему с ней, либо ищем её собственные или сингулярные числа и вектора, то нам надо потратить примерно $\mathcal{O}(N^3)$ арифметических операций, если мы никак специально не используем структуру или свойства этой матрицы.

Это — “цена” полной задачи. Но если матрица обладает специальными свойствами, цену можно сильно снизить — иногда до $\mathcal{O}(N)$ или $\mathcal{O}(N \log N)$. И об этом — ниже.

7.6 Связь сингулярного разложения и собственных значений

А теперь — один из самых красивых фактов во всей линейной алгебре. Пусть у нас есть сингулярное разложение:

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

Давайте посмотрим, что получится, если мы умножим $\mathbf{A}$ на её эрмитово сопряжённую $\mathbf{A}^H$:

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

Здесь мы использовали, что $\mathbf{V}^H\mathbf{V} = \mathbf{I}$ (унитарность $\mathbf{V}$), и $\square^T = \square$ (диагональная матрица).

Аналогично:

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

Что это означает?

- Сингулярные векторы матрицы $\mathbf{A}$ — это *в точности* собственные векторы матриц $\mathbf{A}\mathbf{A}^H$ и $\mathbf{A}^H\mathbf{A}$.
- Сингулярные числа σ_k матрицы $\mathbf{A}$ — это квадратные корни из собственных чисел λ_k матриц $\mathbf{A}\mathbf{A}^H$ или $\mathbf{A}^H\mathbf{A}$:
$$\sigma_k = \sqrt{\lambda_k}$$

Это — глубокая связь между двумя, казалось бы, разными задачами: сингулярным разложением и поиском собственных значений.

Осторожно: квадрат обусловленности!

Но есть одна коварная деталь. Число обусловленности матриц $\mathbf{A}\mathbf{A}^H$ и $\mathbf{A}^H\mathbf{A}$ — это **квадрат** числа обусловленности исходной матрицы:

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

Если $\text{cond}(\mathbf{A}) = 10^8$, то $\text{cond}(\mathbf{A}^H\mathbf{A}) = 10^{16}$ — и мы потеряем все 16 значащих цифр точности!

Поэтому переходить от задачи сингулярного разложения к задаче собственных значений для $\mathbf{A}^H\mathbf{A}$ надо **крайне осторожно**. В современных библиотеках (LAPACK) SVD считается напрямую, без явного построения $\mathbf{A}^H\mathbf{A}$ — именно чтобы избежать этой катастрофы.

8 Классификация матриц: кто есть кто

Теперь, когда понятно, что практически всё базируется на этих “простых” математических задачах, давай немного систематизируем наши знания о таких решениях. Ведь всегда у нас есть матрица, и от свойств этой матрицы зависит очень многое.

8.1 По форме

Тип	Размер	Описание
Квадратная	$N \times N$	Одинаковое число строк и столбцов. Только для таких матриц определён определитель, собственные значения и обратная матрица.
“Стоячая” (высокая)	$M \times N,\ M > N$	Уравнений больше, чем неизвестных. Обычно не имеет точного решения — решаем через МНК.
“Лежачая” (широкая)	$M \times N,\ M < N$	Неизвестных больше, чем уравнений. Решений бесконечно много — ищем минимальное по норме.

8.2 По симметрии и структуре элементов

Тип	Определение и свойства	Сложность
Симметричная действительная	$\mathbf{A} = \mathbf{A}^T$, элементы $\in \mathbb{R}$. Собственные значения **действительные**, собственные векторы — ортогональны.	$\mathcal{O}(N^3/3)$
Эрмитова комплексная	$\mathbf{A} = \mathbf{A}^H$, элементы $\in \mathbb{C}$. Собственные значения **действительные**, собственные векторы — ортонормированы.	$\mathcal{O}(4N^3/3)$
Кососимметричная	$\mathbf{A} = -\mathbf{A}^T$. Собственные значения — чисто мнимые или нули.	$\mathcal{O}(N^3/3)$
Ортогональная / Унитарная	$\mathbf{A}^T\mathbf{A} = \mathbf{I}$ / $\mathbf{A}^H\mathbf{A} = \mathbf{I}$. Сохраняет длины и углы. Собственные значения по модулю равны 1.	$\mathcal{O}(N^2)$ для умножения
Единичная	$\mathbf{A} = \mathbf{I}$. Диагональ из единиц, остальное — нули.	$\mathcal{O}(N)$
Диагональная	$A_{ij} = 0$ при $i \neq j$. Хранится как вектор из N чисел.	$\mathcal{O}(N)$

8.3 По положительной определённости

Это — подкласс симметричных/эрмитовых матриц, и он критически важен для оптимизации и статистики.

Тип	Определение и свойства
Положительно определённая ($\mathbf{A} \succ 0$)	$\forall \mathbf{x} \neq 0:\ \mathbf{x}^H\mathbf{A}\mathbf{x} > 0$. Все собственные значения строго положительны. Обусловленность — отношение максимального и минимального собственных значений. Решается методом Холецкого за $\mathcal{O}(N^3/3)$.
Неотрицательно определённая ($\mathbf{A} \succeq 0$)	$\forall \mathbf{x}:\ \mathbf{x}^H\mathbf{A}\mathbf{x} \geq 0$. Все собственные значения ≥ 0. Может быть вырожденной.

8.4 По структуре расположения ненулевых элементов

Тип	Определение и свойства	Сложность
Ленточная (banded)	Ненулевые элементы только вблизи главной диагонали: $A_{ij} = 0$ при $\|i-j\| > k$. Хранится как $N \times (2k+1)$.	$\mathcal{O}(Nk^2)$
Теплицева	A_{ij} зависит только от $i-j$. Каждая диагональ — константа. Возникает в задачах со стационарными процессами.	$\mathcal{O}(N^2)$, через БПФ — $\mathcal{O}(N \log N)$
Циркулянтная	Теплицева + периодичность: каждая строка — циклический сдвиг предыдущей. Диагонализуется преобразованием Фурье.	$\mathcal{O}(N \log N)$ через БПФ
Блочная	Состоит из блоков, каждый из которых — матрица. Позволяет использовать рекурсивные алгоритмы.	Зависит от структуры блоков
Разреженная (sparse)	Большинство элементов — нули. Хранится в специальных форматах (CSR, CSC). Основа больших вычислений.	$\mathcal{O}(\text{nnz})$, где nnz — число ненулевых

8.5 Вырожденность

Вырожденная (сингулярная) матрица — это матрица, определитель которой равен нулю, или, что эквивалентно, у которой хотя бы одно сингулярное значение равно нулю. Для такой матрицы:

- не существует обратной матрицы,
- система $\mathbf{Ax} = \mathbf{b}$ либо не имеет решений, либо имеет бесконечно много,
- $\text{rank}(\mathbf{A}) < N$.

На практике матрицы почти никогда не бывают *точно* вырожденными — но они могут быть *почти* вырожденными, то есть с очень маленьким $\sigma_{\min}$. И вот это — гораздо более коварный случай, потому что формально решение существует, но оно полностью определяется шумом в данных.

8.6 Дополнительные важные типы матриц

Тип	Определение и свойства	Сложность
Верхнетреугольная	$A_{ij} = 0$ при $i > j$. Все ненулевые элементы выше или на главной диагонали. Решение системы — обратная подстановка.	$\mathcal{O}(N^2)$
Нижнетреугольная	$A_{ij} = 0$ при $i < j$. Все ненулевые элементы ниже или на главной диагонали. Решение системы — прямая подстановка.	$\mathcal{O}(N^2)$
Матрица перестановок	В каждой строке и каждом столбце ровно одна единица, остальное — нули. Умножение на такую матрицу — это перестановка строк/столбцов. $\mathbf{P}^T = \mathbf{P}^{-1}$.	$\mathcal{O}(N)$

Матрицы перестановок — это не просто абстракция. Они критически важны для численной устойчивости алгоритмов (например, в LU-разложении с выбором главного элемента), и мы ещё встретимся с ними.

8.7 Сводная таблица: что и как решать

Задача	Общий случай	Симметричная	Разреженная
Линейная система $\mathbf{Ax} = \mathbf{b}$	LU, $\mathcal{O}(N^3)$	Холецкий, $\mathcal{O}(N^3/3)$	Итерационные, $\mathcal{O}(\text{nnz} \cdot k)$
МНК $\min \|\mathbf{Ax} - \mathbf{b}\|_2$	QR, $\mathcal{O}(MN^2)$	—	LSQR, итерационные
Собственные значения	$\mathcal{O}(N^3)$	$\mathcal{O}(N^3/3)$, все действительные	Lanczos, $\mathcal{O}(\text{nnz} \cdot k)$
SVD	$\mathcal{O}(MN^2)$	Через собственные $\mathbf{A}^T\mathbf{A}$	Усечённое SVD, итерационные

Здесь k — число итераций, которое зависит от желаемой точности и обусловленности.

Главный вывод

Прежде чем решать задачу, *посмотри на матрицу*. Её свойства — симметрия, разреженность, ленточная структура — могут снизить сложность с кубической до линейной. А её обусловленность скажет тебе, стоит ли вообще доверять полученному ответу.

9 Как решают задачи линейной алгебры: от теории к практике

9.1 Одного решения на все случаи — нет

Так как же решают эти задачи линейной алгебры? Наверное, есть одно или может быть пара самых простых и надёжных решений?

К сожалению, даже сейчас одного решения на все случаи жизни — нет и не предвидится. Автор этого скрипта участвовал в своё время в написании более сотни различных таких алгоритмов. Да, таких алгоритмов — очень много, и надо хотя бы коротко, пусть даже ни разу не имплементировав это, понимать, когда и что можно использовать.

9.2 Классификация методов решения

Методы решения можно классифицировать примерно так:

9.2.1 Прямые методы разложения

LU-разложение: $\mathbf{A} = \mathbf{LU}$, где $\mathbf{L}$ — нижнетреугольная, $\mathbf{U}$ — верхнетреугольная матрица. С такой факторизацией решение системы $\mathbf{Ax} = \mathbf{b}$ сводится к двум простым подстановкам:

1. Решаем $\mathbf{Ly} = \mathbf{b}$ (прямая подстановка, $\mathcal{O}(N^2)$)
2. Решаем $\mathbf{Ux} = \mathbf{y}$ (обратная подстановка, $\mathcal{O}(N^2)$)

Но в общем виде $\text{cond}(\mathbf{L}) \cdot \text{cond}(\mathbf{U}) \geq \text{cond}(\mathbf{A})$, а если не делать перестановок, то произведение обусловленностей может стать существенно больше $\text{cond}(\mathbf{A})$. То есть мы можем испортить нашу исходную задачу!

Поэтому на практике почти всегда используют **LUP-разложение**: $\mathbf{PA} = \mathbf{LU}$, где $\mathbf{P}$ — матрица перестановок. Перестановки выбираются так, чтобы на диагонали $\mathbf{U}$ стояли максимально возможные элементы (выбор главного элемента — pivoting). Это гарантирует численную устойчивость.

Для специальных матриц:

- Если $\mathbf{A}$ — ленточная, то $\mathbf{L}$ и $\mathbf{U}$ не выходят за пределы ленты. Сложность $\mathcal{O}(Nk^2)$, где k — ширина ленты.
- Если $\mathbf{A}$ — разреженная, то $\mathbf{L}$ и $\mathbf{U}$ могут содержать существенно больше ненулевых элементов (fill-in), и это может быть большой проблемой.
- Для симметричной матрицы — $\mathbf{A} = \mathbf{L}\mathbf{D}\mathbf{L}^H$ или $\mathbf{P}\mathbf{A}\mathbf{P}^H = \mathbf{L}\mathbf{D}\mathbf{L}^H$.
- Для положительно определённой — $\mathbf{A} = \mathbf{L}\mathbf{L}^H$, так называемый **метод Холецкого**. Но проблема роста числа обусловленности есть даже для положительно определённых матриц.

QR-разложение: $\mathbf{A} = \mathbf{Q}\mathbf{R}$, где $\mathbf{Q}$ — унитарная ($\mathbf{Q}^H\mathbf{Q} = \mathbf{I}$), $\mathbf{R}$ — верхнетреугольная.

Этот метод **гарантирует не увеличение числа обусловленности** решения, так как унитарные преобразования сохраняют длины векторов. Хорош для плотных матриц без структуры.

Но если исходная матрица — разреженная, то $\mathbf{Q}$ практически всегда будет плотной (только для очень специальных случаев и методов), что сильно ограничивает применимость этого метода для решения огромных задач, когда сама матрица помещается в память, а её плотная версия N^2 — не помещается.

Разложение Шура: $\mathbf{A} = \mathbf{Q}\mathbf{G}\mathbf{Q}^H$, где $\mathbf{G}$ — верхнетреугольная (для действительных матриц — верхняя квазитреугольная с блоками 2×2 на диагонали).

Применяется для поиска собственных значений и собственных векторов, которые находят, решая задачу на собственные значения для уже треугольной матрицы $\mathbf{G}$ (а для треугольной матрицы собственные значения — это просто диагональные элементы!).

9.2.2 Итерационные методы

Это методы решения линейной системы или задачи на собственные значения, когда мы можем эффективно умножить на матрицу из-за её структуры, и строим решение итерационно.

Тут есть много методов, но важно заметить: **сходимость итерационных методов зависит от числа обусловленности**. Чем оно больше, тем медленнее они сходятся. (Есть исключения: если в матрице есть только несколько сильно больших и сильно маленьких сингулярных чисел, тогда сходимость может быть существенно быстрее.)

Все эти методы можно также разделить на подклассы:

1. **Специальные методы для положительно определённых матриц** с маленькими затратами по дополнительной памяти и разумным числом итераций. Самый известный — **метод сопряжённых градиентов (Conjugate Gradients, CG)**.
2. **Приведение несимметричной задачи к симметричной:** вместо решения $\mathbf{A}\mathbf{x} = \mathbf{b}$ решать $\mathbf{A}^H\mathbf{A}\mathbf{x} = \mathbf{A}^H\mathbf{b}$. Если число обусловленности $\text{cond}(\mathbf{A}^H\mathbf{A})$ не настолько огромное по сравнению с $\text{cond}(\mathbf{A})$, то это разумно. Но помните: $\text{cond}(\mathbf{A}^H\mathbf{A}) = \text{cond}(\mathbf{A})^2$, так что это палка о двух концах.
3. **Предобуславливатели (preconditioners):** такие специальные матрицы $\mathbf{P} \approx \mathbf{A}^{-1}$, что вместо решения $\mathbf{A}\mathbf{x} = \mathbf{b}$ мы решаем $\mathbf{P}\mathbf{A}\mathbf{x} = \mathbf{P}\mathbf{b}$, предполагая, что $\text{cond}(\mathbf{P}\mathbf{A}) \ll \text{cond}(\mathbf{A})$, что сильно ускоряет решение такими итерационными методами.

   Кстати, часто для больших разреженных матриц предобуславливатели строят как **неполное LU-разложение**: то есть строят для исходной матрицы $\mathbf{A} \approx \mathbf{L}\mathbf{U}$, но стараются “выбросить” новые ненулевые элементы при построении $\mathbf{L}\mathbf{U}$. В таком случае применять эту матрицу как что-то похожее на исходную ещё можно, поэтому, решая с ней, мы предобуславливаем исходную задачу и повышаем сходимость.

9.3 Два мира: плотные и разреженные задачи

На самом деле, большинство алгоритмов решения можно разделить на два класса:

1. Когда мы готовы потратить $\mathcal{O}(N^3)$ арифметических операций и у нас есть $\mathcal{O}(N^2)$ памяти.
2. Когда нам надо ужаться, а матрица настолько огромная и имеет какую-то специальную структуру, что мы на неё можем умножать, но взять её в полном виде — очень сложно.

В первом случае — это **методы разложения** (LU, QR, Холецкий). Во втором — это **итерационные методы**.

Причём из-за специфичной структуры устройства современных процессоров и памяти, итерационные методы имеют немного худшую производительность, чем методы полного разложения. Поэтому применять итерационные методы для очень маленьких матриц практически всегда не бывает оправдано.

9.4 Примеры из квантовой химии: DFT

Пример 1: Небольшая система. Небольшая система на несколько электронов методом DFT, и размер гамильтониана — около 1000×1000. Мы вписываемся в память (это всего ~ 8 МБ для double precision). Мы можем найти все собственные значения и набор тех собственных векторов, которые нас интересуют, и для этого очевидный выбор — прямой метод (например, разложение Шура или QR-алгоритм).

Пример 2: Большая химическая структура. Довольно большая химическая структура методом DFT, в которой есть около тысячи электронных пар на внешних орбиталях. Для этого у нас построился гамильтониан уравнения Шрёдингера, и нам надо найти на каждую электронную пару по своему собственному вектору. А гамильтониан у нас задан так, что мы взяли под сотню базисных функций на каждую орбиталь, то есть размерность этого гамильтониана — около $100\,000 \times 100\,000$.

Это вроде бы и не очень много, но только сама матрица такого гамильтониана уже занимает в оперативной памяти примерно **80 ГБ**, и само решение будет ещё почти гигабайт занимать. Тут гораздо очевиднее итерационное решение (например, метод Ланцоша или Davidson), которое находит только несколько нужных собственных векторов, не работая с полной матрицей.

9.5 Не изобретайте велосипед: библиотеки

Сейчас во многих языках программирования есть хорошо и годами вылизанные библиотеки по решению таких систем уравнений, поиску сингулярных чисел и векторов и собственных векторов и чисел. Достаточно поискать в документации на **NumPy** для Python, и в **LAPACK/BLAS** для C/C++ — и всё будет сразу понятно.

Более того, код очень хорошо оптимизирован под современные компьютеры и довольно сложный. Например, современная версия SVD в LAPACK содержит около **полумиллиона строк кода**, и повторить его или даже сделать лучше — это реально почти невыполнимая задача.

Практический совет

Никогда не пишите свои солверы для линейной алгебры, если только это не учебная задача. Используйте:

- **Python:** NumPy (`numpy.linalg`), SciPy (`scipy.linalg`)
- **C/C++:** LAPACK, BLAS, Eigen
- **Fortran:** LAPACK (это его родная стихия)
- **MATLAB:** встроенные функции (`eig`, `svd`, `lu`, `qr`)

Эти библиотеки прошли десятилетия оптимизации и тестирования. Они знают про кэш-процессора, про SIMD-инструкции, про многопоточность — всё то, что вы не сможете повторить вручную.

10 Нелинейные операторы: когда мир перестает быть прямым

10.1 Что такое нелинейный оператор?

До сих пор мы говорили о линейных операторах — тех, которые можно записать как $\mathbf{Ax}$ или $\int K(x,y)g(y)\mathrm{d}y$. Но реальный мир — нелинеен.

Нелинейный оператор — это такое отображение $\mathcal{F}$, которое действует на функцию или вектор $\mathbf{x}$, но которое *не* обладает свойствами аддитивности и однородности:

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

Грубо говоря, дефинируем теперь, что у нас есть какая-то функция $f(\mathbf{x})$, которая зависит от одного или нескольких параметров. И мы хотим либо:

- Решить уравнение $f(\mathbf{x}) = 0$ (систему нелинейных уравнений),
- Найти минимум $\min_{\mathbf{x}} f(\mathbf{x})$ (задача оптимизации).

10.2 Пример из хроматографии: модель тарелок

Классический пример — система уравнений баланса и констант диссоциации на каждой хроматографической тарелке.

Мы хотим моделировать хроматограф, вернее его колонку. В ней идет разделение, и подвижная фаза, перемещаясь вдоль колонки, на каждом шаге фактически создает такое равновесие.

Вы скажете — ой, да тут всего какие-то 4–5 уравнений, и столько же неизвестных! Да, согласен, немного. Но:

1. Тарелок-то много (пусть тысяча) — это раз.
2. Мы разбиваем время прохождения вещества по колонке на множество шагов по времени — это два.

То есть на каждом шаге по времени мы имеем тысячу раз одну и ту же систему уравнений с разными параметрами входных концентраций. И таких шагов по времени будет существенно больше, чем число тарелок, ведь вещество-то всё-таки колонкой удерживается.

То есть нам надо решить эдак под **десять миллионов раз** систему уравнений из всего-то 5 уравнений с 5 неизвестными. Но стоит хоть один раз не решить — мы не сможем получить ни одно дальнейшее решение.

К сожалению, такая система — не является линейной. Уравнения баланса — линейны, но вот уравнения связи констант диссоциации содержат произведения и отношения неизвестных друг на друга. И, на удивление, такая система иногда может действительно очень плохо решаться.

10.3 Классификация нелинейных задач

Прежде чем говорить о методах, давайте классифицируем сами задачи:

Свойство	Описание
Гладкость:	
Гладкие	Функция имеет непрерывные производные (как минимум первую, а желательно и вторую). Пример: $f(x) = x^2 + \sin x$.
Почти гладкие	Функция гладкая почти везде, но есть точки излома или разрывы производных. Пример: $f(x) = \|x\|$.
Разрывные	Функция имеет разрывы или очень шумная. Пример: экспериментальные данные с артефактами.
Число минимумов:	
Унимодальные	Есть только один глобальный минимум (или максимум). Пример: выпуклые функции.
Мультимодальные	Есть несколько локальных минимумов, и задача — найти глобальный. Пример: потенциалы сложных молекул.

10.4 Методы решения нелинейных задач

Теперь давайте систематизируем методы решения. Каждый метод что-то требует, от чего-то зависит, и у каждого есть свои сильные и слабые стороны.

10.4.1 Методы Монте-Карло

Идея: Случайный поиск. Генерируем случайные точки в пространстве параметров, вычисляем в них функцию, выбираем лучшую.

Требования: Ничего не требует. Может работать даже с разрывными функциями.

Плюсы:

- Не застревает в локальных минимумах (при достаточном числе итераций),
- Прост в реализации,
- Параллелится идеально.

Минусы:

- Очень медленная сходимость: ошибка убывает как $\mathcal{O}(1/\sqrt{N})$, где N — число итераций,
- Не использует информацию о структуре функции.

Когда использовать: Когда функция очень шумная, разрывная, или когда нужно просто найти “хоть какое-то” решение для инициализации других методов.

10.4.2 Градиентные методы (Gradient Descent)

Идея: Двигаться в направлении, противоположном градиенту:

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

где α_k — шаг обучения (learning rate).

Требования: Функция должна быть дифференцируемой (гладкой).

Плюсы:

- Прост в реализации,
- Гарантированная сходимость к локальному минимуму (при правильном выборе α),

- Хорошо работает в высоких размерностях.

Минусы:

- Застревает в локальных минимумах,
- Может очень медленно сходиться в “оврагах” (когда собственные значения гессиана сильно различаются),
- Требует выбора шага α (слишком большой — расходится, слишком маленький — медленно).

10.4.3 Метод сопряженных направлений (Conjugate Gradient для оптимизации)

Идея: Строить направления поиска так, чтобы они были “сопряжены” относительно гессиана функции. Это позволяет найти минимум квадратичной функции за N шагов (где N — размерность).

Важное замечание: Метод сопряженных градиентов для решения линейных систем итерационным методом *только называется* одинаково с методом сопряженных градиентов для нелинейной минимизации. На самом деле, это один и тот же алгоритм, просто примененный к разным задачам:

- Для линейных систем $\mathbf{Ax} = \mathbf{b}$ — это минимизация квадратичной функции $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{Ax} - \mathbf{b}^T\mathbf{x}$,
- Для нелинейной оптимизации — это обобщение той же идеи на произвольные функции.

Требования: Гладкая функция, желательно с непрерывным градиентом.
Плюсы:

- Сходится быстрее обычного градиентного спуска,
- Не требует хранения гессиана (в отличие от метода Ньютона),
- Память — $\mathcal{O}(N)$.

Минусы:

- Застревает в локальных минимумах,
- Для неквадратичных функций требуется “перезапуск” (reset) направлений.

10.4.4 Методы Ньютона

Идея: Использовать не только градиент, но и вторые производные (гессиан). Разлагаем функцию в ряд Тейлора до второго порядка:

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

и находим минимум этой квадратичной аппроксимации. Получаем итерацию:

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

где $\mathbf{H}$ — матрица вторых производных (гессиан).

Требования: Функция должна быть дважды дифференцируемой. Гессиан должен быть вычислим.
Плюсы:

- **Квадратичная сходимость:** если начали близко к решению, то число верных цифр удваивается на каждой итерации,
- Не чувствителен к “оврагам” (в отличие от градиентного спуска).

Минусы:

- Требует вычисления гессиана — $\mathcal{O}(N^2)$ элементов,
- Требует решения линейной системы с гессианом — $\mathcal{O}(N^3)$ операций,
- Может расходиться, если начали далеко от решения,
- Гессиан может быть вырожденным или не положительно определенным.

10.4.5 Метод Ньютона–Рафсона и блочный Ньютон для квантовой механики

Метод Ньютона–Рафсона — это вариант метода Ньютона для решения систем нелинейных уравнений $\mathbf{F}(\mathbf{x}) = \mathbf{0}$:

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

где **J** — матрица Якоби (матрица первых производных $\partial F_i/\partial x_j$).

Блочный Ньютон для квантовой механики: В задачах квантовой химии (например, в методе Хартри–Фока или DFT) часто возникает система уравнений, которую можно разбить на блоки:

- Уравнения для орбиталей (коэффициенты разложения),
- Уравнения для плотности,
- Уравнения для энергии.

Блочный метод Ньютона учитывает эту структуру: гессиан представляется в блочном виде, и каждый блок обрабатывается отдельно. Это позволяет:

- Эффективно использовать структуру задачи,
- Параллелить вычисления,
- Применять разные методы к разным блокам (например, Ньютон для одного блока, градиентный спуск для другого).

10.4.6 Методы BFGS и L-BFGS

BFGS (Broyden–Fletcher–Goldfarb–Shanno) — это квазиньютоновский метод. Идея: не вычислять гессиан явно, а приближать его на каждой итерации, используя информацию о изменении градиента.

Требования: Гладкая функция с непрерывным градиентом.

Плюсы:

- Сходимость почти как у Ньютона, но без вычисления гессиана,
- Гарантированная положительная определенность приближения гессиана,
- Хорошо работает на практике — один из самых популярных методов.

Минусы:

- Требует хранения матрицы $N \times N$ (приближение гессиана) — $\mathcal{O}(N^2)$ памяти.

L-BFGS (Limited-memory BFGS) — модификация BFGS для больших задач. Идея: не хранить полную матрицу, а хранить только последние m пар векторов $(\mathbf{s}_k, \mathbf{y}_k)$, где:

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

Плюсы:

- Память — $\mathcal{O}(mN)$, где $m \sim 5$–20 (не зависит от N^2!),
- Идеален для больших задач ($N > 1000$),
- Очень популярен в машинном обучении и квантовой химии.

Минусы:

- Чуть медленнее сходится, чем полный BFGS,
- Требует настройки параметра m.

10.4.7 Симплекс-методы (Nelder–Mead)

Идея: Строим симплекс (в N-мерном пространстве это $N + 1$ точка), вычисляем функцию в вершинах, и итеративно отражаем, сжимаем или расширяем симплекс, двигаясь к минимуму.

Требования: Никаких! Функция может быть негладкой, шумной, даже разрывной.

Плюсы:

- Не требует градиентов,
- Прост в реализации,
- Хорошо работает для малых размерностей ($N < 10$).

Минусы:

- Очень медленный для больших N,
- Может застревать,
- Нет теоретических гарантий сходимости.

Когда использовать: Когда функция очень шумная или когда N маленькое и лень выводить градиенты.

10.4.8 Simulated Annealing (Имитация отжига)

Идея: Вдохновлен физическим процессом отжига металлов. Начинаем с высокой "температуры" T, которая позволяет алгоритму "перепрыгивать" через локальные минимумы. Постепенно снижаем T, и алгоритм "застывает" в глобальном минимуме.

На каждом шаге:

1. Предлагаем случайное изменение $\mathbf{x} \to \mathbf{x}'$,
2. Вычисляем $\Delta f = f(\mathbf{x}') - f(\mathbf{x})$,
3. Если $\Delta f < 0$ — принимаем изменение,
4. Если $\Delta f > 0$ — принимаем с вероятностью $P = \exp(-\Delta f / T)$.

Требования: Никаких.

Плюсы:

- Может найти глобальный минимум (при правильном “расписании отжига”),
- Не застревает в локальных минимумах на ранних этапах,
- Прост в реализации.

Минусы:

- Очень медленный,
- Требует настройки расписания $T(t)$,
- Нет гарантий сходимости за разумное время.

Когда использовать: Когда задача мультимодальная (много локальных минимумов) и нужно найти глобальный.

10.5 Вычисление градиентов

Все градиентные методы требуют вычисления $\nabla f(\mathbf{x})$. Как это делать?

10.5.1 Конечные разности

Идея: Приближаем производную:

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

где $\mathbf{e}_j$ — j-й базисный вектор.

Проблемы:

1. **Дорого:** Для вычисления градиента в N-мерном пространстве нужно $N + 1$ вычислений функции (или $2N$ для центральной разности).
2. **Неустойчиво:** Если h слишком большое — большая ошибка аппроксимации. Если h слишком маленькое — потеря точности из-за вычитания близких чисел. Оптимальное $h \sim \sqrt{\epsilon_{\text{mach}}}$, где ϵ_{mach} — машинная точность.
3. **Шум:** Если функция вычисляется с шумом (например, экспериментальные данные), то конечные разности усиливают шум.

10.5.2 Автоматическое дифференцирование (AD): Метод Бауэра–Штрассена: аналитическое вычисление градиента

А теперь — магия! Оказывается, можно вычислить градиент функции *аналитически*, используя автоматическое дифференцирование, и сделать это всего в **несколько раз** дороже, чем вычисление самой функции!

Идея: Любая функция $f(\mathbf{x})$ — это композиция элементарных операций ($+$, $-$, $\times$, $\div$, sin, cos, exp, log, и т.д.). Мы можем:

1. Представить вычисление функции как **вычислительный граф**,
2. Применить **цепное правило** (chain rule) в обратном порядке (reverse mode).

Результат: Градиент вычисляется за $\mathcal{O}(1)\times$ стоимость функции (на практике — в 3–5 раз дороже, но не в N раз!).

Как это работает на пальцах:

1. Прямой проход: вычисляем $f(\mathbf{x})$, сохраняя все промежуточные результаты.
2. Обратный проход: идем по графу в обратном направлении, применяя цепное правило:

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

Где это используется:

- **PyTorch, TensorFlow:** Основа всех современных нейросетей — это именно reverse-mode automatic differentiation.
- **Квантовая химия:** Программы типа PySCF, Psi4 используют AD для вычисления градиентов энергии по координатам ядер.
- **Оптимизация:** Все современные библиотеки (SciPy, JAX) используют AD.

Главный вывод

Никогда не вычисляйте градиенты конечными разностями, если можно использовать автоматическое дифференцирование! Это быстрее, точнее и устойчивее.

10.6 Живой пример: силовые поля и конформеры

Ещё один живой пример из химии — моделирование молекул. Для моделирования небольших молекул часто используют такое представление, когда общая энергия записана как сумма простых взаимодействий:

1. **Ван-дер-Ваальсовы взаимодействия** между несвязанными атомами (потенциал Леннард-Джонса):

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

2. **Гармонические связи** между связанными атомами:

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

3. **Угловые взаимодействия** между тремя последовательными атомами:

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

4. **Торсионные (диэдральные) взаимодействия** между четырьмя последовательными атомами:

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

Их будет не много, но около квадрата от числа атомов (для ван-дер-ваальса). Каждая такая функция — это какая-то несложная формула: иногда с синусами, иногда с экспонентами, иногда со степенями. А суммарно у молекулы — $3N$ степеней свободы, так как каждый атом имеет 3 пространственные координаты.

10.6.1 Проблема конформеров

Казалось бы, задача ясна: надо найти минимум этой функции $E(\mathbf{x})$, где $\mathbf{x} \in \mathbb{R}^{3N}$ — координаты всех атомов. Но тут начинается самое интересное.

Функция энергии молекулы — это **мультимодальный ландшафт** с огромным числом локальных минимумов. Каждый локальный минимум соответствует своей **конформации** (конформеру) молекулы — своему способу скручивания молекулы в пространстве.

Например, для белка из 100 аминокислот число возможных конформеров может достигать 10^{100} — это знаменитый **парадокс Левинталя**. И глобальный минимум (нативная структура) — это лишь одна из 10^{100} возможностей.

10.6.2 Почему простая минимизация не работает

Если мы возьмём случайную начальную конформацию и запустим обычный градиентный спуск или BFGS, то мы очень быстро (за несколько сотен итераций) сойдёмся в *ближайший* локальный минимум. Но это почти наверняка *не* будет глобальный минимум — то есть не та конформация, которую молекула принимает в реальности.

10.6.3 Решение: комбинация методов

Поэтому на практике используют комбинацию методов:

1. **Simulated Annealing (имитация отжига):** Начинаем с высокой "температуры", которая позволяет алгоритму перепрыгивать через энергетические барьеры и исследовать разные области конформационного пространства. Постепенно снижаем температуру, "замораживая" систему в низкоэнергетической области.

2. **Локальная минимизация (BFGS):** После того как simulated annealing нашёл "хорошую" область, запускаем быстрый локальный метод (BFGS или метод сопряжённых градиентов) для точной сходимости к локальному минимуму.

3. **Повторение:** Запускаем эту комбинацию много раз с разными начальными условиями и выбираем конформер с наименьшей энергией.

Популярные программы

Этот подход реализован в популярных программах молекулярной динамики:

- **AMBER, GROMACS, NAMD:** Используют силовые поля (AMBER, CHARMM, OPLS) и методы типа simulated annealing + локальная минимизация.
- **Rosetta:** Для предсказания структуры белков использует фрагментный подход + Monte Carlo + локальная минимизация.

11 Быстрое преобразование Фурье: когда $O(N^2)$ превращается в $O(N \log N)$

11.1 Задача поиска паттерна

Иногда нам надо найти какой-то определенный фрагмент в огромной последовательности. Если это — очень короткий фрагмент, то простой перебор всей последовательности и сравнение с этим коротким фрагментом — это довольно хорошая стратегия. Так часто поступают при поиске коротких фрагментов в белковых структурах.

Но иногда размерности того, что надо сравнить, почти совпадают. Например, у нас есть какая-то периодическая кристаллическая структура, и мы знаем, что в ней есть какое-то отклонение, и даже понимаем, что отклонение только частично похоже на то, с чем мы хотим сравнить.

Формально: исходные данные — это v_i, $i = 1, \ldots, N$, и то, с чем надо сравнить — w_j, $j = 1, \ldots, M$, где $M < N$.

Мы можем явно в цикле посчитать:

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

и найти максимум по s.

Но вычислительная сложность такого поиска будет квадратична по N: если $M \simeq N/2$, то надо выполнить $N^2/2$ арифметических операций. Для $N = 10^6$ это 5×10^{11} операций — слишком много!

11.2 Матричная формулировка

Тут люди придумали, как найти такие s_k гораздо быстрее. Давайте представим вычисление этих s в матричном виде. Пусть у нас есть матрица:

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

размера $(N - M + 1) \times N$.

Если мы умножим её на наш вектор $\mathbf{v}$, то получим как раз наши заветные s_k:

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

Но как же нам это сделать быстро?

11.3 Циркулянтная матрица

Если мы дополним эту матрицу снизу ещё $M - 1$ строками вида:

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

то после умножения мы получим тот же вектор $\mathbf{s}$, но в конце к нему будет приписано ещё $M - 1$ каких-то чисел, которые мы можем отбросить.

Но эта расширенная матрица — это **циркулянтная матрица C**! Она обладает замечательным свойством: каждая следующая строка получается циклическим сдвигом предыдущей.

11.4 Диагонализация циркулянтной матрицы

У циркулянтной матрицы есть интересная запись через матрицу Фурье:

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

где:

- $\mathbf{c}$ — первый столбец исходной матрицы $\mathbf{C}$,
- $\mathbf{F}$ — матрица дискретного преобразования Фурье (ДПФ),
- $\mathbf{F}^H$ — эрмитово сопряжённая (комплексно сопряжённая и транспонированная) матрица.

Матрица Фурье определяется как:

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

11.5 Быстрое умножение через FFT

Теперь самое интересное. Если мы умеем быстро умножать матрицу Фурье на вектор, то мы сможем быстрее вычислить этот $\mathbf{s}$:

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

Это вычисление состоит из трёх шагов:

1. $\mathbf{a} = \mathbf{F}\mathbf{v}$ — прямое ДПФ вектора $\mathbf{v}$,
2. $\mathbf{b} = \operatorname{diag}(\mathbf{F}\mathbf{c})\mathbf{a}$ — поэлементное умножение (это $\mathcal{O}(N)$),
3. $\mathbf{s} = \frac{1}{N}\mathbf{F}^H\mathbf{b}$ — обратное ДПФ.

11.6 Разложение матрицы Фурье

Как же быстро умножить $\mathbf{F}$ на вектор? Оказывается, $\mathbf{F}$ можно представить в виде произведения специальных матриц:

1. Одной матрицы перестановки (bit-reversal permutation),
2. Нескольких пар матриц:
   - Блочных матриц вида $\begin{pmatrix} \mathbf{I} & \mathbf{I} \\ \mathbf{I} & -\mathbf{I} \end{pmatrix}$,
   - Диагональных матриц с элементами $e^{-2\pi ik/N}$ (так называемые “twiddle factors”).

Давайте наберём эти матрицы явно для $N = 8$:
Матрица перестановки (bit-reversal):

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

(строки переставлены согласно обращению бит: $0, 4, 2, 6, 1, 5, 3, 7$)

Блочная матрица (первый уровень):

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

Диагональная матрица (twiddle factors, первый уровень):

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

И так далее для $\log_2 N$ уровней.

Тогда каждое такое умножение будет требовать только N и $2N$ арифметических операций, а всего таких шагов будет $\log_2 N$.

Итого: умножить матрицу Фурье на вектор можно за $\mathcal{O}(N \log_2 N)$ арифметических операций!

А значит, и посчитать все s_k можно за те же $\mathcal{O}(N \log_2 N)$, а не за $\mathcal{O}(N^2)$.

Для $N = 10^6$:

- Наивный алгоритм: 10^{12} операций,
- FFT: 2×10^7 операций (в 50 000 раз быстрее!).

11.7 Быстрое преобразование Фурье (FFT)

Это и есть знаменитый **метод Быстрого Преобразования Фурье** (Fast Fourier Transform, FFT), открытый Кули и Тьюки в 1965 году (хотя Гаусс знал его ещё в 1805-м!).

Он является одним из самых востребованных алгоритмов в современной химии.

11.7.1 Применение в ЯМР

С его помощью, например, преобразуют исходные FID (Free Induction Decay) из ЯМР в спектральную область.

Если посмотреть на каждую строку матрицы Фурье, то в ней можно увидеть осциллирующую функцию:

- Чем ближе строка к центру матрицы, тем больше осцилляция,
- У первой строки её вообще нет — это просто константа,
- У самой нижней (или второй) строки период этой осцилляции равен длине строки.

Именно поэтому, умножив матрицу Фурье на FID, мы получаем максимумы там, где есть резонансные частоты: просто именно такая строчка совпала по частоте осцилляции с резонансной частотой.

Почему FFT так важен для ЯМР

Современные FID очень длинные — их часто оцифровывают по сотни тысяч и даже миллионы чисел. Если бы быстрого преобразования Фурье не было, умножение на такую матрицу длилось бы на современных компьютерах даже дольше, чем сам съём ЯМР-спектров — то есть реально минуты и десятки минут даже на современных рабочих станциях.

Благодаря FFT преобразование занимает **доли секунды**.

11.7.2 Другие применения FFT в химии

- **Крио-электронная микроскопия:** Реконструкция 3D-структур из 2D-проекций.
- **Рентгеноструктурный анализ:** Преобразование дифракционной картины в электронную плотность.
- **Молекулярная динамика:** Вычисление электростатических взаимодействий через метод PPPM (Particle-Particle Particle-Mesh).
- **Обработка сигналов:** Фильтрация шума, сжатие данных.
- **Хемометрика:** Быстрая свёртка и корреляция спектров.

Итог

FFT — это один из тех алгоритмов, которые изменили мир. Без него не было бы современной спектроскопии, обработки сигналов, сжатия аудио и видео, и многого другого. Это must-know для любого химика, работающего с данными.

12 Когда БПФ не видит очевидного: утечка спектра и метод Прони

12.1 Парадокс: глаз видит синусоиду, а БПФ — нет

Хотя БПФ — это must-have алгоритм практически везде в экспериментальной химии, у него тоже есть свои особенности, на которые надо однозначно обращать внимание.

Рассмотрим пример. Мы сняли спектр, сигнал осциллирует, и это прямо видно глазами на графике. Но мы смогли записать только слегка больше одного периода. Когда мы применили БПФ, получили что-то очень странное: пик вроде бы есть, но и вроде бы его нет — он получился каким-то очень размазанным. А если у нас было много другого шума, то мы получили в другой части спектра что-то очень похожее, и да, случайно шум оказался “ярче” сигнала, и наша программа выдала совершенно неправильный результат.

Но мы же глазами видим эту синусоиду! Почему же БПФ её не видит?

12.2 Утечка спектра (Spectral Leakage)

БПФ хорош тогда, и *только тогда*, когда мы применили его к сигналу, в котором укладывается **точно целое число периодов** этого сигнала.

Почему? Потому что БПФ неявно предполагает, что наш конечный сигнал *периодически продолжается* за пределы окна измерения. Если в окне укладывается ровно k полных периодов, то периодическое продолжение будет гладким, и БПФ даст один чёткий пик.

Но если сигнал содержит, скажем, 1.4 периода, то при периодическом продолжении возникнет **разрыв** на границе окна. Этот разрыв БПФ вынужден аппроксимировать множеством гармоник — и энергия “утекает” из основного пика во все остальные частоты. Это явление называется **утечкой спектра** (spectral leakage).

Математически: если истинная частота сигнала попадает *между* двумя соседними частотными бинами БПФ, то вместо одной дельта-функции мы получаем размазанный “горб” (функцию вида $\sin(x)/x$, так называемый *синк*).

Правило БПФ

БПФ видит только те частоты, которые укладываются в окно измерения целое число раз. Всё остальное — утечка.

12.3 Zero-padding: простое, но не идеальное решение

Что же нам тогда делать?

Самый простой вариант — добавить много нулей в конец сигнала (zero-padding) и надеяться, что суммарно нам существенно больше повезёт. Ведь если мы, к примеру, добавим нули так, что исходные данные растянутся в 5 раз, то у нас точно получится 7 периодов (вместо 1.4). А если увеличим, скажем, в 7 раз, то будет 9.8 периода — тоже очень близко к целому.

Люди часто добавляют разное число нулей, а потом пытаются объединить получаемые спектры, угадывая, какие пики оказались более точными на соответствующих расширенных спектрах. Это реально помогает!

Важный нюанс zero-padding

Zero-padding *не увеличивает реальное частотное разрешение* — он только интерполирует спектр, делая его более гладким. Разрешение определяется *длиной исходного сигнала*, а не длиной дополненного нулями. Но на практике zero-padding помогает “попасть” в целое число периодов и уменьшить утечку.

12.4 Дрейф частоты: когда сигнал “уплывает”

Но есть ещё одна проблема, с которой zero-padding не справится.

Иногда мы измеряем долго и нудно какой-то спектр, а он “слегка” взял и поплыл. То есть в начале у нас была частота ω, а под конец $\omega + \varepsilon$, и этой маленькой поправки достаточно, чтобы мы совершенно не попали по БПФ и получили вместо чёткого единичного пика что-то очень размытое.

Но ведь глазом-то видно — вот она синусоида, и тут, и там, и в начале, и в конце! Только вот БПФ её опять не видит.

Причина в том, что БПФ предполагает **стационарность** сигнала — то есть что частоты не меняются со временем. Если частота дрейфует, то ни одна гармоника БПФ не может хорошо аппроксимировать весь сигнал целиком.

Что же нам делать?

12.5 Метод Прони: идея сдвигов

На этот вопрос нам несколько веков назад ответил Гаспар Прони (Gaspard de Prony, 1795). Вернее, он придумал, как решать свою задачу (разложение сигнала на сумму экспонент), но она как раз решает и нашу проблему — когда спектр во времени слегка уплывает.

Идея: Возьмём сигнал и сохраним его в вектор $\mathbf{s} = (s_0, s_1, \ldots, s_{N-1})^T$. Далее сдвинем этот вектор вниз на один отсчёт, потом ещё раз, и снова поставим рядом. Сделаем так L раз.

Что же мы будем тут иметь?

Если у нас был периодический сигнал с какой-то неизвестной синусоидой $\sin(\omega t)$, то после сдвига на p отсчётов мы получаем $\sin(\omega t + \omega p)$. Вспоминая формулы сложения из главы 1:

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

Поскольку $\cos(\omega p)$ и $\sin(\omega p)$ — это *константы* для всего вектора (они не зависят от t), то набор таких сдвинутых векторов будет иметь только **столько ненулевых сингулярных чисел, сколько у нас удвоенное число различных осциллирующих гармоник**.

Более того, если какая-то частота со временем немного “ушла”, это *никак не повлияет* на такую аппроксимацию, но совершенно разрушит результат БПФ, как мы отмечали выше.

12.6 Метод Прони, вариант 1: линейное предсказание

Пусть наш сигнал моделируется как сумма K комплексных экспонент:

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

где $z_m = e^{(\alpha_m + \mathrm{i}\omega_m)\Delta t}$ — комплексные "частоты" (содержащие и затухание α_m, и осцилляцию ω_m), а c_m — комплексные амплитуды.

Ключевой факт: Если сигнал состоит из K экспонент и нет шума, то он удовлетворяет **линейному рекуррентному соотношению** порядка K:

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

Это означает, что $(n+1)$-й отсчёт можно *точно предсказать* через K предыдущих!

Коэффициенты $a_1, \ldots, a_K$ — это коэффициенты характеристического полинома, корнями которого являются z_m:

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

Как найти коэффициенты? Запишем рекуррентное соотношение для всех доступных отсчётов в матричном виде:

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

Это переопределённая система (если $N > 2K$), и мы решаем её методом наименьших квадратов:

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

После нахождения $\mathbf{a}$ мы ищем корни полинома $P(z)$ — это и есть наши z_m, из которых извлекаются частоты ω_m и затухания α_m.

12.7 Метод Прони, вариант 2: SVD матрицы сдвигов

Второй вариант — более устойчивый к шуму и более элегантный.

Шаг 1: Строим матрицу Ганкеля из сдвигов сигнала.

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

размера $(N-L+1) \times L$, где $L > K$ (мы выбираем L заведомо больше ожидаемого числа гармоник).

Шаг 2: Делаем SVD.

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

Если сигнал состоит из K экспонент и нет шума, то $\mathrm{rank}(\mathbf{H}) = K$, и последние $L - K$ сингулярных чисел равны нулю:

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

В присутствии шума малые сингулярные числа не равны нулю, но они *существенно меньше* первых K. Мы отбрасываем их (это называется **усечённое SVD** или **регуляризация**).

Шаг 3: Извлекаем полином из нуль-пространства.

Сингулярный вектор $\mathbf{v}_{\min}$, соответствующий *самому маленькому* сингулярному числу (последний столбец $\mathbf{V}$), лежит в (приближённом) нуль-пространстве матрицы $\mathbf{H}$. Это означает:

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

Если записать компоненты $\mathbf{v}_{\min} = (v_0, v_1, \ldots, v_{L-1})^T$, то это соотношение эквивалентно тому, что:

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

То есть $\mathbf{v}_{\min}$ содержит коэффициенты полинома:

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

Шаг 4: Ищем корни полинома.

Корни $Q(z)$ содержат K “сигнальных” корней $z_m = e^{(\alpha_m + \mathrm{i}\omega_m)\Delta t}$ (лежащих внутри или на единичной окружности) и $L - K$ “шумовых” корней (разбросанных случайно).

Из сигнальных корней извлекаем:

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

Почему SVD-вариант лучше?

SVD-вариант метода Прони устойчивее к шуму, потому что:

1. Усечение малых сингулярных чисел автоматически фильтрует шум.
2. Мы не решаем переопределённую систему напрямую (что может быть плохо обусловлено), а используем ортогональное разложение.
3. Число гармоник K определяется автоматически по “скачку” в спектре сингулярных чисел.

12.8 Ограничения метода Прони

Но Прони — это не панацея.

Стоит попасться в исходном сигнале составляющей, которая хорошо аппроксимируется как самой функцией, так и её сдвигами (например, белый шум или очень широкополосный сигнал), как мы сразу получаем на неё “корень”, и потом оказывается, что он — **ложный**.

Другие ограничения:

- Метод предполагает, что сигнал — это сумма *экспонент* (включая синусоиды как частный случай). Если сигнал имеет другую структуру (например, прямоугольные импульсы), метод будет работать плохо.
- Число гармоник K нужно знать или угадывать заранее (хотя SVD помогает с этим).
- Поиск корней полинома высокой степени ($L > 50$) сам по себе может быть численно неустойчивым.
- Метод чувствителен к сильному шуму: при низком отношении сигнал/шум ложные корни могут “замаскироваться” под настоящие.

В то же время этот метод очень активно и довольно издавна применяется в ЯМР-спектроскопии и хорошо дополняет БПФ. На практике часто используют **гибридный подход**:

1. БПФ для быстрого обзора спектра и определения числа пиков,
2. Метод Прони (или его современные варианты: MUSIC, ESPRIT, Matrix Pencil) для точного определения частот, затуханий и амплитуд отдельных пиков.

Итог

БПФ — это мощный, но “слепой” инструмент: он видит только то, что укладывается в его частотную сетку. Метод Прони “видит” частоты между бинами и устойчив к дрейфу, но требует больше вычислений и осторожности. В реальной ЯМР-спектроскопии они работают в паре.

13 Наименьшие квадраты, невязки и регуляризация по Тихонову

13.1 Постановка задачи

Итак, мы часто решаем задачи, которые мы называем задачами наименьших квадратов. Давай рассмотрим какие-нибудь из них, чтобы поточнее понять, как же это можно решать.

13.2 Пример из медицины: компьютерная томография (КТ)

Возьмём пример из смежной области — из медицины. Пусть у нас есть рентгеновский КТ-сканер. Мы поместили пациента (или то, что мы исследуем) в неподвижном виде на какую-то площадку, и вокруг него с противоположных сторон располагаем передатчик и приёмники рентгеновских лучей. Мы выполняем измерения, насколько сильно излучение поглотилось исследуемым телом, и располагаем эти приёмники и передатчики в различных ракурсах.

Обычно пациент лежит на кушетке, а приёмник и передатчики расположены на поверхности кольца. Это кольцо расположено так, что ось находится примерно вдоль кушетки, и само кольцо как вращается вдоль своей оси, так и передвигается вдоль кушетки.

Что же тут происходит?

Мягкие ткани почти не поглощают рентгеновское излучение, в то время как кости — поглощают довольно сильно.

Мы можем в уме разбить всё пространство, где расположен пациент, на небольшие (обычно одинаковые) кубики — **воксели** (volume elements). Если мы знаем точно расположение передатчика и приёмника, то мы можем провести линию (рентгеновский луч), которая пересекает наши кубики.

Пусть в каждом таком кубике у нас есть одно неизвестное — это интенсивность поглощения рентгеновского излучения. Мы пронумеруем все эти кубики и запишем интенсивность их поглощения как неизвестный вектор $\mathbf{x} = (x_1, \dots, x_N)^T$.

Тогда одно взаимное расположение передатчика и приёмника даст нам довольно разреженный набор коэффициентов — длины прохождения луча через соответствующий кубик. Пусть это всё записано в строке матрицы $\mathbf{A}$, и число столбцов в ней равно N (число вокселей), а число строк — как раз равно числу проведённых измерений (пусть равное M).

А вот сами измеренные значения (у нас может быть, например, один источник рентгена и много приёмников; тогда каждое взаимное расположение приёмника и источника — это одна строка) будут записаны в векторе $\mathbf{b}$ тоже длины M.

Тогда мы можем сформулировать задачу наименьших квадратов как:

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

то есть мы хотим, чтобы каждая строка матрицы $\mathbf{A}$, умноженная на $\mathbf{x}$ (суммарная интенсивность поглощения), была равна тому, что мы измеряем.

13.3 Нормальные уравнения

Мы помним, что такую задачу можно решить, но давай посмотрим на неё внимательно.

Функция невязки:

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

Если мы найдём производную E по всем x_j, приравняем к нулю (в минимуме), то получим систему уравнений:

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

Это так называемые **нормальные уравнения**.

Тут вроде всё хорошо: у нас получается неотрицательно определённая матрица $\mathbf{A}^H\mathbf{A}$ и какая-то правая часть, с которой мы можем решить эту задачу.

13.4 Проблема вырожденности

А что будет, если мы построили такие кубики, но у нас не было ни одного эксперимента, в котором рентгеновский луч прошёл через, например, i-ый кубик?

Очевидно, что тогда в исходной матрице $\mathbf{A}$ соответствующий i-ый столбец будет содержать только нули, а матрица $\mathbf{A}^H\mathbf{A}$ будет иметь нули как в i-ом столбце, так и в i-ой строке, то есть иметь гарантированно одно нулевое сингулярное значение.

Пусть этот кубик был где-то в пространстве, и в нём нет части исследуемого тела, то есть — на наше счастье — нам этот кубик и не нужен. Но такая постановка *испортит* решение: мы просто не сможем решить эту задачу, так как матрица $\mathbf{A}^H\mathbf{A}$ будет вырожденной.

Более того, мы не можем гарантировать, что даже если в $\mathbf{A}^H\mathbf{A}$ нет таких нулевых столбцов и строк, то её обусловленность будет хорошей. То есть мы, вместо решения, можем получить совершенно неверные числа.

13.5 Решение через SVD и псевдообратную матрицу

Как же в этом случае быть?

Вернёмся к сингулярному разложению матрицы $\mathbf{A} = \mathbf{U}\square\mathbf{V}^H$. Мы можем заметить, что:

- Если в $\square$ есть нули на диагонали, то они относятся к тем кубикам, через которые не проходил рентгеновский луч.
- Если есть что-то очень маленькое, то оно относится к таким экспериментам, когда рентгеновский луч только несколько раз и, возможно, только касаясь, попадал в какой-то кубик или набор кубиков.

То есть нам надо бы отказаться учитывать такие данные, но их очень сложно отфильтровать ещё на стадии формирования матрицы $\mathbf{A}$.

Но раз эти данные малоинформативны, давай мы их просто “выбросим”! То есть выполним это SVD и занулим маленькие диагональные значения в $\square$, и реконструируем $\mathbf{A}$.

Это, конечно, хорошо, но мы всё равно не знаем, как решить такую задачу, так как $\mathbf{A}^H\mathbf{A}$ тоже будет содержать нулевые сингулярные значения.

Но если для квадратной невырожденной системы $\mathbf{Ax} = \mathbf{b}$ можно представить решение как:

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

то и нашу задачу наименьших квадратов мы, наверное, могли бы решить по аналогии. Правда, там, где в $\square$ у нас будут нули, в $\square^{-1}$ мы их не будем инвертировать, а запишем туда тоже нуль.

Тогда неправильно будет называть её обратной, и будем называть её **псевдообратной**, обозначая как $\square^+$ (или $\mathbf{A}^+$ для всей матрицы).

13.6 Проблема размера: когда SVD не помещается в память

Все бы ничего, только вот в реальных задачах матрица $\mathbf{A}$ бывает огромной. Часто её элементы вычисляются аналитически, и нет необходимости их хранить. А плотная матрица $\mathbf{U}$, у которой размерность равна $\mathbf{A}$, обычно не помещается в память.

Например, для обычного КТ-скана используют кубики размером около 1 мм. За секунду происходит около 100–500 измерений на 100–500 приёмниках, и всё измерение длится около минуты. То есть N и M легко могут быть порядка 10–100 миллионов, и полная сингулярная матрица потребует 100 петабайт памяти, что неразумно много.

13.7 Регуляризация по Тихонову

Можно заметить, что если к вырожденной матрице $\mathbf{A}^H\mathbf{A}$ добавить единичную $\lambda\mathbf{I}$, так что λ будет в диапазоне тех самых маленьких сингулярных чисел, которые мы занулили, то такое добавление можно записать так:

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

И мы видим, что мы добавляем эту лямбду к каждому сингулярному числу, “превращая” вырожденную или плохо обусловленную матрицу в хорошо обусловленную.

Более того, при малых λ (порядка малых сингулярных чисел) мы почти не изменяем большие сингулярные числа, а вот маленькие просто заменяем на λ.

Тогда, решая систему с такой матрицей, мы просто не “портим” большие сингулярные числа и существенно уменьшаем отклик у малых сингулярных чисел.

Итак, обусловленность матрицы стала существенно меньшей, а мы не “портим” матрицей решение. Зная, что матрица разреженная, мы можем применить какие-то итерационные методы (даже метод сопряжённых градиентов) и очень быстро сойтись.

Фактически, такая λ — это так называемый **регуляризатор по Тихонову**, ведь мы можем вместо исходной задачи решать такую:

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

фактически потребовав, чтобы одновременно минимизировались как невязка, так и норма решения, а их взаимная пропорция регулируется как раз значением этой λ.

Тут есть много красивой теории, в своё время выведенной в 70-х годах прошлого века Тихоновым и его последователями, но суть остаётся простой: такой дополнительный “регуляризатор” помогает решать **плохо поставленные** (ill-posed) задачи.

13.8 Итеративное уменьшение λ

Более того, часто вначале ставят λ довольно большим, тогда сопряжённый градиент сходится очень быстро (обусловленность у матрицы будет очень маленькая), а потом уменьшают λ, каждый раз используя предыдущее решение как начальное значение.

Это позволяет:

- Быстро получить “черновое” решение с большим λ,
- Постепенно уточнять его, уменьшая λ,
- Избежать проблем с обусловленностью на ранних этапах.

Итог

Регуляризация по Тихонову — это мощный инструмент для решения плохо обусловленных задач наименьших квадратов. Она добавляет штраф за норму решения, что стабилизирует решение и позволяет использовать итерационные методы даже для вырожденных систем.

14 Базисные функции: от конечных разностей до сплайнов

14.1 Идея: аппроксимировать неизвестное через известное

До сих пор мы работали с дискретными векторами и матрицами. Но реальные задачи — дифференциальные уравнения, интегралы, волновые функции — живут в непрерывном пространстве. Как же нам свести бесконечномерную задачу к конечномерной?

Ответ: **базисные функции**. Мы представляем неизвестную функцию $f(x)$ как линейную комбинацию известных базисных функций $\phi_k(x)$:

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

и ищем коэффициенты c_k. Это превращает задачу поиска функции в задачу поиска вектора $\mathbf{c} \in \mathbb{R}^N$.

14.2 Метод Ритца (вариационный метод)

Один из самых общих подходов — **метод Ритца**. Пусть у нас есть функционал $E[f]$, который мы хотим минимизировать (например, энергия в квантовой механике). Мы подставляем разложение:

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

и получаем функцию от коэффициентов:

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

Теперь мы минимизируем $E(\mathbf{c})$ по $\mathbf{c}$ — это уже обычная задача оптимизации в $\mathbb{R}^N$.

14.3 Конечные разности (Finite Difference Method, FDM)

Самый простой выбор базисных функций — это **локальные константы** или **линейные функции** на равномерной сетке.

14.3.1 Пример из химии: диффузия

Пусть у нас есть уравнение диффузии концентрации вещества $C(x,t)$:

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

Мы дискретизируем пространство с шагом h: $x_j = jh$, и время с шагом Δt: $t_n = n\Delta t$.

Вторую производную аппроксимируем через центральную разность:

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

где $C_j^n \approx C(x_j, t_n)$.

Тогда схема Эйлера во времени:

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

Это даёт явную рекуррентную формулу:

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

Устойчивость

Явная схема устойчива только при $\frac{D\Delta t}{h^2} \leq \frac{1}{2}$. Если шаг по времени слишком большой — решение “развалится”.

14.4 Конечные элементы (Finite Element Method, FEM)

Более сложный, но более гибкий подход — **конечные элементы**. Мы разбиваем область на маленькие элементы (треугольники, тетраэдры) и на каждом элементе аппроксимируем функцию **локальными полиномами** (обычно линейными или квадратичными).

Базисные функции — это **кусочно-линейные “шляпы”** (hat functions), каждая из которых равна 1 в одном узле и 0 во всех остальных.

Преимущества FEM:

- Можно работать со сложными геометриями,
- Можно адаптировать сетку (сгущать там, где решение быстро меняется),
- Хорошо теоретически обоснован.

14.5 Базисные функции в квантовой химии: гауссовы орбитали

А теперь — самое интересное для химиков. В квантовой химии (метод Хартри–Фока, DFT) мы решаем уравнение Шрёдингера:

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

где $\hat{H}$ — оператор Гамильтона, Ψ — волновая функция.

Мы представляем молекулярные орбитали $\psi_i(\mathbf{r})$ как линейную комбинацию **атомных орбиталей** (LCAO — Linear Combination of Atomic Orbitals):

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

где $\phi_\mu(\mathbf{r})$ — базисные функции, центрированные на атомах.

14.5.1 Гауссовы функции (Gaussian Type Orbitals, GTO)

В большинстве современных программ (Gaussian, ORCA, Q-Chem) в качестве базисных функций используют **гауссовы орбитали**:

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

где:

- $\mathbf{R}_A = (X_A, Y_A, Z_A)$ — координаты атома A,
- l, m, n — угловые квантовые числа (s, p, d, f-орбитали),
- α — показатель экспоненты (контролирует "размер" орбитали).

Почему именно гауссовы функции?

- **Произведение гауссов — снова гаусс:** Это критически важно для вычисления многоцентровых интегралов (электрон-электронное отталкивание),
- **Аналитические интегралы:** Все интегралы вычисляются аналитически,
- **Контракты:** Реальные атомные орбитали аппроксимируются суммой нескольких гауссов (контракты), что повышает точность.

14.5.2 Метод Ритца в квантовой химии

Мы подставляем разложение LCAO в функционал энергии Хартри–Фока или DFT и минимизируем по коэффициентам $c_{\mu i}$. Это приводит к задаче на собственные значения:

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

где:

- $\mathbf{F}$ — матрица Фока (или матрица Кона–Шэма в DFT),
- $\mathbf{S}$ — матрица перекрывания ($S_{\mu\nu} = \langle \phi_\mu | \phi_\nu \rangle$),
- ε_i — орбитальные энергии,
- $\mathbf{c}_i$ — коэффициенты разложения i-й молекулярной орбитали.

Это — обобщённая задача на собственные значения, и она решается итерационно (методом самосогласованного поля, SCF).

Базисные наборы

Популярные базисные наборы в квантовой химии:

- **STO-3G:** Минимальный базис (3 гаусса на каждую атомную орбиталь),
- **6-31G*:** Двойной дзета-качество с поляризационными функциями,
- **cc-pVTZ:** Корреляционно-согласованный тройной дзета (высокая точность).

Чем больше базис — тем точнее результат, но тем дороже вычисления ($\mathcal{O}(N^4)$ для Хартри–Фока, где N — число базисных функций).

14.6 Сплайны: гладкие кусочно-полиномиальные функции

А теперь — про сплайны. Это — базисные функции, которые широко используются в обработке сигналов, компьютерной графике и численных методах.

14.6.1 Что такое сплайн?

Сплайн — это кусочно-полиномиальная функция, которая является **гладкой** (имеет непрерывные производные до некоторого порядка) в узлах сшивки.

Самый популярный — **кубический сплайн** (степень 3, гладкость C^2 — непрерывны функция, первая и вторая производные).

14.6.2 B-сплайны (Basis Splines)

B-сплайны — это специальный набор базисных сплайнов с **конечным носителем** (compact support). Каждый B-сплайн отличен от нуля только на небольшом интервале.

Преимущества:

- **Локальность:** Изменение одного коэффициента влияет только на небольшую область,
- **Численная устойчивость:** Матрицы хорошо обусловлены,
- **Рекурсивное определение:** B-сплайны строятся через рекурсию (формула де Бура).

14.6.3 Двумерные сплайны

B-сплайны легко обобщаются на двумерный случай через **тензорное произведение**:

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

Это используется в:

- Обработке изображений (сглаживание, интерполяция),
- Конечных элементах (высшие порядки),
- Компьютерной графике (поверхности).

14.6.4 Сплайны Безье (Bézier Splines)

Сплайны Безье — это параметрические кривые, определяемые **контрольными точками**. Кривая не проходит через контрольные точки, но "притягивается" к ним.

Формула кривой Безье степени n:

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

где $\mathbf{P}_i$ — контрольные точки, а $\binom{n}{i}(1-t)^{n-i}t^i$ — **полиномы Бернштейна**.

Применение:

- Компьютерная графика (векторная графика, шрифты TrueType),
- CAD-системы (AutoCAD, SolidWorks),
- Анимация и траектории.

14.6.5 Связь с конечными элементами

Интересно, что B-сплайны с конечным носителем **плавно перетекают** в базисные конечные элементы. Если взять B-сплайны степени 0 — это кусочно-постоянные функции (как в простейших конечных элементах). Степени 1 — кусочно-линейные "шляпы". Степени 2 и выше — более гладкие базисы.

Это позволяет строить **изопараметрические конечные элементы** высших порядков, которые дают более точную аппроксимацию при меньшем числе узлов.

Итог

Базисные функции — это мост между непрерывным миром дифференциальных уравнений и дискретным миром компьютеров. Конечные разности — самые простые, конечные эле-

менты — самые гибкие, гауссовы орбитали — специализированы для квантовой химии, а сплайны — для гладкой интерполяции и графики. Понимание их свойств — ключ к выбору правильного метода для вашей задачи.

15 Интегрирование: от аналитики до Монте-Карло

15.1 Зачем нам интегрирование?

Интеграл — это одна из самых фундаментальных операций в математике и физике. Мы постоянно вычисляем:

- Площади и объёмы,
- Математические ожидания и дисперсии,
- Нормировочные константы в квантовой механике,
- Многомерные интегралы в статистической физике,
- Свёртки в обработке сигналов.

И если в школе мы учились брать интегралы аналитически, то в реальной жизни — особенно в химии и физике — аналитика часто заканчивается очень быстро.

15.2 Аналитическое интегрирование: когда оно работает

Вспомним азы. Если у нас есть функция $f(x)$, заданная аналитически, и мы знаем её первообразную $F(x)$, то:

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

Например:

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

Красиво, точно, быстро. Но:

- Не все функции имеют элементарную первообразную (например, e^{-x^2} — интеграл вероятностей),
- Не все функции заданы аналитически — часто у нас есть только набор точек (экспериментальные данные),
- В многомерных случаях аналитика почти всегда бессильна.

15.3 Численное интегрирование в 1D: метод Симпсона

Что же делать, если аналитически взять интеграл нельзя? Ответ: аппроксимировать функцию чем-то простым и проинтегрировать это простое.

Мы в прошлой главе научились строить сплайны третьей степени. А тут — мы строим тоже полиномы, и интегрируем такими кусками.

15.3.1 Метод прямоугольников

Самый простой подход: разбиваем отрезок $[a, b]$ на N равных частей шириной $h = (b - a)/N$, и на каждом отрезке аппроксимируем функцию константой (значением в середине):

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

Погрешность — $\mathcal{O}(h^2)$.

15.3.2 Метод трапеций

Чуть лучше: аппроксимируем функцию линейным полиномом на каждом отрезке:

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

Погрешность — $\mathcal{O}(h^2)$.

15.3.3 Метод Симпсона

Ещё лучше: аппроксимируем функцию **квадратичным полиномом** на каждой паре отрезков:

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

где N — чётное число.

Погрешность — $\mathcal{O}(h^4)$! Это уже очень хорошо для гладких функций.

Почему Симпсон так хорош?

Метод Симпсона — это, по сути, интегрирование кусочно-квадратичных сплайнов. Если функция гладкая (имеет непрерывные производные до 4-го порядка), то погрешность убывает как h^4, что намного быстрее, чем у трапеций.
Для очень гладких функций есть ещё более точные методы — квадратуры Гаусса, которые используют оптимально выбранные узлы и веса.

15.4 Проклятие размерности

Но что будет, если мы будем интегрировать 2-, 3-, 4-, 10-мерную функцию?

Нам надо тогда N^{10} точек интегрирования, где N — хотя бы 100... Это как-то много!

Давайте посчитаем. Для 10-мерного интеграла с $N = 100$ точками по каждой оси:

$$100^{10} = 10^{20} \text{ точек}$$

Даже если каждая точка вычисляется за 1 наносекунду, это займёт:

$$10^{20} \text{ нс} = 10^{11} \text{ с} \approx 3000 \text{ лет}$$

Это — **проклятие размерности** (curse of dimensionality).

15.4.1 Где это встречается в химии?

- **Квантовая химия:** Вычисление многоэлектронных интегралов (электрон-электронное отталкивание) — это 6-мерные интегралы (по 3 координаты на каждый электрон).
- **Статистическая механика:** Статистическая сумма — это интеграл по фазовому пространству всех частиц. Для N частиц — это $6N$-мерный интеграл (3 координаты + 3 импульса на частицу).
- **Молекулярная динамика:** Усреднение по конфигурациям — многомерное интегрирование.
- **Машинное обучение:** Байесовский вывод — интегрирование по пространству параметров модели.

15.5 Метод Монте-Карло для интегрирования

И тут на сцену выходит метод Монте-Карло — один из самых элегантных и удивительных алгоритмов в вычислительной математике.

15.5.1 Идея на пальцах

Представим, что мы хотим вычислить интеграл:

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

где d — размерность (может быть очень большой).

Метод Монте-Карло говорит: “Давай мы просто накидаем N случайных точек $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N$ равномерно в область интегрирования, вычислим в них функцию и усредним”.

Формула:

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

где $V = (b-a)^d$ — объём области интегрирования, а $\mathbf{x}_k$ — случайные точки, равномерно распределённые в $[a,b]^d$.

15.5.2 Почему это работает?

Это — просто закон больших чисел. Математическое ожидание случайной величины $f(\mathbf{X})$, где $\mathbf{X}$ равномерно распределена в $[a,b]^d$, равно:

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

По закону больших чисел, среднее значение $\frac{1}{N}\sum f(\mathbf{x}_k)$ сходится к $\mathbb{E}[f(\mathbf{X})]$ при $N \to \infty$.

15.5.3 Скорость сходимости

А вот тут — магия! Погрешность метода Монте-Карло убывает как:

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

где σ — стандартное отклонение функции $f(\mathbf{x})$ в области интегрирования.

Важно: эта скорость сходимости *не зависит от размерности* d!

Для сравнения:

- Метод Симпсона в 1D: погрешность $\sim h^4 \sim N^{-4}$,
- Метод Симпсона в d-мерном случае: погрешность $\sim N^{-4/d}$ (катастрофически медленно при больших d),
- Метод Монте-Карло в любой размерности: погрешность $\sim N^{-1/2}$.

Да, Монте-Карло сходится медленнее, чем Симпсон в 1D. Но в 10-мерном случае:

- Симпсон: $N^{-4/10} = N^{-0.4}$,
- Монте-Карло: $N^{-0.5}$.

Монте-Карло *быстрее*!

Цена Монте-Карло

Метод Монте-Карло сходится как $N^{-1/2}$ — это значит, чтобы увеличить точность в 10 раз, нужно в 100 раз больше точек. Это медленно по абсолютным меркам, но это *единственный* метод, который работает в высоких размерностях.

15.6 Пример из квантовой химии: вычисление орбиталей

Давайте посмотрим, как метод Монте-Карло применяется в реальной квантовой химии.

15.6.1 Задача: нормировка волновой функции

В квантовой механике волновая функция $\Psi(\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_N)$ описывает состояние N электронов. Она должна быть нормирована:

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

Это — $3N$-мерный интеграл! Для молекулы воды ($N = 10$ электронов) — это 30-мерный интеграл.

15.6.2 Применение Монте-Карло

Мы генерируем M случайных конфигураций электронов:

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

где каждая координата $\mathbf{r}_i^{(k)} = (x_i^{(k)}, y_i^{(k)}, z_i^{(k)})$ выбирается из некоторого распределения (например, гауссова).

Вычисляем $|\Psi|^2$ в каждой конфигурации и усредняем:

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

15.6.3 Вариационный метод Монте-Карло (Variational Monte Carlo, VMC)

Но это ещё не всё! В квантовой химии есть метод **Variational Monte Carlo (VMC)**, который использует Монте-Карло для *минимизации* энергии.

Мы выбираем пробную волновую функцию $\Psi_T(\mathbf{r}_1, \ldots, \mathbf{r}_N; \alpha)$ с параметрами α и вычисляем энергию:

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

Оба интеграла — $3N$-мерные, и мы вычисляем их методом Монте-Карло. Затем минимизируем $E(\alpha)$ по параметрам α — и получаем приближённую волновую функцию.

15.6.4 Quantum Monte Carlo (QMC)

Есть ещё более продвинутые методы — **Diffusion Monte Carlo (DMC)**, которые решают уравнение Шрёдингера напрямую, моделируя "диффузию" электронов в воображаемом времени. Эти методы дают *почти точные* решения для небольших молекул и используются как эталон для проверки других методов.

Где используется Монте-Карло в химии?

- **Quantum Monte Carlo (QMC):** Точное решение уравнения Шрёдингера для небольших систем,
- **Молекулярная динамика Монте-Карло:** Сэмплирование конфигураций в статистической механике,
- **Интегрирование в DFT:** Численное интегрирование обменно-корреляционного функционала,
- **Байесовская оптимизация:** Поиск глобального минимума энергии молекулы.

15.7 Улучшения метода Монте-Карло

Базовый метод Монте-Карло — это просто равномерное случайное блуждание. Но есть много улучшений:

15.7.1 Важностное сэмплирование (Importance Sampling)

Вместо равномерного распределения используем распределение, пропорциональное $|f(\mathbf{x})|$. Тогда точки будут чаще попадать в области, где функция большая, и погрешность уменьшится.

15.7.2 Метод Метрополиса (Metropolis-Hastings)

Для сложных многомерных интегралов используем марковские цепи: каждая следующая точка зависит от предыдущей. Это позволяет эффективно исследовать пространство, даже если оно очень большое.

15.7.3 Квази-Монте-Карло (Quasi-Monte Carlo)

Вместо случайных точек используем **детерминированные** последовательности с низким отклонением (последовательности Соболля, Халтона). Они "равномернее" покрывают пространство, чем случайные точки, и сходятся быстрее: погрешность $\sim N^{-1}(\log N)^d$.

Итог

Метод Монте-Карло — это единственный практичный способ вычисления многомерных интегралов. Его скорость сходимости не зависит от размерности, что делает его незаменимым в квантовой химии, статистической физике и машинном обучении. Хотя он сходится медленнее, чем детерминированные методы в низких размерностях, в высоких размерностях у него просто нет конкуренции.

16 Статистика: как отделить информацию от шума

До сих пор мы в основном рассматривали математические задачи, в которых числа считаются известными. Но экспериментальная химия устроена иначе. Если мы измеряем концентрацию ве-

щества, положение спектральной линии или интенсивность сигнала, прибор никогда не сообщает нам “истинное” значение с бесконечной точностью. Каждое измерение немного отличается от другого. Поэтому вместо одного числа нам приходится иметь дело с **случайной величиной**.

Пусть мы несколько раз измеряем одну и ту же величину. Получаем: $(x_1, x_2, \dots, x_N) = \mathbf{x}$, причём $\mathbf{x} = \mu + \varepsilon$, где

- μ — неизвестное истинное или среднее значение;
- ε — случайная ошибка измерения.

Это простое уравнение является одной из основных идей статистики.

Статистика начинается там, где мы понимаем: *мы наблюдаем данные, но интересуемся скрытыми величинами.*

16.1 Среднее

Самая простая оценка μ — среднее арифметическое:

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

Почему именно оно? Если ошибки имеют среднее около нуля, то среднее арифметическое минимизирует сумму квадратов отклонений: $\min_{\mu} \sum_{i=1}^{N} |x_i - \mu|^2$, и решение даётся формулой выше. Кстати, это уже объясняет, почему повторение эксперимента обычно повышает точность результата.

16.2 Дисперсия и стандартное отклонение

Среднее говорит нам, где находится центр данных, но ничего не говорит о том, насколько сильно результаты разбросаны. Для этого вводят дисперсию — меру того, насколько сильно наши измерения “разбросаны” вокруг истинного значения (к которому стремится среднее):

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

Да, мы же только что её минимизировали? Верно, но значение минимума будет равно нулю тогда и только тогда, когда все x_i одинаковы; иначе это значение и есть дисперсия. А чтобы измерять ее в тех же единицах, мы просто вспомним, что вторая норма — содержит в себе корень, и запишем

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

То есть дисперсия — это, по существу, **квадрат длины вектора отклонений от среднего**. Статистика снова превращается в линейную алгебру.

16.3 Нормальное распределение

Во многих физических и химических измерениях случайные ошибки приблизительно описываются нормальным распределением:

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

Не обязательно запоминать эту формулу. Гораздо важнее понять смысл двух параметров: μ и σ. Первый определяет положение центра распределения, второй — его ширину, то есть чем больше σ, тем сильнее разброс измерений.

Интуиция

- Среднее отвечает на вопрос: *“Где находится результат?”*
- Стандартное отклонение отвечает на вопрос: *“Насколько сильно результаты разбросаны?”*

16.4 Почему среднее становится точнее при повторении эксперимента?

Пусть мы сделали N независимых измерений. Для независимых ошибок дисперсия среднего равна $\frac{\sigma^2}{N}$, следовательно, **ошибка среднего уменьшается как** $\frac{\sigma}{\sqrt{N}}$. Поэтому увеличение числа измерений повышает точность, но не линейно (как в Монте-Карло!).

Чтобы уменьшить случайную ошибку в 10 раз, в идеализированном случае понадобится примерно в 100 раз больше независимых измерений.

16.5 Случайная ошибка и систематическая ошибка

Здесь необходимо сделать очень важное различие. Если прибор дает $x = \mu + \varepsilon$, где ε случайно колеблется около нуля, многократные измерения могут уменьшить влияние этой ошибки. Но представим, что прибор систематически завышает результат: $x = \mu + b + \varepsilon$, где b — постоянное смещение. Тогда усреднение дает $\bar{x} \approx \mu + b$. И сколько бы раз мы ни повторяли эксперимент, смещение b никуда не исчезнет.

Важно

Повторение измерений уменьшает случайную ошибку, но не устраняет систематическую ошибку.
Статистика не может исправить неправильно откалиброванный прибор.

Это одна из причин, почему в химии так важны калибровка, контрольные образцы и независимые методы измерения.

16.6 Две величины могут быть связаны

Пусть одновременно измеряются две величины: x_i и y_i. Например:

- концентрация вещества и интенсивность сигнала;
- температура и скорость реакции;
- давление и объем;
- две спектральные характеристики.

Нас может интересовать вопрос: *изменяются ли x и y согласованно?*

Для этого используют ковариацию: $\mathrm{Cov}(x, y) = \frac{1}{N}(x - \mu_x)^T(y - \mu_y)$

Если большие значения x обычно соответствуют большим значениям y, ковариация положительна. Если большие значения x соответствуют маленьким y, она отрицательна. Если линейной связи нет, ковариация может быть близка к нулю.

16.7 Корреляция

Ковариация зависит от единиц измерения. Поэтому часто используют нормированную величину: $\rho_{xy} = \dfrac{\mathrm{Cov}(x,y)}{\sigma_x \sigma_y}$. Для нее $-1 \leq \rho_{xy} \leq 1$. Значение около 1 означает сильную положительную линейную связь, значение около -1 — сильную отрицательную, а значение около 0 означает отсутствие выраженной линейной корреляции.

Но здесь необходимо запомнить одну очень важную вещь:

Корреляция не означает причинность

Если две величины коррелируют, это еще не означает, что одна вызывает другую. Корреляция говорит о статистической связи между данными. Чтобы установить причинную связь, необходимы дополнительные физические, химические или экспериментальные аргументы.

16.8 Ковариационная матрица

Если у нас не две, а p измеряемых величин, $x_1, x_2, \ldots, x_p$, то ковариации всех пар можно собрать в одну матрицу:

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

Эта матрица симметрична: $\Sigma = \Sigma^T$, и она не просто хранит статистическую информацию. Она является математическим объектом линейной алгебры. Именно поэтому собственные значения и собственные векторы, с которыми мы уже встречались раньше, снова возникают в статистике.

16.9 PCA: статистика встречается с линейной алгеброй

Представим, что у нас есть N химических образцов, для каждого из которых измерено p характеристик. Получаем матрицу данных: $X \in \mathbb{R}^{N \times p}$. Некоторые характеристики могут быть сильно связаны между собой. Например, если две измеряемые величины почти всегда меняются вместе, информация о них частично избыточна. Хотелось бы найти новые координаты, в которых:

- первая координата содержит максимально возможную вариацию данных;
- вторая — максимально возможную оставшуюся вариацию;
- третья — следующую и так далее.

Это и есть основная идея **метода главных компонент** (Principal Component Analysis, PCA). Математически PCA тесно связан с собственными векторами ковариационной матрицы: $\Sigma v_i = \lambda_i v_i$. Собственные векторы v_i задают новые направления, а собственные значения λ_i показывают, сколько вариации данных приходится на соответствующее направление.

Если несколько первых собственных значений значительно больше остальных,

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

то большая часть информации может быть представлена всего несколькими координатами. Например,

$$5000 \text{ измеряемых параметров} \quad \longrightarrow \quad 20 \text{ главных компонент}.$$

Это не означает, что мы “выбросили” химическую информацию. Мы нашли более компактное математическое представление данных.

И здесь снова появляется SVD, с которым мы уже встречались:

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

PCA и SVD оказываются двумя сторонами одной и той же линейно-алгебраической идеи.

16.10 Регрессия: когда мы хотим построить модель

Предположим, что мы измерили концентрацию c_i и соответствующий сигнал y_i. Мы предполагаем линейную модель:

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

Поскольку измерения содержат ошибки, точки обычно не лежат точно на одной прямой. Поэтому ищем такие a и b, которые минимизируют сумму квадратов ошибок:

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

Это и есть **метод наименьших квадратов**. Таким образом, регрессия не является чем-то совершенно новым. Мы уже знаем эту математическую конструкцию:

данные $\rightarrow$ модель $\rightarrow$ невязка $\rightarrow$ минимизация

Именно поэтому линейная алгебра и оптимизация так важны для статистики.

16.11 Переобучение

Теперь возникает более сложная проблема. Если данных немного, мы можем подобрать очень сложную функцию, которая пройдет практически через каждую экспериментальную точку. Например, вместо прямой можно использовать полином высокой степени:

$$y = a_0 + a_1x + a_2x^2 + \cdots + a_kx^k.$$

При достаточно большом k можно почти идеально описать имеющиеся измерения. Но это не обязательно означает, что мы нашли правильный физический закон. Мы могли просто подстроиться под шум. Это называется **переобучением** (overfitting).

Именно поэтому в статистике и машинном обучении нас интересует не только вопрос: *“Насколько хорошо модель описывает известные данные?”*, но и: *“Насколько хорошо она работает на новых данных?”*

16.12 Обучение, проверка и тестирование

Если мы строим модель по имеющимся данным, полезно разделить данные на части:

training	+	validation	+	test

На training set модель обучается. Validation set используется для выбора параметров модели. Test set должен оставаться независимым и используется для окончательной оценки способности модели работать на новых данных. Эта идея будет особенно важна, когда мы дойдем до машинного обучения.

16.13 Что статистика действительно пытается сделать

Можно сказать, что статистика занимается не столько “подсчетом средних”, сколько более общей задачей: **извлечь надежную информацию из несовершенных данных.** Мы наблюдаем:

$$\text{данные} = \text{структура} + \text{шум}.$$

Наша задача состоит в том, чтобы по данным восстановить интересующую нас структуру и одновременно оценить, насколько мы можем ей доверять. В самом простом случае это выглядит так:

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

В более сложном случае:

$$X \longrightarrow \text{PCA} \longrightarrow \text{низкоразмерное представление}.$$

А еще сложнее:

$$X \longrightarrow \text{модель} \longrightarrow \text{предсказание} \longrightarrow \text{проверка на новых данных}.$$

Именно отсюда естественным образом вырастает современное машинное обучение.

Что важно запомнить

1. Реальное измерение содержит случайную и, возможно, систематическую ошибку.
2. Среднее описывает центральное значение данных.
3. Стандартное отклонение описывает их разброс.
4. Случайная ошибка среднего уменьшается как $1/\sqrt{N}$.
5. Систематическая ошибка усреднением не устраняется.
6. Ковариация описывает совместное изменение величин.
7. PCA ищет наиболее информативные направления в многомерных данных.
8. Регрессия строит математическую модель по измерениям.
9. Хорошая модель должна работать не только на известных данных, но и на новых.

16.14 И снова линейная алгебра

И, пожалуй, самое приятное для нас наблюдение состоит в том, что статистика оказалась не таким уж чужим предметом.

Мы начали с измерений: $x_1, x_2, \ldots, x_N$,
перешли к векторам: $\mathbf{x}$,
затем к нормам: $\|\mathbf{x}\|_2$,
к матрицам ковариаций: Σ,
к собственным значениям: $\Sigma v = \lambda v$,
к SVD: $X = U\Sigma V^T$,
и наконец к оптимизации: $\min \|Ax - b\|_2^2$.

То есть статистика не разрушает нашу математическую картину. Она показывает, как та же самая математика начинает работать с **реальными, несовершенными и зашумленными данными**. И это именно та ситуация, с которой современный химик сталкивается почти каждый день.

17 От SVD к сжатым измерениям: когда данных меньше, чем нужно

17.1 Практическая задача: разделение спектров

Давайте рассмотрим практическую задачу. Нам в лабораторию принесли массу цветастой органики, под рукой был только VIS/IR спектрометр для флуоресцентных спектров. Сказали, что в семплах есть несколько продуктов реакции с предположительно отличающимися спектрами, и попросили определить как концентрации, так и спектры чистых веществ.

Мы подобрали не сильно УФ-источник, чтобы флюоресценция была очень яркой и давала разные спектры для разных веществ.

Кажется, это задача для SVD! Ведь если мы поставим каждый такой спектр в свой столбец матрицы **A**, положим, что у нас гипотетически имеется матрица чистых спектров этих веществ **P**, и для каждого семпла есть коэффициенты концентрации самих веществ **Q**, тогда:

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

и мы можем получить **P** как левые сингулярные вектора матрицы **A**, а **Q** — как правые сингулярные вектора, умноженные слева на диагональ.

Мы всегда можем также записать сингулярное разложение как:

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

Более того, если у нас было бы одно единственное вещество ($R = 1$), именно так бы это и было.

17.2 Но что-то пошло не так...

Вместо красивых чистых спектров с положительными значениями, мы получили какие-то странные графики в **P** — с отрицательными значениями, осцилляциями, “призрачными” пиками.

На самом деле — всё так и должно было быть, спектры так получить *нельзя*. Сами спектры — это положительно заданные графики, значит скалярное произведение любой пары хотя и может быть очень близким к нулю, не обязательно им является. В сингулярном разложении мы требуем, чтобы все столбцы **U** были **ортогональны** друг другу. А реальные спектры веществ — не ортогональны! Они просто “похожи” или “не похожи”.

Нам нужно что-то ещё, чтобы растащить эти спектры так, чтобы наша математика увидела их по-отдельности.

17.3 Магия третьего измерения: теорема Крускала

Если мы так поставим эксперимент, что получим 3D-данные, вместо 2D как в примере выше, то можно математически доказать, что существует и **не ортогональное** разложение.

Например, если мы сможем снять спектры флюоресценции с несколькими разными длинами волн возбуждения. Тогда у нас уже будут трёхмерные данные:

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

и, по **теореме Крускала** (Kruskal, 1977), у нас нет требований на ортогональность каждой из таких матриц!

17.3.1 Что говорит теорема Крускала

Теорема Крускала — это один из краеугольных камней многомерного анализа данных. Она даёт условия **единственности** разложения тензора (так называемого CANDECOMP/PARAFAC разложения).

Для тензора 3-го порядка $\mathcal{A} = \sum_{r=1}^{R} \mathbf{a}_r \circ \mathbf{b}_r \circ \mathbf{c}_r$ (где $\circ$ — внешнее произведение векторов), разложение **единственно** с точностью до перестановки и масштабирования компонентов, если:

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

где k_A — это **k-ранг** (Kruskal rank) матрицы **A**, то есть максимальное число k такое, что любое подмножество из k столбцов матрицы **A** линейно независимо.

Почему это так важно?

В обычном SVD мы имеем *бесконечно много* разложений $\mathbf{A} = \mathbf{U}\square\mathbf{V}^H$ — любое ортогональное преобразование внутри даёт новое валидное разложение. Поэтому SVD не может выделить “физические” компоненты — только математически удобные (ортогональные). Теорема Крускала говорит: для тензоров 3-го и более высокого порядка при выполнении условия на k-ранги разложение *единственно*! Это значит, что мы можем восстановить именно те физические компоненты, которые были в данных — без требования ортогональности.

17.3.2 Методы решения: PARAFAC и ALS

На практике разложение тензора ищут итерационными методами. Самый популярный — **ALS** (Alternating Least Squares):

1. Фиксируем **B** и **C**, решаем задачу наименьших квадратов для **A**,
2. Фиксируем **A** и **C**, решаем для **B**,
3. Фиксируем **A** и **B**, решаем для **C**,
4. Повторяем до сходимости.

Это — классическая задача нелинейной минимизации (мы её обсуждали в главе про нелинейные операторы), и она может застревать в локальных минимумах. Поэтому на практике запускают ALS много раз с разными начальными приближениями.

Применимость в химии

PARAFAC/CANDECOMP широко используется в:

- Флуоресцентной спектроскопии (EEM — Excitation-Emission Matrix),
- Хромато-масс-спектрометрии,
- ЯМР-спектроскопии,
- Анализе метаболических данных (метаболомика).

17.4 Многомерные спектры ЯМР

А теперь — самое интересное для химиков-структурщиков. В ЯМР-спектроскопии белков можно построить очень многомерные спектры: 2D, 3D, 4D и даже 5D.

Каждое измерение — это своя частота (химический сдвиг определённого ядра: ^{1}H, ^{13}C, ^{15}N). Кросс-пики в таких спектрах показывают, какие ядра находятся близко друг к другу в пространстве, что позволяет восстановить 3D-структуру белка.

Но вот проблема: если мы хотим снять 4D-спектр с разрешением $1024 \times 256 \times 256 \times 64$ точек, то общее число точек — это $\sim 4 \times 10^9$. А каждое измерение требует времени (чтобы дождаться релаксации), и в сумме это — **недели** работы спектрометра!

Когда-то были времена, когда, чтобы определить все кросс-пики у убиквитина (пептид из 76 аминокислот), приходилось снимать спектр около **трёх недель**. Белок за это время мог деградировать!

17.5 Sparse sampling: меньше — значит больше

Но можно снять только случайно разреженные одномерные спектры — причём даже не 10%, и не 1%, а реально очень-очень немного, и применить такое же разложение, только для разреженных данных (sparse data), и получить такие же надёжные результаты!

Применяя такие методы ещё 20 лет назад, автор с его коллегами ускорил съёмку такого многомерного спектра без потери информации с трёх недель до **15 минут**.

Идея проста: если мы знаем, что спектр *разрежен* в некотором базисе (то есть состоит из небольшого числа пиков), то нам не нужно измерять все точки. Достаточно измерить случайное подмножество, и затем *восстановить* пропущенные точки через оптимизацию.

17.6 Compressed Sensing: революция в измерениях

Это подводит нас к одной из самых красивых идей в современной математике и обработке сигналов — **Compressed Sensing** (сжатые измерения, или compressive sampling).

17.6.1 Классическая теория: Найквист–Шеннон

Традиционная теория дискретизации (Найквиста–Шеннона) говорит: чтобы восстановить сигнал с максимальной частотой $f_{\max}$, нужно измерять его с частотой не менее $2f_{\max}$.

Для 4D-ЯМР спектра с разрешением $1024 \times 256 \times 256 \times 64$ это означает, что мы *обязаны* измерить все 4×10^9 точек. Меньше — нельзя, иначе будет aliasing (наложение частот).

17.6.2 Революция: сжатые измерения

Но в 2004–2006 годах Донехо (Donoho), Кантор (Candès), Тао (Tao) и другие математики показали: если сигнал **разрежен** (sparse) в некотором базисе, то его можно восстановить из **существенно меньшего** числа измерений!

Формально: пусть сигнал $\mathbf{x} \in \mathbb{R}^N$ имеет разреженное представление $\mathbf{x} = \square\mathbf{s}$, где $\mathbf{s}$ имеет только $K \ll N$ ненулевых элементов. Тогда мы можем измерить $\mathbf{y} = \square\mathbf{x}$, где $\square \in \mathbb{R}^{M \times N}$ — матрица измерений с $M \sim K \log(N/K) \ll N$, и восстановить $\mathbf{x}$ через решение задачи:

$$\min_{\mathbf{s}} \|\mathbf{s}\|_1 \quad \text{при условии} \quad \mathbf{y} = \square\square\mathbf{s}$$

Почему L1-норма, а не L0?
Идея в том, что мы хотим найти *самое разреженное* решение (минимизировать $\|\mathbf{s}\|_0$ — число ненулевых элементов). Но минимизация L0-нормы — это NP-трудная задача (перебор всех комбинаций).
Магия compressed sensing в том, что при определённых условиях на матрицу $\square$ (так называемая Restricted Isometry Property, RIP), минимизация L1-нормы даёт *точно тот же результат*, что и минимизация L0! А L1-минимизация — это выпуклая задача, которая решается эффективно (это линейное программирование).

17.6.3 Условия применимости

Для успешного восстановления нужно два условия:

1. **Разреженность:** Сигнал должен быть разрежен в некотором базисе (например, спектр ЯМР — это набор дельта-функций, то есть очень разрежен в частотной области).

2. **Некоррелированность:** Матрица измерений □ должна быть “некоррелирована” с базисом □. На практике это означает, что точки измерений должны быть выбраны **случайно** (или псевдослучайно).

17.7 Примеры Compressed Sensing в химии и за её пределами

17.7.1 Быстрая МРТ (Magnetic Resonance Imaging)

Одно из самых известных применений — ускорение МРТ в медицине. Классическая МРТ требует длительного сканирования (k-space заполняется построчно). С compressed sensing можно измерить только **случайные** линии k-space и восстановить полное изображение через L1-минимизацию. Это сокращает время сканирования в 5–10 раз — критично для пациентов, которые не могут долго лежать неподвижно.

17.7.2 Разреженная ЯМР-спектроскопия

В многомерной ЯМР белков compressed sensing позволяет:

- Измерять только случайное подмножество точек в многомерном k-space,
- Восстанавливать полный спектр через L1-минимизацию,
- Сокращать время эксперимента с недель до часов или минут.

Это — именно то, что автор скрипта делал 20 лет назад, и что сейчас стало стандартом в современных ЯМР-спектрометрах (методы sparse sampling, Poisson-gap sampling, и т.д.).

17.7.3 Single-Pixel Camera (камера Райса)

Удивительный пример: камера, у которой **нет матрицы пикселей**, а есть только один детектор. Изображение восстанавливается через сжатые измерения с использованием DMD (Digital Micromirror Device). Это работает, потому что реальные изображения разрежены в вейвлет-базисе.

17.7.4 Масс-спектрометрия

В масс-спектрометрии compressed sensing используется для ускорения FT-ICR (Fourier Transform Ion Cyclotron Resonance) и Orbitrap — можно измерять FID короче и всё равно получить высокое разрешение.

17.7.5 Сжатие данных

JPEG использует дискретное косинусное преобразование (близкий родственник Фурье), и изображения разрежены в этом базисе — поэтому JPEG так хорошо сжимает. Compressed sensing — это следующая ступень: мы не просто сжимаем уже измеренные данные, а сразу измеряем меньше.

17.8 Связь с предыдущими главами

Давайте посмотрим, как compressed sensing связывает всё, что мы прошли:

- **Линейная алгебра:** Мы решаем переопределённую систему □□**s** = **y** через минимизацию L1-нормы.
- **Обусловленность:** Матрица □ должна быть хорошо обусловлена и удовлетворять Restricted Isometry Property (RIP).
- **SVD и тензоры:** Для многомерных данных используем тензорные разложения (PARAFAC), которые дают единственность без ортогональности.
- **Нелинейная оптимизация:** L1-минимизация — это выпуклая задача, но не гладкая (в нуле нет производной). Используем методы типа ISTA (Iterative Shrinkage-Thresholding Algorithm) или ADMM.
- **FFT:** Оператор □ часто — это преобразование Фурье, и мы используем FFT для быстрого умножения.

Главный вывод

Compressed sensing — это не просто ещё один алгоритм. Это *философия измерений*: если мы знаем, что сигнал простой (разреженный), то мы можем измерять его существенно меньше, чем требует классическая теория. Это революционизировало МРТ, ЯМР-спектроскопию, масс-спектрометрию и многие другие области. И всё это базируется на красивой математике: выпуклой оптимизации, теории вероятностей (случайные матрицы) и линейной алгебре.

18 Сжатие информации: от архиваторов до JPEG

18.1 Два мира сжатия

Сжатие информации — это одна из самых практичных областей прикладной математики. Мы постоянно сжимаем и разжимаем данные: архивы, картинки, видео, музыку. Но не все сжатия одинаковы.

Есть два принципиально разных подхода:

- **Сжатие без потерь (lossless):** Мы можем восстановить исходные данные *бит в бит*. Примеры: ZIP, PNG, FLAC.
- **Сжатие с потерями (lossy):** Мы жертвуем частью информации ради большего сжатия. Примеры: JPEG, MP3, MP4.

18.2 Информационная энтропия Шеннона

Прежде чем говорить о методах, давайте поймём, *сколько* вообще можно сжать данные.

Клод Шеннон в 1948 году ввёл понятие **информационной энтропии**. Если у нас есть алфавит из n символов, и символ i встречается с вероятностью p_i, то энтропия (среднее количество информации на символ) равна:

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

Это — **теоретический предел** сжатия без потерь. Мы не можем сжать данные сильнее, чем H бит на символ.

Пример

В английском тексте буква “e” встречается чаще всего ($p \approx 0.13$), а “z” — очень редко ($p \approx 0.001$). Если бы мы кодировали все буквы одинаково (8 бит на символ), мы бы тратили слишком много бит на редкие буквы. Энтропия английского текста — около 4.2 бит на символ, значит, теоретически мы можем сжать его примерно в 2 раза.

18.3 Сжатие без потерь: кодирование Хаффмана

Идея проста: замечаем, что какие-то символы и последовательности по 2–3 символа встречаются чаще, заводим таблицу таких символов, и чем чаще встречается какой-то символ или последовательность, тем меньше бит используем для его кодирования.

18.3.1 Алгоритм Хаффмана

1. Подсчитываем частоты всех символов в тексте.
2. Строим бинарное дерево: два самых редких символа объединяем в узел с суммарной частотой, повторяем, пока не получим одно дерево.
3. Код символа — это путь от корня до листа (0 — влево, 1 — вправо).

Частые символы оказываются ближе к корню — их коды короче. Редкие — дальше, коды длиннее.

Пример

Пусть у нас есть символы A (частота 0.5), B (0.25), C (0.125), D (0.125). Дерево Хаффмана даст коды:

- A: 0 (1 бит)
- B: 10 (2 бита)
- C: 110 (3 бита)
- D: 111 (3 бита)

Средняя длина: $0.5 \times 1 + 0.25 \times 2 + 0.125 \times 3 + 0.125 \times 3 = 1.75$ бит на символ — близко к энтропии!

18.3.2 Словарные методы: LZ77, LZ78, LZW

Алгоритмы семейства LZ (Lempel–Ziv) идут дальше: они ищут не только частые символы, но и **повторяющиеся последовательности**.

Идея: если мы уже видели строку “абракадабра”, то при повторном появлении мы не пишем её заново, а ссылаемся на предыдущее вхождение: “(назад на 11, длина 11)”.

Это основа форматов ZIP, GZIP, PNG.

18.4 Химический пример: поиск белковых последовательностей

А теперь — живой пример из биологии. У нас есть база данных белковых последовательностей (миллионы аминокислот), и нам нужно найти, есть ли в ней гомологи (похожие белки) к нашему новому белку.

Прямое сравнение “наш белок vs каждый белок в базе” — это слишком медленно. Нужен умный алгоритм сжатия и поиска.

18.4.1 BLAST (Basic Local Alignment Search Tool)

BLAST — это один из самых цитируемых алгоритмов в биологии. Идея:

1. Разбиваем наш белок на короткие “слова” (k-меры, обычно $k = 3$ для белков).
2. Для каждого слова ищем похожие слова в базе (с небольшими мутациями).
3. Расширяем совпадения в обе стороны, пока сходство не упадёт ниже порога.

Это — по сути, словарный метод сжатия + быстрый поиск по хеш-таблице. BLAST позволяет найти гомологи за секунды, хотя полное выравнивание заняло бы часы.

Почему это важно?

BLAST и его варианты (PSI-BLAST, BLASTP, BLASTN) — это основа современной биоинформатики. Без них не было бы ни расшифровки генома, ни разработки лекарств, ни эволюционных исследований.

18.5 Сжатие с потерями: JPEG и малоранговая аппроксимация

Теперь — сжатие с потерями. Идея: мы жертвуем частью информации, которую человеческий глаз (или ухо) всё равно не заметит.

18.5.1 JPEG через дискретное косинусное преобразование (DCT)

JPEG работает так:

1. Разбиваем картинку на блоки 8×8 пикселей.
2. Для каждого блока применяем **дискретное косинусное преобразование** (DCT) — это близкий родственник Фурье.
3. DCT раскладывает блок на 64 частотные компоненты (от низких частот — общий фон, до высоких — мелкие детали).
4. Квантуем коэффициенты: делим на матрицу квантования и округляем. Высокие частоты (мелкие детали) квантуются грубее — мы их “теряем”.
5. Оставшиеся ненулевые коэффициенты сжимаем без потерь (Хаффман).

18.5.2 Малоранговая аппроксимация через SVD

Альтернативный взгляд на сжатие картинок — через SVD. Пусть у нас есть матрица изображения $\mathbf{A} \in \mathbb{R}^{M \times N}$. SVD:

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

Мы можем аппроксимировать $\mathbf{A}$ через первые r сингулярных значений:

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

Это — **малоранговая аппроксимация** ранга r. По теореме Эккарта–Янга, это *лучшая* аппроксимация ранга r в смысле L2-нормы.

Степень сжатия

Исходная матрица: $M \times N$ чисел. Аппроксимация ранга r: $r(M+N+1)$ чисел. Если $r \ll \min(M,N)$, то сжатие огромно!
Например, для картинки 1000×1000 и $r = 50$: сжатие в $1000 \times 1000/(50 \times 2001) \approx 10$ раз.

18.6 Вейвлеты: локальные частоты

И DCT, и SVD — это глобальные преобразования: они раскладывают всю картинку по частотам. Но у картинок есть *локальные* особенности: края объектов, текстуры, точки.

Вейвлеты — это базисные функции, которые локализованы и в пространстве, и в частоте. Они позволяют анализировать сигнал на разных масштабах.

18.6.1 Вейвлет-преобразование

Вместо синусоид (как в Фурье) используем “маленькие волны” (wavelets) — функции, которые быстро затухают. Самый популярный — **вейвлет Хаара**:

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

Вейвлет-преобразование раскладывает сигнал по масштабам (частотам) и позициям. Это позволяет:

- Сжимать картинки лучше, чем JPEG (формат JPEG2000 использует вейвлеты),
- Detect edges (края объектов),
- Удалять шум, сохраняя важные детали.

Итог

Сжатие информации — это баланс между математической теорией (энтропия, SVD, вейвлеты) и психофизикой (что видит глаз, что слышит ухо). От архиваторов до JPEG — везде одна идея: найти структуру в данных и использовать её для компактного представления.

19 Сравнение изображений: от свёртки до нейросетей

19.1 Задача: найти объект на картинке

Как сравнить две картинки и понять, что на них есть один и тот же объект?

Мы уже сравнивали две последовательности, строя циркулянтную матрицу (глава про FFT). Наверное, картинки можно также сравнить? Пусть они 2D. Но не всё тут так хорошо, как кажется.

Если одна картинка относительно другой просто **сдвинута** по одной или обеим осям — да, всё будет работать. Мы можем использовать 2D-свёртку (через FFT) и найти максимум — это будет позиция объекта.

Но если есть **поворот**? Или **растяжение**? Или **изменение освещения**? Простая свёртка уже не поможет.

19.2 Edge detection: выделение границ

Первый шаг к пониманию содержимого картинки — это выделение **границ** (edges). Границы — это места, где интенсивность пикселей резко меняется.

19.2.1 Оператор Собеля

Самый простой способ — это вычислить градиент интенсивности. Оператор Собеля использует две маски 3×3 для вычисления производных по x и y:

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

где $*$ — свёртка, I — изображение.

Модуль градиента: $G = \sqrt{G_x^2 + G_y^2}$. Там, где G большой — там граница.

19.2.2 Детектор Кэнни (Canny Edge Detector)

Более продвинутый алгоритм:

1. Сглаживаем картинку гауссовым фильтром (убираем шум).
2. Вычисляем градиент (как у Собеля).
3. Подавляем немаксимумы (оставляем только самые сильные границы).
4. Применяем гистерезис: если граница выше верхнего порога — оставляем, ниже нижнего — удаляем, между — оставляем, если связана с сильной границей.

Результат — тонкие, связные линии границ.

19.3 Особые точки: ключевые точки изображения

Границы — это хорошо, но нам нужно что-то более “точечное”, чтобы сравнивать картинки. Для этого ищут **особые точки** (keypoints, interest points) — места, которые легко найти и которые устойчивы к преобразованиям.

19.3.1 Детектор Харриса (Harris Corner Detector)

Идея: ищем углы — места, где граница меняет направление. Для каждого пикселя вычисляем матрицу вторых моментов градиента:

$$\mathbf{M} = \sum_{x,y} w(x,y) \begin{pmatrix} G_x^2 & G_x G_y \\ G_x G_y & G_y^2 \end{pmatrix}$$

где $w(x,y)$ — окно (например, гауссово).

Если оба собственных значения $\mathbf{M}$ большие — это угол. Если одно большое, другое маленькое — это граница. Если оба маленькие — это плоская область.

19.3.2 SIFT (Scale-Invariant Feature Transform)

SIFT — это один из самых популярных алгоритмов (Lowe, 1999). Он находит особые точки, которые устойчивы к:

- Масштабу (объект может быть ближе или дальше),
- Повороту,
- Изменению освещения,
- Небольшим изменениям ракурса.

Алгоритм:

1. Строим **пирамиду масштабов**: размываем картинку гауссовым фильтром с разным σ и уменьшаем размер.
2. Ищем **экстремумы** в разности гауссиан (DoG — Difference of Gaussians) — это приближение лапласиана.
3. Для каждой особой точки вычисляем **дескриптор**: гистограмму градиентов в окрестности 16×16 пикселей. Это вектор из 128 чисел.
4. Сравниваем дескрипторы между картинками (евклидово расстояние).

19.3.3 SURF, ORB, AKAZE

Есть много улучшений SIFT:

- **SURF** (Speeded-Up Robust Features) — быстрее, использует интегральные изображения.
- **ORB** (Oriented FAST and Rotated BRIEF) — очень быстрый, свободен от патентов.
- **AKAZE** — использует нелинейные диффузионные масштабы.

19.4 Нахождение трансформации: RANSAC

Мы нашли особые точки на двух картинках и сопоставили их (feature matching). Но не все сопоставления правильные — есть выбросы (outliers).

Как найти трансформацию (поворот, масштаб, сдвиг), которая переводит одну картинку в другую, если у нас есть выбросы?

19.4.1 RANSAC (Random Sample Consensus)

RANSAC — это элегантный алгоритм для робастной оценки параметров:

1. Случайно выбираем минимальное число точек (для аффинной трансформации — 3 пары точек).
2. По этим точкам вычисляем параметры трансформации.
3. Считаем, сколько *всех* точек согласуются с этой трансформацией (inliers) — расстояние меньше порога.
4. Повторяем N раз, выбираем трансформацию с максимальным числом inliers.
5. Финально уточняем параметры по всем inliers (метод наименьших квадратов).

Почему RANSAC работает?

Если доля выбросов не слишком велика (скажем, $< 50\%$), то вероятность случайно выбрать только inliers — ненулевая. Повторяя достаточно раз, мы почти гарантированно найдём "правильную" трансформацию.

19.5 Методы роя частиц и оптимизация

Иногда у нас нет явных особых точек, но мы знаем, что одна картинка — это трансформированная версия другой. Как найти параметры трансформации?

Это — задача оптимизации: мы максимизируем **метрику сходства** (например, взаимную информацию — mutual information) по параметрам трансформации.

19.5.1 Particle Swarm Optimization (PSO)

Метод роя частиц — это эвристический метод оптимизации, вдохновлённый поведением стаи птиц или косяка рыб:

1. Инициализируем "рой" из N частиц — каждая частица — это набор параметров трансформации (например, угол поворота, масштаб, сдвиги по x и y).
2. Каждая частица имеет "скорость" и "позицию".
3. На каждом шаге частица "запоминает" свою лучшую позицию (personal best) и знает лучшую позицию всего роя (global best).
4. Скорость частицы обновляется как взвешенная сумма: инерция (продолжать двигаться в том же направлении), когнитивная составляющая (стремление к personal best) и социальная составляющая (стремление к global best).
5. Повторяем до сходимости.

PSO особенно полезен, когда:

- Функция сходства негладкая или имеет много локальных максимумов,
- Пространство параметров большое (например, 3D-трансформация с 12 параметрами),
- Мы не можем вычислить градиент (например, взаимная информация не дифференцируема).

19.6 Нейросети: от ручных признаков к обучаемым

Все методы, которые мы обсудили выше (Собель, Харрис, SIFT) — это **ручное конструирование признаков** (hand-crafted features). Мы сами придумываем, что такое "граница", "угол", "интересная точка", и как из них построить дескриптор.

Но что, если мы позволим компьютеру *самому* научиться находить признаки?

19.6.1 Свёрточные нейронные сети (CNN)

Convolutional Neural Networks — это специальный тип нейросетей, созданный для работы с изображениями. Идея:

1. **Свёрточные слои:** Применяем набор обучаемых фильтров (как маски Собеля, но параметры учатся из данных). Каждый фильтр выделяет свой признак — от простых границ на первых слоях до сложных текстур и частей объектов на глубоких слоях.

2. **Pooling слои:** Уменьшаем размер карты признаков (max pooling — берём максимум в окне 2×2). Это даёт инвариантность к малым сдвигам.

3. **Полносвязные слои:** В конце — обычная нейросеть, которая классифицирует или регрессирует.

19.6.2 Иерархия признаков

Что удивительно, CNN сами учатся той же иерархии, которую мы конструировали вручную:

- **Первые слои:** Простые границы, градиенты (как Собель),
- **Средние слои:** Текстуры, углы, простые формы (как SIFT-дескрипторы),
- **Глубокие слои:** Части объектов (глаза, колёса, листья),
- **Последние слои:** Целые объекты (лицо, машина, дерево).

Это — **обучение признаков** (feature learning) вместо ручного конструирования.

19.7 Предобученные модели и transfer learning

Обучение CNN с нуля требует миллионов размеченных изображений и дней вычислений на GPU. Но есть хитрость — **transfer learning** (перенос обучения).

Идея: берём сеть, предобученную на огромной базе данных (например, ImageNet — 1.4 миллиона изображений, 1000 классов), и используем её как **экстрактор признаков**.

19.7.1 Популярные архитектуры

- **VGG (2014):** Простая, глубокая (16–19 слоёв), хорошие признаки.
- **ResNet (2015):** Очень глубокая (до 152 слоёв) с “skip connections” — решает проблему затухающих градиентов.
- **EfficientNet (2019):** Оптимизирована по всем параметрам (точность, скорость, размер).
- **Vision Transformer (ViT, 2020):** Использует архитектуру трансформера (как в NLP) для изображений.

19.7.2 Как использовать предобученную модель

1. **Feature extraction:** Берём предобученную сеть, отрезаем последний классификационный слой, и используем предпоследний слой как экстрактор признаков. Получаем вектор из, скажем, 2048 чисел для каждого изображения. Сравниваем векторы (косинусное расстояние).

2. **Fine-tuning:** Берём предобученную сеть и дообучаем её на своих данных (например, на микроскопических снимках клеток). Первые слои “замораживаем” (они уже хорошо работают), последние — обучаем.

3. **Siamese networks:** Две одинаковые сети с общими весами, обучаемые так, чтобы похожие изображения имели близкие векторы признаков, а разные — далёкие.

19.8 Применение в химии и биологии

19.8.1 Крио-электронная микроскопия (cryo-EM)

В cryo-EM мы получаем тысячи 2D-проекций белковых молекул в случайных ориентациях. Задача:

1. Найти все молекулы на микрофотографиях (particle picking) — это задача детекции объектов,
2. Определить ориентацию каждой проекции — это задача сравнения изображений,
3. Реконструировать 3D-структуру — это обратная задача томографии.

Современные программы (RELION, cryoSPARC) используют CNN для particle picking и классификации. Это позволило достичь “resolution revolution” — определять структуры белков с атомарным разрешением.

19.8.2 Микроскопия и анализ клеток

В биологических исследованиях нужно:

- Считать клетки на микроскопических снимках,
- Классифицировать типы клеток,
- Отслеживать движение клеток во времени,
- Находить аномалии (раковые клетки).

CNN (особенно U-Net для сегментации) — это стандарт в области.

19.8.3 Хемотомика и drug discovery

В поиске лекарств нужно сравнивать молекулярные структуры. Но молекулы — это не картинки! Однако их можно представить как 2D-изображения (молекулярные графы, отрисованные на сетке) и использовать CNN для предсказания активности.

19.9 Связь с предыдущими главами

Давайте посмотрим, как сравнение изображений связывает всё, что мы прошли:

- **Линейная алгебра:** Свёртка — это умножение матрицы на вектор (или тензор). CNN — это последовательность линейных операций с нелинейностями.
- **SVD и малоранговая аппроксимация:** CNN можно сжимать через SVD (low-rank factorization свёрточных ядер).
- **FFT:** Свёртка через FFT — это $\mathcal{O}(N \log N)$ вместо $\mathcal{O}(N^2)$. В CNN это критично для скорости.
- **Нелинейная оптимизация:** Обучение CNN — это минимизация функции потерь (cross-entropy, MSE) через backpropagation + SGD/Adam (варианты градиентного спуска).
- **Compressed sensing:** В MRI и cryo-EM мы используем сжатые измерения + CNN для реконструкции.

Главный вывод

Сравнение изображений — это от простого к сложному:

1. Свёртка (для сдвигов) — через FFT,
2. Особые точки (SIFT) + RANSAC (для поворотов и масштаба),
3. Оптимизация (PSO) для сложных трансформаций,
4. Нейросети (CNN) — для семантического сходства.

Каждый метод — это компромисс между универсальностью, скоростью и точностью. И все они базируются на математике, которую мы прошли: линейная алгебра, оптимизация, Фурье-анализ, теория вероятностей.

20 Машинное обучение: от перцептрона до AlphaFold

20.1 Что такое машинное обучение?

Машинное обучение (Machine Learning, ML) — это раздел искусственного интеллекта, где компьютер *сам учится* решать задачи на основе данных, а не по жёстко прописанным правилам.

Вместо того чтобы писать программу “если температура > 100, то вода кипит”, мы даём компьютеру тысячи примеров “температура — состояние воды” и просим найти закономерность.

20.1.1 Три типа обучения

- **Обучение с учителем (supervised):** Есть размеченные данные (вход $\rightarrow$ выход). Задача — научиться предсказывать выход для новых входов. Примеры: классификация изображений, предсказание свойств молекул.
- **Обучение без учителя (unsupervised):** Данные без разметки. Задача — найти структуру, кластеры, скрытые закономерности. Примеры: кластеризация спектров, PCA.
- **Обучение с подкреплением (reinforcement):** Агент взаимодействует со средой и получает награды/штрафы. Задача — выучить стратегию, максимизирующую награду. Примеры: игра в шахматы, управление роботом.

20.2 От перцептрона к нейросетям

20.2.1 Перцептрон (1958, Розенблатт)

Самая простая “нейросеть” — это перцептрон. Он вычисляет:

$$y = \sigma\left(\sum_{i=1}^{n} w_i x_i + b\right)$$

где x_i — входы, w_i — веса, b — смещение, σ — функция активации (например, ступенька или сигмоида).

Перцептрон может решать только **линейно разделимые** задачи (например, логическое ИЛИ). Для XOR (исключающее ИЛИ) одного перцептрона недостаточно — нужна многослойная сеть.

20.2.2 Многослойный перцептрон (MLP)

Сеть из нескольких слоёв: входной слой, один или несколько скрытых слоёв, выходной слой. Каждый нейрон связан со всеми нейронами следующего слоя.

Формула для l-го слоя:

$$\mathbf{h}^{(l)} = \sigma\left(\mathbf{W}^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}\right)$$

где $\mathbf{W}^{(l)}$ — матрица весов, $\mathbf{b}^{(l)}$ — вектор смещений, σ — нелинейная функция активации (ReLU, tanh, sigmoid).

20.2.3 Обучение: backpropagation

Как учить нейросеть? Минимизировать функцию потерь L (например, MSE для регрессии или cross-entropy для классификации) через градиентный спуск.

Backpropagation — это эффективный алгоритм вычисления градиентов через цепное правило. Мы идём от выхода к входу, вычисляя $\partial L/\partial w_{ij}$ для каждого веса, и обновляем веса:

$$w_{ij} \leftarrow w_{ij} - \eta\frac{\partial L}{\partial w_{ij}}$$

где η — learning rate.

20.3 Свёрточные нейросети (CNN)

Для изображений MLP неэффективен: слишком много параметров (каждый пиксель связан со всеми нейронами).

CNN используют **свёрточные слои**: маленький фильтр (например, 3×3) скользит по изображению и вычисляет свёртку. Это даёт:

- **Локальность связей:** Каждый нейрон смотрит только на маленькую область,
- **Разделение весов:** Один и тот же фильтр применяется ко всему изображению,
- **Инвариантность к сдвигам:** Объект найдётся где угодно.

Популярные архитектуры: LeNet (1998), AlexNet (2012), VGG (2014), ResNet (2015), EfficientNet (2019).

20.4 Рекуррентные нейросети (RNN)

Для последовательностей (текст, временные ряды) нужны сети с **памятью**. RNN передают скрытое состояние от одного шага к другому:

$$\mathbf{h}_t = \sigma\left(\mathbf{W}_h\mathbf{h}_{t-1} + \mathbf{W}_x\mathbf{x}_t + \mathbf{b}\right)$$

Проблема обычных RNN — **затухающие градиенты** (vanishing gradients): сеть плохо учится на длинных последовательностях.

Решения:

- **LSTM** (Long Short-Term Memory, 1997): Добавляет “ворота” (gates), которые контролируют поток информации.
- **GRU** (Gated Recurrent Unit, 2014): Упрощённая версия LSTM.

20.5 Революция: трансформеры и attention

В 2017 году вышла статья “Attention Is All You Need” (Vaswani et al.), которая перевернула NLP и не только.

20.5.1 Механизм внимания (Attention)

Идея: вместо того чтобы сжимать всю последовательность в один вектор (как в RNN), мы позволяем каждому элементу "смотреть" на все остальные элементы и взвешивать их по важности.

Для входной последовательности $\mathbf{x}_1, \ldots, \mathbf{x}_n$ вычисляем:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V}$$

где $\mathbf{Q}$ (query), $\mathbf{K}$ (key), $\mathbf{V}$ (value) — линейные проекции входов.

20.5.2 Transformer

Архитектура Transformer полностью отказывается от рекуррентности и свёрток. Вместо этого используется только механизм внимания (self-attention) и полносвязные слои.

Ключевые компоненты:

- **Multi-Head Attention:** Несколько параллельных механизмов внимания, каждый из которых учится фокусироваться на разных аспектах данных.
- **Positional Encoding:** Поскольку Transformer не имеет рекуррентности, он не знает порядок элементов. Добавляем позиционные кодировки (синусы/косинусы или обучаемые векторы).
- **Layer Normalization:** Нормализация активаций для стабильности обучения.
- **Residual Connections:** "Skip connections" для борьбы с затухающими градиентами (как в ResNet).

Transformer стал основой для:

- **BERT** (2018): Bidirectional Encoder Representations from Transformers — предобучение на маскированных словах.
- **GPT** (2018–2023): Generative Pre-trained Transformer — авторегрессивная генерация текста. GPT-3 (175 млрд параметров), GPT-4 (мультимодальный).
- **T5, BART**: Encoder-decoder архитектуры для translation, summarization.

20.6 Машинное обучение в химии: эволюция

20.6.1 Эра QSAR (1990-е – 2010-е)

QSAR (Quantitative Structure-Activity Relationship) — это классический подход: вычисляем **молекулярные дескрипторы** (числовые характеристики молекулы) и строим регрессию/классификацию.

Дескрипторы:

- Физико-химические: молярная масса, logP (липофильность), полярная поверхность,
- Топологические: индексы связности, формы графа молекулы,
- Электронные: заряды атомов, энергии орбиталей,
- 3D-дескрипторы: моменты инерции, радиусы gyration.

Методы: PLS (Partial Least Squares), Random Forest, SVM (Support Vector Machines).

Пример

Предсказание токсичности молекулы: вычисляем 200 дескрипторов, собираем базу данных из 10 000 молекул с известной токсичностью, обучаем Random Forest. Точность — 80–85%.

20.6.2 Графовые нейросети (2015 – настоящее время)

Молекула — это **граф**: атомы — узлы, связи — рёбра. Графовые нейросети (Graph Neural Networks, GNN) работают напрямую с графами.

Message Passing Neural Networks (MPNN):

1. Каждый атом имеет начальное представление (вектор признаков: тип атома, заряд, гибридизация).
2. На каждом шаге атомы "обмениваются сообщениями" с соседями через связи.
3. После K шагов каждый атом "знает" о своей окрестности радиуса K связей.
4. Финальное представление молекулы — агрегация всех атомных представлений.

Популярные архитектуры:

- **GCN** (Graph Convolutional Network): Аналог свёртки для графов.
- **GAT** (Graph Attention Network): Attention между атомами.
- **SchNet, DimeNet, SphereNet**: Учитывают 3D-координаты атомов и углы.

Почему GNN так хороши для химии?

- **Инвариантность к перестановкам:** Порядок атомов не важен,
- **Учёт структуры:** GNN видят связи и геометрию,
- **Интерпретируемость:** Можно посмотреть, какие атомы/связи важны для предсказания.

20.6.3 Предсказание свойств молекул

Современные must-have модели для химика:

Задача	Модель	Точность
Энергия молекулы	SchNet, DimeNet++	MAE $\sim$1 kcal/mol
Сольватация	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 Революция AlphaFold: предсказание структуры белков

20.7.1 Проблема

Предсказание 3D-структуры белка по аминокислотной последовательности — это одна из grand challenges биологии. Экспериментальные методы (X-ray, cryo-EM, NMR) — дорогие и медленные.

20.7.2 AlphaFold (2020, DeepMind)

AlphaFold 2 — это прорыв, который решил проблему предсказания структуры белков с атомарной точностью.

Архитектура:

1. **Evoformer:** Transformer, который обрабатывает:
   - Множественное выравнивание последовательностей (MSA) — эволюционная информация,
   - Pair representation — попарные расстояния между остатками.
2. **Structure Module:** Итеративно строит 3D-координаты атомов, минимизируя функцию потерь.
3. **Confidence Estimation:** Предсказывает pLDDT (per-residue confidence) — насколько модель уверена в каждом остатке.

Результаты на CASP14 (Critical Assessment of Structure Prediction):

- Средний GDT_TS (Global Distance Test) — 92.4 (экспериментальная точность — $\sim$90),
- Для 2/3 белков — точность < 1 Å (атомарная точность).

20.7.3 AlphaFold 3 (2024)

Расширение на:

- Белок–лиганд комплексы,
- ДНК/РНК структуры,
- Пост-трансляционные модификации,
- Ионные взаимодействия.

20.8 Фундаментальная модель конформеров

20.8.1 Проблема конформационного пространства

Молекула — это не статичная структура. Она постоянно “дышит”, вращается вокруг связей, меняет конформацию. Для drug design критически важно знать не одну структуру, а **ансамбль** низкоэнергетических конформеров.

Классические методы:

- **Molecular Dynamics (MD):** Моделируем движение атомов во времени, но это медленно (наносекунды — микросекунды).
- **Monte Carlo:** Случайное блуждание по конформационному пространству, но неэффективно для больших молекул.
- **Systematic/Rotamer search:** Перебор всех комбинаций торсионных углов, но экспоненциальный рост.

20.8.2 ML-подход: предсказание конформеров

Современные модели учатся предсказывать распределение конформеров напрямую из данных.

GeoLDM (Geometric Latent Diffusion Models, 2023):

1. **Encoder:** Кодирует 3D-конформацию в латентное пространство.
2. **Diffusion Process:** Добавляет шум к латентным векторам (как в DALL-E, Stable Diffusion).
3. **Decoder:** Генерирует новые конформации из шума через обратный диффузионный процесс.
4. **Energy Model:** Фильтрует сгенерированные конформации по энергии.

ConfGF (Conformation Graph Factor, 2022):

- Использует GNN для предсказания 3D-координат,
- Генерирует ансамбль конформеров через sampling,
- Учитывает распределение Больцмана (низкоэнергетические конформеры вероятнее).

20.8.3 Фундаментальная модель: Uni-Mol (2023)

Uni-Mol — это “foundation model” для молекул, предобученная на миллионах конформаций.

Архитектура:

1. **3D Transformer:** Работает с атомными координатами и признаками.
2. **Pre-training tasks:**
   - Предсказание расстояний между атомами,
   - Предсказание углов и диэдральных углов,
   - Маскированное предсказание атомов (как BERT),
   - Контрастивное обучение (похожие конформеры — близко).
3. **Fine-tuning:** Дообучение на конкретных задачах (энергия, свойства, конформеры).

Результаты:

- Предсказание энергии: MAE $\sim$0.5 kcal/mol (лучше DFT для некоторых классов),
- Генерация конформеров: RMSD < 0.5 Å для 90% молекул,
- Предсказание свойств: state-of-the-art на большинстве бенчмарков.

20.9 Must-have инструменты для современного химика

20.9.1 Программное обеспечение

Инструмент	Назначение
RDKit	Хемоинформатика: дескрипторы, fingerprints, similarity
DeepChem	ML для drug discovery, GNN
PyTorch Geometric	Графовые нейросети
OpenMM	Молекулярная динамика на GPU
ASE (Atomic Simulation Environment)	Квантовая химия + ML
SchNetPack	SchNet и другие GNN для химии
AlphaFold (ColabFold)	Предсказание структуры белков
Uni-Mol	Foundation model для молекул

20.9.2 Типичный workflow

1. **Сбор данных:** PubChem, ChEMBL, PDB (Protein Data Bank).
2. **Предобработка:** RDKit для дескрипторов, генерации конформеров.
3. **Моделирование:** GNN для предсказания свойств, AlphaFold для структуры.
4. **Валидация:** Cross-validation, external test set, experimental verification.
5. **Интерпретация:** Attention weights, SHAP values для понимания предсказаний.

20.10 Связь с предыдущими главами

Давайте посмотрим, как ML связывает всё, что мы прошли:

- **Линейная алгебра:** Нейросети — это последовательность матричных умножений $\mathbf{Wx} + \mathbf{b}$. Backpropagation — это цепное правило для матриц.
- **SVD и тензоры:** Сжатие моделей через low-rank factorization. Tensor decompositions для эффективных вычислений.
- **FFT:** Свёрточные слои используют FFT для ускорения. Attention через FFT (Linear Attention).
- **Нелинейная оптимизация:** Обучение нейросетей — это минимизация loss через SGD, Adam (адаптивные методы), L-BFGS (для fine-tuning).
- **Compressed sensing:** Sparse training, pruning нейросетей.
- **Графы и тензоры:** GNN работают с графами молекул. 3D-модели используют тензоры координат.
- **Монте-Карло:** Diffusion models — это стохастические процессы. Variational inference — байесовский подход.

Главный вывод

Машинное обучение — это не замена классической химии, а **мощный инструмент**, который:

- Ускоряет вычисления в тысячи раз (предсказание свойств vs DFT),
- Открывает новые возможности (предсказание структуры белков, генерация молекул),
- Требует понимания математики (линейная алгебра, оптимизация, теория вероятностей).

Современный химик должен знать:

- Основы ML (нейросети, GNN, transformers),
- Инструменты (RDKit, PyTorch, DeepChem),
- Когда ML работает, а когда — нет (физические ограничения, интерпретируемость).

21 Современная вычислительная техника: от транзистора до суперкомпьютера

21.1 Цифры, которые стоит знать

Современный компьютер — рабочая станция — это:

- 128–1024 ГБ оперативной памяти,
- ∼500 Гфлоп/с пиковой производительности (миллиардов операций с плавающей точкой в секунду),
- Несколько процессоров (от 4 до 128 ядер).

Но вот парадокс: скорость доступа к памяти в **сотни раз** медленнее, чем скорость арифметических операций. А если доступ случайный (не последовательная запись-чтение), то ещё в **сотню раз** медленнее.

Почему так? Давайте разбираться.

21.2 Как устроен процессор

Все говорят, что в процессоре “много транзисторов”. Но что они делают?

На самом деле, процессор — это такая сущность, которая обращается к **инструкциям** и обращается к **данным**. И то, и другое — занимает место в памяти.

21.2.1 Команды процессора

Процессор зачитывает команды, которые обычно позволяют:

- Обратиться к данным в памяти и поместить их в **регистры** — временную очень-очень быструю память,
- Выполнить операцию: сложение, вычитание, умножение, сравнение,
- Совершить **переход**: по умолчанию процессор исполняет команду за командой, но если он попадает на инструкцию перехода, то он может скакнуть вперёд или назад.

Переходы бывают:

- **Безусловные:** Переход происходит всегда,
- **Условные:** Если мы до этого сравнивали числа и имеем ответ “да” или “нет”, то переход происходит только при выполнении условия.

21.2.2 SIMD: одна команда — много данных

Современная процессорная команда может работать сразу с несколькими числами. Например, одна инструкция AVX-512 может выполнить 16 пар сложений одновременно!

Это называется **SIMD** (Single Instruction, Multiple Data) — одна инструкция, множество данных. Если данные уже есть на самом процессоре, это довольно легко сделать.

Откуда взялся SIMD?

Люди заметили, что часто надо выполнять одинаковые операции: например, одинаково трансформировать пиксели для отрисовки картинки, одинаково обрабатывать элементы массива. Зачем выполнять одну и ту же инструкцию 16 раз, если можно один раз — но сразу для 16 чисел?

Но вот чтобы каждой командой выполнять такие операции с *разными* данными, нам нужна была бы очень мощная система по перекачке данных из памяти в процессор и обратно.

21.2.3 Проблема memory wall

К сожалению, такая система заняла бы в сотни и тысячи раз больше транзисторов, чем нужно для арифметического блока.

А люди заметили, что в программах мы часто работаем с каким-то небольшим набором чисел, а потом переходим на другой набор, и так далее. Сделать небольшой блок памяти с быстрым доступом — не так накладно по числу транзисторов. Вот только заставить программиста таскать туда-сюда данные в этот блок из оперативной памяти и обратно — будет сложно.

Поэтому люди автоматизировали это — и появился **процессорный кэш**. Это быстрая память, её немного, но процессор сам "запоминает", что ты используешь сейчас, и некоторое время держит эти данные у себя в кэше, с расчётом, что далее они, может быть, будут тебе снова нужны.

21.3 Иерархия памяти

Современный компьютер — это многослойная структура памяти:

Уровень	Размер	Скорость	Описание
Регистры	32–128 × 64 бит	∼0.3 нс	Ячейки внутри процессора, с которыми он работает напрямую
L1 кэш	∼32–64 КБ	∼1 нс	Отдельно для инструкций и данных, свой для каждого ядра
L2 кэш	∼256 КБ – 1 МБ	∼3–5 нс	Свой для каждого ядра или общий на пару ядер
L3 кэш	∼8–64 МБ	∼10–20 нс	Общий для всех ядер процессора
ОЗУ (DRAM)	128–1024 ГБ	∼50–100 нс	Основная память
Диск (SSD)	1–10 ТБ	∼10–100 мкс	Долговременное хранение

Давайте сравним, сколько за одну наносекунду в среднем может сделать современная рабочая станция:

- **Арифметика:** Все её процессоры, с учётом длинных SIMD-инструкций, могут выполнить около **1000 операций** с плавающей точкой.
- **L1 кэш:** Можно прочитать и записать по 2–3 числа на каждом процессоре, суммарно ∼200 чисел.
- **L2/L3 кэш:** Эта величина падает в десятки раз.
- **ОЗУ:** Только единицы чтений и записей в наносекунду, и то — если происходит большими блоками по 512 байт примерно последовательно.
- **Случайный доступ:** Если каждый раз обращаемся к разным блокам, скорость падает ещё в десятки раз — может потребоваться несколько наносекунд на одно число.

Memory Wall

Случайный доступ к памяти в **десятки тысяч раз** медленнее реальных вычислительных мощностей современного процессора!
Это — главный bottleneck ("бутылочное горлышко") современных вычислений. Именно поэтому:

- Матричные алгоритмы работают быстрее, если данные расположены "плотно" в памяти,

- Разреженные матрицы — медленные (много случайных обращений),
- Кэш-ориентированные алгоритмы — это отдельная наука.

21.4 Графические процессоры (GPU)

Но людям этого было мало. Когда они рисовали на экране 3D-объекты компьютерной графики, они заметили, что всё это можно сделать почти полностью **параллельно**.

Условно: если бы у нас было 1024 процессора, мы бы разбили весь экран на сетку квадратиков размера 32×32, и каждый процессор рисовал бы в своём квадратике. Так появились графические карты — в них есть уйма небольших специализированных процессоров, которые работают полностью параллельно.

21.4.1 От графики к вычислениям

В конце 90-х такие графические карты использовались для ускорения отображения графики. Примерно в 2005 году люди стали выпускать такие карты так, чтобы на них можно было исполнять небольшие куски кода на тысячах таких небольших процессоров — так появились **CUDA** (NVIDIA) и **OpenCL** (открытый стандарт).

К 2010 году все численные методы начали перетекать на графические карты. Постепенно люди к single и double арифметике с плавающей точкой добавляли так называемые **половинчатые** (FP16) и **четвертинные** (FP8, INT8) плавающие точки — так как с ними было быстрее выполнять вычисления и они меньше занимали памяти.

В современных NVIDIA графических картах (H100, B200) имеется возможность выполнить около **миллиона операций** с такими маленькими объектами за ту же одну наносекунду — и это позволило использовать такие графические карты для современных систем искусственного интеллекта.

21.4.2 Архитектура GPU

Современный GPU (например, NVIDIA H100) — это:

- **Streaming Multiprocessors (SM):** $\sim$132 мультипроцессора,
- **CUDA cores:** $\sim$16 896 ядер для FP32 арифметики,
- **Tensor cores:** $\sim$528 специализированных ядер для матричных операций (FP16, FP8, INT8),
- **Память:** 80 ГБ HBM3 (High Bandwidth Memory) с пропускной способностью $\sim$3 ТБ/с.

21.4.3 Треды и warps

GPU работает с **тредами** (потоками). Тысячи тредов выполняются параллельно, но организованы они в группы:

- **Warp:** Группа из 32 тредов, которые выполняются **синхронно** — все 32 треда выполняют одну и ту же инструкцию в один и тот же момент времени.
- **Block:** Группа варпов (до 1024 тредов), которые могут обмениваться данными через общую **shared memory**.
- **Grid:** Все блоки, запущенные на GPU.

Правило эффективности GPU

Чтобы GPU работал эффективно, программы на каждом мультипроцессоре для каждого треда должны **не расползаться**. Это значит:

- Все 32 треда в варпе должны выполнять *одну и ту же* инструкцию (без условных переходов, которые разделяют треды — “warp divergence”),
- Все треды должны обращаться к *последовательным* адресам памяти (coalesced memory access),
- Должно быть достаточно тредов, чтобы “спрятать” задержки памяти.

Если эти условия не выполнены — GPU работает в разы медленнее.

21.5 Параллельные вычисления: таксономия Флинна

В 1966 году Майкл Флинн предложил классификацию параллельных архитектур по потокам инструкций и данных:

Тип	Описание	Пример
SISD	Single Instruction, Single Data. Один поток инструкций, один поток данных. Классический последовательный процессор.	Старые CPU
SIMD	Single Instruction, Multiple Data. Одна инструкция применяется к множеству данных.	Векторные инструкции (AVX), GPU
MISD	Multiple Instruction, Single Data. Разные инструкции к одним данным. Редко встречается.	Некоторые специализированные архитектуры
MIMD	Multiple Instruction, Multiple Data. Множество процессоров, каждый выполняет свои инструкции над своими данными.	Современные мультипроцессоры, кластеры

21.5.1 Общая и распределённая память

В MIMD-системах есть два подхода:

- **Общая память (shared memory):** Все процессоры имеют доступ к единой памяти. Пример: многоядерный CPU. Легко программировать (все видят одни данные), но сложно масштабировать (конфликты доступа к памяти).
- **Распределённая память (distributed memory):** Каждый процессор имеет свою локальную память, обмен данными — через сеть. Пример: кластеры, суперкомпьютеры. Сложнее программировать (нужно явно передавать данные — MPI), но масштабируется до миллионов ядер.

21.6 TOP500: самые мощные суперкомпьютеры мира

Список TOP500 — это рейтинг 500 самых мощных суперкомпьютеров мира, обновляемый дважды в год (июнь и ноябрь). Производительность измеряется в LINPACK benchmark — решении большой системы линейных уравнений.

21.6.1 Современные лидеры (2024–2025)

Имя	Место	Производительность	Архитектура
Frontier	#1	∼1.2 Эфлоп/с	AMD EPYC + AMD Instinct MI250X
Aurora	#2	∼1 Эфлоп/с	Intel Xeon + Intel Max PVC
Eagle	#3	∼561 Пфлоп/с	AMD EPYC + NVIDIA H100
Fugaku	#4	∼442 Пфлоп/с	Fujitsu A64FX (ARM)
LUMI	#5	∼231 Пфлоп/с	AMD EPYC + AMD MI250X

Масштаб

1 Эфлоп/с = 10^{18} операций в секунду. Frontier — это ∼1.2 миллиона миллиардов операций в секунду. Для сравнения: ваш ноутбук — ∼0.1 Тфлоп/с = 10^{11} операций в секунду. Разница — в 10 миллионов раз!

21.6.2 Как устроен современный суперкомпьютер

Типичная архитектура:

1. **Узлы (nodes):** ∼10 000–100 000 серверов, каждый с 2–4 CPU + 4–8 GPU,
2. **Сеть:** Высокоскоростная interconnect (InfiniBand, Slingshot) с пропускной способностью 200–400 Гбит/с на узел,
3. **Хранилище:** Параллельная файловая система (Lustre, GPFS) с пропускной способностью ∼1–10 ТБ/с,
4. **Охлаждение:** Жидкостное охлаждение (вода или даже двухфазное), так как мощность — 20–50 МВт.

21.7 Оцифровщики: мост между аналоговым и цифровым миром

Но процессор — он ведь где-то должен брать для вычисления данные. И тут на сцену выходят **аналого-цифровые преобразователи** (ADC — Analog-to-Digital Converters).

Они определяются двумя параметрами:

- **Точность (разрядность):** Сколько бит используется для кодирования одного отсчёта,
- **Скорость (частота дискретизации):** Сколько отсчётов в секунду.

21.7.1 Компромисс: точность vs скорость

Есть фундаментальный компромисс: чем выше точность, тем ниже скорость.

Разрядность	Скорость	Минимальный бит	Применение
12 бит	∼10 Гвыб/с	∼500 мкВ	Осциллографы, быстрая съёмка
16 бит	∼1 Гвыб/с	∼10–50 мкВ	Аудио, ЯМР, точные измерения
24 бит	∼4 Мвыб/с	∼100–500 нВ	Аудио высокого разрешения, сейсмография

Почему не измеряют нановольты напрямую?

24 бита дают диапазон десятков и сотен нановольт. Но нановольты никто не измеряет напрямую — радиолокационные помехи от всего вокруг составляют около десятков микровольт! То есть измеряют там обычно *токи*, а не вольты — или используют специальные

экранированные камеры.

21.8 Физика переключений: от транзистора до мегавольт

В современном процессоре всё работает, включая-выключая транзисторы на примерно 0.7 В. Скорость такого переключения — десятки гигагерц. Но вот токи там — очень маленькие (микроамперы на транзистор).

21.8.1 Компромисс: напряжение vs скорость

Если увеличивать напряжение, то скорость переключения начинает падать:

Напряжение	Время переключения	Технология
0.7 В	∼0.05 нс (20 ГГц)	Современные CMOS транзисторы
40 В	∼3–5 нс	Силовые MOSFET (обычные номиналы)
1000 В	∼3–5 нс	Силовые MOSF (топовые номиналы)
4.5–5 кВ	∼20–30 нс	IGBT, высоковольтные MOSFET
>10 кВ	∼100 нс – 1 мкс	Один полупроводник уже не справляется
1 МВ	∼100–500 мкс	Лампы, сборки полупроводников, умножители

Хотя современные CMOS транзисторы переключаются на частотах около 20 ГГц, для выполнения одного процессорного такта надо иметь запас в несколько таких переключений, поэтому тактовая частота процессоров находится в диапазоне около 3-4 ГГц

21.8.2 Предел нарастания напряжения

Выше ∼5 кВ один полупроводник уже не справляется — надо переходить на:

- **Старые добрые лампы** (!) — вакуумные триоды, тиратроны,
- **Сборки полупроводников** — последовательное включение нескольких транзисторов,
- **Умножители напряжения** — каскадные схемы с лампами или разрядниками.

Создать даже 1 мегавольт — это не сильно сложно (разрядник Маркса, каскады Кокрофта–Уолтона). Другое дело — включить-выключить этот мегавольт за наносекунду. На это нужны реально сотни микросекунд.

Фундаментальный предел

Фактически $\sim 10^{12}$ В/с — это современный предел нарастания напряжения практически во всём диапазоне возможных напряжений.
Это — физическое ограничение, связанное с:

- Скоростью движения носителей заряда в полупроводниках (предел $\sim 10^7$ см/с),
- Паразитными ёмкостями и индуктивностями,
- Скоростью распространения электромагнитных волн в среде.

21.9 Связь с предыдущими главами

Давайте посмотрим, как всё, что мы прошли, связано с “железом”:

- **Линейная алгебра:** Матричные операции — это основа всех вычислений. GPU оптимизированы именно под них (Tensor Cores).
- **FFT:** Быстрое преобразование Фурье — это $\mathcal{O}(N \log N)$ операций, и оно критично для ЯМР, обработки сигналов, сжатия данных. GPU ускоряют FFT в десятки раз.
- **Итерационные методы:** Метод сопряжённых градиентов, GMRES — это основа решения больших разреженных систем. Они работают на суперкомпьютерах с распределённой памятью (MPI + OpenMP + CUDA).
- **Машинное обучение:** Обучение нейросетей — это миллиарды матричных умножений. Без GPU и Tensor Cores современные модели (GPT-4, AlphaFold) были бы невозможны.
- **Compressed sensing:** L1-минимизация — это итерационный метод, который требует тысяч итераций. GPU ускоряют его в 100–1000 раз.
- **Оцифровщики:** ADC — это мост между аналоговым миром (спектры, сигналы) и цифровым (процессор). Точность ADC определяет, какую математику мы можем применить.

Главный вывод

Современная вычислительная техника — это:

- **Иерархия памяти:** Регистры → L1 → L2 → L3 → ОЗУ → диск. Каждая ступень — в 10–100 раз медленнее, но в 10–1000 раз больше.
- **Параллелизм:** SIMD (векторные инструкции), многоядерность (CPU), массовый параллелизм (GPU), кластеры (суперкомпьютеры).
- **Специализация:** Tensor Cores для AI, FPGA для специфичных задач, ASIC для майнинга.
- **Физические ограничения:** Memory wall, power wall, speed of light — всё это ограничивает рост производительности.

Для химика это значит:

- Понимать, где “узкое горлышко” (память или вычисления),
- Уметь писать код, который эффективно использует кэш и GPU,
- Знать, когда задача решается на рабочей станции, а когда — нужен суперкомпьютер.

22 Послесловие: Как пользоваться этими знаниями

Ранее, после прочтения аналогичной книги, когда вы сталкиваетесь с реальной задачей, возникала необходимость имплементировать решение. Для этого надо было довольно хорошо владеть одним или несколькими языками программирования и понимать, как пользоваться сторонними программными комплексами и пакетами.

В современный век развития искусственного интеллекта всё это можно делать с поддержкой систем ИИ — и это будет значительно эффективнее. Но для этого надо немного понимать ключевые применимости языков программирования и так называемый **workflow** — как происходит разработка программ.

22.1 Два мира языков программирования

Хотя языков программирования существует огромное количество, их можно разделить на два фундаментально разных класса:

Компилируемые	**Интерпретируемые / байт-код**
C, C++, CUDA, Fortran, Rust	Python, JavaScript, Java, C#, R
Код компилируется в машинные инструкции под конкретный процессор	Код интерпретируется на лету или компилируется в байт-код
Максимальная производительность, прямой доступ к “железу”	Гибкость, кросс-платформенность, быстрая разработка
Когда использовать:	**Когда использовать:**
— Тяжёлые вычисления (DFT, MD, ML)	— Прототипирование, анализ данных
— GPU-ускорение (CUDA)	— Визуализация, отчётность
— Встраиваемые системы	— Веб-интерфейсы, скрипты

22.1.1 Почему интерпретируемые языки — это удобно

Интерпретируемые языки позволяют достигать большей гибкости при переходе от одной платформы к другой. Например, мы открываем одну и ту же веб-страницу на десктопе, на смартфоне и на планшете — и видим одинаковый контент. Этот контент обычно генерируется с использованием интерпретируемого языка JavaScript, и если бы мы должны были посылать каждый раз свой код под каждый из процессоров, то сервер должен был бы иметь все варианты такого кода заранее. Это иногда даже невозможно предусмотреть — так как даже разные Android-смартфоны могут иметь различную процессорную архитектуру.

22.1.2 Типичный workflow химика

Для того чтобы быстро отобразить какие-то результаты, часто проще воспользоваться именно интерпретируемыми языками программирования (Python — абсолютный стандарт в науке). А вот когда задача бывает сложной и требует много вычислительных ресурсов, возникает необходимость использовать именно компилируемые языки программирования.

Типичный workflow выглядит так:

1. **Прототип на Python:** Быстро проверить идею, визуализировать данные, понять, работает ли алгоритм.
2. **Оптимизация:** Если код работает, но медленно — переписать “узкие места” на C++/CUDA или использовать готовые библиотеки (NumPy, SciPy, PyTorch).
3. **Продакшн:** Если задача решается регулярно — оформить в пакет, добавить тесты, документацию.

22.2 Работа с ИИ-ассистентами

Сейчас практически любой алгоритм можно имплементировать, написав правильный промпт с помощью чатов. Но стоит заметить: каждый чат — это почти как человек. И если ему подать задачу не до конца продуманную, он может и намудрить, или, не поняв, запрограммировать что-то не совсем то.

22.2.1 Золотые правила работы с ИИ

1. **Разбивайте задачу на небольшие блоки.** Не просите “напиши программу для решения уравнения Шрёдингера”. Просите: “напиши функцию, которая вычисляет матрицу Фока для заданного базиса”.

2. **Просите тесты.** Для каждого блока просите чат писать тестовые проверочные программы, прогонять такие тесты и убеждаться, что всё в порядке.

3. **Объединяйте постепенно.** Только после того как каждый блок работает, объединяйте их для получения окончательного результата.

4. **Требуйте объяснений.** Если чат написал что-то, что вы не понимаете — попросите объяснить. Если объяснение непонятно — попросите проще. Это ваш скрипт, вы должны понимать каждую строчку.

5. **Проверяйте на известных примерах.** Если вы решаете систему уравнений — проверьте на системе, для которой знаете ответ. Если делаете Фурье-преобразование — проверьте на синусоиде с известной частотой.

Типичные ошибки

- **Слепое доверие:** ИИ может написать код, который “выглядит правильно”, но содержит тонкие ошибки (например, неправильная нормировка, неверные границы массивов, утечка памяти).
- **Игнорирование контекста:** ИИ не знает вашу конкретную задачу — вы должны объяснить её подробно, с примерами, с ограничениями.
- **Отсутствие тестов:** Код без тестов — это код, который сломается в самый неподходящий момент.

22.3 Must-have инструменты для химика

Вот минимальный набор инструментов, которые вы должны знать:

Язык	Применение
Python	Абсолютный стандарт: анализ данных, ML, визуализация, скрипты
R	Статистический анализ, биоинформатика
MATLAB	Инженерные расчёты, обработка сигналов (альтернатива Python)
C++/CUDA	Высокопроизводительные вычисления, GPU
Bash/Shell	Автоматизация, работа с кластерами

Библиотека	Назначение
NumPy, SciPy	Линейная алгебра, оптимизация, интеграция
Pandas	Работа с табличными данными
Matplotlib, Seaborn	Визуализация
PyTorch, TensorFlow	Машинное обучение, нейросети
RDKit	Хемоинформатика, молекулы
ASE, PySCF	Квантовая химия
OpenMM, GROMACS	Молекулярная динамика

22.4 Финальный совет

Математика — это не набор формул, которые надо зазубрить. Это — **способ мышления**. Это умение видеть структуру в хаосе, находить закономерности, строить модели, проверять гипотезы.

Когда вы столкнётесь с реальной задачей — не важно, будет ли это предсказание структуры белка, анализ ЯМР-спектра, оптимизация синтеза или что-то совсем новое — вспомните этот скрипт. Вспомните, что:

- Любая задача — это либо система уравнений, либо задача оптимизации, либо задача аппроксимации,
- Для любой задачи есть проверенные математические методы,
- Современные инструменты (Python, GPU, ИИ) позволяют решать задачи, которые 20 лет назад были невозможны,
- Главное — понять суть задачи, а остальное — техника.

Мы верим в вас. У вас всё получится. И если когда-нибудь этот скрипт поможет вам решить задачу, которая изменит мир — мы будем невероятно горды.

Последнее

Не бойтесь ошибаться. Не бойтесь спрашивать. Не бойтесь экспериментировать. Математика — это не экзамен, это приключение. И вы — в начале самого интересного пути.
Удачи!

С любовью,
Папа и Мама

Сентябрь 2026