Mathematik, Numerische Methoden und Signalverarbeitung

Ein Skript für Studierende der Fakultät für Chemie der TUM

Ilgis Ibragimov & Elena Ibragimova & KI*

September 2026
DOI: 10.5281/zenodo.23002397

Gewidmet unseren Töchtern Maria und Alevtina.
Möge die Mathematik für euch nicht eine Sammlung von Formeln werden,
sondern die Sprache, in der das Universum geschrieben ist.

*Korrektur- und Übersetzungsassistent

Inhaltsverzeichnis

1 Vorwort

Liebe Töchter,

wir wollten so sehr, dass auf eurem Lebensweg alles klar und einfach ist. Wir wissen, dass Mathematik oft erklärt wird, indem man den Menschen absichtlich mit Terminologie und komplizierten Formeln verwirrt — so dass man den Wald vor lauter Bäumen nicht sieht.

Dieses Skript ist unser Versuch, einen Überblick über die mathematischen und rechnerischen Ideen zu geben, die für einen modernen Chemiker, der mit experimentellen Daten, Modellierung und Berechnungen arbeitet, besonders nützlich sind, damit du ein *klares Bild* vom modernen Stand des mathematischen Wissens hast. Nicht allen Wissens — das ist in einem Buch unmöglich — sondern genau dessen, was in Chemie, Biochemie, Spektroskopie, molekularer Modellierung und Datenverarbeitung Anwendung findet.

Wir haben gemeinsam einen langen Weg zurückgelegt:

- Von den Grundlagen — Zahlenmengen, komplexe Zahlen, Vektoren und Matrizen,
- über die lineare Algebra — SVD, Kondition, Methoden zur Lösung von Systemen,
- zu nichtlinearen Problemen — Optimierung, Gradienten, Newton-Verfahren,
- durch die Signalverarbeitung — Fourier, Prony, Compressed Sensing,
- zu numerischen Methoden — Integration, finite Elemente, Basisfunktionen,
- und schließlich — zum maschinellen Lernen, zu neuronalen Netzen und zur modernen Rechentechnik.

All das ist nicht nur eine Sammlung zusammenhangloser Themen. Es ist eine **einheitliche Sprache**, in der die moderne Wissenschaft spricht. Und wenn du diese Sprache beherrschst — werden sich vor dir Türen öffnen, von denen du jetzt nicht einmal ahnst.

1.1 Wie man dieses Buch liest

Obwohl der Text so geschrieben ist, dass er möglichst einfach lesbar ist, ohne detaillierte mathematische Beweise, können manche Stellen zu abstrus formuliert sein. Das ist normal — Mathematik kommt nicht auf einmal.

Deshalb ist der Quelltext dieses Buches in LaTeX beigefügt. Wenn etwas nicht ganz klar ist:

1. Finde den entsprechenden Text im Quelltext,
2. Kopiere ihn in einen Chat (Qwen, DeepSeek, Kimi, ChatGPT, GROK, Gemini),
3. Bitte ihn, dies verständlicher zu erklären, oder stelle die Frage, die du hier nicht verstehst.

Mit Hilfe dieser Chats kann man auch den ganzen Text ins Englische oder Deutsche übersetzen — dazu muss man nur bitten, unter Beibehaltung der LaTeX-Auszeichnung zu übersetzen, und den erhaltenen Text mit dem Befehl `xelatex NumMathForChemists.tex` in eine PDF-Datei kompilieren.

Tipp

Versuche nicht, alles auf einmal zu lesen. Mathematik ist wie Musik: man muss sie “spielen”, also Aufgaben lösen, Code ausprobieren, experimentieren. Wenn ein Kapitel schwierig erscheint — komm eine Woche später darauf zurück, und du wirst überrascht sein, wie viel einfacher es geworden ist.

2 Einleitung: Bezeichnungen und die Natur der Zahlen

2.1 Die Hierarchie der Zahlenmengen

Bevor wir anfangen, über komplizierte Dinge zu sprechen, lass uns über die Sprache einig werden. In der Mathematik gruppieren wir Zahlen in Mengen, die bestimmte Eigenschaften haben. Wir verwenden die folgenden Standardbezeichnungen:

- $\mathbb{N} = \{1, 2, 3, \dots\}$ — **natürliche Zahlen**. Sie werden zum Zählen verwendet.
- $\mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$ — **ganze Zahlen**. Sie fügen die Null und negative Zahlen hinzu und erlauben es, Schulden und Vermögen, Temperaturen unter Null usw. zu beschreiben.
- $\mathbb{Q}$ — **rationale Zahlen**. Jede Zahl, die als Bruch $\frac{p}{q}$ dargestellt werden kann, wobei $p \in \mathbb{Z}$, $q \in \mathbb{Z} \setminus \{0\}$.
- $\mathbb{R}$ — **reelle Zahlen**. Sie umfassen alle rationalen sowie irrationale Zahlen (zum Beispiel $\sqrt{2}, \pi, e$), die nicht als gewöhnlicher Bruch dargestellt werden können. Sie füllen die gesamte Zahlengerade kontinuierlich aus.
- $\mathbb{C}$ — **komplexe Zahlen**. Über sie sprechen wir weiter unten ausführlich.

Verschachtelung der Mengen: $\mathbb{N} \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R} \subset \mathbb{C}$.

2.2 Zahlen im Computer

Wir sind daran gewöhnt, dass Zahlen in der Mathematik unendlich genau sind. Aber wenn wir zur **numerischen Mathematik** und zum Programmieren übergehen, ist der Computer gezwungen, Zahlen in Speicherzellen fester Größe (in Bits und Bytes) zu speichern.

- **Ganzzahlige Typen** (int8, int16, int32, int64) speichern exakte Werte aus $\mathbb{Z}$, sind aber im Bereich begrenzt (zum Beispiel kann int32 Zahlen von -2^{31} bis $2^{31} - 1$ speichern).
- **Reelle Typen** (float32, float64 / double) speichern Zahlen aus $\mathbb{R}$ im Exponentialformat (Mantisse und Exponent). Dadurch entstehen Rundungsfehler: das computerliche $0.1 + 0.2$ ist nicht immer exakt gleich 0.3. Das zu verstehen ist für numerische Methoden von entscheidender Bedeutung!

3 Wenn eine Zahl nicht genügt: Komplexe Zahlen

Unsere Welt ist so komplex, dass nicht alles durch eine einzige Zahl beschrieben werden kann. In Physik und Chemie müssen wir oft mit Entitäten arbeiten, die *zwei* Zahlen zu ihrer Beschreibung benötigen (zum Beispiel Amplitude und Phase einer Welle oder Real- und Imaginärteil einer Wellenfunktion).

3.1 Woher kommen sie?

Der einfachste und historischste Weg, "Zahlenpaare" kennenzulernen, ist der Versuch, die Gleichung zu lösen:

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

In der Menge der reellen Zahlen $\mathbb{R}$ ist das Quadrat jeder Zahl nichtnegativ. Es gibt keine Wurzel aus -1. Aber die Mathematiker (und nach ihnen die Physiker) sagten: "Was, wenn wir einfach eine Zahl *erfinden*, die mit sich selbst multipliziert -1 ergibt?".

So entstand die **imaginäre Einheit** i. Wir "ziehen nicht die Wurzel" aus -1, wir *definieren* eine neue Entität:

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

Wichtige Bemerkung: Die Regel $\sqrt{a}\sqrt{b} = \sqrt{ab}$ *gilt nur für* $a, b \geq 0$. *Deshalb schreiben wir nicht* $\sqrt{-1}\sqrt{-1} = \sqrt{(-1)(-1)} = 1$. *Wir akzeptieren einfach* $\mathrm{i}^2 = -1$ *als Tatsache.*

3.2 Algebraische Form und komplexe Ebene

Jede komplexe Zahl $z \in \mathbb{C}$ kann in der Form geschrieben werden:

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

wobei $x = \mathrm{Re}(z)$ der **Realteil** und $y = \mathrm{Im}(z)$ der **Imaginärteil** ist. Beide Teile $x, y \in \mathbb{R}$.

Geometrisch ist eine komplexe Zahl ein Punkt in der **komplexen Ebene**.

- Auf der horizontalen Achse (der Re-Achse) wird der Realteil x aufgetragen.
- Auf der vertikalen Achse (der Im-Achse) wird der Imaginärteil y aufgetragen.

Eine komplexe Zahl kann auch als **Vektor** betrachtet werden, der vom Ursprung $(0, 0)$ zum Punkt (x, y) geht.

3.3 Arithmetik komplexer Zahlen

Die **Addition** ist intuitiv klar. Wenn wir zwei Zahlen $z_1 = x_1 + \mathrm{i}y_1$ und $z_2 = x_2 + \mathrm{i}y_2$ haben, addieren wir einfach ihre Real- und Imaginärteile getrennt:

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

Geometrisch funktioniert dies wie die **Dreiecksregel** für Vektoren: Wir zeichnen zwei Vektoren vom Ursprung, verschieben den Anfang des zweiten Vektors an das Ende des ersten, und die Summe ist der Vektor vom Anfang des ersten bis zum Ende des zweiten.

Die **Multiplikation** in algebraischer Form sieht etwas komplizierter aus. Wir lösen die Klammern wie in der gewöhnlichen Algebra auf und denken daran, dass $\mathrm{i}^2 = -1$:

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

Aber um die *wahre Magie* der Multiplikation komplexer Zahlen zu verstehen, müssen wir zu einer anderen Schreibweise übergehen.

3.4 Trigonometrische und Exponentialform

Statt der Koordinaten (x, y) kann ein Punkt in der Ebene durch Polarkoordinaten angegeben werden: den Abstand vom Ursprung (den Betrag) r und den Winkel φ relativ zur positiven Richtung der Re-Achse.

Der Zusammenhang mit der algebraischen Form ist aus der Trigonometrie offensichtlich:

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

Dann nimmt die komplexe Zahl die **trigonometrische Form** an:

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

3.4.1 Eulers Formel und die Exponentialform

Hier betritt die größte Formel der Mathematik die Bühne — **Eulers Formel**:

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

Dank ihr kann jede komplexe Zahl in der unglaublich bequemen **Exponentialform** geschrieben werden:

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

3.4.2 Die geometrische Bedeutung der Multiplikation

Betrachten wir, was bei der Multiplikation zweier Zahlen in Exponentialform geschieht:

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

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

Die Schlussfolgerung, die man sich für immer merken muss: Die Multiplikation komplexer Zahlen ist die **Multiplikation ihrer Längen (Beträge)** und die **Addition ihrer Winkel (Argumente)**! Geometrisch: Die Multiplikation mit einer komplexen Zahl ist eine Drehung des Vektors um den Winkel φ und seine Streckung/Stauchung um den Faktor r.

3.5 Komplexe Exponentialfunktionen und Logarithmen

3.5.1 Eigenschaften der Exponentialfunktion

Da sich $e^{\mathrm{i}\varphi}$ wie eine gewöhnliche Exponentialfunktion verhält, gelten für sie alle Standardregeln:

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

Für eine komplexe Zahl $z = x + \mathrm{i}y$ zerfällt die Exponentialfunktion in Real- und Imaginärteil:

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

Der Betrag dieser Zahl ist e^x, und das Argument ist y.

3.5.2 Der komplexe Logarithmus

Der Logarithmus ist die zur Exponentialfunktion inverse Operation. Wenn $e^w = z$, dann ist $w = \ln z$. Sei $z = re^{\mathrm{i}\varphi}$ und $w = u + \mathrm{i}v$. Dann:

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

Durch Gleichsetzen von Beträgen und Argumenten erhalten wir:

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

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

Somit ist der **komplexe Logarithmus** mehrwertig:

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

Der Hauptwert des Logarithmus (bei $k = 0$ und $\arg z \in (-\pi, \pi]$) wird mit $\mathrm{Ln}\, z$ bezeichnet. Die Mehrwertigkeit entsteht dadurch, dass eine Drehung um 2π uns zum selben Punkt in der komplexen Ebene zurückbringt.

3.6 Spickzettel: Trigonometrie und hyperbolische Funktionen

Hyperbolische Funktionen erschrecken Chemiestudierende oft, aber tatsächlich sind sie nahe Verwandte der gewöhnlichen Sinus- und Kosinusfunktionen, die einfach in der komplexen Ebene "leben".

3.6.1 Definitionen über die Exponentialfunktion

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

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

3.6.2 Zusammenhang zwischen trigonometrischen und hyperbolischen Funktionen

Wenn wir $\mathrm{i}z$ anstelle von z in die Definitionen einsetzen, sehen wir eine erstaunliche Symmetrie (Osbornsche Regeln):

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

Eselsbrücke: Beim Übergang von der Trigonometrie zur Hyperbolik wird das Argument mit i multipliziert, und vor Sinus/Kosinus können imaginäre Einheiten auftreten. Hyperbolische Funktionen beschreiben nicht Schwingungen (wie der Sinus), sondern exponentielles Wachstum/Abklingen (wie e^x), was für gedämpfte Prozesse von entscheidender Bedeutung ist.

3.7 Wozu braucht man das in der realen Chemie und Physik?

Es mag scheinen, dass imaginäre Zahlen nur eine schöne mathematische Abstraktion sind. Aber unsere Welt ist so eingerichtet, dass wir *ohne* den imaginären Raum reale Dinge nicht beschreiben könnten.

3.7.1 Quantenmechanik und Orbitale

Die Schrödinger-Gleichung, die das Verhalten von Elektronen in Atomen beschreibt, enthält die imaginäre Einheit i explizit:

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

Die Wellenfunktion Ψ ist komplexwertig. Für den Grundzustand des Wasserstoffatoms (1s-Orbital) ist die Wellenfunktion reell, und man kann sie im realen Raum zeichnen. Sobald das Elektron jedoch in einen angeregten Zustand übergeht (zum Beispiel ein 2p-Orbital mit magnetischer Quantenzahl $m \neq 0$), enthält seine Wellenfunktion den Phasenfaktor $e^{\mathrm{i}m\varphi}$. Ohne komplexe Zahlen ist es einfach unmöglich, den Drehimpuls des Elektrons und die Feinstruktur der Spektren zu beschreiben.

3.7.2 Kernspinresonanz (NMR) und Fourier-Analyse

Du wirst viel mit NMR-Spektroskopie arbeiten. Wenn du eine Probe in ein Magnetfeld bringst und einen Radiofrequenzimpuls gibst, beginnen die Kerne zu präzedieren. Der Detektor registriert ein mit der Zeit abklingendes Signal — dies nennt man **FID** (Free Induction Decay).

Das Signal von einem Kerntyp sieht wie eine gedämpfte Schwingung aus:

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

Wenn die Probe viele verschiedene Kerne enthält, verwandelt sich das Signal in einen Brei aus vielen überlagerten Kosinusfunktionen. Wie kann man herausfinden, welche Frequenzen ω darin verborgen sind?

Hier kommt die **Fourier-Transformation** zur Hilfe. Aber mit Kosinusfunktionen zu arbeiten ist schwierig. Es ist viel einfacher, zu komplexen Exponentialfunktionen überzugehen, indem man Eulers Formel verwendet:

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

In NMR-Spektrometern verwendet man Quadraturdetektion, die es erlaubt, direkt ein komplexes Signal zu messen:

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

Wenn wir auf dieses Signal die schnelle Fourier-Transformation (FFT) anwenden, geht die Zeit t in die Frequenz ω über. Eine lange oszillierende Funktion im Zeitbereich verwandelt sich in einen **schmalen Peak (eine Lorentz-Kurve)** im Frequenzbereich! Jeder Kerntyp entspricht seiner eigenen Frequenz

ω, und im resultierenden NMR-Spektrum sehen wir einzelne Peaks. Die gesamte moderne Signalverarbeitung (von der MRT in der Medizin bis zum Audio-MP3) funktioniert genau dank der Magie der komplexen Zahlen und der Euler-Exponentialfunktionen.

4 Wenn eine Messung nicht genügt: Vektoren und Normen

4.1 Von der Ebene zum N-dimensionalen Raum

Nun, wenn wir komplexe Zahlen haben (im Wesentlichen Zahlenpaare), dann gibt es sicher auch etwas Komplizierteres? Quaternionen? Oktaven?

Ja, tatsächlich haben Mathematiker Zahlen von der Ebene auf höhere Dimensionen verallgemeinert. Aber jenseits der komplexen Zahlen und der Quaternionen (die sich übrigens in der Robotik und der 3D-Grafik zur Beschreibung von Drehungen in unserem dreidimensionalen Raum hervorragend eingebürgert haben) werden spezielle "Zahlentypen" praktisch nicht verwendet. Stattdessen gruppieren Menschen einfach Zahlen in Mengen und nennen sie **N-dimensionale Vektoren**.

Das ist einfach der N-dimensionale Raum. N kann zwei sein (dann ist es eine Ebene), drei (unser gewöhnlicher physikalischer Raum) oder sehr, sehr groß. Solche Vektoren bezeichnen wir fett, zum Beispiel $\mathbf{v} \in \mathbb{R}^N$. Das ist ein Vektor:

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

Hier bedeutet T Transposition, um eine Spalte in eine Textzeile zu schreiben. Wir sind frei, uns auszudenken, was wir mit diesem Raum machen, aber das Erste, was wir brauchen, ist zu verstehen, wie man die "Größe" oder "Länge" eines solchen Vektors misst.

4.2 Wozu brauchen wir Vektoren? Beispiele aus der Chemie

Auf den ersten Blick ist ein Vektor einfach eine Menge von Zahlen. Aber woher kommen die Zahlen? Angenommen, es sind Daten von einem Chromatographen oder einem NMR-Spektrometer.

Stell dir vor, wir wollen herausfinden, welcher Peak hier der "hellste" ist. Das ist doch einfach das Maximum unter diesen Zahlen, oder? Und nun nimm ein Chromatogramm von irgendeinem komplexen Schmutz und ein Chromatogramm einer einzigen reinen Substanz. Wir injizieren beide in das Chromatograph in gleichem molarem Volumen (wir wollen vereinbaren, dass der Detektor alle Moleküle gleich empfindet).

- Im Fall der reinen Substanz erhalten wir einen schönen, sehr scharfen und hohen Peak.
- Im Fall des Schmutzes erhalten wir einen Berg von Peaks, vielleicht sogar zu einem durchgehenden flachen Buckel verschmolzen.

Aber interessant ist: Die **Fläche** (integrierte Intensität) beider Chromatogramme wird gleich sein! Der Peak der reinen Substanz wird im Vergleich zum Berg sehr hoch sein, aber die Summe aller Antworten ist gleich der Stoffmenge.

Aus diesem Problem ergeben sich auf natürliche Weise drei verschiedene Möglichkeiten, unseren Datenvektor zu bewerten:

1. **Maximum.** Uns interessiert die maximale Konzentration (oder die maximale Absorption, wenn der Peak "nach unten schaut"). Wir nehmen den größten Wert.
2. **Summe.** Uns interessiert die Gesamtmenge der Substanz. Wir addieren alle Werte (meist im Betrag, da die Grundlinie nicht ideal sein muss).
3. **Wurzel aus der Summe der Quadrate.** Und wozu braucht man das? Stell dir vor, wir beschreiben ein Molekül nicht durch ein Chromatogramm, sondern durch drei Parameter: *Polarität, Molmasse, Siedepunkt*. Das ist ein Vektor in $\mathbb{R}^3$. Um zu verstehen, wie *ähnlich* zwei Moleküle einander sind

(zum Beispiel für das Wirkstoffdesign), können wir nicht einfach die Differenzen der Parameter addieren oder die maximale Differenz nehmen. Wir brauchen genau den **euklidischen Abstand** — den kürzesten Weg in diesem "chemischen Raum". Oder ein anderes Beispiel: In der Physik ist die quadratische Norm eines Vektors (eines Spektrums) proportional zur **Gesamtenergie** dieses Signals.

4.3 Die mathematische Sprache der Normen

Alle diese drei intuitiven Begriffe heißen in der Mathematik **Normen von Vektoren** und werden durch doppelte senkrechte Striche $\|\mathbf{v}\|$ bezeichnet.

- **Maximumnorm (L_∞):**
$$\|\mathbf{v}\|_\infty = \max_{1 \le k \le N} |v_k|$$
- **Manhattan-Abstand / Summe der Beträge (L_1):**
$$\|\mathbf{v}\|_1 = \sum_{k=1}^{N} |v_k|$$
- **Euklidische Norm / Länge des Vektors (L_2):**
$$\|\mathbf{v}\|_2 = \sqrt{\sum_{k=1}^{N} |v_k|^2}$$

Aber Mathematiker lieben Ordnung, deshalb haben sie all das in einer allgemeinen Formel für die L_p**-Norm** zusammengefasst:

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

In dieser Formel ist L_1 der Fall $p = 1$. L_2 ist $p = 2$. Und was wird p für das Maximum? Ja, genau, $p \to \infty$. Wenn man den Grenzwert dieser Formel für $p \to \infty$ bildet, erhält man mathematisch streng genau das maximale Element des Vektors.

4.4 Der kontinuierliche Fall: Funktionen als Vektoren

Darüber hinaus kann unser Vektor ganz kontinuierlich sein. Dann ist er tatsächlich eine **Funktion** $f(x)$. Vorerst merken wir uns nur, dass man eine Funktion als "unendlich-dimensionalen Vektor" betrachten kann, dessen Werte an jedem Punkt x gegeben sind. Und für Funktionen funktionieren die Normen genauso, nur ersetzen wir die Summe durch ein Integral:

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

Wir werden manchmal mit einem Array (einem Vektor), manchmal mit einer Funktion operieren, und später, im Kurs über Signalverarbeitung, werden wir sehen, dass dies zwei Seiten derselben Medaille sind.

4.5 Minkowskis Ungleichung

Es gibt einen entscheidend wichtigen Punkt, den man sich merken muss. Wir werden oft $\|\mathbf{x}+\mathbf{y}\|$ mit $\|\mathbf{x}\|$ und $\|\mathbf{y}\|$ vergleichen. Intuitiv ist klar, dass, wenn wir von Punkt A nach Punkt B und dann von B nach C gehen, der Gesamtweg nicht kürzer sein wird, als wenn wir direkt von A nach C gegangen wären.

Diese geometrische Eigenschaft (die Länge einer Dreiecksseite ist nicht größer als die Summe der beiden anderen) wird für beliebige L_p-Normen in **Minkowskis Ungleichung** formalisiert:

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

Wie verwendet man das? Es erlaubt uns, Fehler abzuschätzen. Wenn $\mathbf{x}$ das wahre Signal und $\mathbf{y}$ das Rauschen (Messfehler) ist, dann garantiert Minkowskis Ungleichung, dass die Norm (Größe) unseres gemessenen Signals $(\mathbf{x}+\mathbf{y})$ die Summe der Norm des wahren Signals und der Norm des Rauschens nicht überschreitet. *(Einen strengen und schönen Beweis dieser Ungleichung findet man in Wikipedia, um dieses Skript nicht zu überladen.)*

5 Zwei Dimensionen und mehr: Matrizen und Tensoren

5.1 Von Vektoren zu Matrizen

Da wir einen Vektor $\mathbf{a} \in \mathbb{R}^N$ haben (eine eindimensionale Entität, das diskrete Analogon der Funktion $f(x)$), ist es logisch zu fragen: Was, wenn wir eine Funktion zweier Variablen $f(x,y)$ haben? Was ist ihr diskretes Analogon? Und für $f(x,y,z)$?

Ja, zweidimensionale Objekte im diskreten Raum sind **Matrizen**. Und wie im Fall der komplexen Zahlen ist hier alles sehr, sehr gut untersucht. Und alles, was drei oder mehr Dimensionen hat, nennt man üblicherweise **Tensoren**.

Übrigens, wenn du jemals auf alte wissenschaftliche Literatur über Chemometrie oder Datenanalyse stößt, wirst du dich über den Zoo der Namen für Tensoren wundern. Sie werden *multilinear, multidimensional, procrustes, multimodal, three-way, multi-way arrays* genannt. Erschrick nicht: Hinter all diesen schönen Wörtern verbirgt sich einfach ein mehrdimensionales Zahlenarray.

Vorerst bleiben wir bei zweidimensionalen Objekten: der Funktion $f(x,y)$ und der Matrix $\mathbf{A} \in \mathbb{R}^{N\times M}$.

5.2 Komplexe Vektoren und Matrizen

Warum habe ich bisher nur $\mathbb{R}$ geschrieben? Sowohl ein Vektor als auch eine Matrix können natürlich komplex sein! Wir können die Räume $\mathbb{C}^N$ und $\mathbb{C}^{N\times M}$ betrachten. Hier ist alles einfach: Statt einer reellen Zahl in jeder Zelle des Vektors oder der Matrix steht ein komplexes Paar.

Aber wie berechnet man die Norm einer komplexen Zahl? Genauso! Der Betrag einer komplexen Zahl $z = x + \mathrm{i}y$ wird über die konjugierte Zahl $\bar{z} = x - \mathrm{i}y$ berechnet:

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

Entsprechend sieht die L_2-Norm eines komplexen Vektors $\mathbf{z} \in \mathbb{C}^N$ so aus:

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

In Matrixschreibweise wird dies über die hermitesche Konjugation (Transposition + komplexe Konjugation) $\mathbf{z}^H$ geschrieben:

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

5.3 Normen von Matrizen

Und für Matrizen können wir ebenfalls Normen definieren! Aber hier gibt es eine wichtige Nuance.

Die meisten einfachen Normen für Matrizen werden so berechnet: Wir "schneiden" die Matrix gedanklich in Zeilen, ordnen alle ihre $N \times M$ Zellen zu einem langen Spaltenvektor an und nehmen die Norm dieses Vektors. Die beliebteste dieser Normen ist die **Frobenius-Norm** (oder euklidische Norm für Matrizen), die das Analogon der L_2-Norm ist:

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

Für sie gibt es, wie für die gewöhnliche L_2, viele wichtige und schöne Eigenschaften, die wir bald betrachten werden.

Jedoch gibt es neben solchen "elementweisen" Normen in der linearen Algebra spezielle **Operatornormen (induzierte Normen)**. Sie beantworten die Frage: "Wie stark kann eine Matrix einen Vektor strecken, wenn wir sie mit ihm multiplizieren?". Aber das ist eine Geschichte für das nächste Kapitel, in dem wir uns eingehend mit linearen Operatoren beschäftigen.

6 Operatoren: wenn die Mathematik beginnt zu handeln

6.1 Was ist ein Operator?

Wir sind zu einem sehr wichtigen Konzept gekommen, das überall in der Mathematik und in der Chemie vorkommt. Das ist der **Operator**.

Ein Operator ist eine gewisse *Wirkung* auf ein Objekt. Aber hier wird es sofort irgendwie verwirrend: eine Wirkung auf was? Auf eine Zahl? Auf einen Vektor? Auf eine Funktion? Betrachten wir gemeinsam mehrere Beispiele, und alles wird sich klären.

6.2 Eine Matrix als Operator

Angenommen, wir haben eine quadratische Matrix $\mathbf{A} \in \mathbb{C}^{N\times N}$ (ich werde im Folgenden fast immer $\mathbb{C}$ verwenden, da $\mathbb{R}$ nur eine Teilmenge von $\mathbb{C}$ ist und alles, was für reelle Zahlen funktioniert, auch für komplexe funktioniert).

Betrachten wir die Wirkung **Ab**, also die Multiplikation einer Matrix mit einem Vektor. Das Ergebnis ist wieder ein Vektor aus $\mathbb{C}^N$. Die Matrix *transformiert* einen Vektor in einen anderen. Dies ist das einfachste Beispiel eines linearen Operators.

6.3 Analogie zum Funktionaloperator

Und nun — das Interessanteste. Das Analogon eines Matrixoperators in der Welt der Funktionen ist der **Integraloperator**:

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

Hier ist $K(x,y)$ der *Kern* des Operators (das Analogon der Matrix **A**), $g(y)$ die Eingabefunktion (das Analogon des Vektors **b**), und das Ergebnis ist eine neue Funktion von x.

Wo, interessant, kommt eine solche Abstraktion im realen Leben vor? Es stellt sich heraus, überall! Zum Beispiel wirkt in der Quantenmechanik der Hamilton-Operator $\hat{H}$ auf die Wellenfunktion Ψ, und dies ist gerade ein Integro-Differentialoperator. In der Spektroskopie wird die Antwort eines Instruments auf ein Eingangssignal oft durch ein Faltungsintegral beschrieben — auch das ist ein Operator.

6.4 Zurück zur Schule: ein Gleichungssystem

Aber fangen wir mit dem Einfachsten an. In der Schule haben wir bereits Gleichungssysteme gelöst, zum Beispiel:

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

Du denkst wahrscheinlich: "Na und? Wir konnten das doch durch Einsetzen oder Additionsverfahren lösen". Ja, konnten wir. Aber schreiben wir es in Matrixform $\mathbf{Ax} = \mathbf{b}$ um:

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

Du wirst mich berechtigterweise fragen: "Warum so verkomplizieren?". Und ich antworte: Nein, wir verkomplizieren nicht — wir *bringen Ordnung* in unser Wissen. Wir isolieren die *Struktur* des Problems. Und nun können wir sagen: Wenn wir einen **linearen Operator A** haben, dann wirkt er auf den Vektor **x**, und das Ergebnis ist der Vektor **b**.

6.5 Erinnerung: grundlegende Operationen

Bevor wir weitergehen, halten wir ausdrücklich die Operationen fest, die wir ständig verwenden werden.

Ein Vektor als Spaltenmatrix. Ein Vektor $\mathbf{x} \in \mathbb{C}^N$ ist tatsächlich eine Matrix der Größe $N \times 1$. Deshalb gelten alle Regeln der Matrixmultiplikation automatisch auch für Vektoren.

Transposition. Die Operation $\mathbf{A}^T$ vertauscht Zeilen und Spalten: $(\mathbf{A}^T)_{ij} = A_{ji}$. Für komplexe Matrizen verwendet man häufiger die **hermitesche Konjugation** $\mathbf{A}^H = \overline{\mathbf{A}^T}$ (Transposition + komplexe Konjugation aller Elemente).

Skalarprodukt. Für zwei Vektoren $\mathbf{x}, \mathbf{y} \in \mathbb{C}^N$ ist das Skalarprodukt definiert als:

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

Für reelle Vektoren ist dies einfach $\mathbf{x}^T \mathbf{y} = \sum x_k y_k$.

Die wichtigste Eigenschaft: Das Skalarprodukt eines Vektors mit sich selbst ist gleich dem **Quadrat seiner euklidischen Norm**:

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

Dies ist die Brücke zwischen Algebra und Geometrie: Die Länge eines Vektors ist die Wurzel aus seinem "skalaren Quadrat".

Multiplikation einer Matrix mit einem Vektor. Wenn $\mathbf{A} \in \mathbb{C}^{M \times N}$ und $\mathbf{x} \in \mathbb{C}^N$, dann wird das Ergebnis $\mathbf{y} = \mathbf{Ax} \in \mathbb{C}^M$ nach der Regel berechnet:

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

Das heißt, die i-te Komponente des Ergebnisses ist das Skalarprodukt der i-ten Zeile der Matrix mit dem Vektor **x**.

Multiplikation von Matrizen. Wenn $\mathbf{A} \in \mathbb{C}^{M \times K}$ und $\mathbf{B} \in \mathbb{C}^{K \times N}$, dann ist das Produkt $\mathbf{C} = \mathbf{AB} \in \mathbb{C}^{M \times N}$:

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

Wichtig: Die Matrixmultiplikation ist im Allgemeinen **nicht kommutativ** — $\mathbf{AB} \neq \mathbf{BA}$. Das ist einer der wichtigsten Unterschiede zur Multiplikation gewöhnlicher Zahlen!

Alle diese Operationen — Skalarprodukt, Matrix-Vektor-Multiplikation, Matrix-Matrix-Multiplikation — haben ihre vollständigen Analoga in der Welt der Funktionen und Integraloperatoren. Nur überall dort, wo wir eine Summe $\sum$ hatten, erscheint ein Integral $\int$, und die Transposition wird durch einen komplizierteren Konjugationsoperator ersetzt. Aber das ist bereits eine Frage der Technik.

6.6 Die vier Hauptprobleme der linearen Algebra

Tatsächlich dreht sich, solange unsere Operatoren linear sind (das heißt, sie können in der Form (1) oder $\mathbf{Ax}$ geschrieben werden), die ganze Welt der möglichen Probleme um einige typische Fragen. Listen wir sie auf.

Problem 1. Ein Skalarprodukt berechnen. Gegeben zwei Vektoren — finde die Zahl $\langle \mathbf{x}, \mathbf{y} \rangle$. Dies ist die grundlegende Operation, aus der wie aus Bausteinen alles andere aufgebaut wird.

Problem 2. Die Wirkung eines Operators berechnen. Gegeben $\mathbf{A}$ und $\mathbf{x}$ — finde $\mathbf{Ax}$. Oder gegeben $\mathbf{A}$ und $\mathbf{B}$ — finde $\mathbf{AB}$. Dies ist eine direkte Berechnung.

Problem 3. Ein lineares Gleichungssystem lösen. Gegeben $\mathbf{A}$ und $\mathbf{b}$ — finde $\mathbf{x}$ so, dass $\mathbf{Ax} = \mathbf{b}$. Dies ist das *direkte* Problem: Der Operator und das Ergebnis sind bekannt, man muss finden, worauf er gewirkt hat.

Problem 4. Das Residuum minimieren (Methode der kleinsten Quadrate). Aber was, wenn das System $\mathbf{Ax} = \mathbf{b}$ keine exakte Lösung hat? (Zum Beispiel gibt es mehr Gleichungen als Unbekannte — ein überbestimmtes System, was für die Verarbeitung experimenteller Daten typisch ist.) Dann suchen wir ein $\mathbf{x}$, das das Residuum *minimiert*:

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

Fast immer wird $p = 2$ als Norm verwendet — dies ist die berühmte **Methode der kleinsten Quadrate (MKQ)**. Sie hat eine tiefe geometrische Interpretation: Wir projizieren den Vektor $\mathbf{b}$ auf den von den Spalten der Matrix $\mathbf{A}$ aufgespannten Unterraum.

6.7 Und was, wenn die rechte Seite null ist?

Und nun — eine raffinierte Wendung. Stellen wir uns vor, dass $\mathbf{b} = \mathbf{0}$. Dann hat das Problem $\min_{\mathbf{x}} \|\mathbf{Ax}\|_2$ die triviale Lösung $\mathbf{x} = \mathbf{0}$. Aber das ist uninteressant!

Stellen wir eine zusätzliche Bedingung: Wir suchen eine **von Null verschiedene** Lösung, zum Beispiel mit der Nebenbedingung $\|\mathbf{x}\|_2 = 1$. Das heißt, wir suchen die Richtung, in der die Matrix $\mathbf{A}$ den Raum am stärksten "staucht" (oder umgekehrt am stärksten streckt — das ist bereits eine Frage des Vorzeichens).

Und hier betreten **Eigenwerte und Eigenvektoren** die Bühne. Wir suchen solche speziellen Vektoren $\mathbf{v}$ und Zahlen λ, für die die Wirkung der Matrix sich einfach auf eine Streckung reduziert:

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

Geometrisch: Ein Eigenvektor ist eine Richtung, die die Matrix *nicht dreht*, sondern nur um den Faktor λ streckt oder staucht.

Die Aufgabe der Minimierung von $\|\mathbf{Ax}\|_2$ unter der Nebenbedingung $\|\mathbf{x}\|_2 = 1$ wird genau über die Eigenwerte von $\mathbf{A}^H\mathbf{A}$ oder die Singulärwerte von $\mathbf{A}$ gelöst: Das Minimum wird beim Eigenvektor der Matrix $\mathbf{A}^H\mathbf{A}$ erreicht, der dem *betragsmäßig kleinsten* Eigenwert entspricht. Und das Maximum — beim Vektor, der dem größten entspricht.

6.8 Und wenn der Operator selbst unbekannt ist?

Du wirst fragen: "Und was, wenn uns der Operator unbekannt ist und wir nur die Eingabefunktionen und die Ergebnisse kennen?"

Hier ist die Antwort etwas mehrdeutig. Denk nach: In Matrixform haben wir nur N Eingabeparameter (den Vektor $\mathbf{x}$), und wir wollen N^2 Parameter der Matrix $\mathbf{A}$ finden. Die Natur ist selten so eingerichtet, dass eine kleine Zahl von Eingaben eine größere Zahl von Parametern gut bestimmt. Dies ist ein klassisches **unterbestimmtes** Problem.

Aber auch hier gibt es eine Lösung! Wenn wir sagen, dass dieselbe Matrix $\mathbf{A}$ auf mehrere verschiedene Vektoren $\mathbf{x}_1, \ldots, \mathbf{x}_K$ mit bekannten Ergebnissen $\mathbf{b}_1, \ldots, \mathbf{b}_K$ wirkt, dann können wir das Problem so formulieren:

$$\forall k = 1, \ldots, K: \quad \mathbf{A}\mathbf{x}_k = \mathbf{b}_k, \quad \text{wobei } \mathbf{A} \text{ unbekannt ist.} \tag{2}$$

Wenn wir die Vektoren zu Matrizen $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_K)$ und $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_K)$ zusammenfassen, wird das Problem umgeschrieben als:

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

Und dies ist, Achtung, **dasselbe Minimierungsproblem**, nur minimieren wir jetzt nicht über $\mathbf{x}$, sondern über $\mathbf{A}$:

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

(Hier ist $\|\cdot\|_F$ die Frobenius-Norm für Matrizen, die wir im vorigen Kapitel besprochen haben.)

Transponieren wir zur Bequemlichkeit: $\mathbf{X}^T\mathbf{A}^T \approx \mathbf{B}^T$. Und wir haben wieder ein Problem der Form "minimiere $\|\mathbf{M}\mathbf{z} - \mathbf{c}\|_2$" erhalten, nur für jede Spalte von $\mathbf{A}^T$ separat. Dies ist die klassische **lineare Regression** — die Grundlage der Chemometrie, QSAR, Gerätekalibrierung und vielem mehr.

6.9 Singulärwertzerlegung: eine Brücke in große Welten

Und hier betritt eine der schönsten Konstruktionen der gesamten linearen Algebra die Bühne — die **Singulärwertzerlegung (SVD, Singular Value Decomposition)**.

Jede Matrix $\mathbf{A} \in \mathbb{C}^{M\times N}$ kann dargestellt werden als:

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

wobei $\mathbf{U} \in \mathbb{C}^{M\times M}$ und $\mathbf{V} \in \mathbb{C}^{N\times N}$ unitäre Matrizen sind (ihre Spalten sind orthonormierte Vektoren) und $\square$ eine Diagonalmatrix mit nichtnegativen Zahlen $\sigma_1 \geq \sigma_2 \geq \cdots \geq 0$ auf der Diagonalen ist. Diese Zahlen heißen **Singulärwerte**.

Die geometrische Bedeutung der SVD ist atemberaubend: Jede Matrix ist einfach eine Drehung ($\mathbf{V}^H$), dann eine Streckung entlang der Achsen ($\square$), dann noch eine Drehung ($\mathbf{U}$). Das ist alles! Mehr können Matrizen nicht.

SVD ist ein universelles Werkzeug. Durch sie werden gelöst:

- Probleme der kleinsten Quadrate,
- die Suche nach der Pseudoinversen,
- Datenkompression und Extraktion von Hauptkomponenten (PCA — Principal Component Analysis, die Grundlage der Chemometrie!),
- Regularisierung schlecht konditionierter Probleme.

Hier gibt es einen wichtigen Punkt: Unitäre Matrizen haben eine sehr wichtige Eigenschaft, nämlich dass immer $\mathbf{V}^H\mathbf{V} = \mathbf{I}$ gilt, wobei $\mathbf{I}$ die Einheitsmatrix ist, also eine, die auf der Hauptdiagonalen Einsen hat und sonst mit Nullen gefüllt ist.

6.10 Eine Brücke zur Funktionalanalysis: Fredholm-Theorie

Und nun — das Interessanteste. Erinnerst du dich an den Integraloperator (2)? Nun, auch für ihn gibt es ein Analogon der SVD! Es heißt **Singulärwertzerlegung eines kompakten Operators** oder, in klassischerer Formulierung, **Fredholm-Theorie**.

Der Kern ist folgender: Der Kern des Integraloperators $K(x, y)$ kann in eine Reihe nach “Singulärfunktionen” $u_k(x)$ und $v_k(y)$ entwickelt werden:

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

wobei σ_k die Singulärwerte sind (gegen Null abnehmend) und u_k, v_k orthonormierte Funktionensysteme sind.

Dies ist genau das Analogon der SVD für Matrizen, nur im unendlich-dimensionalen Raum! Und alle Ideen, die wir an Matrizen verstanden haben, übertragen sich hierher fast wörtlich.

Aber überladen wir unser Bewusstsein jetzt nicht mit Fredholm und Funktionalanalysis. Die gute Nachricht ist, dass **der größte Teil dessen, was wir in Chemie und Signalverarbeitung brauchen, auf Matrixebene erledigt werden kann**. Und wo es nicht mehr geht (zum Beispiel in der Quantenmechanik oder in der strengen Theorie der Integralgleichungen), erinnern wir uns einfach, dass “irgendwo da draußen Fredholm ist”, und fragen bei Bedarf die KI nach Details — sie wird uns gerne über die Fredholm-Theorie, über Hilbert–Schmidt-Kerne und über die Spektren kompakter Operatoren erzählen.

Die Hauptsache ist, die *Struktur* zu verstehen. Und die Struktur ist überall dieselbe: Operator, Raum, Wirkung, Entwicklung nach “Basisrichtungen”.

7 Kondition: wenn eine Matrix uns “täuschen will”

7.1 Wenn eine Lösung existiert, aber es scheint, als gäbe es sie nicht

Also, wir haben eine quadratische Matrix $\mathbf{A}$ und ein lineares System $\mathbf{Ax} = \mathbf{b}$. Was können wir darüber sagen?

Wir erinnern uns aus dem Schulkurs, dass ein Gleichungssystem manchmal so ist, dass, wenn wir mit dem Einsetzen beginnen, entweder keine Lösungen existieren oder wir am Ende zwei oder mehr identische Gleichungen erhalten — und dann gibt es unendlich viele Lösungen.

In der Chemie, und überhaupt in jeder industriellen Aufgabe, kommen die Ausgangsdaten für Matrixgleichungen oft aus Messungen mit Geräten. Und hier kann uns gleich auf drei Arten Pech widerfahren:

1. Unser System wird entartet sein (keine eindeutige Lösung).
2. Unser System wird viele Lösungen haben.
3. **Der heimtückischste Fall:** Wir befinden uns beim Lösen in Maschinenarithmetik *sehr nahe* am schlechten Fall, bemerken es aber nicht. Und wir erhalten eine Antwort, die vernünftig aussieht, aber tatsächlich völliger Unsinn ist.

Wann kann das passieren? Klären wir das an einem lebendigen Beispiel.

7.2 Ein Beispiel aus der Chromatographie: die Falle des Großen und des Kleinen

Angenommen, wir haben zwei Chromatogramme, in denen zwei Substanzen vor dem Hintergrund eines Lösungsmittels aufgenommen wurden. Der Peak des Lösungsmittels kam sehr breit heraus und “kroch” auf die Peaks der gesuchten Substanzen, während die Antworten der gesuchten Substanzen sehr schwach ausfielen.

Tatsächlich haben wir im Peak der gesuchten Substanzen:

- Erstes Chromatogramm: $(a + b_1)$, wobei a eine riesige Antwort des Lösungsmittels ist und b_1 eine sehr kleine Zahl von der gesuchten Substanz.
- Zweites Chromatogramm: $(a + b_2)$, aber da die Temperatur etwas höher wurde, ist diese Messung leicht ungenau, sagen wir, sie erhöhte sich um einen gewissen Wert ε *relativ zum Peak des Lösungsmittels*, also $(1 + \varepsilon)(a + b_2)$.

Wenn wir das zweite vom ersten subtrahieren wollen, um $b_1 - b_2$ zu berechnen, erhalten wir stattdessen:

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

Das heißt, wenn εa größenordnungsmäßig mit $b_1 - b_2$ vergleichbar ist, können wir nicht nur einen großen Fehler erhalten, sondern auch das **entgegengesetzte Vorzeichen**!

Dies ist das Wesen der schlechten Kondition: wenn in einer Zahl gleichzeitig eine sehr große und eine sehr kleine Komponente “leben” und wir versuchen, die kleine durch Subtraktion zu extrahieren.

Regel für die Subtraktion naher Zahlen

Subtrahiere niemals zwei betragsmäßig nahe Zahlen, wenn du relative Genauigkeit brauchst. Der Verlust signifikanter Stellen ist kein Fehler des Computers, sondern eine fundamentale Eigenschaft der Arithmetik.

7.3 Wie die SVD den Mechanismus der Katastrophe aufdeckt

Dasselbe geschieht beim Lösen von Gleichungssystemen. Und wir haben eine schöne Möglichkeit, *im Voraus* abzuschätzen, was zu tun ist.

Angenommen, wir haben ein System $\mathbf{Ax} = \mathbf{b}$, wobei wir $\mathbf{x}$ suchen und alles andere gegeben ist. Erinnern wir uns an die Singulärwertzerlegung $\mathbf{A} = \mathbf{U}\square\mathbf{V}^H$. Dann kann die Lösung in mehreren Schritten umgeschrieben werden:

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

Hier nutzen wir die Eigenschaft, dass das Lösen eines Systems mit unitären Matrizen $\mathbf{U}, \mathbf{V}$ sehr einfach ist — es genügt, mit $\mathbf{U}^H$ oder $\mathbf{V}^H$ zu multiplizieren, da $\mathbf{U}^H\mathbf{U} = \mathbf{I}$. Und ein System mit einer Diagonalmatrix zu lösen ist ohnehin trivial.

Zeichnen wir, wie das für eine 3×3-Matrix aussieht:

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

Siehst du? Jede Gleichung ist einfach eine Division einer Komponente durch den entsprechenden Singulärwert.

Und hier — das Wichtigste. Stellen wir uns vor, wir haben die Singulärwerte der Ausgangsmatrix berechnet und bemerkt, dass der größte Singulärwert σ_1 sehr, sehr viel größer ist als der kleinste σ_N.

Dann geschieht beim Schritt $\mathbf{t}_2 = \square^{-1}\mathbf{t}_1$ Folgendes:

- Im Vektor $\mathbf{t}_1$ ist der Messfehler mehr oder weniger gleichmäßig über alle Komponenten verteilt.

- Nach der Multiplikation mit $\square^{-1}$ wächst die Komponente, die dem *kleinen* σ_N entspricht, um den Faktor σ_1/σ_N!
- Und wenn σ_N sogar null ist? Dann teilen wir durch null — und die Lösung existiert nicht.

Genau deshalb hat man beschlossen, das Verhältnis σ_1/σ_N als **Konditionszahl** der Matrix zu bezeichnen:

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

Tatsächlich ist dies eine Charakteristik dafür, *wie sehr wir die Lösung mit einer solchen Matrix verderben können*. Wenn $\mathrm{cond}(\mathbf{A}) \approx 10^k$, dann verlieren wir beim Lösen des Systems etwa k signifikante Stellen an Genauigkeit. Für double precision (16 signifikante Stellen) bedeutet dies, dass bei $\mathrm{cond}(\mathbf{A}) \approx 10^{16}$ die Antwort vollständig aus Rauschen besteht.

Wichtige Bemerkung

Die Konditionszahl ist eine Charakteristik der *Matrix*, nicht des Algorithmus. Kein noch so genialer Algorithmus kann ein schlecht konditioniertes System genauer lösen, als $\mathrm{cond}(\mathbf{A})$ erlaubt. Dies ist eine fundamentale Beschränkung des Problems, nicht unserer Berechnungsmethoden. Gleichzeitig beeinflusst die Kondition *nicht* die Berechnung der Eigenvektoren symmetrischer/hermitescher Matrizen oder die Suche nach Singulärvektoren — diese Probleme sind in der Regel viel stabiler.

7.4 Wo kommt das alles in der Chemie und darüber hinaus vor?

Wir hatten viel trockene Theorie, und es wäre gut, sich daran zu erinnern, wo das alles in der Chemie angewendet wird.

Quantenchemie: die Schrödinger-Gleichung. Ja, die Schrödinger-Wellengleichung und alle ihre Näherungen — die Hartree–Fock-Gleichungen, Density Functional Theory (DFT) — all dies sind Eigenwertprobleme: die Suche nach Vektoren $\mathbf{x}$ und Werten λ, die die Gleichung $\mathbf{Ax} = \lambda\mathbf{x}$ erfüllen. Die Matrizen hier — Fock-Matrizen oder Hamilton-Matrizen — haben oft Dimensionen von Tausenden und Zehntausenden, und ihre Kondition bestimmt direkt, ob wir überhaupt eine physikalisch sinnvolle Antwort erhalten können.

Massenspektrometrie und Spektroskopie. Die Suche nach einem aufgenommenen Spektrum in einer Datenbank von Antworten in einem Massenspektrometer — das sind Algorithmen, die auf der Lösung linearer Systeme basieren. Wenn du eine Mischung von Substanzen hast und verstehen willst, woraus sie besteht, löst du das System $\mathbf{Ax} = \mathbf{b}$, wobei $\mathbf{A}$ die Matrix der Referenzspektren ist, $\mathbf{b}$ das gemessene Spektrum der Mischung und $\mathbf{x}$ die Konzentrationen der Komponenten. Und wenn die Spektren der Komponenten ähnlich sind — ist die Matrix schlecht konditioniert, und die Konzentrationen werden “springen”.

Kryo-Elektronenmikroskopie. Die Rekonstruktion der 3D-Struktur von Proteinen aus Tausenden von 2D-Projektionen ist ein gigantisches dünnbesetztes Gleichungssystem. Die Kondition ist dort ein Schlüsselfaktor, der die Auflösung der resultierenden Struktur bestimmt.

NMR-Spektroskopie. Die Entfaltung (Dekonvolution) ist ein klassisches schlecht konditioniertes Problem. Genau deshalb verwendet man in der NMR die Tikhonov-Regularisierung und die Fourier-Transformation — sie verwandeln ein schlecht konditioniertes Problem in ein diagonales.

Kalibrierung analytischer Geräte. Die PLS-Regression (Partial Least Squares) ist im Wesentlichen eine SVD mit Regularisierung. Die Kondition der Matrix der Spektren beeinflusst direkt die Genauigkeit der Konzentrationsvorhersage.

Internetsuche (PageRank). Sogar die allererste Google-Suche war so eingerichtet, dass alle zuvor gefundenen Dokumente in eine riesige Matrix eingeordnet wurden und für sie die Singulärwertzerlegung berechnet wurde (oder, was äquivalent ist, der Haupt-Eigenvektor der Übergangsmatrix gesucht wurde),

was half, das gesuchte Dokument sofort anhand einer passenden Wortkombination zu finden. In der modernen KI wird SVD sehr, sehr oft verwendet — von der Kompression der Gewichte neuronaler Netze bis zur Dimensionsreduktion in Empfehlungssystemen.

Signal- und Bildverarbeitung. Die Dekonvolution verschwommener Bilder in der Mikroskopie, die Rauschunterdrückung in EKG/EEG, die Kompression von Audio (MP3) und Video — all dies sind Probleme, bei denen die Kondition eine Schlüsselrolle spielt.

Es gab eine Zeit, da waren diese drei Probleme — das Lösen linearer Systeme, die Suche nach Eigenwerten und die SVD — ein Stolperstein bei der Lösung großer industrieller Aufgaben. Es gab sogar Firmen, die *nur* solche Solver entwickelten — und sonst nichts. Allerdings war das noch in den 1990er Jahren.

7.5 Was kostet das? Rechenkomplexität

Grob gesagt, wenn wir eine quadratische $N \times N$-Matrix haben und entweder ein lineares System mit ihr lösen oder ihre Eigen- oder Singulärwerte und -vektoren suchen, dann müssen wir etwa $\mathcal{O}(N^3)$ arithmetische Operationen aufwenden, wenn wir die Struktur oder Eigenschaften dieser Matrix nicht speziell nutzen.

Dies ist der “Preis” des vollständigen Problems. Aber wenn die Matrix spezielle Eigenschaften hat, kann der Preis stark gesenkt werden — manchmal bis auf $\mathcal{O}(N)$ oder $\mathcal{O}(N \log N)$. Und darüber — weiter unten.

7.6 Der Zusammenhang zwischen Singulärwertzerlegung und Eigenwerten

Und nun — eine der schönsten Tatsachen der gesamten linearen Algebra. Angenommen, wir haben die Singulärwertzerlegung:

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

Betrachten wir, was passiert, wenn wir $\mathbf{A}$ mit ihrer hermitesch konjugierten $\mathbf{A}^H$ multiplizieren:

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

Hier haben wir verwendet, dass $\mathbf{V}^H\mathbf{V} = \mathbf{I}$ (Unitärität von $\mathbf{V}$) und $\square^T = \square$ (Diagonalmatrix).

Analog:

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

Was bedeutet das?

- Die Singulärvektoren der Matrix $\mathbf{A}$ sind *genau* die Eigenvektoren der Matrizen $\mathbf{A}\mathbf{A}^H$ und $\mathbf{A}^H\mathbf{A}$.
- Die Singulärwerte σ_k der Matrix $\mathbf{A}$ sind die Quadratwurzeln aus den Eigenwerten λ_k der Matrizen $\mathbf{A}\mathbf{A}^H$ oder $\mathbf{A}^H\mathbf{A}$:

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

Dies ist ein tiefer Zusammenhang zwischen zwei scheinbar verschiedenen Problemen: der Singulärwertzerlegung und der Eigenwertsuche.

Vorsicht: das Quadrat der Kondition!

Aber es gibt ein heimtückisches Detail. Die Konditionszahl der Matrizen $\mathbf{A}\mathbf{A}^H$ und $\mathbf{A}^H\mathbf{A}$ ist das **Quadrat** der Konditionszahl der Ausgangsmatrix:

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

Wenn $\text{cond}(\mathbf{A}) = 10^8$, dann ist $\text{cond}(\mathbf{A}^H\mathbf{A}) = 10^{16}$ — und wir verlieren alle 16 signifikanten Stellen an Genauigkeit!
Deshalb muss der Übergang vom Problem der Singulärwertzerlegung zum Eigenwertproblem für $\mathbf{A}^H\mathbf{A}$ **äußerst vorsichtig** erfolgen. In modernen Bibliotheken (LAPACK) wird die SVD direkt berechnet, ohne explizite Bildung von $\mathbf{A}^H\mathbf{A}$ — genau um diese Katastrophe zu vermeiden.

8 Klassifikation von Matrizen: Wer ist wer

Nun, da klar ist, dass praktisch alles auf diesen "einfachen" mathematischen Problemen basiert, systematisieren wir ein wenig unser Wissen über solche Lösungen. Denn wir haben immer eine Matrix, und von den Eigenschaften dieser Matrix hängt sehr viel ab.

8.1 Nach der Form

Typ	Größe	Beschreibung
Quadratisch	$N \times N$	Gleiche Anzahl von Zeilen und Spalten. Nur für solche Matrizen sind Determinante, Eigenwerte und inverse Matrix definiert.
"Stehend" (hoch)	$M \times N,\ M > N$	Mehr Gleichungen als Unbekannte. Hat meist keine exakte Lösung — wir lösen über MKQ.
"Liegend" (breit)	$M \times N,\ M < N$	Mehr Unbekannte als Gleichungen. Es gibt unendlich viele Lösungen — wir suchen die mit minimaler Norm.

8.2 Nach Symmetrie und Struktur der Elemente

Typ	Definition und Eigenschaften	Komplexität
Symmetrisch reell	$\mathbf{A} = \mathbf{A}^T$, Elemente $\in \mathbb{R}$. Eigenwerte **reell**, Eigenvektoren — orthogonal.	$\mathcal{O}(N^3/3)$
Hermitesch komplex	$\mathbf{A} = \mathbf{A}^H$, Elemente $\in \mathbb{C}$. Eigenwerte **reell**, Eigenvektoren — orthonormiert.	$\mathcal{O}(4N^3/3)$
Schiefsymmetrisch	$\mathbf{A} = -\mathbf{A}^T$. Eigenwerte — rein imaginär oder null.	$\mathcal{O}(N^3/3)$
Orthogonal / Unitär	$\mathbf{A}^T\mathbf{A} = \mathbf{I}$ / $\mathbf{A}^H\mathbf{A} = \mathbf{I}$. Erhält Längen und Winkel. Eigenwerte betragsmäßig gleich 1.	$\mathcal{O}(N^2)$ für Multiplikation
Einheitsmatrix	$\mathbf{A} = \mathbf{I}$. Diagonale aus Einsen, Rest — Nullen.	$\mathcal{O}(N)$
Diagonal	$A_{ij} = 0$ für $i \neq j$. Wird als Vektor aus N Zahlen gespeichert.	$\mathcal{O}(N)$

8.3 Nach positiver Definitheit

Dies ist eine Unterklasse symmetrischer/hermitescher Matrizen, und sie ist für Optimierung und Statistik von entscheidender Bedeutung.

Typ	**Definition und Eigenschaften**
Positiv definit ($\mathbf{A} \succ 0$)	$\forall \mathbf{x} \neq 0 : \ \mathbf{x}^H \mathbf{A} \mathbf{x} > 0$. Alle Eigenwerte streng positiv. Kondition — Verhältnis des größten zum kleinsten Eigenwert. Wird mit dem Cholesky-Verfahren in $\mathcal{O}(N^3/3)$ gelöst.
Positiv semidefinit ($\mathbf{A} \succeq 0$)	$\forall \mathbf{x} : \ \mathbf{x}^H \mathbf{A} \mathbf{x} \geq 0$. Alle Eigenwerte ≥ 0. Kann entartet sein.

8.4 Nach der Struktur der Anordnung der Nullelemente

Typ	Definition und Eigenschaften	Komplexität
Bandmatrix	Nullelemente nur in der Nähe der Hauptdiagonalen: $A_{ij} = 0$ für $\|i - j\| > k$. Wird als $N \times (2k + 1)$ gespeichert.	$\mathcal{O}(Nk^2)$
Toeplitz-Matrix	A_{ij} hängt nur von $i - j$ ab. Jede Diagonale ist eine Konstante. Tritt bei Problemen mit stationären Prozessen auf.	$\mathcal{O}(N^2)$, über FFT — $\mathcal{O}(N \log N)$
Zirkulante Matrix	Toeplitz + Periodizität: Jede Zeile ist eine zyklische Verschiebung der vorherigen. Wird durch die Fourier-Transformation diagonalisiert.	$\mathcal{O}(N \log N)$ über FFT
Blockmatrix	Besteht aus Blöcken, von denen jeder eine Matrix ist. Erlaubt rekursive Algorithmen.	Hängt von der Blockstruktur ab
Dünnbesetzte Matrix (sparse)	Die meisten Elemente sind Nullen. Wird in speziellen Formaten gespeichert (CSR, CSC). Grundlage großer Berechnungen.	$\mathcal{O}(\text{nnz})$, wobei nnz die Anzahl der Nichtnullen ist

8.5 Entartung

Eine **entartete (singuläre) Matrix** ist eine Matrix, deren Determinante null ist, oder, was äquivalent ist, die mindestens einen Singulärwert gleich null hat. Für eine solche Matrix:

- existiert keine inverse Matrix,

- hat das System $\mathbf{Ax} = \mathbf{b}$ entweder keine Lösungen oder unendlich viele,
- ist $\text{rank}(\mathbf{A}) < N$.

In der Praxis sind Matrizen fast nie *exakt* entartet — aber sie können *fast* entartet sein, also mit einem sehr kleinen σ_{min}. Und dies ist ein viel heimtückischerer Fall, weil formal eine Lösung existiert, sie aber vollständig durch das Rauschen in den Daten bestimmt wird.

8.6 Zusätzliche wichtige Matrizentypen

Typ	Definition und Eigenschaften	Komplexität
Obere Dreiecksmatrix	$A_{ij} = 0$ für $i > j$. Alle Nichtnullen oberhalb oder auf der Hauptdiagonalen. Lösung des Systems — Rückwärtseinsetzen.	$\mathcal{O}(N^2)$
Untere Dreiecksmatrix	$A_{ij} = 0$ für $i < j$. Alle Nichtnullen unterhalb oder auf der Hauptdiagonalen. Lösung des Systems — Vorwärtseinsetzen.	$\mathcal{O}(N^2)$
Permutationsmatrix	In jeder Zeile und jeder Spalte genau eine Eins, der Rest — Nullen. Multiplikation mit einer solchen Matrix ist eine Permutation von Zeilen/Spalten. $\mathbf{P}^T = \mathbf{P}^{-1}$.	$\mathcal{O}(N)$

Permutationsmatrizen sind nicht nur eine Abstraktion. Sie sind entscheidend für die numerische Stabilität von Algorithmen (zum Beispiel bei der LU-Zerlegung mit Pivotierung), und wir werden ihnen noch begegnen.

8.7 Übersichtstabelle: Was und wie lösen

Problem	Allgemeiner Fall	Symmetrisch	Dünnbesetzt
Lineares System $\mathbf{Ax} = \mathbf{b}$	LU, $\mathcal{O}(N^3)$	Cholesky, $\mathcal{O}(N^3/3)$	Iterativ, $\mathcal{O}(\text{nnz} \cdot k)$
MKQ min $\|\mathbf{Ax} - \mathbf{b}\|_2$	QR, $\mathcal{O}(MN^2)$	—	LSQR, iterativ
Eigenwerte	$\mathcal{O}(N^3)$	$\mathcal{O}(N^3/3)$, alle reell	Lanczos, $\mathcal{O}(\text{nnz} \cdot k)$
SVD	$\mathcal{O}(MN^2)$	Über Eigenwerte von $\mathbf{A}^T\mathbf{A}$	Truncated SVD, iterativ

Hier ist k die Anzahl der Iterationen, die von der gewünschten Genauigkeit und der Kondition abhängt.

Hauptschlussfolgerung

Bevor du ein Problem löst, *schau dir die Matrix an*. Ihre Eigenschaften — Symmetrie, Dünnbesetztheit, Bandstruktur — können die Komplexität von kubisch auf linear senken. Und ihre Kondition wird dir sagen, ob es überhaupt lohnt, der erhaltenen Antwort zu vertrauen.

9 Wie lineare Algebra-Probleme gelöst werden: von der Theorie zur Praxis

9.1 Es gibt keine einzige Lösung für alle Fälle

Wie also werden diese linearen Algebra-Probleme gelöst? Sicher gibt es eine oder vielleicht ein paar der einfachsten und zuverlässigsten Lösungen?

Leider gibt es auch jetzt keine einzige Lösung für alle Lebenslagen, und es ist keine in Sicht. Der Autor dieses Skripts hat einst an der Entwicklung von mehr als hundert verschiedenen solcher Algorithmen

mitgewirkt. Ja, es gibt sehr viele solcher Algorithmen, und man muss zumindest kurz, auch ohne es jemals implementiert zu haben, verstehen, wann und was man verwenden kann.

9.2 Klassifikation der Lösungsmethoden

Die Lösungsmethoden lassen sich etwa so klassifizieren:

9.2.1 Direkte Zerlegungsmethoden

LU-Zerlegung: $\mathbf{A} = \mathbf{LU}$, wobei $\mathbf{L}$ eine untere und $\mathbf{U}$ eine obere Dreiecksmatrix ist. Mit einer solchen Faktorisierung reduziert sich die Lösung des Systems $\mathbf{Ax} = \mathbf{b}$ auf zwei einfache Einsetzungen:

1. Wir lösen $\mathbf{Ly} = \mathbf{b}$ (Vorwärtseinsetzen, $\mathcal{O}(N^2)$)
2. Wir lösen $\mathbf{Ux} = \mathbf{y}$ (Rückwärtseinsetzen, $\mathcal{O}(N^2)$)

Aber im Allgemeinen gilt $\mathrm{cond}(\mathbf{L}) \cdot \mathrm{cond}(\mathbf{U}) \geq \mathrm{cond}(\mathbf{A})$, und wenn man keine Permutationen durchführt, kann das Produkt der Konditionen wesentlich größer werden als $\mathrm{cond}(\mathbf{A})$. Das heißt, wir können unser Ausgangsproblem verderben!

Deshalb verwendet man in der Praxis fast immer die **LUP-Zerlegung**: $\mathbf{PA} = \mathbf{LU}$, wobei $\mathbf{P}$ eine Permutationsmatrix ist. Die Permutationen werden so gewählt, dass auf der Diagonalen von $\mathbf{U}$ die größtmöglichen Elemente stehen (Pivotierung). Dies garantiert numerische Stabilität.

Für spezielle Matrizen:

- Wenn $\mathbf{A}$ eine Bandmatrix ist, dann verlassen $\mathbf{L}$ und $\mathbf{U}$ nicht das Band. Komplexität $\mathcal{O}(Nk^2)$, wobei k die Bandbreite ist.
- Wenn $\mathbf{A}$ dünnbesetzt ist, können $\mathbf{L}$ und $\mathbf{U}$ wesentlich mehr Nullelemente enthalten (fill-in), und dies kann ein großes Problem sein.
- Für eine symmetrische Matrix — $\mathbf{A} = \mathbf{LDL}^H$ oder $\mathbf{PAP}^H = \mathbf{LDL}^H$.
- Für eine positiv definite — $\mathbf{A} = \mathbf{LL}^H$, das sogenannte **Cholesky-Verfahren**. Aber das Problem des Wachstums der Konditionszahl existiert auch für positiv definite Matrizen.

QR-Zerlegung: $\mathbf{A} = \mathbf{QR}$, wobei $\mathbf{Q}$ unitär ($\mathbf{Q}^H\mathbf{Q} = \mathbf{I}$) und $\mathbf{R}$ eine obere Dreiecksmatrix ist.

Diese Methode **garantiert keine Erhöhung der Konditionszahl** der Lösung, da unitäre Transformationen die Längen von Vektoren erhalten. Sie ist gut für dichte Matrizen ohne Struktur.

Aber wenn die Ausgangsmatrix dünnbesetzt ist, dann wird $\mathbf{Q}$ fast immer dicht sein (nur für sehr spezielle Fälle und Methoden), was die Anwendbarkeit dieser Methode auf riesige Probleme stark einschränkt, wenn die Matrix selbst in den Speicher passt, ihre dichte Version N^2 aber nicht.

Schur-Zerlegung: $\mathbf{A} = \mathbf{QGQ}^H$, wobei $\mathbf{G}$ eine obere Dreiecksmatrix ist (für reelle Matrizen — eine obere quasi-Dreiecksmatrix mit 2×2-Blöcken auf der Diagonalen).

Sie wird zur Suche nach Eigenwerten und Eigenvektoren verwendet, die man durch Lösen des Eigenwertproblems für die bereits dreieckige Matrix $\mathbf{G}$ findet (und für eine Dreiecksmatrix sind die Eigenwerte einfach die Diagonalelemente!).

9.2.2 Iterative Methoden

Dies sind Methoden zur Lösung eines linearen Systems oder eines Eigenwertproblems, wenn wir aufgrund seiner Struktur effizient mit der Matrix multiplizieren können und die Lösung iterativ aufbauen.

Es gibt hier viele Methoden, aber wichtig ist: **die Konvergenz iterativer Methoden hängt von der Konditionszahl ab**. Je größer sie ist, desto langsamer konvergieren sie. (Es gibt Ausnahmen: Wenn die Matrix nur wenige sehr große und sehr kleine Singulärwerte hat, kann die Konvergenz wesentlich schneller sein.)

Alle diese Methoden lassen sich zudem in Unterklassen einteilen:

1. **Spezielle Methoden für positiv definite Matrizen** mit geringem zusätzlichem Speicheraufwand und vernünftiger Iterationszahl. Die bekannteste ist das **Verfahren der konjugierten Gradienten (Conjugate Gradients, CG)**.

2. **Rückführung eines nichtsymmetrischen Problems auf ein symmetrisches:** Statt $\mathbf{Ax} = \mathbf{b}$ löst man $\mathbf{A}^H\mathbf{Ax} = \mathbf{A}^H\mathbf{b}$. Wenn die Konditionszahl $\mathrm{cond}(\mathbf{A}^H\mathbf{A})$ nicht so enorm im Vergleich zu $\mathrm{cond}(\mathbf{A})$ ist, dann ist das vernünftig. Aber denk daran: $\mathrm{cond}(\mathbf{A}^H\mathbf{A}) = \mathrm{cond}(\mathbf{A})^2$, das ist also ein zweischneidiges Schwert.

3. **Vorkonditionierer (preconditioners):** spezielle Matrizen $\mathbf{P} \approx \mathbf{A}^{-1}$, sodass wir statt $\mathbf{Ax} = \mathbf{b}$ das System $\mathbf{PAx} = \mathbf{Pb}$ lösen, in der Annahme, dass $\mathrm{cond}(\mathbf{PA}) \ll \mathrm{cond}(\mathbf{A})$, was die Lösung durch solche iterativen Methoden stark beschleunigt.

   Übrigens werden für große dünnbesetzte Matrizen Vorkonditionierer oft als **unvollständige LU-Zerlegung** konstruiert: Man konstruiert für die Ausgangsmatrix $\mathbf{A} \approx \mathbf{LU}$, versucht aber, neue Nullelemente bei der Konstruktion von $\mathbf{LU}$ "wegzuwerfen". In diesem Fall kann man diese Matrix als etwas Ähnliches wie die Ausgangsmatrix noch anwenden, und indem man mit ihr löst, vorkonditioniert man das Ausgangsproblem und verbessert die Konvergenz.

9.3 Zwei Welten: dichte und dünnbesetzte Probleme

Tatsächlich lassen sich die meisten Lösungsalgorithmen in zwei Klassen einteilen:

1. Wenn wir bereit sind, $\mathcal{O}(N^3)$ arithmetische Operationen aufzuwenden, und $\mathcal{O}(N^2)$ Speicher haben.

2. Wenn wir uns einschränken müssen und die Matrix so riesig ist und eine spezielle Struktur hat, dass wir mit ihr multiplizieren können, sie aber in voller Form zu nehmen sehr schwierig ist.

Im ersten Fall — das sind **Zerlegungsmethoden** (LU, QR, Cholesky). Im zweiten — das sind **iterative Methoden**.

Da moderne Prozessoren und Speicher eine spezifische Struktur haben, haben iterative Methoden eine etwas schlechtere Leistung als vollständige Zerlegungsmethoden. Deshalb ist die Anwendung iterativer Methoden auf sehr kleine Matrizen praktisch nie gerechtfertigt.

9.4 Beispiele aus der Quantenchemie: DFT

Beispiel 1: Ein kleines System. Ein kleines System mit einigen Elektronen nach der DFT-Methode, und die Größe des Hamilton-Operators ist etwa 1000×1000. Wir passen in den Speicher (das sind nur ~ 8 MB für double precision). Wir können alle Eigenwerte und die Menge der uns interessierenden Eigenvektoren finden, und dafür ist die offensichtliche Wahl eine direkte Methode (zum Beispiel die Schur-Zerlegung oder der QR-Algorithmus).

Beispiel 2: Eine große chemische Struktur. Eine ziemlich große chemische Struktur nach der DFT-Methode, in der es etwa tausend Elektronenpaare in den äußeren Orbitalen gibt. Dafür haben wir den Hamilton-Operator der Schrödinger-Gleichung aufgebaut, und wir müssen für jedes Elektronenpaar einen eigenen Eigenvektor finden. Und der Hamilton-Operator ist so gegeben, dass wir etwa hundert Basisfunktionen pro Orbital genommen haben, das heißt, die Dimension dieses Hamilton-Operators ist etwa $100\,000 \times 100\,000$.

Das scheint nicht sehr viel, aber allein die Matrix eines solchen Hamilton-Operators belegt bereits etwa **80 GB** im Arbeitsspeicher, und die Lösung selbst wird noch fast ein Gigabyte belegen. Hier ist eine iterative Lösung viel naheliegender (zum Beispiel das Lanczos-Verfahren oder Davidson), die nur die wenigen benötigten Eigenvektoren findet, ohne mit der vollen Matrix zu arbeiten.

9.5 Erfinde das Rad nicht neu: Bibliotheken

Heutzutage gibt es in vielen Programmiersprachen gut und über Jahre ausgefeilte Bibliotheken zur Lösung solcher Gleichungssysteme, zur Suche nach Singulärwerten und -vektoren sowie nach Eigenvektoren und -werten. Es genügt, in der Dokumentation von **NumPy** für Python und in **LAPACK/BLAS** für C/C++ nachzuschauen — und alles wird sofort klar.

Darüber hinaus ist der Code sehr gut für moderne Computer optimiert und ziemlich komplex. Zum Beispiel enthält die moderne Version der SVD in LAPACK etwa **eine halbe Million Zeilen Code**, und sie zu wiederholen oder gar besser zu machen ist wirklich eine fast unmögliche Aufgabe.

Praktischer Rat

Schreibe niemals eigene Solver für lineare Algebra, es sei denn, es ist eine Übungsaufgabe. Verwende:

- **Python:** NumPy (`numpy.linalg`), SciPy (`scipy.linalg`)
- **C/C++:** LAPACK, BLAS, Eigen
- **Fortran:** LAPACK (das ist sein natürliches Element)
- **MATLAB:** eingebaute Funktionen (`eig`, `svd`, `lu`, `qr`)

Diese Bibliotheken haben Jahrzehnte der Optimierung und des Testens hinter sich. Sie wissen über den Prozessor-Cache, über SIMD-Instruktionen, über Multithreading — all das, was du nicht von Hand wiederholen kannst.

10 Nichtlineare Operatoren: wenn die Welt aufhört, gerade zu sein

10.1 Was ist ein nichtlinearer Operator?

Bisher haben wir über lineare Operatoren gesprochen — solche, die als $\mathbf{Ax}$ oder $\int K(x,y)g(y)\mathrm{d}y$ geschrieben werden können. Aber die reale Welt ist nichtlinear.

Ein **nichtlinearer Operator** ist eine Abbildung $\mathcal{F}$, die auf eine Funktion oder einen Vektor $\mathbf{x}$ wirkt, aber die Eigenschaften der Additivität und Homogenität *nicht* besitzt:

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

Grob gesagt, definieren wir nun, dass wir eine Funktion $f(\mathbf{x})$ haben, die von einem oder mehreren Parametern abhängt. Und wir wollen entweder:

- Die Gleichung $f(\mathbf{x}) = 0$ lösen (ein System nichtlinearer Gleichungen),
- Das Minimum $\min_{\mathbf{x}} f(\mathbf{x})$ finden (ein Optimierungsproblem).

10.2 Ein Beispiel aus der Chromatographie: das Teller-Modell

Ein klassisches Beispiel ist das System von Bilanzgleichungen und Dissoziationskonstanten auf jedem chromatographischen Teller.

Wir wollen einen Chromatographen modellieren, genauer gesagt seine Säule. In ihr findet eine Trennung statt, und die mobile Phase erzeugt beim Wandern entlang der Säule bei jedem Schritt faktisch ein solches Gleichgewicht.

Du wirst sagen — oh, hier gibt es doch nur etwa 4–5 Gleichungen und ebenso viele Unbekannte! Ja, ich stimme zu, nicht viele. Aber:

1. Es gibt viele Teller (sagen wir tausend) — das ist eins.

2. Wir zerlegen die Durchgangszeit der Substanz durch die Säule in viele Zeitschritte — das ist zwei.

Das heißt, bei jedem Zeitschritt haben wir tausendmal dasselbe Gleichungssystem mit unterschiedlichen Parametern der Eingangskonzentrationen. Und es wird wesentlich mehr solcher Zeitschritte geben als Teller, denn die Substanz wird ja von der Säule zurückgehalten.

Das heißt, wir müssen so etwa **zehn Millionen Mal** ein System aus nur 5 Gleichungen mit 5 Unbekannten lösen. Aber wenn wir es auch nur einmal nicht lösen — können wir keine weitere Lösung erhalten.

Leider ist ein solches System nicht linear. Die Bilanzgleichungen sind linear, aber die Gleichungen, die die Dissoziationskonstanten verknüpfen, enthalten Produkte und Verhältnisse der Unbekannten zueinander. Und, überraschenderweise, kann ein solches System manchmal wirklich sehr schlecht lösbar sein.

10.3 Klassifikation nichtlinearer Probleme

Bevor wir über Methoden sprechen, klassifizieren wir die Probleme selbst:

Eigenschaft	Beschreibung
Glattheit:	
Glatt	Die Funktion hat stetige Ableitungen (mindestens die erste, besser auch die zweite). Beispiel: $f(x) = x^2 + \sin x$.
Fast glatt	Die Funktion ist fast überall glatt, aber es gibt Knickstellen oder Unstetigkeiten der Ableitungen. Beispiel: $f(x) = \lvert x \rvert$.
Unstetig	Die Funktion hat Unstetigkeiten oder ist sehr verrauscht. Beispiel: experimentelle Daten mit Artefakten.
Anzahl der Minima:	
Unimodal	Es gibt nur ein globales Minimum (oder Maximum). Beispiel: konvexe Funktionen.
Multimodal	Es gibt mehrere lokale Minima, und die Aufgabe ist, das globale zu finden. Beispiel: Potentiale komplexer Moleküle.

10.4 Methoden zur Lösung nichtlinearer Probleme

Nun systematisieren wir die Lösungsmethoden. Jede Methode erfordert etwas, hängt von etwas ab, und jede hat ihre Stärken und Schwächen.

10.4.1 Monte-Carlo-Methoden

Idee: Zufällige Suche. Wir erzeugen zufällige Punkte im Parameterraum, werten die Funktion dort aus und wählen die beste.

Anforderungen: Keine. Sie kann sogar mit unstetigen Funktionen arbeiten.

Vorteile:

- Bleibt nicht in lokalen Minima hängen (bei ausreichender Iterationszahl),
- Einfach zu implementieren,
- Parallelisiert perfekt.

Nachteile:

- Sehr langsame Konvergenz: Der Fehler nimmt ab wie $\mathcal{O}(1/\sqrt{N})$, wobei N die Anzahl der Iterationen ist,
- Nutzt keine Informationen über die Struktur der Funktion.

Wann verwenden: Wenn die Funktion sehr verrauscht, unstetig ist oder wenn man einfach "irgendeine" Lösung zur Initialisierung anderer Methoden finden muss.

10.4.2 Gradientenverfahren (Gradient Descent)

Idee: Sich in die dem Gradienten entgegengesetzte Richtung bewegen:

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

wobei α_k die Lernrate ist.

Anforderungen: Die Funktion muss differenzierbar (glatt) sein.
Vorteile:

- Einfach zu implementieren,
- Garantierte Konvergenz zu einem lokalen Minimum (bei richtiger Wahl von α),
- Funktioniert gut in hohen Dimensionen.

Nachteile:

- Bleibt in lokalen Minima hängen,
- Kann in "Schluchten" sehr langsam konvergieren (wenn die Eigenwerte der Hesse-Matrix stark differieren),
- Erfordert die Wahl der Schrittweite α (zu groß — divergiert, zu klein — langsam).

10.4.3 Verfahren der konjugierten Richtungen (Conjugate Gradient für Optimierung)

Idee: Suchrichtungen so konstruieren, dass sie bezüglich der Hesse-Matrix der Funktion "konjugiert" sind. Dies erlaubt, das Minimum einer quadratischen Funktion in N Schritten zu finden (wobei N die Dimension ist).

Wichtige Bemerkung: Das Verfahren der konjugierten Gradienten zur Lösung linearer Systeme iterativ *heißt nur* genauso wie das Verfahren der konjugierten Gradienten zur nichtlinearen Minimierung. Tatsächlich ist es derselbe Algorithmus, nur auf verschiedene Probleme angewendet:

- Für lineare Systeme $\mathbf{Ax} = \mathbf{b}$ — das ist die Minimierung der quadratischen Funktion $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{Ax} - \mathbf{b}^T\mathbf{x}$,
- Für nichtlineare Optimierung — das ist die Verallgemeinerung derselben Idee auf beliebige Funktionen.

Anforderungen: Eine glatte Funktion, möglichst mit stetigem Gradienten.
Vorteile:

- Konvergiert schneller als gewöhnlicher Gradientenabstieg,
- Erfordert keine Speicherung der Hesse-Matrix (im Gegensatz zum Newton-Verfahren),
- Speicher — $\mathcal{O}(N)$.

Nachteile:

- Bleibt in lokalen Minima hängen,
- Für nichtquadratische Funktionen ist ein "Neustart" (reset) der Richtungen erforderlich.

10.4.4 Newton-Verfahren

Idee: Nicht nur den Gradienten, sondern auch die zweiten Ableitungen (die Hesse-Matrix) verwenden. Wir entwickeln die Funktion in eine Taylor-Reihe bis zur zweiten Ordnung:

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

und finden das Minimum dieser quadratischen Approximation. Wir erhalten die Iteration:

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

wobei $\mathbf{H}$ die Matrix der zweiten Ableitungen (die Hesse-Matrix) ist.

Anforderungen: Die Funktion muss zweimal differenzierbar sein. Die Hesse-Matrix muss berechenbar sein.

Vorteile:

- **Quadratische Konvergenz:** Wenn man nahe an der Lösung startet, verdoppelt sich die Anzahl der korrekten Ziffern bei jeder Iteration,
- Unempfindlich gegenüber "Schluchten" (im Gegensatz zum Gradientenabstieg).

Nachteile:

- Erfordert die Berechnung der Hesse-Matrix — $\mathcal{O}(N^2)$ Elemente,
- Erfordert die Lösung eines linearen Systems mit der Hesse-Matrix — $\mathcal{O}(N^3)$ Operationen,
- Kann divergieren, wenn man weit von der Lösung startet,
- Die Hesse-Matrix kann entartet oder nicht positiv definit sein.

10.4.5 Das Newton–Raphson-Verfahren und Block-Newton für die Quantenmechanik

Das **Newton–Raphson-Verfahren** ist eine Variante des Newton-Verfahrens zur Lösung von Systemen nichtlinearer Gleichungen $\mathbf{F}(\mathbf{x}) = \mathbf{0}$:

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

wobei $\mathbf{J}$ die Jacobi-Matrix ist (die Matrix der ersten Ableitungen $\partial F_i/\partial x_j$).

Block-Newton für die Quantenmechanik: In quantenchemischen Problemen (zum Beispiel im Hartree–Fock-Verfahren oder DFT) entsteht oft ein Gleichungssystem, das in Blöcke aufgeteilt werden kann:

- Gleichungen für die Orbitale (Entwicklungskoeffizienten),
- Gleichungen für die Dichte,
- Gleichungen für die Energie.

Das Block-Newton-Verfahren berücksichtigt diese Struktur: Die Hesse-Matrix wird in Blockform dargestellt, und jeder Block wird separat behandelt. Dies erlaubt:

- Effiziente Nutzung der Struktur des Problems,
- Parallelisierung der Berechnungen,
- Anwendung verschiedener Methoden auf verschiedene Blöcke (zum Beispiel Newton für einen Block, Gradientenabstieg für einen anderen).

10.4.6 BFGS- und L-BFGS-Verfahren

BFGS (Broyden–Fletcher–Goldfarb–Shanno) ist ein quasi-Newton-Verfahren. Die Idee: Die Hesse-Matrix nicht explizit zu berechnen, sondern sie bei jeder Iteration unter Verwendung von Informationen über die Änderung des Gradienten anzunähern.

Anforderungen: Eine glatte Funktion mit stetigem Gradienten.

Vorteile:

- Konvergenz fast wie bei Newton, aber ohne Berechnung der Hesse-Matrix,
- Garantierte positive Definitheit der Hesse-Approximation,
- Funktioniert in der Praxis gut — eine der beliebtesten Methoden.

Nachteile:

- Erfordert die Speicherung einer $N \times N$-Matrix (Hesse-Approximation) — $\mathcal{O}(N^2)$ Speicher.

L-BFGS (Limited-memory BFGS) ist eine Modifikation von BFGS für große Probleme. Die Idee: Nicht die volle Matrix zu speichern, sondern nur die letzten m Paare von Vektoren $(\mathbf{s}_k, \mathbf{y}_k)$, wobei:

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

Vorteile:

- Speicher — $\mathcal{O}(mN)$, wobei $m \sim 5$–20 (hängt nicht von N^2 ab!),
- Ideal für große Probleme ($N > 1000$),
- Sehr beliebt im maschinellen Lernen und in der Quantenchemie.

Nachteile:

- Konvergiert etwas langsamer als vollständiges BFGS,
- Erfordert die Einstellung des Parameters m.

10.4.7 Simplex-Verfahren (Nelder–Mead)

Idee: Wir konstruieren ein Simplex (im N-dimensionalen Raum sind das $N + 1$ Punkte), werten die Funktion an den Ecken aus und spiegeln, ziehen zusammen oder erweitern das Simplex iterativ, um uns dem Minimum zu nähern.

Anforderungen: Keine! Die Funktion kann nichtglatt, verrauscht, sogar unstetig sein.

Vorteile:

- Erfordert keine Gradienten,
- Einfach zu implementieren,
- Funktioniert gut für kleine Dimensionen ($N < 10$).

Nachteile:

- Sehr langsam für große N,
- Kann hängen bleiben,
- Keine theoretischen Konvergenzgarantien.

Wann verwenden: Wenn die Funktion sehr verrauscht ist oder wenn N klein ist und man zu faul ist, Gradienten abzuleiten.

10.4.8 Simulated Annealing (Simulierte Abkühlung)

Idee: Inspiriert vom physikalischen Prozess des Ausglühens von Metallen. Wir beginnen mit einer hohen "Temperatur" T, die es dem Algorithmus erlaubt, über lokale Minima "hinwegzuspringen". Wir senken T allmählich, und der Algorithmus "friert" im globalen Minimum ein.

Bei jedem Schritt:

1. Wir schlagen eine zufällige Änderung $\mathbf{x} \to \mathbf{x}'$ vor,
2. Wir berechnen $\Delta f = f(\mathbf{x}') - f(\mathbf{x})$,
3. Wenn $\Delta f < 0$ — akzeptieren wir die Änderung,
4. Wenn $\Delta f > 0$ — akzeptieren wir mit Wahrscheinlichkeit $P = \exp(-\Delta f / T)$.

Anforderungen: Keine.
Vorteile:

- Kann das globale Minimum finden (bei richtigem "Abkühlungsplan"),
- Bleibt in frühen Phasen nicht in lokalen Minima hängen,
- Einfach zu implementieren.

Nachteile:

- Sehr langsam,
- Erfordert die Einstellung des Plans $T(t)$,
- Keine Konvergenzgarantien in vernünftiger Zeit.

Wann verwenden: Wenn das Problem multimodal ist (viele lokale Minima) und das globale gefunden werden muss.

10.5 Berechnung von Gradienten

Alle Gradientenverfahren erfordern die Berechnung von $\nabla f(\mathbf{x})$. Wie macht man das?

10.5.1 Finite Differenzen

Idee: Wir approximieren die Ableitung:

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

wobei $\mathbf{e}_j$ der j-te Basisvektor ist.

Probleme:

1. **Teuer:** Zur Berechnung des Gradienten im N-dimensionalen Raum braucht man $N + 1$ Funktionsauswertungen (oder $2N$ für die zentrale Differenz).
2. **Instabil:** Wenn h zu groß ist — großer Approximationsfehler. Wenn h zu klein ist — Genauigkeitsverlust durch Subtraktion naher Zahlen. Optimal ist $h \sim \sqrt{\epsilon_{\text{mach}}}$, wobei ϵ_{mach} die Maschinengenauigkeit ist.
3. **Rauschen:** Wenn die Funktion mit Rauschen berechnet wird (zum Beispiel experimentelle Daten), dann verstärken finite Differenzen das Rauschen.

10.5.2 Automatisches Differenzieren (AD): analytische Berechnung des Gradienten

Und nun — Magie! Es stellt sich heraus, dass man den Gradienten einer Funktion *analytisch* berechnen kann, indem man automatisches Differenzieren verwendet, und das in nur **einigen wenigen Malen** so teuer wie die Berechnung der Funktion selbst!

Idee: Jede Funktion $f(\mathbf{x})$ ist eine Komposition elementarer Operationen ($+$, $-$, $\times$, $\div$, sin, cos, exp, log usw.). Wir können:

1. Die Berechnung der Funktion als **Berechnungsgraphen** darstellen,
2. Die **Kettenregel** in umgekehrter Reihenfolge anwenden (reverse mode).

Ergebnis: Der Gradient wird zu $\mathcal{O}(1)\times$ den Kosten der Funktion berechnet (in der Praxis — 3–5 Mal teurer, aber nicht N Mal!).

Wie es im Kern funktioniert:

1. Vorwärtspass: Wir berechnen $f(\mathbf{x})$ und speichern alle Zwischenergebnisse.
2. Rückwärtspass: Wir gehen den Graphen in umgekehrter Richtung durch und wenden die Kettenregel an:

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

Wo es verwendet wird:

- **PyTorch, TensorFlow:** Die Grundlage aller modernen neuronalen Netze ist genau das reverse-mode automatic differentiation.
- **Quantenchemie:** Programme wie PySCF, Psi4 verwenden AD zur Berechnung von Energiegradienten bezüglich der Kernkoordinaten.
- **Optimierung:** Alle modernen Bibliotheken (SciPy, JAX) verwenden AD.

Hauptschlussfolgerung

Berechne niemals Gradienten mit finiten Differenzen, wenn automatisches Differenzieren verwendet werden kann! Es ist schneller, genauer und stabiler.

10.6 Ein lebendiges Beispiel: Kraftfelder und Konformere

Ein weiteres lebendiges Beispiel aus der Chemie ist die Modellierung von Molekülen. Zur Modellierung kleiner Moleküle verwendet man oft eine Darstellung, in der die Gesamtenergie als Summe einfacher Wechselwirkungen geschrieben wird:

1. **Van-der-Waals-Wechselwirkungen** zwischen nicht gebundenen Atomen (Lennard-Jones-Potential):

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

2. **Harmonische Bindungen** zwischen gebundenen Atomen:

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

3. **Winkelwechselwirkungen** zwischen drei aufeinanderfolgenden Atomen:

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

4. **Torsions- (Diederv-)Wechselwirkungen** zwischen vier aufeinanderfolgenden Atomen:

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

Es wird nicht viele geben, aber etwa das Quadrat der Anzahl der Atome (für Van-der-Waals). Jede solche Funktion ist eine einfache Formel: manchmal mit Sinus, manchmal mit Exponentialfunktionen, manchmal mit Potenzen. Und insgesamt hat ein Molekül $3N$ Freiheitsgrade, da jedes Atom 3 Raumkoordinaten hat.

10.6.1 Das Konformerproblem

Es scheint, die Aufgabe ist klar: Man muss das Minimum dieser Funktion $E(\mathbf{x})$ finden, wobei $\mathbf{x} \in \mathbb{R}^{3N}$ die Koordinaten aller Atome sind. Aber hier beginnt das Interessanteste.

Die Energiefunktion eines Moleküls ist eine **multimodale Landschaft** mit einer enormen Anzahl lokaler Minima. Jedes lokale Minimum entspricht einer eigenen **Konformation** (Konformer) des Moleküls — einer eigenen Art, das Molekül im Raum zu verdrehen.

Zum Beispiel kann für ein Protein aus 100 Aminosäuren die Anzahl möglicher Konformere 10^{100} erreichen — das ist das berühmte **Levinthal-Paradoxon**. Und das globale Minimum (die native Struktur) ist nur eine von 10^{100} Möglichkeiten.

10.6.2 Warum einfache Minimierung nicht funktioniert

Wenn wir eine zufällige Anfangskonformation nehmen und gewöhnlichen Gradientenabstieg oder BFGS starten, konvergieren wir sehr schnell (innerhalb einiger hundert Iterationen) zum *nächstgelegenen* lokalen Minimum. Aber das wird fast sicher *nicht* das globale Minimum sein — also nicht die Konformation, die das Molekül in der Realität annimmt.

10.6.3 Lösung: eine Kombination von Methoden

Deshalb verwendet man in der Praxis eine Kombination von Methoden:

1. **Simulated Annealing (simulierte Abkühlung):** Wir beginnen mit einer hohen "Temperatur", die es dem Algorithmus erlaubt, über Energiebarrieren zu springen und verschiedene Bereiche des Konformationsraums zu erkunden. Wir senken die Temperatur allmählich und "frieren" das System in einem energiearmen Bereich ein.

2. **Lokale Minimierung (BFGS):** Nachdem Simulated Annealing einen "guten" Bereich gefunden hat, starten wir eine schnelle lokale Methode (BFGS oder das Verfahren der konjugierten Gradienten) für die präzise Konvergenz zu einem lokalen Minimum.

3. **Wiederholung:** Wir starten diese Kombination viele Male mit verschiedenen Anfangsbedingungen und wählen das Konformer mit der niedrigsten Energie.

Beliebte Programme

Dieser Ansatz ist in beliebten Programmen der Molekulardynamik implementiert:

- **AMBER, GROMACS, NAMD:** Verwenden Kraftfelder (AMBER, CHARMM, OPLS)

und Methoden wie Simulated Annealing + lokale Minimierung.

- **Rosetta:** Verwendet für die Vorhersage der Proteinstruktur einen Fragmentansatz + Monte Carlo + lokale Minimierung.

11 Die schnelle Fourier-Transformation: wenn $O(N^2)$ zu $O(N \log N)$ wird

11.1 Das Problem der Mustersuche

Manchmal müssen wir ein bestimmtes Fragment in einer riesigen Sequenz finden. Wenn es ein sehr kurzes Fragment ist, dann ist eine einfache Durchmusterung der ganzen Sequenz und der Vergleich mit diesem kurzen Fragment eine recht gute Strategie. So verfährt man oft bei der Suche nach kurzen Fragmenten in Proteinstrukturen.

Aber manchmal sind die Dimensionen dessen, was verglichen werden soll, fast gleich. Zum Beispiel haben wir eine periodische Kristallstruktur, und wir wissen, dass es in ihr eine gewisse Abweichung gibt, und wir verstehen sogar, dass die Abweichung nur teilweise dem ähnelt, womit wir vergleichen wollen.

Formal: Die Ausgangsdaten sind v_i, $i = 1, \ldots, N$, und das, womit verglichen werden soll, ist w_j, $j = 1, \ldots, M$, wobei $M < N$.

Wir können explizit in einer Schleife berechnen:

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

und das Maximum über s finden.

Aber die Rechenkomplexität einer solchen Suche ist quadratisch in N: Wenn $M \simeq N/2$, dann müssen $N^2/2$ arithmetische Operationen ausgeführt werden. Für $N = 10^6$ sind das 5×10^{11} Operationen — zu viele!

11.2 Matrixformulierung

Hier haben sich die Leute ausgedacht, wie man solche s_k viel schneller finden kann. Stellen wir die Berechnung dieser s in Matrixform dar. Angenommen, wir haben die Matrix:

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

der Größe $(N - M + 1) \times N$.

Wenn wir sie mit unserem Vektor $\mathbf{v}$ multiplizieren, erhalten wir genau unsere ersehnten s_k:

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

Aber wie machen wir das schnell?

11.3 Zirkulante Matrix

Wenn wir diese Matrix von unten um weitere $M-1$ Zeilen der Form ergänzen:

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

dann erhalten wir nach der Multiplikation denselben Vektor **s**, aber am Ende werden ihm noch $M-1$ weitere Zahlen angehängt, die wir verwerfen können.

Aber diese erweiterte Matrix ist eine **zirkulante Matrix C**! Sie hat eine bemerkenswerte Eigenschaft: Jede nachfolgende Zeile entsteht durch zyklische Verschiebung der vorherigen.

11.4 Diagonalisierung der zirkulanten Matrix

Eine zirkulante Matrix hat eine interessante Darstellung über die Fourier-Matrix:

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

wobei:

- **c** die erste Spalte der Ausgangsmatrix **C** ist,
- **F** die Matrix der diskreten Fourier-Transformation (DFT) ist,
- $\mathbf{F}^H$ die hermitesch konjugierte (komplex konjugierte und transponierte) Matrix ist.

Die Fourier-Matrix ist definiert als:

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

11.5 Schnelle Multiplikation über FFT

Nun das Interessanteste. Wenn wir die Fourier-Matrix schnell mit einem Vektor multiplizieren können, dann können wir dieses **s** schneller berechnen:

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

Diese Berechnung besteht aus drei Schritten:

1. $\mathbf{a} = \mathbf{Fv}$ — direkte DFT des Vektors **v**,
2. $\mathbf{b} = \operatorname{diag}(\mathbf{Fc})\mathbf{a}$ — elementweise Multiplikation (das ist $\mathcal{O}(N)$),
3. $\mathbf{s} = \frac{1}{N}\mathbf{F}^H\mathbf{b}$ — inverse DFT.

11.6 Zerlegung der Fourier-Matrix

Wie multipliziert man **F** schnell mit einem Vektor? Es stellt sich heraus, dass **F** als Produkt spezieller Matrizen dargestellt werden kann:

1. Einer Permutationsmatrix (bit-reversal permutation),
2. Mehrere Paare von Matrizen:

- Blockmatrizen der Form $\begin{pmatrix} \mathbf{I} & \mathbf{I} \\ \mathbf{I} & -\mathbf{I} \end{pmatrix}$,
- Diagonalmatrizen mit Elementen $e^{-2\pi ik/N}$ (die sogenannten "twiddle factors").

Schreiben wir diese Matrizen explizit für $N = 8$ auf:

Permutationsmatrix (bit-reversal):

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

(Zeilen gemäß Bit-Umkehrung permutiert: $0, 4, 2, 6, 1, 5, 3, 7$)

Blockmatrix (erste Stufe):

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

Diagonalmatrix (twiddle factors, erste Stufe):

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

Und so weiter für $\log_2 N$ Stufen.

Dann erfordert jede solche Multiplikation nur N und $2N$ arithmetische Operationen, und die Gesamtzahl solcher Schritte ist $\log_2 N$.

Insgesamt: Die Multiplikation der Fourier-Matrix mit einem Vektor kann in $\mathcal{O}(N \log_2 N)$ arithmetischen Operationen durchgeführt werden!

Und damit können auch alle s_k in denselben $\mathcal{O}(N \log_2 N)$ berechnet werden, und nicht in $\mathcal{O}(N^2)$.

Für $N = 10^6$:

- Naiver Algorithmus: 10^{12} Operationen,
- FFT: 2×10^7 Operationen (50 000 Mal schneller!).

11.7 Die schnelle Fourier-Transformation (FFT)

Dies ist die berühmte **schnelle Fourier-Transformation** (Fast Fourier Transform, FFT), entdeckt von Cooley und Tukey im Jahr 1965 (obwohl Gauss sie bereits 1805 kannte!).

Sie ist einer der gefragtesten Algorithmen in der modernen Chemie.

11.7.1 Anwendung in der NMR

Mit ihrer Hilfe wird zum Beispiel das ursprüngliche FID (Free Induction Decay) aus der NMR in den Spektralbereich transformiert.

Wenn man sich jede Zeile der Fourier-Matrix ansieht, kann man in ihr eine oszillierende Funktion erkennen:

- Je näher die Zeile an der Mitte der Matrix liegt, desto größer die Oszillation,
- Die erste Zeile hat überhaupt keine — sie ist einfach eine Konstante,
- Die unterste (oder zweite) Zeile hat eine Oszillationsperiode gleich der Länge der Zeile.

Genau deshalb erhalten wir, nachdem wir die Fourier-Matrix mit dem FID multipliziert haben, Maxima dort, wo die Resonanzfrequenzen sind: Es ist einfach so, dass eine solche Zeile in der Oszillationsfrequenz mit der Resonanzfrequenz übereinstimmte.

Warum ist FFT für die NMR so wichtig

Moderne FIDs sind sehr lang — sie werden oft in Hunderttausende und sogar Millionen von Zahlen digitalisiert. Wenn es die schnelle Fourier-Transformation nicht gäbe, würde die Multiplikation mit einer solchen Matrix auf modernen Computern sogar länger dauern als die NMR-Aufnahme selbst — also wirklich Minuten und Zehner von Minuten selbst auf modernen Workstations. Dank der FFT dauert die Transformation **Bruchteile einer Sekunde**.

11.7.2 Weitere Anwendungen der FFT in der Chemie

- **Kryo-Elektronenmikroskopie:** Rekonstruktion von 3D-Strukturen aus 2D-Projektionen.
- **Röntgenstrukturanalyse:** Umwandlung des Beugungsbildes in Elektronendichte.
- **Molekulardynamik:** Berechnung elektrostatischer Wechselwirkungen über die PPPM-Methode (Particle-Particle Particle-Mesh).
- **Signalverarbeitung:** Rauschfilterung, Datenkompression.
- **Chemometrie:** Schnelle Faltung und Korrelation von Spektren.

Fazit

FFT ist einer jener Algorithmen, die die Welt verändert haben. Ohne ihn gäbe es keine moderne Spektroskopie, Signalverarbeitung, Audio- und Videokompression und vieles mehr. Es ist ein Muss für jeden Chemiker, der mit Daten arbeitet.

12 Wenn die FFT das Offensichtliche nicht sieht: Spektrales Leck und die Prony-Methode

12.1 Das Paradoxon: Das Auge sieht eine Sinuskurve, die FFT nicht

Obwohl die FFT ein Muss-Algorithmus fast überall in der experimentellen Chemie ist, hat auch sie ihre Besonderheiten, auf die man unbedingt achten muss.

Betrachten wir ein Beispiel. Wir haben ein Spektrum aufgenommen, das Signal oszilliert, und das ist mit bloßem Auge im Diagramm sichtbar. Aber wir konnten nur etwas mehr als eine Periode aufzeichnen. Als wir die FFT anwendeten, erhielten wir etwas sehr Merkwürdiges: Der Peak scheint da zu sein, aber

auch wieder nicht — er ist irgendwie sehr verschmiert. Und wenn wir viel anderes Rauschen hatten, erhielten wir in einem anderen Teil des Spektrums etwas sehr Ähnliches, und ja, zufällig erwies sich das Rauschen als "heller" als das Signal, und unser Programm lieferte ein völlig falsches Ergebnis.

Aber wir sehen diese Sinuskurve doch mit unseren Augen! Warum sieht die FFT sie nicht?

12.2 Spektrales Leck (Spectral Leakage)

Die FFT ist dann gut, und *nur dann*, wenn wir sie auf ein Signal angewendet haben, in das **genau eine ganze Zahl von Perioden** dieses Signals passt.

Warum? Weil die FFT implizit annimmt, dass unser endliches Signal *periodisch fortgesetzt* wird über das Messfenster hinaus. Wenn genau k volle Perioden in das Fenster passen, dann ist die periodische Fortsetzung glatt, und die FFT liefert einen klaren Peak.

Aber wenn das Signal zum Beispiel 1.4 Perioden enthält, dann entsteht bei der periodischen Fortsetzung ein **Sprung** an der Grenze des Fensters. Die FFT ist gezwungen, diesen Sprung durch eine Vielzahl von Harmonischen zu approximieren — und Energie "leckt" aus dem Hauptpeak in alle anderen Frequenzen. Dieses Phänomen heißt **spektrales Leck** (spectral leakage).

Mathematisch: Wenn die wahre Frequenz des Signals *zwischen* zwei benachbarte Frequenzbins der FFT fällt, dann erhalten wir statt einer Delta-Funktion einen verschmierten "Buckel" (eine Funktion der Form $\sin(x)/x$, die sogenannte *sinc*).

FFT-Regel

Die FFT sieht nur die Frequenzen, die ganzzahlig oft in das Messfenster passen. Alles andere ist Leck.

12.3 Zero-Padding: eine einfache, aber nicht ideale Lösung

Was sollen wir dann tun?

Die einfachste Variante ist, viele Nullen an das Ende des Signals anzuhängen (zero-padding) und zu hoffen, dass wir insgesamt wesentlich mehr Glück haben. Denn wenn wir zum Beispiel Nullen so anhängen, dass die Ausgangsdaten um den Faktor 5 gestreckt werden, dann erhalten wir sicher 7 Perioden (statt 1.4). Und wenn wir sie zum Beispiel um den Faktor 7 vergrößern, dann sind es 9.8 Perioden — auch sehr nahe an einer ganzen Zahl.

Oft hängt man eine unterschiedliche Anzahl von Nullen an und versucht dann, die erhaltenen Spektren zu kombinieren, indem man errät, welche Peaks in den entsprechenden erweiterten Spektren genauer waren. Das hilft wirklich!

Wichtige Nuance des Zero-Padding

Zero-Padding *erhöht nicht die tatsächliche Frequenzauflösung* — es interpoliert nur das Spektrum und macht es glatter. Die Auflösung wird durch die *Länge des Ausgangssignals* bestimmt, nicht durch die Länge des mit Nullen aufgefüllten. Aber in der Praxis hilft Zero-Padding, eine ganze Zahl von Perioden zu "treffen" und das Leck zu verringern.

12.4 Frequenzdrift: wenn das Signal "davon driftet"

Aber es gibt noch ein Problem, mit dem Zero-Padding nicht fertig wird.

Manchmal messen wir lange und mühsam ein Spektrum, und es "ist leicht" davongedriftet. Das heißt, am Anfang hatten wir die Frequenz ω, und am Ende $\omega + \varepsilon$, und diese kleine Korrektur genügt, damit wir mit der FFT völlig danebenliegen und statt eines klaren Einzelpeaks etwas sehr Verschwommenes erhalten.

Aber mit dem Auge sieht man es doch — da ist die Sinuskurve, hier und dort, am Anfang und am Ende! Nur die FFT sieht sie wieder nicht.

Der Grund ist, dass die FFT **Stationarität** des Signals annimmt — also dass sich die Frequenzen mit der Zeit nicht ändern. Wenn die Frequenz driftet, kann keine Harmonische der FFT das gesamte Signal gut approximieren.

Was sollen wir tun?

12.5 Die Prony-Methode: die Idee der Verschiebungen

Auf diese Frage hat uns vor einigen Jahrhunderten Gaspard de Prony (1795) geantwortet. Genauer gesagt, er hat sich ausgedacht, wie man sein eigenes Problem löst (die Entwicklung eines Signals in eine Summe von Exponentialfunktionen), aber sie löst gerade auch unser Problem — wenn das Spektrum in der Zeit leicht davondriftet.

Idee: Wir nehmen das Signal und speichern es in einem Vektor $\mathbf{s} = (s_0, s_1, \ldots, s_{N-1})^T$. Dann verschieben wir diesen Vektor um einen Abtastwert nach unten, dann noch einmal, und stellen ihn wieder daneben. Wir machen das L Mal.

Was haben wir hier?

Wenn wir ein periodisches Signal mit einer unbekannten Sinuskurve $\sin(\omega t)$ hatten, dann erhalten wir nach einer Verschiebung um p Abtastwerte $\sin(\omega t + \omega p)$. Unter Erinnerung an die Additionsformeln aus Kapitel 1:

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

Da $\cos(\omega p)$ und $\sin(\omega p)$ *Konstanten* für den gesamten Vektor sind (sie hängen nicht von t ab), wird die Menge solcher verschobenen Vektoren nur **so viele von Null verschiedene Singulärwerte haben, wie wir das Doppelte der Anzahl verschiedener oszillierender Harmonischer haben**.

Darüber hinaus, wenn eine Frequenz mit der Zeit ein wenig "davongegangen" ist, *hat das keinerlei Einfluss* auf eine solche Approximation, zerstört aber das Ergebnis der FFT völlig, wie wir oben bemerkt haben.

12.6 Prony-Methode, Variante 1: lineare Prädiktion

Unser Signal werde als Summe von K komplexen Exponentialfunktionen modelliert:

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

wobei $z_m = e^{(\alpha_m + \mathrm{i}\omega_m)\Delta t}$ komplexe "Frequenzen" sind (die sowohl die Dämpfung α_m als auch die Oszillation ω_m enthalten) und c_m komplexe Amplituden.

Schlüsseltatsache: Wenn das Signal aus K Exponentialfunktionen besteht und kein Rauschen vorhanden ist, dann erfüllt es eine **lineare Rekursionsbeziehung** der Ordnung K:

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

Das bedeutet, dass der $(n+1)$-te Abtastwert *exakt vorhergesagt* werden kann aus K vorhergehenden!

Die Koeffizienten $a_1, \ldots, a_K$ sind die Koeffizienten des charakteristischen Polynoms, dessen Wurzeln die z_m sind:

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

Wie findet man die Koeffizienten? Schreiben wir die Rekursionsbeziehung für alle verfügbaren Abtastwerte in Matrixform:

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

Dies ist ein überbestimmtes System (wenn $N > 2K$), und wir lösen es mit der Methode der kleinsten Quadrate:

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

Nachdem wir **a** gefunden haben, suchen wir die Wurzeln des Polynoms $P(z)$ — das sind unsere z_m, aus denen die Frequenzen ω_m und Dämpfungen α_m extrahiert werden.

12.7 Prony-Methode, Variante 2: SVD der Verschiebungsmatrix

Die zweite Variante ist rauschresistenter und eleganter.

Schritt 1: Wir konstruieren die Hankel-Matrix aus den Verschiebungen des Signals.

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

der Größe $(N - L + 1) \times L$, wobei $L > K$ (wir wählen L sicher größer als die erwartete Anzahl von Harmonischen).

Schritt 2: Wir machen die SVD.

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

Wenn das Signal aus K Exponentialfunktionen besteht und kein Rauschen vorhanden ist, dann ist $\text{rank}(\mathbf{H}) = K$, und die letzten $L - K$ Singulärwerte sind null:

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

Bei Rauschen sind die kleinen Singulärwerte nicht null, aber sie sind *wesentlich kleiner* als die ersten K. Wir verwerfen sie (dies nennt man **truncated SVD** oder **Regularisierung**).

Schritt 3: Wir extrahieren das Polynom aus dem Nullraum.

Der Singulärvektor $\mathbf{v}_{\text{min}}$, der dem *kleinsten* Singulärwert entspricht (die letzte Spalte von **V**), liegt im (approximativen) Nullraum der Matrix **H**. Das bedeutet:

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

Wenn wir die Komponenten $\mathbf{v}_{\text{min}} = (v_0, v_1, \ldots, v_{L-1})^T$ schreiben, dann ist diese Beziehung äquivalent zu:

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

Das heißt, $\mathbf{v}_{\text{min}}$ enthält die Koeffizienten des Polynoms:

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

Schritt 4: Wir suchen die Wurzeln des Polynoms.

Die Wurzeln von $Q(z)$ enthalten K "Signal"-Wurzeln $z_m = e^{(\alpha_m + \mathrm{i}\omega_m)\Delta t}$ (die innerhalb oder auf dem Einheitskreis liegen) und $L - K$ "Rausch"-Wurzeln (zufällig verstreut).

Aus den Signalwurzeln extrahieren wir:

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

Warum ist die SVD-Variante besser?

Die SVD-Variante der Prony-Methode ist rauschresistenter, weil:

1. Die Abschneidung kleiner Singulärwerte automatisch das Rauschen filtert.
2. Wir lösen das überbestimmte System nicht direkt (was schlecht konditioniert sein kann), sondern verwenden eine orthogonale Zerlegung.
3. Die Anzahl der Harmonischen K wird automatisch aus dem "Sprung" im Spektrum der Singulärwerte bestimmt.

12.8 Grenzen der Prony-Methode

Aber Prony ist kein Allheilmittel.

Sobald im Ausgangssignal eine Komponente auftaucht, die sowohl durch die Funktion selbst als auch durch ihre Verschiebungen gut approximiert wird (zum Beispiel weißes Rauschen oder ein sehr breitbandiges Signal), erhalten wir sofort eine "Wurzel" dafür, und dann stellt sich heraus, dass sie **falsch** ist.

Weitere Einschränkungen:

- Die Methode setzt voraus, dass das Signal eine Summe von *Exponentialfunktionen* ist (einschließlich Sinuskurven als Spezialfall). Wenn das Signal eine andere Struktur hat (zum Beispiel Rechteckimpulse), wird die Methode schlecht funktionieren.
- Die Anzahl der Harmonischen K muss im Voraus bekannt sein oder erraten werden (obwohl die SVD dabei hilft).
- Die Suche nach den Wurzeln eines Polynoms hohen Grades ($L > 50$) kann selbst numerisch instabil sein.
- Die Methode ist empfindlich gegenüber starkem Rauschen: Bei einem niedrigen Signal-Rausch-Verhältnis können falsche Wurzeln sich als echte "maskieren".

Gleichzeitig wird diese Methode in der NMR-Spektroskopie sehr aktiv und seit langem verwendet und ergänzt die FFT gut. In der Praxis verwendet man oft einen **Hybridansatz**:

1. FFT für einen schnellen Überblick über das Spektrum und die Bestimmung der Anzahl der Peaks,
2. Die Prony-Methode (oder ihre modernen Varianten: MUSIC, ESPRIT, Matrix Pencil) zur genauen Bestimmung von Frequenzen, Dämpfungen und Amplituden einzelner Peaks.

Fazit

Die FFT ist ein mächtiges, aber "blindes" Werkzeug: Sie sieht nur, was in ihr Frequenzraster passt. Die Prony-Methode "sieht" Frequenzen zwischen den Bins und ist robust gegenüber Drift, erfordert aber mehr Berechnungen und Vorsicht. In der realen NMR-Spektroskopie arbeiten sie im Tandem.

13 Kleinste Quadrate, Residuen und Tikhonov-Regularisierung

13.1 Problemstellung

Also, wir lösen oft Probleme, die wir Probleme der kleinsten Quadrate nennen. Betrachten wir einige von ihnen, um genauer zu verstehen, wie man das lösen kann.

13.2 Ein Beispiel aus der Medizin: Computertomographie (CT)

Nehmen wir ein Beispiel aus einem benachbarten Bereich — aus der Medizin. Angenommen, wir haben einen Röntgen-CT-Scanner. Wir haben den Patienten (oder das, was wir untersuchen) unbeweglich auf eine Plattform gelegt und stellen um ihn herum auf gegenüberliegenden Seiten einen Sender und Empfänger von Röntgenstrahlen auf. Wir führen Messungen durch, wie stark die Strahlung vom untersuchten Körper absorbiert wurde, und positionieren diese Empfänger und Sender in verschiedenen Winkeln.

Normalerweise liegt der Patient auf einer Liege, und Empfänger und Sender befinden sich auf der Oberfläche eines Rings. Dieser Ring ist so positioniert, dass seine Achse etwa entlang der Liege verläuft, und der Ring selbst rotiert um seine Achse und bewegt sich entlang der Liege.

Was geschieht hier?

Weiches Gewebe absorbiert Röntgenstrahlen kaum, während Knochen ziemlich stark absorbieren.

Wir können den gesamten Raum, in dem sich der Patient befindet, gedanklich in kleine (normalerweise gleiche) Würfel unterteilen — **Voxel** (volume elements). Wenn wir die Position von Sender und Empfänger genau kennen, können wir eine Linie (den Röntgenstrahl) ziehen, die unsere Würfel schneidet.

Jeder solche Würfel habe eine Unbekannte — die Intensität der Absorption von Röntgenstrahlen. Wir nummerieren alle diese Würfel und schreiben die Intensität ihrer Absorption als unbekannten Vektor $\mathbf{x} = (x_1, \ldots, x_N)^T$.

Dann liefert uns eine gegenseitige Anordnung von Sender und Empfänger eine ziemlich dünnbesetzte Menge von Koeffizienten — die Längen des Durchgangs des Strahls durch den entsprechenden Würfel. All dies sei in einer Zeile der Matrix $\mathbf{A}$ geschrieben, und die Anzahl der Spalten in ihr sei N (die Anzahl der Voxel), und die Anzahl der Zeilen sei gerade gleich der Anzahl der durchgeführten Messungen (sie sei M).

Die gemessenen Werte selbst aber (wir können zum Beispiel eine Röntgenquelle und viele Empfänger haben; dann ist jede gegenseitige Anordnung von Empfänger und Quelle eine Zeile) werden im Vektor $\mathbf{b}$ geschrieben, ebenfalls der Länge M.

Dann können wir das Problem der kleinsten Quadrate formulieren als:

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

das heißt, wir wollen, dass jede Zeile der Matrix $\mathbf{A}$, multipliziert mit $\mathbf{x}$ (die Gesamtabsorptionsintensität), gleich dem ist, was wir messen.

13.3 Normalgleichungen

Wir erinnern uns, dass man ein solches Problem lösen kann, aber betrachten wir es sorgfältig.

Die Residuenfunktion:

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

Wenn wir die Ableitung von E nach allen x_j finden und sie null setzen (im Minimum), erhalten wir das Gleichungssystem:

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

Dies sind die sogenannten **Normalgleichungen**.

Hier scheint alles gut: Wir erhalten eine positiv semidefinite Matrix $\mathbf{A}^H\mathbf{A}$ und eine rechte Seite, mit der wir dieses Problem lösen können.

13.4 Das Entartungsproblem

Aber was, wenn wir solche Würfel konstruiert haben, aber keine einzige Messung hatten, bei der der Röntgenstrahl durch zum Beispiel den i-ten Würfel ging?

Offensichtlich wird dann in der Ausgangsmatrix $\mathbf{A}$ die entsprechende i-te Spalte nur Nullen enthalten, und die Matrix $\mathbf{A}^H\mathbf{A}$ wird Nullen sowohl in der i-ten Spalte als auch in der i-ten Zeile haben, also garantiert einen Singulärwert null haben.

Angenommen, dieser Würfel war irgendwo im Raum, und in ihm ist kein Teil des untersuchten Körpers, das heißt — zu unserem Glück — brauchen wir diesen Würfel nicht. Aber eine solche Formulierung *verdirbt* die Lösung: Wir können dieses Problem einfach nicht lösen, da die Matrix $\mathbf{A}^H\mathbf{A}$ entartet sein wird.

Darüber hinaus können wir nicht garantieren, dass selbst wenn $\mathbf{A}^H\mathbf{A}$ keine solchen Nullspalten und -zeilen hat, ihre Kondition gut sein wird. Das heißt, wir können statt einer Lösung völlig falsche Zahlen erhalten.

13.5 Lösung über SVD und Pseudoinverse

Wie soll man in diesem Fall vorgehen?

Kehren wir zur Singulärwertzerlegung der Matrix $\mathbf{A} = \mathbf{U}\square\mathbf{V}^H$ zurück. Wir können bemerken, dass:

- Wenn in $\square$ Nullen auf der Diagonalen sind, beziehen sie sich auf jene Würfel, durch die der Röntgenstrahl nicht ging.
- Wenn etwas sehr Kleines vorhanden ist, bezieht es sich auf solche Experimente, bei denen der Röntgenstrahl nur einige Male und möglicherweise nur streifend in einen Würfel oder eine Menge von Würfeln gelangte.

Das heißt, wir sollten solche Daten nicht berücksichtigen, aber sie sind sehr schwer bereits bei der Bildung der Matrix $\mathbf{A}$ herauszufiltern.

Aber da diese Daten wenig informativ sind, lass sie uns einfach "wegwerfen"! Das heißt, führen wir diese SVD aus und nullen die kleinen Diagonalwerte in $\square$ und rekonstruieren $\mathbf{A}$.

Das ist natürlich gut, aber wir wissen immer noch nicht, wie wir ein solches Problem lösen sollen, da $\mathbf{A}^H\mathbf{A}$ ebenfalls Null-Singulärwerte enthalten wird.

Aber wenn man für ein quadratisches nicht entartetes System $\mathbf{Ax} = \mathbf{b}$ die Lösung darstellen kann als:

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

dann könnten wir unser Problem der kleinsten Quadrate wohl analog lösen. Zwar werden wir dort, wo $\square$ Nullen hat, in $\square^{-1}$ sie nicht invertieren, sondern dort ebenfalls eine Null eintragen.

Dann wäre es falsch, sie invers zu nennen, und wir nennen sie **Pseudoinverse**, bezeichnet als $\square^+$ (oder $\mathbf{A}^+$ für die gesamte Matrix).

13.6 Das Größenproblem: wenn die SVD nicht in den Speicher passt

Alles wäre gut, nur ist in realen Problemen die Matrix $\mathbf{A}$ oft riesig. Häufig werden ihre Elemente analytisch berechnet, und es besteht keine Notwendigkeit, sie zu speichern. Und die dichte Matrix $\mathbf{U}$, deren Dimension gleich der von $\mathbf{A}$ ist, passt normalerweise nicht in den Speicher.

Zum Beispiel verwendet man für einen gewöhnlichen CT-Scan Würfel von etwa 1 mm Größe. Pro Sekunde erfolgen etwa 100–500 Messungen an 100–500 Empfängern, und die gesamte Messung dauert etwa eine Minute. Das heißt, N und M können leicht in der Größenordnung von 10–100 Millionen liegen, und die vollständige Singulärmatrix würde 100 Petabyte Speicher erfordern, was unvernünftig viel ist.

13.7 Tikhonov-Regularisierung

Man kann bemerken, dass, wenn man zur entarteten Matrix $\mathbf{A}^H\mathbf{A}$ die Einheitsmatrix $\lambda\mathbf{I}$ addiert, sodass λ im Bereich jener sehr kleinen Singulärwerte liegt, die wir genullt haben, dann diese Addition geschrieben werden kann als:

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

Und wir sehen, dass wir dieses Lambda zu jedem Singulärwert addieren und so eine entartete oder schlecht konditionierte Matrix in eine gut konditionierte "verwandeln".

Darüber hinaus ändern wir bei kleinem λ (in der Größenordnung der kleinen Singulärwerte) die großen Singulärwerte fast nicht, ersetzen aber die kleinen einfach durch λ.

Wenn wir dann das System mit einer solchen Matrix lösen, "verderben" wir die großen Singulärwerte einfach nicht und reduzieren die Antwort der kleinen Singulärwerte wesentlich.

Also ist die Kondition der Matrix wesentlich kleiner geworden, und wir "verderben" die Lösung nicht durch die Matrix. Da wir wissen, dass die Matrix dünnbesetzt ist, können wir einige iterative Methoden anwenden (sogar das Verfahren der konjugierten Gradienten) und sehr schnell konvergieren.

Tatsächlich ist ein solches λ der sogenannte **Tikhonov-Regularisierer**, denn wir können statt des Ausgangsproblems das folgende lösen:

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

wobei wir faktisch verlangen, dass sowohl das Residuum als auch die Norm der Lösung gleichzeitig minimiert werden und ihr gegenseitiges Verhältnis gerade durch den Wert dieses λ geregelt wird.

Hier gibt es viel schöne Theorie, die einst in den 1970er Jahren von Tikhonov und seinen Nachfolgern hergeleitet wurde, aber der Kern bleibt einfach: Ein solcher zusätzlicher "Regularisierer" hilft, **schlecht gestellte** (ill-posed) Probleme zu lösen.

13.8 Iterative Verkleinerung von λ

Darüber hinaus setzt man oft zunächst λ ziemlich groß, dann konvergiert der konjugierte Gradient sehr schnell (die Kondition der Matrix wird sehr klein sein), und dann verkleinert man λ, wobei man jedes Mal die vorherige Lösung als Startwert verwendet.

Dies erlaubt:

- Schnell eine "grobe" Lösung mit großem λ zu erhalten,
- Sie allmählich zu verfeinern, indem man λ verkleinert,
- Probleme mit der Kondition in frühen Phasen zu vermeiden.

Fazit

Die Tikhonov-Regularisierung ist ein mächtiges Werkzeug zur Lösung schlecht konditionierter Probleme der kleinsten Quadrate. Sie fügt eine Strafe für die Norm der Lösung hinzu, was die Lösung stabilisiert und die Verwendung iterativer Methoden auch für entartete Systeme erlaubt.

14 Basisfunktionen: von finiten Differenzen zu Splines

14.1 Die Idee: das Unbekannte durch Bekanntes approximieren

Bisher haben wir mit diskreten Vektoren und Matrizen gearbeitet. Aber reale Probleme — Differentialgleichungen, Integrale, Wellenfunktionen — leben im kontinuierlichen Raum. Wie reduzieren wir ein unendlich-dimensionales Problem auf ein endlich-dimensionales?

Antwort: **Basisfunktionen**. Wir stellen die unbekannte Funktion $f(x)$ als Linearkombination bekannter Basisfunktionen $\phi_k(x)$ dar:

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

und suchen die Koeffizienten c_k. Dies verwandelt das Problem der Suche nach einer Funktion in das Problem der Suche nach einem Vektor $\mathbf{c} \in \mathbb{R}^N$.

14.2 Das Ritz-Verfahren (Variationsverfahren)

Einer der allgemeinsten Ansätze ist das **Ritz-Verfahren**. Angenommen, wir haben ein Funktional $E[f]$, das wir minimieren wollen (zum Beispiel die Energie in der Quantenmechanik). Wir setzen die Entwicklung ein:

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

und erhalten eine Funktion der Koeffizienten:

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

Nun minimieren wir $E(\mathbf{c})$ über $\mathbf{c}$ — das ist bereits ein gewöhnliches Optimierungsproblem in $\mathbb{R}^N$.

14.3 Finite Differenzen (Finite Difference Method, FDM)

Die einfachste Wahl von Basisfunktionen sind **lokale Konstanten** oder **lineare Funktionen** auf einem äquidistanten Gitter.

14.3.1 Ein Beispiel aus der Chemie: Diffusion

Angenommen, wir haben die Diffusionsgleichung für die Konzentration eines Stoffes $C(x,t)$:

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

Wir diskretisieren den Raum mit Schrittweite h: $x_j = jh$, und die Zeit mit Schrittweite Δt: $t_n = n\Delta t$.

Die zweite Ableitung approximieren wir durch die zentrale Differenz:

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

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

Dann das Euler-Schema in der Zeit:

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

Dies ergibt eine explizite Rekursionsformel:

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

Stabilität

Das explizite Schema ist nur stabil für $\frac{D\Delta t}{h^2} \leq \frac{1}{2}$. Wenn der Zeitschritt zu groß ist — "zerfällt" die Lösung.

14.4 Finite Elemente (Finite Element Method, FEM)

Ein komplizierterer, aber flexiblerer Ansatz sind **finite Elemente**. Wir unterteilen das Gebiet in kleine Elemente (Dreiecke, Tetraeder) und approximieren auf jedem Element die Funktion durch **lokale Polynome** (üblicherweise linear oder quadratisch).

Die Basisfunktionen sind **stückweise lineare "Hüte"** (hat functions), von denen jede an einem Knoten gleich 1 und an allen anderen gleich 0 ist.

Vorteile der FEM:

- Kann mit komplexen Geometrien arbeiten,
- Kann das Gitter anpassen (verdichten, wo sich die Lösung schnell ändert),
- Gut theoretisch fundiert.

14.5 Basisfunktionen in der Quantenchemie: Gauß-Orbitale

Und nun — das Interessanteste für Chemiker. In der Quantenchemie (Hartree–Fock-Verfahren, DFT) lösen wir die Schrödinger-Gleichung:

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

wobei $\hat{H}$ der Hamilton-Operator und Ψ die Wellenfunktion ist.

Wir stellen die Molekülorbitale $\psi_i(\mathbf{r})$ als Linearkombination von **Atomorbitalen** dar (LCAO — Linear Combination of Atomic Orbitals):

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

wobei $\phi_\mu(\mathbf{r})$ die an den Atomen zentrierten Basisfunktionen sind.

14.5.1 Gauß-Funktionen (Gaussian Type Orbitals, GTO)

In den meisten modernen Programmen (Gaussian, ORCA, Q-Chem) verwendet man als Basisfunktionen **Gauß-Orbitale**:

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

wobei:

- $\mathbf{R}_A = (X_A, Y_A, Z_A)$ die Koordinaten des Atoms A sind,
- l, m, n die Drehimpulsquantenzahlen (s, p, d, f-Orbitale) sind,
- α der Exponent ist (kontrolliert die "Größe" des Orbitals).

Warum gerade Gauß-Funktionen?

- **Das Produkt von Gauß-Funktionen ist wieder eine Gauß-Funktion:** Dies ist entscheidend für die Berechnung von Mehrzentrenintegralen (Elektron-Elektron-Abstoßung),
- **Analytische Integrale:** Alle Integrale werden analytisch berechnet,
- **Kontraktionen:** Reale Atomorbitale werden durch eine Summe mehrerer Gauß-Funktionen (Kontraktionen) approximiert, was die Genauigkeit erhöht.

14.5.2 Das Ritz-Verfahren in der Quantenchemie

Wir setzen die LCAO-Entwicklung in das Hartree–Fock- oder DFT-Energiefunktional ein und minimieren über die Koeffizienten $c_{\mu i}$. Dies führt auf das Eigenwertproblem:

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

wobei:

- $\mathbf{F}$ die Fock-Matrix (oder die Kohn–Sham-Matrix in DFT) ist,
- $\mathbf{S}$ die Überlappungsmatrix ist ($S_{\mu\nu} = \langle\phi_\mu|\phi_\nu\rangle$),

- ε_i die Orbitalenergien sind,
- $\mathbf{c}_i$ die Entwicklungskoeffizienten des i-ten Molekülorbitals sind.

Dies ist ein verallgemeinertes Eigenwertproblem, und es wird iterativ gelöst (mit der Methode des selbstkonsistenten Feldes, SCF).

Basissätze

Beliebte Basissätze in der Quantenchemie:

- **STO-3G:** Minimalbasis (3 Gauß-Funktionen pro Atomorbital),
- **6-31G*:** Double-Zeta-Qualität mit Polarisationsfunktionen,
- **cc-pVTZ:** Korrelationskonsistente Triple-Zeta (hohe Genauigkeit).

Je größer der Basissatz — desto genauer das Ergebnis, aber desto teurer die Berechnungen ($\mathcal{O}(N^4)$ für Hartree–Fock, wobei N die Anzahl der Basisfunktionen ist).

14.6 Splines: glatte stückweise polynomielle Funktionen

Und nun — über Splines. Dies sind Basisfunktionen, die in der Signalverarbeitung, Computergrafik und numerischen Methoden weit verbreitet sind.

14.6.1 Was ist ein Spline?

Ein **Spline** ist eine stückweise polynomielle Funktion, die an den Verbindungsstellen **glatt** ist (stetige Ableitungen bis zu einer bestimmten Ordnung hat).

Der beliebteste ist der **kubische Spline** (Grad 3, Glattheit C^2 — die Funktion, die erste und die zweite Ableitung sind stetig).

14.6.2 B-Splines (Basis Splines)

B-Splines sind ein spezieller Satz von Basissplines mit **kompaktem Träger** (compact support). Jeder B-Spline ist nur auf einem kleinen Intervall von null verschieden.

Vorteile:

- **Lokalität:** Die Änderung eines Koeffizienten beeinflusst nur einen kleinen Bereich,
- **Numerische Stabilität:** Die Matrizen sind gut konditioniert,
- **Rekursive Definition:** B-Splines werden rekursiv konstruiert (Formel von de Boor).

14.6.3 Zweidimensionale Splines

B-Splines lassen sich leicht auf den zweidimensionalen Fall über das **Tensorprodukt** verallgemeinern:

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

Dies wird verwendet in:

- Bildverarbeitung (Glättung, Interpolation),
- Finiten Elementen (höhere Ordnungen),
- Computergrafik (Oberflächen).

14.6.4 Bézier-Splines

Bézier-Splines sind parametrische Kurven, die durch **Kontrollpunkte** definiert werden. Die Kurve verläuft nicht durch die Kontrollpunkte, sondern wird von ihnen "angezogen".

Die Formel einer Bézier-Kurve vom Grad n:

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

wobei $\mathbf{P}_i$ die Kontrollpunkte sind und $\binom{n}{i}(1-t)^{n-i}t^i$ die **Bernstein-Polynome**.

Anwendungen:

- Computergrafik (Vektorgrafik, TrueType-Schriften),
- CAD-Systeme (AutoCAD, SolidWorks),
- Animation und Trajektorien.

14.6.5 Zusammenhang mit finiten Elementen

Interessanterweise gehen B-Splines mit kompaktem Träger **fließend** in Basisfinite Elemente über. Nimmt man B-Splines vom Grad 0 — das sind stückweise konstante Funktionen (wie in den einfachsten finiten Elementen). Grad 1 — stückweise lineare "Hüte". Grad 2 und höher — glattere Basen.

Dies erlaubt die Konstruktion **isoparametrischer finiter Elemente** höherer Ordnung, die bei weniger Knoten eine genauere Approximation liefern.

Fazit

Basisfunktionen sind die Brücke zwischen der kontinuierlichen Welt der Differentialgleichungen und der diskreten Welt der Computer. Finite Differenzen sind die einfachsten, finite Elemente die flexibelsten, Gauß-Orbitale sind auf die Quantenchemie spezialisiert, und Splines sind für glatte Interpolation und Grafik. Das Verständnis ihrer Eigenschaften ist der Schlüssel zur Wahl der richtigen Methode für dein Problem.

15 Integration: von der Analytik zu Monte Carlo

15.1 Wozu brauchen wir Integration?

Das Integral ist eine der fundamentalsten Operationen in Mathematik und Physik. Wir berechnen ständig:

- Flächen und Volumina,
- Erwartungswerte und Varianzen,
- Normierungskonstanten in der Quantenmechanik,
- Mehrdimensionale Integrale in der statistischen Physik,
- Faltungen in der Signalverarbeitung.

Und wenn wir in der Schule gelernt haben, Integrale analytisch zu berechnen, dann endet die Analytik im wirklichen Leben — besonders in Chemie und Physik — oft sehr schnell.

15.2 Analytische Integration: wann sie funktioniert

Erinnern wir uns an die Grundlagen. Wenn wir eine analytisch gegebene Funktion $f(x)$ haben und ihre Stammfunktion $F(x)$ kennen, dann gilt:

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

Zum Beispiel:

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

Schön, exakt, schnell. Aber:

- Nicht alle Funktionen haben eine elementare Stammfunktion (zum Beispiel e^{-x^2} — das Fehlerintegral),
- Nicht alle Funktionen sind analytisch gegeben — oft haben wir nur eine Menge von Punkten (experimentelle Daten),
- In mehrdimensionalen Fällen ist die Analytik fast immer machtlos.

15.3 Numerische Integration in 1D: das Simpson-Verfahren

Was tun, wenn das Integral analytisch nicht berechnet werden kann? Antwort: die Funktion durch etwas Einfaches approximieren und dieses Einfache integrieren.

Im vorigen Kapitel haben wir gelernt, kubische Splines zu konstruieren. Und hier konstruieren wir ebenfalls Polynome und integrieren mit solchen Stücken.

15.3.1 Rechteckverfahren

Der einfachste Ansatz: Wir teilen das Intervall $[a, b]$ in N gleiche Teile der Breite $h = (b - a)/N$ und approximieren die Funktion auf jedem Teilintervall durch eine Konstante (den Wert in der Mitte):

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

Der Fehler ist $\mathcal{O}(h^2)$.

15.3.2 Trapezverfahren

Etwas besser: Wir approximieren die Funktion auf jedem Intervall durch ein lineares Polynom:

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

Der Fehler ist $\mathcal{O}(h^2)$.

15.3.3 Simpson-Verfahren

Noch besser: Wir approximieren die Funktion auf jedem Paar von Intervallen durch ein **quadratisches Polynom**:

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

wobei N eine gerade Zahl ist.

Der Fehler ist $\mathcal{O}(h^4)$! Das ist bereits sehr gut für glatte Funktionen.

Warum ist Simpson so gut?

Das Simpson-Verfahren ist im Wesentlichen die Integration stückweise quadratischer Splines. Wenn die Funktion glatt ist (stetige Ableitungen bis zur 4. Ordnung), dann nimmt der Fehler wie h^4 ab, was viel schneller ist als bei Trapezen.
Für sehr glatte Funktionen gibt es noch genauere Methoden — Gauß-Quadraturen, die optimal gewählte Knoten und Gewichte verwenden.

15.4 Der Fluch der Dimension

Aber was, wenn wir eine 2-, 3-, 4-, 10-dimensionale Funktion integrieren?
Dann brauchen wir N^{10} Integrationspunkte, wobei N mindestens 100 ist... Das ist irgendwie viel!
Rechnen wir. Für ein 10-dimensionales Integral mit $N = 100$ Punkten entlang jeder Achse:

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

Selbst wenn jeder Punkt in 1 Nanosekunde berechnet wird, dauert das:

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

Dies ist der **Fluch der Dimension** (curse of dimensionality).

15.4.1 Wo kommt das in der Chemie vor?

- **Quantenchemie:** Berechnung von Mehrelektronenintegralen (Elektron-Elektron-Abstoßung) — das sind 6-dimensionale Integrale (3 Koordinaten pro Elektron).
- **Statistische Mechanik:** Die Zustandssumme ist ein Integral über den Phasenraum aller Teilchen. Für N Teilchen — das ist ein $6N$-dimensionales Integral (3 Koordinaten + 3 Impulse pro Teilchen).
- **Molekulardynamik:** Mittelung über Konfigurationen — mehrdimensionale Integration.
- **Maschinelles Lernen:** Bayessche Inferenz — Integration über den Raum der Modellparameter.

15.5 Die Monte-Carlo-Methode zur Integration

Und hier betritt die Monte-Carlo-Methode die Bühne — einer der elegantesten und überraschendsten Algorithmen der numerischen Mathematik.

15.5.1 Die Idee im Kern

Stellen wir uns vor, wir wollen das Integral berechnen:

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

wobei d die Dimension ist (kann sehr groß sein).

Die Monte-Carlo-Methode sagt: “Lass uns einfach N zufällige Punkte $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N$ gleichmäßig in das Integrationsgebiet werfen, die Funktion an ihnen auswerten und mitteln”.

Die Formel:

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

wobei $V = (b-a)^d$ das Volumen des Integrationsgebietes ist und $\mathbf{x}_k$ zufällige, in $[a,b]^d$ gleichmäßig verteilte Punkte sind.

15.5.2 Warum funktioniert das?

Dies ist einfach das Gesetz der großen Zahlen. Der Erwartungswert der Zufallsvariablen $f(\mathbf{X})$, wobei $\mathbf{X}$ in $[a,b]^d$ gleichmäßig verteilt ist, ist:

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

Nach dem Gesetz der großen Zahlen konvergiert der Mittelwert $\frac{1}{N}\sum f(\mathbf{x}_k)$ gegen $\mathbb{E}[f(\mathbf{X})]$ für $N \to \infty$.

15.5.3 Konvergenzgeschwindigkeit

Und hier — Magie! Der Fehler der Monte-Carlo-Methode nimmt ab wie:

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

wobei σ die Standardabweichung der Funktion $f(\mathbf{x})$ im Integrationsgebiet ist.

Wichtig: Diese Konvergenzgeschwindigkeit *hängt nicht von der Dimension* d ab!

Zum Vergleich:

- Simpson-Verfahren in 1D: Fehler $\sim h^4 \sim N^{-4}$,
- Simpson-Verfahren im d-dimensionalen Fall: Fehler $\sim N^{-4/d}$ (katastrophal langsam für große d),
- Monte-Carlo-Methode in beliebiger Dimension: Fehler $\sim N^{-1/2}$.

Ja, Monte Carlo konvergiert langsamer als Simpson in 1D. Aber im 10-dimensionalen Fall:

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

Monte Carlo ist *schneller*!

Der Preis von Monte Carlo

Die Monte-Carlo-Methode konvergiert wie $N^{-1/2}$ — das bedeutet, um die Genauigkeit um den Faktor 10 zu erhöhen, braucht man 100-mal mehr Punkte. Das ist nach absoluten Maßstäben langsam, aber es ist die *einzige* Methode, die in hohen Dimensionen funktioniert.

15.6 Ein Beispiel aus der Quantenchemie: Berechnung von Orbitalen

Betrachten wir, wie die Monte-Carlo-Methode in der realen Quantenchemie angewendet wird.

15.6.1 Aufgabe: Normierung der Wellenfunktion

In der Quantenmechanik beschreibt die Wellenfunktion $\Psi(\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_N)$ den Zustand von N Elektronen. Sie muss normiert sein:

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

Das ist ein $3N$-dimensionales Integral! Für ein Wassermolekül ($N = 10$ Elektronen) — das ist ein 30-dimensionales Integral.

15.6.2 Anwendung von Monte Carlo

Wir erzeugen M zufällige Elektronenkonfigurationen:

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

wobei jede Koordinate $\mathbf{r}_i^{(k)} = (x_i^{(k)}, y_i^{(k)}, z_i^{(k)})$ aus einer Verteilung (zum Beispiel einer Gauß-Verteilung) gewählt wird.

Wir berechnen $|\Psi|^2$ in jeder Konfiguration und mitteln:

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

15.6.3 Variational Monte Carlo (VMC)

Aber das ist noch nicht alles! In der Quantenchemie gibt es die **Variational Monte Carlo (VMC)**-Methode, die Monte Carlo zur *Minimierung* der Energie verwendet.

Wir wählen eine Testwellenfunktion $\Psi_T(\mathbf{r}_1, \ldots, \mathbf{r}_N; \alpha)$ mit Parametern α und berechnen die Energie:

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

Beide Integrale sind $3N$-dimensional, und wir berechnen sie mit Monte Carlo. Dann minimieren wir $E(\alpha)$ über die Parameter α — und erhalten eine approximative Wellenfunktion.

15.6.4 Quantum Monte Carlo (QMC)

Es gibt noch fortgeschrittenere Methoden — **Diffusion Monte Carlo (DMC)**, die die Schrödinger-Gleichung direkt lösen, indem sie die "Diffusion" von Elektronen in imaginärer Zeit modellieren. Diese Methoden liefern *fast exakte* Lösungen für kleine Moleküle und werden als Referenz zur Prüfung anderer Methoden verwendet.

Wo wird Monte Carlo in der Chemie verwendet?

- **Quantum Monte Carlo (QMC):** Exakte Lösung der Schrödinger-Gleichung für kleine Systeme,
- **Monte-Carlo-Molekulardynamik:** Sampling von Konfigurationen in der statistischen Mechanik,
- **Integration in DFT:** Numerische Integration des Austausch-Korrelations-Funktionals,
- **Bayessche Optimierung:** Suche nach dem globalen Minimum der Energie eines Moleküls.

15.7 Verbesserungen der Monte-Carlo-Methode

Die grundlegende Monte-Carlo-Methode ist einfach eine gleichmäßige Zufallswanderung. Aber es gibt viele Verbesserungen:

15.7.1 Importance Sampling

Statt einer gleichmäßigen Verteilung verwenden wir eine Verteilung proportional zu $|f(\mathbf{x})|$. Dann fallen die Punkte häufiger in Bereiche, in denen die Funktion groß ist, und der Fehler verringert sich.

15.7.2 Metropolis-Verfahren (Metropolis-Hastings)

Für komplizierte mehrdimensionale Integrale verwenden wir Markov-Ketten: Jeder nächste Punkt hängt vom vorherigen ab. Dies erlaubt eine effiziente Erkundung des Raums, auch wenn er sehr groß ist.

15.7.3 Quasi-Monte-Carlo

Statt zufälliger Punkte verwenden wir **deterministische** Folgen mit geringer Diskrepanz (Sobol-, Halton-Folgen). Sie bedecken den Raum "gleichmäßiger" als zufällige Punkte und konvergieren schneller: Fehler $\sim N^{-1}(\log N)^d$.

Fazit

Die Monte-Carlo-Methode ist die einzige praktikable Möglichkeit zur Berechnung mehrdimensionaler Integrale. Ihre Konvergenzgeschwindigkeit hängt nicht von der Dimension ab, was sie in der Quantenchemie, der statistischen Physik und im maschinellen Lernen unverzichtbar macht. Obwohl sie in niedrigen Dimensionen langsamer konvergiert als deterministische Methoden, hat sie in hohen Dimensionen einfach keine Konkurrenz.

16 Statistik: wie man Information vom Rauschen trennt

Bisher haben wir hauptsächlich mathematische Aufgaben betrachtet, in denen die Zahlen als bekannt gelten. Aber die experimentelle Chemie ist anders eingerichtet. Wenn wir die Konzentration eines Stoffes, die Lage einer Spektrallinie oder die Intensität eines Signals messen, gibt das Gerät uns niemals den "wahren" Wert mit unendlicher Genauigkeit. Jede Messung unterscheidet sich ein wenig von der anderen. Deshalb haben wir es statt mit einer einzigen Zahl mit einer **Zufallsvariablen** zu tun.

Angenommen, wir messen dieselbe Größe mehrere Male. Wir erhalten: $(x_1, x_2, \ldots, x_N) = \mathbf{x}$, wobei $\mathbf{x} = \mu + \varepsilon$, und

- μ der unbekannte wahre oder mittlere Wert ist;
- ε der zufällige Messfehler ist.

Diese einfache Gleichung ist eine der grundlegenden Ideen der Statistik.

Statistik beginnt dort, wo wir verstehen: *Wir beobachten Daten, interessieren uns aber für verborgene Größen.*

16.1 Der Mittelwert

Die einfachste Schätzung von μ ist das arithmetische Mittel:

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

Warum gerade dieses? Wenn die Fehler einen Mittelwert nahe null haben, dann minimiert das arithmetische Mittel die Summe der quadrierten Abweichungen: $\min_\mu \sum_{i=1}^{N} |x_i - \mu|^2$, und die Lösung ist genau durch die obige Formel gegeben. Übrigens erklärt dies bereits, warum die Wiederholung eines Experiments gewöhnlich die Genauigkeit des Ergebnisses erhöht.

16.2 Varianz und Standardabweichung

Der Mittelwert sagt uns, wo das Zentrum der Daten liegt, sagt aber nichts darüber, wie stark die Ergebnisse streuen. Dafür führt man die Varianz ein — ein Maß dafür, wie stark unsere Messungen um den wahren Wert (zu dem der Mittelwert strebt) "streuen":

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

Ja, wir haben sie doch gerade minimiert, oder? Richtig, aber der Wert des Minimums wird genau dann null sein, wenn alle Werte x_i gleich waren; andernfalls — ist dieser Wert gerade die Varianz. Und um sie in denselben Einheiten zu messen, erinnern wir uns einfach, dass die zweite Norm eine Wurzel enthält, und schreiben

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

Das heißt, die Varianz ist im Wesentlichen das **Quadrat der Länge des Vektors der Abweichungen vom Mittelwert**. Die Statistik verwandelt sich wieder in lineare Algebra.

16.3 Die Normalverteilung

In vielen physikalischen und chemischen Messungen werden zufällige Fehler näherungsweise durch die Normalverteilung beschrieben:

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

Es ist nicht notwendig, sich diese Formel zu merken. Es ist viel wichtiger, die Bedeutung der beiden Parameter zu verstehen: μ und σ. Der erste bestimmt die Lage des Zentrums der Verteilung, der zweite — ihre Breite, das heißt, je größer σ, desto stärker die Streuung der Messungen.

Intuition

- Der Mittelwert beantwortet die Frage: *"Wo befindet sich das Ergebnis?"*
- Die Standardabweichung beantwortet die Frage: *"Wie stark streuen die Ergebnisse?"*

16.4 Warum wird der Mittelwert bei Wiederholung des Experiments genauer?

Angenommen, wir haben N unabhängige Messungen gemacht. Für unabhängige Fehler ist die Varianz des Mittelwerts $\frac{\sigma^2}{N}$, folglich **verringert sich der Fehler des Mittelwerts wie** $\frac{\sigma}{\sqrt{N}}$. Deshalb erhöht die Vergrößerung der Anzahl der Messungen die Genauigkeit, aber nicht linear (wie bei Monte Carlo!).

Um den zufälligen Fehler um den Faktor 10 zu verringern, braucht man im idealisierten Fall etwa 100 Mal mehr unabhängige Messungen.

16.5 Zufälliger Fehler und systematischer Fehler

Hier muss eine sehr wichtige Unterscheidung getroffen werden. Wenn das Gerät $x = \mu + \varepsilon$ liefert, wobei ε zufällig um null schwankt, können wiederholte Messungen den Einfluss dieses Fehlers verringern. Aber stellen wir uns vor, das Gerät überschätzt das Ergebnis systematisch: $x = \mu + b + \varepsilon$, wobei b ein konstanter

Versatz ist. Dann ergibt die Mittelung $\bar{x} \approx \mu + b$. Und so oft wir das Experiment auch wiederholen, der Versatz b wird nicht verschwinden.

Wichtig

Die Wiederholung von Messungen verringert den zufälligen Fehler, beseitigt aber nicht den systematischen Fehler.
Die Statistik kann ein falsch kalibriertes Gerät nicht korrigieren.

Dies ist einer der Gründe, warum in der Chemie Kalibrierung, Kontrollproben und unabhängige Messmethoden so wichtig sind.

16.6 Zwei Größen können zusammenhängen

Angenommen, zwei Größen werden gleichzeitig gemessen: x_i und y_i. Zum Beispiel:

- die Konzentration eines Stoffes und die Intensität eines Signals;
- Temperatur und Reaktionsgeschwindigkeit;
- Druck und Volumen;
- zwei spektrale Charakteristiken.

Uns kann die Frage interessieren: *Ändern sich x und y übereinstimmend?*
Dafür verwendet man die Kovarianz: $\mathrm{Cov}(x, y) = \frac{1}{N}(x - \mu_x)^T(y - \mu_y)$
Wenn große Werte von x gewöhnlich großen Werten von y entsprechen, ist die Kovarianz positiv. Wenn große Werte von x kleinen y entsprechen, ist sie negativ. Wenn es keinen linearen Zusammenhang gibt, kann die Kovarianz nahe null sein.

16.7 Korrelation

Die Kovarianz hängt von den Maßeinheiten ab. Deshalb verwendet man oft die normierte Größe: $\rho_{xy} = \frac{\mathrm{Cov}(x, y)}{\sigma_x \sigma_y}$. Für sie gilt $-1 \leq \rho_{xy} \leq 1$. Ein Wert nahe 1 bedeutet einen starken positiven linearen Zusammenhang, ein Wert nahe -1 — einen starken negativen, und ein Wert nahe 0 bedeutet das Fehlen einer ausgeprägten linearen Korrelation.

Aber hier muss man sich eine sehr wichtige Sache merken:

Korrelation bedeutet nicht Kausalität

Wenn zwei Größen korrelieren, bedeutet das noch nicht, dass die eine die andere verursacht.
Die Korrelation spricht von einem statistischen Zusammenhang zwischen den Daten. Um einen kausalen Zusammenhang festzustellen, sind zusätzliche physikalische, chemische oder experimentelle Argumente nötig.

16.8 Die Kovarianzmatrix

Wenn wir nicht zwei, sondern p gemessene Größen haben, $x_1, x_2, \ldots, x_p$, dann können die Kovarianzen aller Paare in einer einzigen Matrix gesammelt werden:

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

Diese Matrix ist symmetrisch: $\Sigma = \Sigma^T$, und sie speichert nicht nur statistische Information. Sie ist ein mathematisches Objekt der linearen Algebra. Genau deshalb tauchen Eigenwerte und Eigenvektoren, denen wir bereits früher begegnet sind, wieder in der Statistik auf.

16.9 PCA: Statistik trifft lineare Algebra

Stellen wir uns vor, wir haben N chemische Proben, für die jeweils p Merkmale gemessen wurden. Wir erhalten eine Datenmatrix: $X \in \mathbb{R}^{N\times p}$. Einige Merkmale können stark miteinander zusammenhängen. Wenn zum Beispiel zwei gemessene Größen sich fast immer gemeinsam ändern, ist die Information über sie teilweise redundant. Es wäre wünschenswert, neue Koordinaten zu finden, in denen:

- die erste Koordinate die maximal mögliche Variation der Daten enthält;
- die zweite — die maximal mögliche verbleibende Variation;
- die dritte — die nächste, und so weiter.

Dies ist die Grundidee der **Hauptkomponentenanalyse** (Principal Component Analysis, PCA). Mathematisch ist PCA eng mit den Eigenvektoren der Kovarianzmatrix verbunden: $\Sigma v_i = \lambda_i v_i$. Die Eigenvektoren v_i geben neue Richtungen an, und die Eigenwerte λ_i zeigen, wie viel Variation der Daten auf die entsprechende Richtung entfällt.

Wenn die ersten wenigen Eigenwerte wesentlich größer sind als die übrigen,

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

dann kann der größte Teil der Information durch nur wenige Koordinaten dargestellt werden.

Zum Beispiel,

$$\text{5000 gemessene Parameter} \quad \longrightarrow \quad \text{20 Hauptkomponenten.}$$

Das bedeutet nicht, dass wir chemische Information "weggeworfen" haben. Wir haben eine kompaktere mathematische Darstellung der Daten gefunden.

Und hier erscheint wieder die SVD, der wir bereits begegnet sind:

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

PCA und SVD erweisen sich als zwei Seiten derselben linear-algebraischen Idee.

16.10 Regression: wenn wir ein Modell bauen wollen

Angenommen, wir haben die Konzentration c_i und das entsprechende Signal y_i gemessen. Wir nehmen ein lineares Modell an:

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

Da die Messungen Fehler enthalten, liegen die Punkte gewöhnlich nicht genau auf einer einzigen Geraden. Deshalb suchen wir solche a und b, die die Summe der Fehlerquadrate minimieren:

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

Dies ist die **Methode der kleinsten Quadrate**. Somit ist die Regression nichts völlig Neues. Wir kennen diese mathematische Konstruktion bereits:

$$\boxed{\text{Daten} \to \text{Modell} \to \text{Residuum} \to \text{Minimierung}}$$

Genau deshalb sind lineare Algebra und Optimierung für die Statistik so wichtig.

16.11 Überanpassung

Nun entsteht ein komplizierteres Problem. Wenn es wenige Daten gibt, können wir eine sehr komplizierte Funktion wählen, die praktisch durch jeden experimentellen Punkt verläuft. Zum Beispiel kann man statt einer Geraden ein Polynom hohen Grades verwenden:

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

Bei genügend großem k kann man die vorhandenen Messungen fast perfekt beschreiben. Aber das bedeutet nicht unbedingt, dass wir das richtige physikalische Gesetz gefunden haben. Wir haben uns möglicherweise einfach an das Rauschen angepasst. Dies nennt man **Überanpassung** (overfitting).

Genau deshalb interessiert uns in Statistik und maschinellem Lernen nicht nur die Frage: *"Wie gut beschreibt das Modell die bekannten Daten?"* sondern auch: *"Wie gut funktioniert es auf neuen Daten?"*

16.12 Training, Validierung und Test

Wenn wir ein Modell aus den vorhandenen Daten bauen, ist es nützlich, die Daten in Teile aufzuteilen:

$$\boxed{\text{Training} \quad + \quad \text{Validierung} \quad + \quad \text{Test}}$$

Auf dem Trainingssatz wird das Modell trainiert. Der Validierungssatz wird zur Wahl der Parameter des Modells verwendet. Der Testsatz muss unabhängig bleiben und wird zur abschließenden Bewertung der Fähigkeit des Modells verwendet, auf neuen Daten zu arbeiten. Diese Idee wird besonders wichtig, wenn wir zum maschinellen Lernen kommen.

16.13 Was die Statistik wirklich zu tun versucht

Man kann sagen, dass sich die Statistik nicht so sehr mit dem "Zählen von Mittelwerten" befasst als mit einer allgemeineren Aufgabe: **zuverlässige Information aus unvollkommenen Daten zu extrahieren.** Wir beobachten:

$$\text{Daten} = \text{Struktur} + \text{Rauschen}.$$

Unsere Aufgabe besteht darin, aus den Daten die uns interessierende Struktur zu rekonstruieren und gleichzeitig abzuschätzen, wie sehr wir ihr vertrauen können. Im einfachsten Fall sieht das so aus:

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

Im komplizierteren Fall:

$$X \longrightarrow \text{PCA} \longrightarrow \text{niedrigdimensionale Darstellung}.$$

Und noch komplizierter:

$$X \longrightarrow \text{Modell} \longrightarrow \text{Vorhersage} \longrightarrow \text{Validierung auf neuen Daten}.$$

Genau von hier aus wächst auf natürliche Weise das moderne maschinelle Lernen.

Was wichtig zu merken ist

1. Eine reale Messung enthält einen zufälligen und möglicherweise einen systematischen Fehler.
2. Der Mittelwert beschreibt den zentralen Wert der Daten.
3. Die Standardabweichung beschreibt ihre Streuung.
4. Der zufällige Fehler des Mittelwerts verringert sich wie $1/\sqrt{N}$.

5. Der systematische Fehler wird durch Mittelung nicht beseitigt.
6. Die Kovarianz beschreibt die gemeinsame Änderung von Größen.
7. PCA sucht die informativsten Richtungen in mehrdimensionalen Daten.
8. Die Regression baut aus Messungen ein mathematisches Modell.
9. Ein gutes Modell muss nicht nur auf bekannten Daten, sondern auch auf neuen Daten funktionieren.

16.14 Und wieder lineare Algebra

Und vielleicht die angenehmste Beobachtung für uns ist, dass sich die Statistik als kein so fremdes Fach erwiesen hat.

Wir begannen mit Messungen: $x_1, x_2, \ldots, x_N$,
gingen zu Vektoren über: $\mathbf{x}$,
dann zu Normen: $\|\mathbf{x}\|_2$,
zu Kovarianzmatrizen: Σ,
zu Eigenwerten: $\Sigma v = \lambda v$,
zur SVD: $X = U\Sigma V^T$,
und schließlich zur Optimierung: $\min \|Ax - b\|_2^2$.

Das heißt, die Statistik zerstört unser mathematisches Bild nicht. Sie zeigt, wie dieselbe Mathematik beginnt, mit **realen, unvollkommenen und verrauschten Daten** zu arbeiten. Und das ist genau die Situation, mit der ein moderner Chemiker fast jeden Tag konfrontiert ist.

17 Von SVD zu Compressed Sensing: wenn weniger Daten vorhanden sind als nötig

17.1 Eine praktische Aufgabe: Trennung von Spektren

Betrachten wir eine praktische Aufgabe. In unser Labor wurde eine Masse bunter organischer Substanzen gebracht, und zur Hand war nur ein VIS/IR-Spektrometer für Fluoreszenzspektren. Man sagte uns, dass die Proben mehrere Reaktionsprodukte mit vermutlich unterschiedlichen Spektren enthalten, und bat uns, sowohl die Konzentrationen als auch die Spektren der reinen Substanzen zu bestimmen.

Wir wählten eine nicht sehr starke UV-Quelle, damit die Fluoreszenz sehr hell war und unterschiedliche Spektren für verschiedene Substanzen lieferte.

Das scheint eine Aufgabe für SVD zu sein! Denn wenn wir jedes solche Spektrum in seine eigene Spalte der Matrix **A** setzen, annehmen, dass wir hypothetisch eine Matrix reiner Spektren dieser Substanzen **P** haben, und für jede Probe Konzentrationskoeffizienten der Substanzen selbst **Q**, dann:

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

und wir können **P** als die linken Singulärvektoren der Matrix **A** erhalten und **Q** als die rechten Singulärvektoren, von links mit der Diagonalen multipliziert.

Wir können die Singulärwertzerlegung auch immer schreiben als:

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

Darüber hinaus, wenn wir eine einzige Substanz hätten ($R = 1$), wäre es genau so.

17.2 Aber etwas ging schief...

Statt schöner reiner Spektren mit positiven Werten erhielten wir in **P** irgendwelche seltsamen Diagramme — mit negativen Werten, Oszillationen, "Geister"-Peaks.

Tatsächlich — alles war genau so, wie es sein sollte, die Spektren *können* so nicht erhalten werden. Die Spektren selbst sind positiv definite Diagramme, also ist das Skalarprodukt eines beliebigen Paares, obwohl es sehr nahe bei null liegen kann, nicht notwendigerweise null. In der Singulärwertzerlegung verlangen wir, dass alle Spalten von **U** zueinander **orthogonal** sind. Und reale Spektren von Substanzen sind nicht orthogonal! Sie sind einfach "ähnlich" oder "nicht ähnlich".

Wir brauchen etwas anderes, um diese Spektren so auseinanderzuziehen, dass unsere Mathematik sie getrennt sieht.

17.3 Die Magie der dritten Dimension: das Kruskal-Theorem

Wenn wir das Experiment so aufsetzen, dass wir 3D-Daten statt 2D wie im obigen Beispiel erhalten, dann kann mathematisch bewiesen werden, dass auch eine **nicht orthogonale** Zerlegung existiert.

Wenn wir zum Beispiel Fluoreszenzspektren bei mehreren verschiedenen Anregungswellenlängen aufnehmen können. Dann haben wir bereits dreidimensionale Daten:

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

und nach dem **Kruskal-Theorem** (Kruskal, 1977) haben wir keine Anforderungen an die Orthogonalität jeder dieser Matrizen!

17.3.1 Was das Kruskal-Theorem besagt

Das Kruskal-Theorem ist einer der Eckpfeiler der mehrdimensionalen Datenanalyse. Es gibt Bedingungen für die **Eindeutigkeit** der Tensorzerlegung (der sogenannten CANDECOMP/PARAFAC-Zerlegung).

Für einen Tensor 3. Ordnung $\mathcal{A} = \sum_{r=1}^{R} \mathbf{a}_r \circ \mathbf{b}_r \circ \mathbf{c}_r$ (wobei $\circ$ das äußere Produkt von Vektoren ist), ist die Zerlegung **eindeutig** bis auf Permutation und Skalierung der Komponenten, wenn:

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

wobei k_A der **k-Rang** (Kruskal-Rang) der Matrix **A** ist, also die maximale Zahl k, sodass jede Teilmenge von k Spalten der Matrix **A** linear unabhängig ist.

Warum ist das so wichtig?

In der gewöhnlichen SVD haben wir *unendlich viele* Zerlegungen $\mathbf{A} = \mathbf{U}\square\mathbf{V}^H$ — jede orthogonale Transformation im Inneren ergibt eine neue gültige Zerlegung. Deshalb kann die SVD nicht die "physikalischen" Komponenten isolieren — nur mathematisch bequeme (orthogonale).
Das Kruskal-Theorem sagt: Für Tensoren 3. und höherer Ordnung ist bei Erfüllung der Bedingung an die k-Ränge die Zerlegung *eindeutig*! Das bedeutet, wir können genau die physikalischen Komponenten wiederherstellen, die in den Daten waren — ohne die Forderung nach Orthogonalität.

17.3.2 Lösungsmethoden: PARAFAC und ALS

In der Praxis sucht man die Tensorzerlegung mit iterativen Methoden. Die beliebteste ist **ALS** (Alternating Least Squares):

1. Wir fixieren **B** und **C** und lösen das Problem der kleinsten Quadrate für **A**,
2. Wir fixieren **A** und **C** und lösen für **B**,

3. Wir fixieren **A** und **B** und lösen für **C**,

4. Wir wiederholen bis zur Konvergenz.

Dies ist ein klassisches nichtlineares Minimierungsproblem (wir haben es im Kapitel über nichtlineare Operatoren besprochen), und es kann in lokalen Minima hängen bleiben. Deshalb startet man ALS in der Praxis viele Male mit verschiedenen Anfangsapproximationen.

Anwendbarkeit in der Chemie

PARAFAC/CANDECOMP wird weit verbreitet verwendet in:

- Fluoreszenzspektroskopie (EEM — Excitation-Emission Matrix),
- Chromatographie-Massenspektrometrie,
- NMR-Spektroskopie,
- Analyse metabolischer Daten (Metabolomik).

17.4 Mehrdimensionale NMR-Spektren

Und nun — das Interessanteste für Strukturchemiker. In der NMR-Spektroskopie von Proteinen kann man sehr mehrdimensionale Spektren konstruieren: 2D, 3D, 4D und sogar 5D.

Jede Dimension ist ihre eigene Frequenz (die chemische Verschiebung eines bestimmten Kerns: ^{1}H, ^{13}C, ^{15}N). Kreuzpeaks in solchen Spektren zeigen, welche Kerne im Raum nahe beieinander liegen, was die Rekonstruktion der 3D-Struktur des Proteins erlaubt.

Aber hier ist das Problem: Wenn wir ein 4D-Spektrum mit Auflösung $1024 \times 256 \times 256 \times 64$ Punkten aufnehmen wollen, dann ist die Gesamtzahl der Punkte $\sim 4 \times 10^9$. Und jede Messung braucht Zeit (um die Relaxation abzuwarten), und insgesamt sind das **Wochen** Spektrometerbetrieb!

Es gab Zeiten, in denen man, um alle Kreuzpeaks von Ubiquitin (ein Peptid aus 76 Aminosäuren) zu bestimmen, etwa **drei Wochen** ein Spektrum aufnehmen musste. Das Protein konnte in dieser Zeit degradieren!

17.5 Sparse Sampling: weniger bedeutet mehr

Aber man kann nur zufällig dünnbesetzte eindimensionale Spektren aufnehmen — und zwar nicht 10%, nicht 1%, sondern wirklich sehr, sehr wenige, und dieselbe Zerlegung anwenden, nur für dünnbesetzte Daten (sparse data), und ebenso zuverlässige Ergebnisse erhalten!

Indem der Autor vor 20 Jahren zusammen mit seinen Kollegen solche Methoden anwendete, beschleunigte er die Aufnahme eines solchen mehrdimensionalen Spektrums ohne Informationsverlust von drei Wochen auf **15 Minuten**.

Die Idee ist einfach: Wenn wir wissen, dass das Spektrum in einer gewissen Basis *dünnbesetzt* ist (das heißt, aus einer kleinen Zahl von Peaks besteht), dann müssen wir nicht alle Punkte messen. Es genügt, eine zufällige Teilmenge zu messen und dann die fehlenden Punkte durch Optimierung *wiederherzustellen*.

17.6 Compressed Sensing: eine Revolution in der Messtechnik

Dies bringt uns zu einer der schönsten Ideen in der modernen Mathematik und Signalverarbeitung — **Compressed Sensing** (komprimierte Messung, oder compressive sampling).

17.6.1 Klassische Theorie: Nyquist–Shannon

Die traditionelle Theorie der Diskretisierung (Nyquist–Shannon) besagt: Um ein Signal mit maximaler Frequenz $f_{\max}$ wiederherzustellen, muss man es mit einer Frequenz von mindestens $2f_{\max}$ messen.

Für ein 4D-NMR-Spektrum mit Auflösung $1024 \times 256 \times 256 \times 64$ bedeutet dies, dass wir *verpflichtet* sind, alle 4×10^9 Punkte zu messen. Weniger — unmöglich, sonst gibt es Aliasing (Frequenzüberlappung).

17.6.2 Die Revolution: Compressed Sensing

Aber in den Jahren 2004–2006 zeigten Donoho, Candès, Tao und andere Mathematiker: Wenn ein Signal in einer gewissen Basis **dünnbesetzt** (sparse) ist, dann kann es aus einer **wesentlich geringeren** Anzahl von Messungen wiederhergestellt werden!

Formal: Das Signal $\mathbf{x} \in \mathbb{R}^N$ habe eine dünnbesetzte Darstellung $\mathbf{x} = \square\mathbf{s}$, wobei $\mathbf{s}$ nur $K \ll N$ von Null verschiedene Elemente hat. Dann können wir $\mathbf{y} = \square\mathbf{x}$ messen, wobei $\square \in \mathbb{R}^{M \times N}$ die Messmatrix mit $M \sim K \log(N/K) \ll N$ ist, und $\mathbf{x}$ durch Lösung des Problems rekonstruieren:

$$\min_{\mathbf{s}} \|\mathbf{s}\|_1 \quad \text{unter der Bedingung} \quad \mathbf{y} = \square\square\mathbf{s}$$

Warum die L1-Norm und nicht L0?
Die Idee ist, dass wir die *am dünnsten besetzte* Lösung finden wollen (minimiere $\|\mathbf{s}\|_0$ — die Anzahl der von Null verschiedenen Elemente). Aber die Minimierung der L0-Norm ist ein NP-schweres Problem (Durchmusterung aller Kombinationen).
Die Magie von Compressed Sensing ist, dass unter bestimmten Bedingungen an die Matrix $\square$ (die sogenannte Restricted Isometry Property, RIP) die Minimierung der L1-Norm *genau dasselbe Ergebnis* liefert wie die Minimierung von L0! Und die L1-Minimierung ist ein konvexes Problem, das effizient gelöst wird (es ist lineare Programmierung).

17.6.3 Anwendbarkeitsbedingungen

Für eine erfolgreiche Rekonstruktion sind zwei Bedingungen nötig:

1. **Dünnbesetztheit:** Das Signal muss in einer gewissen Basis dünnbesetzt sein (zum Beispiel ist ein NMR-Spektrum eine Menge von Delta-Funktionen, also im Frequenzbereich sehr dünnbesetzt).

2. **Inkohärenz:** Die Messmatrix $\square$ muss "inkohärent" mit der Basis $\square$ sein. In der Praxis bedeutet dies, dass die Messpunkte **zufällig** (oder pseudozufällig) gewählt werden müssen.

17.7 Beispiele für Compressed Sensing in der Chemie und darüber hinaus

17.7.1 Schnelle MRT (Magnetic Resonance Imaging)

Eine der bekanntesten Anwendungen ist die Beschleunigung der MRT in der Medizin. Die klassische MRT erfordert eine langwierige Abtastung (der k-Raum wird zeilenweise gefüllt). Mit Compressed Sensing kann man nur **zufällige** Zeilen des k-Raums messen und das vollständige Bild durch L1-Minimierung rekonstruieren. Dies verkürzt die Scanzeit um das 5–10-fache — entscheidend für Patienten, die nicht lange still liegen können.

17.7.2 Dünnbesetzte NMR-Spektroskopie

In der mehrdimensionalen NMR von Proteinen erlaubt Compressed Sensing:

- Nur eine zufällige Teilmenge der Punkte im mehrdimensionalen k-Raum zu messen,

- Das vollständige Spektrum durch L1-Minimierung zu rekonstruieren,
- Die Experimentzeit von Wochen auf Stunden oder Minuten zu verkürzen.

Das ist genau das, was der Autor des Skripts vor 20 Jahren tat und was heute zum Standard in modernen NMR-Spektrometern geworden ist (Methoden wie sparse sampling, Poisson-gap sampling usw.).

17.7.3 Single-Pixel-Kamera (Rice-Kamera)

Ein erstaunliches Beispiel: eine Kamera, die **keine Pixelmatrix** hat, sondern nur einen einzigen Detektor. Das Bild wird durch komprimierte Messungen mit einem DMD (Digital Micromirror Device) rekonstruiert. Dies funktioniert, weil reale Bilder in der Wavelet-Basis dünnbesetzt sind.

17.7.4 Massenspektrometrie

In der Massenspektrometrie wird Compressed Sensing zur Beschleunigung von FT-ICR (Fourier Transform Ion Cyclotron Resonance) und Orbitrap verwendet — man kann das FID kürzer messen und dennoch eine hohe Auflösung erhalten.

17.7.5 Datenkompression

JPEG verwendet die diskrete Kosinustransformation (ein naher Verwandter der Fourier-Transformation), und Bilder sind in dieser Basis dünnbesetzt — deshalb komprimiert JPEG so gut. Compressed Sensing ist die nächste Stufe: Wir komprimieren nicht nur bereits gemessene Daten, sondern messen gleich weniger.

17.8 Zusammenhang mit früheren Kapiteln

Betrachten wir, wie Compressed Sensing alles verbindet, was wir durchgegangen sind:

- **Lineare Algebra:** Wir lösen ein überbestimmtes System $\square\square\mathbf{s} = \mathbf{y}$ durch L1-Norm-Minimierung.
- **Kondition:** Die Matrix $\square$ muss gut konditioniert sein und die Restricted Isometry Property (RIP) erfüllen.
- **SVD und Tensoren:** Für mehrdimensionale Daten verwenden wir Tensorzerlegungen (PARAFAC), die Eindeutigkeit ohne Orthogonalität liefern.
- **Nichtlineare Optimierung:** L1-Minimierung ist ein konvexes, aber nicht glattes Problem (im Nullpunkt gibt es keine Ableitung). Wir verwenden Methoden wie ISTA (Iterative Shrinkage-Thresholding Algorithm) oder ADMM.
- **FFT:** Der Operator $\square$ ist oft die Fourier-Transformation, und wir verwenden FFT zur schnellen Multiplikation.

Hauptschlussfolgerung

Compressed Sensing ist nicht nur ein weiterer Algorithmus. Es ist eine *Philosophie des Messens*: Wenn wir wissen, dass das Signal einfach (dünnbesetzt) ist, dann können wir es wesentlich weniger messen, als die klassische Theorie verlangt. Dies hat MRT, NMR-Spektroskopie, Massenspektrometrie und viele andere Bereiche revolutioniert. Und all dies basiert auf schöner Mathematik: konvexe Optimierung, Wahrscheinlichkeitstheorie (Zufallsmatrizen) und lineare Algebra.

18 Informationskompression: von Archiven bis JPEG

18.1 Zwei Welten der Kompression

Informationskompression ist eine der praktischsten Bereiche der angewandten Mathematik. Wir komprimieren und dekomprimieren ständig Daten: Archive, Bilder, Videos, Musik. Aber nicht alle Kompression ist gleich.

Es gibt zwei grundlegend verschiedene Ansätze:

- **Verlustfreie Kompression (lossless):** Wir können die Ausgangsdaten *Bit für Bit* wiederherstellen. Beispiele: ZIP, PNG, FLAC.
- **Verlustbehaftete Kompression (lossy):** Wir opfern einen Teil der Information für eine stärkere Kompression. Beispiele: JPEG, MP3, MP4.

18.2 Shannons Informationsentropie

Bevor wir über Methoden sprechen, verstehen wir, *wie viel* Daten überhaupt komprimiert werden können.

Claude Shannon führte 1948 den Begriff der **Informationsentropie** ein. Wenn wir ein Alphabet aus n Symbolen haben und Symbol i mit Wahrscheinlichkeit p_i vorkommt, dann ist die Entropie (die durchschnittliche Informationsmenge pro Symbol):

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

Dies ist die **theoretische Grenze** der verlustfreien Kompression. Wir können Daten nicht stärker komprimieren als H Bit pro Symbol.

Beispiel

In englischem Text kommt der Buchstabe “e” am häufigsten vor ($p \approx 0.13$), “z” — sehr selten ($p \approx 0.001$). Wenn wir alle Buchstaben gleich kodieren würden (8 Bit pro Symbol), würden wir zu viele Bits für seltene Buchstaben verschwenden. Die Entropie englischen Textes beträgt etwa 4.2 Bit pro Symbol, also können wir ihn theoretisch etwa um den Faktor 2 komprimieren.

18.3 Verlustfreie Kompression: Huffman-Kodierung

Die Idee ist einfach: Wir bemerken, dass einige Symbole und Sequenzen von 2–3 Symbolen häufiger vorkommen, legen eine Tabelle solcher Symbole an, und je häufiger ein Symbol oder eine Sequenz vorkommt, desto weniger Bits verwenden wir zu seiner Kodierung.

18.3.1 Der Huffman-Algorithmus

1. Wir zählen die Häufigkeiten aller Symbole im Text.
2. Wir bauen einen Binärbaum: Wir vereinigen die zwei seltensten Symbole zu einem Knoten mit der Summenhäufigkeit und wiederholen, bis wir einen Baum erhalten.
3. Der Kode eines Symbols ist der Pfad von der Wurzel zum Blatt (0 — links, 1 — rechts).

Häufige Symbole landen näher an der Wurzel — ihre Kodes sind kürzer. Seltene — weiter weg, Kodes länger.

Beispiel

Wir haben die Symbole A (Häufigkeit 0.5), B (0.25), C (0.125), D (0.125). Der Huffman-Baum ergibt die Kodes:

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

Mittlere Länge: $0.5 \times 1 + 0.25 \times 2 + 0.125 \times 3 + 0.125 \times 3 = 1.75$ Bit pro Symbol — nahe an der Entropie!

18.3.2 Wörterbuchmethoden: LZ77, LZ78, LZW

Algorithmen der LZ-Familie (Lempel–Ziv) gehen weiter: Sie suchen nicht nur häufige Symbole, sondern auch **sich wiederholende Sequenzen**.

Idee: Wenn wir die Zeichenkette "abrakadabra" bereits gesehen haben, dann schreiben wir sie beim erneuten Auftreten nicht neu, sondern verweisen auf das vorherige Vorkommen: "(zurück 11, Länge 11)".

Dies ist die Grundlage der Formate ZIP, GZIP, PNG.

18.4 Ein chemisches Beispiel: Suche nach Proteinsequenzen

Und nun — ein lebendiges Beispiel aus der Biologie. Wir haben eine Datenbank von Proteinsequenzen (Millionen von Aminosäuren) und müssen herausfinden, ob sie Homologe (ähnliche Proteine) zu unserem neuen Protein enthält.

Der direkte Vergleich "unser Protein vs. jedes Protein in der Datenbank" ist zu langsam. Wir brauchen einen cleveren Algorithmus für Kompression und Suche.

18.4.1 BLAST (Basic Local Alignment Search Tool)

BLAST ist einer der meistzitierten Algorithmen in der Biologie. Die Idee:

1. Wir zerlegen unser Protein in kurze "Wörter" (k-Mere, üblicherweise $k = 3$ für Proteine).
2. Für jedes Wort suchen wir ähnliche Wörter in der Datenbank (mit kleinen Mutationen).
3. Wir erweitern die Übereinstimmungen in beide Richtungen, bis die Ähnlichkeit unter eine Schwelle fällt.

Dies ist im Wesentlichen eine Wörterbuchkompressionsmethode + schnelle Suche über eine Hash-Tabelle. BLAST erlaubt es, Homologe in Sekunden zu finden, obwohl ein vollständiges Alignment Stunden dauern würde.

Warum ist das wichtig?

BLAST und seine Varianten (PSI-BLAST, BLASTP, BLASTN) sind die Grundlage der modernen Bioinformatik. Ohne sie gäbe es weder Genomentschlüsselung noch Wirkstoffentwicklung noch Evolutionsforschung.

18.5 Verlustbehaftete Kompression: JPEG und Niedrigrang-Approximation

Nun — verlustbehaftete Kompression. Die Idee: Wir opfern einen Teil der Information, den das menschliche Auge (oder Ohr) sowieso nicht bemerkt.

18.5.1 JPEG über die diskrete Kosinustransformation (DCT)

JPEG funktioniert so:

1. Wir zerlegen das Bild in Blöcke von 8×8 Pixeln.
2. Auf jeden Block wenden wir die **diskrete Kosinustransformation** (DCT) an — ein naher Verwandter der Fourier-Transformation.
3. Die DCT zerlegt den Block in 64 Frequenzkomponenten (von niedrigen Frequenzen — der allgemeine Hintergrund, bis zu hohen Frequenzen — feine Details).
4. Wir quantisieren die Koeffizienten: Wir teilen durch eine Quantisierungsmatrix und runden. Hohe Frequenzen (feine Details) werden gröber quantisiert — wir "verlieren" sie.
5. Die verbleibenden von Null verschiedenen Koeffizienten werden verlustfrei komprimiert (Huffman).

18.5.2 Niedrigrang-Approximation über SVD

Eine alternative Sicht auf die Bildkompression ist über SVD. Wir haben die Bildmatrix $\mathbf{A} \in \mathbb{R}^{M \times N}$. SVD:

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

Wir können $\mathbf{A}$ durch die ersten r Singulärwerte approximieren:

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

Dies ist die **Niedrigrang-Approximation** vom Rang r. Nach dem Eckart–Young-Theorem ist dies die *beste* Approximation vom Rang r im Sinne der L2-Norm.

Kompressionsrate

Ausgangsmatrix: $M \times N$ Zahlen. Approximation vom Rang r: $r(M + N + 1)$ Zahlen. Wenn $r \ll \min(M, N)$, ist die Kompression enorm!
Zum Beispiel für ein Bild 1000×1000 und $r = 50$: Kompression um $1000 \times 1000/(50 \times 2001) \approx 10$ Mal.

18.6 Wavelets: lokale Frequenzen

Sowohl DCT als auch SVD sind globale Transformationen: Sie zerlegen das ganze Bild in Frequenzen. Aber Bilder haben *lokale* Besonderheiten: Kanten von Objekten, Texturen, Punkte.

Wavelets sind Basisfunktionen, die sowohl im Raum als auch in der Frequenz lokalisiert sind. Sie erlauben die Analyse eines Signals auf verschiedenen Skalen.

18.6.1 Die Wavelet-Transformation

Statt Sinuskurven (wie bei Fourier) verwenden wir "kleine Wellen" (wavelets) — Funktionen, die schnell abklingen. Die beliebteste ist das **Haar-Wavelet**:

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

Die Wavelet-Transformation zerlegt ein Signal nach Skalen (Frequenzen) und Positionen. Dies erlaubt:

- Bilder besser zu komprimieren als JPEG (das Format JPEG2000 verwendet Wavelets),
- Kanten zu detektieren (Kanten von Objekten),
- Rauschen zu entfernen und wichtige Details zu bewahren.

Fazit

Informationskompression ist ein Gleichgewicht zwischen mathematischer Theorie (Entropie, SVD, Wavelets) und Psychophysik (was das Auge sieht, was das Ohr hört). Von Archiven bis JPEG — überall eine Idee: Struktur in den Daten finden und sie für eine kompakte Darstellung nutzen.

19 Bildvergleich: von der Faltung zu neuronalen Netzen

19.1 Die Aufgabe: ein Objekt in einem Bild finden

Wie vergleicht man zwei Bilder und versteht, dass sie dasselbe Objekt enthalten?

Wir haben bereits zwei Sequenzen verglichen, indem wir eine zirkulante Matrix konstruiert haben (Kapitel über FFT). Wahrscheinlich kann man Bilder genauso vergleichen? Sie seien 2D. Aber nicht alles ist hier so gut, wie es scheint.

Wenn ein Bild relativ zum anderen einfach **verschoben** ist entlang einer oder beider Achsen — ja, alles wird funktionieren. Wir können die 2D-Faltung (über FFT) verwenden und das Maximum finden — das ist die Position des Objekts.

Aber wenn es eine **Drehung** gibt? Oder eine **Streckung**? Oder eine **Änderung der Beleuchtung**? Einfache Faltung hilft nicht mehr.

19.2 Edge Detection: Kantenerkennung

Der erste Schritt zum Verständnis des Bildinhalts ist die Erkennung von **Kanten** (edges). Kanten sind Stellen, an denen sich die Pixelintensität stark ändert.

19.2.1 Der Sobel-Operator

Der einfachste Weg ist die Berechnung des Intensitätsgradienten. Der Sobel-Operator verwendet zwei 3×3-Masken zur Berechnung der Ableitungen nach x und y:

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

wobei $*$ die Faltung und I das Bild ist.

Gradientenbetrag: $G = \sqrt{G_x^2 + G_y^2}$. Wo G groß ist — dort ist eine Kante.

19.2.2 Canny-Kantendetektor

Ein fortgeschrittenerer Algorithmus:

1. Wir glätten das Bild mit einem Gauß-Filter (Rauschen entfernen).
2. Wir berechnen den Gradienten (wie bei Sobel).
3. Wir unterdrücken Nichtmaxima (nur die stärksten Kanten behalten).
4. Wir wenden Hysterese an: Wenn eine Kante über der oberen Schwelle liegt — behalten, unter der unteren — entfernen, dazwischen — behalten, wenn sie mit einer starken Kante verbunden ist.

Das Ergebnis — dünne, zusammenhängende Linien von Kanten.

19.3 Keypoints: Schlüsselpunkte eines Bildes

Kanten sind gut, aber wir brauchen etwas "punktartigeres", um Bilder zu vergleichen. Dafür sucht man **Keypoints** (interest points) — Stellen, die leicht zu finden und stabil gegenüber Transformationen sind.

19.3.1 Harris-Eckendetektor

Idee: Wir suchen Ecken — Stellen, an denen eine Kante die Richtung ändert. Für jedes Pixel berechnen wir die Matrix der zweiten Momente des Gradienten:

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

wobei $w(x,y)$ ein Fenster ist (zum Beispiel Gauß).

Wenn beide Eigenwerte von $\mathbf{M}$ groß sind — das ist eine Ecke. Wenn einer groß und der andere klein ist — das ist eine Kante. Wenn beide klein sind — das ist ein flacher Bereich.

19.3.2 SIFT (Scale-Invariant Feature Transform)

SIFT ist einer der beliebtesten Algorithmen (Lowe, 1999). Er findet Keypoints, die stabil sind gegenüber:

- Skalierung (das Objekt kann näher oder weiter sein),
- Drehung,
- Änderung der Beleuchtung,
- kleinen Änderungen des Blickwinkels.

Algorithmus:

1. Wir bauen eine **Skalenpyramide**: Wir verwischen das Bild mit einem Gauß-Filter mit unterschiedlichem σ und verkleinern es.
2. Wir suchen **Extrema** in der Differenz von Gauß-Funktionen (DoG — Difference of Gaussians) — das ist eine Approximation des Laplace-Operators.
3. Für jeden Keypoint berechnen wir einen **Deskriptor**: ein Histogramm der Gradienten in einer 16×16-Pixel-Nachbarschaft. Das ist ein Vektor aus 128 Zahlen.
4. Wir vergleichen die Deskriptoren zwischen Bildern (euklidischer Abstand).

19.3.3 SURF, ORB, AKAZE

Es gibt viele Verbesserungen von SIFT:

- **SURF** (Speeded-Up Robust Features) — schneller, verwendet Integralbilder.
- **ORB** (Oriented FAST and Rotated BRIEF) — sehr schnell, patentfrei.
- **AKAZE** — verwendet nichtlineare Diffusionsskalen.

19.4 Finden der Transformation: RANSAC

Wir haben Keypoints in zwei Bildern gefunden und sie einander zugeordnet (feature matching). Aber nicht alle Zuordnungen sind korrekt — es gibt Ausreißer (outliers).

Wie findet man die Transformation (Drehung, Skalierung, Verschiebung), die ein Bild in das andere überführt, wenn Ausreißer vorhanden sind?

19.4.1 RANSAC (Random Sample Consensus)

RANSAC ist ein eleganter Algorithmus zur robusten Parameterschätzung:

1. Wir wählen zufällig die minimale Anzahl von Punkten (für eine affine Transformation — 3 Punktpaare).
2. Aus diesen Punkten berechnen wir die Parameter der Transformation.
3. Wir zählen, wie viele *aller* Punkte mit dieser Transformation übereinstimmen (inliers) — Abstand kleiner als eine Schwelle.
4. Wir wiederholen N Mal, wählen die Transformation mit der maximalen Anzahl von Inliers.
5. Schließlich verfeinern wir die Parameter über alle Inliers (Methode der kleinsten Quadrate).

Warum funktioniert RANSAC?

Wenn der Anteil der Ausreißer nicht zu groß ist (sagen wir, $< 50\%$), dann ist die Wahrscheinlichkeit, zufällig nur Inliers zu wählen, von Null verschieden. Wenn wir es oft genug wiederholen, finden wir fast sicher die "richtige" Transformation.

19.5 Partikelschwarm-Methoden und Optimierung

Manchmal haben wir keine expliziten Keypoints, aber wir wissen, dass ein Bild eine transformierte Version des anderen ist. Wie finden wir die Transformationsparameter?

Dies ist ein Optimierungsproblem: Wir maximieren eine **Ähnlichkeitsmetrik** (zum Beispiel die Mutual Information) über die Transformationsparameter.

19.5.1 Particle Swarm Optimization (PSO)

Die Partikelschwarm-Methode ist eine heuristische Optimierungsmethode, inspiriert vom Verhalten eines Vogelschwarms oder Fischschwarms:

1. Wir initialisieren einen "Schwarm" aus N Partikeln — jedes Partikel ist eine Menge von Transformationsparametern (zum Beispiel Drehwinkel, Skalierung, Verschiebungen entlang x und y).
2. Jedes Partikel hat eine "Geschwindigkeit" und eine "Position".

3. Bei jedem Schritt "merkt" sich das Partikel seine beste Position (personal best) und kennt die beste Position des gesamten Schwarms (global best).

4. Die Geschwindigkeit des Partikels wird als gewichtete Summe aktualisiert: Trägheit (weiter in dieselbe Richtung bewegen), kognitive Komponente (Streben zum personal best) und soziale Komponente (Streben zum global best).

5. Wir wiederholen bis zur Konvergenz.

PSO ist besonders nützlich, wenn:

- Die Ähnlichkeitsfunktion nichtglatt ist oder viele lokale Maxima hat,
- Der Parameterraum groß ist (zum Beispiel eine 3D-Transformation mit 12 Parametern),
- Wir den Gradienten nicht berechnen können (zum Beispiel ist die Mutual Information nicht differenzierbar).

19.6 Neuronale Netze: von handgefertigten Merkmalen zu gelernten

Alle oben besprochenen Methoden (Sobel, Harris, SIFT) sind **handgefertigte Merkmale** (hand-crafted features). Wir selbst denken uns aus, was eine "Kante", eine "Ecke", ein "interessanter Punkt" ist und wie man daraus einen Deskriptor baut.

Aber was, wenn wir den Computer *selbst* lernen lassen, Merkmale zu finden?

19.6.1 Faltungsneuronale Netze (CNN)

Convolutional Neural Networks sind ein spezieller Typ neuronaler Netze, der für die Arbeit mit Bildern geschaffen wurde. Die Idee:

1. **Faltungsschichten:** Wir wenden eine Menge trainierbarer Filter an (wie Sobel-Masken, aber die Parameter werden aus den Daten gelernt). Jeder Filter extrahiert sein eigenes Merkmal — von einfachen Kanten in den ersten Schichten bis zu komplexen Texturen und Objektteilen in den tiefen Schichten.

2. **Pooling-Schichten:** Wir verkleinern die Feature-Map (max pooling — das Maximum in einem 2×2-Fenster nehmen). Dies verleiht Invarianz gegenüber kleinen Verschiebungen.

3. **Vollständig verbundene Schichten:** Am Ende — ein gewöhnliches neuronales Netz, das klassifiziert oder regressiert.

19.6.2 Hierarchie der Merkmale

Erstaunlicherweise lernen CNNs selbst dieselbe Hierarchie, die wir von Hand konstruiert haben:

- **Erste Schichten:** Einfache Kanten, Gradienten (wie Sobel),
- **Mittlere Schichten:** Texturen, Ecken, einfache Formen (wie SIFT-Deskriptoren),
- **Tiefe Schichten:** Objektteile (Augen, Räder, Blätter),
- **Letzte Schichten:** Ganze Objekte (Gesicht, Auto, Baum).

Dies ist **Feature Learning** statt handgefertigter Merkmale.

19.7 Vortrainierte Modelle und Transfer Learning

Das Training eines CNN von Grund auf erfordert Millionen annotierter Bilder und Tage der Berechnung auf einer GPU. Aber es gibt einen Trick — **Transfer Learning**.

Idee: Wir nehmen ein Netz, das auf einer riesigen Datenbank vortrainiert wurde (zum Beispiel ImageNet — 1.4 Millionen Bilder, 1000 Klassen), und verwenden es als **Feature-Extraktor**.

19.7.1 Beliebte Architekturen

- **VGG (2014):** Einfach, tief (16–19 Schichten), gute Merkmale.
- **ResNet (2015):** Sehr tief (bis zu 152 Schichten) mit "skip connections" — löst das Problem der verschwindenden Gradienten.
- **EfficientNet (2019):** In allen Parametern optimiert (Genauigkeit, Geschwindigkeit, Größe).
- **Vision Transformer (ViT, 2020):** Verwendet die Transformer-Architektur (wie in NLP) für Bilder.

19.7.2 Wie man ein vortrainiertes Modell verwendet

1. **Feature Extraction:** Wir nehmen ein vortrainiertes Netz, schneiden die letzte Klassifikationsschicht ab und verwenden die vorletzte Schicht als Feature-Extraktor. Wir erhalten einen Vektor von, sagen wir, 2048 Zahlen für jedes Bild. Wir vergleichen die Vektoren (Kosinus-Abstand).
2. **Fine-Tuning:** Wir nehmen ein vortrainiertes Netz und trainieren es auf unseren Daten weiter (zum Beispiel auf mikroskopischen Bildern von Zellen). Die ersten Schichten werden "eingefroren" (sie funktionieren bereits gut), die letzten — trainiert.
3. **Siamese Networks:** Zwei identische Netze mit gemeinsamen Gewichten, trainiert so, dass ähnliche Bilder nahe Feature-Vektoren haben und unterschiedliche — weit entfernte.

19.8 Anwendungen in Chemie und Biologie

19.8.1 Kryo-Elektronenmikroskopie (cryo-EM)

In der cryo-EM erhalten wir Tausende von 2D-Projektionen von Proteinmolekülen in zufälligen Orientierungen. Die Aufgabe:

1. Alle Moleküle auf den Mikrographien finden (particle picking) — das ist eine Objektdetektionsaufgabe,
2. Die Orientierung jeder Projektion bestimmen — das ist eine Bildvergleichsaufgabe,
3. Die 3D-Struktur rekonstruieren — das ist eine inverse Tomographieaufgabe.

Moderne Programme (RELION, cryoSPARC) verwenden CNNs für particle picking und Klassifikation. Dies hat eine "Resolution Revolution" ermöglicht — Proteinstrukturen mit atomarer Auflösung zu bestimmen.

19.8.2 Mikroskopie und Zellanalyse

In der biologischen Forschung muss man:

- Zellen in mikroskopischen Bildern zählen,
- Zelltypen klassifizieren,
- Zellbewegungen über die Zeit verfolgen,

- Anomalien finden (Krebszellen).

CNN (insbesondere U-Net für die Segmentierung) ist der Standard in diesem Bereich.

19.8.3 Chemometrie und Wirkstoffforschung

In der Wirkstoffforschung muss man molekulare Strukturen vergleichen. Aber Moleküle sind keine Bilder! Man kann sie jedoch als 2D-Bilder darstellen (molekulare Graphen, auf ein Gitter gezeichnet) und CNN zur Aktivitätsvorhersage verwenden.

19.9 Zusammenhang mit früheren Kapiteln

Betrachten wir, wie der Bildvergleich alles verbindet, was wir durchgegangen sind:

- **Lineare Algebra:** Faltung ist die Multiplikation einer Matrix mit einem Vektor (oder Tensor). CNN ist eine Folge linearer Operationen mit Nichtlinearitäten.
- **SVD und Niedrigrang-Approximation:** CNN können über SVD komprimiert werden (Niedrigrang-Faktorisierung von Faltungskernen).
- **FFT:** Faltung über FFT ist $\mathcal{O}(N \log N)$ statt $\mathcal{O}(N^2)$. In CNN ist dies entscheidend für die Geschwindigkeit.
- **Nichtlineare Optimierung:** Das Training eines CNN ist die Minimierung einer Verlustfunktion (cross-entropy, MSE) über Backpropagation + SGD/Adam (Varianten des Gradientenabstiegs).
- **Compressed Sensing:** In MRT und cryo-EM verwenden wir komprimierte Messungen + CNN zur Rekonstruktion.

Hauptschlussfolgerung

Bildvergleich ist von einfach zu komplex:

1. Faltung (für Verschiebungen) — über FFT,
2. Keypoints (SIFT) + RANSAC (für Drehungen und Skalierung),
3. Optimierung (PSO) für komplexe Transformationen,
4. Neuronale Netze (CNN) — für semantische Ähnlichkeit.

Jede Methode ist ein Kompromiss zwischen Universalität, Geschwindigkeit und Genauigkeit. Und alle basieren auf der Mathematik, die wir durchgegangen sind: lineare Algebra, Optimierung, Fourier-Analyse, Wahrscheinlichkeitstheorie.

20 Maschinelles Lernen: vom Perzeptron zu AlphaFold

20.1 Was ist maschinelles Lernen?

Maschinelles Lernen (Machine Learning, ML) ist ein Zweig der künstlichen Intelligenz, in dem der Computer *selbst lernt*, Aufgaben auf der Grundlage von Daten zu lösen, statt nach fest vorgeschriebenen Regeln.

Statt ein Programm “wenn Temperatur > 100, dann siedet Wasser” zu schreiben, geben wir dem Computer Tausende von Beispielen “Temperatur — Zustand des Wassers” und bitten ihn, das Muster zu finden.

20.1.1 Drei Arten des Lernens

- **Überwachtes Lernen (supervised):** Es gibt annotierte Daten (Eingabe $\rightarrow$ Ausgabe). Die Aufgabe ist, die Ausgabe für neue Eingaben vorherzusagen. Beispiele: Bildklassifikation, Vorhersage von Moleküleigenschaften.
- **Unüberwachtes Lernen (unsupervised):** Daten ohne Annotation. Die Aufgabe ist, Struktur, Cluster, verborgene Muster zu finden. Beispiele: Clustering von Spektren, PCA.
- **Verstärkungslernen (reinforcement):** Ein Agent interagiert mit der Umgebung und erhält Belohnungen/Strafen. Die Aufgabe ist, eine Strategie zu lernen, die die Belohnung maximiert. Beispiele: Schachspielen, Robotersteuerung.

20.2 Vom Perzeptron zu neuronalen Netzen

20.2.1 Das Perzeptron (1958, Rosenblatt)

Das einfachste "neuronale Netz" ist das Perzeptron. Es berechnet:

$$y = \sigma\left(\sum_{i=1}^{n} w_i x_i + b\right)$$

wobei x_i die Eingaben, w_i die Gewichte, b der Bias und σ die Aktivierungsfunktion ist (zum Beispiel eine Stufe oder eine Sigmoid-Funktion).

Das Perzeptron kann nur **linear trennbare** Probleme lösen (zum Beispiel logisches ODER). Für XOR (exklusives ODER) genügt ein Perzeptron nicht — man braucht ein mehrschichtiges Netz.

20.2.2 Mehrschichtiges Perzeptron (MLP)

Ein Netz aus mehreren Schichten: Eingabeschicht, eine oder mehrere verborgene Schichten, Ausgabeschicht. Jedes Neuron ist mit allen Neuronen der nächsten Schicht verbunden.

Formel für die l-te Schicht:

$$\mathbf{h}^{(l)} = \sigma\left(\mathbf{W}^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}\right)$$

wobei $\mathbf{W}^{(l)}$ die Gewichtsmatrix, $\mathbf{b}^{(l)}$ der Bias-Vektor und σ eine nichtlineare Aktivierungsfunktion ist (ReLU, tanh, sigmoid).

20.2.3 Training: Backpropagation

Wie trainiert man ein neuronales Netz? Man minimiert die Verlustfunktion L (zum Beispiel MSE für Regression oder cross-entropy für Klassifikation) über Gradientenabstieg.

Backpropagation ist ein effizienter Algorithmus zur Berechnung von Gradienten über die Kettenregel. Wir gehen von der Ausgabe zur Eingabe und berechnen $\partial L/\partial w_{ij}$ für jedes Gewicht und aktualisieren die Gewichte:

$$w_{ij} \leftarrow w_{ij} - \eta \frac{\partial L}{\partial w_{ij}}$$

wobei η die Lernrate ist.

20.3 Faltungsneuronale Netze (CNN)

Für Bilder ist MLP ineffizient: zu viele Parameter (jedes Pixel ist mit allen Neuronen verbunden).

CNN verwenden **Faltungsschichten**: Ein kleiner Filter (zum Beispiel 3×3) gleitet über das Bild und berechnet eine Faltung. Dies ergibt:

- **Lokalität der Verbindungen:** Jedes Neuron schaut nur auf einen kleinen Bereich,

- **Gewichtsteilung:** Derselbe Filter wird auf das gesamte Bild angewendet,
- **Verschiebungsinvarianz:** Das Objekt wird überall gefunden.

Beliebte Architekturen: LeNet (1998), AlexNet (2012), VGG (2014), ResNet (2015), EfficientNet (2019).

20.4 Rekurrente neuronale Netze (RNN)

Für Sequenzen (Text, Zeitreihen) braucht man Netze mit **Gedächtnis**. RNN übertragen den verborgenen Zustand von einem Schritt zum nächsten:

$$\mathbf{h}_t = \sigma\left(\mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{W}_x \mathbf{x}_t + \mathbf{b}\right)$$

Das Problem gewöhnlicher RNN sind **verschwindende Gradienten** (vanishing gradients): Das Netz lernt bei langen Sequenzen schlecht.

Lösungen:

- **LSTM** (Long Short-Term Memory, 1997): Fügt "Tore" (gates) hinzu, die den Informationsfluss steuern.
- **GRU** (Gated Recurrent Unit, 2014): Eine vereinfachte Version von LSTM.

20.5 Die Revolution: Transformer und Attention

Im Jahr 2017 erschien die Arbeit "Attention Is All You Need" (Vaswani et al.), die NLP und vieles mehr auf den Kopf stellte.

20.5.1 Der Attention-Mechanismus

Idee: Statt die ganze Sequenz in einen Vektor zu komprimieren (wie bei RNN), erlauben wir jedem Element, auf alle anderen Elemente zu "schauen" und sie nach Wichtigkeit zu gewichten.

Für eine Eingabesequenz $\mathbf{x}_1, \ldots, \mathbf{x}_n$ berechnen wir:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V}$$

wobei $\mathbf{Q}$ (query), $\mathbf{K}$ (key), $\mathbf{V}$ (value) lineare Projektionen der Eingaben sind.

20.5.2 Transformer

Die Transformer-Architektur verzichtet vollständig auf Rekurrenz und Faltungen. Stattdessen wird nur der Attention-Mechanismus (self-attention) und vollständig verbundene Schichten verwendet.

Schlüsselkomponenten:

- **Multi-Head Attention:** Mehrere parallele Attention-Mechanismen, von denen jeder lernt, sich auf verschiedene Aspekte der Daten zu konzentrieren.
- **Positional Encoding:** Da der Transformer keine Rekurrenz hat, kennt er die Reihenfolge der Elemente nicht. Wir fügen Positionskodierungen hinzu (Sinus/Kosinus oder trainierbare Vektoren).
- **Layer Normalization:** Normalisierung der Aktivierungen für Trainingsstabilität.
- **Residual Connections:** "Skip connections" zur Bekämpfung verschwindender Gradienten (wie in ResNet).

Transformer wurde zur Grundlage für:

- **BERT** (2018): Bidirectional Encoder Representations from Transformers — Vortraining auf maskierten Wörtern.
- **GPT** (2018–2023): Generative Pre-trained Transformer — autoregressive Textgenerierung. GPT-3 (175 Milliarden Parameter), GPT-4 (multimodal).
- **T5, BART**: Encoder-Decoder-Architekturen für Übersetzung, Zusammenfassung.

20.6 Maschinelles Lernen in der Chemie: Evolution

20.6.1 Die QSAR-Ära (1990er – 2010er)

QSAR (Quantitative Structure-Activity Relationship) ist ein klassischer Ansatz: Wir berechnen **molekulare Deskriptoren** (numerische Eigenschaften eines Moleküls) und bauen eine Regression/Klassifikation.

Deskriptoren:

- Physikochemische: Molmasse, logP (Lipophilie), polare Oberfläche,
- Topologische: Konnektivitätsindizes, Formen des Molekülgraphen,
- Elektronische: Atomladungen, Orbitalenergien,
- 3D-Deskriptoren: Trägheitsmomente, Gyrationsradien.

Methoden: PLS (Partial Least Squares), Random Forest, SVM (Support Vector Machines).

Beispiel

Vorhersage der Toxizität eines Moleküls: Wir berechnen 200 Deskriptoren, sammeln eine Datenbank von 10 000 Molekülen mit bekannter Toxizität, trainieren einen Random Forest. Genauigkeit — 80–85%.

20.6.2 Graphneuronale Netze (2015 – heute)

Ein Molekül ist ein **Graph**: Atome sind Knoten, Bindungen sind Kanten. Graphneuronale Netze (Graph Neural Networks, GNN) arbeiten direkt mit Graphen.

Message Passing Neural Networks (MPNN):

1. Jedes Atom hat eine Anfangsdarstellung (Merkmalsvektor: Atomtyp, Ladung, Hybridisierung).
2. Bei jedem Schritt "tauschen Atome Nachrichten" mit Nachbarn über Bindungen aus.
3. Nach K Schritten "kennt" jedes Atom seine Nachbarschaft vom Radius K Bindungen.
4. Die finale Darstellung des Moleküls ist eine Aggregation aller atomaren Darstellungen.

Beliebte Architekturen:

- **GCN** (Graph Convolutional Network): Ein Analogon der Faltung für Graphen.
- **GAT** (Graph Attention Network): Attention zwischen Atomen.
- **SchNet, DimeNet, SphereNet**: Berücksichtigen 3D-Koordinaten der Atome und Winkel.

Warum sind GNN so gut für die Chemie?

- **Permutationsinvarianz:** Die Reihenfolge der Atome ist egal,

- **Strukturbewusstsein:** GNN sehen Bindungen und Geometrie,
- **Interpretierbarkeit:** Man kann schauen, welche Atome/Bindungen für die Vorhersage wichtig sind.

20.6.3 Vorhersage von Moleküleigenschaften

Moderne Muss-Modelle für Chemiker:

Aufgabe	Modell	Genauigkeit
Molekülenergie	SchNet, DimeNet++	MAE $\sim$1 kcal/mol
Solvatation	SolTranX	MAE $\sim$0.5 kcal/mol
pKa	ChemProp, GNN	MAE $\sim$0.3
LogP	MolCLR, GNN	MAE $\sim$0.2
Toxizität	GraphCL, GNN	AUC $\sim$0.9
Wirkstoffähnlichkeit	MolBERT	AUC $\sim$0.85

20.7 Die AlphaFold-Revolution: Vorhersage der Proteinstruktur

20.7.1 Das Problem

Die Vorhersage der 3D-Struktur eines Proteins aus seiner Aminosäuresequenz ist eine der großen Herausforderungen der Biologie. Experimentelle Methoden (X-ray, cryo-EM, NMR) sind teuer und langsam.

20.7.2 AlphaFold (2020, DeepMind)

AlphaFold 2 ist ein Durchbruch, der das Problem der Proteinstrukturvorhersage mit atomarer Genauigkeit gelöst hat.

Architektur:

1. **Evoformer:** Ein Transformer, der verarbeitet:
   - Multiple Sequenzalignment (MSA) — evolutionäre Information,
   - Pair representation — paarweise Abstände zwischen Resten.
2. **Structure Module:** Baut iterativ 3D-Koordinaten der Atome auf und minimiert die Verlustfunktion.
3. **Confidence Estimation:** Sagt pLDDT (per-residue confidence) voraus — wie sicher das Modell bei jedem Rest ist.

Ergebnisse auf CASP14 (Critical Assessment of Structure Prediction):

- Mittleres GDT_TS (Global Distance Test) — 92.4 (experimentelle Genauigkeit — $\sim$90),
- Für 2/3 der Proteine — Genauigkeit < 1 Å (atomare Genauigkeit).

20.7.3 AlphaFold 3 (2024)

Erweiterung auf:

- Protein–Ligand-Komplexe,
- DNA/RNA-Strukturen,
- posttranslationale Modifikationen,
- ionische Wechselwirkungen.

20.8 Ein Fundamentmodell für Konformere

20.8.1 Das Problem des Konformationsraums

Ein Molekül ist keine statische Struktur. Es "atmet" ständig, dreht sich um Bindungen, ändert die Konformation. Für das Wirkstoffdesign ist es entscheidend, nicht eine Struktur zu kennen, sondern ein **Ensemble** niedrigenergetischer Konformere.

Klassische Methoden:

- **Molecular Dynamics (MD):** Wir modellieren die Bewegung der Atome in der Zeit, aber das ist langsam (Nanosekunden — Mikrosekunden).
- **Monte Carlo:** Zufällige Wanderung durch den Konformationsraum, aber ineffizient für große Moleküle.
- **Systematic/Rotamer search:** Durchmusterung aller Kombinationen von Torsionswinkeln, aber exponentielles Wachstum.

20.8.2 ML-Ansatz: Konformer-Vorhersage

Moderne Modelle lernen, die Verteilung der Konformere direkt aus Daten vorherzusagen.

GeoLDM (Geometric Latent Diffusion Models, 2023):

1. **Encoder:** Kodiert eine 3D-Konformation in einen latenten Raum.
2. **Diffusion Process:** Fügt Rauschen zu den latenten Vektoren hinzu (wie in DALL-E, Stable Diffusion).
3. **Decoder:** Erzeugt neue Konformationen aus Rauschen über den umgekehrten Diffusionsprozess.
4. **Energy Model:** Filtert die erzeugten Konformationen nach Energie.

ConfGF (Conformation Graph Factor, 2022):

- Verwendet GNN zur Vorhersage von 3D-Koordinaten,
- Erzeugt ein Ensemble von Konformeren durch Sampling,
- Berücksichtigt die Boltzmann-Verteilung (niedrigenergetische Konformere sind wahrscheinlicher).

20.8.3 Fundamentmodell: Uni-Mol (2023)

Uni-Mol ist ein "Foundation Model" für Moleküle, vortrainiert auf Millionen von Konformationen.

Architektur:

1. **3D Transformer:** Arbeitet mit Atomkoordinaten und Merkmalen.
2. **Pre-training tasks:**
   - Vorhersage von Abständen zwischen Atomen,
   - Vorhersage von Winkeln und Diederwinkeln,
   - Maskierte Atomvorhersage (wie BERT),
   - Kontrastives Lernen (ähnliche Konformere — nah).
3. **Fine-tuning:** Weiteres Training auf konkreten Aufgaben (Energie, Eigenschaften, Konformere).

Ergebnisse:

- Energievorhersage: MAE $\sim$0.5 kcal/mol (besser als DFT für einige Klassen),
- Konformer-Generierung: RMSD < 0.5 Å für 90% der Moleküle,
- Eigenschaftsvorhersage: State-of-the-Art auf den meisten Benchmarks.

20.9 Muss-Werkzeuge für den modernen Chemiker

20.9.1 Software

Werkzeug	Zweck
RDKit	Chemoinformatik: Deskriptoren, Fingerprints, Ähnlichkeit
DeepChem	ML für Wirkstoffforschung, GNN
PyTorch Geometric	Graphneuronale Netze
OpenMM	Molekulardynamik auf GPU
ASE (Atomic Simulation Environment)	Quantenchemie + ML
SchNetPack	SchNet und andere GNN für Chemie
AlphaFold (ColabFold)	Proteinstrukturvorhersage
Uni-Mol	Foundation Model für Moleküle

20.9.2 Typischer Workflow

1. **Datensammlung:** PubChem, ChEMBL, PDB (Protein Data Bank).
2. **Vorverarbeitung:** RDKit für Deskriptoren, Generierung von Konformeren.
3. **Modellierung:** GNN zur Eigenschaftsvorhersage, AlphaFold für Struktur.
4. **Validierung:** Cross-Validation, externer Testsatz, experimentelle Verifikation.
5. **Interpretation:** Attention-Gewichte, SHAP-Werte zum Verständnis der Vorhersagen.

20.10 Zusammenhang mit früheren Kapiteln

Betrachten wir, wie ML alles verbindet, was wir durchgegangen sind:

- **Lineare Algebra:** Neuronale Netze sind eine Folge von Matrixmultiplikationen $\mathbf{Wx} + \mathbf{b}$. Backpropagation ist die Kettenregel für Matrizen.
- **SVD und Tensoren:** Modellkompression durch Niedrigrang-Faktorisierung. Tensorzerlegungen für effiziente Berechnungen.
- **FFT:** Faltungsschichten verwenden FFT zur Beschleunigung. Attention über FFT (Linear Attention).
- **Nichtlineare Optimierung:** Das Training neuronaler Netze ist die Minimierung eines Verlusts über SGD, Adam (adaptive Methoden), L-BFGS (für Fine-Tuning).
- **Compressed Sensing:** Sparse Training, Pruning neuronaler Netze.
- **Graphen und Tensoren:** GNN arbeiten mit Molekülgraphen. 3D-Modelle verwenden Tensoren von Koordinaten.
- **Monte Carlo:** Diffusionsmodelle sind stochastische Prozesse. Variational Inference — ein Bayesscher Ansatz.

Hauptschlussfolgerung

Maschinelles Lernen ist kein Ersatz für klassische Chemie, sondern ein **mächtiges Werkzeug**, das:

- Berechnungen um das Tausendfache beschleunigt (Eigenschaftsvorhersage vs. DFT),

- neue Möglichkeiten eröffnet (Proteinstrukturvorhersage, Molekülgenerierung),
- Verständnis der Mathematik erfordert (lineare Algebra, Optimierung, Wahrscheinlichkeitstheorie).

Der moderne Chemiker muss wissen:

- Grundlagen des ML (neuronale Netze, GNN, Transformer),
- Werkzeuge (RDKit, PyTorch, DeepChem),
- Wann ML funktioniert und wann nicht (physikalische Grenzen, Interpretierbarkeit).

21 Moderne Rechentechnik: vom Transistor zum Supercomputer

21.1 Zahlen, die man kennen sollte

Ein moderner Computer — eine Workstation — ist:

- 128–1024 GB Arbeitsspeicher,
- ∼500 Gflop/s Spitzenleistung (Milliarden Gleitkommaoperationen pro Sekunde),
- mehrere Prozessoren (von 4 bis 128 Kernen).

Aber hier ist ein Paradoxon: Die Geschwindigkeit des Speicherzugriffs ist **Hunderte Male** langsamer als die Geschwindigkeit arithmetischer Operationen. Und wenn der Zugriff zufällig ist (nicht sequentielles Lesen-Schreiben), dann noch einmal **hundert Mal** langsamer.

Warum ist das so? Klären wir das.

21.2 Wie ein Prozessor aufgebaut ist

Alle sagen, dass ein Prozessor “viele Transistoren” hat. Aber was tun sie?

Tatsächlich ist ein Prozessor eine Entität, die auf **Instruktionen** zugreift und auf **Daten** zugreift. Beides belegt Platz im Speicher.

21.2.1 Prozessorbefehle

Der Prozessor liest Befehle, die üblicherweise erlauben:

- Auf Daten im Speicher zuzugreifen und sie in **Register** zu legen — einen temporären, sehr, sehr schnellen Speicher,
- Eine Operation auszuführen: Addition, Subtraktion, Multiplikation, Vergleich,
- Einen **Sprung** auszuführen: Standardmäßig führt der Prozessor Befehl für Befehl aus, aber wenn er auf eine Sprunginstruktion trifft, kann er vorwärts oder rückwärts springen.

Sprünge sind:

- **Unbedingt:** Der Sprung erfolgt immer,
- **Bedingt:** Wenn wir zuvor Zahlen verglichen haben und die Antwort “ja” oder “nein” haben, dann erfolgt der Sprung nur bei Erfüllung der Bedingung.

21.2.2 SIMD: ein Befehl — viele Daten

Ein moderner Prozessorbefehl kann mit mehreren Zahlen gleichzeitig arbeiten. Zum Beispiel kann eine AVX-512-Instruktion 16 Paare von Additionen gleichzeitig ausführen!

Dies nennt man **SIMD** (Single Instruction, Multiple Data) — eine Instruktion, viele Daten. Wenn die Daten bereits auf dem Prozessor selbst sind, ist das ziemlich einfach.

Woher kommt SIMD?

Die Leute bemerkten, dass man oft identische Operationen ausführen muss: zum Beispiel Pixel identisch transformieren, um ein Bild zu zeichnen, Array-Elemente identisch verarbeiten. Warum dieselbe Instruktion 16 Mal ausführen, wenn man es einmal tun kann — aber für 16 Zahlen gleichzeitig?

Aber um mit jedem Befehl solche Operationen mit *verschiedenen* Daten auszuführen, bräuchten wir ein sehr leistungsfähiges System zum Pumpen von Daten aus dem Speicher in den Prozessor und zurück.

21.2.3 Das Memory-Wall-Problem

Leider würde ein solches System Hunderte und Tausende Male mehr Transistoren belegen, als für einen arithmetischen Block nötig sind.

Und die Leute bemerkten, dass wir in Programmen oft mit einer kleinen Menge von Zahlen arbeiten und dann zu einer anderen Menge übergehen, und so weiter. Einen kleinen Speicherblock mit schnellem Zugriff zu bauen ist nicht so kostspielig an Transistoren. Nur den Programmierer zu zwingen, Daten in diesen Block aus dem Arbeitsspeicher und zurück zu schleppen, wäre schwierig.

Deshalb hat man das automatisiert — und der **Prozessor-Cache** entstand. Es ist schneller Speicher, es gibt nicht viel davon, aber der Prozessor "merkt" sich selbst, was du gerade verwendest, und hält diese Daten einige Zeit in seinem Cache, in der Erwartung, dass sie später vielleicht wieder gebraucht werden.

21.3 Speicherhierarchie

Ein moderner Computer ist eine mehrschichtige Speicherstruktur:

Ebene	Größe	Geschwindigkeit	Beschreibung
Register	32–128 × 64 Bit	∼0.3 ns	Zellen im Prozessor, mit denen er direkt arbeitet
L1-Cache	∼32–64 KB	∼1 ns	Getrennt für Instruktionen und Daten, eigener pro Kern
L2-Cache	∼256 KB – 1 MB	∼3–5 ns	Eigener pro Kern oder gemeinsam für ein Kernpaar
L3-Cache	∼8–64 MB	∼10–20 ns	Gemeinsam für alle Kerne des Prozessors
RAM (DRAM)	128–1024 GB	∼50–100 ns	Hauptspeicher
Festplatte (SSD)	1–10 TB	∼10–100 μs	Langzeitspeicherung

Vergleichen wir, wie viel eine moderne Workstation in einer Nanosekunde durchschnittlich leisten kann:

- **Arithmetik:** Alle ihre Prozessoren können unter Berücksichtigung langer SIMD-Instruktionen etwa **1000 Gleitkommaoperationen** ausführen.
- **L1-Cache:** Man kann 2–3 Zahlen pro Prozessor lesen und schreiben, insgesamt ∼200 Zahlen.
- **L2/L3-Cache:** Diese Größe fällt um Zehner.
- **RAM:** Nur wenige Lese- und Schreibvorgänge pro Nanosekunde, und nur, wenn dies in großen Blöcken von etwa 512 Bytes etwa sequenziell geschieht.

- **Zufälliger Zugriff:** Wenn wir jedes Mal auf verschiedene Blöcke zugreifen, fällt die Geschwindigkeit um weitere Zehner — mehrere Nanosekunden können pro Zahl nötig sein.

Memory Wall

Der zufällige Speicherzugriff ist **Zehntausende Male** langsamer als die reale Rechenleistung eines modernen Prozessors!
Dies ist der Haupt-Bottleneck ("Flaschenhals") moderner Berechnungen. Genau deshalb:

- Matrixalgorithmen schneller arbeiten, wenn die Daten "dicht" im Speicher liegen,
- dünnbesetzte Matrizen langsam sind (viele zufällige Zugriffe),
- cache-orientierte Algorithmen eine eigene Wissenschaft sind.

21.4 Grafikprozessoren (GPU)

Aber das war den Leuten nicht genug. Als sie 3D-Objekte der Computergrafik auf dem Bildschirm zeichneten, bemerkten sie, dass all dies fast vollständig **parallel** erledigt werden kann.

Grob: Wenn wir 1024 Prozessoren hätten, würden wir den ganzen Bildschirm in ein Gitter von 32×32 Quadraten aufteilen, und jeder Prozessor würde in seinem Quadrat zeichnen. So entstanden Grafikkarten — sie haben eine Menge kleiner spezialisierter Prozessoren, die vollständig parallel arbeiten.

21.4.1 Von der Grafik zu Berechnungen

Ende der 1990er Jahre wurden solche Grafikkarten zur Beschleunigung der Grafikdarstellung verwendet. Um 2005 begann man, solche Karten so herzustellen, dass kleine Code-Stücke auf Tausenden solcher kleiner Prozessoren ausgeführt werden konnten — so entstanden **CUDA** (NVIDIA) und **OpenCL** (ein offener Standard).

Bis 2010 begannen alle numerischen Methoden auf Grafikkarten überzugehen. Allmählich fügte man zur Single- und Double-Gleitkomma-Arithmetik die sogenannten **Halb-** (FP16) und **Viertel-** (FP8, INT8) Gleitkommazahlen hinzu — weil mit ihnen Berechnungen schneller waren und sie weniger Speicher belegten.

In modernen NVIDIA-Grafikkarten (H100, B200) ist es möglich, etwa **eine Million Operationen** mit solchen kleinen Objekten in derselben einen Nanosekunde auszuführen — und dies hat es ermöglicht, solche Grafikkarten für moderne Systeme der künstlichen Intelligenz zu verwenden.

21.4.2 GPU-Architektur

Eine moderne GPU (zum Beispiel NVIDIA H100) ist:

- **Streaming Multiprocessors (SM):** $\sim$132 Multiprocessoren,
- **CUDA-Kerne:** $\sim$16 896 Kerne für FP32-Arithmetik,
- **Tensor-Kerne:** $\sim$528 spezialisierte Kerne für Matrixoperationen (FP16, FP8, INT8),
- **Speicher:** 80 GB HBM3 (High Bandwidth Memory) mit einer Bandbreite von $\sim$3 TB/s.

21.4.3 Threads und Warps

Eine GPU arbeitet mit **Threads**. Tausende von Threads laufen parallel, aber sie sind in Gruppen organisiert:

- **Warp:** Eine Gruppe von 32 Threads, die **synchron** ausgeführt werden — alle 32 Threads führen dieselbe Instruktion zum selben Zeitpunkt aus.
- **Block:** Eine Gruppe von Warps (bis zu 1024 Threads), die Daten über einen gemeinsamen **shared memory** austauschen können.
- **Grid:** Alle auf der GPU gestarteten Blöcke.

Regel der GPU-Effizienz

Damit eine GPU effizient arbeitet, dürfen die Programme auf jedem Multiprocessor für jeden Thread **nicht auseinanderlaufen**. Das bedeutet:

- Alle 32 Threads in einem Warp müssen *dieselbe* Instruktion ausführen (ohne bedingte Sprünge, die die Threads trennen — "warp divergence"),
- Alle Threads müssen auf *aufeinanderfolgende* Speicheradressen zugreifen (coalesced memory access),
- Es müssen genügend Threads vorhanden sein, um die Speicherlatenzen zu "verstecken".

Wenn diese Bedingungen nicht erfüllt sind — arbeitet die GPU um ein Vielfaches langsamer.

21.5 Paralleles Rechnen: Flynns Taxonomie

Im Jahr 1966 schlug Michael Flynn eine Klassifikation paralleler Architekturen nach Instruktions- und Datenströmen vor:

Typ	Beschreibung	Beispiel
SISD	Single Instruction, Single Data. Ein Instruktionsstrom, ein Datenstrom. Klassischer sequentieller Prozessor.	Alte CPUs
SIMD	Single Instruction, Multiple Data. Eine Instruktion auf viele Daten angewendet.	Vektorinstruktionen (AVX), GPU
MISD	Multiple Instruction, Single Data. Verschiedene Instruktionen auf dieselben Daten. Selten.	Einige spezialisierte Architekturen
MIMD	Multiple Instruction, Multiple Data. Viele Prozessoren, jeder führt seine eigenen Instruktionen auf seinen eigenen Daten aus.	Moderne Multiprozessoren, Cluster

21.5.1 Gemeinsamer und verteilter Speicher

In MIMD-Systemen gibt es zwei Ansätze:

- **Gemeinsamer Speicher (shared memory):** Alle Prozessoren haben Zugriff auf einen einzigen Speicher. Beispiel: Multi-Core-CPU. Leicht zu programmieren (alle sehen dieselben Daten), aber schwer zu skalieren (Speicherzugriffskonflikte).
- **Verteilter Speicher (distributed memory):** Jeder Prozessor hat seinen eigenen lokalen Speicher, Datenaustausch — über ein Netzwerk. Beispiel: Cluster, Supercomputer. Schwieriger zu programmieren (man muss Daten explizit übertragen — MPI), aber skaliert bis zu Millionen von Kernen.

21.6 TOP500: die leistungsstärksten Supercomputer der Welt

Die TOP500-Liste ist eine Rangliste der 500 leistungsstärksten Supercomputer der Welt, die zweimal jährlich aktualisiert wird (Juni und November). Die Leistung wird im LINPACK-Benchmark gemessen — der Lösung eines großen linearen Gleichungssystems.

21.6.1 Moderne Spitzenreiter (2024–2025)

Name	Rang	Leistung	Architektur
Frontier	#1	∼1.2 Eflop/s	AMD EPYC + AMD Instinct MI250X
Aurora	#2	∼1 Eflop/s	Intel Xeon + Intel Max PVC
Eagle	#3	∼561 Pflop/s	AMD EPYC + NVIDIA H100
Fugaku	#4	∼442 Pflop/s	Fujitsu A64FX (ARM)
LUMI	#5	∼231 Pflop/s	AMD EPYC + AMD MI250X

Maßstab

1 Eflop/s = 10^{18} Operationen pro Sekunde. Frontier ist ∼1.2 Millionen Milliarden Operationen pro Sekunde. Zum Vergleich: Dein Laptop ist ∼0.1 Tflop/s = 10^{11} Operationen pro Sekunde. Der Unterschied ist 10 Millionen Mal!

21.6.2 Wie ein moderner Supercomputer aufgebaut ist

Typische Architektur:

1. **Knoten (nodes):** ∼10 000–100 000 Server, jeder mit 2–4 CPUs + 4–8 GPUs,
2. **Netzwerk:** Hochgeschwindigkeits-Interconnect (InfiniBand, Slingshot) mit einer Bandbreite von 200–400 Gbit/s pro Knoten,
3. **Speicher:** Paralleles Dateisystem (Lustre, GPFS) mit einer Bandbreite von ∼1–10 TB/s,
4. **Kühlung:** Flüssigkeitskühlung (Wasser oder sogar zweiphasig), da die Leistung 20–50 MW beträgt.

21.7 Digitalisierer: eine Brücke zwischen der analogen und der digitalen Welt

Aber der Prozessor — er muss die Daten für die Berechnung doch irgendwoher nehmen. Und hier betreten **Analog-Digital-Umsetzer** (ADC — Analog-to-Digital Converters) die Bühne.

Sie werden durch zwei Parameter bestimmt:

- **Genauigkeit (Bit-Tiefe):** Wie viele Bits zur Kodierung eines Abtastwerts verwendet werden,
- **Geschwindigkeit (Abtastrate):** Wie viele Abtastwerte pro Sekunde.

21.7.1 Der Kompromiss: Genauigkeit vs. Geschwindigkeit

Es gibt einen fundamentalen Kompromiss: Je höher die Genauigkeit, desto niedriger die Geschwindigkeit.

Bit-Tiefe	Geschwindigkeit	Minimales Bit	Anwendung
12 Bit	∼10 GS/s	∼500 μV	Oszilloskope, schnelle Bildgebung
16 Bit	∼1 GS/s	∼10–50 μV	Audio, NMR, präzise Messungen
24 Bit	∼4 MS/s	∼100–500 nV	Hochauflösendes Audio, Seismographie

Warum werden Nanovolt nicht direkt gemessen?

24 Bit ergeben einen Bereich von Zehnern und Hunderten von Nanovolt. Aber Nanovolt werden von niemandem direkt gemessen — Radarstörungen von allem um uns herum betragen etwa Zehner von Mikrovolt! Das heißt, man misst dort üblicherweise *Ströme*, nicht Spannungen — oder verwendet spezielle abgeschirmte Kammern.

21.8 Physik des Schaltens: vom Transistor zum Megavolt

In einem modernen Prozessor funktioniert alles, indem Transistoren bei etwa 0.7 V ein- und ausgeschaltet werden. Die Geschwindigkeit eines solchen Schaltens beträgt Zehner von Gigahertz. Aber die Ströme dort sind sehr klein (Mikroampere pro Transistor).

21.8.1 Der Kompromiss: Spannung vs. Geschwindigkeit

Wenn man die Spannung erhöht, beginnt die Schaltgeschwindigkeit zu fallen:

Spannung	Schaltzeit	Technologie
0.7 V	∼0.05 ns (20 GHz)	Moderne CMOS-Transistoren
40 V	∼3–5 ns	Leistungs-MOSFETs (gewöhnliche Nennwerte)
1000 V	∼3–5 ns	Leistungs-MOSFETs (Spitzennennwerte)
4.5–5 kV	∼20–30 ns	IGBT, Hochspannungs-MOSFETs
>10 kV	∼100 ns – 1 μs	Ein einzelner Halbleiter schafft es nicht mehr
1 MV	∼100–500 μs	Röhren, Halbleiterbaugruppen, Vervielfacher

Obwohl moderne CMOS-Transistoren bei Frequenzen um 20 GHz schalten, muss man für die Ausführung eines Prozessortakts einen Spielraum von mehreren solchen Schaltvorgängen haben, deshalb liegt die Taktfrequenz von Prozessoren im Bereich von etwa 3-4 GHz.

21.8.2 Die Grenze des Spannungsanstiegs

Über ∼5 kV schafft es ein einzelner Halbleiter nicht mehr — man muss übergehen auf:

- **Gute alte Röhren** (!) — Vakuumtrioden, Thyratrons,
- **Halbleiterbaugruppen** — Reihenschaltung mehrerer Transistoren,
- **Spannungsvervielfacher** — Kaskadenschaltungen mit Röhren oder Funkenstrecken.

Selbst 1 Megavolt zu erzeugen ist nicht sehr schwierig (Marx-Generator, Cockcroft–Walton-Kaskaden). Eine andere Sache ist, dieses Megavolt in einer Nanosekunde ein- und auszuschalten. Dafür braucht man wirklich Hunderte von Mikrosekunden.

Fundamentale Grenze

Tatsächlich ist $\sim 10^{12}$ V/s die moderne Grenze des Spannungsanstiegs praktisch im gesamten Bereich möglicher Spannungen.
Dies ist eine physikalische Beschränkung, verbunden mit:

- der Geschwindigkeit der Bewegung von Ladungsträgern in Halbleitern (Grenze $\sim 10^7$ cm/s),
- parasitären Kapazitäten und Induktivitäten,
- der Ausbreitungsgeschwindigkeit elektromagnetischer Wellen im Medium.

21.9 Zusammenhang mit früheren Kapiteln

Betrachten wir, wie alles, was wir durchgegangen sind, mit der "Hardware" zusammenhängt:

- **Lineare Algebra:** Matrixoperationen sind die Grundlage aller Berechnungen. GPUs sind genau dafür optimiert (Tensor Cores).
- **FFT:** Die schnelle Fourier-Transformation ist $\mathcal{O}(N \log N)$ Operationen, und sie ist entscheidend für NMR, Signalverarbeitung, Datenkompression. GPUs beschleunigen FFT um Zehner.
- **Iterative Methoden:** Das Verfahren der konjugierten Gradienten, GMRES — das ist die Grundlage zur Lösung großer dünnbesetzter Systeme. Sie arbeiten auf Supercomputern mit verteiltem Speicher (MPI + OpenMP + CUDA).
- **Maschinelles Lernen:** Das Training neuronaler Netze sind Milliarden von Matrixmultiplikationen. Ohne GPUs und Tensor Cores wären moderne Modelle (GPT-4, AlphaFold) unmöglich.
- **Compressed Sensing:** Die L1-Minimierung ist eine iterative Methode, die Tausende von Iterationen erfordert. GPUs beschleunigen sie um das 100–1000-fache.
- **Digitalisierer:** ADC ist die Brücke zwischen der analogen Welt (Spektren, Signale) und der digitalen (Prozessor). Die Genauigkeit des ADC bestimmt, welche Mathematik wir anwenden können.

Hauptschlussfolgerung

Moderne Rechentechnik ist:

- **Speicherhierarchie:** Register → L1 → L2 → L3 → RAM → Festplatte. Jede Stufe ist 10–100 Mal langsamer, aber 10–1000 Mal größer.
- **Parallelismus:** SIMD (Vektorinstruktionen), Mehrkernigkeit (CPU), massiver Parallelismus (GPU), Cluster (Supercomputer).
- **Spezialisierung:** Tensor Cores für KI, FPGA für spezifische Aufgaben, ASIC für Mining.
- **Physikalische Grenzen:** Memory Wall, Power Wall, Lichtgeschwindigkeit — all dies begrenzt das Wachstum der Leistung.

Für einen Chemiker bedeutet dies:

- Zu verstehen, wo der "Flaschenhals" ist (Speicher oder Berechnungen),
- Code schreiben zu können, der Cache und GPU effizient nutzt,
- Zu wissen, wann eine Aufgabe auf einer Workstation gelöst wird und wann ein Supercomputer nötig ist.

22 Nachwort: Wie man dieses Wissen nutzt

Früher, nach dem Lesen eines ähnlichen Buches, entstand, wenn man auf eine reale Aufgabe stieß, die Notwendigkeit, eine Lösung zu implementieren. Dafür musste man eine oder mehrere Programmiersprachen recht gut beherrschen und verstehen, wie man fremde Softwaresysteme und -pakete verwendet.

Im modernen Zeitalter der Entwicklung künstlicher Intelligenz kann man all dies mit Unterstützung von KI-Systemen tun — und es wird deutlich effektiver sein. Aber dafür muss man ein wenig die wichtigsten Anwendungen von Programmiersprachen und den sogenannten **Workflow** verstehen — wie Programmentwicklung abläuft.

22.1 Zwei Welten der Programmiersprachen

Obwohl es eine enorme Anzahl von Programmiersprachen gibt, lassen sie sich in zwei grundlegend verschiedene Klassen einteilen:

Kompiliert	**Interpretiert / Bytecode**
C, C++, CUDA, Fortran, Rust	Python, JavaScript, Java, C#, R
Code wird in Maschineninstruktionen für einen bestimmten Prozessor kompiliert	Code wird zur Laufzeit interpretiert oder in Bytecode kompiliert
Maximale Leistung, direkter Zugriff auf die “Hardware”	Flexibilität, Plattformunabhängigkeit, schnelle Entwicklung
Wann verwenden: — Schwere Berechnungen (DFT, MD, ML) — GPU-Beschleunigung (CUDA) — Eingebettete Systeme	**Wann verwenden:** — Prototyping, Datenanalyse — Visualisierung, Berichte — Web-Schnittstellen, Skripte

22.1.1 Warum interpretierte Sprachen bequem sind

Interpretierte Sprachen erlauben es, größere Flexibilität beim Wechsel von einer Plattform zur anderen zu erreichen. Zum Beispiel öffnen wir dieselbe Webseite auf einem Desktop, einem Smartphone und einem Tablet — und sehen denselben Inhalt. Dieser Inhalt wird üblicherweise mit der interpretierten Sprache JavaScript erzeugt, und wenn wir unseren Code für jeden der Prozessoren jedes Mal senden müssten, müsste der Server alle Varianten solchen Codes im Voraus haben. Dies ist manchmal sogar unmöglich vorherzusehen — da sogar verschiedene Android-Smartphones unterschiedliche Prozessorarchitekturen haben können.

22.1.2 Typischer Workflow eines Chemikers

Um einige Ergebnisse schnell anzuzeigen, ist es oft einfacher, interpretierte Programmiersprachen zu verwenden (Python — der absolute Standard in der Wissenschaft). Aber wenn die Aufgabe komplex ist und viele Rechenressourcen erfordert, wird es notwendig, kompilierte Programmiersprachen zu verwenden.

Ein typischer Workflow sieht so aus:

1. **Prototyp in Python:** Schnell die Idee prüfen, die Daten visualisieren, verstehen, ob der Algorithmus funktioniert.
2. **Optimierung:** Wenn der Code funktioniert, aber langsam ist — die “Engpässe” in C++/CUDA umschreiben oder fertige Bibliotheken verwenden (NumPy, SciPy, PyTorch).

3. **Produktion:** Wenn die Aufgabe regelmäßig gelöst wird — in ein Paket packen, Tests, Dokumentation hinzufügen.

22.2 Arbeit mit KI-Assistenten

Heutzutage kann praktisch jeder Algorithmus implementiert werden, indem man einen korrekten Prompt mit Hilfe von Chats schreibt. Aber es ist zu bemerken: Jeder Chat ist fast wie ein Mensch. Und wenn man ihm eine nicht vollständig durchdachte Aufgabe gibt, kann er auch Unsinn machen oder, ohne zu verstehen, etwas nicht ganz Richtiges programmieren.

22.2.1 Goldene Regeln für die Arbeit mit KI

1. **Zerlege die Aufgabe in kleine Blöcke.** Bitte nicht "schreibe ein Programm zur Lösung der Schrödinger-Gleichung". Bitte: "schreibe eine Funktion, die die Fock-Matrix für eine gegebene Basis berechnet".

2. **Bitte um Tests.** Für jeden Block bitte den Chat, Testprüfprogramme zu schreiben, solche Tests auszuführen und sicherzustellen, dass alles in Ordnung ist.

3. **Kombiniere schrittweise.** Erst nachdem jeder Block funktioniert, kombiniere sie, um das Endergebnis zu erhalten.

4. **Verlange Erklärungen.** Wenn der Chat etwas geschrieben hat, das du nicht verstehst — bitte ihn, es zu erklären. Wenn die Erklärung unklar ist — bitte um eine einfachere. Das ist dein Skript, du musst jede Zeile verstehen.

5. **Prüfe an bekannten Beispielen.** Wenn du ein Gleichungssystem löst — prüfe an einem System, für das du die Antwort kennst. Wenn du eine Fourier-Transformation durchführst — prüfe an einer Sinuskurve mit bekannter Frequenz.

Typische Fehler

- **Blindes Vertrauen:** Die KI kann Code schreiben, der "richtig aussieht", aber subtile Fehler enthält (zum Beispiel falsche Normierung, falsche Array-Grenzen, Speicherlecks).
- **Ignorieren des Kontexts:** Die KI kennt deine konkrete Aufgabe nicht — du musst sie ausführlich erklären, mit Beispielen, mit Einschränkungen.
- **Fehlen von Tests:** Code ohne Tests ist Code, der im unpassendsten Moment brechen wird.

22.3 Muss-Werkzeuge für einen Chemiker

Hier ist die minimale Menge von Werkzeugen, die du kennen solltest:

Sprache	Anwendung
Python	Der absolute Standard: Datenanalyse, ML, Visualisierung, Skripte
R	Statistische Analyse, Bioinformatik
MATLAB	Ingenieurberechnungen, Signalverarbeitung (Alternative zu Python)
C++/CUDA	Hochleistungsrechnen, GPU
Bash/Shell	Automatisierung, Arbeit mit Clustern

Bibliothek	Zweck
NumPy, SciPy	Lineare Algebra, Optimierung, Integration
Pandas	Arbeit mit tabellarischen Daten
Matplotlib, Seaborn	Visualisierung
PyTorch, TensorFlow	Maschinelles Lernen, neuronale Netze
RDKit	Chemoinformatik, Moleküle
ASE, PySCF	Quantenchemie
OpenMM, GROMACS	Molekulardynamik

22.4 Letzter Rat

Mathematik ist keine Sammlung von Formeln, die man auswendig lernen muss. Sie ist eine **Denkweise**. Sie ist die Fähigkeit, Struktur im Chaos zu sehen, Muster zu finden, Modelle zu bauen, Hypothesen zu prüfen.

Wenn du auf eine reale Aufgabe stößt — egal, ob es die Vorhersage einer Proteinstruktur, die Analyse eines NMR-Spektrums, die Optimierung einer Synthese oder etwas völlig Neues ist — erinnere dich an dieses Skript. Erinnere dich daran, dass:

- jede Aufgabe entweder ein Gleichungssystem oder ein Optimierungsproblem oder ein Approximationsproblem ist,
- es für jede Aufgabe bewährte mathematische Methoden gibt,
- moderne Werkzeuge (Python, GPU, KI) es erlauben, Aufgaben zu lösen, die vor 20 Jahren unmöglich waren,
- die Hauptsache ist, das Wesen der Aufgabe zu verstehen, und der Rest ist Technik.

Wir glauben an euch. Ihr werdet es schaffen. Und wenn dieses Skript euch eines Tages hilft, eine Aufgabe zu lösen, die die Welt verändert — werden wir unendlich stolz sein.

Letztes Wort

Hab keine Angst, Fehler zu machen. Hab keine Angst zu fragen. Hab keine Angst zu experimentieren. Mathematik ist keine Prüfung, sie ist ein Abenteuer. Und ihr steht am Anfang des interessantesten Weges.
Viel Erfolg!

Mit Liebe,
Papa und Mama

September 2026