Höhere Mathematik

Jil Zerndt, Lucien Perret
January 2025

Rechnerarithmetik

Zahlendarstellung

Maschinenzahlen Eine maschinendarstellbare Zahl zur Basis B ist ein Element der Menge:

$$M = \{x \in \mathbb{R} \mid x = \pm 0.m_1 m_2 m_3 \dots m_n \cdot B^{\pm e_1 e_2 \dots e_l}\} \cup \{0\}$$

- $m_1 \neq 0$ (Normalisierungsbedingung)
- $m_i, e_i \in \{0, 1, \dots, B-1\}$ für $i \neq 0$
- $B \in \mathbb{N}, B > 1$ (Basis)

Zahlenwert Der Wert $\hat{\omega}$ einer Maschinenzahl berechnet sich durch:

$$\hat{\omega} = \sum_{i=1}^{n} m_i B^{\hat{e}-i}, \quad \text{mit} \quad \hat{e} = \sum_{i=1}^{l} e_i B^{l-i}$$

Werteberechnung Berechnung einer vierstelligen Zahl zur Basis 4:

$\underbrace{0.3211}_{n=4} \cdot \underbrace{4^{12}}_{l=2}$

Exponent: $\hat{e} = 1 \cdot 4^1 + 2 \cdot 4^0 = 6$

Wert: $\hat{\omega} = 3 \cdot 4^3 + 2 \cdot 4^2 + 1 \cdot 4^1 + 1 \cdot 4^0 = 57$

IEEE-754 Standard definiert zwei wichtige Gleitpunktformate:

Single Precision (32 Bit)	**Double Precision** (64 Bit)
Vorzeichen(V): 1 Bit	Vorzeichen(V): 1 Bit
Exponent(E): 8 Bit (Bias 127)	Exponent(E): 11 Bit (Bias 1023)
Mantisse(M):	Mantisse(M):
23 Bit + 1 hidden bit	52 Bit + 1 hidden bit

Darstellungsbereich Für jedes Gleitpunktsystem existieren:

- Grösste darstellbare Zahl: $x_{\max} = (1 - B^{-n}) \cdot B^{e_{\max}}$
- Kleinste darstellbare positive Zahl: $x_{\min} = B^{e_{\min}-1}$

Approximations- und Rundungsfehler

Fehlerarten Sei $\tilde{x}$ eine Näherung des exakten Wertes x:

Absoluter Fehler: $|\tilde{x} - x|$

Relativer Fehler: $\left|\frac{\tilde{x} - x}{x}\right|$ bzw. $\frac{|\tilde{x} - x|}{|x|}$ für $x \neq 0$

Maschinengenauigkeit eps ist die kleinste positive Zahl, für die gilt:

Allgemein: $\text{eps} := \frac{B}{2} \cdot B^{-n}$

Dezimal: $\text{eps}_{10} := 5 \cdot 10^{-n}$

Sie begrenzt den maximalen relativen Rundungsfehler: $\left|\frac{rd(x) - x}{x}\right| \leq \text{eps}$

Rundungseigenschaften Für alle $x \in \mathbb{R}$ mit $|x| \geq x_{\min}$ gilt:

Absoluter Fehler: $|rd(x) - x| \leq \frac{B}{2} \cdot B^{e-n-1}$

Relativer Fehler: $\left|\frac{rd(x) - x}{x}\right| \leq \text{eps}$

Fehlerfortpflanzung

Konditionierung Die Konditionszahl K beschreibt die relative Fehlervergrösserung bei Funktionsauswertungen:

$$K := \frac{|f'(x)| \cdot |x|}{|f(x)|}$$

- $K \leq 1$: gut konditioniert
- $K > 1$: schlecht konditioniert
- $K \gg 1$: sehr schlecht konditioniert

Fehlerfortpflanzung Für f (differenzierbar) gilt näherungsweise:

Absoluter Fehler: $|f(\tilde{x}) - f(x)| \approx |f'(x)| \cdot |\tilde{x} - x|$

Relativer Fehler: $\frac{|f(\tilde{x}) - f(x)|}{|f(x)|} \approx K \cdot \frac{|\tilde{x} - x|}{|x|}$

Analyse der Fehlerfortpflanzung einer Funktion

1. Berechnen Sie $f'(x)$
2. Bestimmen Sie die Konditionszahl K
3. Schätzen Sie den absoluten Fehler ab
4. Schätzen Sie den relativen Fehler ab
5. Beurteilen Sie die Konditionierung anhand von K

$$\underbrace{|f(\tilde{x}) - f(x)|}_{\text{absoluter Fehler von } f(x)} \approx \left|f'(x)\right| \cdot \underbrace{|\tilde{x} - x|}_{\text{absoluter Fehler von } x}$$

$$\underbrace{\frac{|f(\tilde{x}) - f(x)|}{|f(x)|}}_{\text{relativer Fehler von } f(x)} \approx \underbrace{\frac{\left|f'(x)\right| \cdot |x|}{|f(x)|}}_{\text{Konditionszahl } K} \cdot \underbrace{\frac{|\tilde{x} - x|}{|x|}}_{\text{relativer Fehler von } x}$$

Fehleranalyse Beispiel: Fehleranalyse von $f(x) = \sin(x)$

1. $f'(x) = \cos(x)$
2. $K = \frac{|x \cos(x)|}{|\sin(x)|}$
3. Für $x \to 0$: $K \to 1$ (gut konditioniert)
4. Für $x \to \pi$: $K \to \infty$ (schlecht konditioniert)
5. Der absolute Fehler wird nicht vergrössert, da $|\cos(x)| \leq 1$

Praktische Fehlerquellen der Numerik

Kritische Operationen häufigste Fehlerquellen:

- Auslöschung bei Subtraktion ähnlich großer Zahlen
- Überlauf (overflow) bei zu großen Zahlen
- Unterlauf (underflow) bei zu kleinen Zahlen
- Verlust signifikanter Stellen durch Rundung

Vermeidung von Auslöschung

1. Identifizieren Sie Subtraktionen ähnlich großer Zahlen
2. Suchen Sie nach algebraischen Umformungen
3. Prüfen Sie alternative Berechnungswege
4. Verwenden Sie Taylorentwicklungen für kleine Werte

Auslöschung bei der Berechnung von $\sqrt{x^2+1} - 1$:

Für kleine x führt die direkte Berechnung zu Auslöschung:

Für $x = 10^{-8}$: $\sqrt{10^{-16}+1} - 1 \approx 1.000000000 - 1 = 0$

Korrekte Lösung durch Umformung: $\sqrt{x^2+1} - 1 = \frac{x^2}{\sqrt{x^2+1}+1}$

Auslöschung Bei der Subtraktion fast gleich großer Zahlen können signifikante Stellen verloren gehen. Beispiel:

- $1.234567 - 1.234566 = 0.000001$
- Aus 7 signifikanten Stellen wird 1 signifikante Stelle

Analyse von Algorithmen

Fehlerakkumulation Bei n aufeinanderfolgenden Operationen mit relativen Fehlern $\leq \varepsilon$ gilt für den Gesamtfehler:

- Best case: $\mathcal{O}(n\varepsilon)$ bei gleichverteilten Fehlern
- Worst case: $\mathcal{O}(2^n\varepsilon)$ bei systematischen Fehlern

Numerische Stabilität eines Algorithmus

- Kleine Eingabefehler führen zu kleinen Ausgabefehlern
- Rundungsfehler akkumulieren sich nicht übermäßig
- Konditionszahl des Problems wird nicht künstlich verschlechtert

Instabilität bei rekursiver Berechnung: (Fibonacci-Zahlen)

```python
def fib(n):
    if n <= 1:
        return n
    return fib(n-1) + fib(n-2)
```

Exponentielles Wachstum der Operationen → Fehlerfortpflanzung

Stabilitätsanalyse Schritte zur Analyse der numerischen Stabilität:

1. Bestimmen Sie kritische Operationen
2. Schätzen Sie Rundungsfehler pro Operation ab
3. Analysieren Sie die Fehlerfortpflanzung
4. Berechnen Sie die worst-case Fehlerschranke
5. Vergleichen Sie alternative Implementierungen

Praktische Implementierungen

Implementierungsgenauigkeit eines Algorithmus

- Relative Genauigkeit der Ausgabe
- Maximale Anzahl korrekter Dezimalstellen
- Stabilität gegenüber Eingabefehlern

Robuste Implementierung von Algorithmen

1. Verwenden Sie stabile Grundoperationen
2. Vermeiden Sie Differenzen ähnlich großer Zahlen
3. Prüfen Sie auf Über- und Unterlauf
4. Implementieren Sie Fehlerkontrollen
5. Dokumentieren Sie numerische Einschränkungen

Robuste Implementation Beispiel: Quadratische Gleichung

```python
def quadratic_stable(a, b, c):
    # ax^2 + bx + c = 0
    if a == 0:
        return [-c/b] if b != 0 else []

    # Calculate discriminant
    disc = b*b - 4*a*c
    if disc < 0:
        return []

    # Choose numerically stable formula
    if b >= 0:
        q = -0.5*(b + sqrt(disc))
    else:
        q = -0.5*(b - sqrt(disc))
    x1 = q/a
    x2 = c/(q)

    return sorted([x1, x2])
```

Numerische Lösung von Nullstellenproblemen

NSP: Nullstellenproblem, NS: Nullstelle

Fixpunktgleichung ist eine Gleichung der Form: $F(x) = x$
Die Lösungen $\bar{x}$, für die $F(\bar{x}) = \bar{x}$ erfüllt ist, heissen Fixpunkte.

Fixpunktiteration

Grundprinzip der Fixpunktiteration sei $F : [a, b] \to \mathbb{R}$ mit $x_0 \in [a, b]$

$$\text{Die rekursive Folge } x_{n+1} \equiv F(x_n), \quad n = 0, 1, 2, \ldots$$

heisst Fixpunktiteration von F zum Startwert x_0.

Konvergenzverhalten
Sei $F : [a, b] \to \mathbb{R}$ mit stetiger Ableitung F' und $\bar{x} \in [a, b]$ ein Fixpunkt von F. Dann gilt für die Fixpunktiteration $x_{n+1} = F(x_n)$:

Anziehender Fixpunkt: $|F'(\bar{x})| < 1$
x_n konvergiert gegen $\bar{x}$, falls x_0 nahe genug bei $\bar{x}$

Abstossender Fixpunkt: $|F'(\bar{x})| > 1$
x_n konvergiert für keinen Startwert $x_0 \neq \bar{x}$

Banachscher Fixpunktsatz $F : [a, b] \to [a, b]$ und $\exists$ Konstante α:
- $0 < \alpha < 1$ (Lipschitz-Konstante)
- $|F(x) - F(y)| \leq \alpha|x - y|$ für alle $x, y \in [a, b]$

Dann gilt:
- F hat genau einen Fixpunkt $\bar{x}$ in $[a, b]$
- Die Fixpunktiteration konvergiert gegen $\bar{x}$ für alle $x_0 \in [a, b]$

Fehlerabschätzungen:

a-priori: $|x_n - \bar{x}| \leq \dfrac{\alpha^n}{1-\alpha} \cdot |x_1 - x_0|$

a-posteriori: $|x_n - \bar{x}| \leq \dfrac{\alpha}{1-\alpha} \cdot |x_n - x_{n-1}|$

Konvergenznachweis für Fixpunktiteration
So überprüfen Sie, ob eine Fixpunktiteration konvergiert:
1. Prüfen Sie, ob $F : [a, b] \to [a, b]$ gilt: $F(a) > a$ und $F(b) < b$
2. Bestimmen Sie $\alpha = \max_{x \in [a,b]} |F'(x)|$
3. Prüfen Sie, ob $\alpha < 1$
4. Berechnen Sie die nötigen Iterationen für Toleranz tol:

$$n \geq \frac{\ln(\frac{tol \cdot (1-\alpha)}{|x_1 - x_0|})}{\ln \alpha}$$

Fixpunktiteration Nullstellen von $p(x) = x^3 - x + 0.3$
Fixpunktgleichung: $x_{n+1} = F(x_n) = x_n^3 + 0.3$
1. $F'(x) = 3x^2$ steigt monoton
2. Für $I = [0, 0.5]$: $F(0) = 0.3 > 0$, $F(0.5) = 0.425 < 0.5$
3. $\alpha = \max_{x \in [0,0.5]} |3x^2| = 0.75 < 1$
4. Konvergenz für Startwerte in $[0, 0.5]$ gesichert

Fixpunktiteration

```python
def fixed_point_iteration(f, x0, tol=1e-6,
    max_iter=100):
    for n in range(max_iter):
        x1 = f(x0)
        if abs(x1 - x0) < tol:
            return x1
        x0 = x1
    raise ValueError("No convergence")
```

Newton-Verfahren

Grundprinzip Newton-Verfahren
Approximation der NS durch sukzessive Tangentenberechnung:

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

Konvergiert, wenn für alle x im relevanten Intervall gilt:

$$\left| \frac{f(x) \cdot f''(x)}{[f'(x)]^2} \right| < 1$$

Newton-Verfahren anwenden
1. Funktion $f(x)$ und Ableitung $f'(x)$ aufstellen
2. Geeigneten Startwert x_0 nahe der Nullstelle wählen
3. Iterieren bis zur gewünschten Genauigkeit: $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$
4. Konvergenz prüfen durch Vergleich aufeinanderfolgender Werte

Newton-Verfahren Nullstellen von $f(x) = x^2 - 2$
Ableitung: $f'(x) = 2x$, Startwert $x_0 = 1$
1. $x_1 = 1 - \frac{1^2-2}{2 \cdot 1} = 1.5$
2. $x_2 = 1.5 - \frac{1.5^2-2}{2 \cdot 1.5} = 1.4167$
3. $x_3 = 1.4167 - \frac{1.4167^2-2}{2 \cdot 1.4167} = 1.4142$

→ Konvergenz gegen $\sqrt{2}$ nach wenigen Schritten

Newton-Verfahren

```python
def newton(f, df, x0, tol=1e-6, max_iter=100):
    for n in range(max_iter):
        x1 = x0 - f(x0) / df(x0)
        if abs(x1 - x0) < tol:
            return x1
        x0 = x1
    raise ValueError("No convergence")
```

Vereinfachtes Newton-Verfahren
Alternative Variante mit konstanter Ableitung:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_0)}$$

Konvergiert langsamer, aber benötigt weniger Rechenaufwand.

Sekantenverfahren
Alternative zum Newton-Verfahren ohne Ableitungsberechnung. Verwendet zwei Punkte $(x_{n-1}, f(x_{n-1}))$ und $(x_n, f(x_n))$:

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

Benötigt zwei Startwerte x_0 und x_1.

Sekantenverfahren Nullstellen von $f(x) = x^2 - 2$
Startwerte $x_0 = 1$ und $x_1 = 2$
1. $x_2 = 1 - \frac{1-2}{1^2-2} \cdot 1 = 1.5$
2. $x_3 = 1.5 - \frac{1.5-1}{1.5^2-2} \cdot 1.5 = 1.4545$
3. $x_4 = 1.4545 - \frac{1.4545-1.5}{1.4545^2-2} \cdot 1.4545 = 1.4143$

→ Konvergenz gegen $\sqrt{2}$ nach wenigen Schritten

Sekantenverfahren

```python
def secant(f, x0, x1, tol=1e-6, max_iter=100):
    for n in range(max_iter):
        x2 = x1 - (x1 - x0) / (f(x1) - f(x0)) * f(x1)
        if abs(x2 - x1) < tol:
            return x2
        x0, x1 = x1, x2
    raise ValueError("No convergence")
```

Konvergenzverhalten

Konvergenzordnung Sei (x_n) eine gegen $\bar{x}$ konvergierende Folge. Die Konvergenzordnung $q \geq 1$ ist definiert durch:

$$|x_{n+1} - \bar{x}| \leq c \cdot |x_n - \bar{x}|^q$$

wo $c > 0$ eine Konstante. Für $q = 1$ muss zusätzl. $c < 1$ gelten.

Konvergenzordnungen der Verfahren Konvergenzgeschwindigkeiten

Newton-Verfahren: Quadratische Konvergenz: $q = 2$

Vereinfachtes Newton: Lineare Konvergenz: $q = 1$

Sekantenverfahren: Superlineare Konvergenz: $q = \frac{1+\sqrt{5}}{2} \approx 1.618$

Konvergenzgeschwindigkeit Vergleich der Verfahren:
Startwert $x_0 = 1$, Funktion $f(x) = x^2 - 2$, Ziel: $\sqrt{2}$

n	Newton	Vereinfacht	Sekanten
1	1.5000000	1.5000000	1.5000000
2	1.4166667	1.4500000	1.4545455
3	1.4142157	1.4250000	1.4142857
4	1.4142136	1.4125000	1.4142136

Fehlerabschätzung

Nullstellensatz von Bolzano Sei $f : [a, b] \to \mathbb{R}$ stetig. Falls

$$f(a) \cdot f(b) < 0$$

dann existiert mindestens eine Nullstelle $\xi \in (a, b)$.

Fehlerabschätzung für Nullstellen
So schätzen Sie den Fehler einer Näherungslösung ab:
1. Sei x_n der aktuelle Näherungswert
2. Wähle Toleranz $\epsilon > 0$
3. Prüfe Vorzeichenwechsel: $f(x_n - \epsilon) \cdot f(x_n + \epsilon) < 0$
4. Falls ja: Nullstelle liegt in $(x_n - \epsilon, x_n + \epsilon)$
5. Damit gilt: $|x_n - \xi| < \epsilon$

Praktische Fehlerabschätzung Fehlerbestimmung bei $f(x) = x^2 - 2$
1. Näherungswert: $x_3 = 1.4142157$
2. Mit $\epsilon = 10^{-5}$:
3. $f(x_3 - \epsilon) = 1.4142057^2 - 2 < 0$
4. $f(x_3 + \epsilon) = 1.4142257^2 - 2 > 0$

Also: $|x_3 - \sqrt{2}| < 10^{-5}$

→ Nullstelle liegt in $(1.4142057, 1.4142257)$

Abbruchkriterien Praktische Implementierung
In der Praxis verwendet man meist mehrere Abbruchkriterien:
- Absolute Änderung: $|x_n - x_{n-1}| < \epsilon_1$
- Funktionswert: $|f(x_n)| < \epsilon_2$
- Maximale Iterationszahl: $n < n_{max}$
- Kombination dieser Kriterien

Fehlerabschätzung

```python
def error_estimate(f, x, eps=1e-5):
    if f(x - eps) * f(x + eps) < 0:
        return eps
    return None
```

LGS und Matrizen

Matrizen

Matrix, Element, Zeilen, Spalten und Typ

Eine *Matrix* ist (simpel gesagt) ein Vektor mit mehreren Spalten und wird mit Grossbuchstaben bezeichnet. Ein *Element* a_ij ist ein Wert aus dieser Matrix, auf den über die Zeile und Spalte zugegriffen wird (**Z**eile **z**uerst, **Sp**alte **Sp**äter). Der einer Matrix ergibt sich aus der Anzahl ihren Zeilen und Spalten. Matrizen mit m-Zeilen und n-Spalten werden $m \times n$-Matrizen genannt.

Matrix Tabelle mit m Zeilen und n Spalten: $m \times n$-Matrix A
a_{ij}: Element in der i-ten Zeile und j-ten Spalte

Nullmatrix Eine Matrix, deren Elemente alle gleich 0 sind, heisst *Nullmatrix* und wird mit 0 bezeichnet.

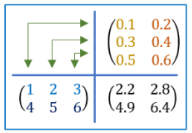
Spaltenmatrix Besteht eine Matrix nur aus einer Spalte, so heisst diese *Spaltenmatrix*. Können als Vektoren aufgefasst werden und können mit einem kleinen Buchstaben sowie einem Pfeil darüber notiert werden ($\vec{a}$).

Addition und Subtraktion
- $A + B = C$
- $c_{ij} = a_{ij} + b_{ij}$

Skalarmultiplikation
- $k \cdot A = B$
- $b_{ij} = k \cdot a_{ij}$

Rechenregeln für die Addition und skalare Multiplikation von Matrizen
- Kommutativ-Gesetz: $A + B = B + A$
- Assoziativ-Gesetz: $A + (B + C) = (A + B) + C$
- Distributiv-Gesetz:
  $\lambda \cdot (A + B) = \lambda \cdot A + \lambda \cdot B$ sowie $(\lambda + \mu) \cdot A = \lambda \cdot A + \mu \cdot A$

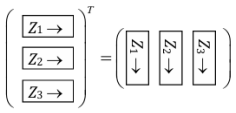
Matrixmultiplikation $A^{m \times n}$, $B^{n \times k}$
Bedingung: A n Spalten, B n Zeilen.
Resultat: C hat m Zeilen und k Spalten.
- $A \cdot B = C$
- $c_{ij} = a_{i1} \cdot b_{1j} + a_{i2} \cdot b_{2j} + \ldots + a_{in} \cdot b_{nj}$
- $A \cdot B \neq B \cdot A$



Rechenregeln für die Multiplikation von Matrizen
- Assoziativ-Gesetz: $A \cdot (B \cdot C) = (A \cdot B) \cdot C$
- Distributiv-Gesetz:
  $A \cdot (B + C) = A \cdot B + A \cdot C$ und $(A + B) \cdot C = A \cdot C + B \cdot C$
- Skalar-Koeffizient: $(\lambda \cdot A) \cdot B = \lambda \cdot (A \cdot B) = A \cdot (\lambda \cdot B)$

Transponierte Matrix $A^{m \times n} \rightarrow (A^T)^{n \times m}$
- A^T: Spalten und Zeilen vertauscht
- $(A^T)_{ij} = A_{ji}$

$$(A \cdot B)^T = B^T \cdot A^T$$



Spezielle Matrizen
- **Symmetrische Matrix**: $A^T = A$
- **Einheitsmatrix/Identitätsmatrix**: E bzw. I
  mit $e_{ij} = 1$ für $i = j$ und $e_{ij} = 0$ für $i \neq j$
- **Diagonalmatrix**: $a_{ij} = 0$ für $i \neq j$
- **Dreiecksmatrix**: $a_{ij} = 0$ für $i > j$ (obere Dreiecksmatrix)
  oder $i < j$ (untere Dreiecksmatrix)

Lineare Gleichungssysteme (LGS)

Lineares Gleichungssystem (LGS) Ein *lineares Gleichungssystem* ist eine Sammlung von Gleichungen, die linear in den Unbekannten sind. Ein LGS kann in Matrixform $A \cdot \vec{x} = \vec{b}$ dargestellt werden.
A: Koeffizientenmatrix
$\vec{x}$: Vektor der Unbekannten
$\vec{b}$: Vektor der Konstanten

$$\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \cdots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix}$$

Rang einer Matrix $rg(A)$ = Anzahl Zeilen - Anzahl Nullzeilen
$\Rightarrow$ Anzahl linear unabhängiger Zeilen- oder Spaltenvektoren

Zeilenstufenform (Gauss)
- Alle Nullen stehen unterhalb der Diagonalen, Nullzeilen zuunterst
- Die erste Zahl $\neq 0$ in jeder Zeile ist eine führende Eins
- Führende Einsen, die weiter unten stehen $\rightarrow$ stehen weiter rechts

Reduzierte Zeilenstufenform: (Gauss-Jordan)
Alle Zahlen links und rechts der führenden Einsen sind Nullen.

Gauss-Jordan-Verfahren
1. bestimme linkeste Spalte mit Elementen $\neq 0$ (Pivot-Spalte)
2. oberste Zahl in Pivot-Spalte $= 0$
   $\rightarrow$ vertausche Zeilen so dass $a_{11} \neq 0$
3. teile erste Zeile durch $a_{11} \rightarrow$ so erhalten wir führende Eins
4. Nullen unterhalb führender Eins erzeugen (Zeilenperationen)

nächste Schritte: ohne bereits bearbeitete Zeilen Schritte 1-4 wiederholen, bis Matrix Zeilenstufenform hat

Zeilenperationen erlaubt bei LGS (z.B. Gauss-Verfahren)
- Vertauschen von Zeilen
- Multiplikation einer Zeile mit einem Skalar
- Addition eines Vielfachen einer Zeile zu einer anderen

Lösbarkeit von linearen Gleichungssystemen
- Lösbar: $rg(A) = rg(A|b)$
- genau eine Lösung: $rg(A) = n$
- unendlich viele Lösungen: $rg(A) < n$

Parameterdarstellung bei unendlich vielen Lösungen
Führende Unbekannte: Spalte mit führender Eins
Freie Unbekannte: Spalten ohne führende Eins

$$\begin{array}{cccc|c} x_1 & x_2 & x_3 & x_4 & \\ \hline 1 & -2 & 0 & 3 & 5 \\ 0 & 0 & 1 & 1 & 3 \end{array}$$

Auflösung nach der führenden Unbekannten:
- $1x_1 - 2x_2 + 0x_3 + 3x_4 = 5 \quad x_2 = \lambda \rightarrow x_1 = 5 + 2 \cdot \lambda - 3 \cdot \mu$
- $0x_1 + 0x_2 + 1x_3 + 1x_4 = 3 \quad x_4 = \mu \rightarrow x_3 = 3 - \mu$

$$\vec{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 5+2\lambda-3\mu \\ \lambda \\ 3-\mu \\ \mu \end{pmatrix} = \begin{pmatrix} 5 \\ 0 \\ 3 \\ 0 \end{pmatrix} + \lambda \begin{pmatrix} 2 \\ 1 \\ 0 \\ 0 \end{pmatrix} + \mu \begin{pmatrix} -3 \\ 0 \\ -1 \\ 1 \end{pmatrix}$$

Homogenes LGS $\vec{b} = \vec{0} \rightarrow A \cdot \vec{x} = \vec{0} \rightarrow rg(A) = rg(A \mid \vec{b})$
nur zwei Möglichkeiten:
- eine Lösung $x_1 = x_2 = \cdots = x_n = 0$, die sog. *triviale Lösung.*
- unendlich viele Lösungen

Koeffizientenmatrix, Determinante, Lösbarkeit des LGS
Für $n \times n$-Matrix A sind folgende Aussagen äquivalent:
- $\det(A) \neq 0$
- $rg(A) = n$
- A ist invertierbar
- Spalten von A sind linear unabhängig.
- Zeilen von A sind linear unabhängig.
- LGS $A \cdot \vec{x} = \vec{0}$ hat eindeutige Lösung $x = A^{-1} \cdot 0 = 0$

Quadratische Matrizen

Umformen bestimme die Matrix X: $A \cdot X + B = 2 \cdot X$
$\Rightarrow A \cdot X = 2 \cdot X - B \Rightarrow A \cdot X - 2 \cdot X = -B \Rightarrow (A - 2 \cdot E) \cdot X = -B$
$\Rightarrow (A - 2 \cdot E) \cdot (A - 2 \cdot E)^{-1} \cdot X = (A - 2 \cdot E)^{-1} \cdot -B$
$\Rightarrow X = (A - 2 \cdot E)^{-1} \cdot -B$

Inverse

Inverse einer quadratischen Matrix A A^{-1}
A^{-1} existiert, wenn $rg(A) = n$. A^{-1} ist eindeutig bestimmt.

Eine Matrix heisst *invertierbar / regulär*, wenn sie eine Inverse hat. Andernfalls heisst sie *singulär*

Eigenschaften invertierbarer Matrizen
- $A \cdot A^{-1} = A^{-1} \cdot A = E$
- $(A^{-1})^{-1} = A$
- $(A \cdot B)^{-1} = B^{-1} \cdot A^{-1}$ Die Reihenfolge ist relevant!
  A und B invertierbar $\Rightarrow$ AB invertierbar
- $(A^T)^{-1} = (A^{-1})^T$ A invertierbar $\Rightarrow$ A^T invertierbar

Inverse einer 2×2-Matrix $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ mit $det(A) = ad - bc$

$$A^{-1} = \frac{1}{\det(A)} \cdot \begin{pmatrix} d & -b \\ -c & a \end{pmatrix}$$

NUR Invertierbar falls $ad - bc \neq 0$

Inverse berechnen einer quadratischen Matrix $A^{n \times n}$

$$A \cdot A^{-1} = E \rightarrow \left(A|E\right) \rightsquigarrow \text{Zeilenoperationen} \rightsquigarrow \left(E|A^{-1}\right)$$

$$\underbrace{\begin{pmatrix} 4 & -1 & 0 \\ 0 & 2 & 1 \\ 3 & -5 & -2 \end{pmatrix}}_{A} \cdot \underbrace{\begin{pmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{pmatrix}}_{A^{-1}} = \underbrace{\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}}_{E}$$

$$\rightarrow \left(\begin{array}{ccc|ccc} 4 & -1 & 0 & 1 & 0 & 0 \\ 0 & 2 & 1 & 0 & 1 & 0 \\ 3 & -5 & -2 & 0 & 0 & 1 \end{array}\right)$$

Zeilenstufenform (linke Seite)

$$\rightsquigarrow \left(\begin{array}{ccc|ccc} 1 & -1/4 & 0 & 1/4 & 0 & 0 \\ 0 & 1 & 1/2 & 0 & 1/2 & 0 \\ 0 & 0 & 1 & -6 & 17 & 8 \end{array}\right)$$

Reduzierte Zeilenstufenform (linke Seite)

$$\rightsquigarrow \left(\begin{array}{ccc|ccc} 1 & 0 & 0 & 1 & -2 & -1 \\ 0 & 1 & 0 & 3 & -8 & -4 \\ 0 & 0 & 1 & -6 & 17 & 8 \end{array}\right) \Rightarrow A^{-1} = \begin{pmatrix} 1 & -2 & -1 \\ 3 & -8 & -4 \\ -6 & 17 & 8 \end{pmatrix}$$

LGS mit Inverse lösen $A \cdot \vec{x} = \vec{b}$

$$A^{-1} \cdot A \cdot \vec{x} = A^{-1} \cdot \vec{b} \rightarrow \vec{x} = A^{-1} \cdot \vec{b}$$

Beispiel:

$$\underbrace{\begin{pmatrix} 1/2 & 0 \\ 0 & 1/3 \end{pmatrix}}_{A^{-1}} \cdot \underbrace{\begin{pmatrix} 2 & 0 \\ 0 & 3 \end{pmatrix}}_{A} \cdot \underbrace{\begin{pmatrix} x \\ y \end{pmatrix}}_{\vec{x}} = \underbrace{\begin{pmatrix} 1/2 & 0 \\ 0 & 1/3 \end{pmatrix}}_{A^{-1}} \cdot \underbrace{\begin{pmatrix} 4 \\ 5 \end{pmatrix}}_{\vec{b}}$$

Numerische Lösung linearer Gleichungssysteme

Permutationsmatrix P ist eine Matrix, die aus der Einheitsmatrix durch Zeilenvertauschungen entsteht.

Für die Vertauschung der i-ten und j-ten Zeile hat P_k die **Form**:
- $p_{ii} = p_{jj} = 0$
- $p_{ij} = p_{ji} = 1$
- Sonst gleich wie in E_n

Wichtige Eigenschaften:
- $P^{-1} = P^T = P$
- Mehrere Vertauschungen: $P = P_l \cdot \ldots \cdot P_1$

Zeilenvertauschung für Matrix A mit Permutationsmatrix P_1:

$$\underbrace{\begin{pmatrix}1&2&3\\4&5&6\\7&8&9\end{pmatrix}}_{A} \cdot \underbrace{\begin{pmatrix}0&0&1\\0&1&0\\1&0&0\end{pmatrix}}_{P_1} = \begin{pmatrix}7&8&9\\4&5&6\\1&2&3\end{pmatrix}$$

$\Rightarrow A \cdot P_1$ bewirkt die Vertauschung von Zeile 1 und 3

Pivotisierung

Spaltenpivotisierung

Strategie zur numerischen Stabilisierung des Gauss-Algorithmus durch Auswahl des betragsmäßig größten Elements als Pivotelement. Vor jedem Eliminationsschritt in Spalte i:
- Suche k mit $|a_{ki}| = \max\{|a_{ji}| \mid j = i, \ldots, n\}$
- Falls $a_{ki} \neq 0$: Vertausche Zeilen i und k
- Falls $a_{ki} = 0$: Matrix ist singulär

Gauss-Algorithmus mit Pivotisierung

1. Elimination (Vorwärts):
- Für $i = 1, \ldots, n-1$:
  - Finde $k \geq i$ mit $|a_{ki}| = \max\{|a_{ji}| \mid j = i, \ldots, n\}$
  - Falls $a_{ki} = 0$: Stop (Matrix singulär)
  - Vertausche Zeilen i und k
  - Für $j = i+1, \ldots, n$:
    - $z_j := z_j - \frac{a_{ji}}{a_{ii}} z_i$

2. Rückwärtseinsetzen: $x_i = \frac{b_i - \sum_{j=i+1}^{n} a_{ij} x_j}{a_{ii}}, \quad i = n, n-1, \ldots, 1$

Gauss mit Pivotisierung $A = \begin{pmatrix}0&1&1\\2&4&-2\\0&3&15\end{pmatrix}, \quad b = \begin{pmatrix}4\\2\\36\end{pmatrix}$

Eliminationsschritte:

$$\left(\begin{array}{ccc|c}2&4&-2&2\\0&3&15&36\\0&1&1&4\end{array}\right) \Rightarrow \left(\begin{array}{ccc|c}2&4&-2&2\\0&3&15&36\\0&0&-2&-8\end{array}\right)$$

Rückwärtseinsetzen:

$$\begin{aligned} x_3 &= \tfrac{-8}{-2} = 4 \\ x_2 &= \tfrac{36-15(4)}{3} = 1 \\ x_1 &= \tfrac{2-4(4)+2}{2} = -6 \end{aligned}$$

Vorteile der Permutationsmatrix
- Exakte Nachverfolgung aller Zeilenvertauschungen
- Einfache Rückführung auf ursprüngliche Reihenfolge durch P^{-1}
- Kompakte Darstellung mehrerer Vertauschungen
- Numerisch stabile Implementierung der Pivotisierung

Zeilenvertauschungen verfolgen
1. Initialisiere $P = I_n$
2. Für jede Vertauschung von Zeile i und j:
   - Erstelle P_k durch Vertauschen von Zeilen i, j in I_n
   - Aktualisiere $P = P_k \cdot P$
   - Wende Vertauschung auf Matrix an: $A := P_k A$
3. Bei der LR-Zerlegung mit Pivotisierung:
   - $PA = LR$
   - Löse $Ly = Pb$ und $Rx = y$

Gauss-Algorithmus mit Pivotisierung

```python
def gauss_elimination(A, b):
    n = len(b)
    for i in range(n-1):

        # Pivotisierung
        k = np.argmax(abs(A[i:, i])) + i
        if A[k, i] == 0:
            raise ValueError("Matrix ist singulaer")
        A[[i, k]] = A[[k, i]]
        b[[i, k]] = b[[k, i]]
        # Elimination
        for j in range(i+1, n):
            factor = A[j, i] / A[i, i]
            A[j, i:] -= factor * A[i, i:]
            b[j] -= factor * b[i]

    # Rueckwaertseinsetzen
    x = np.zeros(n)
    for i in range(n-1, -1, -1):
        x[i] = (b[i] - np.dot(A[i, i+1:], x[i+1:])) /
             A[i, i]

    return x
```

Matrix-Zerlegungen

Dreieckszerlegung Eine Matrix $A \in \mathbb{R}^{n \times n}$ kann zerlegt werden in:

Untere Dreiecksmatrix L:
$l_{ij} = 0$ für $j > i$
Diagonale normiert ($l_{ii} = 1$)

Obere Dreiecksmatrix R:
$r_{ij} = 0$ für $i > j$
Diagonalelemente $\neq 0$

LR-Zerlegung

LR-Zerlegung mit Pivotisierung

```python
def lr_decomposition_with_pivoting(A):
    n = len(A)
    P = np.eye(n)     # Permutationsmatrix
    L = np.eye(n)     # Untere Dreiecksmatrix
    R = A.copy()      # Wird zur oberen Dreiecksmatrix
    for k in range(n-1):

        # Finde Pivotelement
        pivot = np.argmax(abs(R[k:,k])) + k
        if pivot != k:

            # Erzeuge Permutationsmatrix
            P_k = np.eye(n)
            P_k[[k,pivot]] = P_k[[pivot,k]]

            # Aktualisiere Matrizen
            P = P_k @ P
            R[[k,pivot]] = R[[pivot,k]]
            if k > 0:
                L[[k,pivot], :k] = L[[pivot,k], :k]

        # Elimination durchfuehren
        for i in range(k+1, n):
            factor = R[i,k] / R[k,k]
            L[i,k] = factor
            R[i,k:] -= factor * R[k,k:]

    return P, L, R
```

LR-Zerlegung

Jede reguläre Matrix A, für die der Gauss-Algorithmus ohne Zeilenvertauschungen durchführbar ist, lässt sich zerlegen in: $A = LR$ wobei L eine normierte untere und R eine obere Dreiecksmatrix ist.

Berechnung der LR-Zerlegung

So berechnen Sie die LR-Zerlegung:
1. Führen Sie Gauss-Elimination durch
2. R ist die resultierende obere Dreiecksmatrix
3. Die Eliminationsfaktoren $-\frac{a_{ji}}{a_{ii}}$ bilden L
4. Lösen Sie dann nacheinander:
   - $Ly = b$ (Vorwärtseinsetzen)
   - $Rx = y$ (Rückwärtseinsetzen)

LR-Zerlegung $A = \begin{pmatrix}-1&1&1\\1&-3&-2\\5&1&4\end{pmatrix}, \quad b = \begin{pmatrix}0\\5\\3\end{pmatrix}$

Schritt 1: Erste Spalte

Max. Element in 1. Spalte: $|a_{31}| = 5$, also Z1 und Z3 tauschen:

$$P_1 = \begin{pmatrix}0&0&1\\0&1&0\\1&0&0\end{pmatrix}, \quad A^{(1)} = \begin{pmatrix}5&1&4\\1&-3&-2\\-1&1&1\end{pmatrix}$$

Berechne Eliminationsfaktoren: $l_{21} = \frac{1}{5}, \quad l_{31} = -\frac{1}{5}$

Nach Elimination: $A^{(2)} = \begin{pmatrix}5&1&4\\0&-3.2&-2.8\\0&1.2&1.8\end{pmatrix}$

Schritt 2: Zweite Spalte

Max. Element in 2. Spalte unter Diagonale: $|-3.2| > |1.2|$, keine Vertauschung nötig.

Berechne Eliminationsfaktor: $l_{32} = -\frac{1.2}{-3.2} = \frac{3}{8}$

Nach Elimination: $R = \begin{pmatrix}5&1&4\\0&-3.2&-2.8\\0&0&2.85\end{pmatrix}$

Endergebnis

Die LR-Zerlegung mit $PA = LR$ ist:

$$P = P_1 = \begin{pmatrix}0&0&1\\0&1&0\\1&0&0\end{pmatrix}, L = \begin{pmatrix}1&0&0\\\frac{1}{5}&1&0\\-\frac{1}{5}&\frac{3}{8}&1\end{pmatrix}, R = \begin{pmatrix}5&1&4\\0&-3.2&-2.8\\0&0&2.85\end{pmatrix}$$

Lösung des Systems

1. $Pb = \begin{pmatrix}3\\5\\0\end{pmatrix}$
2. Löse $Ly = Pb$ durch Vorwärtseinsetzen: $y = \begin{pmatrix}3\\4.4\\2.85\end{pmatrix}$
3. Löse $Rx = y$ durch Rückwärtseinsetzen: $x = \begin{pmatrix}1\\-1\\1\end{pmatrix}$

Probe

$$Ax = \begin{pmatrix}-1&1&1\\1&-3&-2\\5&1&4\end{pmatrix}\begin{pmatrix}1\\-1\\1\end{pmatrix} = \begin{pmatrix}0\\5\\3\end{pmatrix} = b$$

QR-Zerlegung

QR-Zerlegung

Eine orthogonale Matrix $Q \in \mathbb{R}^{n \times n}$ erfüllt: $Q^T Q = QQ^T = I_n$

Die QR-Zerlegung einer Matrix A ist: $A = QR$

wobei Q orthogonal und R eine obere Dreiecksmatrix ist.

Householder-Transformation

Eine Householder-Matrix hat die Form: $H = I_n - 2uu^T$

mit $u \in \mathbb{R}^n$, $\|u\| = 1$. Es gilt:

- H ist orthogonal ($H^T = H^{-1}$)
- H ist symmetrisch ($H^T = H$)
- $H^2 = I_n$

QR-Zerlegung mit Householder

1. Initialisierung: $R := A$, $Q := I_n$
2. Für $i = 1, \ldots, n-1$:
   - Bilde Vektor v_i aus i-ter Spalte von R ab Position i
   - $w_i := v_i + \text{sign}(v_{i1})\|v_i\|e_1$
   - $u_i := w_i / \|w_i\|$
   - $H_i := I_{n-i+1} - 2u_i u_i^T$
   - Erweitere H_i zu Q_i durch I_{i-1} links oben
   - $R := Q_i R$ und $Q := QQ_i^T$

QR-Zerlegung Implementation

```python
def householder_vector(x):
    # Berechne Householder-Vektor fuer Spalte x
    alpha = np.linalg.norm(x)
    v = x.copy()
    v[0] += np.sign(x[0]) * alpha
    v = v / np.linalg.norm(v)
    return v

def householder_reflection(A, k):
    m, n = A.shape
    v = householder_vector(A[k:, k])
    # Householder-Matrix anwenden
    H = np.eye(m-k)
    H -= 2 * np.outer(v, v)
    # Auf Untermatrix anwenden
    A[k:, k:] = H @ A[k:, k:]
    return A

def qr_householder(A):
    m, n = A.shape
    R = A.copy()
    Q = np.eye(m)

    for k in range(n):
        v = householder_vector(R[k:, k])
        H = np.eye(m)
        H[k:, k:] -= 2 * np.outer(v, v)
        R = H @ R
        Q = Q @ H.T

    return Q, R
```

Numerische Vorteile

- Numerisch stabil
- Keine Wurzeloperationen während der Elimination
- Orthogonalität der Transformation bleibt erhalten
- Gute Eignung für Eigenwertberechnung

QR-Zerlegung mit Householder $A = \begin{pmatrix} 2 & 5 & -1 \\ -1 & -4 & 2 \\ 0 & 2 & 1 \end{pmatrix}$

Schritt 1: Erste Spalte

Erste Spalte a_1 und Einheitsvektor e_1: $a_1 = \begin{pmatrix} 2 \\ -1 \\ 0 \end{pmatrix}$, $e_1 = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$

Householder-Vektor für erste Spalte:

1. Berechne Norm: $|a_1| = \sqrt{2^2 + (-1)^2 + 0^2} = \sqrt{5}$
2. Bestimme Vorzeichen: $\text{sign}(a_{11}) = \text{sign}(2) = 1$
   - Wähle positives Vorzeichen, da erstes Element positiv
   - Dies maximiert die erste Komponente von v_1
   - Verhindert Auslöschung bei der Subtraktion
3. $v_1 = a_1 + \text{sign}(a_{11})|a_1|e_1 = \begin{pmatrix} 2 \\ -1 \\ 0 \end{pmatrix} + \sqrt{5}\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 2+\sqrt{5} \\ -1 \\ 0 \end{pmatrix}$
4. Normiere v_1: $|v_1| = \sqrt{(2+\sqrt{5})^2 + 1} \Rightarrow u_1 = \frac{v_1}{|v_1|} = \begin{pmatrix} 0.91 \\ -0.41 \\ 0 \end{pmatrix}$

Householder-Matrix berechnen:

$$H_1 = I - 2u_1 u_1^T = \begin{pmatrix} -0.67 & -0.75 & 0 \\ -0.75 & 0.67 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

A nach erster Transformation:

$$A^{(1)} = H_1 A = \begin{pmatrix} -\sqrt{5} & -6.71 & 0.45 \\ 0 & -0.89 & 1.79 \\ 0 & 2.00 & 1.00 \end{pmatrix}$$

Schritt 2: Zweite Spalte

Untermatrix für zweite Transformation: $A_2 = \begin{pmatrix} -0.89 & 1.79 \\ 2.00 & 1.00 \end{pmatrix}$

Householder-Vektor für zweite Spalte:

1. $|a_2| = \sqrt{(-0.89)^2 + 2^2} = 2.19$
2. $\text{sign}(a_{22}) = \text{sign}(-0.89) = -1$ (da erstes Element negativ)
3. $v_2 = \begin{pmatrix} -0.89 \\ 2.00 \end{pmatrix} - 2.19\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} -3.09 \\ 2.00 \end{pmatrix}$
4. $u_2 = \frac{v_2}{|v_2|} = \begin{pmatrix} -0.84 \\ 0.54 \end{pmatrix}$

Erweiterte Householder-Matrix: $Q_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & -0.41 & -0.91 \\ 0 & -0.91 & 0.41 \end{pmatrix}$

nach 2. Transformation: $R = Q_2 A^{(1)} = \begin{pmatrix} -\sqrt{5} & -6.71 & 0.45 \\ 0 & -2.19 & 1.34 \\ 0 & 0 & -1.79 \end{pmatrix}$

Endergebnis

Die QR-Zerlegung $A = QR$ ist:

$$Q = H_1^T Q_2^T = \begin{pmatrix} -0.89 & -0.45 & 0 \\ 0.45 & -0.89 & 0 \\ 0 & 0 & 1 \end{pmatrix}, R = \begin{pmatrix} -\sqrt{5} & -6.71 & 0.45 \\ 0 & -2.19 & 1.34 \\ 0 & 0 & -1.79 \end{pmatrix}$$

Probe

1. $QR = A$ (bis auf Rundungsfehler)
2. $Q^T Q = QQ^T = I$ (Orthogonalität)
3. R ist obere Dreiecksmatrix

Wichtige Beobachtungen

- Die Wahl des Vorzeichens bei der Berechnung von v_k ist entscheidend für die numerische Stabilität
- Ein falsches Vorzeichen kann zu Auslöschung führen
- Der Betrag der Diagonalelemente in R entspricht der Norm der transformierten Spalten
- Q ist orthogonal: Spaltenvektoren sind orthonormal

Fehleranalyse

Matrix- und Vektornormen

Eine Vektornorm $\|\cdot\|$ erfüllt für alle $x, y \in \mathbb{R}^n, \lambda \in \mathbb{R}$:

- $\|x\| \geq 0$ und $\|x\| = 0 \Leftrightarrow x = 0$
- $\|\lambda x\| = |\lambda| \cdot \|x\|$
- $\|x + y\| \leq \|x\| + \|y\|$ (Dreiecksungleichung)

Wichtige Normen

1-Norm:

$$\|x\|_1 = \sum_{i=1}^{n} |x_i|, \|A\|_1 = \max_j \sum_{i=1}^{n} |a_{ij}|$$

2-Norm:

$$\|x\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2}, \|A\|_2 = \sqrt{\rho(A^T A)}$$

∞-Norm:

$$\|x\|_\infty = \max_i |x_i|, \|A\|_\infty = \max_i \sum_{j=1}^{n} |a_{ij}|$$

Fehlerabschätzung für LGS

Sei $\|\cdot\|$ eine Norm, $A \in \mathbb{R}^{n \times n}$ regulär und $Ax = b$, $A\tilde{x} = \tilde{b}$

Absoluter Fehler:

$$\|x - \tilde{x}\| \leq \|A^{-1}\| \cdot \|b - \tilde{b}\|$$

Relativer Fehler:

$$\frac{\|x - \tilde{x}\|}{\|x\|} \leq \text{cond}(A) \cdot \frac{\|b - \tilde{b}\|}{\|b\|}$$

Mit der Konditionszahl $\text{cond}(A) = \|A\| \cdot \|A^{-1}\|$

Konditionierung

Die Konditionszahl beschreibt die numerische Stabilität eines LGS:

- $\text{cond}(A) \approx 1$: gut konditioniert
- $\text{cond}(A) \gg 1$: schlecht konditioniert
- $\text{cond}(A) \to \infty$: singulär

Konditionierung $A = \begin{pmatrix} 1 & 1 \\ 1 & 1.01 \end{pmatrix}$, $b = \begin{pmatrix} 2 \\ 2.01 \end{pmatrix}$

Konditionszahl: $\text{cond}(A) = \|A\| \cdot \|A^{-1}\| \approx 400$

Fehlerabschätzung

$$\text{Absoluter Fehler: } \|x - \tilde{x}\| \leq 400 \cdot 0.01 = 4$$

$$\text{Relativer Fehler: } \frac{\|x - \tilde{x}\|}{\|x\|} \leq 400 \cdot \frac{0.01}{2} = 2$$

Fehlerabschätzung

```python
def error_estimate(A, b, x, b_tilde):
    # Absoluter Fehler
    abs_error = np.linalg.norm(x - np.linalg.solve(A,
        b_tilde))
    # Relativer Fehler
    rel_error = abs_error / np.linalg.norm(x)
    return abs_error, rel_error
```

Iterative Verfahren

Zerlegung der Systemmatrix A zerlegt in: $A = L + D + R$

- L: streng untere Dreiecksmatrix
- D: Diagonalmatrix
- R: streng obere Dreiecksmatrix

Jacobi-Verfahren Gesamtschrittverfahren mit der Iteration:

$$x^{(k+1)} = -D^{-1}(L+R)x^{(k)} + D^{-1}b$$

Komponentenweise: $x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1, j\neq i}^{n} a_{ij}x_j^{(k)}\right)$

Gauss-Seidel-Verfahren Einzelschrittverfahren mit der Iteration:

$$x^{(k+1)} = -(D+L)^{-1}Rx^{(k)} + (D+L)^{-1}b$$

Komponentenweise:
$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}\right)$

Konvergenzkriterien Ein iteratives Verfahren konvergiert, wenn:

1. Die Matrix A diagonaldominant ist:
   $|a_{ii}| > \sum_{j\neq i} |a_{ij}|$ für alle i
2. Der Spektralradius der Iterationsmatrix kleiner 1 ist:
   $\rho(B) < 1$ mit B als jeweilige Iterationsmatrix

Implementation iterativer Verfahren

1. Wählen Sie Startvektor $x^{(0)}$
2. Wählen Sie Abbruchkriterien:
   - Maximale Iterationszahl k_{max}
   - Toleranz ϵ für Änderung $\|x^{(k+1)} - x^{(k)}\|$
   - Toleranz für Residuum $\|Ax^{(k)} - b\|$
3. Führen Sie Iteration durch bis Kriterien erfüllt

Iterative Verfahren Vergleich Jacobi und Gauss-Seidel System:

$$\begin{pmatrix} 4 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 4 \end{pmatrix} x = \begin{pmatrix} 1 \\ 5 \\ 0 \end{pmatrix}$$

k	Jacobi		Gauss-Seidel	
0	$(0,0,0)^T$		$(0,0,0)^T$	
1	$(0.25, 1.25, 0)^T$	1.25	$(0.25, 1.31, 0.08)^T$	1.31
2	$(0.31, 1.31, 0.31)^T$	0.31	$(0.33, 1.33, 0.33)^T$	0.02
3	$(0.33, 1.33, 0.33)^T$	0.02	$(0.33, 1.33, 0.33)^T$	0.00

Jacobi- und Gauss-Seidel-Verfahren

```python
def jacobi_iteration(A, b, x):
    D = np.diag(np.diag(A))
    L = np.tril(A, -1)
    R = np.triu(A, 1)
    x_new = np.linalg.solve(D, b - (L + R) @ x)
    return x_new

def gauss_seidel_iteration(A, b, x):
    D = np.diag(np.diag(A))
    L = np.tril(A, -1)
    R = np.triu(A, 1)
    x_new = np.linalg.solve(D + L, b - R @ x)
    return x_new
```

Eigenwerte und Eigenvektoren

Komplexe Zahlen

Fundamentalsatz der Algebra
Eine algebraische Gleichung n-ten Grades mit komplexen Koeffizienten:

$$a_n z^n + a_{n-1}z^{n-1} + \cdots + a_1 z + a_0 = 0$$

besitzt in $\mathbb{C}$ genau n Lösungen (mit Vielfachheiten gezählt).

Komplexe Zahlen
Die Menge der komplexen Zahlen $\mathbb{C}$ erweitert die reellen Zahlen $\mathbb{R}$ durch Einführung der imaginären Einheit i mit der Eigenschaft:

$$i^2 = -1$$

Eine komplexe Zahl z ist ein geordnetes Paar (x, y) mit $x, y \in \mathbb{R}$:

$$z = x + iy$$

Die Menge aller komplexen Zahlen ist definiert als:

$$\mathbb{C} = \{z \mid z = x + iy \text{ mit } x, y \in \mathbb{R}\}$$

Bestandteile komplexer Zahlen

Realteil: $\mathrm{Re}(z) = x$ **Konjugation:** $\overline{z} = x - iy$

Imaginärteil: $\mathrm{Im}(z) = y$ **Betrag:** $|z| = \sqrt{x^2 + y^2} = \sqrt{z \cdot z^*}$

Rechenoperationen mit komplexen Zahlen
Für $z_1 = x_1 + iy_1$ und $z_2 = x_2 + iy_2$ gilt:

Addition:
$z_1 + z_2 = (x_1 + x_2) + i(y_1 + y_2)$

Subtraktion:
$z_1 - z_2 = (x_1 - x_2) + i(y_1 - y_2)$

Multiplikation:

$$\begin{aligned} z_1 \cdot z_2 &= (x_1x_2 - y_1y_2) + i(x_1y_2 + x_2y_1) \\ &= r_1r_2e^{i(\varphi_1+\varphi_2)} \text{ (in Exponentialform)} \end{aligned}$$

Division:

$$\begin{aligned} \frac{z_1}{z_2} &= \frac{z_1 \cdot z_2^*}{z_2 \cdot z_2^*} = \frac{(x_1x_2 + y_1y_2) + i(y_1x_2 - x_1y_2)}{x_2^2 + y_2^2} \\ &= \frac{r_1}{r_2}e^{i(\varphi_1-\varphi_2)} \text{ (in Exponentialform)} \end{aligned}$$

Potenzen und Wurzeln
Für eine komplexe Zahl in Exponentialform $z = re^{i\varphi}$ gilt:

- n-te Potenz: $z^n = r^n e^{in\varphi} = r^n(\cos(n\varphi) + i\sin(n\varphi))$
- n-te Wurzel: $z_k = \sqrt[n]{r}e^{i\frac{\varphi+2\pi k}{n}}$, $k = 0, 1, \ldots, n-1$

Darstellungsformen

- Normalform: $z = x + iy$
- Trigonometrische Form: $z = r(\cos\varphi + i\sin\varphi)$
- Exponentialform: $z = re^{i\varphi}$

$x = r\cos\varphi, \quad y = r\sin\varphi, \quad r = \sqrt{x^2 + y^2}$
$\varphi = \arcsin\left(\frac{y}{r}\right) = \arccos\left(\frac{x}{r}\right)$
$e^{i\varphi} = \cos\varphi + i\sin\varphi$ (Euler-Formel)

Umrechnung zwischen Darstellungsformen komplexer Zahlen

Von Normalform in trigonometrische Form/Exponentialform

1. Berechne Betrag $r = \sqrt{x^2 + y^2}$
2. Berechne Winkel mit einer der Formeln:
   - $\varphi = \arctan(\frac{y}{x})$ falls $x > 0$
   - $\varphi = \arctan(\frac{y}{x}) + \pi$ falls $x < 0$
   - $\varphi = \frac{\pi}{2}$ falls $x = 0, y > 0$
   - $\varphi = -\frac{\pi}{2}$ falls $x = 0, y < 0$
   - φ unbestimmt falls $x = y = 0$
3. Trigonometrische Form: $z = r(\cos\varphi + i\sin\varphi)$
4. Exponentialform: $z = re^{i\varphi}$

Von trigonometrischer Form in Normalform

1. Realteil: $x = r\cos\varphi$
2. Imaginärteil: $y = r\sin\varphi$
3. Normalform: $z = x + iy$

Von Exponentialform in Normalform/trigonometrische Form

1. Trigonometrische Form durch Euler-Formel:
   $re^{i\varphi} = r(\cos\varphi + i\sin\varphi)$
2. Dann wie oben in Normalform umrechnen

Wichtige Hinweise:

- Achten Sie auf das korrekte Quadranten beim Winkel
- Winkelfunktionen im Bogenmaß verwenden
- Bei Umrechnung in Normalform Euler-Formel nutzen
- Vorzeichen bei Exponentialform beachten

Darstellungsformen Gegeben: $z = 3 - 11i$ in Normalform

$$r = \sqrt{3^2 + 11^2} = \sqrt{130}, \quad \varphi = \arcsin(\frac{11}{\sqrt{130}}) = 1.3\text{rad} = 74.74°$$

Trigonometrische Form: $z = \sqrt{130}(\cos(1.3) + i\sin(1.3))$

Exponentialform: $z = \sqrt{130}e^{i\cdot 1.3}$

Komplexe Zahlen in Python

```python
import numpy as np
z1 = 3 - 11j
z2 = 2 + 5j
# Addition
z_add = z1 + z2
# Subtraktion
z_sub = z1 - z2
# Multiplikation
z_mul = z1 * z2
# Division
z_div = z1 / z2
# Betrag
r = np.abs(z1)
# Winkel
phi = np.angle(z1)
# Exponentialform
z_exp = r * np.exp(1j * phi)
# Potenz
z_pow = z1 ** 2
# Wurzel
z_sqrt = np.sqrt(z1)

# Darstellungsformen
z_trig = r * (np.cos(phi) + 1j * np.sin(phi))
z_norm = z_trig.real + 1j * z_trig.imag
```

Eigenwerte und Eigenvektoren

Eigenwerte und Eigenvektoren

Für eine Matrix $A \in \mathbb{R}^{n\times n}$ heißt $\lambda \in \mathbb{C}$ Eigenwert von A, wenn es einen Vektor $x \in \mathbb{C}^n \backslash \{0\}$ gibt mit:

$$Ax = \lambda x$$

Der Vektor x heißt dann Eigenvektor zum Eigenwert λ.

Bestimmung von Eigenwerten

Ein Skalar λ ist genau dann Eigenwert von A, wenn gilt:

$$\det(A - \lambda I_n) = 0$$

Diese Gleichung heißt charakteristische Gleichung. Das zugehörige Polynom

$$p(\lambda) = \det(A - \lambda I_n)$$

ist das charakteristische Polynom von A.

Eigenschaften von Eigenwerten

Für eine Matrix $A \in \mathbb{R}^{n\times n}$ gilt:

- $\det(A) = \prod_{i=1}^{n} \lambda_i$ (Produkt der Eigenwerte)
- $\mathrm{tr}(A) = \sum_{i=1}^{n} \lambda_i$ (Summe der Eigenwerte)
- Bei einer Dreiecksmatrix sind die Diagonalelemente die Eigenwerte
- Ist λ Eigenwert von A, so ist $\frac{1}{\lambda}$ Eigenwert von A^{-1}

Vielfachheiten

Für einen Eigenwert λ unterscheidet man:

- Algebraische Vielfachheit: Vielfachheit als Nullstelle des charakteristischen Polynoms
- Geometrische Vielfachheit: Dimension des Eigenraums $= n - \mathrm{rg}(A - \lambda I_n)$

Die geometrische Vielfachheit ist stets kleiner oder gleich der algebraischen Vielfachheit.

Bestimmung von Eigenwerten und Eigenvektoren

1. Charakteristisches Polynom aufstellen: $p(\lambda) = \det(A - \lambda I_n)$
2. Eigenwerte durch Lösen von $p(\lambda) = 0$ bestimmen
3. Für jeden Eigenwert λ_i:
   - System $(A - \lambda_i I_n)x = 0$ aufstellen
   - Lösungsraum = Eigenraum bestimmen
   - Basis des Eigenraums = linear unabhängige Eigenvektoren

Eigenwertberechnung

$$A = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 3 & 0 \\ 0 & 1 & 2 \end{pmatrix}$$

1. Da A eine Dreiecksmatrix ist, sind die Diagonalelemente die Eigenwerte:
   $$\lambda_1 = 1, \lambda_2 = 3, \lambda_3 = 2$$
2. $\det(A) = \lambda_1 \cdot \lambda_2 \cdot \lambda_3 = 6$
3. $\mathrm{tr}(A) = \lambda_1 + \lambda_2 + \lambda_3 = 6$
4. Spektrum: $\sigma(A) = \{1, 2, 3\}$

EW und EV über Charakteristisches Polynom

```python
A = np.array([[1, 0, 0], [2, 3, 0], [0, 1, 2]])
# Charakteristisches Polynom
p = np.poly(A)
# Eigenwerte
eigenvalues = np.roots(p)
# Eigenvektoren
eigenvectors = []
for l in eigenvalues:
    eigenvectors.append(np.linalg.solve(A - l *
        np.eye(A.shape[0]), np.zeros(A.shape[0])))
```

Numerische Berechnung von Eigenwerten

Ähnliche Matrizen

Zwei Matrizen $A, B \in \mathbb{R}^{n\times n}$ heißen ähnlich, wenn es eine reguläre Matrix T gibt mit:

$$B = T^{-1}AT$$

Eine Matrix A heißt diagonalisierbar, wenn sie ähnlich zu einer Diagonalmatrix D ist:

$$D = T^{-1}AT$$

Eigenschaften ähnlicher Matrizen

Für ähnliche Matrizen A und $B = T^{-1}AT$ gilt:

1. A und B haben dieselben Eigenwerte mit gleichen algebraischen Vielfachheiten
2. Ist x Eigenvektor von B zum Eigenwert λ, so ist Tx Eigenvektor von A zum Eigenwert λ
3. Bei Diagonalisierbarkeit:
   - Die Diagonalelemente von D sind die Eigenwerte von A
   - Die Spalten von T sind die Eigenvektoren von A

Spektralradius

Der Spektralradius einer Matrix A ist definiert als:

$$\rho(A) = \max\{|\lambda| \mid \lambda \text{ ist Eigenwert von } A\}$$

Er gibt den Betrag des betragsmäßig größten Eigenwerts an.

Iterative Verfahren

Von-Mises-Iteration (Vektoriteration)

Für eine diagonalisierbare Matrix A mit Eigenwerten $|\lambda_1| > |\lambda_2| \geq \cdots \geq |\lambda_n|$ konvergiert die Folge:

$$v^{(k+1)} = \frac{Av^{(k)}}{\|Av^{(k)}\|_2}, \quad \lambda^{(k+1)} = \frac{(v^{(k)})^T Av^{(k)}}{(v^{(k)})^T v^{(k)}}$$

gegen einen Eigenvektor v zum betragsmäßig größten Eigenwert λ_1.

Von-Mises-Iteration durchführen

1. Wähle Startvektor $v^{(0)}$ mit $\|v^{(0)}\|_2 = 1$
2. Für $k = 0, 1, 2, \ldots$:
   - Berechne $w^{(k)} = Av^{(k)}$
   - Normiere: $v^{(k+1)} = \frac{w^{(k)}}{\|w^{(k)}\|_2}$
   - Berechne Rayleigh-Quotienten $\lambda^{(k+1)}$
   - Prüfe Konvergenz

Von-Mises-Iteration Berechnung des größten Eigenwerts

```python
import numpy as np
def power_iteration(A, tol=1e-10, max_iter=100):
    n = A.shape[0]
    v = np.random.rand(n)
    v = v / np.linalg.norm(v)
    for i in range(max_iter):
        w = A @ v
        v_new = w / np.linalg.norm(w)
        # Rayleigh-Quotient
        lambda_k = v_new.T @ A @ v_new
        if np.linalg.norm(v_new - v) < tol:
            return lambda_k, v_new
        v = v_new
    return lambda_k, v_new
```

QR-Verfahren

Das QR-Verfahren transformiert die Matrix A iterativ in eine obere Dreiecksmatrix, deren Diagonalelemente die Eigenwerte sind:

1. Initialisierung: $A_0 := A$, $P_0 := I_n$
2. Für $i = 0, 1, 2, \ldots$:
   - QR-Zerlegung: $A_i = Q_i R_i$
   - Neue Matrix: $A_{i+1} = R_i Q_i$
   - Update: $P_{i+1} = P_i Q_i$

QR-Verfahren anwenden

1. Matrix $A_0 = A$ vorbereiten
2. In jedem Schritt i:
   - QR-Zerlegung mit Householder oder Givens
   - Neue Matrix durch Multiplikation $R_i Q_i$
   - Konvergenz prüfen: Subdiagonalelemente ≈ 0?
3. Eigenwerte: Diagonalelemente der Endmatrix
4. Eigenvektoren: Spalten von $P = P_1 P_2 \cdots P_k$

QR-Verfahren Implementation in Python

```python
def qr_algorithm(A, tol=1e-10, max_iter=100):
    n = A.shape[0]
    Q_prod = np.eye(n)
    A_k = A.copy()

    for k in range(max_iter):
        # QR decomposition
        Q, R = np.linalg.qr(A_k)
        # New iteration
        A_k = R @ Q
        # Update transformation matrix
        Q_prod = Q_prod @ Q

        # Check convergence
        if np.abs(np.tril(A_k, -1)).max() < tol:
            break

    return np.diag(A_k), Q_prod
```

Numerische Stabilität

- QR-Verfahren ist numerisch stabiler als Vektoriteration
- Findet alle Eigenwerte, nicht nur den größten
- Benötigt mehr Rechenaufwand
- Konvergiert linear für reelle, quadratisch für komplexe Eigenwerte

Python

Numerische Bibliotheken Verwendung spezialisierter Bibliotheken
Für kritische numerische Berechnungen:

- NumPy: Optimierte Array-Operationen
- SciPy: Wissenschaftliches Rechnen
- Mpmath: Beliebige Präzision
- Decimal: Dezimalarithmetik

Bibliotheksverwendung Beispiel: Präzise Berechnung mit Decimal

```python
from decimal import Decimal, getcontext

# Set precision
getcontext().prec = 40

# Precise calculation
x = Decimal('1.0') / Decimal('7.0')
print(x)  # 0.1428571428571428571428571428571428571428
```

NumPy

NumPy NumPy: Numerische Python-Bibliothek

- Effiziente Implementierung von Arrays
- Vektorisierte Operationen
- Lineare Algebra, Fourier-Transformation, Zufallszahlen

ACHTUNG: darf an der Prüfung höchstwahrscheinlich nicht verwendet werden! aber falls doch, hier die einfachen Implementationen von allem :D

Eigenwerte und Eigenvektoren

```python
import numpy as np
A = np.array([[1, 0, 0], [2, 3, 0], [0, 1, 2]])
# Eigenwerte
eigenvalues = np.linalg.eigvals(A)
# Eigenwerte und Eigenvektoren
eigenvalues, eigenvectors = np.linalg.eig(A)
```

Examples