

Rechnerarithmetik

Zahlendarstellung

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

$$M = \{x \in \mathbb{R} \mid x = \pm 0.m_1m_2m_3 \dots m_n \cdot B^{\pm e_1e_2 \dots e_l}\} \cup \{0\}$$

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

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

Werteberechnung einer Maschinenzahl

- Normalisierung überprüfen:  $m_1 \neq 0$  (für  $x \neq 0$ )
  - Sonst: Mantisse verschieben und Exponent anpassen
- Exponent berechnen:  $\hat{e} = \sum_{i=1}^l e_i B^{l-i}$ 
  - Von links nach rechts: Stelle  $\cdot$  Basis hochgestellt zur Position
- Wert berechnen:  $\hat{\omega} = \sum_{i=1}^n m_i B^{\hat{e}-i}$ 
  - Mantissenstellen  $\cdot$  Basis hochgestellt zu (Exponent - Position)
- Vorzeichen berücksichtigen

**Werteberechnung** Berechnung einer vierstelligen Zahl zur Basis 4:

$$\underbrace{0.3211}_{n=4} \cdot \underbrace{4^{12}}_{l=2} \quad \text{Exponent: } \hat{e} = 1 \cdot 4^1 + 2 \cdot 4^0 = 6$$

$$\text{Wert: } \hat{\omega} = 3 \cdot 4^5 + 2 \cdot 4^4 + 1 \cdot 4^3 + 1 \cdot 4^2 = 3664$$

**IEEE-754 Standard** definiert zwei wichtige Gleitpunktformate:

<b>Single Precision</b> (32 Bit)	<b>Double Precision</b> (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  $x$ :

<b>Absoluter Fehler:</b>	<b>Relativer Fehler:</b>
$ \tilde{x} - x $	$\frac{ \tilde{x} - x }{ x }$ 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:

<b>Absoluter Fehler:</b>	<b>Relativer Fehler:</b>
$ rd(x) - x  \leq \frac{B}{2} \cdot B^{e-n-1}$	$\left  \frac{rd(x) - x}{x} \right  \leq \text{eps}$

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

- Berechnen Sie  $f'(x)$
- Bestimmen Sie die Konditionszahl  $K$
- Schätzen Sie den absoluten Fehler ab
- Schätzen Sie den relativen Fehler ab
- Beurteilen Sie die Konditionierung anhand von  $K$

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

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

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

- $f'(x) = \cos(x)$
- $K = \frac{|x \cos(x)|}{|\sin(x)|}$
- Für  $x \rightarrow 0$ :  $K \rightarrow 1$  (gut konditioniert)
- Für  $x \rightarrow \pi$ :  $K \rightarrow \infty$  (schlecht konditioniert)
- Für  $x = 0$ :  $\lim_{x \rightarrow 0} K = 1$  (gut konditioniert)
- 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

- Identifizieren Sie Subtraktionen ähnlich großer Zahlen
- Suchen Sie nach algebraischen Umformungen
- Prüfen Sie alternative Berechnungswege
- Verwenden Sie Taylorentwicklungen für kleine Werte

**Auslöschung** Kritische Berechnungen für kleine  $x$  (Auslöschung):

- $\sqrt{1+x^2} - 1$ : **Besser:**  $\frac{x^2}{\sqrt{1+x^2}+1}$
- $1 - \cos(x)$ : **Besser:**  $2 \sin^2(x/2)$

**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

Numerische Lösung von Nullstellenproblemen

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

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

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

Systematisches Vorgehen bei Nullstellenproblemen

- Newton-Verfahren: wenn Ableitung leicht berechenbar
- Sekantenverfahren: wenn Ableitung schwierig
- Fixpunktiteration: wenn geeignete Umformung möglich

NSP: Nullstellenproblem, NS: Nullstelle

Fixpunktiteration

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

**Grundprinzip der Fixpunktiteration** sei  $F : [a, b] \rightarrow \mathbb{R}$  mit  $x_0 \in [a, b]$   
Die rekursive Folge  $x_{n+1} \equiv F(x_n)$ ,  $n = 0, 1, 2, \dots$

heisst Fixpunktiteration von  $F$  zum Startwert  $x_0$ .

Konvergenzverhalten

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

<b>Anziehender Fixpunkt:</b>	<b>Abtossender Fixpunkt:</b>
$ F'(\tilde{x})  < 1$	$ F'(\tilde{x})  > 1$

$x_n$  konvergiert gegen  $\tilde{x}$ , falls  $x_0$  nahe genug bei  $\tilde{x}$        $x_n$  konvergiert für keinen Startwert  $x_0 \neq \tilde{x}$

**Banachscher Fixpunktsatz**  $F : [a, b] \rightarrow [a, b]$  und  $\exists$  Konstante  $\alpha$ :

- $0 < \alpha < 1$  (Lipschitz-Konstante)
- $|F(x) - F(y)| \leq \alpha |x - y|$  für alle  $x, y \in [a, b]$

Dann gilt:

<b>Fehlerabschätzungen:</b>
<b>a-priori:</b> $ x_n - \tilde{x}  \leq \frac{\alpha^n}{1 - \alpha} \cdot  x_1 - x_0 $
<b>a-posteriori:</b> $ x_n - \tilde{x}  \leq \frac{\alpha}{1 - \alpha} \cdot  x_n - x_{n-1} $ $\tilde{x}$ für alle $x_0 \in [a, b]$

Konvergenznachweis für Fixpunktiteration

- Bringe die Gleichung in Fixpunktform:  $f(x) = 0 \Rightarrow x = F(x)$
- Prüfe, ob  $F$  das Intervall  $[a, b]$  in sich abbildet:
  - Wähle geeignetes Intervall ( $[a, b]$   $F(a) \geq a$  und  $F(b) \leq b$ )
- Bestimme die Lipschitz-Konstante  $\alpha$ :  $\rightarrow$  Berechne  $F'(x)$ 
  - Finde  $\alpha = \max_{x \in [a, b]} |F'(x)|$  und prüfe  $\alpha < 1$
- Berechnen Sie die nötigen Iterationen für Genauigkeit  $\text{tol}$ :  $n \geq \frac{\ln(\frac{\text{tol} \cdot (1 - \alpha)}{|x_1 - x_0|})}{\ln \alpha}$

**Fixpunktiteration** Nullstellen von  $f(x) = e^x - x - 2$   
Umformung in Fixpunktform:  $x = \ln(x + 2)$ , also  $F(x) = \ln(x + 2)$

- $F'(x) = \frac{1}{x+2}$  monoton fallend
- Für  $I = [1, 2]$ :  $F(1) = 1.099 > 1$ ,  $F(2) = 1.386 < 2$
- $\alpha = \max_{x \in [1, 2]} |\frac{1}{x+2}| = \frac{1}{3} < 1$
- Konvergenz für Startwerte in  $[1, 2]$  gesichert
- Für Genauigkeit  $10^{-6}$  benötigt:  $n \geq 12$  Iterationen

Grundprinzip Newton-Verfahren

Approximation der NS durch sukzessive Tangentenberechnung:  
Konvergiert, wenn für alle  $x$  im relevanten Intervall gilt:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$
$$\left| \frac{f(x) \cdot f''(x)}{[f'(x)]^2} \right| < 1$$

Newton-Verfahren anwenden

- Funktion  $f(x)$  und Ableitung  $f'(x)$  aufstellen
- Geeigneten Startwert  $x_0$  nahe der Nullstelle wählen
  - Prüfen, ob  $f'(x_0) \neq 0$
- Iterieren bis zur gewünschten Genauigkeit:  $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$
- Abbruchkriterien prüfen:
  - Funktionswert:  $|f(x_n)| < \epsilon_1$
  - Änderung aufeinanderfolgenden Werte:  $|x_{n+1} - x_n| < \epsilon_2$
  - Maximale Iterationszahl nicht überschritten

**Newton-Verfahren** Nullstellen von  $f(x) = x^2 - 2$   
Ableitung:  $f'(x) = 2x$ , Startwert  $x_0 = 1$

- $x_1 = 1 - \frac{1^2-2}{2 \cdot 1} = 1.5 \rightarrow$  Konvergenz gegen  $\sqrt{2}$  nach wenigen Schritten
- $x_2 = 1.5 - \frac{1.5^2-2}{2 \cdot 1.5} = 1.4167$
- $x_3 = 1.4167 - \frac{1.4167^2-2}{2 \cdot 1.4167} = 1.4142$

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  $x_1 = 2$

- $x_2 = 1 - \frac{1-2}{1^2-2} \cdot 1 = 1.5 \rightarrow$  Konvergenz gegen  $\sqrt{2}$  nach wenigen Schritten
- $x_3 = 1.5 - \frac{1.5-1}{1.5^2-2} \cdot 1.5 = 1.4545$
- $x_4 = 1.4545 - \frac{1.4545-1.5}{1.4545^2-2} \cdot 1.4545 = 1.4143$

Fehlerabschätzung

Fehlerabschätzung für Nullstellen

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

**Praktische Fehlerabschätzung** Fehlerbestimmung bei  $f(x) = x^2 - 2$

- Näherungswert:  $x_3 = 1.4142157$
- Mit  $\epsilon = 10^{-5}$ : **Also:**  $|x_3 - \sqrt{2}| < 10^{-5}$
- $f(x_3 - \epsilon) = 1.4142057^2 - 2 < 0 \rightarrow$  Nullstelle liegt in  $(1.4142057, 1.4142257)$
- $f(x_3 + \epsilon) = 1.4142257^2 - 2 > 0$

**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

LGS und Matrizen

Matrizen

**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

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

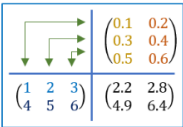
Kommutativ-, Assoziativ- und Distributiv-Gesetz gelten für Matrix-Addition

Matrixmultiplikation  $A^{m \times n}, B^{n \times k}$

Bedingung:  $A$   $n$  Spalten,  $B$   $n$  Zeilen.

Resultat:  $C$  hat  $m$  Zeilen und  $k$  Spalten.

- $A \cdot B = C$
- $c_{ij} = a_{i1} \cdot b_{1j} + a_{i2} \cdot b_{2j} + \dots + a_{in} \cdot b_{nj}$
- $A \cdot B \neq B \cdot A$

	$\begin{pmatrix} 0.1 & 0.2 \\ 0.3 & 0.4 \\ 0.5 & 0.6 \end{pmatrix}$
$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$	$\begin{pmatrix} 2.2 & 2.8 \\ 4.9 & 6.4 \end{pmatrix}$

**Rechenregeln für die Multiplikation von Matrizen** Assoziativ, Distributiv, nicht Kommutativ!

Transponierte Matrix  $A^{m \times n} \rightarrow (A^T)^{n \times m}$

- $A^T$ : Spalten und Zeilen vertauscht
- $(A^T)_{ij} = A_{ji}$  und  $(A \cdot B)^T = B^T \cdot A^T$

$$\left( \begin{array}{c} \boxed{Z_1 \rightarrow} \\ \boxed{Z_2 \rightarrow} \\ \boxed{Z_3 \rightarrow} \end{array} \right)^T = \left( \begin{array}{ccc} \boxed{\leftarrow Z_1} & \boxed{\leftarrow Z_2} & \boxed{\leftarrow Z_3} \end{array} \right)$$

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)

**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  $\begin{pmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \dots & a_{mn} \end{pmatrix}$   
 $\vec{x}$ : Vektor der Unbekannten  $\begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}$   
 $\vec{b}$ : Vektor der Konstanten  $\begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix}$

**Rang einer Matrix**  $rg(A) = \text{Anzahl Zeilen} - \text{Anzahl Nullzeilen}$   
 $\Rightarrow$  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

- bestimme linkeste Spalte mit Elementen  $\neq 0$  (Pivot-Spalte)
- oberste Zahl in Pivot-Spalte = 0  
 $\rightarrow$  vertausche Zeilen so dass  $a_{11} \neq 0$
- teile erste Zeile durch  $a_{11} \rightarrow$  so erhalten wir führende Eins
- Nullen unterhalb führender Eins erzeugen (Zeilenoperationen)  
nächste Schritte: ohne bereits bearbeitete Zeilen Schritte 1-4 wiederholen, bis Matrix Zeilenstufenform hat

**Zeilenoperationen** 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:  $rg(A) = rg(A|b)$   $\bullet$  unendlich viele Lösungen:
- genau eine Lösung:  $rg(A) = n$   $rg(A) < n$

**Parameterdarstellung** bei unendlich vielen Lösungen

Führende Unbekannte: Spalte mit führender Eins  $\begin{pmatrix} x_1 & x_2 & x_3 & x_4 \\ 1 & -2 & 0 & 3 \\ 0 & 0 & 1 & 1 \end{pmatrix} \begin{array}{c} 5 \\ 3 \end{array}$   
Freie Unbekannte: Spalten ohne führende Eins

Auflösung nach der führenden Unbekannten:

- $1x_1 - 2x_2 + 0x_3 + 3x_4 = 5$   $x_2 = \lambda \rightarrow x_1 = 5 + 2 \cdot \lambda - 3 \cdot \mu$
- $0x_1 + 0x_2 + 1x_3 + 1x_4 = 3$   $x_4 = \mu \rightarrow x_3 = 3 - \mu$

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

**Homogenes LGS**  $\vec{b} = \vec{0} \rightarrow A \cdot \vec{x} = \vec{0} \rightarrow rg(A) = rg(A | \vec{b})$

nur zwei Möglichkeiten:

- eine Lösung  $x_1 = x_2 = \dots = x_n = 0$ , 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  $x = A^{-1} \cdot 0 = 0$

**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 x 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  $A^{n \times n}$

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

$$\underbrace{\begin{pmatrix} 4 & -1 & 0 \\ 0 & 2 & 1 \\ 3 & -5 & -2 \end{pmatrix}}_A \cdot \underbrace{\begin{pmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{pmatrix}}_{A^{-1}} = \underbrace{\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}}_E$$
  
$$\rightarrow \left( \begin{array}{ccc|ccc} 4 & -1 & 0 & 1 & 0 & 0 \\ 0 & 2 & 1 & 0 & 1 & 0 \\ 3 & -5 & -2 & 0 & 0 & 1 \end{array} \right)$$

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:

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

**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:  
 $P = P_1 \cdot \dots \cdot P_l$

**Zeilenvertauschung** für Matrix  $A$  mit Permutationsmatrix  $P_1$ :

$$\underbrace{\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}}_A \cdot \underbrace{\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}}_{P_1} = \begin{pmatrix} 7 & 8 & 9 \\ 4 & 5 & 6 \\ 1 & 2 & 3 \end{pmatrix} \Rightarrow A \cdot P_1 \text{ bewirkt die Vertauschung von Zeile 1 und 3}$$

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

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

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

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:

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

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

- Initialisiere  $P = I_n$
- 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:  $A := P_k A$
- Bei der LR-Zerlegung mit Pivotisierung:
  - $PA = LR$
  - Löse  $Ly = Pb$  und  $Rx = y$

**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

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:

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

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

Schritt 1: Erste Spalte

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

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

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

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

Schritt 2: Zweite Spalte

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

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

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

Endergebnis

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

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

Lösung des Systems

- $Pb = \begin{pmatrix} 3 \\ 5 \\ 0 \end{pmatrix}$
- Löse  $Ly = Pb$  durch Vorwärtseinsetzen:  $y = \begin{pmatrix} 3 \\ 4.4 \\ 2.85 \end{pmatrix}$
- 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

Eine orthogonale Matrix  $Q \in \mathbb{R}^{n \times n}$  erfüllt:  $Q^T Q = Q Q^T = I_n$   
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 ( $H^T = H^{-1}$ ) und symmetrisch ( $H^T = H$ )  
•  $H^2 = I_n$

QR-Zerlegung mit Householder

1. Initialisierung:  $R := A, Q := I_n$
2. Für  $i = 1, \dots, n - 1$ :
  - 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  $Q := Q Q_i^T$

QR-Zerlegung mit Householder

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 1. 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.  $Q^T Q = Q Q^T = I$  (Orthogonalität)
3.  $R$  ist obere Dreiecksmatrix

Wichtige Beobachtungen

- Vorzeichenwahl bei  $v_k$  ist entscheidend für numerische Stabilität
- Ein falsches Vorzeichen kann zu Auslöschung führen
- Betrag der Diagonalelemente in  $R$  = Norm transformierter 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:**  $\|x\|_2 = \sqrt{\sum_{i=1}^n x_i^2}, \|A\|_2 = \sqrt{\rho(A^T A)}$

**$\infty$ -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
- $\text{cond}(A) \rightarrow \infty$ : singular

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

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

Fehlerabschätzung

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

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

**Vergleich Lösungsverfahren**  $A = \begin{pmatrix} 1 & 2 & 0 \\ 2 & 1 & 2 \\ 0 & 2 & 1 \end{pmatrix}, b = \begin{pmatrix} 1 \\ 3 \\ 3 \end{pmatrix}$

- Matrix ist symmetrisch und nicht streng diagonaldominant
- $\text{cond}_\infty(A) \approx 12.5$

Verfahren	Iterationen	Residuum	Zeit
LR mit Pivot	1	$2.2 \cdot 10^{-16}$	1.0
QR	1	$2.2 \cdot 10^{-16}$	2.3
Jacobi	12	$1.0 \cdot 10^{-6}$	1.8
Gauss-Seidel	7	$1.0 \cdot 10^{-6}$	1.4

- Direkte Verfahren erreichen höhere Genauigkeit
- Iterative Verfahren brauchen mehrere Schritte

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

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

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

Komponentenweise:

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

Implementierung von Jacobi- und Gauss-Seidel-Verfahren

Vorbereitungsphase

- Matrix zerlegen in  $A = L + D + R$
- Diagonaldominanz prüfen:  $|a_{ii}| > \sum_{j \neq i} |a_{ij}|$  für alle  $i$
- Sinnvolle Startwerte wählen (z.B.  $x^{(0)} = 0$  oder  $x^{(0)} = b$ )
- Toleranz  $\epsilon$  und max. Iterationszahl  $n_{max}$  festlegen

Verfahren durchführen

- **Jacobi**: Komponentenweise parallel berechnen
- **Gauss-Seidel**: Komponentenweise sequentiell berechnen

Konvergenzprüfung

- Absolute Änderung:  $\|x^{(k+1)} - x^{(k)}\| < \epsilon$
- Relatives Residuum:  $\frac{\|Ax^{(k)} - b\|}{\|b\|} < \epsilon$
- Maximale Iterationszahl:  $k < n_{max}$

A-priori Fehlerabschätzung

- Spektralradius  $\rho$  der Iterationsmatrix bestimmen
- Benötigte Iterationen  $n$  für Genauigkeit  $\epsilon$ :

$n \geq \frac{\ln(\epsilon(1-\rho)/\|x^{(1)} - x^{(0)}\|)}{\ln(\rho)}$

**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

**Konvergenzverhalten**  $\begin{pmatrix} 4 & 1 & 0 \\ 1 & 4 & 1 \\ 0 & 1 & 4 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 3 \\ 3 \end{pmatrix}$

Die Matrix ist diagonaldominant:  $|a_{ii}| = 4 > 1 = \sum_{j \neq i} |a_{ij}|$

k	Residuum		Rel. Fehler	
	Jacobi	G-S	Jacobi	G-S
0	3.74	3.74	-	-
1	0.94	0.47	0.935	0.468
2	0.23	0.06	0.246	0.125
3	0.06	0.01	0.065	0.017
4	0.01	0.001	0.016	0.002

Beobachtungen:

- Gauss-Seidel konvergiert etwa doppelt so schnell wie Jacobi
- Das Residuum fällt linear (geometrische Folge)
- Die Konvergenz ist gleichmäßig (keine Oszillationen)



## Komplexe Zahlen

### Fundamentalsatz der Algebra

Eine algebraische Gleichung n-ten Grades mit komplexen Koeffizienten:

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

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:**  $\operatorname{Re}(z) = x$

**Imaginärteil:**  $\operatorname{Im}(z) = y$

**Betrag:**  $|z| = \sqrt{x^2 + y^2} = \sqrt{z \cdot z^*}$

**Konjugation:**  $\bar{z} = x - iy$



### Darstellungsformen

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

### 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  $x = 0, y < 0$
  - $\varphi$  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

### 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:**

$$\begin{aligned} z_1 \cdot z_2 &= (x_1 x_2 - y_1 y_2) + i(x_1 y_2 + x_2 y_1) \\ &= r_1 r_2 e^{i(\varphi_1 + \varphi_2)} \quad (\text{in Exponentialform}) \end{aligned}$$

**Division:**

$$\begin{aligned} \frac{z_1}{z_2} &= \frac{z_1 \cdot z_2^*}{z_2 \cdot z_2^*} = \frac{(x_1 x_2 + y_1 y_2) + i(y_1 x_2 - x_1 y_2)}{x_2^2 + y_2^2} \\ &= \frac{r_1}{r_2} e^{i(\varphi_1 - \varphi_2)} \quad (\text{in Exponentialform}) \end{aligned}$$

### 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:  $z_k = \sqrt[n]{r} e^{i \frac{\varphi + 2\pi k}{n}}, k = 0, 1, \dots, n-1$

## 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  $x \in \mathbb{C}^n \setminus \{0\}$  gibt mit:

$$Ax = \lambda x$$

Der Vektor  $x$  heißt dann Eigenvektor zum Eigenwert  $\lambda$ .

### Bestimmung von Eigenwerten

Ein Skalar  $\lambda$  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:

$$\det(A) = \prod_{i=1}^n \lambda_i \quad (\text{Produkt der Eigenwerte})$$

$$\operatorname{tr}(A) = \sum_{i=1}^n \lambda_i \quad (\text{Summe der Eigenwerte})$$

- Bei Dreiecksmatrix sind die Diagonalelemente die Eigenwerte
- Ist  $\lambda$  Eigenwert von A, so ist  $\frac{1}{\lambda}$  Eigenwert von  $A^{-1}$

**Vielfachheiten** Für einen Eigenwert  $\lambda$  unterscheidet man:

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

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

### Bestimmung von Eigenwerten und Eigenvektoren

Vorbereitung

- Matrix  $A \in \mathbb{R}^{n \times n}$  aufschreiben
- Charakteristische Matrix  $(A - \lambda I)$  aufstellen

Eigenwerte bestimmen

1. Charakteristisches Polynom aufstellen:
  - Bei  $2 \times 2$  Matrizen direkt:  $\det(A - \lambda I)$
  - Bei  $3 \times 3$  Matrizen: Entwicklung nach einer Zeile/Spalte
  - Bei größeren Matrizen: Spezielle Eigenschaften nutzen (z.B. Dreiecksform, Symmetrie)
2. Polynom vereinfachen und auf Nullform bringen:
  - Ausmultiplizieren
  - Zusammenfassen nach Potenzen von  $\lambda$
  - Form:  $p(\lambda) = (-1)^n \lambda^n + a_{n-1} \lambda^{n-1} + \dots + a_1 \lambda + a_0$
3. Nullstellen bestimmen:
  - Bei quadratischer Gleichung: Mitternachtsformel
  - Bei Grad 3: Substitution oder Cardanische Formeln
  - Bei höherem Grad: Numerische Verfahren

Eigenvektoren bestimmen

1. Für jeden Eigenwert  $\lambda_i$ :
  - Matrix  $(A - \lambda_i I)$  aufstellen
  - Homogenes LGS  $(A - \lambda_i I)x = 0$  lösen
  - Lösungsvektor normieren falls gewünscht
2. Bei mehrfachen Eigenwerten:
  - Basis des Eigenraums bestimmen
  - Linear unabhängige Eigenvektoren finden

Kontrolle

- Für jeden Eigenvektor  $x_i$  prüfen:  $Ax_i = \lambda_i x_i$
- Bei  $2 \times 2$  Matrix:  $\lambda_1 + \lambda_2 = \operatorname{tr}(A)$  und  $\lambda_1 \cdot \lambda_2 = \det(A)$
- Bei  $3 \times 3$  Matrix zusätzlich:  $\sum \lambda_i = \operatorname{tr}(A)$  und  $\prod \lambda_i = \det(A)$
- Bei reellen Matrizen: Komplexe Eigenwerte treten in konjugierten Paaren auf

Spezialfälle beachten

- Bei Dreiecksmatrizen: Eigenwerte sind die Diagonalelemente
- Bei symmetrischen Matrizen: Alle Eigenwerte sind reell
- Bei orthogonalen Matrizen:  $|\lambda_i| = 1$  für alle Eigenwerte
- Bei nilpotenten Matrizen: Alle Eigenwerte sind 0

**Eigenwertberechnung** Gegeben ist die Matrix  $A = \begin{pmatrix} 2 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 1 & 2 \end{pmatrix}$

1. Charakteristisches Polynom aufstellen:

$$\det(A - \lambda I) = \begin{vmatrix} 2-\lambda & 1 & 0 \\ 1 & 2-\lambda & 1 \\ 0 & 1 & 2-\lambda \end{vmatrix}$$

2. Entwicklung nach 1. Zeile:

$$p(\lambda) = (2 - \lambda) \begin{vmatrix} 2-\lambda & 1 \\ 1 & 2-\lambda \end{vmatrix} - 1 \begin{vmatrix} 1 & 1 \\ 1 & 2-\lambda \end{vmatrix}$$

3. Ausrechnen:

$$p(\lambda) = (2 - \lambda)((2 - \lambda)^2 - 1) - ((2 - \lambda) - 1) = -\lambda^3 + 6\lambda^2 - 11\lambda + 6$$

4. Nullstellen bestimmen:  $\lambda_1 = 1, \lambda_2 = 2, \lambda_3 = 3$

5. Eigenvektoren bestimmen für  $\lambda_1 = 1$ :

$$(A - I)x = 0 \text{ führt zu } x_1 = \begin{pmatrix} 1 \\ -2 \\ 1 \end{pmatrix}$$

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  $\lambda$ , so ist  $Tx$  Eigenvektor von  $A$  zum Eigenwert  $\lambda$
- 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.

Von-Mises-Iteration

Von-Mises-Iteration (Vektoriteration)

Für eine diagonalisierbare Matrix  $A$  mit Eigenwerten  $|\lambda_1| > |\lambda_2| \geq \dots \geq |\lambda_n|$  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  $\lambda_1$ .

Von-Mises-Iteration / Vektoriteration

Algorithmus

- 1. Startvektor  $v^{(0)}$  wählen:
  - Zufälligen Vektor oder  $(1, \dots, 1)^T$  wählen
  - Auf Länge 1 normieren:  $\|v^{(0)}\|_2 = 1$
- 2. Für  $k = 0, 1, 2, \dots$  bis zur Konvergenz:
  - Iterationsvektor berechnen:  $w^{(k)} = Av^{(k)}$
  - Normieren:  $v^{(k+1)} = \frac{w^{(k)}}{\|w^{(k)}\|_2}$
  - Eigenwertapproximation (Rayleigh-Quotient):

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

- 3. Abbruchkriterien prüfen:
  - Änderung des Eigenvektors:  $\|v^{(k+1)} - v^{(k)}\| < \varepsilon$
  - Änderung des Eigenwertes:  $|\lambda^{(k+1)} - \lambda^{(k)}| < \varepsilon$
  - Maximale Iterationszahl erreicht

Verifikation

- Prüfen ob  $Av^{(k)} \approx \lambda^{(k)}v^{(k)}$
- Residuum berechnen:  $\|Av^{(k)} - \lambda^{(k)}v^{(k)}\|$
- Orthogonalität zu anderen Eigenvektoren prüfen

**Von-Mises-Iteration** Gegeben sei die Matrix  $A = \begin{pmatrix} 4 & -1 & 1 \\ -1 & 3 & -2 \\ 1 & -2 & 3 \end{pmatrix}$

Mit Startvektor  $v^{(0)} = \frac{1}{\sqrt{3}}(1, 1, 1)^T$ :

- 1. Erste Iteration:
  - $w^{(0)} = Av^{(0)} = \frac{1}{\sqrt{3}}(4, 0, 2)^T$
  - $v^{(1)} = \frac{w^{(0)}}{\|w^{(0)}\|} = \frac{1}{\sqrt{20}}(4, 0, 2)^T$
  - $\lambda^{(1)} = (v^{(0)})^T Av^{(0)} = 3.33$
- 2. Zweite Iteration:
  - $w^{(1)} = Av^{(1)} = \frac{1}{\sqrt{20}}(18, -2, 8)^T$
  - $v^{(2)} = \frac{w^{(1)}}{\|w^{(1)}\|} = \frac{1}{\sqrt{388}}(18, -2, 8)^T$
  - $\lambda^{(2)} = 5.12$

Konvergenz gegen  $\lambda_1 \approx 5.17$  und  $v = (0.89, -0.10, 0.39)^T$

QR-Verfahren

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  $i = 0, 1, 2, \dots$ :
  - 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

Voraussetzungen

- Matrix  $A \in \mathbb{R}^{n \times n}$
- Eigenwerte sollten verschiedene Beträge haben für gute Konvergenz

Algorithmus

- 1. Initialisierung:
  - $A_0 := A$
  - $Q_0 := I_n$
- 2. Für  $k = 0, 1, 2, \dots$  bis zur Konvergenz:
  - QR-Zerlegung von  $A_k$  berechnen:  $A_k = Q_k R_k$
  - Neue Matrix berechnen:  $A_{k+1} = R_k Q_k$
  - Transformationsmatrix aktualisieren:  $P_{k+1} = P_k Q_k$
- 3. Abbruchkriterien prüfen:
  - Subdiagonalelemente nahe Null:  $|a_{i+1,i}| < \varepsilon$
  - Änderung der Diagonalelemente klein
  - Maximale Iterationszahl erreicht

Auswertung

- Eigenwerte: Diagonalelemente von  $A_k$
- Eigenvektoren: Spalten der Matrix  $P_k$
- Bei  $2 \times 2$ -Blöcken: Komplexe Eigenwertpaare

**QR-Verfahren** Gegeben sei die Matrix  $A = \begin{pmatrix} 1 & 2 & 0 \\ 2 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix}$

- 1. Erste Iteration:
  - QR-Zerlegung:  $Q_1 = \begin{pmatrix} 0.45 & 0.89 & 0 \\ 0.89 & -0.45 & 0 \\ 0 & 0 & 1 \end{pmatrix}, R_1 = \begin{pmatrix} 2.24 & 2.24 & 0.45 \\ 0 & -1 & 0.89 \\ 0 & 0 & 1 \end{pmatrix}$
  - $A_1 = R_1 Q_1 = \begin{pmatrix} 2.24 & 0.45 & 0.45 \\ 0.45 & 0.38 & 0.89 \\ 0.45 & 0.89 & 1 \end{pmatrix}$
- 2. Nach Konvergenz:  $A_k \approx \begin{pmatrix} 3 & * & * \\ 0 & 0 & * \\ 0 & 0 & 0 \end{pmatrix}$

Eigenwerte sind also  $\lambda_1 = 3, \lambda_2 = 0, \lambda_3 = 0$

Numerische Aspekte

- 1. Wahl des Startpunkts:
  - Von-Mises: zufälliger normierter Vektor
  - Inverse Iteration: Näherung für  $\mu$  wichtig
  - QR: Matrix vorher auf Hessenberg-Form
- 2. Konvergenzprüfung:
  - Residuum  $\|Ax^{(k)} - \lambda^{(k)}x^{(k)}\|$
  - Änderung in aufeinanderfolgenden Iterationen
  - Subdiagonalelemente bei QR
- 3. Spezialfälle:
  - Mehrfache Eigenwerte
  - Komplexe Eigenwerte/vektoren
  - Schlecht konditionierte Matrizen

Python Implementationen

Hilfsfunktionen

Matrixoperationen

```
1 def matrix_vector_mult(A, v): # Matrix-Vektor Mult.
2     n = len(A)
3     result = [0] * n
4     for i in range(n):
5         result[i] = sum(A[i][j] * v[j] for j in
6                         range(n))
7     return result
8
9 def matrix_mult(A, B): # Matrix-Matrix Multiplikation
10    m, n = len(A), len(B[0])
11    p = len(B)
12    C = [[0.0] * n for _ in range(m)]
13    for i in range(m):
14        for j in range(n):
15            C[i][j] = sum(A[i][k] * B[k][j] for k in
16                          range(p))
17    return C
18
19 def transpose(A): # Matrix transponieren
20     n = len(A)
21     return [[A[j][i] for j in range(n)] for i in
22             range(n)]
23
24 def vector_norm(v): # Euklidische Norm eines Vektors
25     return sum(x*x for x in v) ** 0.5
26
27 def normalize_vector(v): # Vektor auf L. 1 normieren
28     norm = vector_norm(v)
29     return [x/norm for x in v] if norm > 0 else v
30
31 def copy_matrix(A): # Tiefe Kopie einer Matrix
32     return [[A[i][j] for j in range(len(A[0]))] for i
33             in range(len(A))]
```

**is\_diagonally\_dominant** Diagonaldominanz prüfen

```
1 def is_diagonally_dominant(A):
2     n = len(A)
3     for i in range(n):
4         if abs(A[i][i]) <= sum(abs(A[i][j]) for j in
5                               range(n) if j != i):
6             return False
7     return True
```

## convergence\_check Konvergenzkriterien

```
1 def convergence_check(x_new, x_old, f_new, f_old,
2                       tol=1e-6):
3     # Absoluter Fehler im Funktionswert
4     if abs(f_new) < tol:
5         return True, "Funktionswert < tol"
6     # Relative Aenderung der x-Werte
7     if abs(x_new - x_old) < tol * (1 + abs(x_new)):
8         return True, "Relative Aenderung < tol"
9     # Relative Aenderung der Funktionswerte
10    if abs(f_new - f_old) < tol * (1 + abs(f_new)):
11        return True, "Funktionsaenderung < tol"
12    # Divergenzcheck
13    if abs(f_new) > 2 * abs(f_old):
14        return False, "Divergenz detektiert"
15
16    return False, "Noch nicht konvergiert"
```

## error\_estimate Fehlerabschätzung durch Vorzeichenwechsel

```
1 def error_estimate(f, x, eps=1e-5):
2     fx_left = f(x - eps)
3     fx_right = f(x + eps)
4
5     if fx_left * fx_right < 0:
6         return eps # Nullstelle liegt in (x-eps,
7                     # x+eps)
8     return None
```

## Numerische Lösung von Nullstellenproblemen

### root\_finder\_with\_error Nullstellensuche mit Fehlerabschaetzung

```
1 def root_finder_with_error(f, x0, tol=1e-6,
2                             max_iter=100):
3     x_old = x0
4     f_old = f(x_old)
5
6     for i in range(max_iter):
7         # Iterationsschritt (hier Newton als Beispiel)
8         x_new = x_old - f_old/derivative(f, x_old)
9         f_new = f(x_new)
10
11        # Pruefe Konvergenzkriterien
12        converged, reason = convergence_check(
13            x_new, x_old, f_new, f_old, tol)
14
15        if converged:
16            # Schaeetze finalen Fehler
17            error = error_estimate(f, x_new, tol)
18            return {
19                'root': x_new,
20                'iterations': i+1,
21                'error_bound': error,
22                'convergence_reason': reason
23            }
24
25        x_old, f_old = x_new, f_new
26
27    raise ValueError(f"Keine Konvergenz nach
28                    {max_iter} Iterationen")
```

## fixed\_point\_it Fixpunktiteration

```
1 def fixed_point_it(f, x0, tol=1e-6, max_it=100):
2     x = x0
3     for i in range(max_it):
4         x_new = f(x)
5         if abs(x_new - x) < tol:
6             return x_new, i+1
7         x = x_new
8     raise ValueError("Keine Konvergenz")
9
10 # Optimierte Version mit Fehlerschaetzung
11 def fixed_point_it_opt(f, x0, tol=1e-6, max_it=100):
12     x = x0
13     alpha = None # Schaetzung fuer Lipschitz-Konstante
14     for i in range(max_iter):
15         x_new = f(x)
16         dx = abs(x_new - x)
17         # Lipschitz-Konstante schaeetzen
18         if i > 0 and dx > 0:
19             alpha_new = dx / dx_old
20             if alpha is None or alpha_new > alpha:
21                 alpha = alpha_new
22         # A-posteriori Fehlerabschaetzung
23         if alpha is not None and alpha < 1:
24             error = alpha * dx / (1 - alpha)
25             if error < tol:
26                 return x_new, i+1
27         x = x_new
28         dx_old = dx
29    raise ValueError("Keine Konvergenz")
```

### newton Newton-Verfahren mit Konvergenzprüfung

```
1 def newton(f, df, x0, tol=1e-6, max_iter=100):
2     x = x0
3     fx = f(x)
4
5     for i in range(max_iter):
6         dfx = df(x)
7         if abs(dfx) < 1e-10:
8             raise ValueError("Ableitung nahe Null")
9
10        dx = fx/dfx
11        x_new = x - dx
12        fx_new = f(x_new)
13
14        # Konvergenzpruefung
15        converged, reason = convergence_check(
16            x_new, x, fx_new, fx, tol)
17        if converged:
18            return {
19                'root': x_new,
20                'iterations': i+1,
21                'residual': abs(fx_new),
22                'convergence_reason': reason
23            }
24
25        if abs(fx_new) >= abs(fx):
26            raise ValueError("Divergenz detektiert")
27
28        x, fx = x_new, fx_new
29
30    raise ValueError("Keine Konvergenz")
```

## secant Sekantenverfahren mit Konvergenzprüfung

```
1 def secant(f, x0, x1, tol=1e-6, max_iter=100):
2     fx0 = f(x0)
3     fx1 = f(x1)
4
5     # Stelle mit kleinerem f-Wert als x1
6     if abs(fx0) < abs(fx1):
7         x0, x1 = x1, x0
8         fx0, fx1 = fx1, fx0
9
10    for i in range(max_iter):
11        if abs(fx1) < tol:
12            return x1, i+1
13        if fx1 == fx0:
14            raise ValueError("Division durch Null")
15        # Sekanten-Schritt
16        d = fx1 * (x1 - x0)/(fx1 - fx0)
17        x2 = x1 - d
18
19        # Konvergenzpruefungen
20        if abs(d) < tol * (1 + abs(x1)): # Relative
21            Aenderung
22            return {
23                'root': x2,
24                'iterations': i+1,
25                'residual': abs(f(x2))
26            }
27        fx2 = f(x2)
28        if abs(fx2) >= abs(fx1): # Divergenzcheck
29            if i == 0:
30                raise ValueError("Schlechte
31                                Startwerte")
32            return {
33                'root': x1,
34                'iterations': i+1,
35                'residual': abs(fx1)
36            }
37
38        x0, x1 = x1, x2
39        fx0, fx1 = fx1, fx2
40
41    raise ValueError("Keine Konvergenz")
```

### Nullstellenverfahren - Praktisches Vorgehen

1. Voraussetzungen prüfen:
  - Existiert Nullstelle? (z.B. Vorzeichenwechsel)
  - Sind Startwerte geeignet?
  - Ist Funktion ausreichend glatt? (für Newton)
2. Verfahren wählen:
  - Newton: Wenn Ableitung verfügbar und Startwert nahe Lösung
  - Sekanten: Wenn keine Ableitung aber zwei Startwerte nahe Lösung
  - Fixpunkt: Wenn Funktion kontraktiv
3. Implementierung:
  - Konvergenzkriterien definieren
  - Maximale Iterationszahl festlegen
  - Fehlerabschätzung einbauen
  - Divergenzschutz implementieren
4. Auswertung:
  - Konvergenzverhalten prüfen
  - Fehler abschätzen
  - Ergebnis validieren

## root\_finder\_with\_error Nullstellensuche mit Fehlerabschaetzung

```
1 def root_finder_with_error(f, x0, tol=1e-6,
2   max_iter=100):
3     x_old = x0
4     f_old = f(x_old)
5
6     for i in range(max_iter):
7         # Iterationsschritt (hier Newton als Beispiel)
8         x_new = x_old - f_old/derivative(f, x_old)
9         f_new = f(x_new)
10
11        # Pruefe Konvergenzkriterien
12        converged, reason = convergence_criteria(
13            x_new, x_old, f_new, f_old, tol)
14        if converged:
15            # Schaetze finalen Fehler
16            error = error_estimate(f, x_new, tol)
17            return {
18                'root': x_new, 'iterations': i+1,
19                'error_bound': error,
20                'convergence_reason': reason
21            }
22        x_old, f_old = x_new, f_new
23
24    raise ValueError(f"Keine Konvergenz nach
25        {max_iter} Iterationen")
26
27    # Returns: Dictionary mit Ergebnissen
```

## Numerische Lösung linearer Gleichungssysteme

### gauss\_elimination Gauss-Elimination mit Spaltenpivotisierung

```
1 def gauss_elimination(A, b, tol=1e-10):
2     n = len(A)
3     M = copy_matrix(A) # Tiefe Kopie von A und b
4     x = [0] * n
5     b = b.copy()
6
7     # Vorwaertselimination mit Pivotisierung
8     for i in range(n):
9         # Pivotisierung
10        pivot_row = i
11        for j in range(i+1, n):
12            if abs(M[j][i]) > abs(M[pivot_row][i]):
13                pivot_row = j
14        if pivot_row != i:
15            M[i], M[pivot_row] = M[pivot_row], M[i]
16            b[i], b[pivot_row] = b[pivot_row], b[i]
17        # Pruefe auf singulaere Matrix
18        if abs(M[i][i]) < tol:
19            raise ValueError("Matrix (fast) singulaer")
20
21        # Elimination
22        for j in range(i+1, n):
23            factor = M[j][i] / M[i][i]
24            for k in range(i, n):
25                M[j][k] -= factor * M[i][k]
26            b[j] -= factor * b[i]
27
28        # Rueckwaertssubstitution
29        for i in range(n-1, -1, -1):
30            x[i] = (b[i] - sum(M[i][j] * x[j]
31                            for j in range(i+1, n))) / M[i][i]
32
33    return {
34        'solution': x, 'matrix': M,
35        'condition': estimate_condition(A)
```

## lr\_decomposition LR-Zerlegung mit Zeilenvertauschung

```
1 def lr_decomposition(A, tol=1e-10):
2     n = len(A)
3
4     # Initialisiere L, R und P
5     R = copy_matrix(A)
6     L = [[1.0 if i == j else 0.0 for j in range(n)]
7           for i in range(n)]
8     P = [[1.0 if i == j else 0.0 for j in range(n)]
9           for i in range(n)]
10
11    for k in range(n-1):
12        # Pivotisierung
13        pivot = k
14        for i in range(k+1, n):
15            if abs(R[i][k]) > abs(R[pivot][k]):
16                pivot = i
17
18        if abs(R[pivot][k]) < tol:
19            raise ValueError("Matrix (fast) singulaer")
20
21        # Zeilenvertauschung falls noetig
22        if pivot != k:
23            R[k], R[pivot] = R[pivot], R[k]
24
25        # L und P anpassen fuer Zeilen < k
26        for j in range(k):
27            L[k][j], L[pivot][j] = L[pivot][j],
28            L[k][j]
29        P[k], P[pivot] = P[pivot], P[k]
30
31    # Elimination
32    for i in range(k+1, n):
33        factor = R[i][k] / R[k][k]
34        L[i][k] = factor
35        for j in range(k, n):
36            R[i][j] -= factor * R[k][j]
37
38    return {
39        'L': L,
40        'R': R,
41        'P': P
```

### solve\_lr LGS mit LR-Zerlegung lösen

```
1 def solve_lr(L, R, P, b):
2     n = len(L)
3     # Pb berechnen
4     pb = matrix_vector_mult(P, b)
5
6     # Vorwaertseinsetzen Ly = Pb
7     y = [0] * n
8     for i in range(n):
9         y[i] = pb[i] - sum(L[i][j] * y[j] for j in
10                            range(i))
11
12    # Rueckwaertseinsetzen Rx = y
13    x = [0] * n
14    for i in range(n-1, -1, -1):
15        x[i] = (y[i] - sum(R[i][j] * x[j] for j in
16                        range(i+1, n))) / R[i][i]
17
18    return x
```

## householder\_vector Householder-Vektor zu x berechnen

```
1 def householder_vector(x):
2     n = len(x)
3     v = x.copy()
4     sigma = sum(v[i]*v[i] for i in range(1, n))
5
6     if sigma == 0 and x[0] >= 0:
7         return [0] * n, 0
8     elif sigma == 0 and x[0] < 0:
9         return [2] + [0]*(n-1), -2
10    mu = (x[0]*x[0] + sigma)**0.5
11    if x[0] <= 0:
12        v[0] = x[0] - mu
13    else:
14        v[0] = -sigma/(x[0] + mu)
15    beta = 2*v[0]*v[0]/(sigma + v[0]*v[0])
16    return normalize_vector(v), beta
```

### qr\_decomposition QR-Zerlegung mittels Householder-Transformationen

```
1 def qr_decomposition(A):
2     m = len(A)
3     n = len(A[0])
4     R = copy_matrix(A)
5     Q = [[1.0 if i == j else 0.0 for j in range(m)]
6           for i in range(m)]
7
8     for k in range(min(m-1, n)):
9         # Extrahiere k-te Spalte ab k-ter Zeile
10        x = [R[i][k] for i in range(k, m)]
11        # Berechne Householder-Transformation
12        v, beta = householder_vector(x)
13
14        # Wende Householder auf R an
15        for j in range(k, n):
16            # w = beta * (v^T * R_{kj})
17            w = beta * sum(v[i-k]*R[i][j]
18                          for i in range(k, m))
19            R[i][j] -= v[i-k] * w
20        for j in range(m): # Update Q
21            w = beta * sum(v[i-k]*Q[j][i+k]
22                          for i in range(len(v)))
23            Q[j][k+i] -= v[i] * w
24
25    Q = transpose(Q) # Transponiere Q am Ende
26    return {
27        'Q': Q, 'R': R
28    }
```

### solve\_qr Lösen von QRx = b

```
1 def solve_qr(Q, R, b):
2     # Berechne Q^T * b
3     y = matrix_vector_mult(transpose(Q), b)
4     # Rueckwaertseinsetzen
5     n = len(R)
6     x = [0] * n
7     for i in range(n-1, -1, -1):
8         x[i] = (y[i] - sum(R[i][j] * x[j] for j in
9                        range(i+1, n))) / R[i][i]
10
11    return x
```



## Iterative Löser für lineare Gleichungssysteme

### `jacobi_method` Jacobi-Verfahren

```
1 def jacobi_method(A, b, tol=1e-6, max_iter=100):
2     n = len(A)
3     # Prüfe Diagonaldominanz
4     if not is_diagonally_dominant(A):
5         print("Warnung: Matrix nicht diagonaldominant")
6     # Initialisiere mit Nullvektor
7     x = [0.0] * n
8     iterations = []
9     residuals = []
10    for iter in range(max_iter):
11        x_new = [0.0] * n
12        # Jacobi-Iteration
13        for i in range(n):
14            sum_term = sum(A[i][j] * x[j]
15                           for j in range(n) if j != i)
16            x_new[i] = (b[i] - sum_term) / A[i][i]
17        # Berechne Residuum
18        res = max(abs(sum(A[i][j] * x_new[j]
19                          for j in range(n)) - b[i])
20                  for i in range(n))
21        # Konvergenzcheck
22        diff = max(abs(x_new[i] - x[i]) for i in
23                   range(n))
24        iterations.append(x_new.copy())
25        residuals.append(res)
26        if diff < tol:
27            return {
28                'solution': x_new,
29                'iterations': iterations,
30                'residuals': residuals,
31                'iteration_count': iter + 1
32            }
33        x = x_new.copy()
34    raise ValueError("Keine Konvergenz nach
35                     {max_iter}, Iterationen\nLetztes Residuum:
36                     {res}")
```

### `gauss_seidel_method` Gauss-Seidel-Verfahren

```
1 def gauss_seidel_method(A, b, tol=1e-6, max_iter=100):
2     n = len(A)
3     iterations = []
4     residuals = []
5
6     # Prüfe Diagonaldominanz
7     if not is_diagonally_dominant(A):
8         print("Warnung: Matrix nicht diagonaldominant")
9     x = [0.0] * n
10    # Gauss-Seidel-Iteration
11    for iter in range(max_iter):
12        x_old = x.copy()
13        for i in range(n):
14            sum1 = sum(A[i][j] * x[j] for j in
15                      range(i))
16            sum2 = sum(A[i][j] * x_old[j]
17                       for j in range(i+1, n))
18            x[i] = (b[i] - sum1 - sum2) / A[i][i]
19        # Berechne Residuum und relative Änderung
20        res = max(abs(sum(A[i][j] * x[j]
21                          for j in range(n)) - b[i])
22                  for i in range(n))
23        diff = max(abs(x[i] - x_old[i]) for i in
24                   range(n))
```

```
23
24        iterations.append(x.copy())
25        residuals.append(res)
26        if diff < tol:
27            return {
28                'solution': x,
29                'iterations': iterations,
30                'residuals': residuals,
31                'iteration_count': iter + 1
32            }
33
34        raise ValueError(f"Keine Konvergenz nach
35                          {max_iter} "
36                          f"Iterationen\nLetztes Residuum:
37                          {res}")
```

### `analyze_convergence` Konvergenzanalyse

```
1 def analyze_convergence(method_name, iterations,
2                           residuals):
3     """Analysiert Konvergenzverhalten eines iterativen
4     Verfahrens"""
5     n = len(residuals)
6     if n < 2:
7         return {
8             'method': method_name,
9             'converged': False,
10            'reason': 'Zu wenige Iterationen'
11        }
12
13    # Schätze Konvergenzrate
14    rates = [abs(residuals[i]/residuals[i-1])
15             for i in range(1, n)]
16    avg_rate = sum(rates) / len(rates)
17
18    # Schätze asymptotische Konvergenzrate
19    asymp_rate = rates[-1] if rates else None
20
21    # Berechne empirische Konvergenzordnung
22    if n > 2:
23        q = [abs(log(residuals[i+1]/residuals[i]) /
24                    log(residuals[i]/residuals[i-1]))
25             for i in range(n-2)]
26    avg_order = sum(q) / len(q) if q else None
27    else:
28        avg_order = None
29
30    return {
31        'method': method_name,
32        'converged': residuals[-1] < residuals[0],
33        'iterations': n,
34        'final_residual': residuals[-1],
35        'avg_rate': avg_rate,
36        'asymp_rate': asymp_rate,
37        'conv_order': avg_order
38    }
```

### Implementation iterativer Verfahren

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

## Eigenwerte und Eigenvektoren

### `complex_operations` Komplexe Zahlen

```
1 def complex_operations(z1, z2):
2     """Grundlegende Operationen mit komplexen
3     Zahlen."""
4     # Basisfunktionen
5     def to_polar(z):
6         r = (z.real**2 + z.imag**2)**0.5
7         phi = math.atan2(z.imag, z.real)
8         return r, phi
9
10    def from_polar(r, phi):
11        return r * (math.cos(phi) + 1j*math.sin(phi))
12
13    try:
14        # Addition und Subtraktion
15        z_add = z1 + z2
16        z_sub = z1 - z2
17        # Multiplikation und Division
18        z_mul = z1 * z2
19        z_div = z1 / z2 if z2 != 0 else None
20        # Polarform
21        r1, phi1 = to_polar(z1)
22        r2, phi2 = to_polar(z2)
23        # Exponentialform
24        z1_exp = from_polar(r1, phi1)
25        z2_exp = from_polar(r2, phi2)
26
27        return {
28            'addition': z_add,
29            'subtraktion': z_sub,
30            'multiplikation': z_mul,
31            'division': z_div,
32            'polar_z1': (r1, phi1),
33            'polar_z2': (r2, phi2)
34        }
35
36    except Exception as e:
37        print(f"Fehler bei Berechnung: {e}")
38        return None
```

### Determinante

```
1 def det_2x2(matrix):
2     return matrix[0][0]*matrix[1][1] -
3           matrix[0][1]*matrix[1][0]
4
5 def det_3x3(matrix):
6     det = 0
7     # Entwicklung nach erster Zeile
8     for i in range(3):
9         minor = []
10        for j in range(1,3):
11            row = []
12            for k in range(3):
13                if k != i:
14                    row.append(matrix[j][k])
15            minor.append(row)
16        det += ((-1)**i) * matrix[0][i] *
17              det_2x2(minor)
18    return det
```

### characteristic\_polynomial

Charakteristisches Polynom einer 2x2 oder 3x3 Matrix

```
1 def characteristic_polynomial(A):
2     n = len(A)
3     if n == 2:
4         a, d = A[0][0], A[1][1]
5         # det(A - lambda*I) = lambda^2 - tr(A)*lambda
6         # + det(A)
7         return [1, -(a + d), det_2x2(A)]
8     elif n == 3:
9         # Hier nur Koeffizienten, keine
10        # Polynomauswertung
11        trace = sum(A[i][i] for i in range(3))
12        det = det_3x3(A)
13        return [1, -trace, None, det] # Mittlerer
14        # Koeff. kompliziert
15    else:
16        raise ValueError("Nur fuer 2x2 oder 3x3
17        Matrizen")
```

### find\_eigenvalues\_2x2

 Eigenwerte einer 2x2 Matrix

```
1 def find_eigenvalues_2x2(A, tol=1e-10):
2     coeff = characteristic_polynomial(A)
3     # Quadratische Formel
4     p, q = -coeff[1], coeff[2]
5     disc = p*p/4 - q
6     if abs(disc) < tol: # Doppelte Eigenwerte
7         return [-p/2, -p/2]
8     elif disc > 0: # Reelle Eigenwerte
9         root = disc**0.5
10        return [-p/2 + root, -p/2 - root]
11    else: # Komplexe Eigenwerte
12        root = (-disc)**0.5
13        return [-p/2 + 1j*root, -p/2 - 1j*root]
```

### find\_eigenvector

 Eigenvektor zu gegebenem Eigenwert

```
1 def find_eigenvector(A, eigenval, tol=1e-10):
2     n = len(A)
3     # A - lambda*I
4     M = [[A[i][j] - (eigenval if i==j else 0)
5           for j in range(n)] for i in range(n)]
6
7     # Loese homogenes System (A - lambda*I)x = 0
8     # Nehme eine Komponente als 1 an
9     for i in range(n):
10        if abs(M[i][i]) > tol:
11            vec = [0] * n
12            vec[i] = 1
13            for j in range(n):
14                if j != i:
15                    s = sum(-M[j][k]*vec[k] for k in
16                          range(n) if k != j)
17                    if abs(M[j][j]) > tol:
18                        vec[j] = s/M[j][j]
19            return normalize_vector(vec)
20
21    raise ValueError("Kein Eigenvektor gefunden")
```

### power\_iteration

 Von-Mises-Iteration für grössten Eigenwert

```
1 def power_iteration(A, tol=1e-10, max_iter=100):
2     n = len(A)
3     # Starte mit Einheitsvektor
4     v = normalize_vector([1] + [0]*(n-1))
5     lambda_old = 0
6
7     for _ in range(max_iter):
8         # Matrix-Vektor-Multiplikation
9         w = matrix_vector_mult(A, v)
10        # Normiere neuen Vektor
11        v = normalize_vector(w)
12
13        # Rayleigh-Quotient
14        lambda_k = sum(v[i] * A[i][j] * v[j]
15                      for i in range(n)
16                      for j in range(n))
17
18        if abs(lambda_k - lambda_old) < tol:
19            return {
20                'eigenvalue': lambda_k,
21                'eigenvector': v
22            }
23
24        lambda_old = lambda_k
25
26    raise ValueError("Keine Konvergenz")
```

### inverse\_iteration

 Inverse Iteration für Eigenwert nahe  $\mu$ 

```
1 def inverse_iteration(A, mu, tol=1e-10, max_iter=100):
2     n = len(A)
3     # Starte mit Einheitsvektor
4     v = normalize_vector([1] + [0]*(n-1))
5     lambda_old = 0
6
7     # (A - mu*I) berechnen
8     M = [[A[i][j] - (mu if i==j else 0)
9           for j in range(n)] for i in range(n)]
10
11    for _ in range(max_iter):
12        # Loese (A - mu*I)w = v
13        w = gauss_elimination(M, v)['solution']
14        # Normiere neuen Vektor
15        v = normalize_vector(w)
16
17        # Rayleigh-Quotient
18        lambda_k = sum(v[i] * A[i][j] * v[j]
19                      for i in range(n)
20                      for j in range(n))
21
22        if abs(lambda_k - lambda_old) < tol:
23            return {
24                'eigenvalue': lambda_k,
25                'eigenvector': v
26            }
27
28        lambda_old = lambda_k
29
30    raise ValueError("Keine Konvergenz")
```

### qr\_algorithm

 QR-Algorithmus für alle Eigenwerte

```
1 def qr_algorithm(A, tol=1e-10, max_iter=100):
2     n = len(A)
3     A_k = copy_matrix(A)
4
5     for k in range(max_iter):
6         # QR-Zerlegung
7         qr = qr_decomposition(A_k)
8         Q, R = qr['Q'], qr['R']
9
10        # Neue Iteration A_(k+1) = RQ
11        A_k = matrix_mult(R, Q)
12
13        # Pruefe ob Diagonalelemente konvergiert sind
14        if all(abs(A_k[i][j]) < tol
15              for i in range(1, n)
16              for j in range(i)):
17            return {
18                'eigenvalues': [A_k[i][i] for i in
19                              range(n)],
20                'iterations': k+1,
21                'final_matrix': A_k
22            }
23
24    raise ValueError("Keine Konvergenz")
25
26 def deflation(A, eigenval, eigenvector):
27     """Deflation nach Hotelling"""
28     n = len(A)
29     v = normalize_vector(eigenvector)
30
31     # Berechne A - lambda*vv^T
32     vvt = [[v[i]*v[j] for j in range(n)] for i in
33            range(n)]
34     return [[A[i][j] - eigenval*vvt[i][j]
35            for j in range(n)] for i in range(n)]
```