

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:

$$\underbrace{0.3211}_{n=4} \cdot \underbrace{4^{12}}_{l=2}$$
 Exponent:  $\hat{e} = 1 \cdot 4^1 + 2 \cdot 4^0 = 6$   
Wert:  $\hat{\omega} = 3 \cdot 4^3 + 2 \cdot 4^2 + 1 \cdot 4^1 + 1 \cdot 4^0 = 3664$

**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>Abstossender 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:

<ul style="list-style-type: none"><li><math>F</math> hat genau einen Fixpunkt <math>\tilde{x}</math> in <math>[a, b]</math></li><li>Die Fixpunktiteration konvergiert gegen <math>\tilde{x}</math> für alle <math>x_0 \in [a, b]</math></li></ul>	<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} $
---	---

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 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  $x_0 = 1$

- $x_1 = 1 - \frac{1^2-2}{2 \cdot 1} = 1.5$  → 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:  
Konvergiert langsamer, aber benötigt weniger Rechenaufwand.

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

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  $x_1 = 2$

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

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:  $x_3 = 1.4142157$  **Also:**  $|x_3 - \sqrt{2}| < 10^{-5}$
- Mit  $\epsilon = 10^{-5}$ :
- $f(x_3 - \epsilon) = 1.4142057^2 - 2 < 0$  → 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.

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

				0.1	0.2
				0.3	0.4
				0.5	0.6
1	2	3		2.2	2.8
4	5	6		4.9	6.4

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

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

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**  $rg(A) = \text{Anzahl Zeilen} - \text{Anzahl Nullzeilen}$   
⇒ 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 → 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  
→ vertausche Zeilen so dass  $a_{11} \neq 0$
- teile erste Zeile durch  $a_{11}$  → 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:  $rg(A) = rg(A|b)$  • 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. Rückwärtseinsetzen:**  $x_i = \frac{b_i - \sum_{j=i+1}^n a_{ij} x_j}{a_{ii}}$ ,  $i = n, n - 1, \dots, 1$

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

**Eliminationsschritte:**  $\begin{pmatrix} 2 & 4 & -2 & | & 2 \\ 0 & 3 & 15 & | & 36 \\ 0 & 1 & 1 & | & 4 \end{pmatrix} \Rightarrow \begin{pmatrix} 2 & 4 & -2 & | & 2 \\ 0 & 3 & 15 & | & 36 \\ 0 & 0 & -2 & | & -8 \end{pmatrix}$

**Rückwärtseinsetzen:**  
 $x_3 = \frac{-8}{-2} = 4$   
 $x_2 = \frac{36 - 15(4)}{3} = 1$   
 $x_1 = \frac{2 - 4(4) + 2}{2} = -6$

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  $j > i$   
**Obere Dreiecksmatrix R:**  $r_{ij} = 0$  für  $i > j$   
Diagonale normiert ( $l_{ii} = 1$ )  
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:  
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**  $A = \begin{pmatrix} -1 & 1 & 1 \\ 1 & -3 & -2 \\ 5 & 1 & 4 \end{pmatrix}$ ,  $b = \begin{pmatrix} 0 \\ 5 \\ 3 \end{pmatrix}$

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

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:

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

Lösung des Systems

- $Pb = \begin{pmatrix} 3 \\ 5 \\ 0 \end{pmatrix}$
- Löse  $Ly = Pb$  durch Vorwärtseinsetzen:  $y = \begin{pmatrix} 3 \\ 2.85 \\ \frac{4.4}{2.85} \end{pmatrix}$
- 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  $i$
  - $w_i := v_i + \text{sign}(v_{i1})\|v_i\|e_1$
  - $u_i := w_i/\|w_i\|$
  - $H_i := I_{n-i+1} - 2u_iu_i^T$
  - 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:  $H_1 = I - 2u_1u_1^T = \begin{pmatrix} -0.67 & -0.75 & 0 \\ -0.75 & 0.67 & 0 \\ 0 & 0 & 1 \end{pmatrix}$

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:**  $z_1 \cdot z_2 = (x_1 x_2 - y_1 y_2) + i(x_1 y_2 + x_2 y_1)$

$$= r_1 r_2 e^{i(\varphi_1 + \varphi_2)} \text{ (in Exponentialform)}$$

**Division:**

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

### 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:

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

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

- 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:

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

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 range(n))
6     return result
7
8 def matrix_mult(A, B): # Matrix-Matrix Multiplikation
9     m, n = len(A), len(B[0])
10    p = len(B)
11    C = [[0.0] * n for _ in range(m)]
12    for i in range(m):
13        for j in range(n):
14            C[i][j] = sum(A[i][k] * B[k][j] for k in range(p))
15    return C
16
17 def transpose(A): # Matrix transponieren
18     n = len(A)
19     return [[A[j][i] for j in range(n)] for i in range(n)]
20
21 def vector_norm(v): # Euklidische Norm eines Vektors
22     return sum(x*x for x in v) ** 0.5
23
24 def normalize_vector(v): # Vektor auf L. 1 normieren
25     norm = vector_norm(v)
26     return [x/norm for x in v] if norm > 0 else v
27
28 def copy_matrix(A): # Tiefe Kopie einer Matrix
29     return [[A[i][j] for j in range(len(A[0]))] for i 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 range(n) if j != i):
5             return False
6     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
15        if converged:
16            # Schaetze 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")
29
30    # 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     # Vorwaartselimination 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
15        if pivot_row != i:
16            M[i], M[pivot_row] = M[pivot_row], M[i]
17            b[i], b[pivot_row] = b[pivot_row], b[i]
18
19        # Pruefe auf singulaere Matrix
20        if abs(M[i][i]) < tol:
21            raise ValueError("Matrix (fast) singulaer")
22
23        # Elimination
24        for j in range(i+1, n):
25            factor = M[j][i] / M[i][i]
26            for k in range(i, n):
27                M[j][k] -= factor * M[i][k]
28            b[j] -= factor * b[i]
29
30    # Rueckwaertssubstitution
31    for i in range(n-1, -1, -1):
32        x[i] = (b[i] - sum(M[i][j] * x[j]
33            for j in range(i+1, n))) / M[i][i]
34
35    return {
36        'solution': x,
37        'matrix': M,
38        'condition': estimate_condition(A)
39    }
```

## Gauss-Elimination mit Pivotisierung

```
1 def gauss_elimination(A, b):
2     n = len(A)
3     # Erweiterte Matrix erstellen
4     M = [[A[i][j] for j in range(n)] + [b[i]] for i in
5         range(n)]
6
7     # Vorwaartselimination
8     for i in range(n):
9         pivot = M[i][i]
10        if abs(pivot) < 1e-10:
11            continue
12
13        for j in range(i+1, n):
14            factor = M[j][i] / pivot
15            for k in range(i, n):
16                M[j][k] -= factor * M[i][k]
17
18    # Rueckwaertssubstitution
19    x = [0] * n
20    for i in range(n-1, -1, -1):
21        if abs(M[i][i]) < 1e-10:
22            x[i] = 1 # Freie Variable
23            continue
24
25        sum_val = sum(M[i][j] * x[j] for j in
26            range(i+1, n))
27        x[i] = (M[i][n] - sum_val) / M[i][i]
28
29    return x
```



## lr\_decomposition

## LR-Zerlegung mit Zeilenvertauschung

```

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

```

**solve\_lr** 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]
16                             for j in range(i+1, n))) / R[i][i]
17
18     return x

```

## LR-Zerlegung Implementation

```

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

```

## QR-Zerlegung

```

1 def householder_vector(x):
2     """Berechnet Householder-Vektor zu x"""
3     n = len(x)
4     v = x.copy()
5     sigma = sum(v[i]*v[i] for i in range(1, n))
6
7     if sigma == 0 and x[0] >= 0:
8         return [0] * n, 0
9     elif sigma == 0 and x[0] < 0:
10        return [2] + [0]*(n-1), -2
11
12    mu = (x[0]*x[0] + sigma)**0.5
13    if x[0] <= 0:
14        v[0] = x[0] - mu
15    else:
16        v[0] = -sigma/(x[0] + mu)
17
18    beta = 2*v[0]*v[0]/(sigma + v[0]*v[0])
19    return normalize_vector(v), beta
20
21 def qr_decomposition(A):
22     """QR-Zerlegung mittels
23        Householder-Transformationen"""
24     m = len(A)
25     n = len(A[0])
26     R = copy_matrix(A)
27     Q = [[1.0 if i == j else 0.0 for j in range(m)]
28          for i in range(m)]
29
30     for k in range(min(m-1, n)):
31         # Extrahiere k-te Spalte ab k-ter Zeile
32         x = [R[i][k] for i in range(k, m)]
33
34         # Berechne Householder-Transformation
35         v, beta = householder_vector(x)
36
37         # Wende Householder auf R an
38         for j in range(k, n):
39             # w = beta * (v^T * R_{:,j})
40             w = beta * sum(v[i-k]*R[i][j]
41                           for i in range(k, m))
42
43             # Update R
44             for i in range(k, m):
45                 R[i][j] -= v[i-k] * w
46
47         # Update Q
48         for j in range(m):
49             w = beta * sum(v[i-k]*Q[j][i+k]
50                           for i in range(len(v)))
51             for i in range(len(v)):
52                 Q[j][k+i] -= v[i] * w
53
54     # Transponiere Q am Ende
55     Q = transpose(Q)
56
57     return {
58         'Q': Q,
59         'R': R
60     }
61
62 def solve_qr(Q, R, b):
63     """Löst QRx = b"""
64     # Berechne Q^T * b
65     y = matrix_vector_mult(transpose(Q), b)
66
67     # Rueckwaertseinsetzen
68     n = len(R)
69     x = [0] * n
70     for i in range(n-1, -1, -1):
71         x[i] = (y[i] - sum(R[i][j] * x[j]
72

```

## QR-Zerlegung Implementation

```

1 def qr_decomposition(A):
2     m = len(A)
3     n = len(A[0])
4     # Kopiere A nach R (deep copy)
5     R = [[A[i][j] for j in range(n)] for i in range(m)]
6     # Initialisiere Q als Einheitsmatrix
7     Q = [[1.0 if i == j else 0.0 for j in range(m)]
8           for i in range(m)]
9
10    def vector_norm(v): # Norm eines Vektors
11        return (sum(x*x for x in v)) ** 0.5
12
13    def matrix_mult(A, B): # Matrixmultiplikation
14        m, n = len(A), len(B[0])
15        p = len(B)
16        C = [[0.0] * n for _ in range(m)]
17
18        for i in range(m):
19            for j in range(n):
20                C[i][j] = sum(A[i][k] * B[k][j]
21                              for k in range(p))
22
23        return C
24
25    def householder_reflection(x):
26        n = len(x)
27        v = [xi for xi in x] # Kopiere x
28
29        # Berechne Norm des Teilvektors
30        sigma = sum(v[i]*v[i] for i in range(1, n))
31        if sigma == 0 and x[0] >= 0:
32            beta = 0
33        elif sigma == 0 and x[0] < 0:
34            beta = -2
35        else:
36            mu = (x[0]*x[0] + sigma)**0.5
37            if x[0] <= 0:
38                v[0] = x[0] - mu
39            else:
40                v[0] = -sigma/(x[0] + mu)
41
42        beta = 2*v[0]*v[0]/(sigma + v[0]*v[0])
43
44        # Normiere v
45        temp = v[0]
46        for i in range(n):
47            v[i] /= temp
48
49        return v, beta

```

## QR-Zerlegung Implementation (Fortsetzung)

```

1 # Hauptschleife der QR-Zerlegung
2 for k in range(n):
3     # Extrahiere k-te Spalte ab k-ter Zeile
4     x = [R[i][k] for i in range(k, m)]
5     if len(x) > 1: # wenn noch Untermatrix
6         # Berechne Householder-Transformation
7         v, beta = householder_reflection(x)
8         # Wende Householder auf R an
9         for j in range(k, n):
10            # Berechne w = beta * (v^T * R_j)
11            w = beta * sum(v[i-k]*R[i][j]
12                          for i in range(k, m))
13
14            # Update R
15            for i in range(k, m):
16                R[i][j] -= v[i-k] * w
17
18            # Update Q
19            for j in range(m):
20                w = beta * sum(v[i-k]*Q[j][i+k]
21                              for i in range(len(v)))
22                for i in range(len(v)):
23                    Q[j][k+i] -= v[i] * w
24
25        # Transponiere Q am Ende
26        Q = [[Q[j][i] for j in range(m)] for i in range(m)]
27
28    return Q, R # Q (orthogonal) und R (obere
29               # Dreiecksmatrix)
30
31 # Beispiel fuer Verwendung
32 def solve_qr(A, b): # Loest Ax = b mittels QR-Zerlegung
33     Q, R = qr_decomposition(A)
34     # Berechne Q^T * b
35     y = [sum(Q[i][j] * b[j]
36             for j in range(len(b)))
37          for i in range(len(b))]
38
39     # Rueckwaertseinsetzen
40     n = len(R)
41     x = [0] * n
42     for i in range(n-1, -1, -1):
43         s = sum(R[i][j] * x[j] for j in range(i+1, n))
44         if abs(R[i][i]) < 1e-10:
45             raise ValueError("Matrix (fast) singular")
46         x[i] = (y[i] - s) / R[i][i]
47
48     return x

```

## QR-Zerlegung - Praktisches Vorgehen

1. Vorbereitungen
  - Matrix A kopieren für R
  - Q als Einheitsmatrix initialisieren
  - Householder-Vektoren speichern
2. Für jede Spalte  $k = 1, \dots, n-1$ :
  - Untervektor  $v_k$  aus k-ter Spalte extrahieren
  - Householder-Vektor berechnen:
    - $w_k = v_k + \text{sign}(v_{k1}) \|v_k\| e_1$
    - $u_k = \frac{w_k}{\|w_k\|}$
  - Householder-Matrix auf Untermatrix anwenden:
    - $H_k = I - 2u_k u_k^T$
    - $R_{k:n, k:n} = H_k \cdot R_{k:n, k:n}$
  - Q aktualisieren:  $Q = Q \cdot H_k^T$
3. System lösen durch
  - $y = Q^T b$  berechnen
  - Rückwärtseinsetzen:  $Rx = y$

## Zerlegungsverfahren - Vergleich und Anwendung

1. LR-Zerlegung
  - Vorteil: Effizient für mehrere rechte Seiten
  - Nachteil: Numerisch weniger stabil
  - Aufwand:  $\frac{2}{3}n^3$  Operationen
2. QR-Zerlegung
  - Vorteil: Numerisch stabiler
  - Nachteil: Höherer Rechenaufwand
  - Aufwand:  $\frac{4}{3}n^3$  Operationen
3. Wann welches Verfahren?
  - LR: Für gut konditionierte Systeme
  - QR: Für schlecht konditionierte Systeme
  - Gauss: Wenn nur eine Lösung benötigt wird

## Iterative Löser für lineare Gleichungssysteme

### Analyse der Genauigkeit einer Näherungslösung

```

1 def error_analysis(A, x, b, x_approx):
2     n = len(A)
3     # Residuum berechnen
4     r = [b[i] - sum(A[i][j] * x_approx[j]
5                   for j in range(n))
6          for i in range(n)]
7     res_norm = max(abs(ri) for ri in r)
8     # Relativer Fehler (falls exakte Loesung bekannt)
9     if x is not None:
10        rel_error = max(abs(x[i] - x_approx[i])
11                       for i in range(n)) / \
12                      max(abs(x[i]) for i in range(n))
13    else:
14        rel_error = None
15    # Absoluter Fehler (falls exakte Loesung bekannt)
16    if x is not None:
17        abs_error = max(abs(x[i] - x_approx[i])
18                       for i in range(n))
19    else:
20        abs_error = None
21
22    return {
23        'residual_norm': res_norm,
24        'relative_error': rel_error,
25        'residual': r
26    }

```

## Jacobi-Verfahren

```
1 def jacobi_method(A, b, tol=1e-6, max_iter=100):
2     """Jacobi-Iterationsverfahren"""
3     n = len(A)
4
5     # Pruefe Diagonaldominanz
6     if not is_diagonally_dominant(A):
7         print("Warnung: Matrix nicht diagonaldominant")
8
9     # Initialisiere mit Nullvektor
10    x = [0.0] * n
11    iterations = []
12    residuals = []
13
14    for iter in range(max_iter):
15        x_new = [0.0] * n
16        # Jacobi-Iteration
17        for i in range(n):
18            sum_term = sum(A[i][j] * x[j]
19                           for j in range(n) if j != i)
20            x_new[i] = (b[i] - sum_term) / A[i][i]
21
22        # Berechne Residuum
23        res = max(abs(sum(A[i][j] * x_new[j]
24                           for j in range(n)) - b[i])
25                  for i in range(n))
26
27        # Konvergenzcheck
28        diff = max(abs(x_new[i] - x[i]) for i in
29                     range(n))
30
31        iterations.append(x_new.copy())
32        residuals.append(res)
33
34        if diff < tol:
35            return {
36                'solution': x_new,
37                'iterations': iterations,
38                'residuals': residuals,
39                'iteration_count': iter + 1
40            }
41
42        x = x_new.copy()
43
44    raise ValueError(f"Keine Konvergenz nach
45                     {max_iter} Iterationen nach
46                     f\"Iterationen\\nLetztes Residuum:
47                     {res}\"")
```

## Jacobi-Verfahren Implementation

```
1 def jacobi_method(A, b, x0, tol=1e-6, max_iter=100):
2
3     n = len(A)
4     x = x0.copy()
5     x_new = [0.0] * n
6
7     for iter in range(max_iter):
8
9         # Jacobi-Iteration
10        for i in range(n):
11            sum1 = sum(A[i][j] * x[j] for j in
12                      range(i))
13            sum2 = sum(A[i][j] * x[j] for j in
14                      range(i+1, n))
15            x_new[i] = (b[i] - sum1 - sum2) / A[i][i]
16
17        # Konvergenzpruefung
18        diff = max(abs(x_new[i] - x[i]) for i in
19                   range(n))
20
21        if diff < tol:
22            return x_new, iter + 1
23
24        x = x_new.copy()
25
26    raise ValueError(f"Keine Konvergenz nach
27                     {max_iter} Iterationen")
```

## Gauss-Seidel-Verfahren

```
1 def gauss_seidel_method(A, b, tol=1e-6, max_iter=100):
2     """Gauss-Seidel-Iterationsverfahren"""
3     n = len(A)
4
5     # Pruefe Diagonaldominanz
6     if not is_diagonally_dominant(A):
7         print("Warnung: Matrix nicht diagonaldominant")
8
9     x = [0.0] * n
10    iterations = []
11    residuals = []
12
13    for iter in range(max_iter):
14        x_old = x.copy()
15
16        # Gauss-Seidel-Iteration
17        for i in range(n):
18            sum1 = sum(A[i][j] * x[j] for j in
19                      range(i))
20            sum2 = sum(A[i][j] * x_old[j]
21                      for j in range(i+1, n))
22            x[i] = (b[i] - sum1 - sum2) / A[i][i]
23
24        # Berechne Residuum und relative Aenderung
25        res = max(abs(sum(A[i][j] * x[j]
26                           for j in range(n)) - b[i])
27                  for i in range(n))
28
29        iterations.append(x.copy())
30        residuals.append(res)
31
32        if diff < tol:
33            return {
34                'solution': x,
35                'iterations': iterations,
36                'residuals': residuals,
37                'iteration_count': iter + 1
38            }
39
40    raise ValueError(f"Keine Konvergenz nach
41                     {max_iter} Iterationen nach
42                     f\"Iterationen\\nLetztes Residuum:
43                     {res}\"")
```

## Gauss-Seidel-Verfahren Implementation

```

1 def gauss_seidel_method(A, b, x0, tol=1e-6,
2   max_iter=100):
3
4     n = len(A)
5     x = x0.copy()
6
7     for iter in range(max_iter):
8         x_old = x.copy()
9
10        # Gauss-Seidel-Iteration
11        for i in range(n):
12            sum1 = sum(A[i][j] * x[j] for j in
13                      range(i))
14            sum2 = sum(A[i][j] * x_old[j] for j in
15                      range(i+1, n))
16            x[i] = (b[i] - sum1 - sum2) / A[i][i]
17
18        # Konvergenzpruefung
19        diff = max(abs(x[i] - x_old[i]) for i in
20                  range(n))
21
22        if diff < tol:
23            return x, iter + 1
24
25        raise ValueError(f"Keine Konvergenz nach
26                          {max_iter} Iterationen")

```

## Analyse von LGS auf numerische Stabilität

```

1 def analyze_matrix(A, b):
2     n = len(A)
3     # 1. Grundlegende Eigenschaften
4     diag_dom = is_diagonally_dominant(A)
5     scaling = max(abs(A[i][j]) for i in range(n)
6                   for j in range(n))
7     # 2. Konditionszahl schätzen
8     def matrix_norm_inf(M):
9         return max(sum(abs(M[i][j]) for j in
10                        range(len(M)))
11                    for i in range(len(M)))
12
13     def inverse_power_iteration(M, max_iter=100):
14         x = [1.0] * n
15         for _ in range(max_iter):
16             y = solve_triangular(M, x)
17             norm = max(abs(yi) for yi in y)
18             x = [yi/norm for yi in y]
19             return 1.0/norm
20
21     norm_A = matrix_norm_inf(A)
22     try:
23         norm_Ainv = inverse_power_iteration(A)
24         cond = norm_A * norm_Ainv
25     except:
26         cond = float('inf')
27
28     # 3. Analyse der Diagonalelemente
29     min_diag = min(abs(A[i][i]) for i in range(n))
30     max_offdiag = max(abs(A[i][j]) for i in range(n)
31                      for j in range(n) if i != j)
32
33     # 4. Empfehlungen generieren
34     recommendations = []
35     if not diag_dom:
36         recommendations.append(
37             "Matrix nicht diagonaldominant - "
38             "Iterative Verfahren koennten divergieren")
39
40     if cond > 1e4:
41         recommendations.append(
42             f"Hohe Konditionszahl ({cond:.1e}) - "
43             "Ergebnisse koennten ungenau sein")
44
45     if min_diag < max_offdiag/100:
46         recommendations.append(
47             "Kleine Diagonalelemente - "
48             "Pivotisierung empfohlen")
49
50     if scaling > 1e8:
51         recommendations.append(
52             "Grosse Zahlenunterschiede - "
53             "Skalierung empfohlen")
54
55     return {
56         "recommendations": recommendations,
57         "results": cond, diag_dom, scaling, min_diag,
58         max_offdiag
59     }

```

## SOR-Verfahren (Relaxationsverfahren)

```

1 def sor_method(A, b, omega=1.5, tol=1e-6,
2   max_iter=100):
3     """Successive Over-Relaxation Verfahren"""
4     n = len(A)
5     x = [0.0] * n
6     iterations = []
7     residuals = []
8
9     for iter in range(max_iter):
10        x_old = x.copy()
11
12        for i in range(n):
13            sum1 = sum(A[i][j] * x[j] for j in
14                      range(i))
15            sum2 = sum(A[i][j] * x_old[j]
16                      for j in range(i+1, n))
17
18            x_new = (b[i] - sum1 - sum2) / A[i][i]
19            # Relaxation
20            x[i] = (1 - omega) * x_old[i] + omega *
21                  x_new
22
23        res = max(abs(sum(A[i][j] * x[j]
24                          for j in range(n)) - b[i])
25                  for i in range(n))
26        diff = max(abs(x[i] - x_old[i]) for i in
27                  range(n))
28
29        iterations.append(x.copy())
30        residuals.append(res)
31
32        if diff < tol:
33            return {
34                'solution': x,
35                'iterations': iterations,
36                'residuals': residuals,
37                'iteration_count': iter + 1
38            }
39
40        raise ValueError(f"Keine Konvergenz nach
41                          {max_iter} "
42                          f"Iterationen\nLetztes Residuum:
43                          {res}")

```

## 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     }
```

## Iterative Verfahren - Praktisches Vorgehen

1. Vorbereitung
  - Matrix auf Diagonaldominanz prüfen
  - Geeignete Startlösung wählen
  - Konvergenzkriterien festlegen
2. Verfahrenswahl
  - Jacobi: Einfach zu parallelisieren
  - Gauss-Seidel: Schnellere Konvergenz
  - SOR: Bei bekanntem optimalem  $\omega$
3. Durchführung
  - Residuen überwachen
  - Konvergenzrate beobachten
  - Abbruchkriterien prüfen
4. Nachbereitung
  - Qualität der Lösung prüfen
  - Konvergenzverhalten analysieren
  - Verfahren ggf. anpassen

## 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

## Hilfsfunktion für Optimierte iterative Verfahren

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

## Optimierte iterative Verfahren Implementation

- Optimierte Version mit Konvergenzanalyse
- Löst  $Ax = b$  mit verschiedenen iterativen Verfahren
- method: 'jacobi' oder 'gauss-seidel'
- omega: Relaxationsparameter (1.0 = Standard)

```
1 def iterative_solver(A, b, method='gauss_seidel',
2   tol=1e-6, max_iter=100, omega=1.0):
3
4     n = len(A)
5     x = [0.0] * n # Startvektor
6     D = [[A[i][j] if i == j else 0 for j in range(n)]
7           for i in range(n)] # Diagonalmatrix
8     L = [[A[i][j] if i > j else 0 for j in range(n)]
9           for i in range(n)] # Untere Dreiecksmatrix
10    U = [[A[i][j] if i < j else 0 for j in range(n)]
11          for i in range(n)] # Obere Dreiecksmatrix
12
13    # Konvergenzcheck
14    if not is_diagonally_dominant(A):
15        print("Warnung: Matrix nicht diagonaldominant")
16
17    iterations = []
18    residuals = []
19
20    for iter in range(max_iter):
21        x_old = x.copy()
22        if method == 'jacobi':
23            for i in range(n):
24                sum_term = sum(A[i][j] * x_old[j]
25                               for j in range(n) if j !=
26                               i)
27                x[i] = (1-omega) * x_old[i] + \
28                    (omega/A[i][i]) * (b[i] -
29                    sum_term)
30        else: # gauss_seidel
31            for i in range(n):
32                sum1 = sum(A[i][j] * x[j] for j in
33                           range(i))
34                sum2 = sum(A[i][j] * x_old[j]
35                           for j in range(i+1, n))
36                x[i] = (1-omega) * x_old[i] + \
37                    (omega/A[i][i]) * (b[i] - sum1 -
38                    sum2)
39
40        # Berechne Residuum und relative Aenderung
41        res = max(abs(sum(A[i][j] * x[j] for j in
42                           range(n)) - b[i]) for i in
43                   range(n))
44        diff = max(abs(x[i] - x_old[i]) for i in
45                   range(n))
46
47        iterations.append(x.copy())
48        residuals.append(res)
49
50        if diff < tol:
51            return {
52                'solution': x,
53                'iterations': iterations,
54                'residuals': residuals,
55                'iteration_count': iter + 1
56            }
57
58    raise ValueError(f"Keine Konvergenz nach
59                    {max_iter} " +
60                    f"Iterationen\nLetztes Residuum:
61                    {res}")
```



## Eigenwerte und Eigenvektoren

### Komplexe Zahlen in Python

```
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
18        # Multiplikation und Division
19        z_mul = z1 * z2
20        z_div = z1 / z2 if z2 != 0 else None
21
22        # Polarform
23        r1, phi1 = to_polar(z1)
24        r2, phi2 = to_polar(z2)
25
26        # Exponentialform
27        z1_exp = from_polar(r1, phi1)
28        z2_exp = from_polar(r2, phi2)
29
30        return {
31            'addition': z_add,
32            'subtraktion': z_sub,
33            'multiplikation': z_mul,
34            'division': z_div,
35            'polar_z1': (r1, phi1),
36            'polar_z2': (r2, phi2)
37        }
38    except Exception as e:
39        print(f"Fehler bei Berechnung: {e}")
40        return None
41
42 # Beispiel
43 z1 = 3 - 11j
44 z2 = 2 + 5j
45 results = complex_operations(z1, z2)
```

### 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
```

### Basisoperationen

```
1 def characteristic_polynomial(A):
2     """Charakteristisches Polynom einer 2x2 oder 3x3
3     Matrix"""
4     n = len(A)
5     if n == 2:
6         a, b = A[0][0], A[0][1]
7         c, d = A[1][0], A[1][1]
8         # det(A - lambda*I) = lambda^2 - tr(A)*lambda
9         # + det(A)
10        return [1, -(a + d), a*d - b*c]
11    elif n == 3:
12        # Hier nur Koeffizienten, keine
13        # Polynomauswertung
14        trace = sum(A[i][i] for i in range(3))
15        det = (A[0][0]*A[1][1]*A[2][2] +
16            A[0][1]*A[1][2]*A[2][0] +
17            A[0][2]*A[1][0]*A[2][1] -
18            A[0][2]*A[1][1]*A[2][0] -
19            A[0][1]*A[1][0]*A[2][2] -
20            A[0][0]*A[1][2]*A[2][1])
21        return [1, -trace, None, det] # Mittlerer
22        # Koeff. kompliziert
23    else:
24        raise ValueError("Nur fuer 2x2 oder 3x3
25        Matrizen")
26
27 def find_eigenvalues_2x2(A, tol=1e-10):
28     """Eigenwerte einer 2x2 Matrix"""
29     coeff = characteristic_polynomial(A)
30     # Quadratische Formel
31     p, q = -coeff[1], coeff[2]
32     disc = p*p/4 - q
33     if abs(disc) < tol: # Doppelte Eigenwerte
34         return [-p/2, -p/2]
35     elif disc > 0: # Reelle Eigenwerte
36         root = disc**0.5
37         return [-p/2 + root, -p/2 - root]
38     else: # Komplexe Eigenwerte
39         root = (-disc)**0.5
40         return [-p/2 + 1j*root, -p/2 - 1j*root]
41
42 def find_eigenvector(A, eigenval, tol=1e-10):
43     """Eigenvektor zu gegebenem Eigenwert"""
44     n = len(A)
45     # A - lambda*I
46     M = [[A[i][j] - (eigenval if i==j else 0)
47         for j in range(n)] for i in range(n)]
48
49     # Loese homogenes System (A - lambda*I)x = 0
50     # Nehme eine Komponente als 1 an
51     for i in range(n):
52         if abs(M[i][i]) > tol:
53             vec = [0] * n
54             vec[i] = 1
55             for j in range(n):
56                 if j != i:
57                     s = sum(-M[j][k]*vec[k] for k in
58                         range(n) if k != j)
59                     if abs(M[j][j]) > tol:
60                         vec[j] = s/M[j][j]
61             return normalize_vector(vec)
62
63     raise ValueError("Kein Eigenvektor gefunden")
```

## Charakteristisches Polynom Koeffizienten berechnen

```
1 def char_poly_coeff(matrix):
2     n = len(matrix)
3     coeffs = [0] * (n+1)
4     # Lambda^n Koeffizient
5     coeffs[n] = (-1)**n
6     # Spur (Lambda^(n-1) Koeffizient)
7     trace = sum(matrix[i][i] for i in range(n))
8     coeffs[n-1] = (-1)**(n-1) * trace
9     # Determinante (konstanter Term)
10    coeffs[0] = det_3x3(matrix)
11
12    return coeffs
13
14 # Beispielmatrix
15 A = [[2, 1, 0],
16      [1, 2, 1],
17      [0, 1, 2]]
18
19 # Charakteristisches Polynom berechnen
20 poly = char_poly_coeff(A)
21 print("Charakteristisches Polynom:")
22 print(f"p(lambda) = lambda^3 {poly[2]:+}lambda^2
23       {poly[1]:+}lambda {poly[0]:+}")
```

## Eigenvektoren berechnen use Gauss from previous example

```
1 def find_eigenvector(A, eigenvalue):
2     n = len(A)
3     # A - lambda*I
4     A_lambda = [[A[i][j] - (eigenvalue if i==j else 0)
5                  for j in range(n)] for i in range(n)]
6
7     # Loese (A - lambda*I)x = 0
8     b = [0] * n
9     return gauss_elimination(A_lambda, b)
10
11 # Eigenvektoren fuer lambda = 1,2,3 berechnen
12 eigenvalues = [1, 2, 3]
13 for ev in eigenvalues:
14     vec = find_eigenvector(A, ev)
15     print(f"Eigenvektor fuer lambda={ev}: ", vec)
```

## Vektoriteration und inverse Iteration

```
1 def power_iteration(A, tol=1e-10, max_iter=100):
2     """Von-Mises-Iteration fuer groessten Eigenwert"""
3     n = len(A)
4     # Starte mit Einheitsvektor
5     v = normalize_vector([1] + [0]*(n-1))
6     lambda_old = 0
7
8     for _ in range(max_iter):
9         # Matrix-Vektor-Multiplikation
10        w = matrix_vector_mult(A, v)
11        # Normiere neuen Vektor
12        v = normalize_vector(w)
13
14        # Rayleigh-Quotient
15        lambda_k = sum(v[i] * A[i][j] * v[j]
16                      for i in range(n)
17                      for j in range(n))
18
19        if abs(lambda_k - lambda_old) < tol:
20            return {
21                'eigenvalue': lambda_k,
22                'eigenvector': v
23            }
24
25        lambda_old = lambda_k
26
27    raise ValueError("Keine Konvergenz")
28
29 def inverse_iteration(A, mu, tol=1e-10, max_iter=100):
30     """Inverse Iteration fuer Eigenwert nahe mu"""
31     n = len(A)
32     # Starte mit Einheitsvektor
33     v = normalize_vector([1] + [0]*(n-1))
34     lambda_old = 0
35
36     # (A - mu*I) berechnen
37     M = [[A[i][j] - (mu if i==j else 0)
38          for j in range(n)] for i in range(n)]
39
40     for _ in range(max_iter):
41         # Loese (A - mu*I)w = v
42         w = gauss_elimination(M, v)['solution']
43         # Normiere neuen Vektor
44         v = normalize_vector(w)
45
46         # Rayleigh-Quotient
47         lambda_k = sum(v[i] * A[i][j] * v[j]
48                       for i in range(n)
49                       for j in range(n))
50
51         if abs(lambda_k - lambda_old) < tol:
52             return {
53                 'eigenvalue': lambda_k,
54                 'eigenvector': v
55             }
56
57        lambda_old = lambda_k
58
59    raise ValueError("Keine Konvergenz")
```

## Von-Mises-Iteration

```
1 def matrix_vector_mult(A, v):
2     n = len(A)
3     result = [0] * n
4     for i in range(n):
5         for j in range(n):
6             result[i] += A[i][j] * v[j]
7     return result
8
9 def vector_norm(v):
10    return sum(x*x for x in v) ** 0.5
11
12 def normalize_vector(v):
13    norm = vector_norm(v)
14    return [x/norm for x in v]
15
16 def power_iteration(A, tol=1e-10, max_iter=100):
17     n = len(A)
18     # Startvektor
19     v = normalize_vector([1] + [0]*(n-1))
20
21     for _ in range(max_iter):
22         # Matrix-Vektor-Multiplikation
23         w = matrix_vector_mult(A, v)
24         # Normierung
25         v_new = normalize_vector(w)
26
27         # Rayleigh-Quotient
28         lambda_k = sum(v_new[i] * A[i][j] * v_new[j]
29                       for i in range(n)
30                       for j in range(n))
31
32         # Konvergenzpruefung
33         if vector_norm([v_new[i]-v[i] for i in
34                       range(n)]) < tol:
35             return lambda_k, v_new
36
37        v = v_new
38    return lambda_k, v
```

## QR-Iteration

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

## QR-Verfahren

```
1 def matmul(A, B):
2     n = len(A)
3     C = [[0.0] * n for _ in range(n)]
4     for i in range(n):
5         for j in range(n):
6             C[i][j] = sum(A[i][k] * B[k][j]
7                           for k in range(n))
8     return C
9
10 def transpose(A):
11     n = len(A)
12     return [[A[j][i] for j in range(n)]
13             for i in range(n)]
14
15 def householder(x):
16     n = len(x)
17     # Norm berechnen
18     s = sum(x[i]**2 for i in range(1, n))
19     v = [0.0] * n
20     if s == 0:
21         return v
22
23     v[0] = x[0]
24     norm_x = (x[0]**2 + s)**0.5
25     if x[0] <= 0:
26         v[0] = x[0] - norm_x
27     else:
28         v[0] = -s/(x[0] + norm_x)
29
30     for i in range(1, n):
31         v[i] = x[i]
32
33     return normalize_vector(v)
34
35 def qr_algorithm(A, tol=1e-10, max_iter=100):
36     n = len(A)
37     A_k = [row[:] for row in A] # Kopiere A
38
39     for k in range(max_iter):
40         # QR-Zerlegung mit Householder
41         Q = [[1.0 if i==j else 0.0
42               for j in range(n)]
43               for i in range(n)]
44         R = [row[:] for row in A_k]
45
46         for j in range(n-1):
47             v = householder([R[i][j]
48                               for i in range(j, n)])
49             # Householder-Transformation anwenden
50
51         # Neue Iteration A_(k+1) = RQ
52         A_k = matmul(R, Q)
53
54         # Konvergenztest
55         if max(abs(A_k[i][j])
56               for i in range(1, n)
57               for j in range(i)) < tol:
58             break
59
60     return [A_k[i][i] for i in range(n)]
```

## Eigenwertberechnung - Praktisches Vorgehen

1. Voranalyse
  - Matrix auf spezielle Struktur prüfen (symmetrisch, etc.)
  - Grobe Eigenwertabschätzung durch Gerschgorin-Kreise
  - Kondition der Matrix prüfen
2. Verfahrenswahl
  - QR: Für alle Eigenwerte
  - Vektoriteration: Für größten Eigenwert
  - Inverse Iteration: Für bekannte Näherung
3. Durchführung
  - Geeignete Toleranz wählen
  - Konvergenz überwachen
  - Bei Bedarf Shifts einsetzen
4. Validation
  - Residuen prüfen:  $(A - \lambda I)v$
  - Orthogonalität der Eigenvektoren testen
  - Ergebnisse physikalisch interpretieren

# Python Implementationen

## Hilfsfunktionen

## Numerische Lösung von Nullstellenproblemen

### 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
9
10 #Returns: Fehlerschranke eps wenn
11 #Vorzeichenwechsel, sonst None
```

### Verschiedene Abbruchkriterien prüfen Konvergenzkriterien

```
1 def convergence_criteria(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
7     # Relative Aenderung der x-Werte
8     if abs(x_new - x_old) < tol * (1 + abs(x_new)):
9         return True, "Relative Aenderung < tol"
10
11     # Relative Aenderung der Funktionswerte
12     if abs(f_new - f_old) < tol * (1 + abs(f_new)):
13         return True, "Funktionsaenderung < tol"
14
15     # Divergenzcheck
16     if abs(f_new) > 2 * abs(f_old):
17         return False, "Divergenz detektiert"
18
19     return False, "Noch nicht konvergiert"
20
21 #Returns: (konvergiert?, grund)
```

## 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-Verfahren

```
1 def newton(f, df, x0, tol=1e-6, max_iter=100):
2     x = x0
3     for i in range(max_iter):
4         fx = f(x)
5         if abs(fx) < tol:
6             return x, i+1
7         dfx = df(x)
8         if dfx == 0:
9             raise ValueError("Ableitung Null")
10         x = x - fx/dfx
11     raise ValueError("Keine Konvergenz")
12
13 # Optimierte Version mit Fehlerkontrolle
14 def newton_safe(f, df, x0, tol=1e-6, max_it=100):
15     x = x0
16     fx = f(x)
17     for i in range(max_it):
18         dfx = df(x)
19         if dfx == 0:
20             raise ValueError("Ableitung Null")
21         dx = fx/dfx
22         x_new = x - dx
23         fx_new = f(x_new)
24         # Verschiedene Konvergenzkriterien
25         if abs(fx_new) < tol: # Funktionswert
26             return x_new, i+1
27         if abs(dx) < tol * (1 + abs(x)): # Relative
28             # Aenderung
29             return x_new, i+1
30         if abs(fx_new) >= abs(fx): # Divergenzcheck
31             raise ValueError("Divergenz detektiert")
32         x, fx = x_new, fx_new
33     raise ValueError("Keine Konvergenz")
```

## Sekantenverfahren

```
1 # Einfache Version
2 def secant(f, x0, x1, tol=1e-6, max_iter=100):
3     fx0 = f(x0)
4     fx1 = f(x1)
5
6     for i in range(max_iter):
7         if abs(fx1) < tol:
8             return x1, i+1
9
10        if fx1 == fx0:
11            raise ValueError("Division durch Null")
12
13        x2 = x1 - fx1 * (x1 - x0)/(fx1 - fx0)
14        x0, x1 = x1, x2
15        fx0, fx1 = fx1, f(x2)
16
17        raise ValueError("Keine Konvergenz")
18
19 # Optimierte Version mit Fehlerkontrolle
20 def secant_safe(f, x0, x1, tol=1e-6, max_iter=100):
21     fx0 = f(x0)
22     fx1 = f(x1)
23
24     if abs(fx0) < abs(fx1): # Stelle mit kleinerem
25         # f-Wert als x1
26         x0, x1 = x1, x0
27         fx0, fx1 = fx1, fx0
28
29     for i in range(max_iter):
30         if abs(fx1) < tol:
31             return x1, i+1
32
33         if fx1 == fx0:
34             raise ValueError("Division durch Null")
35
36         # Sekanten-Schritt
37         d = fx1 * (x1 - x0)/(fx1 - fx0)
38         x2 = x1 - d
39
40         # Konvergenzpruefungen
41         if abs(d) < tol * (1 + abs(x1)): # Relative
42             # Aenderung
43             return x2, i+1
44
45         fx2 = f(x2)
46         if abs(fx2) >= abs(fx1): # Divergenzcheck
47             if i == 0:
48                 raise ValueError("Schlechte
49                 Startwerte")
50             return x1, i+1
51
52         x0, x1 = x1, x2
53         fx0, fx1 = fx1, fx2
54
55         raise ValueError("Keine Konvergenz")
```

## Nullstellensuche mit Fehlerabschätzung

```
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        # Prüfe Konvergenzkriterien
12        converged, reason = convergence_criteria(
13            x_new, x_old, f_new, f_old, tol)
14
15        if converged:
16            # Schätze 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")
29
30    # Returns: Dictionary mit Ergebnissen
31
32    # Beispielnutzung
33    def example_function(x):
34        return x**2 - 2
35
36    result = root_finder_with_error(example_function, 1.0)
37    print(f"Nullstelle: {result['root']:.10f}")
38    print(f"Iterationen: {result['iterations']}")
39    print(f"Fehlerschranke: {result['error_bound']:.10f}")
40    print(f"Konvergenzgrund:
41        {result['convergence_reason']}")
42
43    # Ausgabe etwa:
44    # Nullstelle: 1.4142135624
45    # Iterationen: 5
46    # Fehlerschranke: 1e-06
47    # Konvergenzgrund: Funktionswert < tol
```

## Numerische Lösung von Gleichungssystemen

### Gauss-Elimination mit Pivottisierung

```
1 def gauss_elimination(A, b):
2     n = len(A)
3     # Erweiterte Matrix erstellen
4     M = [[A[i][j] for j in range(n)] + [b[i]] for i in
5         range(n)]
6
7     # Vorwärtselimination
8     for i in range(n):
9         pivot = M[i][i]
10        if abs(pivot) < 1e-10:
11            continue
12        for j in range(i+1, n):
13            factor = M[j][i] / pivot
14            for k in range(i, n+1):
15                M[j][k] -= factor * M[i][k]
16
17    # Rückwärtssubstitution
18    x = [0] * n
19    for i in range(n-1, -1, -1):
20        if abs(M[i][i]) < 1e-10:
21            x[i] = 1 # Freie Variable
22            continue
23        sum_val = sum(M[i][j] * x[j] for j in
24            range(i+1, n))
25        x[i] = (M[i][n] - sum_val) / M[i][i]
26
27    return x
```

## LR-Zerlegung Implementation

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



## QR-Zerlegung Implementation

```
1 def qr_decomposition(A):
2     m = len(A)
3     n = len(A[0])
4     # Kopiere A nach R (deep copy)
5     R = [[A[i][j] for j in range(n)] for i in range(m)]
6     # Initialisiere Q als Einheitsmatrix
7     Q = [[1.0 if i == j else 0.0 for j in range(m)]
8           for i in range(m)]
9
10    def vector_norm(v): # Norm eines Vektors
11        return (sum(x*x for x in v)) ** 0.5
12
13    def matrix_mult(A, B): # Matrixmultiplikation
14        m, n = len(A), len(B[0])
15        p = len(B)
16        C = [[0.0] * n for _ in range(m)]
17
18        for i in range(m):
19            for j in range(n):
20                C[i][j] = sum(A[i][k] * B[k][j]
21                               for k in range(p))
22
23    return C
24
25    def householder_reflection(x):
26        n = len(x)
27        v = [xi for xi in x] # Kopiere x
28
29        # Berechne Norm des Teilvektors
30        sigma = sum(v[i]*v[i] for i in range(1, n))
31        if sigma == 0 and x[0] >= 0:
32            beta = 0
33        elif sigma == 0 and x[0] < 0:
34            beta = -2
35        else:
36            mu = (x[0]*x[0] + sigma)**0.5
37            if x[0] <= 0:
38                v[0] = x[0] - mu
39            else:
40                v[0] = -sigma/(x[0] + mu)
41
42        beta = 2*v[0]*v[0]/(sigma + v[0]*v[0])
43
44        # Normiere v
45        temp = v[0]
46        for i in range(n):
47            v[i] /= temp
48
49    return v, beta
```

## QR-Zerlegung - Praktisches Vorgehen

1. Vorbereitungen
  - Matrix A kopieren für R
  - Q als Einheitsmatrix initialisieren
  - Householder-Vektoren speichern
2. Für jede Spalte  $k = 1, \dots, n-1$ :
  - Untervektor  $v_k$  aus k-ter Spalte extrahieren
  - Householder-Vektor berechnen:
    - $w_k = v_k + \text{sign}(v_{k1}) \|v_k\| e_1$
    - $u_k = \frac{w_k}{\|w_k\|}$
  - Householder-Matrix auf Untermatrix anwenden:
    - $H_k = I - 2u_k u_k^T$
    - $R_{k:n, k:n} = H_k \cdot R_{k:n, k:n}$
  - Q aktualisieren:  $Q = Q \cdot H_k^T$
3. System lösen durch
  - $y = Q^T b$  berechnen
  - Rückwärtseinsetzen:  $Rx = y$

## QR-Zerlegung Implementation (Fortsetzung)

```
1 # Hauptschleife der QR-Zerlegung
2 for k in range(n):
3     # Extrahiere k-te Spalte ab k-ter Zeile
4     x = [R[i][k] for i in range(k, m)]
5     if len(x) > 1: # wenn noch Untermatrix
6         # Berechne Householder-Transformation
7         v, beta = householder_reflection(x)
8         # Wende Householder auf R an
9         for j in range(k, n):
10            # Berechne w = beta * (v^T * R_j)
11            w = beta * sum(v[i-k]*R[i][j]
12                           for i in range(k, m))
13            # Update R
14            for i in range(k, m):
15                R[i][j] -= v[i-k] * w
16            # Update Q
17            for j in range(m):
18                w = beta * sum(v[i-k]*Q[j][i+k]
19                              for i in range(len(v)))
20                for i in range(len(v)):
21                    Q[j][k+i] -= v[i] * w
22            # Transponiere Q am Ende
23            Q = [[Q[j][i] for j in range(m)] for i in range(m)]
24
25    return Q, R # Q (orthogonal) und R (obere
26                Dreiecksmatrix)
27
28 # Beispiel fuer Verwendung
29 def solve_qr(A, b): # Loest Ax = b mittels QR-Zerlegung
30     Q, R = qr_decomposition(A)
31     # Berechne Q^T * b
32     y = [sum(Q[i][j] * b[j]
33              for j in range(len(b)))
34          for i in range(len(b))]
35     # Rueckwaertseinsetzen
36     n = len(R)
37     x = [0] * n
38     for i in range(n-1, -1, -1):
39         s = sum(R[i][j] * x[j] for j in range(i+1, n))
40         if abs(R[i][i]) < 1e-10:
41             raise ValueError("Matrix (fast) singular")
42         x[i] = (y[i] - s) / R[i][i]
43     return x
```

## Analyse der Genauigkeit einer Näherungslösung

```
1 def error_analysis(A, x, b, x_approx):
2     n = len(A)
3     # Residuum berechnen
4     r = [b[i] - sum(A[i][j] * x_approx[j]
5                    for j in range(n))
6          for i in range(n)]
7     res_norm = max(abs(ri) for ri in r)
8     # Relativer Fehler (falls exakte Loesung bekannt)
9     if x is not None:
10        rel_error = max(abs(x[i] - x_approx[i])
11                        for i in range(n)) / \
12                    max(abs(x[i]) for i in range(n))
13    else:
14        rel_error = None
15    # Absoluter Fehler (falls exakte Loesung bekannt)
16    if x is not None:
17        abs_error = max(abs(x[i] - x_approx[i])
18                        for i in range(n))
19    else:
20        abs_error = None
21
22    return {
23        'residual_norm': res_norm,
24        'relative_error': rel_error,
25        'residual': r
26    }
```

## Jacobi-Verfahren Implementation

```
1 def jacobi_method(A, b, x0, tol=1e-6, max_iter=100):
2
3     n = len(A)
4     x = x0.copy()
5     x_new = [0.0] * n
6
7     for iter in range(max_iter):
8
9         # Jacobi-Iteration
10        for i in range(n):
11            sum1 = sum(A[i][j] * x[j] for j in
12                      range(i))
13            sum2 = sum(A[i][j] * x[j] for j in
14                      range(i+1, n))
15            x_new[i] = (b[i] - sum1 - sum2) / A[i][i]
16
17        # Konvergenzpruefung
18        diff = max(abs(x_new[i] - x[i]) for i in
19                  range(n))
20
21        if diff < tol:
22            return x_new, iter + 1
23
24    x = x_new.copy()
25
26    raise ValueError(f"Keine Konvergenz nach
27                    {max_iter} Iterationen")
```

## Gauss-Seidel-Verfahren Implementation

```
1 def gauss_seidel_method(A, b, x0, tol=1e-6,
2   max_iter=100):
3
4     n = len(A)
5     x = x0.copy()
6
7     for iter in range(max_iter):
8         x_old = x.copy()
9
10        # Gauss-Seidel-Iteration
11        for i in range(n):
12            sum1 = sum(A[i][j] * x[j] for j in
13                      range(i))
14            sum2 = sum(A[i][j] * x_old[j] for j in
15                      range(i+1, n))
16            x[i] = (b[i] - sum1 - sum2) / A[i][i]
17
18        # Konvergenzpruefung
19        diff = max(abs(x[i] - x_old[i]) for i in
20                  range(n))
21
22        if diff < tol:
23            return x, iter + 1
24
25        raise ValueError(f"Keine Konvergenz nach
26                          {max_iter} Iterationen")
```

## Analyse von LGS auf numerische Stabilität

```
1 def analyze_matrix(A, b):
2     n = len(A)
3     # 1. Grundlegende Eigenschaften
4     diag_dom = is_diagonally_dominant(A)
5     scaling = max(abs(A[i][j]) for i in range(n)
6                   for j in range(n))
7     # 2. Konditionszahl schätzen
8     def matrix_norm_inf(M):
9         return max(sum(abs(M[i][j]) for j in
10                        range(len(M)))
11                    for i in range(len(M)))
12
13     def inverse_power_iteration(M, max_iter=100):
14         x = [1.0] * n
15         for _ in range(max_iter):
16             y = solve_triangular(M, x)
17             norm = max(abs(yi) for yi in y)
18             x = [yi/norm for yi in y]
19         return 1.0/norm
20
21     norm_A = matrix_norm_inf(A)
22     try:
23         norm_Ainv = inverse_power_iteration(A)
24         cond = norm_A * norm_Ainv
25     except:
26         cond = float('inf')
27
28     # 3. Analyse der Diagonalelemente
29     min_diag = min(abs(A[i][i]) for i in range(n))
30     max_offdiag = max(abs(A[i][j]) for i in range(n)
31                       for j in range(n) if i != j)
32
33     # 4. Empfehlungen generieren
34     recommendations = []
35     if not diag_dom:
36         recommendations.append(
37             "Matrix nicht diagonaldominant - "
38             "Iterative Verfahren koennten divergieren")
39
40     if cond > 1e4:
41         recommendations.append(
42             f"Hohe Konditionszahl ({cond:.1e}) - "
43             "Ergebnisse koennten ungenau sein")
44
45     if min_diag < max_offdiag/100:
46         recommendations.append(
47             "Kleine Diagonalelemente - "
48             "Pivotisierung empfohlen")
49
50     if scaling > 1e8:
51         recommendations.append(
52             "Grosse Zahlenunterschiede - "
53             "Skalierung empfohlen")
54
55     return {
56         "recommendations": recommendations,
57         "results": cond, diag_dom, scaling, min_diag,
58         max_offdiag
59     }
```

### 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

## Hilfsfunktion für Optimierte iterative Verfahren

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

## Optimierte iterative Verfahren Implementation

- Optimierte Version mit Konvergenzanalyse
- Löst  $Ax = b$  mit verschiedenen iterativen Verfahren
- method: 'jacobi' oder 'gauss-seidel'
- omega: Relaxationsparameter (1.0 = Standard)

```
1 def iterative_solver(A, b, method='gauss_seidel',
2   tol=1e-6, max_iter=100, omega=1.0):
3
4     n = len(A)
5     x = [0.0] * n # Startvektor
6     D = [[A[i][j] if i == j else 0 for j in range(n)]
7           for i in range(n)] # Diagonalmatrix
8     L = [[A[i][j] if i > j else 0 for j in range(n)]
9           for i in range(n)] # Untere Dreiecksmatrix
10    U = [[A[i][j] if i < j else 0 for j in range(n)]
11          for i in range(n)] # Obere Dreiecksmatrix
12
13    # Konvergenzcheck
14    if not is_diagonally_dominant(A):
15        print("Warnung: Matrix nicht diagonaldominant")
16
17    iterations = []
18    residuals = []
19
20    for iter in range(max_iter):
21        x_old = x.copy()
22        if method == 'jacobi':
23            for i in range(n):
24                sum_term = sum(A[i][j] * x_old[j]
25                               for j in range(n) if j != i)
26                x[i] = (1-omega) * x_old[i] + \
27                       (omega/A[i][i]) * (b[i] -
28                                           sum_term)
29        else: # gauss_seidel
30            for i in range(n):
31                sum1 = sum(A[i][j] * x[j] for j in
32                           range(i))
33                sum2 = sum(A[i][j] * x_old[j]
34                           for j in range(i+1, n))
35                x[i] = (1-omega) * x_old[i] + \
36                       (omega/A[i][i]) * (b[i] - sum1 -
37                                           sum2)
38
39    # Berechne Residuum und relative Aenderung
40    res = max(abs(sum(A[i][j] * x[j] for j in
41                      range(n)) - b[i]) for i in range(n))
42    diff = max(abs(x[i] - x_old[i]) for i in
43               range(n))
44
45    iterations.append(x.copy())
46    residuals.append(res)
47
48    if diff < tol:
49        return {
50            'solution': x,
51            'iterations': iterations,
52            'residuals': residuals,
53            'iteration_count': iter + 1
54        }
55
56    raise ValueError(f"Keine Konvergenz nach
57                      {max_iter} " +
58                      f"Iterationen\nLetztes Residuum:
59                      {res}")
```

## Eigenwerte und Eigenvektoren

### Komplexe Zahlen in Python

```
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
18        # Multiplikation und Division
19        z_mul = z1 * z2
20        z_div = z1 / z2 if z2 != 0 else None
21
22        # Polarform
23        r1, phi1 = to_polar(z1)
24        r2, phi2 = to_polar(z2)
25
26        # Exponentialform
27        z1_exp = from_polar(r1, phi1)
28        z2_exp = from_polar(r2, phi2)
29
30        return {
31            'addition': z_add,
32            'subtraktion': z_sub,
33            'multiplikation': z_mul,
34            'division': z_div,
35            'polar_z1': (r1, phi1),
36            'polar_z2': (r2, phi2)
37        }
38
39    except Exception as e:
40        print(f"Fehler bei Berechnung: {e}")
41        return None
42
43    # Beispiel
44    z1 = 3 - 11j
45    z2 = 2 + 5j
46    results = complex_operations(z1, z2)
```

### 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
```

### Charakteristisches Polynom Koeffizienten berechnen

```
1 def char_poly_coeff(matrix):
2     n = len(matrix)
3     coeffs = [0] * (n+1)
4     # Lambda^n Koeffizient
5     coeffs[n] = (-1)**n
6     # Spur (Lambda^(n-1) Koeffizient)
7     trace = sum(matrix[i][i] for i in range(n))
8     coeffs[n-1] = (-1)**(n-1) * trace
9     # Determinante (konstanter Term)
10    coeffs[0] = det_3x3(matrix)
11
12    return coeffs
13
14    # Beispielmatrix
15    A = [[2, 1, 0],
16         [1, 2, 1],
17         [0, 1, 2]]
18
19    # Charakteristisches Polynom berechnen
20    poly = char_poly_coeff(A)
21    print("Charakteristisches Polynom:")
22    print(f"p(lambda) = lambda^3 {poly[2]:+}lambda^2
23          {poly[1]:+}lambda {poly[0]:+}")
```

### Eigenvektoren berechnen use Gauss from previous example

```
1 def find_eigenvector(A, eigenvalue):
2     n = len(A)
3     # A - lambda*I
4     A_lambda = [[A[i][j] - (eigenvalue if i==j else 0)
5                   for j in range(n)] for i in range(n)]
6
7     # Loese (A - lambda*I)x = 0
8     b = [0] * n
9     return gauss_elimination(A_lambda, b)
10
11    # Eigenvektoren fuer lambda = 1,2,3 berechnen
12    eigenvalues = [1, 2, 3]
13    for ev in eigenvalues:
14        vec = find_eigenvector(A, ev)
15        print(f"Eigenvektor fuer lambda={ev}:", vec)
```

## Von-Mises-Iteration

```
1 def matrix_vector_mult(A, v):
2     n = len(A)
3     result = [0] * n
4     for i in range(n):
5         for j in range(n):
6             result[i] += A[i][j] * v[j]
7     return result
8
9 def vector_norm(v):
10    return sum(x*x for x in v) ** 0.5
11
12 def normalize_vector(v):
13    norm = vector_norm(v)
14    return [x/norm for x in v]
15
16 def power_iteration(A, tol=1e-10, max_iter=100):
17    n = len(A)
18    # Startvektor
19    v = normalize_vector([1] + [0]*(n-1))
20
21    for _ in range(max_iter):
22        # Matrix-Vektor-Multiplikation
23        w = matrix_vector_mult(A, v)
24        # Normierung
25        v_new = normalize_vector(w)
26
27        # Rayleigh-Quotient
28        lambda_k = sum(v_new[i] * A[i][j] * v_new[j]
29                        for i in range(n)
30                        for j in range(n))
31
32        # Konvergenzpruefung
33        if vector_norm([v_new[i]-v[i] for i in
34                        range(n)]) < tol:
35            return lambda_k, v_new
36
37        v = v_new
38    return lambda_k, v
```

## QR-Verfahren

```
1 def matmul(A, B):
2     n = len(A)
3     C = [[0.0] * n for _ in range(n)]
4     for i in range(n):
5         for j in range(n):
6             C[i][j] = sum(A[i][k] * B[k][j]
7                           for k in range(n))
8     return C
9
10 def transpose(A):
11     n = len(A)
12     return [[A[j][i] for j in range(n)]
13             for i in range(n)]
14
15 def householder(x):
16     n = len(x)
17     # Norm berechnen
18     s = sum(x[i]**2 for i in range(1, n))
19     v = [0.0] * n
20     if s == 0:
21         return v
22
23     v[0] = x[0]
24     norm_x = (x[0]**2 + s)**0.5
25     if x[0] <= 0:
26         v[0] = x[0] - norm_x
27     else:
28         v[0] = -s/(x[0] + norm_x)
29
30     for i in range(1, n):
31         v[i] = x[i]
32
33     return normalize_vector(v)
34
35 def qr_algorithm(A, tol=1e-10, max_iter=100):
36     n = len(A)
37     A_k = [row[:] for row in A] # Kopiere A
38
39     for k in range(max_iter):
40         # QR-Zerlegung mit Householder
41         Q = [[1.0 if i==j else 0.0
42               for j in range(n)]
43              for i in range(n)]
44         R = [row[:] for row in A_k]
45
46         for j in range(n-1):
47             v = householder([R[i][j]
48                             for i in range(j, n)])
49             # Householder-Transformation anwenden
50
51             # Neue Iteration A_(k+1) = RQ
52             A_k = matmul(R, Q)
53
54             # Konvergenztest
55             if max(abs(A_k[i][j])
56                   for i in range(1, n)
57                   for j in range(i)) < tol:
58                 break
59
60     return [A_k[i][i] for i in range(n)]
```

Additional Examples

Rechnerarithmetik

Werteberechnung ausführlich Gegeben sei die Maschinenzahl zur Basis  $B = 2$ :

$$x = \underbrace{0.1101}_{n=4} \cdot \underbrace{2_2^{101}}_{l=3}$$

1. Normalisierung prüfen:

- $m_1 = 1 \neq 0 \rightarrow$  normalisiert

2. Exponent berechnen:

$$\hat{e} = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$$
$$= 4 + 0 + 1 = 5$$

3. Wert berechnen:

$$\hat{\omega} = 1 \cdot 2^{5-1} + 1 \cdot 2^{5-2} + 0 \cdot 2^{5-3} + 1 \cdot 2^{5-4}$$
$$= 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1$$
$$= 16 + 8 + 0 + 2$$
$$= 26$$

Also ist  $x = 26$

Weitere Beispiele

- 1. Basis 10:  $0.3141 \cdot 10^2$ 
  - Normalisiert, da  $m_1 = 3 \neq 0$
  - $\hat{e} = 2$
  - $\hat{\omega} = 3 \cdot 10^1 + 1 \cdot 10^0 + 4 \cdot 10^{-1} + 1 \cdot 10^{-2} = 31.41$
- 2. Basis 16 (hex):  $0.A5F \cdot 16^3$ 
  - Normalisiert, da  $m_1 = A = 10 \neq 0$
  - $\hat{e} = 3$
  - $\hat{\omega} = 10 \cdot 16^2 + 5 \cdot 16^1 + 15 \cdot 16^0 = 2655$

Werteberechnung Berechnung einer Zahl zur Basis B=2:

$$\underbrace{0.1011}_{n=4} \cdot \underbrace{2^3}_{l=1}$$

1. Exponent:  $\hat{e} = 3$

2. Wert:  $\hat{\omega} = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1}$

$= 4 + 0 + 1 + 0.5 = 5.5$

Numerische Lösung von Nullstellenproblemen

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

Newton-Verfahren Berechnung von  $\sqrt[3]{2}$  Nullstellenproblem:  $f(x) = x^3 - 2$

- Ableitung:  $f'(x) = 3x^2$ , Startwert  $x_0 = 1$       Quadratische Konvergenz sichtbar
- 1.  $x_1 = 1 - \frac{1^3 - 2}{3 \cdot 1^2} = 1.333333$       durch schnelle Annäherung an
  - 2.  $x_2 = 1.333333 - \frac{1.333333^3 - 2}{3 \cdot 1.333333^2} = 1.259921$        $\sqrt[3]{2} \approx 1.259921$
  - 3.  $x_3 = 1.259921 - \frac{1.259921^3 - 2}{3 \cdot 1.259921^2} = 1.259921$

Numerische Lösung von LGS

Pivotisierung in der Praxis Betrachten Sie das System:

$$\begin{pmatrix} 0.001 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

Ohne Pivotisierung:

Division durch 0.001 führt zu großen Rundungsfehlern:

$$x_1 \approx 1000 \cdot (1 - x_2)$$

Mit Pivotisierung:

Nach Zeilenvertauschung:

$$\begin{pmatrix} 1 & 1 \\ 0.001 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \end{pmatrix}$$

Liefert stabile Lösung:  $x_1 = 1$ ,  $x_2 = 1$

LR-Zerlegung mit Pivotisierung Gegeben sei das System:

$$A = \begin{pmatrix} 1 & 2 & 1 \\ 3 & 8 & 1 \\ 0 & 4 & 1 \end{pmatrix}, \quad b = \begin{pmatrix} 2 \\ 3 \\ 5 \end{pmatrix}$$

1. Erste Spalte

Max Element in 1. Spalte:  $|a_{21}| = 3$ , tausche Z1 und Z2:

$$P_1 = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad A^{(1)} = \begin{pmatrix} 3 & 8 & 1 \\ 1 & 2 & 1 \\ 0 & 4 & 1 \end{pmatrix}$$

Eliminationsfaktoren:  $l_{21} = \frac{1}{3}$ ,  $l_{31} = 0$

Nach Elimination:

$$A^{(2)} = \begin{pmatrix} 3 & 8 & 1 \\ 0 & -\frac{2}{3} & \frac{1}{3} \\ 0 & 4 & 1 \end{pmatrix}$$

2. Zweite Spalte

Max Element:  $|a_{32}| = 4$ , tausche Z2 und Z3:

$$P_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

Eliminationsfaktor:  $l_{32} = -\frac{1}{6}$

Nach Elimination:

$$R = \begin{pmatrix} 3 & 8 & 1 \\ 0 & 4 & 1 \\ 0 & 0 & \frac{5}{6} \end{pmatrix}$$

Endergebnis

$$P = P_2 P_1 = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}, \quad L = \begin{pmatrix} 1 & 0 & 0 \\ \frac{1}{3} & 1 & 0 \\ 0 & -\frac{1}{6} & 1 \end{pmatrix}$$

Lösung des Systems

- 1.  $Pb = \begin{pmatrix} 3 \\ 3 \\ 2 \end{pmatrix}$
- 2.  $Ly = Pb$ :  $y = \begin{pmatrix} 3 \\ 4 \\ 1 \end{pmatrix}$
- 3.  $Rx = y$ :  $x = \begin{pmatrix} 1 \\ 0 \\ \frac{6}{5} \end{pmatrix}$

QR-Zerlegung Gegeben sei die Matrix:

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

1. Erste Spalte

$$v_1 = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}, \quad \|v_1\| = \sqrt{2}$$

Householder-Vektor:  $w_1 = v_1 + \sqrt{2} \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1+\sqrt{2} \\ 1 \\ 0 \end{pmatrix}$

Normierung:  $u_1 = \frac{1}{\sqrt{4+2\sqrt{2}}} \begin{pmatrix} 1+\sqrt{2} \\ 1 \\ 0 \end{pmatrix}$

Erste Householder-Matrix:

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

2. Zweite Spalte

Nach Anwendung von  $H_1$ :

$$H_1 A = \begin{pmatrix} -\sqrt{2} & -\frac{1}{\sqrt{2}} \\ 0 & \frac{1}{\sqrt{2}} \\ 0 & 1 \end{pmatrix}$$

Untervektor für zweite Transformation:  $v_2 = \begin{pmatrix} \frac{1}{\sqrt{2}} \\ 1 \end{pmatrix}$

Analog zur ersten Transformation erhält man:

$$H_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & -\frac{1}{\sqrt{5}} & -\frac{2}{\sqrt{5}} \\ 0 & -\frac{2}{\sqrt{5}} & \frac{1}{\sqrt{5}} \end{pmatrix}$$

Endergebnis

$$Q = H_1^T H_2^T = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$R = H_2 H_1 A = \begin{pmatrix} \sqrt{2} & 1 \\ 0 & \sqrt{2} \\ 0 & 0 \end{pmatrix}$$

Verifikation

- $Q^T Q = Q Q^T = I$  (Orthogonalität)
- $Q R = A$  (bis auf Rundungsfehler)
- R ist obere Dreiecksmatrix

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



Eigenvektoren und Eigenwerte

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

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

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

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

**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.  $\text{tr}(A) = \lambda_1 + \lambda_2 + \lambda_3 = 6$

4. Spektrum:  $\sigma(A) = \{1, 2, 3\}$

**Von-Mises-Iteration**    Berechne größten Eigenwert der Matrix:

$A = \begin{pmatrix} 4 & -1 & 1 \\ -1 & 3 & -2 \\ 1 & -2 & 3 \end{pmatrix}, \quad \text{Startvektor: } v^{(0)} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$

k	$v^{(k)}$	$\lambda^{(k)}$
0	$(1, 0, 0)^T$	-
1	$(0.970, -0.213, 0.119)^T$	4.000
2	$(0.957, -0.239, 0.164)^T$	4.827
3	$(0.953, -0.244, 0.178)^T$	4.953
4	$(0.952, -0.245, 0.182)^T$	4.989

Konvergenz gegen  $\lambda_1 \approx 5$   
Eigenvektor  $v \approx (0.952, -0.245, 0.182)^T$

**QR-Verfahren**    Matrix:

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

**QR-Iteration:**

1.  $A_0 = A$   
2. Nach erster Iteration:

$A_1 = \begin{pmatrix} 3.21 & -0.83 & 0.62 \\ -0.83 & 2.13 & 0.41 \\ 0.62 & 0.41 & 0.66 \end{pmatrix}$

3. Nach 5 Iterationen:

$A_5 \approx \begin{pmatrix} 4 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$

Die Diagonalelemente von  $A_5$  sind die Eigenwerte:  $\lambda_1 = 4, \lambda_2 = 1, \lambda_3 = 1$