

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

**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 $	$\left  \frac{\tilde{x} - x}{x} \right $ bzw. $\left  \frac{\tilde{x} - x}{ x } \right $ 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

Analyse von Algorithmen

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

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

Numerische Stabilität eines Algorithmus

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

**Numerische Stabilität** Fibonacci-Zahlen:

```
1 def fib_unstable(n): # Instabile rekursive Version
2     if n <= 1:
3         return n
4     return fib_unstable(n-1) + fib_unstable(n-2)
5
6 def fib_stable(n): # Stabile iterative Version
7     if n <= 1:
8         return n
9     a, b = 0, 1
10    for _ in range(2, n + 1):
11        a, b = b, a + b
12    return b
```

**Stabilitätsanalyse** Schritte zur Analyse der numerischen Stabilität:

- Bestimmen Sie kritische Operationen
- Schätzen Sie Rundungsfehler pro Operation ab
- Analysieren Sie die Fehlerfortpflanzung
- Berechnen Sie die worst-case Fehlerschranke
- Vergleichen Sie alternative Implementierungen

Implementierungsgenauigkeit eines Algorithmus

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

Robuste Implementierung von Algorithmen

- Verwenden Sie stabile Grundoperationen
- Vermeiden Sie Differenzen ähnlich großer Zahlen
- Prüfen Sie auf Über- und Unterlauf
- Implementieren Sie Fehlerkontrollen
- Dokumentieren Sie numerische Einschränkungen

**Robuste Implementation** Quadratische Gleichung:

```
1 # Numerisch stabile Implementation
2 def solve_quadratic_stable(a, b, c):
3     if a == 0:
4         return [-c/b] if b != 0 else []
5     d = b**2 - 4*a*c
6     if d < 0:
7         return []
8     if b >= 0:
9         q = -0.5*(b + d**0.5)
10    else:
11        q = -0.5*(b - d**0.5)
12    x1 = q/a
13    x2 = c/q
14    return sorted([x1, x2])
```

## 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  $\bar{x}$ , für die  $F(\bar{x}) = \bar{x}$  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  $\bar{x} \in [a, b]$  ein Fixpunkt von  $F$ . Dann gilt für die Fixpunktiteration  $x_{n+1} = F(x_n)$ :

**Anziehender Fixpunkt:**

$$|F'(\bar{x})| < 1$$

$x_n$  konvergiert gegen  $\bar{x}$ ,  
falls  $x_0$  nahe genug bei  $\bar{x}$

**Abstossender Fixpunkt:**

$$|F'(\bar{x})| > 1$$

$x_n$  konvergiert für keinen  
Startwert  $x_0 \neq \bar{x}$

**Banachscher Fixpunktsatz**  $F : [a, b] \rightarrow [a, b]$  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:

**Fehlerabschätzungen:**

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

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

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

### Konvergenznachweis für Fixpunktiteration

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

**Fixpunktiteration** Nullstellen von  $f(x) = e^x - x - 2$

Umformung in Fixpunktform:  $x = \ln(x + 2)$ , also  $F(x) = \ln(x + 2)$

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

## Newton-Verfahren

### 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

1. Funktion  $f(x)$  und Ableitung  $f'(x)$  aufstellen
2. Geeigneten Startwert  $x_0$  nahe der Nullstelle wählen
  - Prüfen, ob  $f'(x_0) \neq 0$
3. Iterieren bis zur gewünschten Genauigkeit:  $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$
4. 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  $x_0 = 1$

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

### Vereinfachtes Newton-Verfahren

Alternative Variante mit

konstanter Ableitung:

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

Konvergiert langsamer, aber benötigt weniger Rechenaufwand.

### Sekantenverfahren

Alternative zum Newton-Verfahren ohne Ableitungsberechnung.

Verwendet zwei Punkte  $(x_{n-1}, f(x_{n-1}))$  und  $(x_n, f(x_n))$ :

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

Benötigt zwei Startwerte  $x_0$  und  $x_1$ .

**Sekantenverfahren** Nullstellen von  $f(x) = x^2 - 2$

Startwerte  $x_0 = 1$  und  $x_1 = 2$

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

## Fehlerabschätzung

### Fehlerabschätzung für Nullstellen

So schätzen Sie den Fehler einer Näherungslösung ab:

1. Sei  $x_n$  der aktuelle Näherungswert
2. Wähle Toleranz  $\epsilon > 0$
3. Prüfe Vorzeichenwechsel:  $f(x_n - \epsilon) \cdot f(x_n + \epsilon) < 0$
4. Falls ja: Nullstelle liegt in  $(x_n - \epsilon, x_n + \epsilon)$
5. Damit gilt:  $|x_n - \xi| < \epsilon$

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

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

## Konvergenzverhalten

**Konvergenzordnung** Sei  $(x_n)$  eine gegen  $\bar{x}$  konvergente 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

## Implementationen

### Fixpunktiteration

```
1 # Einfache Version
2 def fixed_point_iter(f, x0, tol=1e-6, max_iter=100):
3     x = x0
4     for i in range(max_iter):
5         x_new = f(x)
6         if abs(x_new - x) < tol:
7             return x_new, i+1
8     x = x_new
9     raise ValueError("Keine Konvergenz")
10
11 # Optimierte Version mit Fehlerschaetzung
12 def fixed_point_iter_opt(f, x0, tol=1e-6, max_iter=100):
13     x = x0
14     alpha = None # Schaetzung fuer Lipschitz-Konstante
15     for i in range(max_iter):
16         x_new = f(x)
17         dx = abs(x_new - x)
18
19         # Lipschitz-Konstante schaeetzen
20         if i > 0 and dx > 0:
21             alpha_new = dx / dx_old
22             if alpha is None or alpha_new > alpha:
23                 alpha = alpha_new
24
25         # A-posteriori Fehlerabschaetzung
26         if alpha is not None and alpha < 1:
27             error = alpha * dx / (1 - alpha)
28             if error < tol:
29                 return x_new, i+1
30
31     x = x_new
32     dx_old = dx
33
34     raise ValueError("Keine Konvergenz")
```

## Newton-Verfahren

```
1 # Einfache Version
2 def newton(f, df, x0, tol=1e-6, max_iter=100):
3     x = x0
4     for i in range(max_iter):
5         fx = f(x)
6         if abs(fx) < tol:
7             return x, i+1
8         dfx = df(x)
9         if dfx == 0:
10             raise ValueError("Ableitung Null")
11         x = x - fx/dfx
12     raise ValueError("Keine Konvergenz")
13
14 # Optimierte Version mit Fehlerkontrolle
15 def newton_safe(f, df, x0, tol=1e-6, max_iter=100):
16     x = x0
17     fx = f(x)
18
19     for i in range(max_iter):
20         dfx = df(x)
21         if dfx == 0:
22             raise ValueError("Ableitung Null")
23
24         dx = fx/dfx
25         x_new = x - dx
26         fx_new = f(x_new)
27
28         # Verschiedene Konvergenzkriterien
29         if abs(fx_new) < tol: # Funktionswert
30             return x_new, i+1
31         if abs(dx) < tol * (1 + abs(x)): # Relative
32             # Aenderung
33             return x_new, i+1
34         if abs(fx_new) >= abs(fx): # Divergenzcheck
35             raise ValueError("Divergenz detektiert")
36
37         x, fx = x_new, fx_new
38
39     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")
```

## Fehlerabschätzung und Abbruchkriterien

```
1 def error_estimate(f, x, eps=1e-5):
2     """
3     Schaetzt den Fehler durch Vorzeichenwechsel-Test
4     Returns: Fehlerschranke eps wenn
5             Vorzeichenwechsel, sonst None
6     """
7     fx_left = f(x - eps)
8     fx_right = f(x + eps)
9
10    if fx_left * fx_right < 0:
11        return eps # Nullstelle liegt in (x-eps,
12            x+eps)
13    return None
14
15 def convergence_criteria(x_new, x_old, f_new, f_old,
16     tol=1e-6):
17     """
18     Prueft verschiedene Abbruchkriterien
19     Returns: (bool, str) - (konvergiert?, grund)
20     """
21     # Absoluter Fehler im Funktionswert
22     if abs(f_new) < tol:
23         return True, "Funktionswert < tol"
24
25     # Relative Aenderung der x-Werte
26     if abs(x_new - x_old) < tol * (1 + abs(x_new)):
27         return True, "Relative Aenderung < tol"
28
29     # Relative Aenderung der Funktionswerte
30     if abs(f_new - f_old) < tol * (1 + abs(f_new)):
31         return True, "Funktionsaenderung < tol"
32
33     # Divergenzcheck
34     if abs(f_new) > 2 * abs(f_old):
35         return False, "Divergenz detektiert"
36
37     return False, "Noch nicht konvergiert"
38
39 def root_finder_with_error(f, x0, tol=1e-6,
40     max_iter=100):
41     """
42     Nullstellensuche mit verschiedenen
43     Konvergenzkriterien
44     und Fehlerabschaetzung
45     """
46     x_old = x0
47     f_old = f(x_old)
48
49     for i in range(max_iter):
50         # Iterationsschritt (hier Newton als Beispiel)
51         x_new = x_old - f_old/derivative(f, x_old)
52         f_new = f(x_new)
53
54         # Pruefe Konvergenzkriterien
55         converged, reason = convergence_criteria(
56             x_new, x_old, f_new, f_old, tol)
57
58         if converged:
59             # Schaetze finalen Fehler
60             error = error_estimate(f, x_new, tol)
61             return {
62                 'root': x_new,
63                 'iterations': i+1,
64                 'error_bound': error,
65                 'convergence_reason': reason
66             }
67
68     x_old, f_old = x_new, f_new
69
70     raise ValueError(f"Keine Konvergenz nach
```

## LGS und Matrizen

### Matrizen

#### Matrix, Element, Zeilen, Spalten und Typ

Eine *Matrix* ist (simpler gesagt) ein Vektor mit mehreren Spalten und wird mit Grossbuchstaben bezeichnet. Ein *Element*  $a_{ij}$  ist ein Wert aus dieser Matrix, auf den über die Zeile und Spalte zugegriffen wird (**Zeile** zuerst, **Spalte** später). Der einer Matrix ergibt sich aus der Anzahl ihren Zeilen und Spalten. Matrizen mit  $m$ -Zeilen und  $n$ -Spalten werden  $m \times n$ -Matrizen genannt.

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

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

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

#### Addition und Subtraktion

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

#### Skalarmultiplikation

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

#### Rechenregeln für die Addition und skalare Multiplikation von Matrizen

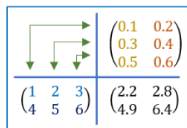
- Kommutativ-Gesetz:  $A + B = B + A$
- Assoziativ-Gesetz:  $A + (B + C) = (A + B) + C$
- Distributiv-Gesetz:  
 $\lambda \cdot (A + B) = \lambda \cdot A + \lambda \cdot B$  sowie  $(\lambda + \mu) \cdot A = \lambda \cdot A + \mu \cdot A$

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

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

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

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



#### Rechenregeln für die Multiplikation von Matrizen

- Assoziativ-Gesetz:  $A \cdot (B \cdot C) = (A \cdot B) \cdot C$
- Distributiv-Gesetz:  
 $A \cdot (B + C) = A \cdot B + A \cdot C$  und  $(A + B) \cdot C = A \cdot C + B \cdot C$
- Skalar-Koeffizient:  $(\lambda \cdot A) \cdot B = \lambda \cdot (A \cdot B) = A \cdot (\lambda \cdot B)$

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

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

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

#### Spezielle Matrizen

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

## Lineare Gleichungssysteme (LGS)

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

$A$ : Koeffizientenmatrix

$\vec{x}$ : Vektor der Unbekannten

$\vec{b}$ : Vektor der Konstanten

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

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

$\Rightarrow$  Anzahl linear unabhängiger Zeilen- oder Spaltenvektoren

#### Zeilenstufenform (Gauss)

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

#### Reduzierte Zeilenstufenform: (Gauss-Jordan)

Alle Zahlen links und rechts der führenden Einsen sind Nullen.

#### Gauss-Jordan-Verfahren

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

**Zeilenoperationen** erlaubt bei LGS (z.B. Gauss-Verfahren)

- Vertauschen von Zeilen
- Multiplikation einer Zeile mit einem Skalar
- Addition eines Vielfachen einer Zeile zu einer anderen

#### Lösbarkeit von linearen Gleichungssystemen

- Lösbar:  $rg(A) = rg(A|b)$       • unendlich viele Lösungen:
- genau eine Lösung:  $rg(A) = n$        $rg(A) < n$

**Parameterdarstellung** bei unendlich vielen Lösungen

Führende Unbekannte: Spalte mit führender Eins

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:

- $\det(A) \neq 0$
- Spalten von  $A$  sind linear unabhängig.
- $rg(A) = n$
- Zeilen von  $A$  sind linear unabhängig.
- $A$  ist invertierbar
- LGS  $A \cdot \vec{x} = \vec{0}$  hat eindeutige Lösung  $x = A^{-1} \cdot \vec{0} = \vec{0}$

## Quadratische Matrizen

**Umformen** bestimme die Matrix  $X$ :  $A \cdot X + B = 2 \cdot X$

$$\Rightarrow A \cdot X = 2 \cdot X - B \Rightarrow A \cdot X - 2 \cdot X = -B \Rightarrow (A - 2 \cdot E) \cdot X = -B$$

$$\Rightarrow (A - 2 \cdot E) \cdot (A - 2 \cdot E)^{-1} \cdot X = (A - 2 \cdot E)^{-1} \cdot -B$$

$$\Rightarrow X = (A - 2 \cdot E)^{-1} \cdot -B$$

### Inverse

**Inverse einer quadratischen Matrix  $A$**   $A^{-1}$

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

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

#### Eigenschaften invertierbarer Matrizen

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

**Inverse einer  $2 \times 2$ -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:

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



## Numerische Lösung linearer Gleichungssysteme

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

Für die Vertauschung der  $i$ -ten

und  $j$ -ten Zeile hat  $P_k$  die **Form**:

- Wichtige Eigenschaften:**
- $p_{ii} = p_{jj} = 0$
  - $p_{ij} = p_{ji} = 1$
  - Sonst gleich wie in  $E_n$
  - $P^{-1} = P^T = P$
  - 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:

#### Rückwärtseinsetzen:

$$\left( \begin{array}{ccc|c} 2 & 4 & -2 & 2 \\ 0 & 3 & 15 & 36 \\ 0 & 1 & 1 & 4 \end{array} \right) \Rightarrow \left( \begin{array}{ccc|c} 2 & 4 & -2 & 2 \\ 0 & 3 & 15 & 36 \\ 0 & 0 & -2 & -8 \end{array} \right) \begin{array}{l} x_3 = \frac{-8}{-2} = 4 \\ x_2 = \frac{36 - 15(4)}{-2} = 1 \\ x_1 = \frac{2 - 4(4) + 2}{2} = -6 \end{array}$$

### Vorteile der Permutationsmatrix

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

### Zeilenvertauschungen verfolgen

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

## Gauss-Algorithmus mit Pivotisierung

```
1 def gauss_elimination(A, b):
2     n = len(b)
3     for i in range(n-1):
4
5         # Pivotisierung
6         k = np.argmax(abs(A[i:, i])) + i
7         if A[k, i] == 0:
8             raise ValueError("Matrix ist singulaer")
9         A[[i, k]] = A[[k, i]]
10        b[[i, k]] = b[[k, i]]
11
12        # Elimination
13        for j in range(i+1, n):
14            factor = A[j, i] / A[i, i]
15            A[j, i:] -= factor * A[i, i:]
16            b[j] -= factor * b[i]
17
18        # Rueckwaertseinsetzen
19        x = np.zeros(n)
20        for i in range(n-1, -1, -1):
21            x[i] = (b[i] - np.dot(A[i, i+1:], x[i+1:])) / A[i, i]
22
23    return x
```

**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:  $x_1 = 1, x_2 = 1$

## Matrix-Zerlegungen

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

**Untere Dreiecksmatrix L:**

$l_{ij} = 0$  für  $j > i$

Diagonale normiert ( $l_{ii} = 1$ )

**Obere Dreiecksmatrix R:**

$r_{ij} = 0$  für  $i > j$

Diagonalelemente  $\neq 0$

## Auswahl des Lösungsverfahrens

### 1. Analyse der Matrix:

- Dimensionen ( $n \times n$ )
- Struktur (dicht/dünn besetzt)
- Konditionszahl (falls berechenbar)

### 2. Direkte Verfahren wenn:

- Matrix dicht besetzt und  $n < 1000$
- Hohe Genauigkeit gefordert
- Mehrere rechte Seiten zu lösen

### 3. Iterative Verfahren wenn:

- Matrix dünn besetzt
- Matrix sehr groß ( $n > 1000$ )
- Moderate Genauigkeit ausreichend
- Matrix diagonaldominant

### 4. Empfohlene Methoden:

- Standard: LR mit Pivotisierung
- Symmetrisch positiv definit: QR
- Große dünn besetzte Systeme: Gauss-Seidel
- Schlecht konditioniert: QR

## LR-Zerlegung

### LR-Zerlegung mit Pivotisierung

```
1 def lr_decomposition_with_pivoting(A):
2     n = len(A)
3     P = np.eye(n) # Permutationsmatrix
4     L = np.eye(n) # Untere Dreiecksmatrix
5     R = A.copy() # Wird zur oberen Dreiecksmatrix
6     for k in range(n-1):
7
8         # Finde Pivotelement
9         pivot = np.argmax(abs(R[k:,k])) + k
10        if pivot != k:
11
12            # Erzeuge Permutationsmatrix
13            P_k = np.eye(n)
14            P_k[[k,pivot]] = P_k[[pivot,k]]
15
16            # Aktualisiere Matrizen
17            P = P_k @ P
18            R[[k,pivot]] = R[[pivot,k]]
19            if k > 0:
20                L[[k,pivot], :k] = L[[pivot,k], :k]
21
22        # Elimination durchfuehren
23        for i in range(k+1, n):
24            factor = R[i,k] / R[k,k]
25            L[i,k] = factor
26            R[i,k:] -= factor * R[k,k:]
27
28    return P, L, R
```

## LR-Zerlegung Implementation

```
1 # Einfache Version ohne externe Bibliotheken
2 def lr_decomposition(A):
3     n = len(A)
4     # Kopiere A um Original nicht zu veraendern
5     R = [[A[i][j] for j in range(n)] for i in range(n)]
6     L = [[1.0 if i == j else 0.0 for j in range(n)]
7           for i in range(n)]
8     P = [[1.0 if i == j else 0.0 for j in range(n)]
9           for i in range(n)]
10
11     for k in range(n-1):
12         # Pivotisierung
13         pivot = k
14         for i in range(k+1, n):
15             if abs(R[i][k]) > abs(R[pivot][k]):
16                 pivot = i
17
18         if abs(R[pivot][k]) < 1e-10: # Numerische Null
19             raise ValueError("Matrix ist (fast)
20                                 singulaer")
21
22         # Zeilenvertauschung falls noetig
23         if pivot != k:
24             R[k], R[pivot] = R[pivot], R[k]
25             # L und P anpassen fuer Zeilen < k
26             for j in range(k):
27                 L[k][j], L[pivot][j] = L[pivot][j],
28                 L[k][j]
29             P[k], P[pivot] = P[pivot], P[k]
30
31         # Elimination
32         for i in range(k+1, n):
33             factor = R[i][k] / R[k][k]
34             L[i][k] = factor
35             for j in range(k, n):
36                 R[i][j] -= factor * R[k][j]
37
38     return P, L, R
39
40 # Optimierte Version mit NumPy
41 def lr_decomposition_numpy(A):
42     n = len(A)
43     R = np.array(A, dtype=float)
44     L = np.eye(n)
45     P = np.eye(n)
46
47     for k in range(n-1):
48         # Pivotisierung
49         pivot = np.argmax(abs(R[k:,k])) + k
50
51         if abs(R[pivot,k]) < 1e-10:
52             raise ValueError("Matrix ist (fast)
53                                 singulaer")
54
55         if pivot != k:
56             # Zeilenvertauschung
57             R[[k,pivot]] = R[[pivot,k]]
58             L[[k,pivot], :k] = L[[pivot,k], :k]
59             P[[k,pivot]] = P[[pivot,k]]
60
61         # Elimination
62         L[k+1:,k] = R[k+1:,k] / R[k,k]
63         R[k+1:] -= np.outer(L[k+1:,k], R[k])
64
65     return P, L, R
```

## LR-Zerlegung

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

### Berechnung der LR-Zerlegung

So berechnen Sie die LR-Zerlegung:

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

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

#### Schritt 1: Erste Spalte

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

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

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

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

#### Schritt 2: Zweite Spalte

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

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

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

#### Endergebnis

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

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

#### Lösung des Systems

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

#### Probe

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

## LR-Zerlegung - Praktisches Vorgehen

1. Voraussetzungen prüfen
  - Matrix regulär?
  - Diagonalelemente ungleich Null?
  - Pivotisierung nötig?
2. Zerlegung durchführen
  - Matrix kopieren für  $R$
  - $L$  als Einheitsmatrix initialisieren
  - $P$  als Einheitsmatrix initialisieren (falls Pivotisierung)
3. Für jede Spalte  $k = 1, \dots, n-1$ :
  - Falls Pivotisierung: Größtes Element in Spalte  $k$  finden
  - Zeilenvertauschung in  $R$  und  $P$  dokumentieren
  - Eliminationsfaktoren  $l_{ik} = \frac{r_{ik}}{r_{kk}}$  berechnen
  - Zeile  $i$  von Zeile  $k$  subtrahieren:  $r_{ij} := r_{ij} - l_{ik}r_{kj}$
  - Eliminationsfaktoren in  $L$  speichern
4. System lösen durch
  - Vorwärtseinsetzen:  $Ly = Pb$
  - Rückwärtseinsetzen:  $Rx = y$

**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:

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

**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**

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

**QR-Zerlegung**

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

**QR-Zerlegung mit Householder**

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

**QR-Zerlegung - 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** Gegeben sei die Matrix:

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

**1. Erste Spalte**

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

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

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

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  $H_1$ :

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

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

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)
- $QR = A$  (bis auf Rundungsfehler)
- $R$  ist obere Dreiecksmatrix

## QR-Zerlegung Implementation

```
1 def householder_vector(x):
2     # Berechne Householder-Vektor fuer Spalte x
3     alpha = np.linalg.norm(x)
4     v = x.copy()
5     v[0] += np.sign(x[0]) * alpha
6     v = v / np.linalg.norm(v)
7     return v
8
9 def householder_reflection(A, k):
10     m, n = A.shape
11     v = householder_vector(A[k:, k])
12     # Householder-Matrix anwenden
13     H = np.eye(m-k)
14     H -= 2 * np.outer(v, v)
15     # Auf Untermatrix anwenden
16     A[k:, k:] = H @ A[k:, k:]
17     return A
18
19 def qr_householder(A):
20     m, n = A.shape
21     R = A.copy()
22     Q = np.eye(m)
23
24     for k in range(n):
25         v = householder_vector(R[k:, k])
26         H = np.eye(m)
27         H[k:, k:] -= 2 * np.outer(v, v)
28         R = H @ R
29         Q = Q @ H.T
30
31     return Q, R
```

### Numerische Vorteile

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

## QR-Zerlegung Implementation

```
1 # Einfache Version ohne externe Bibliotheken
2 def qr_decomposition(A):
3     """
4     Berechnet QR-Zerlegung einer Matrix A ohne NumPy.
5     Returns: Q (orthogonal) und R (obere
6             Dreiecksmatrix)
7     """
8     m = len(A)
9     n = len(A[0])
10
11     # Kopiere A nach R (deep copy)
12     R = [[A[i][j] for j in range(n)] for i in range(m)]
13     # Initialisiere Q als Einheitsmatrix
14     Q = [[1.0 if i == j else 0.0 for j in range(m)]
15           for i in range(m)]
16
17     def vector_norm(v):
18         """Euklidische Norm eines Vektors"""
19         return (sum(x*x for x in v)) ** 0.5
20
21     def matrix_mult(A, B):
22         """Matrix Multiplikation"""
23         m, n = len(A), len(B[0])
24         p = len(B)
25         C = [[0.0] * n for _ in range(m)]
26         for i in range(m):
27             for j in range(n):
28                 C[i][j] = sum(A[i][k] * B[k][j]
29                               for k in range(p))
30         return C
31
32     def householder_reflection(x):
33         """
34         Berechnet Householder-Vektor und Beta fuer
35         einen Vektor x.
36         Returns: v (Householder-Vektor) und beta
37         """
38         n = len(x)
39         v = [xi for xi in x] # Kopiere x
40
41         # Berechne Norm des Teilvektors
42         sigma = sum(v[i]*v[i] for i in range(1, n))
43
44         if sigma == 0 and x[0] >= 0:
45             beta = 0
46         elif sigma == 0 and x[0] < 0:
47             beta = -2
48         else:
49             mu = (x[0]*x[0] + sigma)**0.5
50             if x[0] <= 0:
51                 v[0] = x[0] - mu
52             else:
53                 v[0] = -sigma/(x[0] + mu)
54
55             beta = 2*v[0]*v[0]/(sigma + v[0]*v[0])
56             # Normiere v
57             temp = v[0]
58             for i in range(n):
59                 v[i] /= temp
60
61         return v, beta
62
63     # Hauptschleife der QR-Zerlegung
64     for k in range(n):
65         # Extrahiere k-te Spalte ab k-ter Zeile
66         x = [R[i][k] for i in range(k, m)]
67         if len(x) > 1: # Nur wenn noch Untermatrix
68             # Berechne Householder-Transformation
69             v, beta = householder_reflection(x)
```

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

### Schritt 1: Erste Spalte

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



Householder-Vektor für erste Spalte:

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

Householder-Matrix berechnen:

$$H_1 = I - 2u_1u_1^T = \begin{pmatrix} -0.67 & -0.75 & 0 \\ -0.75 & 0.67 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

A nach erster Transformation:

$$A^{(1)} = H_1A = \begin{pmatrix} -\sqrt{5} & -6.71 & 0.45 \\ 0 & -0.89 & 1.79 \\ 0 & 2.00 & 1.00 \end{pmatrix}$$

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:  $R = Q_2A^{(1)} = \begin{pmatrix} -\sqrt{5} & -6.71 & 0.45 \\ 0 & -2.19 & 1.34 \\ 0 & 0 & -1.79 \end{pmatrix}$

Endergebnis

Die QR-Zerlegung  $A = QR$  ist:

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

Probe

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

Wichtige Beobachtungen

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

Fehleranalyse

Matrix- und Vektornormen

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

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

Wichtige Normen

1-Norm:

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

2-Norm:

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

$\infty$ -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:

Relativer Fehler:

$$\|x - \tilde{x}\| \leq \|A^{-1}\| \cdot \|b - \tilde{b}\| \quad \frac{\|x - \tilde{x}\|}{\|x\|} \leq \text{cond}(A) \cdot \frac{\|b - \tilde{b}\|}{\|b\|}$$

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

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

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

Fehlerabschätzung

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

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

Fehlerabschätzung

```
1 def error_estimate(A, b, x, b_tilde):
2     # Absoluter Fehler
3     abs_error = np.linalg.norm(x - np.linalg.solve(A,
4     b_tilde))
5     # Relativer Fehler
6     rel_error = abs_error / np.linalg.norm(x)
7     return abs_error, rel_error
```

Fehleranalyse in der Praxis

1. Analyse der Eingangsdaten
  - Konditionszahl der Matrix bestimmen
  - Struktur der Matrix untersuchen
  - Größenordnung der Einträge prüfen
2. Fehlerquellen identifizieren
  - Rundungsfehler bei der Eingabe
  - Fehler bei der Elimination
  - Akkumulation von Rundungsfehlern
  - Instabilitäten durch schlechte Kondition
3. Fehlerabschätzungen berechnen
  - A-priori:  $\frac{\|\Delta x\|}{\|x\|} \leq \text{cond}(A) \cdot \frac{\|\Delta b\|}{\|b\|}$
  - A-posteriori:  $\|Ax - b\|$  (Residuum)
  - Iterative Verfahren: Konvergenzrate
4. Maßnahmen zur Fehlerreduktion
  - Pivotisierung verwenden
  - Skalierung der Matrix
  - Höhere Genauigkeit verwenden
  - Alternatives Lösungsverfahren wählen

## Fehleranalyse Implementation

```
1 def analyze_matrix(A, b):
2     """Analysiert ein LGS auf numerische Probleme"""
3     n = len(A)
4
5     # 1. Grundlegende Eigenschaften
6     diag_dom = is_diagonally_dominant(A)
7     scaling = max(abs(A[i][j]) for i in range(n)
8                   for j in range(n))
9
10    # 2. Konditionszahl schätzen (ohne numpy)
11    def matrix_norm_inf(M):
12        return max(sum(abs(M[i][j]) for j in
13                      range(len(M)))
14                  for i in range(len(M)))
15
16    def inverse_power_iteration(M, max_iter=100):
17        x = [1.0] * n
18        for _ in range(max_iter):
19            y = solve_triangular(M, x)
20            norm = max(abs(yi) for yi in y)
21            x = [yi/norm for yi in y]
22        return 1.0/norm
23
24    norm_A = matrix_norm_inf(A)
25    try:
26        norm_Ainv = inverse_power_iteration(A)
27        cond = norm_A * norm_Ainv
28    except:
29        cond = float('inf')
30
31    # 3. Analyse der Diagonalelemente
32    min_diag = min(abs(A[i][i]) for i in range(n))
33    max_offdiag = max(abs(A[i][j]) for i in range(n)
34                     for j in range(n) if i != j)
35
36    # 4. Empfehlungen generieren
37    recommendations = []
38    if not diag_dom:
39        recommendations.append(
40            "Matrix nicht diagonaldominant - "
41            "Iterative Verfahren koennten divergieren")
42
43    if cond > 1e4:
44        recommendations.append(
45            f"Hohe Konditionszahl ({cond:.1e}) - "
46            "Ergebnisse koennten ungenau sein")
47
48    if min_diag < max_offdiag/100:
49        recommendations.append(
50            "Kleine Diagonalelemente - "
51            "Pivotisierung empfohlen")
52
53    if scaling > 1e8:
54        recommendations.append(
55            "Grosse Zahlenunterschiede - "
56            "Skalierung empfohlen")
57
58    return {
59        'condition_number': cond,
60        'is_diagonally_dominant': diag_dom,
61        'scaling_factor': scaling,
62        'min_diagonal': min_diag,
63        'max_offdiagonal': max_offdiag,
64        'recommendations': recommendations
65    }
66
67 def error_analysis(A, x, b, x_approx):
68     """Analysiert die Genauigkeit einer
69     Naeherungsloesung"""
70     n = len(A)
```

Vergleich verschiedener Lösungsverfahren Betrachten Sie das folgende System:

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

### 1. Systemanalyse

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

### 2. Verschiedene Lösungsansätze

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

### 3. Interpretation

- Direkte Verfahren erreichen höhere Genauigkeit
- LR schneller als QR bei moderater Kondition
- Iterative Verfahren brauchen mehrere Schritte
- Gauss-Seidel konvergiert schneller als Jacobi

## Iterative Verfahren

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

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

**Jacobi-Verfahren** Gesamtschrittverfahren mit der Iteration:

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

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

**Gauss-Seidel-Verfahren** Einzelschrittverfahren mit der Iteration:

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

Komponentenweise:

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

**Konvergenzkriterien** Ein iteratives Verfahren konvergiert, wenn:

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

### Implementation iterativer Verfahren

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

Iterative Verfahren Vergleich Jacobi und Gauss-Seidel System:

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

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

### Jacobi- und Gauss-Seidel-Verfahren

```
1 def jacobi_iteration(A, b, x):
2     D = np.diag(np.diag(A))
3     L = np.tril(A, -1)
4     R = np.triu(A, 1)
5     x_new = np.linalg.solve(D, b - (L + R) @ x)
6     return x_new
7
8 def gauss_seidel_iteration(A, b, x):
9     D = np.diag(np.diag(A))
10    L = np.tril(A, -1)
11    R = np.triu(A, 1)
12    x_new = np.linalg.solve(D + L, b - R @ x)
13    return x_new
```

Iterative Verfahren Implementation

```
1 # Einfache Version ohne externe Bibliotheken
2 def jacobi_method(A, b, x0, tol=1e-6, max_iter=100):
3     n = len(A)
4     x = x0.copy()
5     x_new = [0.0] * n
6
7     for iter in range(max_iter):
8         # Jacobi-Iteration
9         for i in range(n):
10             sum1 = sum(A[i][j] * x[j] for j in
11                        range(i))
12             sum2 = sum(A[i][j] * x[j] for j in
13                      range(i+1, n))
14             x_new[i] = (b[i] - sum1 - sum2) / A[i][i]
15
16         # Konvergenzpruefung
17         diff = max(abs(x_new[i] - x[i]) for i in
18                   range(n))
19         if diff < tol:
20             return x_new, iter + 1
21         x = x_new.copy()
22
23     raise ValueError(f"Keine Konvergenz nach
24                     {max_iter} Iterationen")
25
26 def gauss_seidel_method(A, b, x0, tol=1e-6,
27                        max_iter=100):
28     n = len(A)
29     x = x0.copy()
30
31     for iter in range(max_iter):
32         x_old = x.copy()
33         # Gauss-Seidel-Iteration
34         for i in range(n):
35             sum1 = sum(A[i][j] * x[j] for j in
36                      range(i))
37             sum2 = sum(A[i][j] * x_old[j] for j in
38                      range(i+1, n))
39             x[i] = (b[i] - sum1 - sum2) / A[i][i]
40
41         # Konvergenzpruefung
42         diff = max(abs(x[i] - x_old[i]) for i in
43                   range(n))
44         if diff < tol:
45             return x, iter + 1
46
47     raise ValueError(f"Keine Konvergenz nach
48                     {max_iter} Iterationen")
49
50 # Optimierte Version mit Konvergenzanalyse
51 def iterative_solver(A, b, method='gauss_seidel',
52                    tol=1e-6,
53                    max_iter=100, omega=1.0):
54     """
55     Loest Ax = b mit verschiedenen iterativen Verfahren
56     method: 'jacobi' oder 'gauss_seidel'
57     omega: Relaxationsparameter (1.0 = standard)
58     """
59     n = len(A)
60     x = [0.0] * n # Startvektor
61     D = [[A[i][j] if i == j else 0 for j in range(n)]
62          for i in range(n)] # Diagonalmatrix
63     L = [[A[i][j] if i > j else 0 for j in range(n)]
64          for i in range(n)] # Untere Dreiecksmatrix
65     U = [[A[i][j] if i < j else 0 for j in range(n)]
66          for i in range(n)] # Obere Dreiecksmatrix
67
68     # Konvergenzcheck
69     if not is_diagonally_dominant(A):
70         print("Warnung: Matrix nicht diagonaldominant")
```

Konvergenzverhalten Betrachten Sie das System:

$$\begin{pmatrix} 4 & 1 & 0 \\ 1 & 4 & 1 \\ 0 & 1 & 4 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

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)

Eigenwerte und Eigenvektoren

Komplexe Zahlen

Fundamentalsatz der Algebra

Eine algebraische Gleichung n-ten Grades mit komplexen Koeffizienten:

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

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:  $\text{Re}(z) = x$

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

Imaginärteil:  $\text{Im}(z) = y$

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

Rechenoperationen mit komplexen Zahlen

Für  $z_1 = x_1 + iy_1$  und  $z_2 = x_2 + iy_2$  gilt:

Addition:

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

Subtraktion:

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

Multiplikation:

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

Division:

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

Potenzen und Wurzeln

Für eine komplexe Zahl in Exponentialform  $z = r e^{i\varphi}$  gilt:

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

## Darstellungsformen

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

$$x = r \cos \varphi, \quad y = r \sin \varphi, \quad r = \sqrt{x^2 + y^2}$$

$$\varphi = \arcsin\left(\frac{y}{r}\right) = \arccos\left(\frac{x}{r}\right)$$

$$e^{i\varphi} = \cos \varphi + i \sin \varphi \text{ (Euler-Formel)}$$

## Umrechnung zwischen Darstellungsformen komplexer Zahlen

Von Normalform in trigonometrische Form/Exponentialform

1. Berechne Betrag  $r = \sqrt{x^2 + y^2}$
2. Berechne Winkel mit einer der Formeln:
  - $\varphi = \arctan\left(\frac{y}{x}\right)$  falls  $x > 0$
  - $\varphi = \arctan\left(\frac{y}{x}\right) + \pi$  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:  $z = x + iy$

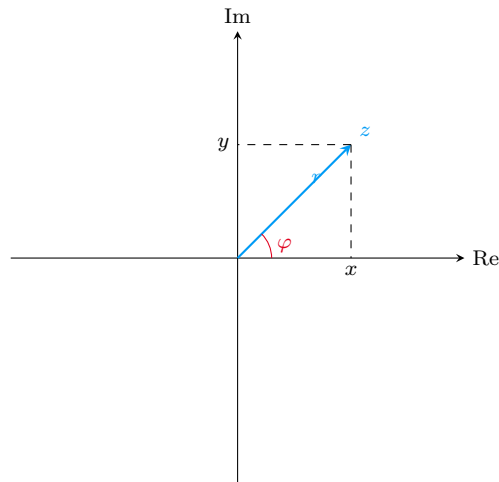
Von Exponentialform in Normalform/trigonometrische Form

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

## Umrechnung zwischen Darstellungsformen komplexer Zahlen [Previous content stays the same]



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

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

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

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

Darstellungsformen

Aufgabe:

Wandle  $z = 3 - 11i$  in trigonometrische Form und Exponentialform um.

Lösung:

1. Berechne Betrag:

$$r = \sqrt{3^2 + (-11)^2} = \sqrt{130}$$

2. Berechne Winkel:

$$\varphi = \arctan\left(\frac{-11}{3}\right) + \pi = 1.3 \text{ rad} = 74.74^\circ$$

Da  $x = 3 > 0$  und  $y = -11 < 0$  ist im 4. Quadranten

3. Darstellung:

- Trigonometrisch:  $z = \sqrt{130}(\cos(1.3) - i \sin(1.3))$
- Exponentialform:  $z = \sqrt{130}e^{-i \cdot 1.3}$

## Komplexe Zahlen in Python

```
1 import numpy as np
2 z1 = 3 - 11j
3 z2 = 2 + 5j
4 # Addition
5 z_add = z1 + z2
6 # Subtraktion
7 z_sub = z1 - z2
8 # Multiplikation
9 z_mul = z1 * z2
10 # Division
11 z_div = z1 / z2
12 # Betrag
13 r = np.abs(z1)
14 # Winkel
15 phi = np.angle(z1)
16 # Exponentialform
17 z_exp = r * np.exp(1j * phi)
18 # Potenz
19 z_pow = z1 ** 2
20 # Wurzel
21 z_sqrt = np.sqrt(z1)
22
23 # Darstellungsformen
24 z_trig = r * (np.cos(phi) + 1j * np.sin(phi))
25 z_norm = z_trig.real + 1j * z_trig.imag
```

## 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)
```

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:

- $\det(A) = \prod_{i=1}^n \lambda_i$  (Produkt der Eigenwerte)
- $\text{tr}(A) = \sum_{i=1}^n \lambda_i$  (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  $= n - \text{rg}(A - \lambda I_n)$
- Die geometrische Vielfachheit ist stets kleiner oder gleich der algebraischen Vielfachheit.

Bestimmung von Eigenwerten und Eigenvektoren

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

Bestimmung von Eigenwerten und Eigenvektoren

1. Charakteristisches Polynom aufstellen:
  - Matrix  $(A - \lambda I_n)$  bilden
  - Determinante berechnen:  $p(\lambda) = \det(A - \lambda I_n)$
2. Eigenwerte bestimmen:
  - $p(\lambda) = 0$  lösen
  - Bei Dreiecksmatrix: Diagonalelemente sind Eigenwerte
3. Für jeden Eigenwert  $\lambda_i$ :
  - $(A - \lambda_i I_n)x = 0$  aufstellen
  - Gaußverfahren anwenden
  - Lösung parametrisieren
  - Parameter  $\neq 0$  wählen für Eigenvektor
4. Überprüfung:
  - $Ax = \lambda x$  für jeden Eigenvektor
  - Anzahl linear unabhängiger Eigenvektoren kleiner als algebraische Vielfachheit

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

EW und EV über Charakteristisches Polynom

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

Numerische Berechnung von Eigenwerten

Ähnliche Matrizen

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

$B = T^{-1}AT$

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

$D = T^{-1}AT$

Eigenschaften ähnlicher Matrizen

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

1.  $A$  und  $B$  haben dieselben Eigenwerte mit gleichen algebraischen Vielfachheiten
2. Ist  $x$  Eigenvektor von  $B$  zum Eigenwert  $\lambda$ , 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 durchführen

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

Von-Mises-Iteration

Von-Mises-Iteration Berechne größten Eigenwert der Matrix:

$$A = \begin{pmatrix} 4 & -1 & 1 \\ -1 & 3 & -2 \\ 1 & -2 & 3 \end{pmatrix}$$

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

Von-Mises-Iteration Berechnung des größten Eigenwerts

```
1 def power_iteration(A, tol=1e-10, max_iter=100):
2     n = len(A)
3     v = [random.random() for _ in range(n)]
4     v = [x / np.linalg.norm(v) for x in v]
5     for i in range(max_iter):
6         w = [sum(A[i][j] * v[j] for j in range(n)) for
7               i in range(n)]
8         v_new = [x / np.linalg.norm(w) for x in w]
9         # Rayleigh-Quotient
10        lambda_k = sum(v_new[i] * A[i][j] * v_new[j]
11                        for i in range(n) for j in range(n))
12        if np.linalg.norm([v_new[i] - v[i] for i in
13                          range(n)]) < tol:
14            return lambda_k, v_new
15        v = v_new
16    return lambda_k, v_new
```



## Von-Mises-Iteration

```

1 def matrix_vector_mult(A, v):
2     n = len(v)
3     return [sum(A[i][j] * v[j] for j in range(n))
4             for i in range(n)]
5
6 def vector_norm(v):
7     return (sum(x*x for x in v)) ** 0.5
8
9 def normalize_vector(v):
10    norm = vector_norm(v)
11    return [x/norm for x in v]
12
13 def rayleigh_quotient(A, v):
14    numerator = sum(v[i] * sum(A[i][j] * v[j]
15                             for j in range(len(v))))
16                for i in range(len(v))
17    denominator = sum(x*x for x in v)
18    return numerator / denominator
19
20 def power_method(A, tol=1e-10, max_iter=100):
21    n = len(A)
22    # Startvektor [1,0,0,...,0]
23    v = [1.0] + [0.0] * (n-1)
24    v = normalize_vector(v)
25
26    for k in range(max_iter):
27        # Matrixmultiplikation
28        w = matrix_vector_mult(A, v)
29        # Normierung
30        v_new = normalize_vector(w)
31        # Eigenwertapproximation
32        lambda_k = rayleigh_quotient(A, v_new)
33        # Konvergenztest
34        if vector_norm([v_new[i]-v[i]
35                       for i in range(n)]) < tol:
36            return lambda_k, v_new
37        v = v_new
38
39    return lambda_k, v

```

## Von-Mises-Iteration ohne spezielle Bibliotheken

```

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

### 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 anwenden

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

## QR-Verfahren

## QR-Verfahren

```

1 def qr_algorithm(A, tol=1e-10, max_iter=100):
2     n = A.shape[0]
3     Q_prod = np.eye(n)
4     A_k = A.copy()
5
6     for k in range(max_iter):
7         # QR decomposition
8         Q, R = np.linalg.qr(A_k)
9         # New iteration
10        A_k = R @ Q
11        # Update transformation matrix
12        Q_prod = Q_prod @ Q
13
14        # Check convergence
15        if np.abs(np.tril(A_k, -1)).max() < tol:
16            break
17
18    return np.diag(A_k), Q_prod

```

## 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:  $\lambda_1 = 4, \lambda_2 = 1, \lambda_3 = 1$

## 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)]
```

## QR-Verfahren ohne spezielle Bibliotheken

```
1 def matrix_mult(A, B):
2     n = len(A)
3     C = [[0]*n for _ in range(n)]
4     for i in range(n):
5         for j in range(n):
6             for k in range(n):
7                 C[i][j] += A[i][k] * B[k][j]
8     return C
9
10 def gram_schmidt(A):
11     n = len(A)
12     Q = [[0]*n for _ in range(n)]
13     R = [[0]*n for _ in range(n)]
14
15     # Extrahiere Spalten von A
16     V = [[A[i][j] for i in range(n)] for j in range(n)]
17
18     for i in range(n):
19         # Berechne Q[:, i]
20         Q_col = V[i]
21         for j in range(i):
22             # Berechne Projektion und subtrahiere
23             R[j][i] = sum(Q[k][j] * V[i][k] for k in
24                           range(n))
25             Q_col = [Q_col[k] - R[j][i] * Q[k][j]
26                     for k in range(n)]
27
28         # Normiere
29         R[i][i] = vector_norm(Q_col)
30         Q_col = [x/R[i][i] for x in Q_col]
31
32         # Speichere normierte Spalte
33         for j in range(n):
34             Q[j][i] = Q_col[j]
35
36     return Q, R
37
38 def qr_algorithm(A, tol=1e-10, max_iter=100):
39     n = len(A)
40     A_k = [row[:] for row in A] # Kopiere A
41
42     for _ in range(max_iter):
43         Q, R = gram_schmidt(A_k)
44         A_k = matrix_mult(R, Q)
45
46         # Pruefe Konvergenz (Subdiagonalelemente)
47         converged = True
48         for i in range(1, n):
49             if abs(A_k[i][i-1]) > tol:
50                 converged = False
51                 break
52         if converged:
53             break
54
55     return [A_k[i][i] for i in range(n)] # Eigenwerte
```

### Numerische Stabilität

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

## Praktische Anwendungen

### Anwendungen von Eigenwerten

- Bestimmung von Schwingungsmoden in mechanischen Systemen
- Hauptkomponentenanalyse in der Datenanalyse
- Stabilität von dynamischen Systemen
- PageRank-Algorithmus für Webseitenranking
- Bildkompression und Signalverarbeitung

**PageRank-Algorithmus** Google-Matrix für ein kleines Webnetzwerk mit 3 Seiten:

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

Der PageRank entspricht dem Eigenvektor zum Eigenwert 1:

- $\lambda = 1$  ist größter Eigenwert
- PageRank:  $v \approx (0.39, 0.41, 0.20)^T$
- Seite 2 hat höchste Relevanz, Seite 3 niedrigste

### Praktische Eigenwertberechnung

1. Vorverarbeitung der Matrix:
  - Auf Symmetrie prüfen
  - Ggf. auf Hessenberg-Form transformieren
  - Kondition der Matrix prüfen
2. Wahl des geeigneten Verfahrens:
  - Nur größter EW  $\rightarrow$  Von-Mises
  - Alle EW  $\rightarrow$  QR-Verfahren
  - Symmetrisch  $\rightarrow$  Symmetrisches QR
  - Einzelner EW  $\rightarrow$  Inverse Iteration
3. Implementierungsaspekte:
  - Konvergenzkriterien definieren
  - Numerische Stabilität sicherstellen
  - Abbruchkriterien festlegen
  - Fehlerbehandlung implementieren
4. Nachbearbeitung:
  - Genauigkeit überprüfen
  - EV auf Länge 1 normieren
  - Komplexe EW/EV behandeln
  - Ergebnisse validieren

## Inverse Iteration

```
1 def inverse_iteration(A, mu, tol=1e-10, max_iter=100):
2     n = len(A)
3     # Startvektor
4     v = [1.0] + [0.0] * (n-1)
5     v = normalize_vector(v)
6
7     # Matrix (A - mu*I) erstellen
8     A_shift = [[A[i][j] if i != j
9                  else A[i][j] - mu
10                 for j in range(n)]
11                for i in range(n)]
12
13     for k in range(max_iter):
14         # Löse (A - mu*I)w = v
15         w = solve_linear_system(A_shift, v)
16         # Normiere Ergebnisvektor
17         v_new = normalize_vector(w)
18
19         # Konvergenztest
20         if vector_norm([v_new[i]-v[i]
21                         for i in range(n)]) < tol:
22             # Berechne Rayleigh-Quotienten
23             lambda_k = rayleigh_quotient(A, v_new)
24             return lambda_k, v_new
25
26     v = v_new
27
28     raise ValueError("Keine Konvergenz")
29
30 def solve_linear_system(A, b):
31     # LR-Zerlegung und Rueckwaertseinsetzen
32     n = len(A)
33     x = gauss_solve(A, b) # aus LGS Kapitel
34     return x
```

Inverse Iteration Matrix mit bekanntem Eigenwert nahe  $\mu = 2$ :

$$A = \begin{pmatrix} 2.1 & -0.1 & 0.1 \\ -0.1 & 2.0 & -0.2 \\ 0.1 & -0.2 & 1.9 \end{pmatrix}$$

Iterationsverlauf mit  $\mu = 2.0$ :

k	$v^{(k)}$	$\lambda^{(k)}$
0	$(1, 0, 0)^T$	-
1	$(0.82, -0.41, 0.39)^T$	2.091
2	$(0.81, -0.42, 0.41)^T$	2.083
3	$(0.81, -0.42, 0.41)^T$	2.082

Konvergenz gegen den Eigenwert  $\lambda \approx 2.082$

## 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

- Numerische Bibliotheken**    Verwendung spezialisierter Bibliotheken
- Für kritische numerische Berechnungen:
- NumPy: Optimierte Array-Operationen
  - SciPy: Wissenschaftliches Rechnen
  - Mpmath: Beliebige Präzision
  - Decimal: Dezimalarithmetik

**Bibliotheksverwendung**    Beispiel: Präzise Berechnung mit Decimal

```
1 from decimal import Decimal, getcontext
2
3 # Set precision
4 getcontext().prec = 40
5
6 # Precise calculation
7 x = Decimal('1.0') / Decimal('7.0')
8 print(x)    # 0.1428571428571428571428571428571428571428
```

NumPy

- NumPy**    NumPy: Numerische Python-Bibliothek
- Effiziente Implementierung von Arrays
  - Vektorisierte Operationen
  - Lineare Algebra, Fourier-Transformation, Zufallszahlen
- ACHTUNG: darf an der Prüfung höchstwahrscheinlich nicht verwendet werden! aber falls doch, hier die einfachen Implementationen von allem :D

Eigenwerte und Eigenvektoren

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

Von-Mises-Iteration

```
1 import numpy as np
2 def power_iteration(A, tol=1e-10, max_iter=100):
3     n = A.shape[0]
4     v = np.random.rand(n)
5     v = v / np.linalg.norm(v)
6     for i in range(max_iter):
7         w = A @ v
8         v_new = w / np.linalg.norm(w)
9         # Rayleigh-Quotient
10        lambda_k = v_new.T @ A @ v_new
11        if np.linalg.norm(v_new - v) < tol:
12            return lambda_k, v_new
13        v = v_new
14    return lambda_k, v_new
```

Examples

Rechnerarithmetik

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

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

- 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:**

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

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

Fixpunktiteration

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

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

- Ableitung:  $f'(x) = 3x^2$ , Startwert  $x_0 = 1$
1.  $x_1 = 1 - \frac{1^3 - 2}{3 \cdot 1^2} = 1.333333$

2.  $x_2 = 1.333333 - \frac{1.333333^3 - 2}{3 \cdot 1.333333^2} = 1.259921$

3.  $x_3 = 1.259921 - \frac{1.259921^3 - 2}{3 \cdot 1.259921^2} = 1.259921$
- Quadratische Konvergenz sichtbar durch schnelle Annäherung an  $\sqrt[3]{2} \approx 1.259921$