Höhere Mathematik

Jil Zerndt, Lucien Perret
January 2025

Rechnerarithmetik

Zahlendarstellung

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

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

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

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

Werteberechnung einer Maschinenzahl

1. Normalisierung überprüfen: $m_1 \neq 0$ (für $x \neq 0$)
   - Sonst: Mantisse verschieben und Exponent anpassen
2. Exponent berechnen: $\hat{e} = \sum_{i=1}^{l} e_i B^{l-i}$
   - Von links nach rechts: Stelle · Basis hochgestellt zur Position
3. Wert berechnen: $\hat{\omega} = \sum_{i=1}^{n} m_i B^{\hat{e}-i}$
   - Mantissenstellen · Basis hochgestellt zu (Exponent - Position)
4. 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:

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

Darstellungsbereich Für jedes Gleitpunktsystem existieren:

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

Approximations- und Rundungsfehler

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

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

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

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

Allgemein: $\text{eps} := \frac{B}{2} \cdot B^{-n}$ **Dezimal:** $\text{eps}_{10} := 5 \cdot 10^{-n}$

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

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

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

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

Fehlerfortpflanzung

Konditionierung Die Konditionszahl K beschreibt die relative Fehlervergrösserung bei Funktionsauswertungen:

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

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

Fehlerfortpflanzung Für f (differenzierbar) gilt näherungsweise:

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

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

Analyse der Fehlerfortpflanzung einer Funktion

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

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

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

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

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

Praktische Fehlerquellen der Numerik

Kritische Operationen häufigste Fehlerquellen:

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

Vermeidung von Auslöschung

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

Auslöschung Kritische Berechnungen für kleine x (Auslöschung):

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

```python
def fib_unstable(n): # Instabile rekursive Version
    if n <= 1:
        return n
    return fib_unstable(n-1) + fib_unstable(n-2)

def fib_stable(n): # Stabile iterative Version
    if n <= 1:
        return n
    a, b = 0, 1
    for _ in range(2, n + 1):
        a, b = b, a + b
    return b
```

Stabilitätsanalyse Schritte zur Analyse der numerischen Stabilität:

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

Implementierungsgenauigkeit eines Algorithmus

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

Robuste Implementierung von Algorithmen

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

Robuste Implementation Quadratische Gleichung:

```python
# Numerisch stabile Implementation
def solve_quadratic_stable(a, b, c):
    if a == 0:
        return [-c/b] if b != 0 else []
    d = b**2 - 4*a*c
    if d < 0:
        return []
    if b >= 0:
        q = -0.5*(b + d**0.5)
    else:
        q = -0.5*(b - d**0.5)
    x1 = q/a
    x2 = c/q
    return sorted([x1, x2])
```

Numerische Lösung von Nullstellenproblemen

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

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

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

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

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

heisst Fixpunktiteration von F zum Startwert x_0.

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

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

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

Banachscher Fixpunktsatz $F : [a, b] \to [a, b]$ und $\exists$ Konstante α:

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

Dann gilt:

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

Fehlerabschätzungen:

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

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

Konvergenznachweis für Fixpunktiteration

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 α: $\to$ Berechne $F'(x)$
   - Finde $\alpha = \max_{x \in [a,b]} |F'(x)|$ und prüfe $\alpha < 1$
4. Berechnen Sie die nötigen Iterationen für Genauigkeit tol: $n \geq \dfrac{\ln(\frac{tol \cdot (1-\alpha)}{|x_1 - x_0|})}{\ln \alpha}$

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:

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

Konvergiert, wenn für alle x im relevanten Intervall gilt:

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

Newton-Verfahren anwenden

1. Funktion $f(x)$ und Ableitung $f'(x)$ aufstellen
2. Geeigneten Startwert x_0 nahe der Nullstelle wählen
   - 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. $x_1 = 1 - \frac{1^2-2}{2 \cdot 1} = 1.5$
2. $x_2 = 1.5 - \frac{1.5^2-2}{2 \cdot 1.5} = 1.4167$
3. $x_3 = 1.4167 - \frac{1.4167^2-2}{2 \cdot 1.4167} = 1.4142$

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

Vereinfachtes Newton-Verfahren
Alternative Variante mit konstanter Ableitung:

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

Konvergiert langsamer, aber benötigt weniger Rechenaufwand.

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

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

Benötigt zwei Startwerte x_0 und x_1.

Sekantenverfahren Nullstellen von $f(x) = x^2 - 2$
Startwerte $x_0 = 1$ und $x_1 = 2$

1. $x_2 = 1 - \frac{1-2}{1^2-2} \cdot 1 = 1.5$
2. $x_3 = 1.5 - \frac{1.5-1}{1.5^2-2} \cdot 1.5 = 1.4545$
3. $x_4 = 1.4545 - \frac{1.4545-1.5}{1.4545^2-2} \cdot 1.4545 = 1.4143$

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

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 $\epsilon = 10^{-5}$:
3. $f(x_3 - \epsilon) = 1.4142057^2 - 2 < 0$
4. $f(x_3 + \epsilon) = 1.4142257^2 - 2 > 0$

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

$\to$ Nullstelle liegt in $(1.4142057, 1.4142257)$

Konvergenzverhalten

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

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

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

Konvergenzordnungen der Verfahren Konvergenzgeschwindigkeiten

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

Vereinfachtes Newton: Lineare Konvergenz: $q = 1$

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

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

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

Implementationen

Fehlerabschätzung durch Vorzeichenwechsel

```python
def error_estimate(f, x, eps=1e-5):
    fx_left = f(x - eps)
    fx_right = f(x + eps)

    if fx_left * fx_right < 0:
        return eps  # Nullstelle liegt in (x-eps, x+eps)
    return None

    #Returns: Fehlerschranke eps wenn Vorzeichenwechsel, sonst None
```

Verschiedene Abbruchkriterien prüfen Konvergenzkriterien

```python
def convergence_criteria(x_new, x_old, f_new, f_old, tol=1e-6):
    # Absoluter Fehler im Funktionswert
    if abs(f_new) < tol:
        return True, "Funktionswert < tol"

    # Relative Aenderung der x-Werte
    if abs(x_new - x_old) < tol * (1 + abs(x_new)):
        return True, "Relative Aenderung < tol"

    # Relative Aenderung der Funktionswerte
    if abs(f_new - f_old) < tol * (1 + abs(f_new)):
        return True, "Funktionsaenderung < tol"

    # Divergenzcheck
    if abs(f_new) > 2 * abs(f_old):
        return False, "Divergenz detektiert"

    return False, "Noch nicht konvergiert"

    #Returns: (konvergiert?, grund)
```

Fixpunktiteration

```python
def fixed_point_it(f, x0, tol=1e-6, max_it=100):
    x = x0
    for i in range(max_it):
        x_new = f(x)
        if abs(x_new - x) < tol:
            return x_new, i+1
        x = x_new
    raise ValueError("Keine Konvergenz")

# Optimierte Version mit Fehlerschaetzung
def fixed_point_it_opt(f, x0, tol=1e-6, max_it=100):
    x = x0
    alpha = None  # Schaetzung fuer Lipschitz-Konstante
    for i in range(max_iter):
        x_new = f(x)
        dx = abs(x_new - x)
        # Lipschitz-Konstante schaetzen
        if i > 0 and dx > 0:
            alpha_new = dx / dx_old
            if alpha is None or alpha_new > alpha:
                alpha = alpha_new
        # A-posteriori Fehlerabschaetzung
        if alpha is not None and alpha < 1:
            error = alpha * dx / (1 - alpha)
            if error < tol:
                return x_new, i+1
        x = x_new
        dx_old = dx
    raise ValueError("Keine Konvergenz")
```

Newton-Verfahren

```python
def newton(f, df, x0, tol=1e-6, max_iter=100):
    x = x0
    for i in range(max_iter):
        fx = f(x)
        if abs(fx) < tol:
            return x, i+1
        dfx = df(x)
        if dfx == 0:
            raise ValueError("Ableitung Null")
        x = x - fx/dfx
    raise ValueError("Keine Konvergenz")

# Optimierte Version mit Fehlerkontrolle
def newton_safe(f, df, x0, tol=1e-6, max_it=100):
    x = x0
    fx = f(x)
    for i in range(max_it):
        dfx = df(x)
        if dfx == 0:
            raise ValueError("Ableitung Null")
        dx = fx/dfx
        x_new = x - dx
        fx_new = f(x_new)
        # Verschiedene Konvergenzkriterien
        if abs(fx_new) < tol:  # Funktionswert
            return x_new, i+1
        if abs(dx) < tol * (1 + abs(x)):  # Relative
            Aenderung
            return x_new, i+1
        if abs(fx_new) >= abs(fx):  # Divergenzcheck
            raise ValueError("Divergenz detektiert")
        x, fx = x_new, fx_new
    raise ValueError("Keine Konvergenz")
```

Sekantenverfahren

```python
# Einfache Version
def secant(f, x0, x1, tol=1e-6, max_iter=100):
    fx0 = f(x0)
    fx1 = f(x1)

    for i in range(max_iter):
        if abs(fx1) < tol:
            return x1, i+1

        if fx1 == fx0:
            raise ValueError("Division durch Null")

        x2 = x1 - fx1 * (x1 - x0)/(fx1 - fx0)
        x0, x1 = x1, x2
        fx0, fx1 = fx1, f(x2)

    raise ValueError("Keine Konvergenz")

# Optimierte Version mit Fehlerkontrolle
def secant_safe(f, x0, x1, tol=1e-6, max_iter=100):
    fx0 = f(x0)
    fx1 = f(x1)

    if abs(fx0) < abs(fx1):  # Stelle mit kleinerem
        f-Wert als x1
        x0, x1 = x1, x0
        fx0, fx1 = fx1, fx0

    for i in range(max_iter):
        if abs(fx1) < tol:
            return x1, i+1

        if fx1 == fx0:
            raise ValueError("Division durch Null")

        # Sekanten-Schritt
        d = fx1 * (x1 - x0)/(fx1 - fx0)
        x2 = x1 - d

        # Konvergenzpruefungen
        if abs(d) < tol * (1 + abs(x1)):  # Relative
            Aenderung
            return x2, i+1

        fx2 = f(x2)
        if abs(fx2) >= abs(fx1):  # Divergenzcheck
            if i == 0:
                raise ValueError("Schlechte
                    Startwerte")
            return x1, i+1

        x0, x1 = x1, x2
        fx0, fx1 = fx1, fx2

    raise ValueError("Keine Konvergenz")
```

Nullstellensuche mit Fehlerabschätzung

Praktische Implementierung

```python
def root_finder_with_error(f, x0, tol=1e-6,
    max_iter=100):
    x_old = x0
    f_old = f(x_old)

    for i in range(max_iter):
        # Iterationsschritt (hier Newton als Beispiel)
        x_new = x_old - f_old/derivative(f, x_old)
        f_new = f(x_new)

        # Pruefe Konvergenzkriterien
        converged, reason = convergence_criteria(
            x_new, x_old, f_new, f_old, tol)

        if converged:
            # Schaetze finalen Fehler
            error = error_estimate(f, x_new, tol)
            return {
                'root': x_new,
                'iterations': i+1,
                'error_bound': error,
                'convergence_reason': reason
            }

        x_old, f_old = x_new, f_new

    raise ValueError(f"Keine Konvergenz nach
        {max_iter} Iterationen")

    # Returns: Dictionary mit Ergebnissen

# Beispielnutzung
def example_function(x):
    return x**2 - 2

result = root_finder_with_error(example_function, 1.0)
print(f"Nullstelle: {result['root']:.10f}")
print(f"Iterationen: {result['iterations']}")
print(f"Fehlerschranke: {result['error_bound']:.10f}")
print(f"Konvergenzgrund:
    {result['convergence_reason']}")

# Ausgabe etwa:
# Nullstelle: 1.4142135624
# Iterationen: 5
# Fehlerschranke: 1e-06
# Konvergenzgrund: Funktionswert < tol
```

LGS und Matrizen

Matrizen

Matrix, Element, Zeilen, Spalten und Typ

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

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

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

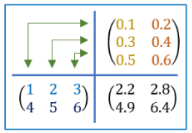
Spaltenmatrix Besteht eine Matrix nur aus einer Spalte, so heisst diese *Spaltenmatrix*. Können als Vektoren aufgefasst werden und können mit einem kleinen Buchstaben sowie einem Pfeil darüber notiert werden ($\vec{a}$).

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

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

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

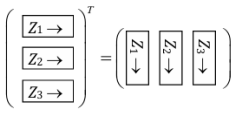
Matrixmultiplikation $A^{m \times n}$, $B^{n \times k}$
Bedingung: A n Spalten, B n Zeilen.
Resultat: C hat m Zeilen und k Spalten.
- $A \cdot B = C$
- $c_{ij} = a_{i1} \cdot b_{1j} + a_{i2} \cdot b_{2j} + \ldots + a_{in} \cdot b_{nj}$
- $A \cdot B \neq B \cdot A$



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

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

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



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

Lineare Gleichungssysteme (LGS)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Quadratische Matrizen

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

Inverse

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

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

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

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

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

NUR Invertierbar falls $ad - bc \neq 0$

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

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

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

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

Zeilenstufenform (linke Seite)

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

Reduzierte Zeilenstufenform (linke Seite)

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

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

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

Beispiel:

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

Numerische Lösung linearer Gleichungssysteme

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

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

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

Zeilenvertauschung für Matrix A mit Permutationsmatrix P_1:

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

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

Pivotisierung

Spaltenpivotisierung

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

Gauss-Algorithmus mit Pivotisierung

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

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

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

Eliminationsschritte:

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

Rückwärtseinsetzen:

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

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

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

Gauss-Algorithmus mit Pivotisierung

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

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

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

    return x
```

LR-Zerlegung Implementation

```python
# Einfache Version ohne externe Bibliotheken
def lr_decomposition(A):
    n = len(A)
    # Kopiere A um Original nicht zu veraendern
    R = [[A[i][j] for j in range(n)] for i in range(n)]
    L = [[1.0 if i == j else 0.0 for j in range(n)]
         for i in range(n)]
    P = [[1.0 if i == j else 0.0 for j in range(n)]
         for i in range(n)]

    for k in range(n-1):
        # Pivotisierung
        pivot = k
        for i in range(k+1, n):
            if abs(R[i][k]) > abs(R[pivot][k]):
                pivot = i

        if abs(R[pivot][k]) < 1e-10:  # Numerische Null
            raise ValueError("Matrix ist (fast)
                 singulaer")

        # Zeilenvertauschung falls noetig
        if pivot != k:
            R[k], R[pivot] = R[pivot], R[k]
            # L und P anpassen fuer Zeilen < k
            for j in range(k):
                L[k][j], L[pivot][j] = L[pivot][j],
                    L[k][j]
            P[k], P[pivot] = P[pivot], P[k]

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

    return P, L, R
```

Matrix-Zerlegungen

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

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

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

LR-Zerlegung

LR-Zerlegung

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

Berechnung der LR-Zerlegung

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

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

Schritt 1: Erste Spalte

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

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

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

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

Schritt 2: Zweite Spalte

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

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

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

Endergebnis

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

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

Lösung des Systems

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

Probe

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

QR-Zerlegung

QR-Zerlegung

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

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

wobei Q orthogonal und R eine obere Dreiecksmatrix ist.

Householder-Transformation

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

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

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

QR-Zerlegung mit Householder

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

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

Schritt 1: Erste Spalte

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

Householder-Vektor für erste Spalte:

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

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

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

Schritt 2: Zweite Spalte

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

Householder-Vektor für zweite Spalte:

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

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

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

Endergebnis

Die QR-Zerlegung $A = QR$ ist:

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

Probe

1. $QR = A$ (bis auf Rundungsfehler)
2. $Q^T Q = QQ^T = I$ (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

QR-Zerlegung Implementation

```python
def qr_decomposition(A):
    m = len(A)
    n = len(A[0])
    # Kopiere A nach R (deep copy)
    R = [[A[i][j] for j in range(n)] for i in range(m)]
    # Initialisiere Q als Einheitsmatrix
    Q = [[1.0 if i == j else 0.0 for j in range(m)]
         for i in range(m)]

    def vector_norm(v): # Norm eines Vektors
        return (sum(x*x for x in v)) ** 0.5

    def matrix_mult(A, B): # Matrixmultiplikation
        m, n = len(A), len(B[0])
        p = len(B)
        C = [[0.0] * n for _ in range(m)]

        for i in range(m):
            for j in range(n):
                C[i][j] = sum(A[i][k] * B[k][j]
                              for k in range(p))

        return C

    def householder_reflection(x):
        n = len(x)
        v = [xi for xi in x]  # Kopiere x

        # Berechne Norm des Teilvektors
        sigma = sum(v[i]*v[i] for i in range(1, n))
        if sigma == 0 and x[0] >= 0:
            beta = 0
        elif sigma == 0 and x[0] < 0:
            beta = -2
        else:
            mu = (x[0]*x[0] + sigma)**0.5
            if x[0] <= 0:
                v[0] = x[0] - mu
            else:
                v[0] = -sigma/(x[0] + mu)

            beta = 2*v[0]*v[0]/(sigma + v[0]*v[0])

            # Normiere v
            temp = v[0]
            for i in range(n):
                v[i] /= temp

        return v, beta
```

QR-Zerlegung Implementation (Fortsetzung)

```python
    # Hauptschleife der QR-Zerlegung
    for k in range(n):
        # Extrahiere k-te Spalte ab k-ter Zeile
        x = [R[i][k] for i in range(k, m)]
        if len(x) > 1:  # wenn noch Untermatrix
             existiert
            # Berechne Householder-Transformation
            v, beta = householder_reflection(x)
            # Wende Householder auf R an
            for j in range(k, n):
                # Berechne w = beta * (v^T * R_j)
                w = beta * sum(v[i-k]*R[i][j]
                               for i in range(k, m))
                # Update R
                for i in range(k, m):
                    R[i][j] -= v[i-k] * w
            # Update Q
            for j in range(m):
                w = beta * sum(v[i-k]*Q[j][i+k]
                               for i in range(len(v)))
                for i in range(len(v)):
                    Q[j][k+i] -= v[i] * w
    # Transponiere Q am Ende
    Q = [[Q[j][i] for j in range(m)] for i in range(m)]

    return Q, R # Q (orthogonal) und R (obere
         Dreiecksmatrix)

# Beispiel fuer Verwendung
def solve_qr(A, b): # Loest Ax = b mittels QR-Zerlegung
    Q, R = qr_decomposition(A)
    # Berechne Q^T * b
    y = [sum(Q[i][j] * b[j]
             for j in range(len(b)))
         for i in range(len(b))]
    # Rueckwaertseinsetzen
    n = len(R)
    x = [0] * n
    for i in range(n-1, -1, -1):
        s = sum(R[i][j] * x[j] for j in range(i+1, n))
        if abs(R[i][i]) < 1e-10:
            raise ValueError("Matrix (fast) singulaer")
        x[i] = (y[i] - s) / R[i][i]
    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,...,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: $Rx = y$

Fehleranalyse

Matrix- und Vektornormen

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

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

Wichtige Normen

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

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

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

Fehlerabschätzung für LGS

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

Absoluter Fehler:

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

Relativer Fehler:

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

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

Konditionierung

Die Konditionszahl beschreibt die numerische Stabilität eines LGS:

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

Konditionierung

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

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

Fehlerabschätzung

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

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

Analyse der Genauigkeit einer Näherungslösung

```python
def error_analysis(A, x, b, x_approx):
    n = len(A)
    # Residuum berechnen
    r = [b[i] - sum(A[i][j] * x_approx[j]
                    for j in range(n))
         for i in range(n)]
    res_norm = max(abs(ri) for ri in r)
    # Relativer Fehler (falls exakte Loesung bekannt)
    if x is not None:
        rel_error = max(abs(x[i] - x_approx[i])
                        for i in range(n)) / \
                    max(abs(x[i]) for i in range(n))
    else:
        rel_error = None
    # Absoluter Fehler (falls exakte Loesung bekannt)
    if x is not None:
        abs_error = max(abs(x[i] - x_approx[i])
                        for i in range(n))
    else:
        abs_error = None

    return {
        'residual_norm': res_norm,
        'relative_error': rel_error,
        'residual': r
    }
```

Iterative Verfahren

Zerlegung der Systemmatrix

A zerlegt in: $A = L + D + R$

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

Jacobi-Verfahren

Gesamtschrittverfahren mit der Iteration:

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

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

Gauss-Seidel-Verfahren

Einzelschrittverfahren mit der Iteration:

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

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:

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

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

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)

Vergleich Lösungsverfahren

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

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
- Iterative Verfahren brauchen mehrere Schritte

Implementation iterativer Verfahren

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

Jacobi-Verfahren Implementation

```python
def jacobi_method(A, b, x0, tol=1e-6, max_iter=100):

    n = len(A)
    x = x0.copy()
    x_new = [0.0] * n

    for iter in range(max_iter):

        # Jacobi-Iteration
        for i in range(n):
            sum1 = sum(A[i][j] * x[j] for j in
                 range(i))
            sum2 = sum(A[i][j] * x[j] for j in
                 range(i+1, n))
            x_new[i] = (b[i] - sum1 - sum2) / A[i][i]

        # Konvergenzpruefung
        diff = max(abs(x_new[i] - x[i]) for i in
             range(n))

        if diff < tol:
            return x_new, iter + 1

        x = x_new.copy()

    raise ValueError(f"Keine Konvergenz nach
         {max_iter} Iterationen")
```

Gauss-Seidel-Verfahren Implementation

```python
def gauss_seidel_method(A, b, x0, tol=1e-6,
     max_iter=100):

    n = len(A)
    x = x0.copy()

    for iter in range(max_iter):
        x_old = x.copy()

        # Gauss-Seidel-Iteration
        for i in range(n):
            sum1 = sum(A[i][j] * x[j] for j in
                 range(i))
            sum2 = sum(A[i][j] * x_old[j] for j in
                 range(i+1, n))
            x[i] = (b[i] - sum1 - sum2) / A[i][i]

        # Konvergenzpruefung
        diff = max(abs(x[i] - x_old[i]) for i in
             range(n))

        if diff < tol:
            return x, iter + 1

    raise ValueError(f"Keine Konvergenz nach
         {max_iter} Iterationen")
```

Analyse von LGS auf numerische Stabilität

```python
def analyze_matrix(A, b):
    n = len(A)
    # 1. Grundlegende Eigenschaften
    diag_dom = is_diagonally_dominant(A)
    scaling = max(abs(A[i][j]) for i in range(n)
                  for j in range(n))
    # 2. Konditionszahl schaetzen
    def matrix_norm_inf(M):
        return max(sum(abs(M[i][j]) for j in
            range(len(M)))
                   for i in range(len(M)))

    def inverse_power_iteration(M, max_iter=100):
        x = [1.0] * n
        for _ in range(max_iter):
            y = solve_triangular(M, x)
            norm = max(abs(yi) for yi in y)
            x = [yi/norm for yi in y]
        return 1.0/norm

    norm_A = matrix_norm_inf(A)
    try:
        norm_Ainv = inverse_power_iteration(A)
        cond = norm_A * norm_Ainv
    except:
        cond = float('inf')
    # 3. Analyse der Diagonalelemente
    min_diag = min(abs(A[i][i]) for i in range(n))
    max_offdiag = max(abs(A[i][j]) for i in range(n)
                      for j in range(n) if i != j)
    # 4. Empfehlungen generieren
    recommendations = []
    if not diag_dom:
        recommendations.append(
            "Matrix nicht diagonaldominant - "
            "Iterative Verfahren koennten divergieren")
    if cond > 1e4:
        recommendations.append(
            f"Hohe Konditionszahl ({cond:.1e}) - "
            "Ergebnisse koennten ungenau sein")
    if min_diag < max_offdiag/100:
        recommendations.append(
            "Kleine Diagonalelemente - "
            "Pivotisierung empfohlen")
    if scaling > 1e8:
        recommendations.append(
            "Grosse Zahlenunterschiede - "
            "Skalierung empfohlen")
    return {
        "recommandations": recommendations,
        "results": cond, diag_dom, scaling, min_diag,
            max_offdiag
    }
```

Hilfsfunktion für Optimiere iterative Verfahren

```python
def is_diagonally_dominant(A): # Matrix
    diagonaldominant?
    n = len(A)
    for i in range(n):
        if abs(A[i][i]) <= sum(abs(A[i][j]) for j in
            range(n) if j != i):
            return False
    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)

```python
def iterative_solver(A, b, method='gauss_seidel',
    tol=1e-6, max_iter=100, omega=1.0):

    n = len(A)
    x = [0.0] * n  # Startvektor
    D = [[A[i][j] if i == j else 0 for j in range(n)]
         for i in range(n)]  # Diagonalmatrix
    L = [[A[i][j] if i > j else 0 for j in range(n)]
         for i in range(n)]  # Untere Dreiecksmatrix
    U = [[A[i][j] if i < j else 0 for j in range(n)]
         for i in range(n)]  # Obere Dreiecksmatrix

    # Konvergenzcheck
    if not is_diagonally_dominant(A):
        print("Warnung: Matrix nicht diagonaldominant")

    iterations = []
    residuals = []

    for iter in range(max_iter):
        x_old = x.copy()
        if method == 'jacobi':
            for i in range(n):
                sum_term = sum(A[i][j] * x_old[j]
                               for j in range(n) if j !=
                                   i)
                x[i] = (1-omega) * x_old[i] + \
                       (omega/A[i][i]) * (b[i] -
                           sum_term)
        else:  # gauss_seidel
            for i in range(n):
                sum1 = sum(A[i][j] * x[j] for j in
                    range(i))
                sum2 = sum(A[i][j] * x_old[j]
                           for j in range(i+1, n))
                x[i] = (1-omega) * x_old[i] + \
                    (omega/A[i][i]) * (b[i] - sum1 -
                    sum2)

        # Berechne Residuum und relative Aenderung
        res = max(abs(sum(A[i][j] * x[j] for j in
            range(n)) - b[i]) for i in range(n))
        diff = max(abs(x[i] - x_old[i]) for i in
            range(n))

        iterations.append(x.copy())
        residuals.append(res)

        if diff < tol:
            return {
                'solution': x,
                'iterations': iterations,
                'residuals': residuals,
                'iteration_count': iter + 1
            }

    raise ValueError(f"Keine Konvergenz nach
        {max_iter} " +
                     f"Iterationen\nLetztes Residuum:
                         {res}")
```

Komplexe Zahlen

Komplexe Zahlen in Python

```python
def complex_operations(z1, z2):
    """Grundlegende Operationen mit komplexen
        Zahlen."""
    # Basisfunktionen
    def to_polar(z):
        r = (z.real**2 + z.imag**2)**0.5
        phi = math.atan2(z.imag, z.real)
        return r, phi

    def from_polar(r, phi):
        return r * (math.cos(phi) + 1j*math.sin(phi))

    try:
        # Addition und Subtraktion
        z_add = z1 + z2
        z_sub = z1 - z2

        # Multiplikation und Division
        z_mul = z1 * z2
        z_div = z1 / z2 if z2 != 0 else None

        # Polarform
        r1, phi1 = to_polar(z1)
        r2, phi2 = to_polar(z2)

        # Exponentialform
        z1_exp = from_polar(r1, phi1)
        z2_exp = from_polar(r2, phi2)

        return {
            'addition': z_add,
            'subtraktion': z_sub,
            'multiplikation': z_mul,
            'division': z_div,
            'polar_z1': (r1, phi1),
            'polar_z2': (r2, phi2)
        }

    except Exception as e:
        print(f"Fehler bei Berechnung: {e}")
        return None

# Beispiel
z1 = 3 - 11j
z2 = 2 + 5j
results = complex_operations(z1, z2)
```

Komplexe Zahlen

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

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

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

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

$$i^2 = -1$$

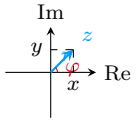
Eine komplexe Zahl z ist ein geordnetes Paar (x, y) mit $x, y \in \mathbb{R}$:

$$z = x + iy$$

Die Menge aller komplexen Zahlen ist definiert als:

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

Bestandteile komplexer Zahlen
Realteil: $\mathrm{Re}(z) = x$
Imaginärteil: $\mathrm{Im}(z) = y$
Betrag: $|z| = \sqrt{x^2 + y^2} = \sqrt{z \cdot z^*}$
Konjugation: $\bar{z} = x - iy$



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

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

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

Umrechnung zwischen Darstellungsformen komplexer Zahlen

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

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

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

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

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

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

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

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

Eigenwerte und Eigenvektoren

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

$$Ax = \lambda x$$

Der Vektor x heißt dann Eigenvektor zum Eigenwert λ.

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

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

Diese Gleichung heißt charakteristische Gleichung. Das zugehörige Polynom

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

ist das charakteristische Polynom von A.

Eigenschaften von Eigenwerten
Für eine Matrix $A \in \mathbb{R}^{n \times n}$ gilt:
- $\det(A) = \prod_{i=1}^{n} \lambda_i$ (Produkt der Eigenwerte)
- $\mathrm{tr}(A) = \sum_{i=1}^{n} \lambda_i$ (Summe der Eigenwerte)
- Bei Dreiecksmatrix sind die Diagonalelemente die Eigenwerte
- Ist λ Eigenwert von A, so ist $\frac{1}{\lambda}$ Eigenwert von A^{-1}

Vielfachheiten
Für einen Eigenwert λ unterscheidet man:
- Algebraische Vielfachheit:
  Vielfachheit als Nullstelle des charakteristischen Polynoms
- Geometrische Vielfachheit:
  Dimension des Eigenraums $= n - \mathrm{rg}(A - \lambda I_n)$

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

Bestimmung von Eigenwerten und Eigenvektoren
1. Charakteristisches Polynom aufstellen:
   - 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 λ_i:
   - System $(A - \lambda_i I_n)x = 0$ aufstellen
   - Gaussverfahren anwenden
   - Lösungsraum = Eigenraum bestimmen
   - Basis des Eigenraums = linear unabhängige Eigenvektoren
   - Lösung parametrisieren
   - Parameter != 0 waehlen für Eigenvektor
4. Überprüfung:
   - $Ax = \lambda x$ für jeden Eigenvektor
   - Anzahl linear unabhaengiger Eigenvektoren kleiner als algebraische Vielfachheit

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

1. Charakteristisches Polynom aufstellen:
$\det(A - \lambda I) = \begin{vmatrix} 2-\lambda & 1 & 0 \\ 1 & 2-\lambda & 1 \\ 0 & 1 & 2-\lambda \end{vmatrix}$

2. Entwicklung nach 1. Zeile: $p(\lambda) = (2-\lambda)\begin{vmatrix} 2-\lambda & 1 \\ 1 & 2-\lambda \end{vmatrix} - 1\begin{vmatrix} 1 & 1 \\ 1 & 2-\lambda \end{vmatrix}$

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

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

5. Eigenvektoren bestimmen für $\lambda_1 = 1$:
$(A - I)x = 0$ führt zu $x_1 = \begin{pmatrix} 1 \\ -2 \\ 1 \end{pmatrix}$

Charakteristisches Polynom und Eigenwerte

```python
def det_2x2(matrix):
    return matrix[0][0]*matrix[1][1] -
        matrix[0][1]*matrix[1][0]

def det_3x3(matrix):
    det = 0
    # Entwicklung nach erster Zeile
    for i in range(3):
        minor = []
        for j in range(1,3):
            row = []
            for k in range(3):
                if k != i:
                    row.append(matrix[j][k])
            minor.append(row)
        det += ((-1)**i) * matrix[0][i] *
            det_2x2(minor)
    return det

def char_poly_coeff(matrix):
    """Berechnet Koeffizienten des charakteristischen
        Polynoms"""
    n = len(matrix)
    coeffs = [0] * (n+1)

    # Lambda^n Koeffizient
    coeffs[n] = (-1)**n

    # Spur (Lambda^(n-1) Koeffizient)
    trace = sum(matrix[i][i] for i in range(n))
    coeffs[n-1] = (-1)**(n-1) * trace

    # Determinante (konstanter Term)
    coeffs[0] = det_3x3(matrix)

    return coeffs

# Beispielmatrix
A = [[2, 1, 0],
     [1, 2, 1],
     [0, 1, 2]]

# Charakteristisches Polynom berechnen
poly = char_poly_coeff(A)
print("Charakteristisches Polynom:")
print(f"p(lambda) = lambda^3 {poly[2]:+}lambda^2
    {poly[1]:+}lambda {poly[0]:+}")
```

Eigenvektoren berechnen

```python
def gauss_elimination(A, b):
    """Loest Ax=b mit Gauss-Elimination"""
    n = len(A)
    # Erweiterte Matrix erstellen
    M = [[A[i][j] for j in range(n)] + [b[i]] for i in
        range(n)]

    # Vorwaertselimination
    for i in range(n):
        pivot = M[i][i]
        if abs(pivot) < 1e-10:
            continue
        for j in range(i+1, n):
            factor = M[j][i] / pivot
            for k in range(i, n+1):
                M[j][k] -= factor * M[i][k]

    # Rueckwaertssubstitution
    x = [0] * n
    for i in range(n-1, -1, -1):
        if abs(M[i][i]) < 1e-10:
            x[i] = 1  # Freie Variable
            continue
        sum_val = sum(M[i][j] * x[j] for j in
            range(i+1, n))
        x[i] = (M[i][n] - sum_val) / M[i][i]

    return x

def find_eigenvector(A, eigenvalue):
    """Findet Eigenvektor zum Eigenwert"""
    n = len(A)
    # A - lambda*I
    A_lambda = [[A[i][j] - (eigenvalue if i==j else 0)
                 for j in range(n)] for i in range(n)]

    # Loese (A - lambda*I)x = 0
    b = [0] * n
    return gauss_elimination(A_lambda, b)

# Eigenvektoren fuer lambda = 1,2,3 berechnen
eigenvalues = [1, 2, 3]
for ev in eigenvalues:
    vec = find_eigenvector(A, ev)
    print(f"Eigenvektor fuer lambda={ev}:", vec)
```

Numerische Berechnung von Eigenwerten

Ähnliche Matrizen

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

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

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

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

Eigenschaften ähnlicher Matrizen

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

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

Spektralradius Der Spektralradius einer Matrix A ist definiert als:

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

Er gibt den Betrag des betragsmäßig größten Eigenwerts an.

Von-Mises-Iteration

Von-Mises-Iteration (Vektoriteration)

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

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

gegen einen Eigenvektor v zum betragsmäßig größten Eigenwert λ_1.

Von-Mises-Iteration durchführen

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

Von-Mises-Iteration 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

```python
def power_iteration(A, tol=1e-10, max_iter=100):
    n = len(A)
    v = [random.random() for _ in range(n)]
    v = [x / np.linalg.norm(v) for x in v]
    for i in range(max_iter):
        w = [sum(A[i][j] * v[j] for j in range(n)) for
             i in range(n)]
        v_new = [x / np.linalg.norm(w) for x in w]
        # Rayleigh-Quotient
        lambda_k = sum(v_new[i] * A[i][j] * v_new[j]
             for i in range(n) for j in range(n))
        if np.linalg.norm([v_new[i] - v[i] for i in
             range(n)]) < tol:
            return lambda_k, v_new
        v = v_new
    return lambda_k, v_new
```

Von-Mises-Iteration

```python
def matrix_vector_mult(A, v):
    n = len(v)
    return [sum(A[i][j] * v[j] for j in range(n))
            for i in range(n)]

def vector_norm(v):
    return (sum(x*x for x in v)) ** 0.5

def normalize_vector(v):
    norm = vector_norm(v)
    return [x/norm for x in v]

def rayleigh_quotient(A, v):
    numerator = sum(v[i] * sum(A[i][j] * v[j]
                    for j in range(len(v)))
                    for i in range(len(v)))
    denominator = sum(x*x for x in v)
    return numerator / denominator

def power_method(A, tol=1e-10, max_iter=100):
    n = len(A)
    # Startvektor [1,0,0,...,0]
    v = [1.0] + [0.0] * (n-1)
    v = normalize_vector(v)

    for k in range(max_iter):
        # Matrixmultiplikation
        w = matrix_vector_mult(A, v)
        # Normierung
        v_new = normalize_vector(w)
        # Eigenwertapproximation
        lambda_k = rayleigh_quotient(A, v_new)
        # Konvergenztest
        if vector_norm([v_new[i]-v[i]
                        for i in range(n)]) < tol:
            return lambda_k, v_new
        v = v_new

    return lambda_k, v
```

Von-Mises-Iteration ohne spezielle Bibliotheken

```python
def matrix_vector_mult(A, v):
    n = len(A)
    result = [0] * n
    for i in range(n):
        for j in range(n):
            result[i] += A[i][j] * v[j]
    return result

def vector_norm(v):
    return sum(x*x for x in v) ** 0.5

def normalize_vector(v):
    norm = vector_norm(v)
    return [x/norm for x in v]

def power_iteration(A, tol=1e-10, max_iter=100):
    n = len(A)
    # Startvektor
    v = normalize_vector([1] + [0]*(n-1))

    for _ in range(max_iter):
        # Matrix-Vektor-Multiplikation
        w = matrix_vector_mult(A, v)
        # Normierung
        v_new = normalize_vector(w)

        # Rayleigh-Quotient
        lambda_k = sum(v_new[i] * A[i][j] * v_new[j]
                       for i in range(n)
                       for j in range(n))

        # Konvergenzpruefung
        if vector_norm([v_new[i]-v[i] for i in
            range(n)]) < tol:
            return lambda_k, v_new

        v = v_new
    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 $i = 0, 1, 2, \ldots$:
   - QR-Zerlegung: $A_i = Q_i R_i$
   - Neue Matrix: $A_{i+1} = R_i Q_i$
   - Update: $P_{i+1} = P_i Q_i$

QR-Verfahren anwenden

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

QR-Verfahren

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

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

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

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

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

```python
def matmul(A, B):
    n = len(A)
    C = [[0.0] * n for _ in range(n)]
    for i in range(n):
        for j in range(n):
            C[i][j] = sum(A[i][k] * B[k][j]
                          for k in range(n))
    return C

def transpose(A):
    n = len(A)
    return [[A[j][i] for j in range(n)]
            for i in range(n)]

def householder(x):
    n = len(x)
    # Norm berechnen
    s = sum(x[i]**2 for i in range(1, n))
    v = [0.0] * n
    if s == 0:
        return v

    v[0] = x[0]
    norm_x = (x[0]**2 + s)**0.5
    if x[0] <= 0:
        v[0] = x[0] - norm_x
    else:
        v[0] = -s/(x[0] + norm_x)

    for i in range(1, n):
        v[i] = x[i]

    return normalize_vector(v)

def qr_algorithm(A, tol=1e-10, max_iter=100):
    n = len(A)
    A_k = [row[:] for row in A]  # Kopiere A

    for k in range(max_iter):
        # QR-Zerlegung mit Householder
        Q = [[1.0 if i==j else 0.0
              for j in range(n)]
              for i in range(n)]
        R = [row[:] for row in A_k]

        for j in range(n-1):
            v = householder([R[i][j]
                 for i in range(j, n)])
            # Householder-Transformation anwenden

        # Neue Iteration A_(k+1) = RQ
        A_k = matmul(R, Q)

        # Konvergenztest
        if max(abs(A_k[i][j])
               for i in range(1, n)
               for j in range(i)) < tol:
            break

    return [A_k[i][i] for i in range(n)]
```

QR-Verfahren ohne spezielle Bibliotheken

```python
def matrix_mult(A, B):
    n = len(A)
    C = [[0]*n for _ in range(n)]
    for i in range(n):
        for j in range(n):
            for k in range(n):
                C[i][j] += A[i][k] * B[k][j]
    return C

def gram_schmidt(A):
    n = len(A)
    Q = [[0]*n for _ in range(n)]
    R = [[0]*n for _ in range(n)]

    # Extrahiere Spalten von A
    V = [[A[i][j] for i in range(n)] for j in range(n)]

    for i in range(n):
        # Berechne Q[:, i]
        Q_col = V[i]
        for j in range(i):
            # Berechne Projektion und subtrahiere
            R[j][i] = sum(Q[k][j] * V[i][k] for k in
                 range(n))
            Q_col = [Q_col[k] - R[j][i] * Q[k][j]
                     for k in range(n)]

        # Normiere
        R[i][i] = vector_norm(Q_col)
        Q_col = [x/R[i][i] for x in Q_col]

        # Speichere normierte Spalte
        for j in range(n):
            Q[j][i] = Q_col[j]

    return Q, R

def qr_algorithm(A, tol=1e-10, max_iter=100):
    n = len(A)
    A_k = [row[:] for row in A]  # Kopiere A

    for _ in range(max_iter):
        Q, R = gram_schmidt(A_k)
        A_k = matrix_mult(R, Q)

        # Pruefe Konvergenz (Subdiagonalelemente)
        converged = True
        for i in range(1, n):
            if abs(A_k[i][i-1]) > tol:
                converged = False
                break
        if converged:
            break

    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 → Von-Mises
   - Alle EW → QR-Verfahren
   - Symmetrisch → Symmetrisches QR
   - Einzelner EW → 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

```python
def inverse_iteration(A, mu, tol=1e-10, max_iter=100):
    n = len(A)
    # Startvektor
    v = [1.0] + [0.0] * (n-1)
    v = normalize_vector(v)

    # Matrix (A - mu*I) erstellen
    A_shift = [[A[i][j] if i != j
                else A[i][j] - mu
                for j in range(n)]
               for i in range(n)]

    for k in range(max_iter):
        # Loese (A - mu*I)w = v
        w = solve_linear_system(A_shift, v)
        # Normiere Ergebnisvektor
        v_new = normalize_vector(w)

        # Konvergenztest
        if vector_norm([v_new[i]-v[i]
                        for i in range(n)]) < tol:
            # Berechne Rayleigh-Quotienten
            lambda_k = rayleigh_quotient(A, v_new)
            return lambda_k, v_new

        v = v_new

    raise ValueError("Keine Konvergenz")

def solve_linear_system(A, b):
    # LR-Zerlegung und Rueckwaertseinsetzen
    n = len(A)
    x = gauss_solve(A, b)  # aus LGS Kapitel
    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 μ 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

```python
from decimal import Decimal, getcontext

# Set precision
getcontext().prec = 40

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

NumPy

NumPy NumPy: Numerische Python-Bibliothek
- Effiziente Implementierung von Arrays
- Vektorisierte Operationen
- Lineare Algebra, Fourier-Transformation, Zufallszahlen

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

LR-Zerlegung Implementation

```python
# Optimierte Version mit NumPy
def lr_decomposition_numpy(A):
    n = len(A)
    R = np.array(A, dtype=float)
    L = np.eye(n)
    P = np.eye(n)

    for k in range(n-1):
        # Pivotisierung
        pivot = np.argmax(abs(R[k:,k])) + k

        if abs(R[pivot,k]) < 1e-10:
            raise ValueError("Matrix ist (fast)
                singulaer")

        if pivot != k:
            # Zeilenvertauschung
            R[[k,pivot]] = R[[pivot,k]]
            L[[k,pivot], :k] = L[[pivot,k], :k]
            P[[k,pivot]] = P[[pivot,k]]

        # Elimination
        L[k+1:,k] = R[k+1:,k] / R[k,k]
        R[k+1:] -= np.outer(L[k+1:,k], R[k])

    return P, L, R
```

Eigenwerte und Eigenvektoren

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

Von-Mises-Iteration

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

Fehlerabschätzung

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

Jacobi- und Gauss-Seidel-Verfahren

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

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

Komplexe Zahlen in Python

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

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

EW und EV über Charakteristisches Polynom

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

Examples

Rechnerarithmetik

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

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

1. Normalisierung prüfen:
- $m_1 = 1 \neq 0 \rightarrow$ normalisiert

2. Exponent berechnen:

$$\begin{aligned}\hat{e} &= 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \\ &= 4 + 0 + 1 = 5\end{aligned}$$

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:

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

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 $x_0 = 1$

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 $\sqrt[3]{2} \approx 1.259921$

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:

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

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
1. $Pb = \begin{pmatrix} 3 \\ 5 \\ 2 \end{pmatrix}$
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
$v_1 = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}$, $\|v_1\| = \sqrt{2}$
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)
- $QR = A$ (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

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)

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