

Lineare Gleichungssysteme

Die Gleichung, die in der Finite-Elemente-Methode (FEM) gelöst wird, lässt sich in Matrix- und Vektorform wie folgt darstellen:

$$K \cdot u = F$$

wobei:

- K die **Steifigkeitsmatrix** ist (in der Regel eine symmetrische, dünnbesetzte Matrix),
- u der **Verschiebungsvektor** (Unbekannte) ist, und
- F der **Kraftvektor** (Bekannte) ist.

Für ein einfaches Beispiel einer 6x6-Matrix, wie sie in der FEM vorkommen könnte, nehmen wir eine symmetrische Matrix, die eine Struktur repräsentiert, bei der Knotenpunkte durch Steifigkeitsbeziehungen miteinander verbunden sind. Die Matrix könnte wie folgt aussehen:

$$K = \begin{pmatrix} k_{11} & k_{12} & 0 & 0 & 0 & 0 \\ k_{12} & k_{22} & k_{23} & 0 & 0 & 0 \\ 0 & k_{23} & k_{33} & k_{34} & 0 & 0 \\ 0 & 0 & k_{34} & k_{44} & k_{45} & 0 \\ 0 & 0 & 0 & k_{45} & k_{55} & k_{56} \\ 0 & 0 & 0 & 0 & k_{56} & k_{66} \end{pmatrix}$$

Hierbei sind:

- Die Diagonalelemente k_{ii} die Steifigkeiten des jeweiligen Knotens.
- Die Nicht-Null-Nebendiagonalelemente k_{ij} die Kopplungen zwischen benachbarten Knoten.

Lineare Gleichungssysteme

Beispielwerte für K

Um ein numerisches Beispiel zu erstellen, setzen wir exemplarische Werte ein:

$$K = \begin{pmatrix} 10 & -2 & 0 & 0 & 0 & 0 \\ -2 & 12 & -3 & 0 & 0 & 0 \\ 0 & -3 & 15 & -4 & 0 & 0 \\ 0 & 0 & -4 & 18 & -5 & 0 \\ 0 & 0 & 0 & -5 & 20 & -6 \\ 0 & 0 & 0 & 0 & -6 & 22 \end{pmatrix}$$

Vektorform

Gegeben diese Matrix K , könnten wir die Vektoren u (Unbekannte Verschiebungen) und F (bekannte Kräfte) wie folgt definieren:

$$u = \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \end{pmatrix}, \quad F = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ f_6 \end{pmatrix}$$

Das vollständige Gleichungssystem

Damit sieht das Gleichungssystem in ausgearbeiteter Form wie folgt aus:

$$\begin{pmatrix} 10 & -2 & 0 & 0 & 0 & 0 \\ -2 & 12 & -3 & 0 & 0 & 0 \\ 0 & -3 & 15 & -4 & 0 & 0 \\ 0 & 0 & -4 & 18 & -5 & 0 \\ 0 & 0 & 0 & -5 & 20 & -6 \\ 0 & 0 & 0 & 0 & -6 & 22 \end{pmatrix} \cdot \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ f_6 \end{pmatrix}$$

Dies stellt ein typisches Gleichungssystem dar, das in der FEM gelöst werden muss, um die Verschiebungen u bei gegebenen Kräften F zu bestimmen.

In der Finite-Elemente-Methode (FEM) treten häufig **dünn besetzte Matrizen** (Sparse Matrices) in Form von **Bandmatrizen** auf, im Gegensatz zu voll besetzten Matrizen (Dense Matrices). Diese beiden Matrizenarten haben unterschiedliche Eigenschaften und Anforderungen an die Speicherverwaltung und Rechenverfahren. Hier einige wesentliche Unterschiede und Eigenschaften:

1. Struktur und Speicherbedarf

- **Dünn besetzte Matrix (Bandmatrix):**
 - In einer dünn besetzten Matrix sind die meisten Elemente Null, außer in bestimmten Bereichen, die das **Band** bilden. Dies ist typisch in FEM, da nur direkt benachbarte Knoten in der Steifigkeitsmatrix gekoppelt sind.
 - Nur die nicht-Null-Elemente müssen gespeichert werden, was den Speicherbedarf erheblich reduziert.
 - Beispielsweise hat eine dünn besetzte Matrix für ein großes FEM-Modell vielleicht weniger als 1 % nicht-Null-Elemente.
- **Voll besetzte Matrix:**
 - Hier haben die meisten Elemente einen Wert (selten Null). Das bedeutet, dass alle Elemente, unabhängig davon, ob sie Null sind oder nicht, gespeichert werden müssen.
 - Der Speicherbedarf skaliert schnell mit der Matrixgröße, sodass voll besetzte Matrizen bei großen Modellen in FEM ineffizient und speicherintensiv werden.

2. Effizienz bei Rechenoperationen

- **Dünn besetzte Matrix:**
 - Die geringe Anzahl von nicht-Null-Elementen ermöglicht spezielle Algorithmen und Speichertechniken, die nur die tatsächlich besetzten Elemente verarbeiten.
 - Viele iterative Löser, wie das Conjugate Gradient (CG) oder GMRES-Verfahren, sind für dünn besetzte Matrizen optimiert, da sie den Zugriff auf nur die relevanten Elemente benötigen.
 - Diese Effizienz macht es möglich, sehr große Systeme, wie sie in der FEM auftreten, zu lösen, ohne die gesamte Matrix zu verarbeiten.
- **Voll besetzte Matrix:**
 - Direkte Löser (z. B. LU-Zerlegung) arbeiten effizienter auf voll besetzten Matrizen, da sie für alle Matrixelemente Zugriffe vornehmen müssen.
 - Voll besetzte Matrizen profitieren weniger von Iterationsmethoden, weil die hohe Anzahl an Nicht-Null-Elementen das Verfahren ineffizient macht.

3. Rechenzeit und Algorithmenkomplexität

- **Dünn besetzte Matrix:**
 - Die Rechenzeit für dünn besetzte Matrizen hängt stark von der Bandbreite ab. Verfahren wie bandstrukturierte Cholesky- oder Sparse-LU-Zerlegung sind speziell darauf optimiert, nur das Band zu verarbeiten, was die Rechenzeit reduziert.
 - Iterative Verfahren können bei dünn besetzten Matrizen besonders effizient sein, da die Konvergenz bei kleineren Anzahl an Operationen erreicht wird.
- **Voll besetzte Matrix:**
 - Die Rechenzeit ist hier proportional zur Anzahl der Elemente ($O(n^3)$ für Direkte Methoden wie LU-Zerlegung). Daher sind große voll besetzte Matrizen für FEM ungeeignet, besonders für iterative Methoden, da die Konvergenz langsamer erreicht wird.

4. Numerische Stabilität

- **Dünn besetzte Matrix:**
 - Bei dünn besetzten Matrizen kann die numerische Stabilität durch große Bandlücken beeinflusst werden. Die Wahl geeigneter Preconditioner kann hier hilfreich sein, um die Konvergenz zu beschleunigen und Stabilität zu gewährleisten.
 - Eine typische dünn besetzte Steifigkeitsmatrix in FEM ist oft symmetrisch und positiv definit, was die Anwendung von stabilen Lösungsverfahren wie dem Conjugate Gradient ermöglicht.
- **Voll besetzte Matrix:**
 - Voll besetzte Matrizen bieten häufig eine höhere numerische Stabilität bei Direkten Methoden, da sie weniger anfällig für schlecht konditionierte Systeme sind.
 - Aufgrund der höheren Stabilität sind sie oft für kleinere Systeme bevorzugt, für die eine exakte Lösung in einer Iteration gewünscht ist.

Direkte Methoden

Direkte Methoden lösen das Gleichungssystem durch eine **exakte Berechnung**, was besonders nützlich für kleinere Systeme und dichte Matrizen ist. Zu den Hauptmethoden zählen:

1. LU-Zerlegung:

- Die Steifigkeitsmatrix K wird in eine untere (L) und eine obere (U) Dreiecksmatrix zerlegt, sodass $K = L \cdot U$.
- Vorteilhaft für kleinere, dichte Systeme oder Matrizen mit breiten Bändern.

2. Cholesky-Zerlegung (für symmetrisch positive Matrizen):

- Spezialfall der LU-Zerlegung, der bei symmetrischen und positiv definiten Matrizen eine effizientere Zerlegung ermöglicht.

3. Gaußsche Eliminationsmethode:

- Standardverfahren, das alle Elemente der Matrix verarbeitet, um eine exakte Lösung zu finden.
- **Einschränkungen für Bandmatrizen:**
 - Da die Gaußsche Methode alle Elemente der Matrix verwendet, ist sie ineffizient bei dünnbesetzten oder Bandmatrizen, da unnötige Speicher- und Rechenressourcen verbraucht werden.
 - Die Eliminationsschritte führen bei Bandmatrizen oft zur **Füll-in-Problematik** (es entstehen zusätzliche Nicht-Null-Elemente außerhalb des Bandes), was den Speicherbedarf erhöht und den Lösungsprozess verlangsamt.

Lineare Gleichungssysteme

Iterative Methode

Iterative Methoden

Iterative Methoden nähern sich der Lösung schrittweise an und eignen sich besonders gut für große, dünnbesetzte Matrizen. Die bekanntesten Methoden sind:

1. Gauß-Seidel-Verfahren:

- Einfache iterative Methode, bei der jede Variable schrittweise aktualisiert wird, indem die bisher berechneten Werte in der nächsten Berechnung verwendet werden.
- Der Wert jeder Variablen wird so angepasst, dass er sofort in die folgenden Berechnungen einfließt.
- Vorteile:
 - Effizient für diagonaldominante Matrizen und Systeme, bei denen eine moderate Genauigkeit genügt.
 - Wenig Speicherbedarf, da keine zusätzlichen Matrizen gespeichert werden müssen.
- Nachteile:
 - Konvergiert langsam und kann bei schlecht konditionierten oder nicht diagonaldominanten Matrizen divergieren.
 - Bei sehr großen und komplexen Systemen oft ineffektiv, da viele Iterationen notwendig sind, um zur Lösung zu gelangen.

2. Conjugate Gradient (CG) Methode:

- Iterative Methode speziell für symmetrische, positiv definite Matrizen.
- Die CG-Methode minimiert den Fehlervektor entlang von Konjugationsrichtungen, wodurch eine schnellere Konvergenz erreicht wird.
- Vorteile:
 - Schnelle Konvergenz für dünnbesetzte FEM-Matrizen, insbesondere bei großer Systemgröße.
 - Benötigt wenig Speicher und nutzt nur die nicht-Null-Elemente der Matrix.
- Nachteile:
 - Funktioniert nur für symmetrische, positiv definite Matrizen, daher ist es nicht universell anwendbar.
 - Kann bei schlecht konditionierten Matrizen langsam konvergieren oder sogar divergieren; ein Preconditioner ist oft erforderlich, um die Konvergenz zu beschleunigen.

Lineare Gleichungssysteme

Methode	Vorteile	Nachteile
Direkte Methoden		
LU-Zerlegung	<ul style="list-style-type: none"> - Exakte Lösung für kleine bis mittelgroße Systeme - Hohe numerische Stabilität und Genauigkeit 	<ul style="list-style-type: none"> - Hoher Speicherbedarf, ineffizient bei großen, dünnbesetzten Matrizen - Füll-in-Problematik erhöht Speicherbedarf und Rechenzeit
Cholesky-Zerlegung	<ul style="list-style-type: none"> - Effizient für symmetrisch positive Matrizen - Spart Speicher im Vergleich zur vollständigen LU-Zerlegung 	<ul style="list-style-type: none"> - Eingeschränkt auf symmetrisch positive Matrizen
Gaußsche Eliminationsmethode	<ul style="list-style-type: none"> - Einfach und stabil bei gut konditionierten Matrizen - Liefert immer exakte Lösung 	<ul style="list-style-type: none"> - Ineffizient bei Band- und dünnbesetzten Matrizen wegen Füll-in-Problematik - Hohe Speicher- und Rechenanforderungen
Iterative Methoden		
Gauß-Seidel-Verfahren	<ul style="list-style-type: none"> - Wenig Speicherbedarf - Effizient bei diagonaldominanten Matrizen - Einfach zu implementieren 	<ul style="list-style-type: none"> - Langsame Konvergenz, besonders bei schlecht konditionierten Matrizen - Kann divergieren bei nicht diagonaldominanten Matrizen
Conjugate Gradient (CG)	<ul style="list-style-type: none"> - Schnelle Konvergenz bei großen, dünnbesetzten und symmetrisch positiven Matrizen - Weniger Speicherbedarf - Nutzt nur nicht-Null-Elemente 	<ul style="list-style-type: none"> - Funktioniert nur für symmetrisch positive Matrizen - Kann bei schlecht konditionierten Matrizen langsamer konvergieren oder divergieren, oft Preconditioner erforderlich

Lineare Gleichungssysteme

Direkte Methode

LU-Zerlegung

Die **LU-Zerlegung** ist ein Verfahren zur Lösung eines linearen Gleichungssystems $K \cdot u = F$, indem die Matrix K in das Produkt einer unteren Dreiecksmatrix L und einer oberen Dreiecksmatrix U zerlegt wird:

$$K = L \cdot U$$

Damit kann das Gleichungssystem in zwei einfachere Gleichungssysteme umgeschrieben werden:

1. **Schritt 1:** Berechne L und U aus der Matrix K .
2. **Schritt 2:** Löse das Gleichungssystem in zwei Schritten:
 - Vorwärtseinsetzen für $L \cdot y = F$
 - Rückwärtseinsetzen für $U \cdot u = y$

Beispiel einer LU-Zerlegung für eine 6x6-Matrix

Nehmen wir eine 6x6-Matrix K als Beispiel:

$$K = \begin{pmatrix} 10 & -2 & 0 & 0 & 0 & 0 \\ -2 & 12 & -3 & 0 & 0 & 0 \\ 0 & -3 & 15 & -4 & 0 & 0 \\ 0 & 0 & -4 & 18 & -5 & 0 \\ 0 & 0 & 0 & -5 & 20 & -6 \\ 0 & 0 & 0 & 0 & -6 & 22 \end{pmatrix}$$

Lineare Gleichungssysteme

Direkte Methode

LU-Zerlegung

Beispiel einer LU-Zerlegung für eine 6x6-Matrix

Nehmen wir eine 6x6-Matrix K als Beispiel:

$$K = \begin{pmatrix} 10 & -2 & 0 & 0 & 0 & 0 \\ -2 & 12 & -3 & 0 & 0 & 0 \\ 0 & -3 & 15 & -4 & 0 & 0 \\ 0 & 0 & -4 & 18 & -5 & 0 \\ 0 & 0 & 0 & -5 & 20 & -6 \\ 0 & 0 & 0 & 0 & -6 & 22 \end{pmatrix}$$

Schritt 1: Zerlegung von K in L und U

Die Matrix L ist eine untere Dreiecksmatrix mit Einsen auf der Diagonale, und U ist eine obere Dreiecksmatrix. Wir möchten K so zerlegen, dass:

$$K = \begin{pmatrix} 10 & -2 & 0 & 0 & 0 & 0 \\ -2 & 12 & -3 & 0 & 0 & 0 \\ 0 & -3 & 15 & -4 & 0 & 0 \\ 0 & 0 & -4 & 18 & -5 & 0 \\ 0 & 0 & 0 & -5 & 20 & -6 \\ 0 & 0 & 0 & 0 & -6 & 22 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ l_{21} & 1 & 0 & 0 & 0 & 0 \\ 0 & l_{32} & 1 & 0 & 0 & 0 \\ 0 & 0 & l_{43} & 1 & 0 & 0 \\ 0 & 0 & 0 & l_{54} & 1 & 0 \\ 0 & 0 & 0 & 0 & l_{65} & 1 \end{pmatrix} \cdot \begin{pmatrix} u_{11} & u_{12} & 0 & 0 & 0 & 0 \\ 0 & u_{22} & u_{23} & 0 & 0 & 0 \\ 0 & 0 & u_{33} & u_{34} & 0 & 0 \\ 0 & 0 & 0 & u_{44} & u_{45} & 0 \\ 0 & 0 & 0 & 0 & u_{55} & u_{56} \\ 0 & 0 & 0 & 0 & 0 & u_{66} \end{pmatrix}$$

Dabei werden die Werte von L und U berechnet, indem für jedes Element der Matrix das Gleichgewicht mit K hergestellt wird:

- Diagonalelemente von U :

$$u_{ii} = k_{ii} - \sum_{j=1}^{i-1} l_{ij} \cdot u_{ji}$$

- Nicht-Diagonalelemente von U (für $i < j$):

$$u_{ij} = k_{ij} - \sum_{k=1}^{i-1} l_{ik} \cdot u_{kj}$$

- Nicht-Diagonalelemente von L (für $i > j$):

$$l_{ij} = \frac{k_{ij} - \sum_{k=1}^{j-1} l_{ik} \cdot u_{kj}}{u_{jj}}$$

Lineare Gleichungssysteme

Direkte Methode

LU-Zerlegung

Schritt 2: Vorwärtseinsetzen und Rückwärtseinsetzen

Mit der Zerlegung $K = L \cdot U$ können wir das Gleichungssystem $K \cdot u = F$ in zwei Schritte aufteilen:

1. **Vorwärtseinsetzen** für $L \cdot y = F$:
 - Löse das Gleichungssystem für den Zwischenvektor y .
2. **Rückwärtseinsetzen** für $U \cdot u = y$:
 - Verwende den Zwischenvektor y , um den Unbekanntenvektor u zu berechnen.

Lineare Gleichungssysteme

Direkte Methode

LU-Zerlegung

```
python Code kopieren

import numpy as np
from scipy.linalg import lu

# 6x6 Matrix K
K = np.array([
    [10, -2, 0, 0, 0, 0],
    [-2, 12, -3, 0, 0, 0],
    [0, -3, 15, -4, 0, 0],
    [0, 0, -4, 18, -5, 0],
    [0, 0, 0, -5, 20, -6],
    [0, 0, 0, 0, -6, 22]
])

# Kraftvektor F
F = np.array([1, 2, 3, 4, 5, 6])

# LU-Zerlegung der Matrix K
P, L, U = lu(K)

# Funktion für das Vorwärtseinsetzen (L * y = F)
def forward_substitution(L, b):
    """Löst L * y = b durch Vorwärtseinsetzen."""
    y = np.zeros_like(b, dtype=np.double)
    for i in range(len(b)):
        y[i] = b[i] - np.dot(L[i, :i], y[:i])
    return y

# Funktion für das Rückwärtseinsetzen (U * u = y)
def backward_substitution(U, y):
    """Löst U * u = y durch Rückwärtseinsetzen."""
    u = np.zeros_like(y, dtype=np.double)
    for i in range(len(y) - 1, -1, -1):
        u[i] = (y[i] - np.dot(U[i, i + 1:], u[i + 1:])) / U[i, i]
    return u

# Schritt 1: Vorwärtseinsetzen, um y zu berechnen
y = forward_substitution(L, F)

# Schritt 2: Rückwärtseinsetzen, um u zu berechnen
u = backward_substitution(U, y)

print("Lösung des Unbekanntenvektors u:", u)
```

Dieser Code führt die folgenden Schritte aus:

1. Zerlegt die Matrix K in L und U mit der `lu`-Funktion.
2. Berechnet den Zwischenvektor y durch Vorwärtseinsetzen.
3. Berechnet den Unbekanntenvektor u durch Rückwärtseinsetzen.

Lineare Gleichungssysteme

Direkte Methode

LU-Zerlegung

```
import numpy as np
from scipy.linalg import lu
```

```
# 6x6 Matrix K
```

```
K = np.array([
    [10, -2, 0, 0, 0, 0],
    [-2, 12, -3, 0, 0, 0],
    [0, -3, 15, -4, 0, 0],
    [0, 0, -4, 18, -5, 0],
    [0, 0, 0, -5, 20, -6],
    [0, 0, 0, 0, -6, 22]
])
```

```
# Kraftvektor F
```

```
F = np.array([1, 2, 3, 4, 5, 6])
```

```
# LU-Zerlegung der Matrix K
```

```
P, L, U = lu(K)
```

```
# Funktion für das Vorwärtseinsetzen ( $L * y = F$ )
```

```
def forward_substitution(L, b):
```

```
    """Löst  $L * y = b$  durch Vorwärtseinsetzen."""
```

```
    y = np.zeros_like(b, dtype=np.double)
```

```
    for i in range(len(b)):
```

```
        y[i] = b[i] - np.dot(L[i, :], y[:i])
```

```
    return y
```

```
# Funktion für das Rückwärtseinsetzen ( $U * u = y$ )
```

```
def backward_substitution(U, y):
```

```
    """Löst  $U * u = y$  durch Rückwärtseinsetzen."""
```

```
    u = np.zeros_like(y, dtype=np.double)
```

```
    for i in range(len(y) - 1, -1, -1):
```

```
        u[i] = (y[i] - np.dot(U[i, i + 1:], u[i + 1:])) / U[i, i]
```

```
    return u
```

```
# Schritt 1: Vorwärtseinsetzen, um y zu berechnen
```

```
y = forward_substitution(L, F)
```

```
# Schritt 2: Rückwärtseinsetzen, um u zu berechnen
```

```
u = backward_substitution(U, y)
```

```
print("Lösung des Unbekanntenvektors u:", u)
```

Lineare Gleichungssysteme

Direkte Methode

LU-Zerlegung

Eigenschaft	LU-Zerlegung	Cholesky-Zerlegung
Matrixanforderung	Beliebige quadratische, nicht-singuläre Matrix	Symmetrisch und positiv definit
Form der Zerlegung	$K = L \cdot U$	$K = L \cdot L^T$
Berechnungskosten	Höher, da zwei Matrizen berechnet werden	Geringer, da nur eine Matrix L benötigt wird
Anwendung	Allgemeine lineare Systeme	Symmetrische Systeme, besonders in FEM

Lineare Gleichungssysteme

Iterative Methode

Gauß-Seidel

Die **Gauß-Seidel-Methode** ist ein iteratives Verfahren zur Lösung linearer Gleichungssysteme der Form $A \cdot x = b$. Sie nähert sich der Lösung schrittweise an, indem die Lösung jeder Variablen iterativ aktualisiert wird. Dies ist besonders für große, dünn besetzte und diagonaldominante Matrizen effizient.

Grundidee der Gauß-Seidel-Methode

Angenommen, wir haben ein lineares Gleichungssystem:

$$A \cdot x = b$$

mit:

- A als $n \times n$ Matrix,
- $x = (x_1, x_2, \dots, x_n)^T$ als Vektor der Unbekannten,
- $b = (b_1, b_2, \dots, b_n)^T$ als Vektor der Konstanten.

Die Gauß-Seidel-Methode berechnet die Werte jeder Unbekannten sukzessive, indem sie in jedem Schritt den zuletzt berechneten Wert der aktuellen Iteration verwendet. Dies führt oft zu schnellerer Konvergenz als die Jacobi-Methode.

Lineare Gleichungssysteme

Iterative Methode

Gauß-Seidel

Iterationsformel

Für das Gleichungssystem

$$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1$$

$$a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2$$

$$\vdots$$

$$a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n$$

lässt sich die Gauß-Seidel-Methode für jede Variable x_i wie folgt schreiben:

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

Dabei:

- $x_i^{(k+1)}$ ist der neue Wert der i -ten Unbekannten in der $(k+1)$ -ten Iteration.
- $x_j^{(k+1)}$ sind die bereits aktualisierten Werte der aktuellen Iteration.
- $x_j^{(k)}$ sind die Werte der k -ten Iteration für die anderen noch nicht aktualisierten Unbekannten.

Lineare Gleichungssysteme

Iterative Methode

Gauß-Seidel

Schritte der Gauß-Seidel-Methode

1. **Initialisierung:** Starte mit einem Schätzwert für den Vektor x , oft $x^{(0)} = (0, 0, \dots, 0)^T$.
2. **Iterative Berechnung:** Verwende die obige Iterationsformel für jede Variable x_i nacheinander, um die Werte zu aktualisieren.
3. **Abbruchkriterium:** Beende die Iteration, wenn die Änderung in x zwischen den Iterationen kleiner als eine vordefinierte Toleranz ist oder eine maximale Anzahl von Iterationen erreicht wurde.

Beispiel

Angenommen, wir haben ein Gleichungssystem:

$$4x_1 - x_2 + x_3 = 7$$

$$-x_1 + 3x_2 - x_3 = 4$$

$$x_1 - x_2 + 3x_3 = 3$$

Die Iterationsformeln für die Gauß-Seidel-Methode sind:

1. Für x_1 :

$$x_1^{(k+1)} = \frac{1}{4} \left(7 + x_2^{(k)} - x_3^{(k)} \right)$$

2. Für x_2 :

$$x_2^{(k+1)} = \frac{1}{3} \left(4 + x_1^{(k+1)} + x_3^{(k)} \right)$$

3. Für x_3 :

$$x_3^{(k+1)} = \frac{1}{3} \left(3 - x_1^{(k+1)} + x_2^{(k+1)} \right)$$

Lineare Gleichungssysteme

Iterative Methode

Gauß-Seidel

Vorteile und Nachteile der Gauß-Seidel-Methode

Vorteile:

- Einfach zu implementieren.
- Kann für diagonaldominante und dünn besetzte Matrizen schnell konvergieren.

Nachteile:

- Konvergiert nur, wenn die Matrix A diagonaldominant oder positiv definit ist.
- Kann langsam sein für schlecht konditionierte Matrizen oder für große Systeme ohne diagonale Dominanz.

Lineare Gleichungssysteme

Iterative Methode

Gauß-Seidel

Dieser Code führt die folgenden Schritte aus:

1. Definiert die Matrix K und den Vektor F .
2. Implementiert die Gauß-Seidel-Methode mit einem Abbruchkriterium, das die Änderung zwischen Iterationen prüft.
3. Führt die Berechnung mit einem Startvektor von Nullen aus und gibt die berechnete Lösung x und die Anzahl der benötigten Iterationen aus.

python

 Code kopieren

```
import numpy as np

# Definition der 6x6 Matrix K
K = np.array([
    [10, -2, 0, 0, 0, 0],
    [-2, 12, -3, 0, 0, 0],
    [0, -3, 15, -4, 0, 0],
    [0, 0, -4, 18, -5, 0],
    [0, 0, 0, -5, 20, -6],
    [0, 0, 0, 0, -6, 22]
])

# Der Kraftvektor F
F = np.array([1, 2, 3, 4, 5, 6])

# Gauß-Seidel-Iterative Methode
def gauss_seidel(A, b, x0=None, tol=1e-10, max_iterations=1000):
    """Löst Ax = b mit der Gauß-Seidel-Methode."""
    n = len(b)
    x = x0 if x0 is not None else np.zeros(n) # Startwert x0 oder Nullvektor

    for k in range(max_iterations):
        x_old = x.copy()

        for i in range(n):
            sum1 = np.dot(A[i, :i], x[:i]) # Summe der vorherigen Werte
            sum2 = np.dot(A[i, i+1:], x_old[i+1:]) # Summe der "alten" Werte
            x[i] = (b[i] - sum1 - sum2) / A[i, i]

        # Überprüfe Abbruchkriterium
        if np.linalg.norm(x - x_old, ord=np.inf) < tol:
            return x, k + 1 # gibt Lösung und Anzahl der Iterationen zurück

    return x, max_iterations # gibt Lösung nach max. Iterationen zurück

# Ausführen der Gauß-Seidel-Methode mit dem Startwert x0 = [0, 0, 0, 0, 0, 0]
solution, iterations = gauss_seidel(K, F)

print("Lösung des Unbekanntenvektors x:", solution)
print("Anzahl der Iterationen:", iterations)
```

Lineare Gleichungssysteme

Iterative Methode

Gauß-Seidel

```
import numpy as np
```

```
# Definition der 6x6 Matrix K
```

```
K = np.array([
    [10, -2, 0, 0, 0, 0],
    [-2, 12, -3, 0, 0, 0],
    [0, -3, 15, -4, 0, 0],
    [0, 0, -4, 18, -5, 0],
    [0, 0, 0, -5, 20, -6],
    [0, 0, 0, 0, -6, 22]
])
```

```
# Der Kraftvektor F
```

```
F = np.array([1, 2, 3, 4, 5, 6])
```

```
# Gauß-Seidel-Iterative Methode
```

```
def gauss_seidel(A, b, x0=None, tol=1e-10, max_iterations=1000):
```

```
    """Löst  $Ax = b$  mit der Gauß-Seidel-Methode."""
```

```
    n = len(b)
```

```
    x = x0 if x0 is not None else np.zeros(n) # Startwert x0 oder Nullvektor
```

```
    for k in range(max_iterations):
```

```
        x_old = x.copy()
```

```
        for i in range(n):
```

```
            sum1 = np.dot(A[i, :i], x[:i]) # Summe der vorherigen Werte
```

```
            sum2 = np.dot(A[i, i+1:], x_old[i+1:]) # Summe der "alten" Werte
```

```
            x[i] = (b[i] - sum1 - sum2) / A[i, i]
```

```
        # Überprüfe Abbruchkriterium
```

```
        if np.linalg.norm(x - x_old, ord=np.inf) < tol:
```

```
            return x, k + 1 # gibt Lösung und Anzahl der Iterationen zurück
```

```
    return x, max_iterations # gibt Lösung nach max. Iterationen zurück
```

```
# Ausführen der Gauß-Seidel-Methode mit dem Startwert  $x_0 = [0, 0, 0, 0, 0, 0]$ 
```

```
solution, iterations = gauss_seidel(K, F)
```

```
print("Lösung des Unbekanntenvektors x:", solution)
```

```
print("Anzahl der Iterationen:", iterations)
```

Lineare Gleichungssysteme

Iterative Methode

CG-Methode

Die **Conjugate Gradient (CG) Methode** ist eine iterative Methode zur Lösung von linearen Gleichungssystemen der Form:

$$A \cdot x = b$$

Sie ist besonders effizient für **große, dünn besetzte, symmetrische und positiv definite Matrizen**, wie sie oft in der Finite-Elemente-Analyse (FEA) vorkommen. Die CG-Methode arbeitet, indem sie den Fehlervektor in aufeinanderfolgenden Schritten minimiert, ohne die Matrix direkt zu verändern oder zu speichern.

Grundprinzip der CG-Methode

Die CG-Methode verwendet die Idee der **Konjugation** von Suchrichtungen. Im Gegensatz zur Gradientenmethode, die die Richtung des Fehlers in jeder Iteration ändert, wählt die CG-Methode eine Reihe von „konjugierten“ Suchrichtungen, die unabhängig voneinander sind und die Lösung schrittweise verbessern.

Das Gleichungssystem $A \cdot x = b$

Gegeben:

- A : Symmetrische und positiv definite Matrix
- x : Unbekannter Lösungsvektor
- b : Vektor der Konstanten

Das Verfahren minimiert den Fehlervektor $r = b - A \cdot x$ in einem **Krylov-Unterraum**, der iterativ erzeugt wird.

Lineare Gleichungssysteme

Iterative Methode

CG-Methode

Iterationsformeln der CG-Methode

Die CG-Methode verwendet die folgenden Schritte und Formeln:

1. Initialisierung:

- Setze den Startwert für $x^{(0)}$, typischerweise $x^{(0)} = 0$.
- Berechne den Anfangs-Fehler $r^{(0)} = b - A \cdot x^{(0)}$.
- Setze die erste Suchrichtung: $p^{(0)} = r^{(0)}$.

2. Iterative Schritte: Für jede Iteration k :

- Berechne die Schrittweite $\alpha^{(k)}$:

$$\alpha^{(k)} = \frac{(r^{(k)})^T \cdot r^{(k)}}{(p^{(k)})^T \cdot A \cdot p^{(k)}}$$

- Aktualisiere den Lösungsvektor $x^{(k+1)}$:

$$x^{(k+1)} = x^{(k)} + \alpha^{(k)} \cdot p^{(k)}$$

- Berechne den neuen Fehlervektor $r^{(k+1)}$:

$$r^{(k+1)} = r^{(k)} - \alpha^{(k)} \cdot A \cdot p^{(k)}$$

- Überprüfe das Abbruchkriterium: Wenn $r^{(k+1)}$ klein genug ist (d. h. $\|r^{(k+1)}\| < \epsilon$), dann stoppe die Iteration.

- Berechne den Skalierungsfaktor $\beta^{(k)}$:

$$\beta^{(k)} = \frac{(r^{(k+1)})^T \cdot r^{(k+1)}}{(r^{(k)})^T \cdot r^{(k)}}$$

- Aktualisiere die Suchrichtung $p^{(k+1)}$:

$$p^{(k+1)} = r^{(k+1)} + \beta^{(k)} \cdot p^{(k)}$$

3. Wiederhole die Iterationen, bis das Abbruchkriterium erfüllt ist.

Lineare Gleichungssysteme

Iterative Methode

CG-Methode


Vorteile	Nachteile
Effizient für große, dünnbesetzte, symmetrische und positiv definite Matrizen	Funktioniert nur für symmetrische und positiv definite Matrizen
Benötigt weniger Speicher, da keine explizite Speicherung der gesamten Matrix erforderlich ist	Konvergiert langsam bei schlecht konditionierten Matrizen
Die Methode ist skalierbar und daher für sehr große Systeme geeignet	Benötigt einen Preconditioner, um bei schlecht konditionierten Systemen gut zu funktionieren
Setzt Suchrichtungen fest, die sich nicht überschneiden, und ist dadurch oft schneller als einfache Gradientenmethoden	Die Wahl eines geeigneten Startvektors kann die Konvergenz beeinflussen

Lineare Gleichungssysteme

Iterative Methode

CG-Methode

python

 Code kopieren

```
import numpy as np

# Definition der 6x6 Matrix K (symmetrisch und positiv definit)
K = np.array([
    [10, -2, 0, 0, 0, 0],
    [-2, 12, -3, 0, 0, 0],
    [0, -3, 15, -4, 0, 0],
    [0, 0, -4, 18, -5, 0],
    [0, 0, 0, -5, 20, -6],
    [0, 0, 0, 0, -6, 22]
])

# Der Kraftvektor F
F = np.array([1, 2, 3, 4, 5, 6])

# Conjugate Gradient Methode
def conjugate_gradient(A, b, x0=None, tol=1e-10, max_iterations=1000):
    """Löst Ax = b mit der Conjugate Gradient Methode."""
    n = len(b)
    x = x0 if x0 is not None else np.zeros(n) # Startwert x0 oder Nullvektor
    r = b - A.dot(x) # Initialer Fehlervektor
    p = r.copy() # Initiale Suchrichtung
    rs_old = np.dot(r, r)

    for k in range(max_iterations):
        Ap = A.dot(p)
        alpha = rs_old / np.dot(p, Ap)
        x = x + alpha * p
        r = r - alpha * Ap
        rs_new = np.dot(r, r)

        # Überprüfe Abbruchkriterium
        if np.sqrt(rs_new) < tol:
            return x, k + 1 # gibt Lösung und Anzahl der Iterationen zurück

        p = r + (rs_new / rs_old) * p
        rs_old = rs_new

    return x, max_iterations # gibt Lösung nach max. Iterationen zurück

# Ausführen der CG-Methode mit dem Startwert x0 = [0, 0, 0, 0, 0, 0]
solution, iterations = conjugate_gradient(K, F)

print("Lösung des Unbekanntenvektors x:", solution)
print("Anzahl der Iterationen:", iterations)
```

Lineare Gleichungssysteme

Iterative Methode

CG-Methode

```
import numpy as np

# Definition der 6x6 Matrix K (symmetrisch und positiv definit)
K = np.array([
    [10, -2, 0, 0, 0, 0],
    [-2, 12, -3, 0, 0, 0],
    [0, -3, 15, -4, 0, 0],
    [0, 0, -4, 18, -5, 0],
    [0, 0, 0, -5, 20, -6],
    [0, 0, 0, 0, -6, 22]
])

# Der Kraftvektor F
F = np.array([1, 2, 3, 4, 5, 6])

# Conjugate Gradient Methode
def conjugate_gradient(A, b, x0=None, tol=1e-10, max_iterations=1000):
    """Löst Ax = b mit der Conjugate Gradient Methode."""
    n = len(b)
    x = x0 if x0 is not None else np.zeros(n) # Startwert x0 oder Nullvektor
    r = b - A.dot(x) # Initialer Fehlervektor
    p = r.copy() # Initiale Suchrichtung
    rs_old = np.dot(r, r)

    for k in range(max_iterations):
        Ap = A.dot(p)
        alpha = rs_old / np.dot(p, Ap)
        x = x + alpha * p
        r = r - alpha * Ap
        rs_new = np.dot(r, r)

        # Überprüfe Abbruchkriterium
        if np.sqrt(rs_new) < tol:
            return x, k + 1 # gibt Lösung und Anzahl der Iterationen zurück

        p = r + (rs_new / rs_old) * p
        rs_old = rs_new

    return x, max_iterations # gibt Lösung nach max. Iterationen zurück

# Ausführen der CG-Methode mit dem Startwert x0 = [0, 0, 0, 0, 0, 0]
solution, iterations = conjugate_gradient(K, F)

print("Lösung des Unbekanntenvektors x:", solution)
print("Anzahl der Iterationen:", iterations)
```

Lineare Gleichungssysteme

Iterative Methode

Merkmal	Gauß-Seidel-Methode	Conjugate Gradient (CG)-Methode
Anwendbarkeit	Funktioniert für beliebige quadratische Matrizen, konvergiert jedoch nur zuverlässig bei diagonaldominanten oder positiv definiten Matrizen.	Funktioniert nur für symmetrische und positiv definite Matrizen, z. B. für die meisten Steifigkeitsmatrizen in FEM.
Konvergenzrate	Langsame Konvergenzrate, besonders bei schlecht konditionierten Matrizen und großen Systemen.	Schnellere Konvergenz als Gauß-Seidel, da konjugierte Richtungen gewählt werden, was die Iterationen effizienter macht.
Speicherbedarf	Geringer Speicherbedarf, da die Matrix nicht explizit gespeichert oder transformiert werden muss.	Geringer Speicherbedarf, da nur Vektoren und Matrix-Vektor-Produkte erforderlich sind. Kein explizites Speichern der Matrix.
Effizienz	Bei diagonaldominanten und dünn besetzten Matrizen geeignet, weniger effizient für große Systeme.	Sehr effizient bei großen, dünn besetzten, symmetrisch-positiv definiten Matrizen, wie sie in der FEA oft auftreten.
Robustheit	Konvergiert nicht immer, insbesondere wenn die Matrix nicht diagonaldominant oder schlecht konditioniert ist.	Robust für symmetrische und positiv definite Matrizen. Kann jedoch langsam konvergieren, wenn die Matrix schlecht konditioniert ist.
Preconditioner	Üblicherweise ohne Preconditioner angewendet.	Funktioniert besser mit einem Preconditioner, wenn die Matrix schlecht konditioniert ist.
Iterative Methode	Ja, verwendet einfaches Vorwärts-Iterieren der Variablen in Reihenfolge.	Ja, basiert auf Gradientenabstieg mit konjugierten Suchrichtungen, wodurch die Konvergenz effizienter wird.
Vorteile	Einfach zu implementieren; benötigt nur die Matrixelemente und den Startvektor.	Schnellere Konvergenz und weniger Iterationen als Gauß-Seidel für große FEM-Systeme.
Nachteile	Langsame Konvergenz und hohe Iterationsanzahl bei schlecht konditionierten Matrizen.	Funktioniert nur für symmetrisch positiv definite Matrizen; braucht Preconditioner für schlecht konditionierte Matrizen.

1. Positiv definite Matrix

Eine Matrix A ist **positiv definit**, wenn für jeden nicht-null Vektor x :

$$x^T A x > 0$$

Dies bedeutet, dass alle Eigenwerte der Matrix positiv sind. Positive Definitheit ist ein wichtiges Kriterium in der Conjugate Gradient Methode, da sie die Konvergenz der Methode garantiert. In der Praxis findet man positiv definite Matrizen oft in physikalischen Systemen (z. B. Steifigkeitsmatrizen in der FEM), wo sie Stabilität und Realisierbarkeit des Modells gewährleisten.

2. Singuläre Matrix

Eine **singuläre Matrix** ist eine Matrix, die **keine Inverse** besitzt. Eine Matrix ist singulär, wenn ihre Determinante null ist. Dies kann daran liegen, dass die Zeilen oder Spalten der Matrix linear abhängig sind, also nicht alle Informationen im System unabhängig voneinander sind. Singuläre Matrizen können nicht verwendet werden, um eindeutige Lösungen für lineare Gleichungssysteme zu berechnen, und treten in der Praxis auf, wenn das System überbestimmt oder schlecht definiert ist.

3. Symmetrische Matrix

Eine Matrix A ist **symmetrisch**, wenn sie gleich ihrer Transponierten ist:

$$A = A^T$$

Das bedeutet, dass die Elemente auf beiden Seiten der Hauptdiagonale identisch sind, d. h. $a_{ij} = a_{ji}$ für alle i, j . Symmetrische Matrizen treten häufig in physikalischen und mechanischen Systemen auf, z. B. bei Steifigkeitsmatrizen, da die Wechselwirkung zwischen den Komponenten in beiden Richtungen gleich ist.

4. Transponierte Matrix

Die **Transponierte** einer Matrix A wird erzeugt, indem die Zeilen von A zu Spalten werden und umgekehrt. Die transponierte Matrix von A wird als A^T geschrieben:

Wenn:

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$$

dann ist die Transponierte:

$$A^T = \begin{pmatrix} a_{11} & a_{21} \\ a_{12} & a_{22} \end{pmatrix}$$

Die Transponierte einer Matrix ist nützlich, um die Symmetrie zu überprüfen und um Matrixoperationen wie das Skalarprodukt korrekt zu berechnen.

5. Invertieren von Matrizen

Die **Inverse** einer Matrix A , bezeichnet als A^{-1} , ist die Matrix, die, wenn sie mit A multipliziert wird, die Einheitsmatrix ergibt:

$$A \cdot A^{-1} = I$$

Eine Matrix ist nur dann invertierbar, wenn sie **nicht singular** ist. Die Inversion einer Matrix wird verwendet, um lineare Gleichungssysteme zu lösen, aber für große Matrizen ist die explizite Berechnung der Inversen oft ineffizient und speicherintensiv.

6. Kondition einer Matrix

Die **Kondition** einer Matrix ist ein Maß dafür, wie empfindlich die Lösung eines linearen Gleichungssystems $A \cdot x = b$ gegenüber Änderungen in der Matrix A oder dem Vektor b ist. Eine schlecht konditionierte Matrix hat einen hohen Konditionswert und kann zu numerisch instabilen Lösungen führen. Der **Konditionswert** kann als Verhältnis des größten zum kleinsten Eigenwert betrachtet werden. Eine hohe Konditionszahl bedeutet, dass kleine Änderungen oder Rundungsfehler zu großen Änderungen in der Lösung führen können, was die Stabilität des Systems beeinträchtigt.

7. Preconditioning

Preconditioning ist eine Methode, um die Konvergenz einer iterativen Methode (wie der Conjugate Gradient Methode) bei der Lösung eines Gleichungssystems zu verbessern, insbesondere wenn die Matrix schlecht konditioniert ist. Ein Preconditioner ist eine Matrix M , die A so transformiert, dass das resultierende System besser konditioniert ist:

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

Ziel des Preconditioning ist es, die Eigenwerte der Matrix in einen Bereich zu verschieben, in dem die Iterationsmethode schneller konvergiert. Ein guter Preconditioner führt zu einem System, das numerisch stabiler ist und weniger Iterationen benötigt.

Lineare Gleichungssysteme

Iterative Methode

GMRES-Methode

Die **GMRES-Methode** (Generalized Minimal Residual Method) ist ein iteratives Verfahren zur Lösung von linearen Gleichungssystemen der Form:

$$A \cdot x = b$$

Sie wird häufig verwendet, wenn die Matrix A **groß, dünn besetzt und nicht symmetrisch** ist. Anders als die Conjugate Gradient (CG)-Methode, die nur für symmetrisch positiv definite Matrizen funktioniert, kann GMRES für allgemeine nicht-symmetrische und sogar unsymmetrische Matrizen eingesetzt werden.

Grundidee der GMRES-Methode

Die GMRES-Methode gehört zur Klasse der **Krylov-Unterraumverfahren**. Das Ziel der Methode ist es, die Lösung x durch Minimierung des **Residuumvektors** (Restvektor) $r = b - A \cdot x$ zu berechnen. GMRES erzeugt eine Approximation der Lösung, indem sie das Residuum in einem wachsenden Krylov-Unterraum minimiert, der durch die Vektoren $b, A \cdot b, A^2 \cdot b, \dots$ aufgespannt wird.

Krylov-Unterraum

Der Krylov-Unterraum der Dimension k , erzeugt durch die Matrix A und den Startvektor b , ist definiert als:

$$\mathcal{K}_k(A, b) = \text{span}\{b, A \cdot b, A^2 \cdot b, \dots, A^{k-1} \cdot b\}$$

GMRES sucht nach einer Lösung im Krylov-Unterraum, die das Residuum minimiert.

Lineare Gleichungssysteme

Iterative Methode

GMRES-Methode

1. Krylov-Unterraum

Der **Krylov-Unterraum** der Dimension k , erzeugt durch die Matrix A und den Vektor b , ist definiert als:

$$\mathcal{K}_k(A, b) = \text{span}\{b, A \cdot b, A^2 \cdot b, \dots, A^{k-1} \cdot b\}$$

GMRES sucht eine Approximation der Lösung x_k im k -dimensionalen Krylov-Unterraum $\mathcal{K}_k(A, b)$, die das Residuum minimiert.

2. Orthogonalisierung mit dem Arnoldi-Verfahren

Die Vektoren, die den Krylov-Unterraum bilden, müssen **orthogonalisiert** werden. Das Arnoldi-Verfahren erzeugt eine orthogonale Basis $V_k = [v_1, v_2, \dots, v_k]$ für $\mathcal{K}_k(A, b)$ und eine obere Hessenbergmatrix H_k , sodass:

$$A \cdot V_k \approx V_k \cdot H_k$$

wobei:

- V_k ist die Matrix der orthogonalen Basisvektoren.
- H_k ist eine $k \times k$ -Hessenbergmatrix (eine fast dreieckige Matrix mit Einträgen nur auf der Diagonale und einer Spalte darüber).

Lineare Gleichungssysteme

Iterative Methode

GMRES-Methode

3. Berechnung der Hessenbergmatrix H_k

Im j -ten Schritt des Arnoldi-Verfahrens wird der neue Basisvektor berechnet und orthogonalisiert:

$$w = A \cdot v_j$$

wobei w durch die Projektionen auf die bisherigen Basisvektoren v_1, v_2, \dots, v_j orthogonalisiert wird:

$$h_{ij} = v_i^T \cdot w, \quad w = w - h_{ij} \cdot v_i \quad \text{für } i = 1, 2, \dots, j$$

Der neue Basisvektor wird dann normalisiert, um v_{j+1} zu bilden:

$$h_{j+1,j} = \|w\|, \quad v_{j+1} = \frac{w}{h_{j+1,j}}$$

4. Minimierung des Residuals

Die GMRES-Methode minimiert das Residuum $r = b - A \cdot x$ in jedem Schritt im Krylov-Unterraum.

Die Lösung x_k im k -ten Schritt kann im Krylov-Unterraum dargestellt werden durch:

$$x_k = x_0 + V_k \cdot y_k$$

wobei y_k ein Vektor ist, der das Residuum $\|b - A \cdot (x_0 + V_k \cdot y_k)\|$ minimiert. Das minimierte Residuum ist dann:

$$\min_{y_k} \|\beta e_1 - H_k \cdot y_k\|$$

mit:

- $\beta = \|r_0\|$,
- e_1 ist der erste Einheitsvektor.

Lineare Gleichungssysteme

Iterative Methode

GMRES-Methode

5. Lösung des kleineren linearen Systems

Das Problem reduziert sich auf die Lösung des kleineren, einfacheren Gleichungssystems:

$$H_k \cdot y_k = \beta e_1$$

Sobald y_k gefunden ist, kann x_k berechnet werden als:

$$x_k = x_0 + V_k \cdot y_k$$

6. Restarted GMRES

Da der Krylov-Unterraum V_k bei jeder Iteration wächst, benötigt GMRES mehr Speicher. Die **Restarted GMRES**-Variante, auch GMRES(m) genannt, „startet neu“ nach m Iterationen mit dem aktuellen x_k als Startwert und reduziert so den Speicherbedarf.

Zusammenfassung der wichtigsten Formeln

1. Krylov-Unterraum:

$$\mathcal{K}_k(A, b) = \text{span}\{b, A \cdot b, A^2 \cdot b, \dots, A^{k-1} \cdot b\}$$

2. Arnoldi-Verfahren:

$$A \cdot V_k \approx V_k \cdot H_k$$

3. Residualminimierung:

$$\min_{y_k} \|\beta e_1 - H_k \cdot y_k\|$$

4. Lösungsberechnung:

$$x_k = x_0 + V_k \cdot y_k$$

Lineare Gleichungssysteme

Iterative Methode

GMRES-Methode

Vorteile und Nachteile der GMRES-Methode

Vorteile	Nachteile
Funktioniert für nicht-symmetrische Matrizen.	Hoher Speicherbedarf, da der Krylov-Unterraum wächst.
Minimiert das Residuum in jedem Schritt.	Jede Iteration erfordert die Speicherung aller Basisvektoren.
Effektiv bei dünn besetzten, großen Matrizen.	Kann langsam konvergieren, wenn die Matrix schlecht konditioniert ist.
Kann mit einem Preconditioner kombiniert werden.	Die Methode muss periodisch neu gestartet werden (z. B. „GMRES with Restart“), um Speicherbedarf zu begrenzen.

GMRES with Restart

Da der Krylov-Unterraum bei jeder Iteration wächst und mehr Speicher benötigt, wird oft die **Restart-Variante** verwendet. Hier wird nach einer festen Anzahl von Iterationen (z. B. 30 oder 50) der Algorithmus „neu gestartet“ mit dem aktuellen x_k als Startwert, um den Speicherbedarf zu reduzieren.

Anwendungen der GMRES-Methode

Die GMRES-Methode wird vor allem in der numerischen linearen Algebra und Finite-Elemente-Analyse eingesetzt, wenn die zu lösende Matrix nicht symmetrisch ist, wie z. B. bei Problemen in der Strömungsmechanik oder Wärmeleitung.

Eigenwertprobleme

Eigenwertprobleme in der Finite-Elemente-Methode (FEM) sind zentrale Aufgaben, besonders in der Strukturmechanik und Schwingungsanalyse. Sie helfen, **natürliche Frequenzen**, **Eigenformen** (Modenformen) und **Stabilitätskriterien** von Strukturen zu bestimmen. Die natürlichen Frequenzen zeigen, wie eine Struktur auf externe Kräfte reagiert und ob es Resonanzphänomene geben kann, die zu Strukturversagen führen.

Eigenwertprobleme in der FEM

Das allgemeine Eigenwertproblem in der FEM lautet:

$$K \cdot u = \lambda \cdot M \cdot u$$

wobei:

- K die **Steifigkeitsmatrix** ist, die die Starrheit der Struktur beschreibt,
- M die **Massenmatrix** ist, die die Masseneigenschaften der Struktur repräsentiert,
- u die **Eigenvektoren** sind, die die Modenformen beschreiben,
- λ die **Eigenwerte** sind, die quadratisch zur Frequenz stehen ($\lambda = \omega^2$).

Dieses verallgemeinerte Eigenwertproblem tritt auf, wenn sowohl K als auch M Matrizen enthalten, die die Steifigkeit und Masse der Struktur beschreiben.

Wichtige Formeln im Eigenwertproblem

1. Allgemeines verallgemeinertes Eigenwertproblem:

$$K \cdot u = \lambda \cdot M \cdot u$$

2. Normales Eigenwertproblem (wenn $M = I$, die Einheitsmatrix):

$$K \cdot u = \lambda \cdot u$$

3. Berechnung der Eigenfrequenz: Die Eigenfrequenz ist über den Eigenwert λ gegeben durch:

$$\omega = \sqrt{\lambda}$$

und die Frequenz f ist dann:

$$f = \frac{\omega}{2\pi}$$

Eigenwertprobleme

Direkte Methode

Direkte und iterative Löser für Eigenwertprobleme

Je nach Größe und Struktur des Gleichungssystems werden unterschiedliche Lösungsverfahren eingesetzt:

Direkte Eigenwertlöser

1. QR-Zerlegung:

- Die QR-Zerlegung zerlegt die Matrix in orthogonale und obere Dreiecksmatrizen und berechnet die Eigenwerte durch iteratives Verfahren.
- Liefert genaue Lösungen, ist aber speicherintensiv und rechenaufwendig, daher geeignet für kleinere bis mittelgroße Matrizen.

2. Householder-Transformation und Givens-Rotation:

- Diese Methoden transformieren die Matrix in eine Hessenberg-Form, die mit der QR-Methode weiter verarbeitet wird.
- Für dichte Matrizen konzipiert; speicher- und rechenintensiv, weniger für FEM-typische große dünnbesetzte Matrizen geeignet.

Eigenwertprobleme

Iterative Methode

Iterative Eigenwertlöser

Iterative Verfahren sind für große und dünnbesetzte Matrizen wie in der FEM optimiert, da sie eine Auswahl der gewünschten Eigenwerte und Eigenvektoren effizient berechnen.

1. Unterraum-Methode (Subspace Iteration Method):

- Diese Methode berechnet mehrere Eigenwerte und Eigenvektoren gleichzeitig, indem sie einen Unterraum der Matrix iteriert.
- Zunächst wird ein Satz von Startvektoren definiert, die den gewünschten Unterraum aufspannen. Diese Vektoren werden iterativ über die Matrizenoperation aktualisiert und orthogonalisiert, bis sie konvergieren.
- Nach der Konvergenz wird ein kleineres Eigenwertproblem im erzeugten Unterraum gelöst, um die sogenannten **Ritz-Werte** (approximierte Eigenwerte) zu berechnen.
- Besonders geeignet für die niedrigsten Eigenfrequenzen und Modenformen in großen FEM-Systemen, die parallelisiert werden können.

2. Lanczos-Algorithmus:

- Ein Krylov-Unterraum-Verfahren, das eine schrittweise Approximation der größten oder kleinsten Eigenwerte und der zugehörigen Eigenvektoren liefert.
- Sehr effizient für große, dünn besetzte und symmetrische Matrizen, wie sie in der FEM häufig auftreten.
- Der Algorithmus erzeugt eine reduzierte tridiagonale Matrix, die leichter zu handhaben ist, und berechnet eine Reihe von Eigenwerten durch Projektion auf diesen Krylov-Unterraum.
- Wird oft mit Preconditioning kombiniert, um die Konvergenz zu beschleunigen.

Eigenwertprobleme

Iterative Methode

3. Arnoldi-Verfahren:

- Eine Verallgemeinerung des Lanczos-Algorithmus, die auch für nicht-symmetrische Matrizen funktioniert.
- Baut eine orthogonale Basis des Krylov-Unterraums auf und verwendet das Resultat zur Berechnung der führenden Eigenwerte und Eigenvektoren.
- Nützlich, wenn nur die führenden Eigenwerte oder eine bestimmte Anzahl von Eigenwerten benötigt werden.

4. Inverse Iteration und Rayleigh-Quotienten-Iteration:

- Die Inverse Iteration nutzt die Inverse der Matrix und konvergiert schnell zu einem bestimmten Eigenwert in der Nähe eines Startwerts.
- Die Rayleigh-Quotienten-Iteration kombiniert die Inverse Iteration mit einer genaueren Näherung und ist besonders effizient bei symmetrischen Matrizen.
- Diese Methode ist geeignet, wenn eine gute Anfangsnäherung vorhanden ist und nur wenige Eigenwerte in einem bestimmten Bereich gesucht werden.

Zusammenfassung der Löser

Methode	Typ	Vorteile	Nachteile
QR-Zerlegung	Direkt	Liefert alle Eigenwerte, genau	Sehr speicher- und rechenintensiv
Householder & Givens	Direkt	Geeignet für dichte Matrizen	Nicht geeignet für große, dünnbesetzte Matrizen
Unterraum-Methode	Iterativ	Berechnet mehrere Eigenwerte parallel, gut für FEM	Konvergenz kann langsam sein
Lanczos-Algorithmus	Iterativ	Effizient für große, dünnbesetzte, symmetrische Matrizen	Rechnet nur einige Eigenwerte
Arnoldi-Verfahren	Iterativ	Geeignet für nicht-symmetrische Matrizen	Komplexer und speicherintensiver als Lanczos
Inverse Iteration & Rayleigh	Iterativ	Schnell konvergent, wenn gute Näherung vorhanden	Benötigt Inversion der Matrix, geeignet für kleine Systeme

Anwendung in der FEM

Die Wahl der Methode hängt von der Größe und Struktur der Matrizen sowie vom gewünschten Eigenwertbereich ab. **Iterative Methoden** wie die **Unterraum-Methode** und der **Lanczos-Algorithmus** sind bevorzugte Wahl bei großen FEM-Problemen, da sie speziell für dünnbesetzte Matrizen und die Berechnung von Eigenwerten und Eigenvektoren im interessierenden Bereich (wie die niedrigsten Frequenzen) optimiert sind.

Eigenwertprobleme

Iterative Methode

Grundidee der Unterraum-Methode

Die Unterraum-Methode verwendet einen Vektorsatz (Unterraum), der gleichzeitig iteriert und orthogonalisiert wird, um mehrere Eigenwerte und Eigenvektoren parallel zu berechnen. Diese Methode ist besonders effizient, wenn nur die niedrigsten Eigenwerte und Eigenvektoren (Grundfrequenzen und Modenformen) eines Systems von Interesse sind.

Zusammenfassung der wichtigen Formeln

1. Berechnung des Unterraums:

$$Y^{(k+1)} = K^{-1} M X^{(k)}$$

2. Orthogonalisierung (QR-Zerlegung):

$$Y^{(k+1)} = Q^{(k+1)} R^{(k+1)}, \quad X^{(k+1)} = Q^{(k+1)}$$

3. Ritz-Werte und Ritz-Vektoren:

- Reduziertes Eigenwertproblem:

$$T = (X^{(k)})^T K X^{(k)}, \quad S = (X^{(k)})^T M X^{(k)}$$

$$T \cdot z = \lambda S \cdot z$$

- Rückprojektion zur Berechnung der Ritz-Vektoren:

$$u = X^{(k)} \cdot z$$

Die Unterraum-Methode ist ein effektives Werkzeug zur Berechnung der niedrigsten Eigenwerte und Eigenvektoren bei großen FEM-Systemen und wird oft für Frequenz- und Stabilitätsanalysen in der Strukturmechanik eingesetzt.

Eigenwertprobleme

Iterative Methode

Schritte und Formeln der Unterraum-Methode

1. Initialisierung

Wähle einen Satz von Startvektoren $\{x_1, x_2, \dots, x_p\}$, der den Unterraum für die gewünschten Eigenvektoren aufspannt. Typischerweise sind diese Startvektoren zufällig gewählt, und p entspricht der Anzahl der Eigenwerte, die berechnet werden sollen.

Definiere:

- $X^{(0)} = [x_1, x_2, \dots, x_p]$, die Startmatrix aus den Vektoren x_i .

2. Iterative Berechnung des Unterraums

In jedem Iterationsschritt k :

1. Berechne den aktualisierten Vektorsatz Y durch die Matrixoperationen:

$$Y^{(k+1)} = K^{-1} M X^{(k)}$$

wobei K^{-1} die Inverse der Steifigkeitsmatrix ist. Dies ist die Kernoperation der Methode, die den Unterraum in die richtige Richtung lenkt. Alternativ wird oft eine approximative Lösung für $K \cdot Y^{(k+1)} = M X^{(k)}$ verwendet, um die Inversion zu vermeiden.

2. Orthogonalisierung:

- Orthogonalisiere den neuen Vektorsatz $Y^{(k+1)}$, um lineare Abhängigkeit zwischen den Vektoren zu verhindern. Die **QR-Zerlegung** wird oft verwendet, um eine orthogonale Matrix Q zu erhalten:

$$Y^{(k+1)} = Q^{(k+1)} R^{(k+1)}$$

- Setze $X^{(k+1)} = Q^{(k+1)}$ für die nächste Iteration.

Eigenwertprobleme

Iterative Methode

3. Berechnung der Ritz-Werte und Ritz-Vektoren

Nach der Konvergenz der Iterationen wird der Unterraum auf ein kleineres Eigenwertproblem reduziert:

1. Berechne die Ritz-Werte (approximierte Eigenwerte), indem du das reduzierte Eigenwertproblem in $X^{(k)}$ löst:

$$T = (X^{(k)})^T K X^{(k)}, \quad S = (X^{(k)})^T M X^{(k)}$$

Dies führt zu einem kleineren Eigenwertproblem:

$$T \cdot z = \lambda S \cdot z$$

wobei λ die gesuchten Ritz-Werte sind.

2. Die **Ritz-Vektoren** (approximierte Eigenvektoren) werden dann berechnet, indem der reduzierte Eigenvektor z auf den ursprünglichen Unterraum zurückprojiziert wird:

$$u = X^{(k)} \cdot z$$

Damit sind die Ritz-Werte λ die gesuchten Eigenwerte und die Ritz-Vektoren u die gesuchten Eigenvektoren des Originalproblems.

4. Abbruchkriterium

Die Iteration wird beendet, wenn die Änderung der Eigenwerte und Eigenvektoren zwischen den Iterationen kleiner als eine vorgegebene Toleranz ist.

Eigenwertprobleme

Iterative Methode

Erklärung des Codes

1. Initialisierung:

- Ein zufälliger Start-Unterraum X wird definiert, der $p = 2$ Dimensionen hat, da wir nur die niedrigsten zwei Eigenwerte berechnen möchten.

2. Iteration:

- Berechne $Y = K^{-1}MX$ durch Lösen des Gleichungssystems $K \cdot Y = M \cdot X$.
- Verwende die QR-Zerlegung zur Orthogonalisierung des neuen Vektorsatzes Y , um linear abhängige Vektoren zu vermeiden.

3. Reduziertes Eigenwertproblem:

- Das reduzierte Eigenwertproblem wird gelöst, indem die Ritz-Werte und Ritz-Vektoren (approximierte Eigenwerte und Eigenvektoren) im Unterraum Q berechnet werden.
- Die Ritz-Eigenvektoren $X_{\text{new}} = Q \cdot Z$ werden als Näherungen der Eigenvektoren des Originalproblems verwendet.

4. Abbruchkriterium:

- Die Iteration wird beendet, wenn sich die Lösung stabilisiert und die Änderung zwischen den Iterationen unter der Toleranz tol liegt.

5. Sortierung und Ausgabe:

- Die Eigenwerte und Eigenvektoren werden sortiert, und die kleinsten p Werte werden ausgegeben.

```
python
import numpy as np
from scipy.linalg import qr

# Definition der 6x6 Steifigkeitsmatrix K und der Massenmatrix M
K = np.array([
    [10, -2, 0, 0, 0, 0],
    [-2, 12, -3, 0, 0, 0],
    [0, -3, 15, -4, 0, 0],
    [0, 0, -4, 18, -5, 0],
    [0, 0, 0, -5, 20, -6],
    [0, 0, 0, 0, -6, 22]
])

M = np.array([
    [2, 0, 0, 0, 0, 0],
    [0, 2, 0, 0, 0, 0],
    [0, 0, 2, 0, 0, 0],
    [0, 0, 0, 2, 0, 0],
    [0, 0, 0, 0, 2, 0],
    [0, 0, 0, 0, 0, 2]
])

# Parameter
tol = 1e-6 # Toleranz für Konvergenz
max_iterations = 100 # Maximale Anzahl der Iterationen
p = 3 # Anzahl der zu berechnenden Eigenwerte (Unterraumdimension)

# Schritt 1: Initialisierung des Startunterraums mit zufälligen Vektoren
np.random.seed(0)
X = np.random.rand(K.shape[0], p)

for iteration in range(max_iterations):
    # Schritt 2: Berechne Y = K^-1 * M * X
    # Statt die Inverse zu berechnen, lösen wir das Gleichungssystem K * Y = M * X
    Y = np.linalg.solve(K, M @ X)

    # Schritt 3: Orthogonalisierung von Y (QR-Zerlegung)
    Q, R = qr(Y, mode='economic')

    # Schritt 4: Reduziertes Eigenwertproblem
    T = Q.T @ K @ Q
    S = Q.T @ M @ Q

    # Eigenwerte und Eigenvektoren des reduzierten Problems
    eigenvalues, Z = np.linalg.eig(np.linalg.solve(S, T))

    # Ritz-Eigenvektoren des Originalproblems
    X_new = Q @ Z

    # Abbruchkriterium: Prüfe die Konvergenz der Eigenvektoren
    if np.linalg.norm(X - X_new) < tol:
        print("Konvergenz erreicht nach", iteration + 1, "Iterationen")
        break

    # Aktualisiere den Unterraum für die nächste Iteration
    X = X_new

# Sortiere die Eigenwerte und die zugehörigen Eigenvektoren
sorted_indices = np.argsort(eigenvalues)
eigenvalues = eigenvalues[sorted_indices]
eigenvectors = X[:, sorted_indices]

# Ausgabe der berechneten Eigenwerte und Eigenvektoren
print("Berechnete Eigenwerte:", eigenvalues[:p])
print("Berechnete Eigenvektoren (spaltenweise):\n", eigenvectors[:, :p])
```

Eigenwertprobleme

Iterative Methode

```
import numpy as np
from scipy.linalg import qr

# Definition der 6x6 Steifigkeitsmatrix K und der Massenmatrix M
K = np.array([
    [10, -2, 0, 0, 0, 0],
    [-2, 12, -3, 0, 0, 0],
    [0, -3, 15, -4, 0, 0],
    [0, 0, -4, 18, -5, 0],
    [0, 0, 0, -5, 20, -6],
    [0, 0, 0, 0, -6, 22]
])

M = np.array([
    [2, 0, 0, 0, 0, 0],
    [0, 2, 0, 0, 0, 0],
    [0, 0, 2, 0, 0, 0],
    [0, 0, 0, 2, 0, 0],
    [0, 0, 0, 0, 2, 0],
    [0, 0, 0, 0, 0, 2]
])

# Parameter
tol = 1e-6      # Toleranz für Konvergenz
max_iterations = 100 # Maximale Anzahl der Iterationen
p = 3          # Anzahl der zu berechnenden Eigenwerte (Unterraumdimension)

# Schritt 1: Initialisierung des Startunterraums mit zufälligen Vektoren
np.random.seed(0)
X = np.random.rand(K.shape[0], p)

for iteration in range(max_iterations):
    # Schritt 2: Berechne  $Y = K^{-1} M X$ 
    # Statt die Inverse zu berechnen, lösen wir das Gleichungssystem  $K * Y = M * X$ 
    Y = np.linalg.solve(K, M @ X)

    # Schritt 3: Orthogonalisierung von Y (QR-Zerlegung)
    Q, R = qr(Y, mode='economic')

    # Schritt 4: Reduziertes Eigenwertproblem
    T = Q.T @ K @ Q
    S = Q.T @ M @ Q

    # Eigenwerte und Eigenvektoren des reduzierten Problems
    eigenvalues, Z = np.linalg.eig(np.linalg.solve(S, T))

    # Ritz-Eigenvektoren des Originalproblems
    X_new = Q @ Z

    # Abbruchkriterium: Prüfe die Konvergenz der Eigenvektoren
    if np.linalg.norm(X - X_new) < tol:
        print("Konvergenz erreicht nach", iteration + 1, "Iterationen")
        break

    # Aktualisiere den Unterraum für die nächste Iteration
    X = X_new

# Sortiere die Eigenwerte und die zugehörigen Eigenvektoren
sorted_indices = np.argsort(eigenvalues)
eigenvalues = eigenvalues[sorted_indices]
eigenvectors = X[:, sorted_indices]

# Ausgabe der berechneten Eigenwerte und Eigenvektoren
print("Berechnete Eigenwerte:", eigenvalues[:p])
print("Berechnete Eigenvektoren (spaltenweise):\n", eigenvectors[:, :p])
```


Eigenwertprobleme

Direkte Methode

Erklärung des Codes:

1. **Definieren der Matrizen K und M :** Die Steifigkeitsmatrix K und die Massenmatrix M werden als 6x6-Matrizen definiert.
2. **Verallgemeinertes Eigenwertproblem lösen:** Die Funktion `np.linalg.solve(M, K)` berechnet $M^{-1}K$, und `np.linalg.eig()` berechnet die Eigenwerte und Eigenvektoren.
3. **Sortieren der Eigenwerte und Eigenvektoren:** Die Eigenwerte werden in aufsteigender Reihenfolge sortiert, um die niedrigsten Eigenwerte zuerst auszugeben.
4. **Ausgabe:** Die exakten Eigenwerte werden angezeigt.

python

 Code kopieren

```
import numpy as np

# Definition der 6x6 Steifigkeitsmatrix K und der Massenmatrix M
K = np.array([
    [10, -2, 0, 0, 0, 0],
    [-2, 12, -3, 0, 0, 0],
    [0, -3, 15, -4, 0, 0],
    [0, 0, -4, 18, -5, 0],
    [0, 0, 0, -5, 20, -6],
    [0, 0, 0, 0, -6, 22]
])

M = np.array([
    [2, 0, 0, 0, 0, 0],
    [0, 2, 0, 0, 0, 0],
    [0, 0, 2, 0, 0, 0],
    [0, 0, 0, 2, 0, 0],
    [0, 0, 0, 0, 2, 0],
    [0, 0, 0, 0, 0, 2]
])

# Berechnung der exakten Eigenwerte und Eigenvektoren des verallgemeinerten Eigenwertprobl
eigenvalues, eigenvectors = np.linalg.eig(np.linalg.solve(M, K))

# Sortiere die Eigenwerte und die zugehörigen Eigenvektoren
sorted_indices = np.argsort(eigenvalues)
eigenvalues = eigenvalues[sorted_indices]
eigenvectors = eigenvectors[:, sorted_indices]

# Ausgabe der berechneten Eigenwerte
print("Exakte Eigenwerte:", eigenvalues)
```

Eigenwertprobleme

Direkte Methode

```
import numpy as np

# Definition der 6x6 Steifigkeitsmatrix K und der Massenmatrix M
K = np.array([
    [10, -2, 0, 0, 0, 0],
    [-2, 12, -3, 0, 0, 0],
    [0, -3, 15, -4, 0, 0],
    [0, 0, -4, 18, -5, 0],
    [0, 0, 0, -5, 20, -6],
    [0, 0, 0, 0, -6, 22]
])

M = np.array([
    [2, 0, 0, 0, 0, 0],
    [0, 2, 0, 0, 0, 0],
    [0, 0, 2, 0, 0, 0],
    [0, 0, 0, 2, 0, 0],
    [0, 0, 0, 0, 2, 0],
    [0, 0, 0, 0, 0, 2]
])

# Berechnung der exakten Eigenwerte und Eigenvektoren des verallgemeinerten
Eigenwertproblems
eigenvalues, eigenvectors = np.linalg.eig(np.linalg.solve(M, K))

# Sortiere die Eigenwerte und die zugehörigen Eigenvektoren
sorted_indices = np.argsort(eigenvalues)
eigenvalues = eigenvalues[sorted_indices]
eigenvectors = eigenvectors[:, sorted_indices]

# Ausgabe der berechneten Eigenwerte
print("Exakte Eigenwerte:", eigenvalues)
```

Eigenwertprobleme

- Exakte Eigenwerte (alle sechs): $[4.0260, 5.1527, 6.3351, 8.1033, 10.6928, 14.1900]$
- Eigenwerte aus der Unterraum-Methode (drei niedrigste): $[4.0260, 5.1527, 6.3351]$

Nicht-Lineare Gleichungssysteme

Nichtlineare Gleichungssysteme sind Gleichungen, bei denen die Unbekannten in einer nichtlinearen Weise auftreten. In der FEM und Strukturmechanik entstehen solche Gleichungssysteme häufig aufgrund von Material- oder Geometrienichtlinearitäten, z. B. bei großen Verformungen, Kontaktproblemen oder plastischem Verhalten.

Ein **nichtlineares Gleichungssystem** kann allgemein geschrieben werden als:

$$F(x) = 0$$

wobei $F(x)$ eine nichtlineare Funktion von x ist und die Lösung x gesucht wird, die diese Gleichung erfüllt.

Nicht-Lineare Gleichungssysteme

Beispielhaftes nichtlineares Gleichungssystem für eine 6x6 Matrix in der FEM

Ein solches System kann als **nichtlineares Gleichungssystem** formuliert werden, bei dem die interne Kraft $F_{\text{int}}(u)$ von den Verschiebungen u abhängt und in Abhängigkeit von der äußeren Kraft F_{ext} berechnet wird.

Das Gleichgewicht lautet:

$$F_{\text{int}}(u) = F_{\text{ext}}$$

Da $F_{\text{int}}(u)$ nichtlinear in den Verschiebungen u ist, entsteht ein nichtlineares Gleichungssystem. Eine exemplarische nichtlineare Form könnte folgendermaßen aussehen:

$$K(u) \cdot u = F_{\text{ext}}$$

Dabei ist $K(u)$ die **Tangentsteifigkeitsmatrix**, die von u abhängt und sich während der Verformung verändert.

Beispielmatrix für das nichtlineare Gleichungssystem

Angenommen, die Tangentsteifigkeitsmatrix $K(u)$ ist:

$$K(u) = \begin{pmatrix} 10 + 0.5u_1 & -2 & 0 & 0 & 0 & 0 \\ -2 & 12 + 0.3u_2 & -3 & 0 & 0 & 0 \\ 0 & -3 & 15 + 0.4u_3 & -4 & 0 & 0 \\ 0 & 0 & -4 & 18 + 0.2u_4 & -5 & 0 \\ 0 & 0 & 0 & -5 & 20 + 0.1u_5 & -6 \\ 0 & 0 & 0 & 0 & -6 & 22 + 0.3u_6 \end{pmatrix}$$

Hier hängt jeder Diagonaleintrag nichtlinear von den Verschiebungen u_i ab.

Nicht-Lineare Gleichungssysteme

Externes Kraftvektorelement

Ein Beispiel für den externen Kraftvektor F_{ext} könnte sein:

$$F_{\text{ext}} = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{pmatrix}$$

Das nichtlineare Gleichungssystem

Unser Gleichungssystem lautet dann:

$$K(u) \cdot u = F_{\text{ext}}$$

oder in expliziter Form:

$$\begin{pmatrix} 10 + 0.5u_1 & -2 & 0 & 0 & 0 & 0 \\ -2 & 12 + 0.3u_2 & -3 & 0 & 0 & 0 \\ 0 & -3 & 15 + 0.4u_3 & -4 & 0 & 0 \\ 0 & 0 & -4 & 18 + 0.2u_4 & -5 & 0 \\ 0 & 0 & 0 & -5 & 20 + 0.1u_5 & -6 \\ 0 & 0 & 0 & 0 & -6 & 22 + 0.3u_6 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{pmatrix}$$

Nicht-Lineare Gleichungssysteme

Iterative Methode

Newton-Raphson

1. Newton-Raphson-Verfahren

Das **Newton-Raphson-Verfahren** ist ein iteratives Verfahren zur Lösung nichtlinearer Gleichungssysteme und basiert auf der Annäherung der Funktion durch eine Taylorreihe. In der Strukturmechanik wird es verwendet, um das Gleichgewicht einer Struktur unter Last zu erreichen.

Formeln und Algorithmus

1. Gegeben ein nichtlineares Gleichungssystem $F(x) = 0$, nähert das Newton-Raphson-Verfahren die Lösung iterativ an.
2. In jedem Schritt wird die Funktion $F(x)$ um den aktuellen Punkt $x^{(k)}$ linearisiert:

$$x^{(k+1)} = x^{(k)} - J^{-1}(x^{(k)}) \cdot F(x^{(k)})$$

wobei:

- $J(x^{(k)}) = \frac{\partial F}{\partial x}$ die **Jacobi-Matrix** (Ableitungsmatrix) von F am Punkt $x^{(k)}$ ist,
 - $F(x^{(k)})$ der Wert der Funktion bei der aktuellen Annäherung $x^{(k)}$.
3. Die Iteration wird solange wiederholt, bis $F(x^{(k+1)}) \approx 0$ ist oder die Änderung in x unter einer bestimmten Toleranz liegt.

Vorteile und Nachteile

Vorteile	Nachteile
Quadratische Konvergenz nahe der Lösung (schnell konvergent)	Benötigt die Berechnung und Invertierung der Jacobi-Matrix, was aufwendig ist
Gut geeignet für stark nichtlineare Probleme	Kann bei schlechten Startwerten divergieren
Stabil und präzise bei gut konditionierten Problemen	Nicht immer effizient bei großen Problemen mit dünn besetzten Matrizen

Nicht-Lineare Gleichungssysteme

Iterative Methode

Newton-Raphson

```
python Code kopieren

import numpy as np

# Definition des externen Kraftvektors
F_ext = np.array([1, 2, 3, 4, 5, 6])

# Parameter für das Newton-Raphson-Verfahren
tol = 1e-6          # Toleranz für Konvergenz
max_iterations = 50 # Maximale Anzahl der Iterationen

# Startwert für die Verschiebungen
u = np.zeros(6)

# Funktion zur Berechnung der Tangentsteifigkeitsmatrix K(u)
def tangent_stiffness_matrix(u):
    return np.array([
        [10 + 0.5 * u[0], -2, 0, 0, 0, 0],
        [-2, 12 + 0.3 * u[1], -3, 0, 0, 0],
        [0, -3, 15 + 0.4 * u[2], -4, 0, 0],
        [0, 0, -4, 18 + 0.2 * u[3], -5, 0],
        [0, 0, 0, -5, 20 + 0.1 * u[4], -6],
        [0, 0, 0, 0, -6, 22 + 0.3 * u[5]]
    ])

# Newton-Raphson-Iteration
for iteration in range(max_iterations):
    # Berechne Tangentsteifigkeitsmatrix K(u) und internen Kraftvektor F_int = K(u) * u
    K_u = tangent_stiffness_matrix(u)
    F_int = K_u @ u

    # Berechne Restkraftvektor R = F_ext - F_int
    R = F_ext - F_int

    # Überprüfe Konvergenz
    if np.linalg.norm(R) < tol:
        print("Konvergenz erreicht nach", iteration + 1, "Iterationen.")
        break

    # Löse für Inkrement Delta_u
    delta_u = np.linalg.solve(K_u, R)

    # Aktualisiere Verschiebungsvektor u
    u = u + delta_u

# Ausgabe der berechneten Verschiebungen
print("Berechnete Verschiebungen u:", u)
```

Für dieses nichtlineare System kann das **Newton-Raphson-Verfahren** verwendet werden:

1. Starte mit einer Näherung $u^{(0)}$.
2. Berechne die Tangentsteifigkeitsmatrix $K(u^{(k)})$ und den Restkraftvektor $R = F_{\text{ext}} - K(u^{(k)}) \cdot u^{(k)}$.
3. Aktualisiere u iterativ durch:

$$u^{(k+1)} = u^{(k)} + \Delta u$$

wobei Δu aus dem linearen Gleichungssystem berechnet wird:

$$K(u^{(k)}) \cdot \Delta u = F_{\text{ext}} - K(u^{(k)}) \cdot u^{(k)}$$

Die Iteration wird wiederholt, bis die Änderung in u klein genug ist oder das Gleichgewicht erreicht ist. Dieses Beispiel zeigt die typischen nichtlinearen Effekte, die in großen FEM-Systemen auftreten, bei denen geometrische oder Materialverformungen nichtlinear sind.

Erklärung des Codes:

1. **Initialisierung:**
 - Der externe Kraftvektor F_{ext} und der Startwert für die Verschiebungen u (Nullvektor) werden festgelegt.
2. **Funktion `tangent_stiffness_matrix(u)`:**
 - Diese Funktion berechnet die Tangentsteifigkeitsmatrix $K(u)$, die von u abhängt und bei jeder Iteration aktualisiert wird.
3. **Newton-Raphson-Iteration:**
 - Berechnung des Restkraftvektors $R = F_{\text{ext}} - K(u) \cdot u$.
 - Überprüfung der Konvergenzbedingung anhand der Norm von R .
 - Berechnung des Inkrements Δu und Aktualisierung von u .
4. **Abbruchkriterium:**
 - Die Iteration wird beendet, wenn R kleiner als die Toleranz ist oder die maximale Anzahl von Iterationen erreicht wurde.

Am Ende gibt der Code die berechneten Verschiebungen u aus.

Nicht-Lineare Gleichungssysteme

Iterative Methode

Newton-Raphson

```
import numpy as np

# Definition des externen Kraftvektors
F_ext = np.array([1, 2, 3, 4, 5, 6])

# Parameter für das Newton-Raphson-Verfahren
tol = 1e-6      # Toleranz für Konvergenz
max_iterations = 50 # Maximale Anzahl der Iterationen

# Startwert für die Verschiebungen
u = np.zeros(6)

# Funktion zur Berechnung der Tangentsteifigkeitsmatrix K(u)
def tangent_stiffness_matrix(u):
    return np.array([
        [10 + 0.5 * u[0], -2, 0, 0, 0, 0],
        [-2, 12 + 0.3 * u[1], -3, 0, 0, 0],
        [0, -3, 15 + 0.4 * u[2], -4, 0, 0],
        [0, 0, -4, 18 + 0.2 * u[3], -5, 0],
        [0, 0, 0, -5, 20 + 0.1 * u[4], -6],
        [0, 0, 0, 0, -6, 22 + 0.3 * u[5]]
    ])

# Newton-Raphson-Iteration
for iteration in range(max_iterations):
    # Berechne Tangentsteifigkeitsmatrix K(u) und internen Kraftvektor F_int = K(u) * u
    K_u = tangent_stiffness_matrix(u)
    F_int = K_u @ u

    # Berechne Restkraftvektor R = F_ext - F_int
    R = F_ext - F_int

    # Überprüfe Konvergenz
    if np.linalg.norm(R) < tol:
        print("Konvergenz erreicht nach", iteration + 1, "Iterationen.")
        break

    # Löse für Inkrement Delta_u
    delta_u = np.linalg.solve(K_u, R)

    # Aktualisiere Verschiebungsvektor u
    u = u + delta_u

# Ausgabe der berechneten Verschiebungen
print("Berechnete Verschiebungen u:", u)
```

Nicht-Lineare Gleichungssysteme

Iterative Methode

Newton-Raphson

Erklärung der Jacobi-Matrix-Berechnung

In diesem Codebeispiel wird die Tangentsteifigkeitsmatrix (Jacobi-Matrix) $K(u)$ von den Verschiebungen u abhängig gemacht. Die Matrix wird in jeder Iteration des Newton-Raphson-Verfahrens neu berechnet, was die Änderungen in der Steifigkeit der Struktur aufgrund der Verschiebungen berücksichtigt.

1. **Nichtlineare Terme in $K(u)$:** Die Diagonaleinträge enthalten Terme wie `10 + 0.5 * u[0]`, die von den Verschiebungen u abhängen. Das führt dazu, dass sich $K(u)$ mit jeder Änderung in u verändert.
2. **Rückgabe der Tangentsteifigkeitsmatrix:** Die Funktion `tangent_stiffness_matrix(u)` gibt die aktuelle Tangentsteifigkeitsmatrix basierend auf den aktuellen Verschiebungen u zurück.

Iteration im Newton-Raphson-Verfahren

Während des Newton-Raphson-Solvers wird die Jacobi-Matrix (Tangentsteifigkeitsmatrix) in jedem Iterationsschritt berechnet:

python

 Code kopieren

```
for iteration in range(max_iterations):  
    # Berechne Tangentsteifigkeitsmatrix K(u) und internen Kraftvektor F_int = K(u) * u  
    K_u = tangent_stiffness_matrix(u)  
    F_int = K_u @ u
```

Hier wird die Funktion `tangent_stiffness_matrix(u)` aufgerufen, und die Tangentsteifigkeitsmatrix $K(u)$ wird als K_u gespeichert. Anschließend wird die Matrix verwendet, um das **lineare Gleichungssystem** zu lösen und das Inkrement Δu zu berechnen, das u in die Richtung der Lösung aktualisiert.

Nicht-Lineare Gleichungssysteme

Iterative Methode

Riks

2. Riks-Verfahren

Das Riks-Verfahren, auch als **Bogenlängenverfahren** bekannt, ist eine Erweiterung des Newton-Raphson-Verfahrens zur Lösung stark nichtlinearer Probleme. Es eignet sich insbesondere für **stabilitätskritische Probleme**, bei denen das Gleichgewicht verloren gehen könnte, z. B. bei **Knicken** oder **Snap-through-Phänomenen**.

Prinzip und Formeln

Das Riks-Verfahren verwendet die **Bogenlänge** entlang der Last-Verformungs-Kurve als zusätzliches Steuerparameter, anstelle der Last direkt zu kontrollieren. Dieses Verfahren ermöglicht es, **gleichgewichtsunabhängige Lösungen** zu finden, die über instabile Punkte hinausgehen.

1. Die Gleichung wird um eine zusätzliche Bedingung ergänzt, die die Bogenlänge s entlang der Last-Verformungs-Kurve konstant hält:

$$\Delta u^{(k+1)} = \Delta u^{(k)} + \lambda^{(k)} \cdot \Delta F$$

2. Die Bogenlängenbedingung wird formuliert als:

$$(\Delta u)^T \cdot (\Delta u) + (\Delta \lambda)^2 = s^2$$

wobei:

- Δu die Änderung im Verschiebungsvektor,
 - $\Delta \lambda$ die Änderung im Lastfaktor,
 - s die Bogenlänge entlang des Lastpfades ist.
3. Iteriere, um Δu und $\Delta \lambda$ zu finden, bis das Gleichgewicht erreicht ist.

Vorteile und Nachteile

Vorteile	Nachteile
Ermöglicht die Lösung instabiler und kritischer Punkte	Erfordert komplexe Implementierung und viel Rechenleistung
Findet Lösungen bei Lasten, die über Instabilitäten hinausgehen	Schwierig, die Bogenlänge und andere Parameter richtig zu wählen
Geeignet für Systeme mit großen Verschiebungen	Langsame Konvergenz bei schlechten Startbedingungen

Nicht-Lineare Gleichungssysteme

Iterative Methode

Vergleich und Anwendung

- **Newton-Raphson** ist effizient und gut für Probleme geeignet, bei denen die Lösung stetig und stabil verläuft, also keine Instabilität auftritt.
- **Riks-Verfahren** wird für stabilitätskritische Systeme eingesetzt und ermöglicht es, die Last-Verformungs-Kurve auch über kritische Punkte hinaus zu verfolgen, was besonders bei Knick- und Biegeproblemen in der Strukturmechanik wichtig ist.

Beide Verfahren sind in der FEM weit verbreitet und dienen unterschiedlichen Zwecken, wobei die Wahl des Verfahrens stark vom zu lösenden Problem und den gewünschten Ergebnissen abhängt.

In der Finite-Elemente-Methode (FEM) und der numerischen Mathematik im Allgemeinen sind **numerische Integration** und **numerische Differentiation** von zentraler Bedeutung. In der FEM wird die Integration zur Berechnung von Steifigkeitsmatrizen, Massenmatrizen und Kraftvektoren verwendet, während die Differentiation oft für die Berechnung von Spannungen, Verzerrungen und zur Bestimmung der Jacobi- oder Tangentsteifigkeitsmatrix in nichtlinearen Problemen eingesetzt wird.

1. Numerische Integration

Numerische Integration wird verwendet, um Integrale über **Elementvolumina** oder **Oberflächen** zu berechnen, insbesondere wenn geschlossene Lösungen nicht möglich sind.

Wichtige Methoden der numerischen Integration

1. Trapezregel:

- Die Trapezregel approximiert das Integral durch das Summieren der Flächen unter kleinen Trapezen, die zwischen den Funktionswerten und der x-Achse liegen.
- Einfach zu implementieren, aber für glatte Funktionen weniger genau.
- Formel (für ein Intervall $[a, b]$ mit n Teilintervallen):

$$\int_a^b f(x) dx \approx \frac{h}{2} \left(f(x_0) + 2 \sum_{i=1}^{n-1} f(x_i) + f(x_n) \right)$$

- Wird in FEM nur selten verwendet, da es für glatte Integranden weniger effizient ist.

2. Simpson-Regel:

- Die Simpson-Regel verwendet Parabeln zur Approximation der Funktionskurve und ist genauer als die Trapezregel, insbesondere für glatte Funktionen.
- Formel:

$$\int_a^b f(x) dx \approx \frac{h}{3} \left(f(x_0) + 4 \sum_{i=1, \text{odd}}^{n-1} f(x_i) + 2 \sum_{i=2, \text{even}}^{n-2} f(x_i) + f(x_n) \right)$$

- Bietet höhere Genauigkeit, aber wird in FEM weniger eingesetzt als die Gauß-Quadratur.

3. Gauß-Quadratur:

- Die Gauß-Quadraturmethode ist die gängigste Integrationsmethode in der FEM und verwendet spezielle Abszissen und Gewichte, um das Integral eines Polynoms zu approximieren. Sie ist besonders effizient für Integranden, die sich durch Polynome darstellen lassen.
- Typische Anwendung bei Elementintegrationen, insbesondere für unregelmäßige Geometrien.
- Formel (für n Abszissen x_i und Gewichte w_i):

$$\int_{-1}^1 f(x) dx \approx \sum_{i=1}^n w_i f(x_i)$$

- Für FEM in 1D, 2D und 3D (Tensorprodukt-Quadratur) gut geeignet. In 2D und 3D oft mit **Gauß-Legendre-Quadratur** oder **Gauß-Lobatto-Quadratur** kombiniert.

4. Monte-Carlo-Integration:

- Stochastische Methode, die zufällige Punkte zur Approximation des Integrals verwendet. Gut geeignet für Integrale in hohen Dimensionen oder bei komplizierten Integrationsgrenzen.
- Wird in der FEM selten angewandt, außer bei speziellen Problemen, bei denen traditionelle Methoden unpraktisch sind.

2. Numerische Differentiation

Numerische Differentiation wird verwendet, um Ableitungen von Funktionen oder diskreten Daten zu berechnen. In der FEM hilft die Differentiation beispielsweise, Spannungen aus Verschiebungen abzuleiten oder die Tangentsteifigkeitsmatrix für nichtlineare Probleme zu berechnen.

Wichtige Methoden der numerischen Differentiation

1. Vorwärtsdifferenzen:

- Approximiert die Ableitung an einem Punkt x durch die Differenz zu einem Punkt rechts davon.
- Formel:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

- Gut geeignet für einfache Probleme, aber weniger genau als zentrische Differenzen.

2. Rückwärtsdifferenzen:

- Approximiert die Ableitung an einem Punkt x durch die Differenz zu einem Punkt links davon.
- Formel:

$$f'(x) \approx \frac{f(x) - f(x-h)}{h}$$

- Ebenfalls einfach zu implementieren, aber weniger genau für glatte Funktionen.

3. Zentrale Differenzen:

- Bietet die genaueste numerische Ableitungsmethode unter den Differenzenmethoden, indem sie einen symmetrischen Unterschied zwischen links und rechts des Punktes verwendet.
- Formel:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

- Diese Methode ist stabiler und genauer, wird daher oft in FEM verwendet, etwa für die Berechnung von Spannungen.

4. Finite-Differenzen-Methode:

- Diese Methode ist nützlich für die numerische Ableitung höherer Ordnung und wird häufig zur Approximation der Ableitungen für das Lösen von Differentialgleichungen verwendet.
- Für FEM spezifisch wird sie oft zur Approximation von Spannungen und Verzerrungen bei der Berechnung der Steifigkeitsmatrix eingesetzt.

5. Automatische Differentiation (AD):

- AD ist eine exakte Differenzierungsmethode, die oft in der numerischen Optimierung und FEM für komplexe nichtlineare Probleme genutzt wird.
- Vorteile sind höhere Genauigkeit und der automatische Aufbau der Tangentsteifigkeitsmatrix, besonders bei komplexen und nichtlinearen Modellen.

Zusammenfassung: Anwendung in der FEM

- **Numerische Integration:** Die Gauß-Quadratur ist die Standardmethode in der FEM, da sie effizient und genau für Integrale über die Elemente ist.
- **Numerische Differentiation:** Zentrale Differenzen und automatische Differentiation sind wichtig für die Berechnung von Spannungen, Verzerrungen und für die Tangentsteifigkeitsmatrix in nichtlinearen Problemen.

Die Wahl der Methoden hängt vom Problem und den Anforderungen an Genauigkeit und Rechenaufwand ab. In der FEM sind Gauß-Quadratur für die Integration und zentrale Differenzen sowie AD für die Differentiation am weitesten verbreitet.

Numerische Integration

Trapez-Regel

Die allgemeine Formel für die Trapezregel zur numerischen Integration eines Intervalls $[a, b]$ unterteilt das Intervall in n gleich große Teilintervalle und approximiert das Integral durch die Summe der Flächen von Trapezen zwischen den Funktionswerten. Die allgemeine Formel lautet:

$$\int_a^b f(x) dx \approx \frac{h}{2} \left(f(x_0) + 2 \sum_{i=1}^{n-1} f(x_i) + f(x_n) \right)$$

wobei:

- $h = \frac{b-a}{n}$ die Breite eines jeden Intervalls ist,
- $f(x_i)$ die Funktionswerte an den Punkten x_i sind.

python

 Code kopieren

```
import numpy as np

# Beispielhafte Spannungs-Dehnungs-Daten (z. B. aus einem Zugversuch)
epsilon = np.array([0.0, 0.01, 0.02, 0.03, 0.04, 0.05]) # Dehnung
sigma = np.array([0, 150, 280, 400, 450, 480])          # Spannung in MPa

# Schrittweite h zwischen den Datenpunkten (angenommen konstant, für allgemeine Formel gee
h = (epsilon[-1] - epsilon[0]) / (len(epsilon) - 1)

# Berechnung der Formänderungsenergie nach der allgemeinen Formel der Trapezregel
U = (h / 2) * (sigma[0] + 2 * np.sum(sigma[1:-1]) + sigma[-1])

# Ausgabe der berechneten Formänderungsenergie
print(f"Formänderungsenergie U: {U} MPa")
```

Erklärung des Codes:

1. Schrittweite h :

- Die Schrittweite h wird berechnet als der Abstand zwischen den Start- und Endwerten der Dehnung, geteilt durch die Anzahl der Intervalle.

2. Trapezregel:

- Die Formänderungsenergie U wird durch die allgemeine Formel der Trapezregel berechnet:

$$U = \frac{h}{2} \left(\sigma_0 + 2 \sum_{i=1}^{n-1} \sigma_i + \sigma_n \right)$$

- Hier wird der erste und letzte Spannungswert einmal und alle inneren Werte doppelt berücksichtigt.

Numerische Integration

Trapez-Regel

```
import numpy as np
```

```
# Beispielhafte Spannungs-Dehnungs-Daten (z. B. aus  
einem Zugversuch)
```

```
epsilon = np.array([0.0, 0.01, 0.02, 0.03, 0.04, 0.05]) #  
Dehnung
```

```
sigma = np.array([0, 150, 280, 400, 450, 480]) #  
Spannung in MPa
```

```
# Schrittweite h zwischen den Datenpunkten (angenommen  
konstant, für allgemeine Formel geeignet)
```

```
h = (epsilon[-1] - epsilon[0]) / (len(epsilon) - 1)
```

```
# Berechnung der Formänderungsenergie nach der  
allgemeinen Formel der Trapezregel
```

```
U = (h / 2) * (sigma[0] + 2 * np.sum(sigma[1:-1]) + sigma[-1])
```

```
# Ausgabe der berechneten Formänderungsenergie  
print(f"Formänderungsenergie U: {U} MPa")
```

Numerische Integration

Gauß-Quadratur

Die **Gauß-Quadratur** ist eine Methode zur numerischen Integration, die verwendet wird, um ein Integral durch eine gewichtete Summe der Funktionswerte an bestimmten Punkten zu approximieren. Diese Methode ist besonders effizient und genau für die Integration von Polynomen und wird daher häufig in der Finite-Elemente-Methode (FEM) verwendet.

Grundidee der Gauß-Quadratur

Das Ziel der Gauß-Quadratur ist es, das Integral einer Funktion $f(x)$ über ein Intervall $[a, b]$ zu berechnen:

$$\int_a^b f(x) dx$$

Die Gauß-Quadratur approximiert dieses Integral als gewichtete Summe der Funktionswerte an speziellen Punkten (Abszissen) innerhalb des Intervalls:

$$\int_a^b f(x) dx \approx \sum_{i=1}^n w_i f(x_i)$$

wobei:

- x_i die **Integrationspunkte** (Abszissen) sind, an denen die Funktion ausgewertet wird,
- w_i die **Gewichte** sind, die jedem Funktionswert zugeordnet werden,
- n die Anzahl der Punkte der Gauß-Quadratur ist.

Die Wahl von x_i und w_i erfolgt so, dass die Integration möglichst genau ist.

Wichtige Eigenschaften der Gauß-Quadratur

1. Optimale Punkte und Gewichte:

- Die Punkte x_i und Gewichte w_i werden so gewählt, dass die Methode für Polynome bis zu einem bestimmten Grad genau ist.
- Für ein n -Punkt-Gauß-Quadraturverfahren ist das Integral exakt für alle Polynome bis zum Grad $2n - 1$.

2. Orthogonale Polynome:

- Die Abszissenpunkte x_i sind die Nullstellen eines bestimmten orthogonalen Polynoms. Für die Standard-Gauß-Quadratur, die auf dem Intervall $[-1, 1]$ arbeitet, sind die Punkte die Nullstellen der **Legendre-Polynome**.
- Die Wahl der Polynome hängt vom Integrationsbereich und der Gewichtsfunktion ab (z. B. Legendre-Polynome für das Intervall $[-1, 1]$, Tschebyscheff-Polynome für andere Intervalle).

3. Transformation des Intervalls:

- Die Gauß-Quadratur wird standardmäßig für das Intervall $[-1, 1]$ berechnet. Falls das Integrationsintervall ein anderes ist, wird das Intervall durch eine **lineare Transformation** angepasst.

Numerische Integration

Gauß-Quadratur

Gauß-Legendre-Quadratur

Die **Gauß-Legendre-Quadratur** ist eine häufig verwendete Form der Gauß-Quadratur, die auf dem Intervall $[-1, 1]$ mit Gewichten und Abszissen basiert, die auf den Nullstellen der Legendre-Polynome beruhen.

Beispiel für die Gauß-Legendre-Quadratur mit 2 Punkten

Für eine 2-Punkt-Gauß-Quadratur (d. h. $n = 2$) auf dem Intervall $[-1, 1]$ lauten die Punkte und Gewichte:

- $x_1 = -\frac{1}{\sqrt{3}}, x_2 = \frac{1}{\sqrt{3}}$
- $w_1 = 1, w_2 = 1$

Damit kann das Integral wie folgt approximiert werden:

$$\int_{-1}^1 f(x) dx \approx w_1 f(x_1) + w_2 f(x_2) = f\left(-\frac{1}{\sqrt{3}}\right) + f\left(\frac{1}{\sqrt{3}}\right)$$

Wenn das Integrationsintervall nicht $[-1, 1]$ ist, beispielsweise $[a, b]$, wird eine Transformation verwendet:

1. Die Punkte werden transformiert:

$$x_i = \frac{b-a}{2} \cdot \xi_i + \frac{a+b}{2}$$

wobei ξ_i die Abszissenpunkte auf dem Intervall $[-1, 1]$ sind.

2. Die Gewichte werden entsprechend angepasst:

$$w_i = \frac{b-a}{2} \cdot \text{Gewicht bei } \xi_i$$

Vorteile der Gauß-Quadratur

- **Hohe Genauigkeit mit wenigen Punkten:** Mit nur n Punkten kann die Gauß-Quadratur Polynome bis zum Grad $2n - 1$ genau integrieren. Daher ist sie sehr effizient.
- **Anwendung in FEM:** In der Finite-Elemente-Methode ist die Gauß-Quadratur ideal, um Integrale über Elementvolumina zu berechnen, insbesondere bei komplexen Formfunktionen.

Numerische Integration

Gauß-Quadratur

Ja, das ist korrekt! Die Gauß-Quadratur nutzt eine Transformation, um ein Integral über ein beliebiges Intervall $[a, b]$ auf das Standardintervall $[-1, 1]$ abzubilden. Dadurch können immer dieselben Gauß-Punkte und Gewichte für die Berechnung der Integralnäherung verwendet werden, unabhängig von den ursprünglichen Intervallgrenzen.

Erklärung des Codes

1. Gauß-Quadraturpunkte und Gewichte:

- Die Abszissen und Gewichte für die 2-Punkt-Gauß-Quadratur auf dem Intervall $[-1, 1]$ sind:
 - Punkte: $\pm \frac{1}{\sqrt{3}}$
 - Gewichte: beide gleich 1

2. Transformation der Punkte auf das Intervall $[a, b]$:

- Die Punkte ξ_i auf $[-1, 1]$ werden für jedes Intervall $[a, b]$ in der Dehnungsachse auf das Intervall $[a, b]$ abgebildet:

$$x_i = 0.5 \cdot (b - a) \cdot \xi_i + 0.5 \cdot (a + b)$$

- Diese Transformation ermöglicht die Anwendung der Gauß-Quadratur auf beliebige Intervalle in der Dehnungsachse.

3. Berechnung der Funktionswerte an den transformierten Punkten:

- Die Spannung σ wird an den transformierten Punkten mithilfe der **Interpolation** berechnet (`np.interp(mapped_points, epsilon, sigma)`), um die Spannungswerte an den Abszissenpunkten der Gauß-Quadratur zu bestimmen.

4. Gauß-Quadratur für jedes Intervall:

- Die Formänderungsenergie U wird durch Summation der gewichteten Spannungswerte über jedes Intervall berechnet:

$$U = \sum 0.5 \cdot (b - a) \cdot w_i \cdot f(x_i)$$

- Dabei ist $0.5 \cdot (b - a)$ der Skalenfaktor für die Intervallbreite.

python

```
import numpy as np
```

Details immer anzeigen  Code kopieren

```
# Beispielhafte Spannungs-Dehnungs-Daten (z. B. aus einem Zugversuch)
```

```
epsilon = np.array([0.0, 0.01, 0.02, 0.03, 0.04, 0.05]) # Dehnung
```

```
sigma = np.array([0, 150, 280, 400, 450, 480]) # Spannung in MPa
```

```
# Ziel: Berechnung der Formänderungsenergie (Fläche unter der Spannungs-Dehnungs-Kurve) mit
```

```
# Gauß-Quadraturpunkte und Gewichte für 2-Punkt-Gauß-Quadratur auf [-1, 1]
```

```
gauss_points = np.array([-1/np.sqrt(3), 1/np.sqrt(3)]) # Abszissen
```

```
weights = np.array([1, 1]) # Gewichte
```

```
# Berechnung der Formänderungsenergie
```

```
U_gauss = 0 # Initialisierung der Formänderungsenergie
```

```
# Iteration über jedes Intervall in der Spannungs-Dehnungs-Kurve
```

```
for i in range(len(epsilon) - 1):
```

```
    # Intervallgrenzen
```

```
    a = epsilon[i]
```

```
    b = epsilon[i + 1]
```

```
    h = b - a # Intervallbreite
```

```
# Transformation der Gauß-Punkte auf das Intervall [a, b]
```

```
mapped_points = 0.5 * (b - a) * gauss_points + 0.5 * (a + b)
```

```
# Berechnung der Spannungswerte an den transformierten Gauß-Punkten
```

```
sigma_mapped = np.interp(mapped_points, epsilon, sigma) # Interpolation der Spannungs
```

```
# Gauß-Quadratur über das Intervall
```

```
U_gauss += np.sum(0.5 * (b - a) * weights * sigma_mapped)
```

```
# Ausgabe der berechneten Formänderungsenergie
```

```
U_gauss
```

Ergebnis

15.200000000000001

Numerische Integration

Gauß-Quadratur

```
import numpy as np
import matplotlib.pyplot as plt

# Beispielhafte Spannungs-Dehnungs-Daten (z. B. aus einem Zugversuch)
epsilon = np.array([0.0, 0.01, 0.02, 0.03, 0.04, 0.05]) # Dehnung
sigma = np.array([0, 150, 280, 400, 450, 480]) # Spannung in MPa

# Ziel: Berechnung der Formänderungsenergie (Fläche unter der Spannungs-Dehnungs-Kurve) mit Gauß-Quadratur

# Gauß-Quadraturpunkte und Gewichte für 2-Punkt-Gauß-Quadratur auf [-1, 1]
gauss_points = np.array([-1/np.sqrt(3), 1/np.sqrt(3)]) # Abszissen
weights = np.array([1, 1]) # Gewichte

# Berechnung der Formänderungsenergie
U_gauss = 0 # Initialisierung der Formänderungsenergie

# Iteration über jedes Intervall in der Spannungs-Dehnungs-Kurve
for i in range(len(epsilon) - 1):
    # Intervallgrenzen
    a = epsilon[i]
    b = epsilon[i + 1]
    h = b - a # Intervallbreite

    # Transformation der Gauß-Punkte auf das Intervall [a, b]
    mapped_points = 0.5 * (b - a) * gauss_points + 0.5 * (a + b)

    # Berechnung der Spannungswerte an den transformierten Gauß-Punkten
    sigma_mapped = np.interp(mapped_points, epsilon, sigma) # Interpolation der Spannungswerte

    # Gauß-Quadratur über das Intervall
    U_gauss += np.sum(0.5 * (b - a) * weights * sigma_mapped)

# Ausgabe der berechneten Formänderungsenergie
U_gauss

# Ausgabe der berechneten Formänderungsenergie
print(f"Formänderungsenergie U: {U_gauss} MPa")

# Plotten des Spannungs-Dehnungs-Diagramms
plt.figure(figsize=(8, 6))
plt.plot(epsilon, sigma, marker='o', linestyle='-', color='b', label='Spannungs-Dehnungs-Kurve')
plt.fill_between(epsilon, sigma, color="lightblue", alpha=0.3, label="Formänderungsenergie")

# Diagrammanpassungen
plt.title("Spannungs-Dehnungs-Diagramm")
plt.xlabel("Dehnung  $\epsilon$ ")
plt.ylabel("Spannung  $\sigma$  (MPa)")
plt.legend()
plt.grid(True)
plt.show()
```

Numerische Differentiation

Zentrale Differenzenmethode

In diesem Beispiel ist h die Schrittweite zwischen den Dehnungswerten ϵ . Die Daten sind gleichmäßig verteilt, daher ist h konstant und entspricht dem Abstand zwischen zwei benachbarten Dehnungswerten:

$$h = \epsilon[i + 1] - \epsilon[i] = 0.01$$

Dieser Wert wird in den zentralen Differenzen für alle inneren Punkte verwendet, um die Ableitung zu berechnen. Der Python-Code berechnet allerdings h für jedes Intervall individuell durch $\epsilon[i + 1] - \epsilon[i - 1]$, sodass er auch dann korrekt funktioniert, wenn die Abstände ungleichmäßig sind.

Die zentrale Differenzenmethode ist eine numerische Technik zur Berechnung der Ableitung einer Funktion an einem gegebenen Punkt. Im Kontext der Spannungs-Dehnungs-Kurve verwenden wir sie, um die Steifigkeit $\frac{d\sigma}{d\epsilon}$ zu berechnen, die der Ableitung der Spannung σ nach der Dehnung ϵ entspricht.

1. Allgemeine Formel der zentralen Differenzenmethode

Die zentrale Differenzenmethode approximiert die Ableitung einer Funktion $f(x)$ an einem Punkt x mithilfe der Funktionswerte an den Punkten $x + h$ und $x - h$. Die allgemeine Formel lautet:

$$f'(x) \approx \frac{f(x + h) - f(x - h)}{2h}$$

wobei:

- $f(x + h)$ und $f(x - h)$ die Funktionswerte an den Punkten links und rechts von x sind,
- h die Schrittweite zwischen diesen Punkten ist.

Diese Methode liefert eine gute Annäherung an die Ableitung, wenn h klein ist, und ist genauer als die Vorwärts- oder Rückwärtsdifferenzenmethoden, da sie beide Seiten eines Punktes zur Ableitungsberechnung einbezieht.

2. Anwendung auf die Spannungs-Dehnungs-Kurve

In der Spannungs-Dehnungs-Kurve haben wir diskrete Punkte (ϵ_i, σ_i) , wobei:

- ϵ_i die Dehnung an Punkt i ,
- σ_i die Spannung an Punkt i ist.

Die zentrale Differenzenmethode wird auf die Daten angewendet, um die Steifigkeit $\frac{d\sigma}{d\epsilon}$ für jeden inneren Punkt i zu berechnen. Die Formel für die zentrale Differenzenmethode an Punkt i lautet:

$$\frac{d\sigma}{d\epsilon} \approx \frac{\sigma_{i+1} - \sigma_{i-1}}{\epsilon_{i+1} - \epsilon_{i-1}}$$

Hier ist:

- σ_{i+1} die Spannung am Punkt rechts von i ,
- σ_{i-1} die Spannung am Punkt links von i ,
- ϵ_{i+1} und ϵ_{i-1} die zugehörigen Dehnungen.

Numerische Differentiation

Zentrale Differenzenmethode

python

Code kopieren

```
import numpy as np

# Beispielhafte Spannungs-Dehnungs-Daten (z. B. aus einem Zugversuch)
epsilon = np.array([0.0, 0.01, 0.02, 0.03, 0.04, 0.05]) # Dehnung
sigma = np.array([0, 150, 280, 400, 450, 480]) # Spannung in MPa

# Berechnung der Ableitung der Spannungs-Dehnungs-Kurve mit zentralen Differenzen
d_sigma_d_epsilon = np.zeros(len(sigma)) # Initialisierung des Ableitungsvektors

# Zentrale Differenzen für alle inneren Punkte
for i in range(1, len(epsilon) - 1):
    d_sigma_d_epsilon[i] = (sigma[i + 1] - sigma[i - 1]) / (epsilon[i + 1] - epsilon[i - 1])

# Vorwärtsdifferenz für den ersten Punkt
d_sigma_d_epsilon[0] = (sigma[1] - sigma[0]) / (epsilon[1] - epsilon[0])

# Rückwärtsdifferenz für den letzten Punkt
d_sigma_d_epsilon[-1] = (sigma[-1] - sigma[-2]) / (epsilon[-1] - epsilon[-2])

# Ausgabe der berechneten Ableitung
print("Ableitung der Spannungs-Dehnungs-Kurve:", d_sigma_d_epsilon)
```

Erklärung:

1. Schrittweite h :

- Der Wert $h = 0.01$ wird für alle Berechnungen verwendet, da die Dehnungswerte gleichmäßig verteilt sind.

2. Zentrale Differenzen:

- Für alle inneren Punkte wird die zentrale Differenzenformel verwendet:

$$\frac{d\sigma}{d\epsilon} \approx \frac{\sigma_{i+1} - \sigma_{i-1}}{2h}$$

3. Vorwärts- und Rückwärtsdifferenzen:

- Die Vorwärts- und Rückwärtsdifferenzen werden für die Ränder berechnet:
 - Erster Punkt (Vorwärtsdifferenz): $\frac{\sigma_1 - \sigma_0}{h}$
 - Letzter Punkt (Rückwärtsdifferenz): $\frac{\sigma_n - \sigma_{n-1}}{h}$

Erklärung:

1. Zentrale Differenzen:

- Für alle inneren Punkte wird die Ableitung mit der zentralen Differenzenmethode berechnet:

$$\frac{d\sigma}{d\epsilon} \approx \frac{\sigma_{i+1} - \sigma_{i-1}}{\epsilon_{i+1} - \epsilon_{i-1}}$$

2. Vorwärts- und Rückwärtsdifferenzen:

- Am Anfang und Ende der Daten werden die Vorwärts- bzw. Rückwärtsdifferenzen verwendet, um die Ableitung zu berechnen, da die zentralen Differenzen dort nicht anwendbar sind.

Numerische Differentiation

Zentrale Differenzenmethode

```
import numpy as np
import matplotlib.pyplot as plt

# Beispielhafte Spannungs-Dehnungs-Daten (z. B. aus einem Zugversuch)
epsilon = np.array([0.0, 0.01, 0.02, 0.03, 0.04, 0.05]) # Dehnung
sigma = np.array([0, 150, 280, 400, 450, 480]) # Spannung in MPa

# Berechnung der Ableitung der Spannungs-Dehnungs-Kurve mit zentralen Differenzen
d_sigma_d_epsilon = np.zeros(len(sigma)) # Initialisierung des Ableitungsvektors

# Zentrale Differenzen für alle inneren Punkte
for i in range(1, len(epsilon) - 1):
    d_sigma_d_epsilon[i] = (sigma[i + 1] - sigma[i - 1]) / (epsilon[i + 1] - epsilon[i - 1])

# Vorwärtsdifferenz für den ersten Punkt
d_sigma_d_epsilon[0] = (sigma[1] - sigma[0]) / (epsilon[1] - epsilon[0])

# Rückwärtsdifferenz für den letzten Punkt
d_sigma_d_epsilon[-1] = (sigma[-1] - sigma[-2]) / (epsilon[-1] - epsilon[-2])

# Erstellen der Diagramme
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))

# Plot 1: Spannungs-Dehnungs-Diagramm
ax1.plot(epsilon, sigma, marker='o', linestyle='-', color='b', label='Spannungs-Dehnungs-Kurve')
ax1.fill_between(epsilon, sigma, color="lightblue", alpha=0.3, label="Formänderungsenergie")
ax1.set_title("Spannungs-Dehnungs-Diagramm")
ax1.set_xlabel("Dehnung  $\epsilon$ ")
ax1.set_ylabel("Spannung  $\sigma$  (MPa)")
ax1.legend()
ax1.grid(True)

# Plot 2: Ableitung der Spannungs-Dehnungs-Kurve
ax2.plot(epsilon, d_sigma_d_epsilon, marker='o', linestyle='-', color='r', label="d $\sigma$ /d $\epsilon$ ")
ax2.set_title("Ableitung der Spannungs-Dehnungs-Kurve")
ax2.set_xlabel("Dehnung  $\epsilon$ ")
ax2.set_ylabel("Steifigkeit d $\sigma$ /d $\epsilon$  (MPa)")
ax2.legend()
ax2.grid(True)

# Anzeigen der Diagramme
plt.tight_layout()
plt.show()
```

Numerische Integration

1. Numerische Integration in der FEM

Numerische Integration wird in der FEM eingesetzt, um die Elementsteifigkeits- und Massenmatrizen sowie Kraftvektoren zu berechnen. Da die Formfunktionen oft komplex sind und über den Elementbereich integriert werden müssen, sind analytische Lösungen in der Regel nicht möglich. Deshalb werden **Gauß-Quadratur** oder andere numerische Integrationsmethoden verwendet.

Beispiel: Steifigkeitsmatrix eines Elements

Die **Steifigkeitsmatrix** K_e für ein Element e ergibt sich aus der Variation der Gesamtenergie des Systems. Für ein elastisches Element lautet die Steifigkeitsmatrix:

$$K_e = \int_{\Omega_e} B^T D B d\Omega$$

wobei:

- B die Ableitungsmatrix der Formfunktionen ist, die die Dehnungen aus den Verschiebungen berechnet,
- D die Materialmatrix (z. B. Elastizitätsmatrix) ist, die die Materialeigenschaften enthält,
- Ω_e das Volumen des Elements ist.

Da B und D meist von der Position im Element abhängen, wird dieses Integral numerisch berechnet, oft mit der **Gauß-Quadratur**:

$$K_e \approx \sum_{i=1}^n w_i B^T(x_i) D B(x_i)$$

Hier:

- x_i sind die Integrationspunkte,
- w_i die entsprechenden Gewichte.

Beispiel: Kraftvektor eines Elements

Der **Kraftvektor** F_e eines Elements wird durch die Integration der Formfunktion N über die Volumen- oder Oberflächenlast f gegeben:

$$F_e = \int_{\Omega_e} N^T f d\Omega$$

Auch dies wird numerisch integriert:

$$F_e \approx \sum_{i=1}^n w_i N^T(x_i) f(x_i)$$

2. Numerische Differentiation in der FEM

Numerische Differentiation wird in der FEM zur Berechnung von Dehnungen, Spannungen und der Jacobi-Matrix in nichtlinearen Problemen eingesetzt.

Beispiel: Dehnung- und Spannungsberechnung

Die **Dehnung** ε in einem Element wird aus den Verschiebungen u mithilfe der **Formfunktionsableitungen** B berechnet:

$$\varepsilon = B \cdot u$$

Hierbei enthält B die Ableitungen der Formfunktionen N nach den räumlichen Koordinaten x, y, z .

Zum Beispiel für ein zweidimensionales Element:

$$B = \begin{pmatrix} \frac{\partial N_1}{\partial x} & \frac{\partial N_2}{\partial x} & \cdots \\ \frac{\partial N_1}{\partial y} & \frac{\partial N_2}{\partial y} & \cdots \end{pmatrix}$$

Die Ableitungen der Formfunktionen werden oft über eine numerische Differentiation am Integrationspunkt berechnet.

3. Jacobi-Matrix in der FEM (Tangentsteifigkeitsmatrix für nichtlineare Probleme)

In nichtlinearen Problemen ist die **Jacobi-Matrix** die Ableitung des Restkraftvektors $R(u) = F_{\text{ext}} - F_{\text{int}}(u)$ nach den Verschiebungen u :

$$J(u) = \frac{\partial R}{\partial u}$$

Für die Jacobi-Matrix wird oft die **Tangentsteifigkeitsmatrix** K_{tan} verwendet, die die Änderung der internen Kräfte $F_{\text{int}}(u)$ in Bezug auf die Verschiebungen u darstellt. Die Tangentsteifigkeitsmatrix lautet:

$$K_{\text{tan}} = \int_{\Omega_e} B^T \frac{\partial \sigma}{\partial \varepsilon} B d\Omega$$

wobei:

- $\frac{\partial \sigma}{\partial \varepsilon}$ die **materialabhängige Steifigkeitsmatrix** ist,
- B die **Ableitungsmatrix** der Formfunktionen.

Da sich diese Matrix über das Elementvolumen erstreckt, wird die numerische Integration (z. B. Gauß-Quadratur) eingesetzt, um die Tangentsteifigkeitsmatrix zu berechnen.

Dynamische Gleichungssysteme

In der Finite-Elemente-Methode (FEM) werden dynamische Gleichungssysteme verwendet, um zeitabhängige Probleme wie Schwingungen, Stoßbelastungen und wellenausbreitende Phänomene zu analysieren. Das dynamische Gleichungssystem einer Struktur lautet in der allgemeinen Form:

$$M\ddot{u}(t) + C\dot{u}(t) + Ku(t) = F(t)$$

wobei:

- M die Massenmatrix ist,
- C die Dämpfungsmatrix,
- K die Steifigkeitsmatrix,
- $u(t)$ der Verschiebungsvektor,
- $F(t)$ der zeitabhängige Lastvektor,
- $\dot{u}(t)$ und $\ddot{u}(t)$ die Geschwindigkeit bzw. Beschleunigung von $u(t)$ sind.

Dynamische Gleichungssysteme

Direkte Methode

Zentrale Differenzenmethode

1. Explizite Zeitschrittverfahren

Explizite Verfahren eignen sich gut für **hochdynamische Probleme** wie Stoßbelastungen und nichtlineare Probleme, weil sie keine Invertierung der Systemmatrix benötigen. Sie haben jedoch eine Einschränkung hinsichtlich der maximalen Zeitschrittgröße.

- **Zentrale Differenzenmethode (Central Difference Method):**

- Ein explizites Verfahren, bei dem der Beschleunigungswert $\ddot{u}(t)$ aus dem Kräfteungleichgewicht berechnet wird. Die Verschiebung und Geschwindigkeit werden dann in expliziten Zeitschritten aktualisiert.

- Aktualisierungsschritte:

$$\ddot{u}(t) = M^{-1} (F(t) - C\dot{u}(t) - Ku(t))$$

$$\dot{u}(t + \Delta t/2) = \dot{u}(t - \Delta t/2) + \ddot{u}(t)\Delta t$$

$$u(t + \Delta t) = u(t) + \dot{u}(t + \Delta t/2)\Delta t$$

- **Vor- und Nachteile:**

- **Vorteile:** Einfach zu implementieren, benötigt keine Invertierung von M , daher effizient für große Probleme.
- **Nachteile:** Stabilität ist vom Zeitschritt abhängig; das Verfahren ist nur stabil, wenn Δt klein genug ist (CFL-Kriterium).

Dynamische Gleichungssysteme

Iterative Methode

Newmark-beta

2. Implizite Zeitschrittverfahren

Implizite Verfahren sind stabiler und erlauben größere Zeitschritte, eignen sich jedoch eher für Probleme mit mäßiger Dynamik, da sie eine Invertierung der Systemmatrix erfordern.

- **Newmark-Verfahren:**

- Ein weit verbreitetes implizites Verfahren in der Strukturmechanik, das sich durch seine allgemeine Stabilität auszeichnet und sowohl zur Schwingungsanalyse als auch für langsam variierende dynamische Belastungen genutzt wird.
- Die Methode basiert auf den Parametern β und γ , die das Verfahren steuern und für bedingte (oder unbedingte) Stabilität angepasst werden können.

- Die Gleichungen lauten:

$$u(t + \Delta t) = u(t) + \Delta t \dot{u}(t) + \frac{\Delta t^2}{2} ((1 - 2\beta)\ddot{u}(t) + 2\beta\ddot{u}(t + \Delta t))$$

$$\dot{u}(t + \Delta t) = \dot{u}(t) + \Delta t ((1 - \gamma)\ddot{u}(t) + \gamma\ddot{u}(t + \Delta t))$$

- Typische Werte für unbedingte Stabilität: $\beta = 0.25$ und $\gamma = 0.5$.

- **Vor- und Nachteile:**

- **Vorteile:** Größere Zeitschritte möglich, stabil auch bei steifen Systemen, eignet sich für Schwingungsanalysen und quasistatische Lasten.
- **Nachteile:** Matrixinversion ist rechenintensiv, insbesondere bei großen Systemen.

3. Modal-Analyse (Modale Superposition)

Die Modal-Analyse ist eine Methode, die besonders effektiv für Systeme mit geringer Dämpfung ist. Das System wird in **Eigenmoden** zerlegt, die unabhängig voneinander analysiert werden können. Diese Methode ist jedoch nur für lineare Systeme geeignet.

- **Vorgehen:**

- Berechne die Eigenwerte und Eigenvektoren des Systems.
- Stelle das Verschiebungsfeld als Summe der Moden dar:

$$u(t) = \sum_{i=1}^n q_i(t) \phi_i$$

wobei ϕ_i die Eigenvektoren und $q_i(t)$ die modalen Koordinaten sind.

- Die Differentialgleichungen für $q_i(t)$ sind entkoppelt und können unabhängig gelöst werden.
- **Vor- und Nachteile:**
 - **Vorteile:** Effizient bei schwach gedämpften Systemen, reduziert die Anzahl der Gleichungen, ermöglicht eine schnelle Lösung.
 - **Nachteile:** Nur für lineare Systeme anwendbar; nicht geeignet für nichtlineare Dynamik.

Vergleich der Methoden

Methode	Typ	Anwendungen	Vorteile	Nachteile
Zentrale Differenzen	Explizit	Hochdynamische Probleme	Einfache Implementierung, keine Matrixinversion	Zeitschrittgröße begrenzt (CFL-Kriterium)
Newmark-Verfahren	Implizit	Mäßige Dynamik, Schwingungen	Stabilität auch bei großen Zeitschritten	Matrixinversion notwendig
Modal-Analyse	Modale Superposition	Schwach gedämpfte lineare Systeme	Reduzierte Rechenkosten bei schwach gedämpften Systemen	Nur für lineare, nicht für nichtlineare Systeme

Für die FEM-Dynamik ist die Wahl der Methode abhängig vom Problemtyp:

- Für schnelle, impulsartige oder stoßartige Belastungen (z. B. Crash) sind **explizite Verfahren** wie die zentrale Differenzenmethode vorzuziehen.
- Für langsamere, schwingende Systeme und Schwingungsanalysen sind **implizite Verfahren** wie das Newmark-Verfahren oder die **Modal-Analyse** geeignet.

1. Stabilitätskriterium für explizite Zeitschrittverfahren

In expliziten Zeitschrittverfahren (wie der zentralen Differenzenmethode) hängt die **Stabilität** stark von der Größe des Zeitschritts Δt ab. Damit ein explizites Verfahren stabil ist, muss das Zeitschrittintervall kleiner als ein bestimmtes **stabiles Zeitinkrement** sein. Dieses Kriterium wird oft als **Courant-Friedrichs-Lewy-Kriterium (CFL-Kriterium)** bezeichnet und lautet in der FEM:

$$\Delta t \leq \frac{L}{c}$$

wobei:

- L die charakteristische Elementlänge ist,
- c die Ausbreitungsgeschwindigkeit der Welle im Material (Schallgeschwindigkeit) ist, berechnet als $c = \sqrt{\frac{E}{\rho}}$ für ein ungedämpftes, isotropes Material.

Ein stabiles Zeitinkrement hängt also direkt von den Materialeigenschaften ab. Bei kleinen Elementen (also kleiner charakteristischer Länge L) wird auch der maximale Zeitschritt sehr klein, was zu einer hohen Anzahl an Zeitschritten und damit zu längeren Rechenzeiten führt.

2. Massen-Skalierung

Massen-Skalierung ist eine Technik, um das stabile Zeitinkrement Δt zu vergrößern, ohne die Elementgröße zu ändern. Dabei wird die **Masse** des Systems künstlich erhöht, wodurch die Wellen- oder Schallgeschwindigkeit c im Material gesenkt wird und sich dadurch ein größerer stabiler Zeitschritt ergibt.

Idee der Massen-Skalierung

Die Schallgeschwindigkeit c im Material hängt von der **Dichte** ρ des Materials ab:

$$c = \sqrt{\frac{E}{\rho}}$$

Durch Erhöhen der Dichte ρ wird die Schallgeschwindigkeit reduziert, was zu einem größeren stabilen Zeitinkrement führt:

$$\Delta t \leq \frac{L}{c} = \frac{L}{\sqrt{E/\rho}}$$

Diese zusätzliche Masse kann entweder **global** oder **lokal** zu bestimmten Elementen hinzugefügt werden.

Arten der Massen-Skalierung

- **Explizite Massen-Skalierung:** Die Masse wird direkt zu den Elementen hinzugefügt oder die Dichte ρ der Elemente wird erhöht. Dies ist besonders bei kleinen Elementen nützlich, die das Zeitinkrement stark beeinflussen.
- **Implizite Massen-Skalierung:** Der Zeitschritt wird durch Anpassung des Dämpfungs- und Trägheitsverhaltens der Simulation gesteuert. Hierbei wird die Masse so angepasst, dass die Energieverteilung während des Zeitschritts stabil bleibt.

Vorteile und Nachteile der Massen-Skalierung

Vorteile	Nachteile
Erhöht das stabile Zeitinkrement und reduziert die Rechenzeit	Kann zu unphysikalischen Ergebnissen führen, wenn die zusätzliche Masse zu groß ist
Ermöglicht die Simulation sehr dynamischer Prozesse	Kann das dynamische Verhalten des Systems verfälschen, insbesondere bei großen Skalierungen
Gut geeignet für hochdynamische Simulationen wie Crash	Muss vorsichtig angewandt werden, um Genauigkeit zu wahren

Dynamische Gleichungssysteme

Stabiles Zeitinkrement

Das **Courant-Friedrichs-Lewy-Kriterium** (CFL-Kriterium) ist ein Stabilitätskriterium, das in numerischen Berechnungen angewendet wird, insbesondere bei **expliziten Zeitschrittverfahren** für die Lösung von **partiellen Differentialgleichungen** (PDEs), wie sie in der Finite-Elemente-Methode (FEM) und Finite-Differenzen-Methode (FDM) verwendet werden. Es stellt sicher, dass sich die numerische Lösung im Zeit- und Raumgitter stabil verhält und physikalisch sinnvolle Ergebnisse liefert.

Grundidee des CFL-Kriteriums

Das CFL-Kriterium besagt, dass der Zeitschritt Δt klein genug sein muss, damit **Informationen** (z. B. Wellen oder Störungen) sich pro Zeitschritt höchstens über die Länge eines Gitterelements Δx ausbreiten können. Andernfalls ist die numerische Lösung instabil, was zu falschen Ergebnissen führt.

Für das CFL-Kriterium gilt allgemein:

$$\Delta t \leq \frac{\Delta x}{c}$$

wobei:

- Δx die **Gittergröße** oder charakteristische Länge des Elements ist,
- c die **Ausbreitungsgeschwindigkeit** der Welle im Material (oft die Schallgeschwindigkeit),
- Δt der **Zeitschritt**.

Anwendung in der FEM

In der **expliziten FEM** zur Lösung dynamischer Probleme wird das CFL-Kriterium verwendet, um die Stabilität der Berechnung sicherzustellen. Typische Anwendungen sind hochdynamische Probleme wie **Crash-Simulationen**, **Stoßbelastungen** oder **Schwingungen**. Das CFL-Kriterium verhindert, dass instabile Schwingungen in der Lösung auftreten, indem der Zeitschritt auf eine maximale Größe begrenzt wird.

Beispiel: Berechnung des stabilen Zeitschritts

In der Strukturmechanik ist die Ausbreitungsgeschwindigkeit c oft die **Schallgeschwindigkeit im Material** und wird berechnet als:

$$c = \sqrt{\frac{E}{\rho}}$$

wobei:

- E der **Elastizitätsmodul** des Materials ist,
- ρ die **Dichte** des Materials.

Das maximale stabile Zeitschrittintervall Δt_{\max} wird dann durch:

$$\Delta t_{\max} \approx \frac{L}{c} = \frac{L}{\sqrt{\frac{E}{\rho}}}$$

wobei L die charakteristische Länge des kleinsten Elements in der Struktur ist.

Beispiel für das CFL-Kriterium

Angenommen, wir haben ein Material mit folgenden Eigenschaften:

- Elastizitätsmodul $E = 210 \text{ GPa}$,
- Dichte $\rho = 7850 \text{ kg/m}^3$,
- charakteristische Elementlänge $L = 0.01 \text{ m}$.

1. Berechnung der Schallgeschwindigkeit c :

$$c = \sqrt{\frac{E}{\rho}} = \sqrt{\frac{210 \times 10^9 \text{ Pa}}{7850 \text{ kg/m}^3}} \approx 5188 \text{ m/s}$$

2. Berechnung des maximalen Zeitschritts Δt_{\max} nach dem CFL-Kriterium:

$$\Delta t_{\max} \approx \frac{L}{c} = \frac{0.01}{5188} \approx 1.93 \times 10^{-6} \text{ s} = 1.93 \mu\text{s}$$

In dieser Simulation muss also der Zeitschritt kleiner oder gleich $1.93 \mu\text{s}$ gewählt werden, um eine stabile und genaue Lösung zu erhalten.

Zusammenfassung

Das **CFL-Kriterium** stellt sicher, dass der Zeitschritt in expliziten Berechnungen so gewählt wird, dass sich Wellen oder Störungen nur innerhalb der räumlichen Gittergröße fortpflanzen können. Es sorgt für die Stabilität und Genauigkeit der Lösung und ist besonders wichtig für Simulationen mit hochdynamischen Prozessen.

Dynamische Gleichungssysteme

Iterative Methode

Newmark-beta

Dynamisches Gleichungssystem in der FEM

Die Bewegungsgleichung für ein dynamisches System lautet:

$$M\ddot{u}(t) + C\dot{u}(t) + Ku(t) = F(t)$$

wobei:

- M die Massenmatrix ist,
- C die Dämpfungsmatrix,
- K die Steifigkeitsmatrix,
- $u(t)$ der Verschiebungsvektor zur Zeit t ,
- $\dot{u}(t)$ die Geschwindigkeit und $\ddot{u}(t)$ die Beschleunigung,
- $F(t)$ der Lastvektor zur Zeit t .

Grundidee der Newmark- β -Methode

Die Newmark- β -Methode verwendet zwei Parameter, β und γ , um die Integration zu steuern. Die Methode basiert auf einer diskreten Näherung der Geschwindigkeit und Beschleunigung in Abhängigkeit von den Verschiebungen:

1. Verschiebung u_{n+1} :

$$u_{n+1} = u_n + \Delta t \dot{u}_n + \frac{\Delta t^2}{2} ((1 - 2\beta)\ddot{u}_n + 2\beta\ddot{u}_{n+1})$$

2. Geschwindigkeit \dot{u}_{n+1} :

$$\dot{u}_{n+1} = \dot{u}_n + \Delta t ((1 - \gamma)\ddot{u}_n + \gamma\ddot{u}_{n+1})$$

wobei:

- Δt das Zeitschrittintervall ist,
- $u_n, \dot{u}_n, \ddot{u}_n$ die Verschiebung, Geschwindigkeit und Beschleunigung zum Zeitpunkt t_n sind,
- $u_{n+1}, \dot{u}_{n+1}, \ddot{u}_{n+1}$ die entsprechenden Werte zum Zeitpunkt $t_{n+1} = t_n + \Delta t$.

Dynamische Gleichungssysteme

Iterative Methode

Newmark-beta

Typische Wahl der Parameter

Die Parameter β und γ steuern die Genauigkeit und Stabilität der Methode:

- $\beta = 0.25$ und $\gamma = 0.5$: Standardwahl für unbedingte Stabilität, geeignet für quasistatische und langsame dynamische Systeme.
- $\beta = 0$ und $\gamma = 0.5$: Entspricht der Zentrale-Differenzen-Methode, stabil nur für kleine Zeitschritte.

Formeln zur Berechnung der Beschleunigung, Geschwindigkeit und Verschiebung

Um die Bewegungsgleichung zu lösen, berechnen wir \ddot{u}_{n+1} , \dot{u}_{n+1} und u_{n+1} iterativ für jeden Zeitschritt Δt .

Schritt 1: Berechnung der effektiven Steifigkeitsmatrix und des effektiven Lastvektors

Die Bewegungsgleichung für u_{n+1} wird durch Einsetzen in die Verschiebungsgleichung umgestellt. Daraus ergibt sich eine modifizierte Gleichung:

$$(M + \gamma \Delta t C + \beta \Delta t^2 K) \ddot{u}_{n+1} = F_{\text{eff}}$$

wobei F_{eff} der effektive Lastvektor ist:

$$F_{\text{eff}} = F_{n+1} - C(\dot{u}_n + (1 - \gamma)\Delta t \ddot{u}_n) - K\left(u_n + \Delta t \dot{u}_n + \frac{\Delta t^2}{2}(1 - 2\beta)\ddot{u}_n\right)$$

Schritt 2: Berechnung der Beschleunigung

Die Beschleunigung \ddot{u}_{n+1} wird durch Lösen der modifizierten Gleichung bestimmt:

$$\ddot{u}_{n+1} = (M + \gamma \Delta t C + \beta \Delta t^2 K)^{-1} F_{\text{eff}}$$

Schritt 3: Aktualisierung von Geschwindigkeit und Verschiebung

Mit der berechneten Beschleunigung \ddot{u}_{n+1} werden dann die Geschwindigkeit und Verschiebung berechnet:

1. Geschwindigkeit \dot{u}_{n+1} :

$$\dot{u}_{n+1} = \dot{u}_n + \Delta t ((1 - \gamma)\ddot{u}_n + \gamma \ddot{u}_{n+1})$$

2. Verschiebung u_{n+1} :

$$u_{n+1} = u_n + \Delta t \dot{u}_n + \frac{\Delta t^2}{2} ((1 - 2\beta)\ddot{u}_n + 2\beta \ddot{u}_{n+1})$$

Dynamische
Gleichungssysteme

Iterative Methode

Newmark-beta

Vorteile und Anwendungsbereiche der Newmark- β -Methode

Eigenschaften	Beschreibung
Unbedingte Stabilität	Für $\beta = 0.25$ und $\gamma = 0.5$ unbedingte Stabilität für lineare Systeme
Gute Genauigkeit für Schwingungsprobleme	Ideal für Schwingungen und langsam variierende dynamische Lasten
Quasistatische und dynamische Analyse	Anwendung in der linearen und nichtlinearen Dynamik und bei langsamen Lastverläufen
Anpassbare Parameter	Flexibilität durch β und γ -Anpassung für spezifische Stabilitätsanforderungen

Die **Newmark- β -Methode** ist eine vielseitige und weit verbreitete Methode in der dynamischen FEM-Analyse. Sie ermöglicht es, durch die Wahl von β und γ entweder Stabilität oder Genauigkeit zu priorisieren, und bietet so eine flexible und effiziente Lösung für eine Vielzahl von dynamischen Systemen.

Zusammenfassung der Effekte:

Effekt	Wert von γ	Wert von β	Auswirkung auf die Lösung
Unbedingte Stabilität	$\gamma = 0.5$	$\beta = 0.25$	Standardwahl, stabil, wenig numerische Dämpfung
Mehr Dämpfung	$\gamma > 0.5$	$\beta > 0.25$	Starke numerische Dämpfung, hohe Frequenzen werden gedämpft
Phasengenauigkeit	$\gamma = 0.5$	$\beta < 0.25$	Bessere Erhaltung von Schwingungen, weniger Phasenverschiebung
Instabilität	$\gamma < 0.5$	$\beta < 0.25$	Kann zu Instabilität und unphysikalischen Schwingungen führen

Dynamische Gleichungssysteme

Iterative Methode

Newmark-beta

Erklärungen:

1. Newmark-Beta-Methode:

- Die Funktion `newmark_beta` implementiert die Newmark- β -Methode zur Berechnung von Verschiebung, Geschwindigkeit und Beschleunigung über die Zeit. Die Parameter β und γ können angepasst werden, um die Stabilität und Genauigkeit zu steuern.

2. Plotting:

- Die Ergebnisse für Verschiebung, Geschwindigkeit und Beschleunigung werden über die Zeit geplottet. Jeder Freiheitsgrad (Degree of Freedom, DOF) wird separat dargestellt, sodass die Dynamik jedes einzelnen Freiheitsgrads verfolgt werden kann.

3. Anwendung:

- Der Code zeigt den zeitlichen Verlauf der dynamischen Reaktionen des Systems, indem er die Ergebnisse für jede Variable und jeden Freiheitsgrad in separaten Diagrammen darstellt. Dies hilft bei der Analyse des Systemverhaltens unter konstanter Krafteinwirkung.

```
python
import numpy as np
import matplotlib.pyplot as plt

def newmark_beta(M, C, K, F, u0, v0, a0, dt, steps, beta=0.25, gamma=0.5):
    """
    Newmark-beta method for solving a dynamic system.

    Parameters:
    M (ndarray): Mass matrix (6x6).
    C (ndarray): Damping matrix (6x6).
    K (ndarray): Stiffness matrix (6x6).
    F (ndarray): Force matrix over time (steps x 6).
    u0 (ndarray): Initial displacement vector (6,).
    v0 (ndarray): Initial velocity vector (6,).
    a0 (ndarray): Initial acceleration vector (6,).
    dt (float): Time step.
    steps (int): Number of time steps.
    beta (float): Newmark-beta parameter.
    gamma (float): Newmark-gamma parameter.

    Returns:
    ndarray: Displacement, velocity, and acceleration over time.
    """
    # Initialize displacement, velocity, and acceleration arrays
    u = np.zeros((steps, len(u0)))
    v = np.zeros((steps, len(u0)))
    a = np.zeros((steps, len(u0)))

    # Set initial conditions
    u[0, :] = u0
    v[0, :] = v0
    a[0, :] = a0

    # Effective stiffness matrix (constant for linear system)
    K_eff = M + gamma * dt * C + beta * dt**2 * K

    # Time-stepping loop
    for n in range(1, steps):
        # Effective force vector at time step n+1
        F_eff = F[n, :] - C @ (v[n-1, :] + (1 - gamma) * dt * a[n-1, :]) - K @ (u[n-1, :]

        # Solve for acceleration at time step n+1
        a[n, :] = np.linalg.solve(K_eff, F_eff)

        # Update velocity and displacement for time step n+1
        v[n, :] = v[n-1, :] + dt * ((1 - gamma) * a[n-1, :] + gamma * a[n, :])
        u[n, :] = u[n-1, :] + dt * v[n-1, :] + 0.5 * dt**2 * ((1 - 2 * beta) * a[n-1, :] +

    return u, v, a

# Define system matrices (6x6 example)
K = np.array([
    [10, -2, 0, 0, 0, 0],
    [-2, 12, -3, 0, 0, 0],
    [0, -3, 15, -4, 0, 0],
    [0, 0, -4, 18, -5, 0],
    [0, 0, 0, -5, 20, -6],
    [0, 0, 0, 0, -6, 22]
])
```

```
M = np.eye(6) * 2 # Mass matrix (diagonal)
C = np.eye(6) * 0.1 # Damping matrix (diagonal)

# Initial conditions (system at rest initially)
u0 = np.zeros(6)
v0 = np.zeros(6)
a0 = np.zeros(6)

# Time parameters
dt = 0.01 # Time step size
steps = 100 # Number of time steps

# Constant external force applied (for simplicity)
F = np.ones((steps, 6)) * 10 # Constant force on each degree of freedom over time

# Solve the system with Newmark-beta method
u, v, a = newmark_beta(M, C, K, F, u0, v0, a0, dt, steps, beta=0.25, gamma=0.5)

# Time vector for plotting
time = np.linspace(0, dt * (steps - 1), steps)

# Plot Displacement over time
plt.figure(figsize=(10, 6))
for i in range(6):
    plt.plot(time, u[:, i], label=f"DOF {i+1}")
plt.title("Displacement over Time for Each Degree of Freedom")
plt.xlabel("Time [s]")
plt.ylabel("Displacement")
plt.legend()
plt.grid(True)
plt.show()

# Plot Velocity over time
plt.figure(figsize=(10, 6))
for i in range(6):
    plt.plot(time, v[:, i], label=f"DOF {i+1}")
plt.title("Velocity over Time for Each Degree of Freedom")
plt.xlabel("Time [s]")
plt.ylabel("Velocity")
plt.legend()
plt.grid(True)
plt.show()

# Plot Acceleration over time
plt.figure(figsize=(10, 6))
for i in range(6):
    plt.plot(time, a[:, i], label=f"DOF {i+1}")
plt.title("Acceleration over Time for Each Degree of Freedom")
plt.xlabel("Time [s]")
plt.ylabel("Acceleration")
plt.legend()
plt.grid(True)
plt.show()
```

Dynamische Gleichungssysteme

Iterative Methode

Newmark-beta

```
import numpy as np
import matplotlib.pyplot as plt

def newmark_beta(M, C, K, F, u0, v0, a0, dt, steps, beta=0.25, gamma=0.5):
    """
    Newmark-beta method for solving a dynamic system.

    Parameters:
    M (ndarray): Mass matrix (6x6).
    C (ndarray): Damping matrix (6x6).
    K (ndarray): Stiffness matrix (6x6).
    F (ndarray): Force matrix over time (steps x 6).
    u0 (ndarray): Initial displacement vector (6,).
    v0 (ndarray): Initial velocity vector (6,).
    a0 (ndarray): Initial acceleration vector (6,).
    dt (float): Time step.
    steps (int): Number of time steps.
    beta (float): Newmark-beta parameter.
    gamma (float): Newmark-gamma parameter.

    Returns:
    ndarray: Displacement, velocity, and acceleration over time.
    """
    # Initialize displacement, velocity, and acceleration arrays
    u = np.zeros(steps, len(u0))
    v = np.zeros(steps, len(u0))
    a = np.zeros(steps, len(u0))

    # Set initial conditions
    u[0, :] = u0
    v[0, :] = v0
    a[0, :] = a0

    # Effective stiffness matrix (constant for linear system)
    K_eff = M + gamma * dt * C + beta * dt**2 * K

    # Time-stepping loop
    for n in range(1, steps):
        # Effective force vector at time step n+1
        F_eff = F[n, :] - C @ (v[n-1, :] + (1 - gamma) * dt * a[n-1, :]) - K @ (u[n-1, :] + dt * v[n-1, :] + 0.5 * (1 - 2 * beta) * dt**2 * a[n-1, :])

        # Solve for acceleration at time step n+1
        a[n, :] = np.linalg.solve(K_eff, F_eff)

        # Update velocity and displacement for time step n+1
        v[n, :] = v[n-1, :] + dt * ((1 - gamma) * a[n-1, :] + gamma * a[n, :])
        u[n, :] = u[n-1, :] + dt * v[n-1, :] + 0.5 * dt**2 * ((1 - 2 * beta) * a[n-1, :] + 2 * beta * a[n, :])

    return u, v, a

# Define system matrices (6x6 example)
K = np.array([
    [10, -2, 0, 0, 0, 0],
    [-2, 12, -3, 0, 0, 0],
    [0, -3, 15, -4, 0, 0],
    [0, 0, -4, 18, -5, 0],
    [0, 0, 0, -5, 20, -6],
    [0, 0, 0, 0, -6, 22]
])

M = np.eye(6) * 2 # Mass matrix (diagonal)
C = np.eye(6) * 0.1 # Damping matrix (diagonal)

# Initial conditions (system at rest initially)
u0 = np.zeros(6)
v0 = np.zeros(6)
a0 = np.zeros(6)

# Time parameters
dt = 0.01 # Time step size
steps = 100 # Number of time steps

# Constant external force applied (for simplicity)
F = np.ones(steps, 6) * 10 # Constant force on each degree of freedom over time

# Solve the system with Newmark-beta method
u, v, a = newmark_beta(M, C, K, F, u0, v0, a0, dt, steps, beta=0.25, gamma=0.5)

# Time vector for plotting
time = np.linspace(0, dt * (steps - 1), steps)

# Plot Displacement over time
plt.figure(figsize=(10, 6))
for i in range(6):
    plt.plot(time, u[:, i], label=f'DOF {i+1}')
plt.title("Displacement over Time for Each Degree of Freedom")
plt.xlabel("Time [s]")
plt.ylabel("Displacement")
plt.legend()
plt.grid(True)
plt.show()

# Plot Velocity over time
plt.figure(figsize=(10, 6))
for i in range(6):
    plt.plot(time, v[:, i], label=f'DOF {i+1}')
plt.title("Velocity over Time for Each Degree of Freedom")
plt.xlabel("Time [s]")
plt.ylabel("Velocity")
plt.legend()
plt.grid(True)
plt.show()

# Plot Acceleration over time
plt.figure(figsize=(10, 6))
for i in range(6):
    plt.plot(time, a[:, i], label=f'DOF {i+1}')
plt.title("Acceleration over Time for Each Degree of Freedom")
plt.xlabel("Time [s]")
plt.ylabel("Acceleration")
plt.legend()
plt.grid(True)
plt.show()
```