

Algorithmik kontinuierlicher Systeme Zusammenfassung

Autor: Julian Kotzur - julian-kotzur@live.de

Matrix-Vektor-Multiplikation $A\vec{b}$ mit vollbesetzter Matrix A	$\mathcal{O}(n^2)$
Matrix-Vektor-Multiplikation $A\vec{b}$ mit m-diagonaler Matrix A	$\mathcal{O}(m \cdot n)$
Matrix-Vektor-Multiplikation $A\vec{b}$ mit tridiagonaler Matrix A	$\mathcal{O}(n)$
Matrix-Vektor-Multiplikation mit Rang-1-Matrix und $A = uv^T$	$\mathcal{O}(n)$
Berechnung der euklidischen Vektor-Norm	$\mathcal{O}(n)$
Matrix-Matrix-Multiplikation AB mit vollbesetzten Matrizen A und B	$\mathcal{O}(n^3)$
Matrix-Matrix-Multiplikation AB mit vollbesetzten Matrizen A und m-diagonaler Matrix B	$\mathcal{O}(m \cdot n^2)$
Matrix-Matrix-Multiplikation AB mit vollbesetzter Matrix A und tridiagonaler Matrix B	$\mathcal{O}(n^2)$
Berechnung der Inversen einer invertierbaren (n×n)-Diagonalmatrix	$\mathcal{O}(n)$
Rang einer invertierbaren n×n-Matrix	n
Bestimmung der LR- bzw. QR-Zerlegung	$\mathcal{O}(n^3)$
Bestimmung der LR-Zerlegung einer m-diagonalen Matrix	$\mathcal{O}(m^2 \cdot n)$
Lösen eines Gleichungssystems mit gegebener LR-/ QR-/ SVD-Zerlegung	$\mathcal{O}(n^2)$
Lösen eines Gleichungssystems mit gegebener LR-/ QR-Zerlegung bei m-diagonaler Matrix	$\mathcal{O}(m \cdot n)$
Lösen eines Gleichungssystems ohne LR-Zerlegung	$\mathcal{O}(n^3)$
Vorwärts- / Rückwärtssubstitution	$\mathcal{O}(n^2)$
Bestimmung der Determinante bei gegebener LR-Zerlegung	$\mathcal{O}(n)$
Bestimmung der Determinante bei gegebener PLR-Zerlegung	$\mathcal{O}(n^2)$
Bestimmung der Betragsdeterminante bei gegebener QR-/ PLR-Zerlegung	$\mathcal{O}(n)$
Auswertung eines Punktes auf einer Bezier-Kurve mit De Casteljaun	$\mathcal{O}(n^2)$
Anzahl der Kontrollpunkte einer Bezier-Kurve mit Grad n	n+1
Grad einer Bezier-Kurve mit n Kontrollpunkten	n-1
Ein Iterationsschritt bei Jacobi bzw. Gauss-Seidel bzw. SOR	$\mathcal{O}(n^2)$
Anzahl der Iterationsschritte bei SOR bzw. Aitken-Neville	$\mathcal{O}(\sqrt{n})$
Anzahl der Iterationsschritte bei Jacobi bzw. Gauss-Seidel	$\mathcal{O}(n)$
Konvergenzordnung Jacobi, Gauss-Seidel oder SOR	p = 1
Berechnung der Fourier-Transformation OHNE FFT	$\mathcal{O}(n^2)$
Berechnung der Fourier-Transformation MIT FFT	$\mathcal{O}(n \cdot \log(n))$
Bestimmung der Koeffizienten bei Interpolation mit Newtonbasis	$\mathcal{O}(n^2)$
Approximationsfehler Stückweise konstante Interpolation	$\mathcal{O}(h)$
Approximationsfehler stückweise linearer Interpolation mit h	$\mathcal{O}(h^2)$
Approximationsfehler Catmull-Rom-Interpolation	$\mathcal{O}(h^3)$
Approximationsfehler Trapez-Regel mit $h = (b - a)/n$	$\mathcal{O}(h^2)$
Approximationsfehler iterierter Simpson-Regel mit $h = (b - a)/n$	$\mathcal{O}(h^4)$
Konditionszahl einer Rotationsmatrix	1

Python Grundlagen

Grundlagen

1. Einrücken ist Pflicht
2. +, -, *, %, **, /-Operatoren sind kanonisch
3. // - Operator dividiert und rundet ab (danach float var)
4. Variablenzuweisung:
Mehrfachzuweisung: a, b = 5, 6 (links nach rechts)
Werte tauschen: a, b = b, a
Gleichen Wert zuweisen: a = b = c = 0
5. Strings: name = 'string' → kann man addieren
6. Boolesche Ausdrücke werden kanonisch verwendet
7. Boolesche Funktionen werden ausgeschrieben: and/ not/ or
8. Funktionen: def <name> (<parameter>):
9. Nichts tun: pass
10. Modul: import <modulname> as <Pseudonym>
11. Typkonvertierung: datentyp(Variable)
12. Eingabe: s = input("Text fuer Ausgabe")
input wird immer als String übergeben
13. Ausgabe: print(...)

Datenstrukturen

1. Listen:
liste = [3, 2, 1, 'foo'] // Erstellen
liste.append(val) // item hinten anhängen
liste.insert(idx, val) // item an index einfügen
liste.remove(val) // entfernt erstes item mit wert val
liste.sort // sortiert liste nach gröÙe
tuple(liste) // Wandelt Liste in Tupel um
2. Tupel: tupel = (3, 2, 1, 'foo')
3. Hashtable: hash = {1:'Hallo', 2:'Welt', 3:42}
4. Zugriff: name[<index>]. Index bei Hashtable individuell
5. Teilmengen auswählen: Start : Ende, wobei Start eingeschlossen ist, Ende nicht: [0,1,2,3,4][1:3] → [1, 2]
6. Sprunggröße von Teilmengen: [Start:Ende:Sprunggröße]
7. in-Operator: Testet auf Zugehörigkeit: 5 in liste → true

Kontrollfluss

1. Bedingungen wie in Java nur mit Einrückungen
2. while-Schleife: while <booleanFlag>
3. for-Schleife:
(a) for i in [1,2,3]
(b) for i in range(n) // geht von 0 bis n-1
(c) for i in range(n,m,x) // jedes xte Element

Lambda-Funktion

1. Kleine anonyme Funktion
2. Syntax: lambda arguments : expression
3. Beispiel Summe: x = lambda a, b, c : a + b + c
print(x(5, 6, 2))

Numpy

1. import numpy as np
2. Arraygröße / Form ausgeben als Tupel:
mtx.shape → (2, 3)
3. Array erstellen:
mtx = np.array([[1,2,3],[4,5,6]]) // Definierte Matrix
mtx = np.eye(<dimension>) // Einheitsmatrix
mtx = np.full(<shape>,<wert>) // Alle Elemente = wert
4. Wichtige Matrix Operationen:
mtx = np.transpose(mtx) // Transponieren
mtx = np.dot(matrix1, matrix2) // Matrixmultiplikation
vec = mtx.flatten() // Zeilenweise ausgelesener Vektor
vec = mtx.flatten('F') // Spaltenweise ausgelesener Vektor
vec = np.arange(wert,wert) // Vektor analog zu range
vec = np.linalg.solve(matrix,vektor) // LGS lösen
5. Wichtige Mathe Operationen:
x = np.absolute(y) // Betrag eines Wertes

Slicing

1. Anwendbar auf Strings, Arrays, Listen, Tupeln, etc.
2. Damit wird ein Teil einer Datenstruktur ausgewählt:
a[start:stop] // items start through stop-1
a[start:] // items start through the rest of the array
a[:stop] // items from the beginning through stop-1
a[:] // a copy of the whole array
a[start:stop:step] // start through not past stop, by step
3. Mit negativen Werten kann man auch arbeiten:
a[-1] // last item in the array
a[-2:] // last two items in the array
a[:-2] // everything except the last two items
4. Weitere Beispiele:
a[::-1] // all items in the array, reversed
a[1::-1] // the first two items, reversed
a[:-3:-1] // the last two items, reversed
a[-3::-1] // everything except the last two items, reversed

Zusatz

1. Zuweisungen kopieren nicht (Referenzübergabe)
2. Kopieren einer Variable:
varNeu = varAlt.copy()
3. Variable Anzahl an Argumenten: def name(*args):
4. Standartwerte für Argumente: def name(x, y=2):
5. Fehlertesten: try ... except (Einrücken!)
6. Klassen und OOP:
(a) Erstellen: class klassenName:
(b) Kontruktor: def __init__(self, <attribute...>)
(c) Klassenmethode: def name(self, <parameter>)
(d) Objekt erstellen: name = klassenName(<attribute>)
(e) Methode aufrufen:name.methodenname(<parameter>)

Grundwissen

Grundlagen 2x2-Matrix

Determinante:

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

Charakteristisches Polynom:

$$\det \begin{pmatrix} a_{11} - \lambda & a_{12} \\ a_{21} & a_{22} - \lambda \end{pmatrix} = (a_{11} - \lambda) \cdot (a_{22} - \lambda) - a_{12}a_{21}$$

Inverse der Matrix:

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \Rightarrow A^{-1} = \frac{1}{ad - bc} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix}$$

Rotationsmatrix:

$$R_\omega = \begin{pmatrix} \cos(\omega) & -\sin(\omega) \\ \sin(\omega) & \cos(\omega) \end{pmatrix}$$

Grundlagen 3x3-Matrix

Determinante:

$$\det \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = \begin{pmatrix} = a_{11}a_{22}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{21}a_{32} \\ -a_{13}a_{22}a_{31} - a_{12}a_{21}a_{33} - a_{11}a_{23}a_{32} \end{pmatrix}$$

Grundlagen Matrizen Allgemein:

1. Charakteristisches Polynom:

Determinante der Matrix mit Diagonaleinträge - λ

2. Eigenwerte:

Nullstellen des charakteristischen Polynoms

3. Eigenvektoren:

Eigenwerte einsetzen und Gaußen

4. Zusammenhang zw. Eigenwerten und Eigenvektoren:

$$Av = \lambda v \text{ Beispiel: } \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 3 \\ 3 \end{pmatrix} = 3 \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

5. Rang:

Anzahl der linear unabhängigen Zeilen
Gaußen und alle nicht-null-Zeilen zählen

6. Determinante einer Dreiecksmatrix:

Diagonalen zusammenmultiplizieren

Vektor-Matrix-Multiplikation:

$$\text{Red} = \begin{pmatrix} \text{Blue1} \cdot \text{Green1} \\ \text{Blue2} \cdot \text{Green2} \\ \text{Blue3} \cdot \text{Green3} \end{pmatrix} +$$

Matrix-Matrix-Multiplikation:

Grundlagen Matrizen speziell:

1. Eigenschaften orthogonaler Matrizen Q:

$Q^T Q = \text{Einheitsmatrix}$

$Q Q^T = \text{Einheitsmatrix}$

Spalten oder Zeilen von Q bilden eine Orthonormalbasis

2. Matrizen transponieren:

Zeilen Spaltenweise auslesen

3. Diagonalmatrizen invertieren:

Diagonaleinträge hoch -1 rechnen

4. Matrizen Definitheit:

Positiv Definit: Alle Eigenwerte größer 0

Positiv Semidefinit: Alle Eigenwerte größer gleich 0

Negativ Definit: Alle Eigenwerte kleiner 0

Negativ Semidefinit: Alle Eigenwerte kleiner gleich 0

5. Singuläre / Reguläre Matrizen:

Eine Matrix ist Reguläre, wenn sie eine Inverse besitzt. Ansonsten ist sie singular. Eine Inverse lässt sich bilden, wenn alle Spalten linear unabhängig sind (Keine null Zeile beim Gaußen)

6. Euklidische Vektornorm:

Wurzel der (Summe der (Einträge zum Quadrat))

Gauß-Algorithmus:

1. Auswahl des Ausgangswertes x (Diagonalenwert)

2. Pro Element i der Spalte von x:

(a) Divident D zwischen i und x bilden: i/x

(b) Zeile von i mit D * (Zeile von x) subtrahieren

Python-Code

Vorwärtssubstitution:

→ Zur Lösung von linken unteren Dreiecksmatrizen
→ Lösung in x[]

```
for j = 1...n
  x[j] = c[j]
  for i = 1...j-1
    x[j] -= L[j, i] * x[i]
  x[j] = x[j] / L[j, j]
```

Rückwärtssubstitution:

→ Zur Lösung von rechten oberen Dreiecksmatrizen
→ Lösung in x[]

```
for j = n...1
  x[j] = c[j]
  for i = j+1...n
    x[j] -= R[j, i] * x[i]
  x[j] = x[j] / R[j, j]
```

Quadratische Konvergenz:

Fehler zur optimalen Lösung nimmt quadratisch ab

Lattice-Boltzmann-Methode

1. Dichte: Summe aller Einträge einer Zelle
2. Geschwindigkeit: Unterscheidung zw. x und y Richtung. Summe (Einträge mal Richtung). Das ganz noch mit 1 / Dichte multiplizieren

Gleitkommaarithmetik

1. Assoziativgesetz gilt nicht: $(x + y) + z \neq x + (y + z)$
2. Distributivgesetz gilt nicht: $x \cdot (y + z) \neq x \cdot y + x \cdot z$

Matrizen - Datenstrukturen und Verfahren

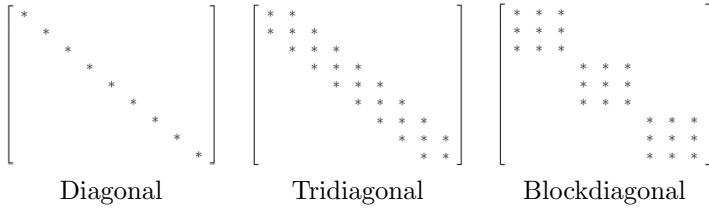
Dünn besetzte Matrix:

Eine Matrix in der ein Großteil der Einträge 0 ist. Dies tritt oft bei Gleichungssystemen auf.

Hinweis:

Die Indizierung in AlgoKs beginnt üblicherweise bei 1.

Matrix-Arten



Compressed Row Storage (CRS)

Vorgehensweise:

1. Value-Array mit allen Werten der Matrix erstellen. Diese Zeilenweise auslesen
2. Spalten-Array anlegen, welches für jeden Wert im Value-Array die dazugehörige Spalte speichert. Hierbei auf Indizes achten!
3. Pointer-Array anlegen, welches die Indizes des Value-Array speichert, ab welchem eine neue Zeile beginnt. Bei einer Nullzeile wird ein Pointer 2 mal gesetzt (d.h. 2 mal gleiche Zahl direkt hintereinander in ptr gibt null-Zeile an)

Nachteil:

→ Einfügen und Löschen ist schwierig

→ Kein beliebiger Zugriff auf einzelne Elemente

$$\text{Beispiel: } \begin{pmatrix} 5 & 0 & 0 \\ 0 & 3 & 7 \\ 0 & 4 & 0 \end{pmatrix} \begin{array}{l} \text{val} \\ \text{col_ind} \\ \text{row_ptr} \end{array} = \begin{array}{l} [5, 3, 7, 4] \\ [1, 2, 3, 2] \\ [1, 2, 4, 5] \end{array}$$

Compressed Column Storage (CCS)

Funktioniert Analog zu CRS.

$$\text{Beispiel: } \begin{pmatrix} 7 & 1 & 0 \\ 2 & 0 & 7 \\ 0 & 4 & 0 \end{pmatrix} \begin{array}{l} \text{val} \\ \text{row_ind} \\ \text{col_ptr} \end{array} = \begin{array}{l} [7, 2, 1, 4, 7] \\ [1, 2, 1, 3, 2] \\ [1, 3, 5, 6] \end{array}$$

Blocked Compressed Row Storage (BCRS)

Im Value Array werden Matrizen gespeichert.

$$\text{Beispiel: } \begin{pmatrix} 5 & 1 & 0 & 0 \\ 9 & 8 & 0 & 0 \\ 0 & 0 & 0 & 6 \\ 0 & 0 & 3 & 2 \end{pmatrix} \begin{array}{l} \text{val} \\ \text{col_ind} \\ \text{row_ptr} \end{array} = \left[\begin{pmatrix} 5 & 1 \\ 9 & 8 \end{pmatrix}, \begin{pmatrix} 0 & 6 \\ 3 & 2 \end{pmatrix} \right]$$

Multiplikation mit CRS

1. Zeile aus ptr-Array wählen.
2. Value anhand col-Array mit Eintrag aus Vek multiplizieren
3. Bei mehreren Values pro Zeile: Ergebnisse addieren
4. Hinweis: Reihenfolge und Transponierungen beachten!

Komprimierte Matrix transponieren

1. val bleibt gleich
2. col_ptr = row_ptr
3. row_ind = col_ind

Diskretisierung

Grundwissen

1. Uniform: Nahezu gleichverteilte Punkte
2. Nicht Uniform: Ungleich verteilte Punkte
3. Uniforme Quantisierung: Zerlegung eines Wertebereiches in disjunkte, gleich große Teilintervalle
4. Kondition: Eigenschaft des Problems. Ein Problem ist gut konditioniert, wenn Fehler in den Eingangsdaten nicht verstärkt werden, d.h. kleine Eingabestörungen führen nur zu kleinen Ergebnisänderungen (nicht zu großen)
5. Stabilität: Eigenschaft des Algorithmus. Dieser ist stabil, wenn sich die Rechenfehler nicht anhäufen. Dies gilt nicht für mehrfache Berechnung mit demselben Algorithmus.
6. Kondition linearer Gleichungssysteme:
 - (a) Matrixnorm wählen: Spalten-/Zeilensummennorm
 - (b) Entsprechende Normen berechnen
 - (c) Maximalwert durch Minimalwert teilen

$$\text{Beispiel: } A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

$$\text{Spaltensummennorm} = \frac{\max\{|1| + |3|, |2| + |4|\}}{\min\{|1| + |3|, |2| + |4|\}}$$

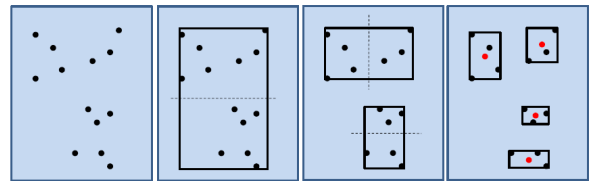
$$\text{Zeilensummennorm} = \frac{\max\{|1| + |2|, |3| + |4|\}}{\min\{|1| + |2|, |3| + |4|\}}$$

7. Fehlerquellen numerischer Algorithmen:

- (a) Eingabeungenauigkeit, Rundungsfehler
- (b) Diskretisierungs- /Quantisierungsfehler

Median-Cut-Algorithmus

1. Finde best-fittende bounding box (Rechteck / Quadrat)
2. Teile Quader an längster Kante so dass beide Teile gleich viele Punkte enthalten
3. Schritt 1 und 2 rekursiv wiederholen



Vektorielle Variante

1. Initialisierung: Wähle k zufällige Mittelwerte
2. Zuordnung: Jedes Datenobjekt wird demjenigen Cluster zugeworfen, bei dem die Cluster Varianz am wenigsten erhöht wird
3. Berechne die Mittelpunkte der Cluster neu
4. Schritt 2 und 3 werden bis zur Konvergenz wiederholt

LR-Zerlegung

L ist linke untere, R ist rechte obere Dreiecksmatrix
Lösung des Gleichungssystems:

$$Ax = b \Leftrightarrow LRx = b \Rightarrow Ly = b \Rightarrow Rx = y$$

Algorithmus ohne Pivotsuche

Wann kann man das verwenden:

1. Wenn eine Permutationsmatrix gegeben ist und ich diese mit der Ausgangsmatrix A multiplizieren
2. symmetrisch positiv definite Ausgangsmatrix

L-Matrix und R-Matrix erstellen:

1. L Matrix als Einheitsmatrix erstellen
2. Gauß-Algorithmus auf Ausgangsmatrix anwenden
3. Zeilenmultiplikatoren der aktuelle Spalte in L eintragen

Beispiel:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ -2 & 1 & 0 & 0 \\ 3 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 2 & 4 & 3 & 5 \\ -2 \rightarrow 0 & 1 & 1 & 2 \\ +3 \rightarrow 0 & -4 & -7 & -6 \\ +2 \rightarrow 0 & 1 & -8 & 4 \end{pmatrix} \begin{pmatrix} \text{II} - (-2) \text{ I} \\ \text{III} - 3 \text{ I} \\ \text{IV} - 2 \text{ I} \end{pmatrix}$$

Algorithmus mit Pivotsuche

Ausgangsmatrix A mit Dimension n ist gegeben:

1. Initialisieren: L als Nullmatrix, R als Kopie von A, P als Einheitsmatrix. Alles mit Dimension n
2. Für i = 1 bis i = n-1:
 - (a) Initialisiere Permutationsmatrix P_i als Einheitsmatrix
 - (b) Spalte j des Pivots aus Spalte i ab Zeile i finden
 - (c) In L / R / P_i Zeilen i und j tauschen
 - (d) In R in Spalte i ab Zeile j Gaußen und in L die Multiplikatoren (Gekürztes / Pivot) eintragen
3. $P = \prod_{i=1}^{n-1} P_i$, Diagonale in L mit 1er befüllen, $R = R$

Tipp: Kürzen ist nicht erlaubt!

Beispiel:

Initialisieren:

$$L = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}, R = \begin{pmatrix} 1 & 7 & 0 \\ 4 & 9 & 2 \\ 2 & 1 & 0 \end{pmatrix}, P_1 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

1. Schritt:

$$L = \begin{pmatrix} 0 & 0 & 0 \\ 1/4 & 0 & 0 \\ 1/2 & 0 & 0 \end{pmatrix}, R = \begin{pmatrix} 4 & 9 & 2 \\ 0 & 19/4 & 1/2 \\ 0 & -7/2 & 2 \end{pmatrix}, P_2 = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

2. Schritt:

$$L = \begin{pmatrix} 0 & 0 & 0 \\ 1/4 & 0 & 0 \\ 1/2 & -14/19 & 0 \end{pmatrix}, R = \begin{pmatrix} 4 & 9 & 2 \\ 0 & 19/4 & 1/2 \\ 0 & 0 & 45/19 \end{pmatrix}, P_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Fazit:

$$L' = \begin{pmatrix} 1 & 0 & 0 \\ 1/4 & 1 & 0 \\ 1/2 & -14/19 & 1 \end{pmatrix}, R = \begin{pmatrix} 4 & 9 & 2 \\ 0 & 19/4 & 1/2 \\ 0 & 0 & 45/19 \end{pmatrix}, P = P_1 \cdot P_2 \cdot P_3$$

QR-Zerlegung

Q ist orthogonal, R ist rechte obere Dreiecksmatrix
Lösung des Gleichungssystems:

$$Ax = b \Leftrightarrow QRx = b \Rightarrow Rx = Q^T b$$

Algorithmus mittels Givensrotation

$$\text{Rotationsmatrix: } J_{ij}(\Phi) = \begin{pmatrix} 1 & & & \\ & \cos_{jj}(\Phi) & & \\ & \sin_{ji}(\Phi) & & \\ & & & 1 \end{pmatrix}$$

1. Initialisiere Q = Einheitsmatrix und R = A
2. Für j = 1 bis j = m // Spalten m-lang

- (a) Für i = j+1 bis i = n // Zeilen n-lang
- (b) Bilde $J_{ij}(\Phi)$ mit: $\text{sign}()$ = Vorzeichen()

$$\cos(\Phi) = \frac{\text{sign}(a_{jj}) \cdot a_{jj}}{\sqrt{a_{jj}^2 + a_{ij}^2}} \quad \sin(\Phi) = -\frac{\text{sign}(a_{ij}) \cdot a_{ij}}{\sqrt{a_{jj}^2 + a_{ij}^2}}$$

- (c) Setze in $R = J_{ij}(\Phi) \cdot R$
- (d) Setze $Q = Q \cdot J_{ij}(\Phi)^T$

3. Gib Q und R aus

Tipp: $a \cdot \cos(w) + b \cdot \sin(w) = 0 \Rightarrow w = -\tan^{-1}(a/b)$

Beispiel:

$$R = \begin{pmatrix} 4 & 3 \\ 3 & 5 \end{pmatrix}, Q = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \Rightarrow \begin{matrix} \cos(\Phi) = 4/5 \\ \sin(\Phi) = -3/5 \end{matrix} \Rightarrow J_{ij} = \begin{pmatrix} 4/5 & 3/5 \\ -3/5 & 4/5 \end{pmatrix}$$

$$\text{Result: } R = \begin{pmatrix} 5 & 27/5 \\ 0 & 11/5 \end{pmatrix}, Q = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 4/5 & -3/5 \\ 3/5 & 4/5 \end{pmatrix} = \begin{pmatrix} 4/5 & -3/5 \\ 3/5 & 4/5 \end{pmatrix}$$

Algorithmus mittels Householder

Note: n-1 Householder-Spiegelungen werden benötigt

1. Setze R = A und Q = Einheitsmatrix
2. Für i = 1 bis i = n-1: // Spalten

- (a) Setze $v_i = a_{*i}$ // v_i = i-te Spalte von A
- (b) Berechne: $x_i = \left(\sum_{j=i}^n (a_{ji}^2) \right)^{1/2} \cdot \text{sign}(a_{ii})$ // Norm
- (c) Setze $v_i = v_i + x_i \cdot e_i$ mit e_i = i-ter Einheitsvektor
- (d) Berechne Q_i = Einheitsmatrix $-\frac{2v_i v_i^T}{v_i^T v_i}$
- (e) Setze $R = Q_i \cdot R$ und $Q = Q \cdot Q_i^T$

3. Gib Q und R zurück

$$\text{Beispiel: } A = R = \begin{pmatrix} 4 & 3 \\ 3 & 5 \end{pmatrix} Q = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}:$$

$$v_0 = \begin{pmatrix} 4 \\ 3 \end{pmatrix}, x = \sqrt{4^2 + 3^2} \cdot \text{sign}(+4) = 5 \Rightarrow v_0 = \begin{pmatrix} 9 \\ 3 \end{pmatrix} \rightarrow \begin{pmatrix} 3 \\ 1 \end{pmatrix}$$

$$Q_0 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} - \frac{2 \cdot \begin{pmatrix} 3 \\ 1 \end{pmatrix} \cdot \begin{pmatrix} 3 & 1 \end{pmatrix}}{\begin{pmatrix} 3 & 1 \end{pmatrix} \cdot \begin{pmatrix} 3 \\ 1 \end{pmatrix}} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} - \begin{pmatrix} 18 & 6 \\ 6 & 2 \end{pmatrix} / 10$$

$$\Rightarrow Q = Q \cdot Q_0^T = \begin{pmatrix} -0.8 & -0.6 \\ -0.6 & 0.8 \end{pmatrix}, R = Q_0 \cdot R = \begin{pmatrix} -5 & -5.4 \\ 0 & 2.2 \end{pmatrix}$$

Betrag der Determinante berechnen

1. $|\det(A)| = |\det(Q)| \cdot |\det(R)|$
2. $|\det(Q)| = 1$ da orthogonal
3. $|\det(R)|$ = Produkt der Diagonaleinträge

Lineare Ausgleichsproblem /-polynom

Überbestimmte Ausgleichungsprobleme

1. A sei $n \times m$ -Matrix mit $n > m$ (n Zeilen, m Spalten)
2. Mehr Gleichungen als Variablen: Überbestimmt
3. Weniger Gleichungen als Variablen: Unterbestimmt
4. Das Residuum $r(x) = b - Ax$ kann durch kein $x=0$ werden
5. Daher versucht man das Residuum zu minimieren. Dieser Ansatz wird Lineares Ausgleichsproblem genannt.

Theorie - Least Square Minimierung

1. Methode der kleinsten Quadrate
2. Finde \hat{x} , so dass $\|r(\hat{x})\| \leq \|r(x)\|$ für alle $x \in \mathbb{R}^m$
3. Minimierung über die euklidische Norm $\|r\| = \sqrt{\sum r_i^2}$.
4. \Rightarrow Immer eindeutig bestimmtes kleinstes Residuum.
5. \Rightarrow ABER: \hat{x} ist nur dann eindeutig, wenn spalten von A linear unabhängig.

Bestimmung von $Ax=b$ aus Messpunkten

1. Gegeben: Messpunkte $(x_1, y_1), \dots, (x_n, y_n)$
2. Gegeben: gesuchtes Polynom: $y = a_m x^m + \dots + a_1 x + a_0$
3. Die Matrix A hat $m+1$ Spalten und n Zeilen. Bei einer Geraden Gleichung hat A somit 2 Spalten.
Matrix Baukonstrukt:

$$A = \begin{pmatrix} 1 & x_1 & \dots & x_1^m \\ \vdots & \vdots & \vdots & \vdots \\ 1 & x_n & \dots & x_n^m \end{pmatrix} \text{ mit } b = \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix}$$

Methodik: QR-Zerlegung

1. Gegeben: QR-Zerlegung von A
2. Es können nur die oberen m Gleichungen einer überbestimmten Matrix erfüllt werden, alle anderen Zeilen werden verworfen.
3. Gelöst wird dies durch Rückwärtssubstitution
4. Die verworfenen Zeilen bestimmen das Residuum

Methodik: Normalengleichung

1. Nur für Überbestimmte Gleichungssysteme mit vollem Rang (alle Zeilen linear unabhängig)
2. Theorie: Bestimme $\min \|b - Ax\|^2$

Löse: $A^T A x = A^T b$

3. Bei Rückwärtssubstitution entspricht die erste Lösung dem a des höchsten Index. Somit entspricht die letzte Lösung dem a mit dem kleinsten Index

Cramersche Regel

Zu lösen ist das Gleichungssystem $Ax = b$, $A \in \mathbb{R}^{n \times n}$.

1. Berechne die Determinante $\det(A)$ der Matrix A.
2. Für $i = 1, \dots, n$:
Ersetze Spalte i durch b. (Nenne neue Matrix A_i)
Berechne Determinante $\det(A_i)$.
Berechne $x_i = \frac{\det(A_i)}{\det(A)}$
3. Gib Lösungsvektor $x = (x_1, \dots, x_n)^T$ aus.

Interpolation

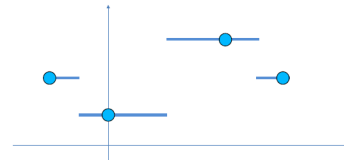
Interpolation: Kurve verläuft exakt durch alle Punkte

Stückweise konstante Interpolation / Nearest Neighbor

$$p(x) = \begin{cases} y_1 & \text{falls } x_1 \leq x \leq 1/2(x_1 + x_2) \\ y_1 & \text{falls } 1/2(x_1 + x_2) \leq x \leq 1/2(x_2 + x_3) \\ \vdots & \vdots \\ y_n & \text{falls } 1/2(x_{n-1} + x_n) \leq x \leq x_n \end{cases}$$

Beispiel:

x_i	-1	0	2	3
y_i	2	1	3	2



Stückweise lineare Interpolation

1. Um den Wert an Stelle x anzunähern, suche die nächstgelegenen Stützstellen x_i und x_{i+1}
2. Interpolierter Wert:
$$p(x) = m_i(x - x_i) + y_i \text{ mit } m_i = \frac{y_{i+1} - y_i}{x_{i+1} - x_i}$$
3. Somit wird für jedes Punkte-Paar eine Gerade berechnet, welche diese verbindet

Catmull-Rom-Interpolation

1. lokale und glatte Interpolation
2. Gegeben: Stützstellen x_n und Stützwerte y_n
3. Erster Punkt durch Vorwärtsdifferenz: $y'_1 = \frac{y_2 - y_1}{x_2 - x_1}$
4. Letzter Punkt durch Rückwärtsdifferenz: $y'_n = \frac{y_n - y_{n-1}}{x_n - x_{n-1}}$
5. Zwischenpunkte aus zentraler Differenz: $y'_i = \frac{y_{i+1} - y_{i-1}}{x_{i+1} - x_{i-1}}$
6. Berechne auf jedem Teilintervall $[x_i, x_{i+1}]$ das Polynom:

$$p_i(x) = a_0(x_{i+1} - x)^3 + a_1(x_{i+1} - x)^2(x - x_i) + a_2(x_{i+1} - x)(x - x_i)^2 + a_3(x - x_i)^3$$

$$\text{mit } a_0 = \frac{y_i}{(x_{i+1} - x_i)^3}, a_1 = 3a_0 + \frac{y'_i}{(x_{i+1} - x_i)^2}$$

$$\text{und } a_3 = \frac{y_{i+1}}{(x_{i+1} - x_i)^3}, a_2 = 3a_3 - \frac{y'_{i+1}}{(x_{i+1} - x_i)^2}$$

Newton Polynom aufstellen

Verwendet wird der Algorithmus von Aitken-Neville

Gegeben: n Punkte (x_i, y_i)

1. Punkte als Tabelle schreiben
2. Jeweils zwei Punkte für neues y berechnen: $\frac{y_i - y_{i+1}}{x_i - x_{i+1}}$

Dies passiert pyramidenförmig, bei jeder Pyramidenstufe nimmt der Indexabstand von x um 1 zu, die verwendeten y sind diejenigen der vorherigen Stufe

3. Wiederhole dies, bis nur noch ein y Wert vorhanden ist
4. Die Koeffizienten a_i sind die obersten y-Werte der Pyramide startend mit $a_0 = y_0$
5. Polynom: $p(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) + \dots$

Lagrange Polynom

Gegeben: n Punkte (x_i, y_i)

1. Berechne i Lagrange-Basen mit:

$$L_i(x) = \frac{\prod_{j=0, j \neq i}^n (x - x_j)}{\prod_{j=0, j \neq i}^n (x_i - x_j)} \text{ mit: } j \neq i$$

2. Berechne Lagrange Polynom:

$$p(x) = \sum_{i=0}^n y_i \cdot L_i(x)$$

Interpolation am Dreieck

1. Gegeben sind 3 Punkte R,S,T in einer Ebene, welche ein Dreieck aufspannen
2. Im Bezug auf das Dreieck lassen sich Punkte darstellen anhand der Formel $P = \alpha R + \beta S + \gamma T$. Die griechischen Koordinaten nennt man baryzentrische Koordinaten
3. Berechnung der Koordinaten eines Punktes ausgehend von den Baryzentrischen Koordinaten:

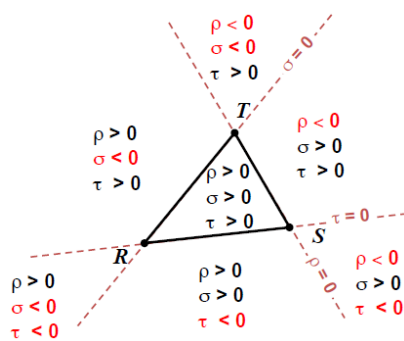
$$x(P) = \frac{\alpha x_R + \beta x_S + \gamma x_T}{\alpha + \beta + \gamma} \quad y(P) = \frac{\alpha y_R + \beta y_S + \gamma y_T}{\alpha + \beta + \gamma}$$

4. Berechnung der baryzentrischen Koordinaten ausgehend von einem Punkt: $Ax = b$, mit x Koordinaten von R,S,T als erste Zeile, mit y Koordinaten von R,S,T als zweite Zeile und einer 1er Zeile als letztes. b sind die gegebenen Punktkoordinaten $(x, y, 1)^T$. Aus der letzten Zeile wird γ , aus der zweiten β und aus der ersten α
5. Funktionswerte mittels Interpolation bestimmen. Dazu sind die 3 Punkte R,S,T gegeben und eine baryzentrische Koordinate:

$$f(x, y) = \alpha f(x_R, y_R) + \beta f(x_S, y_S) + \gamma f(x_T, y_T)$$

Ist keine Funktion gegeben, können die Funktionswerte aus den Dreieckskoordinaten gegeben sein und ersetzen somit die $f(\dots)$ -Rechnungen rechts der Gleichung

6. Vorzeichen der baryzentrischen Koordinaten:



Grundwissen - Gerade durch 2 Punkte

- (a) Gegeben: 2 Punkte a, b
- (b) Gesucht: $mx + t$
- (c) $m = \frac{b_y - a_y}{b_x - a_x}$ und $t = b_y - mb_x$

Bilineare Interpolation

1. Gesucht: Wert einer Funktion an Punkt $P=(x,y)$
2. Gegeben: 4 Punkte mit Funktionswerten:
 $A = (x_1, y_1), B = (x_2, y_1), C = (x_1, y_2), D = (x_2, y_2)$
3. Berechnung:

$$f(x, y_1) = \frac{x_2 - x}{x_2 - x_1} f(A) + \frac{x - x_1}{x_2 - x_1} f(B)$$

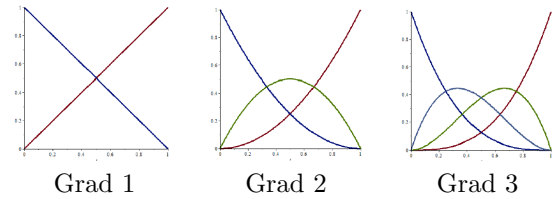
$$f(x, y_2) = \frac{x_2 - x}{x_2 - x_1} f(C) + \frac{x - x_1}{x_2 - x_1} f(D)$$

$$f(x, y) = \frac{y_2 - y}{y_2 - y_1} f(x, y_1) + \frac{y - y_1}{y_2 - y_1} f(x, y_2)$$

4. Hint: Es gibt dazu auch eine einzelne Formel

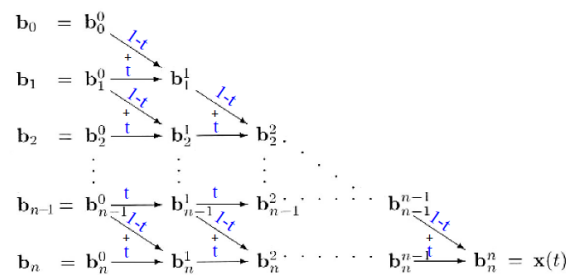
Freiformkurven

Bernsteinpolynome



Bezierkurven

1. Gegeben sind Kontrollpunkte bzw. Bezier-Punkte
2. Aus den Bezier-Punkte erhält man das Kontrollpolygon
3. Eigenschaften von Bezier-Kurven:
 - (a) Kurve liegt in der konvexen Hülle der Kontrollpunkte
 - (b) Endpunkt-Interpolation: Die Endpunkte des Kontrollpolygons sind die Endpunkte der Bezier-Kurve
 - (c) Affine Invarianz: Für eine affine Abbildung, müssen nur die Kontrollpunkte affin transformiert werden
 - (d) Variantsreduktion: Die Kurve schwankt höchstens so stark wie das Kontrollpolygon
4. Berechnung mit de Casteljaun:



5. Midpoint Subdivision Algorithmus:

Aus dem deCasteljau-Dreieck lassen sich zwei weitere Kontrollpolygone bilden (Mittelpunkt als gemeinsamer Punkt), auf welche der Algorithmus rekursiv angewendet wird

Singulärwertzerlegung - Erstellung

Gegeben: Eine Matrix $A \in R^{m \times n}$

Gesucht: Matrizen $U \in R^{m \times m}, \Sigma \in R^{m \times n}, V^T \in R^{n \times n}$

1. Schritt: $A^T A$ berechnen
2. Schritt: Eigenwerte von $A^T A$ bestimmen
3. Schritt: Eigenvektoren v_i berechnen und normieren
4. Schritt: triviale Singulärwerte berechnen: Wurzeln der Eigenwerte. Und nach Größe sortieren!
5. Schritt: Aus Singulärwerten α wird U berechnet:

$$u_i = Av_i / \alpha_i$$

6. Schritt: Letzt Spalten von U durch gleichsetzen der bekannten u mit 0 berechnen
7. Schritt: Matrix U erstellen (Nach Größe der Eigenvektoren absteigend und am Ende zusätzlicher Vektor)
8. Schritt: Matrix Σ erstellen:
Wurzel der Eigenwerte als Diagonale der $R^{m \times n}$ -Matrix
9. Schritt: Matrix V^T erstellen
Eigenvektormatrix transformieren

Eigenschaften der SVD

1. Spektralsatz sichert die Existenz von U, V
2. U ist orthogonal
3. V ist orthogonal

Matrixnormen

Spaltensummennorm

Gegeben: Matrix $A \in R^{m \times n}$

Es wird die Summe der Beträge der Spalten betrachtet

$$\|A\|_1 = \max\{\Sigma \text{ Spalte } 1, \dots, \Sigma \text{ Spalte } n\}$$

Zeilensummennorm

Gegeben: Matrix $A \in R^{m \times n}$

Es wird die Summe der Beträge der Zeilen betrachtet

$$\|A\|_\infty = \max\{\Sigma \text{ Zeile } 1, \dots, \Sigma \text{ Zeile } n\}$$

Frobeniusnorm

Gegeben: Matrix $A \in R^{m \times n}$

$$\|A\|_F = \sqrt{a_{11}^2 + a_{12}^2 + \dots + a_{nm}^2}$$

In der SVD von A nur Σ betrachten (U und V^T ignorieren)

Spektralnorm

Gegeben: Matrix $A \in R^{m \times n}$

Bei SVD ist $A^T A = \Sigma$

$$\|A\|_2 = \sqrt{\text{Maximaler Eigenwert der Matrix } A^T A}$$

Konditionszahl berechnen:

Bei Normen mit einer Max-Suche, teilt man den Max-Wert durch den Min-Wert

SVD Zusatzaufgaben:

Pseudoinverse

1. Sei A^+ die Pseudoinverse zu A
2. Eigenschaften: $A \cdot A^+ \cdot A = A, A^{++} = A$
3. Bei SVD ist die Pseudoinverse: $V \Sigma^+ U^T$
4. Lösen eines Gleichungssystems mit Pseudoinverse:
 $U \Sigma V^T x = b \Rightarrow x = U^T \Sigma (V^T)^T b$
5. In welchem Sinne löst die Pseudo-Inverse das LGS:
Minimiert das Residuum

Rang bestimmen

$\text{rang}(A) = r =$ Anzahl der von null verschiedenen Singulärwerte

Kern bestimmen

Letzten $n-r$ Spalten aus V^T : $\ker(A) = \text{span}\{V_{r+1}^T, \dots, V_n^T\}$

Bild bestimmen

Ersten r Spalten aus U: $\text{im}(A) = \text{span}\{U_1, \dots, U_r\}$

Rang-n-Approximation

Nur die ersten n Singulärwerte verbleiben in der Σ -Matrix. Der Rest wird auf 0 gesetzt. Dann muss Sigma nur noch in der Ausgangs SVD ersetzt werden.

SVD und Vektoren

1. $V^T \vec{x}$: Drehung von x
2. $\Sigma \vec{x}$: Streckung / Stauchung von x
3. $U \vec{x}$: Drehung von x

PCA

1. Nützlich für Datenkompression
2. Gegeben: Eine Menge an Punkten mit Anzahl n
3. Mittelpunkt aus Punkten berechnen:

$$M = \frac{1}{n} \cdot \sum_{i=1}^n \text{Punkt}_i$$

4. Mittelpunkt für jeden Punkt berechnen:

$$P'_i = P_i - M$$

5. symmetrische Kovarianzmatrix C bilden:

$$A = (P'_0 \ P'_1 \ P'_n) \Rightarrow C = AA^T$$

6. Eigenwerte λ_m von C berechnen und nach Größe ordnen
7. Eigenvektoren (Hauptachsen) v_m berechnen und nach Größe von λ ordnen
8. C ähnlich zu SVD aufspalten:

$$C = (v_1, \dots, v_m) \begin{pmatrix} \lambda_1 & 0 & 0 \\ \vdots & \ddots & \vdots \\ 0 & 0 & \lambda_m \end{pmatrix} \begin{pmatrix} v_1 \\ \vdots \\ v_m \end{pmatrix}$$

9. object oriented bounding box: Rechteck parallel zu den Eigenvektoren

Iterative Verfahren mit Vektoren

Gauss-Seidel Verfahren

1. Im Allgemeinen nicht parallelisierbar
2. Konvergiert doppelt so schnell wie das Jacobi-Verfahren.
3. Gegeben: Matrix A, Startvektor x^0 .
4. Gauss-Seidel-Iterationsschritt:

$$(D + L)x^{i+1} = b - Rx^i$$

→ Lösbar durch Vorwärtssubstitution.

5. Hinweis: SOR-Verfahren mit $w = 1$
6. Tipp: In der Matrix zum Vorwärtssubstituieren dürfen Zeilen vertauscht werden

```
x = np.zeros((n,1))
while np.linalg.norm(b - A@x) > eps:
    for k in range(n):
        left = sum(a[k,0:k]*x[0:k,0])
        right = sum(a[k,k+1:]*x[k+1:,0])
        x[k] = (b[k]-left-right) / a[k,k]
return x
```

Jacobi Verfahren

1. Besser parallelisierbar.
2. Gegeben: Matrix A, Startvektor x^0 .
3. Jacobi-Iterationsschritt:

$$Dx^{i+1} = b - (L + R)x^i$$

```
x = np.zeros((n,1))
while np.linalg.norm(b-A@x) >eps:
    x_new = np.zeros((n,1))
    for k in range(n):
        left = sum(a[k,0:k]*x[0:k,0])
        right = sum(a[k,k+1:]*x[k+1:,0])
        x_new[k] = (b[k] - left-right)/a[k,k]
    return x_new
```

SOR Verfahren

1. Schnelle Konvergenz durch Überrelaxation
2. Gegeben: Matrix A, Startvektor x^0 , Faktor $w \in (1, 2)$.
3. SOR-Iterationsschritt:

$$(1/wD + L)x^{i+1} = b - ((1 - 1/w)D + R)x^i$$

```
x = np.zeros((n,1))
while np.linalg.norm(b-Ax) > eps:
    for k in range(n):
        left = sum(a[k,0:k]*x[0:k,0])
        right = sum(a[k,k+1:]*x[k+1:,0])
        x_tmp = (b[k]-left-right)/a[k,k]
        x[k] = (1-w) * x[k] + x_tmp * w
    return x
```

Praktische Konvergenzkriterien

1. A strikt diagonaldominant (Für alle Spalten gilt: Betrag des Diagonalelements $>$ Summe des Betrags der restlichen Spaltenelemente)
⇒ Alle drei Verfahren konvergieren.
2. A ist positiv definit (d.h alle Eigenwerte von A sind > 0)
⇒ SOR und Gauss-Seidel konvergieren.
3. Sind A und $D - 2A$ positiv definit so konvergiert auch das Jacobi-Verfahren.
4. A schwach-diagonaldominant (Für jede Spalte muss gelten: Betrag des Diagonalelements \geq Summe des Betrags der restlichen Spaltenelemente) und Matrix unzerlegbar, d.h. der Graph von A ist stark zusammenhängend (jeder Knoten ist von jedem Knoten aus erreichbar)
→ Alle drei Verfahren konvergieren.

Graph einer Matrix A

1. Gegeben: Matrix $A \in \mathbb{R}^{n \times n}$
2. Erstelle Graph mit n Knoten.
3. Für alle nicht null Einträge der Matrix a_{ij} der Matrix muss eine Kante von i nach j eingetragen werden.

Additive Zerlegung einer Matrix A

1. Gegeben: Matrix $A \in \mathbb{R}^{n \times n}$,
2. Gesucht: Matrixzerlegung $A = L + D + R$
3. L ist linke untere Dreiecksmatrix der Einträge von A ohne Diagonale.
4. D sind nur die Diagonalelemente der Einträge von A.
5. R ist rechte obere Dreiecksmatrix der Einträge von A ohne Diagonale

Poissongleichung - Tabellenaufgabe

1. Gegeben: Tabelle mit Gitterweite h und Fehlertoleranz r
2. Gesucht: Vollständige Tabelle
3. Jacobi-Tabelle:
 - (a) h gleich, r kleiner: Letzter Eintrag + Differenz der letzten beiden Einträge
 - (b) h kleiner, r gleich: Letzter Eintrag * 4
4. Gauss-Seidel: Halb so viele Schritte wie Jacobi
5. SOR: Wurzel an Schritten wie Jacobi und ansonsten mit den gleichen Regeln wie Jacobi

Iterationsmatrix

1. $V_J = -D^{-1}(L + R)$
2. $V_{GS} = -(L + D)^{-1}R$
3. $V_{SOR} = -(1/wD + L)^{-1}(w-1/wD + R)$

Iterative Verfahren Allgemein

Newton-Verfahren

→ Konvergiert nur für Startwerte nahe der Nullstelle

→ Konvergiert sehr schnell (quadratisch)

Gegeben: Funktion $f(x)$ und Startwert x_0

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

Newton-Verfahren im R^n

1. Gegeben: Funktion $f(x)$ und Startwerte x_0
2. Hierfür wird mit dem Gradientenverfahren die Jacobi-Matrix berechnet.

$$J(x) = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \cdots & \frac{\partial f_n}{\partial x_n} \end{pmatrix}$$

$$\Rightarrow x_{i+1} = x_i - (J(x_i))^{-1} f(x_i)$$

Sekanten-Verfahren

1. Konvergiert nur für Startwerte nahe der Nullstelle
2. Konvergiert schnell
3. Gegeben: zwei Startwerte x_0, x_1
4. Für x_n wird eine Gerade g zw. y_{n-1} und y_{n-2} gezogen
5. x_n ist der Schnittpunkt von g mit der x-Achse

Bisektionsverfahren

1. Konvergiert immer, aber langsam
2. Gegeben: zwei Startwerte x_0, x_1
3. Mittelpunkt x_2 bestimme und das ganze mit x_0, x_2 wiederholen, falls $f(x_0) \cdot f(x_2) < 0$, ansonsten mit x_1, x_2 wiederholen

Regula falsi

1. Bisektionsverfahren, nur statt dem Mittelpunkt x_2 zw. x_0, x_1 nimmt man den Schnittpunkt der Geraden zw. x_0, x_1 mit der x-Achse

Hinweis - Ausgelassen

Banachscher Fixpunktsatz, weil der halt echt kacke in den Folien erklärt ist. Sagt einem, dass das Verfahren Konvergiert

Numerische Integration / Quadratur

Newton-Cotes-Formel

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

Rechtecksregel:

1. Gegeben: Sample Punkt x_0
2. Linksregel: $x_0 = a$ bzw. Rechtsregel: $x_0 = b$
3. Mittelpunkregel: $x_0 = a+b/2$

$$\int_a^b f(x) dx = (b-a)f(x_0)$$

Trapezregel:

1. Gegeben: Zwei Sample Punkte: a, b

$$\int_a^b f(x) dx = (b-a) \frac{f(a) + f(b)}{2}$$

Partition:

1. obrige Regeln auf Teilmengen anwenden
2. Alle Zwischenergebnisse addieren

Romberg-Verfahren:

1. Vergleich zweier Verfahren
2. Gegeben: Schrittweite h (Abstand zw. Punkten)

$$T_f^1(h) = \frac{4 \cdot (\text{Result klein } h) - (\text{Result groß } h)}{3}$$

Nichtlineare Optimierung

Optimierung mittels Newton-Verfahren

1. Abstiegsverfahren, falls $\nabla^2 f(x')$ positiv definit
2. Sehr aufwändig für große n
3. Hesse-Matrix: In der Diagonalen 2 mal nach Variable abgeleitet, ansonsten einmal nach Zeilen und einmal nach Spalten Variable

$$x_{i+1} = x_i - [\nabla^2 f(x_i)]^{-1} \nabla f(x_i)$$

Gradientenverfahren / Steepest descent

1. Gegeben: Startwert x_0 und Schrittweite t_0
2. Abstiegsverfahren, falls $\nabla f(x_i)^T s_i < 0 \Rightarrow s_i = -\nabla f(x_i)$
3. Optimale Schrittweite t kann näherungsweise durch das Lösen von $f_i(t) = f(x_i + t s_i)$ bestimmt werden.

$$x_{i+1} = x_i + t_i s_i$$

Konjugiertheit zweier Vektoren zu A

1. Gegeben: Matrix A und 2 Vektoren u, v
2. A-konjugiert, falls $u^T A v = 0$

Abbruchbedingungen

1. Maximale Anzahl an Iterationsschritte gemacht
2. Gewünschte Genauigkeit erreicht