Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät
Department Informatik
Lehrstuhl für Systemsimulation 10

Skript der Veranstaltung

Algorithmik kontinuierlicher Systeme (AlgoKS)

gehalten im Sommersemester 2018
von Prof. Dr. Ulrich Rüde

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Vorbemerkung

Diese(s) Skript enthält den Inhalt der Vorlesung „Algorithmik kontinuierlicher Systeme“ des Sommersemester 2018 bei Prof. Dr. Ulrich Rüde. Es wurde anhand von Mitschriften in LaTeX gesetzt und erhebt daher weder Anspruch auf Korrektheit noch Vollständigkeit und ist ***offensichtlich inoffiziell***. Bei Unstimmigkeiten und evtl. vorhandenen Fehlern bitte ich um eine Email an untenstehende Adresse. Dieses Skript stellt damit insbesondere **keine** offizielle Veröffentlichung des Lehrstuhl für Systemsimulation 10 am Department Informatik der Friedrich-Alexander-Universität Erlangen-Nürnberg dar.

Florian Frank — florian.ff.frank@fau.de
Philipp Hutterer — philipp.hutterer@fau.de
Version vom 14. April 2019

Inhaltsverzeichnis

KAPITEL 1

Direkte Verfahren für Lineare Gleichungssysteme

Wie seit den einführenden Mathematikvorlesungen bekannt ist, lassen sich lineare Gleichungssysteme im Allgemeinen als eine Gleichung der Form

$$Ax = b,$$

mit $A \in \mathbb{K}^{n\times m}, b \in \mathbb{K}^n$ und $x \in \mathbb{K}^m$ darstellen. Vorerst wollen wir den Standardfall einer $n \times n$-Matrix betrachten.
Wie der gemeine Informatikstudierende bereits seit letztem Semester weiß, unterteilt man Algorithmen zum Lösen von linearen Gleichungssystemen in zwei Kategorien, **direkte** und *iterative* Verfahren. Letztere werden dann zusammen mit einer Daseinsberechtigung in Kapitel 7 näher besprochen. Wir beschäftigen uns damit jetzt mit den direkten Verfahren zum Lösen von linearen Gleichungssystemen, wobei hier die Lösung – bei Existenz – nach *endlich* vielen Schritten erreicht wird, vorausgesetzt man rechnet exakt[1]. Wir klassifizieren diese Verfahren nun weiter und schränken uns dabei zunächst auf (voll besetzte) **nicht singuläre** Systemmatrizen mit einer rechten Seite ein. Eine oft verwendete Strategie dabei ist es die Systemmatrix A so zu faktorisieren, dass sich die daraus entstehenden Teilprobleme leicht lösen lassen. Zu den wichtigsten und hier besprochenen Verfahren zählen die **LR-Zerlegung** (→ Kapitel 1.1), sowie die **QR-Zerlegung** (→ Kapitel 1.2).

1.1 LR-Zerlegung

Ziel der LR-Zerlegung ist eine Faktorisierung der Matrix A in eine linke (untere) und eine rechte (obere) Dreieckmatrix. Der klassische Gauss-Eliminations-Algorithmus, welcher sich bemüht die Matrix sukzessive in Dreiecksgestalt zu überführen, baut dabei auf Transformationen über *Zeilenoperationen* und *Permutationen* auf. Zeilenoperationen ergeben sich dabei durch die Multiplikation mit geeigneten Matrizen, wir wollen diese dann auch Gauß-Scherung nennen.

1.1.1 Gauß-Scherungen

Bevor wir Gauß-Scherungen definieren können, müssen wir uns Matrizen definieren, die zu einer Zeile i einer Matrix das α-fache einer anderen Zeile j addieren. Wir definieren also

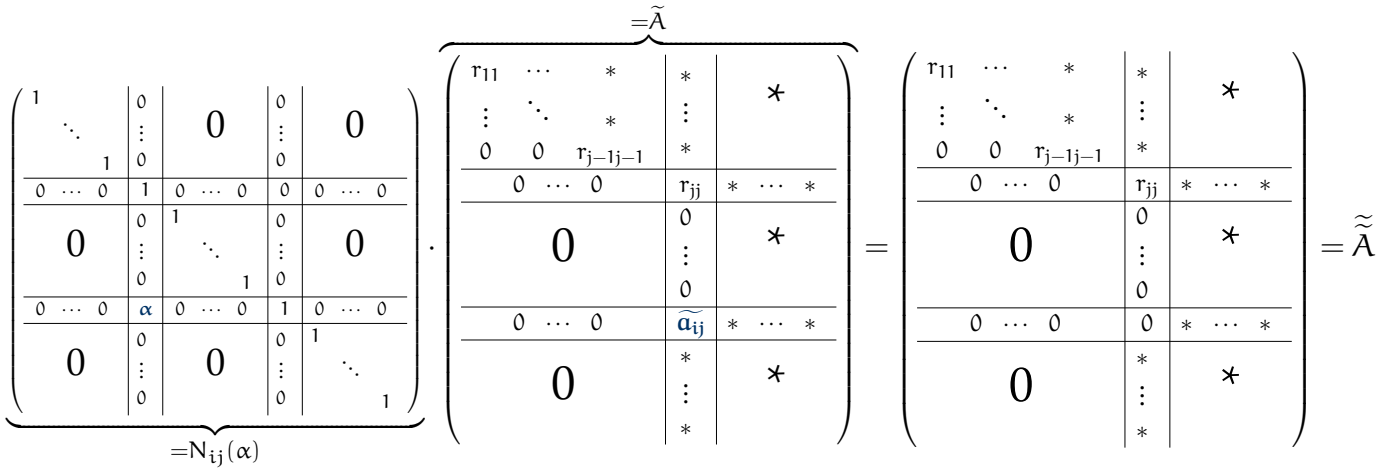
$$N_{ij}(\alpha) = \left(\eta_{ij}\right)_{rc} = \begin{cases} 1 & \text{, wenn } r = c \\ \alpha & \text{, wenn } r = i \text{ und } c = j \\ 0 & \text{sonst} \end{cases}.$$

Ebenfalls erkennen wir leicht, dass aufgrund der Gestalt von $N_{ij}(\alpha)$ sich das Inverse der Matrix leicht berechnen lässt mit

$$N_{ij}^{-1}(\alpha) = N_{ij}(-\alpha).$$

[1] Das das *„exakte“* Rechnen zum Problem werden kann, haben wir bereits am einführenden Beispiel gesehen ... (genaueres siehe später)

Wollen wir nun einen Wert einer „Matrixzelle“ eliminieren, so braucht es nun nur eine bestimmte Matrix $N_{ij}(\alpha)$. Wir überlegen uns, dass der Zusammenhang



gelten muss, weswegen wir leicht feststellen können, dass r_{jj} das Pivotelement und α den Zusammenhang $\alpha = -\frac{\widetilde{a_{ij}}}{r_{jj}}$ erfüllen muss. Merkt man sich nun nur die Eliminationsfaktoren der Gauss-Elimination, so generiert man leicht eine LR-Zerlegung der Ursprungsmatrix A. Wir eliminieren dabei Spalte für Spalte mit den Spalteneliminationsfaktoren α_i, ein solcher Schritt heißt dann „Gauß-Scherung“.
Zum Faktorisieren braucht es dann bei einer $n \times n$-Matrix $n-1$ solcher Scherungen, dabei gilt, sofern A_k den Zustand der Matrix nach der k-ten Scherung bezeichnet, dass

$$A_k = L_k A_{k-1} = L_k \cdot L_{k-1} \cdot \dots \cdot L_1 \cdot A.$$

Nach der $n-1$-ten Scherung erhalten wir mit A_{n-1} genau eine rechte obere Dreiecksmatrix. Wir sehen somit auch, dass mit

$$R = A_{n-1} \quad \text{und} \quad L = R^{-1} = \prod_{i=1}^{n-1} L_i^{-1}$$

eine LR-Zerlegung der Matrix A gegeben ist.

Berechnung von L

! Wichtig anzumerken ist, dass die Berechnung von L nicht über klassische Berechnungen der Inversen stattfinden darf. Stattdessen nutzt man die Form der L_i aus und erkennt dadurch, dass sich L aus den Spalten der L_i mit umgekehrten Vorzeichen zusammensetzt.

Für diese hier vorgestelle Variante der klassischen LR-Zerlegung *ohne Pivotsuche* ist ein Pythonkodesschnipsel rechts dargestellt. Es sei selbstverständlich angenommen, dass zuvor ein **import** numpy as np stattgefunden hat. Der Aufwand für die LR-Zerlegung lässt sich auch somit herleiten. Wir brauchen $\sum_{i=1}^{n-1} i$ Divisionen und jeweils $\sum_{i=1}^{n-1} i^2$ Additionen und Multiplikationen, was somit insgesamt einen Gesamtaufwand von $\frac{2n^3}{3}$ ergibt.

```python
def lu_decompose(A):
    """Return (L, R), such that L @ R = A"""
    #  You will do this in exercise 3 ...
    pass
```

1.1.2 LR-Zerlegung mit Spaltenpivotsuche

Wir erkennen selbstverständlich das größte Manko an dem eben entwickelten Verfahren, denn es funktioniert nur, wenn die Pivotelemente $a_{ii} \neq 0$ sind. Sollte dem nicht so sein, so müssen zwingend Zeilenvertauschungen durchgeführt werden. Diese werden durch Multiplikationen mit Permutationsmatrizen dargestellt. Man sollte die Pivotsuche allerdings vor allem wegen der numerischen Stabilität der Verfahren immer durchführen, man kann auf sie verzichten, sofern symmetrisch positiv definite Matrizen vorliegen.

Eine mögliche Strategie der Pivotsuche

Man sucht in der Restspalte das **betragsmäßig größte** Element und versucht dies durch das Vertauschen von Zeilen auf die **Diagonale** zu bringen.

Sei A eine $n \times n$-Matrix und $(e_i)_{1 \leqslant i \leqslant n}$ die Standardbasisvektoren, so folgen die Permutationsmatrizen, welche wir mit $P_{i,j}$ – für das Vertauschen der i-ten mit der j-ten Zeile – bezeichnen wollen, dem Schema

$$P_{i,j} = \Big(\underset{\text{1. Spalte}}{e_1}, \quad \ldots, \quad \underset{\text{i. Spalte}}{e_j}, \quad \ldots, \quad \underset{\text{j. Spalte}}{e_i}, \quad \ldots, \quad \underset{\text{n. Spalte}}{e_n} \Big).$$

Wir erkennen dann, dass sich Permutationen und Gauß-Scherungen gegenseitig abwechseln, aufgrund der einfachen Struktur der $P_{i,j}$ diese sich aber „vorziehen“ lassen. Im Allgemeinen gilt somit, dass sich die rechte obere Dreiecksmatrix durch

$$R = L_{n-1} \cdot P_{n-1,a} \cdot \cdots \cdot L_1 \cdot P_{1,i} \cdot A$$

berechnet und damit dann

$$A = P_{1,i} \cdot L_1^{-1} \cdot \cdots \cdot P_{n-1,a} \cdot L_{n-1}^{-1} = \prod_{i=1}^{n-1} P_{i,r_i} \cdot \prod_{i=1}^{n-1} \widetilde{L_i} \cdot R = P \cdot L \cdot R,$$

wobei sich dann $\widetilde{L_i}$ aus L_i^{-1} durch das Vertauschen der jeweiligen Elemente in der Spalte hervorgeht, sobald man ein P vorzieht.

Verfahren 1.1 (*Allgemeines Vorgehen*)

Für $i = 1, \ldots, n-1$ mache:

(1) *Finde das betragsmäßig größte Element in der i-ten (Rest-)spalte.*
Wir wollen dann die Zeile des Elements mit ℓ bezeichnen.

(2) *Vertausche nun in A und b die beiden (Rest-)zeilen i und ℓ $\rightsquigarrow$ Merke Permutation in Permutationsmatrix $P_{i,\ell}$*

(3) *Schreibe die i-te Zeile von R und i-te Spalte von L und führe die Gaußelimination durch. $\rightsquigarrow$ Erhalte dadurch ein aktualisiertes A.*

Man löst das System $Ax = b$ dann mittels $LRx = P^{-1}b$, wobei die Inverse von P schnell bestimmt ist, man dreht dazu einfach das Produkt um.

1.1.3 Vorwärts- und Rückwärtselimination

1.1.3.1 Vorwärtselimination

$$x_i = \frac{c_i - \sum_{j=1}^{i-1} l_{ij} \cdot x_j}{l_{ii}} \quad \text{für } 1 \leqslant i \leqslant n$$

```
def forward_substitute(L, b):
    """Return y such that L @ y = b"""
    ...
    for j in ...:
        ...
        for i in range(j):
            ...
        ...
    ...
    pass
```

1.1.3.2 Rückwärtselimination

$$x_i = \frac{c_i - \sum_{j=i+1}^{n} r_{ij} \cdot x_j}{r_{ii}} \quad \text{für } 1 \leqslant i \leqslant n$$

```
def backward_substitute(R, b):
    """Return x such that R @ x = y"""
    ...
    for j in ...:
        ...
        for i in range(j):
            ...
        ...
    ...
    pass
```

1.1.4 Anwendungen

1.1.4.1 Determinanten

Benötigt man die Determinante einer $n \times n$-Matrix (und nur einer solchen, da Determinanten nur für quadratische Matrizen definiert sind), so löst man dies über eine LR-Zerlegung (ohne Pivotsuche). Man nutzt dazu die Formeigenschaften von L und R aus, indem man feststellt, dass

$$\det(A) = \det(L \cdot R) = \det(L) \cdot \det(R) = 1 \cdot \prod_{i=1}^{n} r_{ii}.$$

Für die Determinante mit Pivotsuche ergibt sich ein Vorfaktor:

$$\det(A) = \det(P \cdot L \cdot R) = \det(P) \cdot \det(L) \cdot \det(R) = \pm 1 \cdot 1 \cdot \prod_{i=1}^{n} r_{ii}.$$

Die Determinante von P ist ± 1 je nach gerader $(+1)$ oder ungerader (-1) Anzahl der Vertauschungen (Zeile- oder Spalte), die benötigt werden, um daraus eine Einheitsmatrix zu machen ($\rightarrow$ siehe C1).

1.1.4.2 Effizientes Lösen mehrerer Gleichungssysteme mit gleicher Koeffizientenmatrix A

Der allgemeine Ansatz ist es, nur **einmal** die LR-Zerlegung von A zu bestimmen (im Aufwand $\mathcal{O}(n^3)$) und danach für jede rechte Seite eine Lösung mittels Vorwärts- und Rückwärtssubstitution zu bestimmen. Sind jedoch **alle rechten Seiten** zu Beginn bereits bekannt, so werden sie in einer m-spaltigen Matrix B zusammengefasst. Man löst dann $AX = B$, wobei $X, B \in \mathbb{R}^{n \times m}$ durch die LR-Zerlegung der „erweiterten Matrix“ $\left(A \mid B \right)$ und erhält somit dann L und $\left(R \mid B^* \right)$. Damit ist X bestimmbar durch Rückwärtssubstitution von $RX = B^*$, die Vorwärtssubstitution entfällt hierbei.

Berechnung der Inversen

! Das Berechnen der Inversen mag zwar im Allgemeinen ein Kunstfehler und Entlassungsgrund sein, manchmal möchte man diese jedoch wirklich ohne größere Umstände ausrechnen. Im Normalfall gelingt dies ohne Probleme mit obiger Formel, setzt man für B einfach die Einheitsmatrix ein. X ist dann die inverse Matrix zu A. Aufgrund von sehr sehr starken Rundungsfehlern, welche beim Berechnen der inversen durchaus auftreten können, ist dieser Ansatz allerdings nicht als Zwischenschritt zu empfehlen!

1.1.5 Sonderfälle

1.1.5.1 LR-Zerlegungen für nicht vollbesetzte Matrizen — Bandmatrizen

Bei Bandmatrizen beschränken wir die Elimination auf diejenigen Elemente, welche **innerhalb** des Bandes und **unterhalb** der Diagonalen liegen. Es ergibt sich somit ein Aufwand bei Bandbreite b und Matrixordnung n von $\mathcal{O}(b^2 n)$.
Bei tridiagonalen Matrizen ist die LR-Zerlegung mit $\mathcal{O}(n)$ Operationen lösbar – mit insgesamt jeweils $n-1$ Divisionen, Additionen und Multiplikationen -.

$$r_1 = b_1 \qquad\qquad l_1 = \frac{a_1}{r_1}$$

$$r_i = b_i - l_{i-1} \quad \text{für } 1 \leqslant i \leqslant n \cdot c_{i-1} \qquad\qquad l_i = \frac{a_i}{r_i} \quad \text{für } 1 \leqslant i < n$$

.

Probleme bei dünnbesetzten Matrizen

! Allgemein erhalten wir bei dünnbesetzten Matrizen einen sogenannten „Fill-In“, bei dem bereits vorhandene Nullen in der Matrix durch die Elimination zerstört werden. Dies treibt selbstverständlich neben der Zahl der Operationen auch den Speicherbedarf in die Höhe.
Mittels sog. *„sparse matrix solvers“* soll der Fill-In durch eine geeignete Permutation der Matrix minimiert werden. Gleichzeitig ist das Thema längst noch nicht abgeforscht, insbesonderte die Entwicklung solcher Algorithmen für Parallelrechner ist weiterhin ein wichtiges Thema.

1.1.5.2 Cholesky-Zerlegung für positiv definite Matrizen

Ist A symmetrisch und positiv definit, so kann man A auch faktorisieren in

$$A = L \cdot D \cdot L^T,$$

wobei das L aus der LR-Zerlegung und das D die Diagonalanteile des R darstellt. Aufgrund der positiven Definitheit kann man schlussfolgern, dass D in der Diagonale nur positive Elemente hat, in Alternative lässt sich somit die Faktorisierung als

$$A = L \cdot D \cdot L^T = L \cdot D^{1/2} \cdot D^{1/2} \cdot L^T = \widetilde{L} \cdot \widetilde{L^T}$$

schreiben. Der schlussendliche Algorithmus nutzt dann aus, dass die beiden Dreiecksfaktoren gleich sind, man braucht damit später nur einen der beiden explizit berechnen und spart damit grob die Hälfte an Operationen.
Ein weiterer Vorteil ist das Nichtauftreten von 0-Pivots und die damit einhergehende Stabilität des Verfahrens ohne Pivotsuche, was eine Pivotsuche überflüssig macht.
Der einzige Nachteil ist die starke Einschränkung auf die Matrizenfamilie.

1.2 QR-Zerlegung

Ziel der QR-Zerlegung ist eine Faktorisierung der Matrix A in eine orthogonale Matrix Q und eine rechte obere Dreiecksmatrix R.

1.2.1 Orthogonale Matrizen

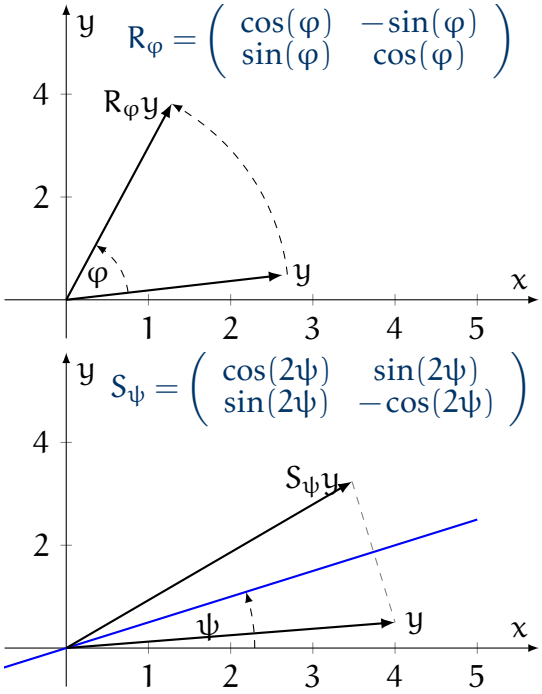
Definition 1.1 (*Orthogonale Matrix*)

Eine Matrix $Q \in \mathbb{R}^{n\times n}$ heißt **orthogonal**, falls eine der folgenden äquivalenten Bedingungen erfüllt ist:

- $Q^TQ = E_n = QQ^T$
- $Q^{-1} = Q^T$
- Die Spalten oder Zeilen von Q bilden eine Orthonormalbasis
- Die Abbildung Q ist winkel- und längentreu.
- Q erhält das Skalarprodukt, es gilt also:

$$Qx \circ Qy = x \circ y$$

Die Determinante einer orthonormalen Matrix ist immer ± 1.



orthogonale Matrizen im zweidimensionalen Fall: Rotationsmatrizen oder Spiegelungen

1.2.2 QR-Zerlegung mittels Givens-Rotationen

Während wir uns Rotationsmatrizen im zweidimensionalen bereits im obigen Beispiel überlegt haben, wollen wir nun den Aufbau einer solchen Rotationsmatrix auf den n-dimensionalen Raum verallgemeinern. Im Allgemeinen betrachten wir dabei allerdings weiterhin bloß endlichdimensionale Räume. Wir definieren die Givensrotationsmatrix an der Achse ij mit dem Winkel φ mit

$$J_{ij}(\varphi) = (g_{kl})_{k,l\leqslant n} = \begin{pmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & c & \cdots & -s & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & s & \cdots & c & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{pmatrix} = \begin{cases} 1 & \text{für } k = l, k \neq i, k \neq j \\ c = \cos(\varphi) & \text{für } k = l = j, k = l = i \\ s = \sin(\varphi) & \text{für } k = j, l = i \\ -s = -\sin(\varphi) & \text{für } k = i, l = j \\ 0 & \text{sonst} \end{cases}.$$

Nutzen der Givensrotationsmatrizen Mit einer Givensrotation J_{ij} kann man – unter dem richtigen Winkel φ – Nullen in eine Matrix an gewissen Stellen einführen. Dabei ist die Rotationsmatrix $J_{ij}(\varphi_{\text{korrekt}})$ für die Einführung der 0 an der Stelle a_{ij} der Matrix A zuständig. c und s berechnen sich dann über

$$c = \frac{\operatorname{sgn}(a_{jj}) \cdot a_{jj}}{\sqrt{a_{jj}^2 + a_{ij}^2}} \quad \text{und} \quad s = -\frac{\operatorname{sgn}(a_{jj}) \cdot a_{ij}}{\sqrt{a_{jj}^2 + a_{ij}^2}}.$$

Für Givensrotationsmatrizen gilt im Allgemeinen:

$$J_{ij}^{-1}(\varphi) = J_{ij}(-\varphi) = J_{ij}^{T}(\varphi)$$

Umsetzung Wichtig bei der Umsetzung der QR-Zerlegung mit Givenrotationsmatrizen ist es eine Reihenfolge zu schaffen, in der einmal geschaffene Nullen *nicht wieder zerstört* werden. Sei $A \in \mathbb{R}^{m\times n}$ und $n^* = \min\{m-1,\ n\}$, so ist eine QR-Zerlegung der Matrix A durch

$$\begin{aligned} A &= \underbrace{J_{2,1}^{T}\cdot\ldots\cdot J_{m,n^*}^{T}}_{=Q}\cdot\underbrace{J_{m,n^*}\cdot\ldots\cdot J_{2,1}\cdot A}_{=R} \\ &= \quad Q \quad\cdot\quad R \end{aligned}$$

gegeben. Dabei ist das Produkt aller Rotationen selbst eine orthogonale $m\times m$-Matrix, was meistens nicht explizit mitberechnet werden muss.

```
1 def givens_rotation(A, i, j):
2     # You will do this in exercise 3 ...
3     pass
4
5
6 def qr_decompose(A):
7     # You will do this in exercise 3 ...
8     pass
```

Eigenschaften der Givensrotation Das Verfahren ist im Vergleich zur LR-Zerlegung auch ohne Pivotsuche numerisch stabil und hat insbesondere eine bessere Konditionierung als die LR-Zerlegung mit

$$\kappa_2(A) = \kappa_2(R) \quad \text{und} \quad \kappa_2(Q) = 1.$$

Ebenfalls ist es einfach auf Matrizen $\in \mathbb{R}^{m\times n}$ und $m > n$ anwendbar, wird also auch oft zur Ausgleichsrechnung verwendet.
Das Verfahren mit Givensrotationen ist per se ebenfalls leichter parallelisierbar als das Verfahren der Householderspiegelungen, aber dafür etwas schwerer zu implementieren.

Aufwandsverbesserung

! Eines der Hauptprobleme der QR-Zerlegung mit Givensrotationsmatrizen ist der deutlich höhere Aufwand im Vergleich zur LR-Zerlegung. (**4-mal so hoch**)
Ein Verbesserungsvorschlag ist es die Rotationen durch Skalierungen so zu modifizieren, dass die Diagonalelemente alle gleich 1 sind. Die so erfolgten Skalierungen werden dann seperat aufmultipliziert, wodurch man dann insgesamt in etwa die Hälfte des Rechenaufwands spart. Dieses Verfahren ist auch unter dem Namen der ***rationalen Givensrotationen*** bekannt.

1.2.3 QR-Zerlegung mittels Householderspiegelungen

Wir betrachten nun eine andere Art der QR-Zerlegung durch Spiegelungen. Wir wollen durch eine solche Spiegelung eine ganze Zeile mit den notwendigen Nullen füllen. Wir überlegen uns dazu zuerst, was eigentlich eine Householdertransformation darstellen soll.

Householdertransformationen Eine Householdertransformation berschreibt im Allgemeinen die Spiegelung eines Vektors an einer Hyperebene durch die Null im euklidischen Raum, welche durch ihren Normalenvektor v mit Einheitslänge $v^T v = \|v\|^2 = 1$ repräsentiert wird.

Die Householdertransformation eines Vektors x findet man mit

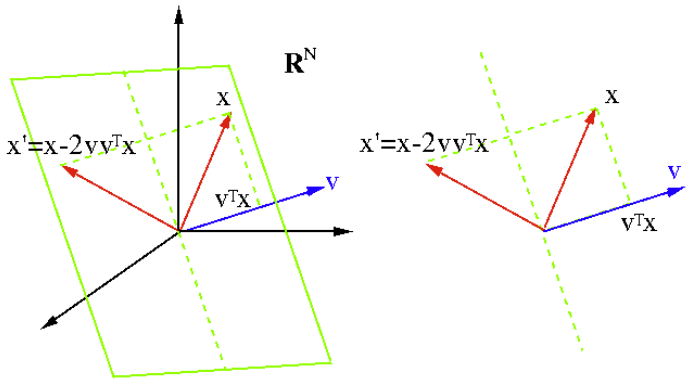
$$x' = x - 2vv^T x,$$

mit Überlegungen stellt man dann fest, dass

$$x' = (E_n - 2vv^T)x.$$

Wir wollen dann die Householdermatrix H definieren als

$$H = E_n - 2vv^T.$$



Wir erkennen, dass H sowohl symmetrisch als auch zu sich selbst invers ist, woraus unmittelbar folgt, dass H orthogonal ist. Da ebenfalls ein jeder Vektor u durch

$$v = \frac{u}{\|u\|}$$

normalisiert werden kann, ist H ebenfalls durch

$$H = E_n - 2 \cdot \frac{uu^T}{\|u\|^2}$$

darstellbar. Mit einem speziell definierten Vektor

$$u = x - \|x\| \cdot e_1,$$

wobei e_1 der erste Standardbasisvektor des $\mathbb{R}^n$ ist, können wir die Spiegelung x' von x einfach angeben mit

$$x' = Hx = \|x\| \cdot e_1.$$

Das heißt, dass durch die Spiegelung alle Positionen bis auf die erste ausgenullt werden. Damit lässt sich leicht eine obere Dreieckmatrix erzeugen.

QR-Zerlegung durch Householdertransformationen

Verfahren 1.2 (*QR-Zerlegung mit Householdertransformationen*)

Sei $A \in \mathbb{R}^{n \times m}$ und $n < m$, so gilt:

(1) Zu Beginn des Verfahrens wird die erste Householderspiegelung H_1 mittels

$$H_1 = E_n - 2 \cdot \frac{uu^T}{\|u\|^2} \quad \text{und} \quad u = x - \|x\| \cdot e_1,$$

wobei sich u durch

$$u = c_1 - \|c_1\| \cdot e_1,$$

wobei c_1 der erste Spaltenvektor der Matrix A ist, darstellen lässt.

(2) Sei $H_{(k-1):1}A = \prod_{i=0}^{k-2} H_{(k-1)-i}$ mit

$$H_{(k-1):1}A = \left(\begin{array}{c|c} R_{k-1} & * \\ \hline 0 & A^{(k-1)} \end{array}\right)$$

die multiplikative Verknüpfung der bereits konstruierten Householderspiegelungen mit der Ursprungsmatrix A. Dann konstruieren wir wie in Schritt (1) eine Householderspiegelung H'_k für die Matrix $A^{(k-1)}$ und erstellen dadurch dann die Householdermatrix H_k mit

$$H_k = \begin{pmatrix} E_{k-1} & 0 \\ 0 & H'_k \end{pmatrix}.$$

(3) Nach $n-1$ Schritten (*falls* $n > m$: n *Schritten*) setzen wir dann R und Q als

$$R = H_{(n-1):1} \cdot A \quad \text{und} \quad Q = H^{-1}_{(n-1):1} = H_{1:(n-1)}.$$

Der Aufwand der QR-Zerlegung durch Householderspiegelungen ist mit $\mathcal{O}(n^3)$ beschrieben, er ist in etwa doppelt so groß wie der einer LR-Zerlegung.

1.2.4 Orthonormalisierung

Orthonormalisierung

! Das Gram-Schmidt-Verfahren (*Standardverfahren der lineraren Algebra für die Orthonomalisierung von linear unabhängigen Vektoren*) ist in der klassischen Variante **numerisch instabil**. Zur Verbesserung nehme man das modifizierte Gram-Schmidt-Verfahren oder aber eine QR-Zerlegung.

Verfahren 1.3 (*QR-Zerlegung mit Householdertransformationen*)

Seien mit b_i $(1 \leqslant i \leqslant k)$ k linear unabhängige Vektoren gegeben und mit $B = \begin{pmatrix} b_1 & \cdots & b_k \end{pmatrix}$ die zugehörige Matrix, deren QR-Zerlegung mit

$$B = Q \cdot R = \begin{pmatrix} q_1 & \cdots & q_k \end{pmatrix} \cdot R$$

bekannt ist.
Dann ist ein System von *orthonormalen* Vektoren mit den q_i gegeben.

1.3 Aufgaben

Wir wollen nunmehr das Stoffgebiet durch einige (*weiterführende*) Aufgaben aufarbeiten.

Aufgabe 1.1

Bestimmen Sie die LR-Zerlegung der folgenden Matrix A_1 mit

$$A_1 = \begin{pmatrix} 2 & -1 & 1 & -1 \\ -2 & 2 & -2 & 2 \\ 2 & -2 & 4 & -3 \\ 0 & 1 & -3 & 5 \end{pmatrix}$$

Aufgabe 1.2

Von der Matrix A_2 ist die QR-Zerlegung bekannt, es gilt

$$A_2 = \begin{pmatrix} -1 & 1 & 1 & 1 \\ 1 & 1 & -1 & 1 \\ -1 & -1 & 2 & 0 \\ 1 & -1 & 0 & 2 \end{pmatrix} = \underbrace{\frac{1}{2} \cdot \begin{pmatrix} -1 & 1 & 1 & 1 \\ 1 & 1 & 1 & -1 \\ -1 & -1 & 1 & -1 \\ 1 & -1 & 1 & 1 \end{pmatrix}}_{=: Q} \cdot \underbrace{\begin{pmatrix} 2 & 0 & -2 & 1 \\ 0 & 2 & -1 & 0 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{=: R}$$

Teilaufgabe 1.2a

Lösen Sie das folgende lineare Gleichungssystem (unter Verwendung der QR-Zerlegung)

$$Ax = b \quad \text{für} \quad b = \begin{pmatrix} 4 & 2 & 2 & 4 \end{pmatrix}^T.$$

Teilaufgabe 1.2b

Bestimmen Sie einen Vektor w, so dass die zugehörige Householderspiegelung H_w bei Anwendung auf A (von Teilaufgabe 1.2a) Nullen in der ersten Spalte erzeugt, das heißt

$$H_w A = \begin{pmatrix} * & * & * & * \\ 0 & * & * & * \\ 0 & * & * & * \\ 0 & * & * & * \end{pmatrix}.$$

Teilaufgabe 1.2c

Berechnen Sie den Betrag der Determinante von A, also $|\det A|$

Aufgabe 1.3

Bestimmen Sie die QR-Zerlegung der Matrix A_3 mit

$$A_3 = \begin{pmatrix} 4 & 1 \\ 3 & 2 \end{pmatrix}$$

Aufgabe 1.4

Gegeben ist eine pentadiagonale $n \times n$-Matrix $A = (a_{ij})$, das heißt $a_{ij} = 0$ falls $|i - j| > 2$.
Für ebendiese Matrix soll mit Hilfe von Givensrotationen die QR-Zerlegung bestimmt werden. Wieviele solcher Givensrotationen werden dazu benötigt?

$$C = \begin{pmatrix} * & * & * & 0 & 0 & & \cdots & \cdots & & 0 \\ * & * & * & * & 0 & & & & & \\ * & * & * & * & * & \ddots & & & & \vdots \\ 0 & * & * & * & * & \ddots & \ddots & & & \vdots \\ 0 & 0 & * & * & * & \ddots & \ddots & \ddots & & \\ & & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & 0 & 0 \\ \vdots & & & \ddots & \ddots & \ddots & * & * & * & 0 \\ \vdots & & & & \ddots & \ddots & * & * & * & * \\ & & & & & 0 & * & * & * & * \\ 0 & & \cdots & \cdots & & 0 & 0 & * & * & * \end{pmatrix}$$

Struktur einer pentadiagonalen Matrix

Aufgabe 1.5

Teilaufgabe 1.5a

Bestimmen Sie die Householdertransformation S, die den Vektor $a = \begin{pmatrix} 4 & 2 & 5 & 2 \end{pmatrix}^T$ auf ein Vielfaches des zweiten Einheitsvektors $e_2 = \begin{pmatrix} 0 & 1 & 0 & 0 \end{pmatrix}^T$ spiegelt.

Teilaufgabe 1.5b

Bestimmen Sie die QR-Zerlegung der Matrix A mit

$$A = \begin{pmatrix} 1 & 0 & 3 \\ 0 & 3/4 & -3 \\ 0 & -1 & -2 \end{pmatrix}$$

mit einer Householdertransformation.

Teilaufgabe 1.5c

Sei dann die partielle Zerlegung Z der Matrix $A = Q_1 Z$ mittels einer Givensrotation Q_1 durch

$$Q_1 = \begin{pmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad \text{und} \quad Z = \frac{1}{5} \cdot \begin{pmatrix} 4 & 8 & 9 \\ 0 & 15 & 20 \\ 3 & 6 & 13 \end{pmatrix}$$

bekannt. Beenden Sie die begonnene QR-Zerlegung mit Hilfe von Givensrotationen.

Aufgabe 1.6

Gegeben sei die Matrix A_5 sowie deren LR-Zerlegung und ein Vektor b

$$A_5 = \begin{pmatrix} 2 & 1 & 1 \\ 6 & 4 & 4 \\ 4 & 1 & 3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 3 & 1 & 0 \\ 2 & -1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 2 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix} \quad \text{und} \quad b = \begin{pmatrix} -1 \\ 2 \\ 1 \end{pmatrix}.$$

Lösen Sie das lineare Gleichungssystem $A_5 x = b$ unter Zuhilfenahme der LR-Zerlegung.

Aufgabe 1.7

Bestimmen Sie die LR-Zerlegung folgender Matrix mit Pivotsuche:

$$A = \begin{pmatrix} 0 & 4 & 2 \\ 4 & 6 & 4 \\ -2 & -1 & 3 \end{pmatrix}$$

Aufgabe 1.8

Bestimmen Sie die Komplexität der folgenden Operationen. Sie dürfen annehmen, dass die Länge der Spaltenvektoren n und die Größe von Matrizen $n \times n$ ist.

Szenario	**Komplexität**
Bestimmung der LR-Zerlegung einer k-diagonalen Matrix	$\mathcal{O}\left(\quad\right)$
Bestimmung der LR-Zerlegung einer 6-diagonalen Matrix	$\mathcal{O}\left(\quad\right)$
Lösen von $Rx = b$ für untere Dreiecksmatrizen R	$\mathcal{O}\left(\quad\right)$
Lösen von $Ax = b$ für vollbesetzte symmetrische Matrizen A	$\mathcal{O}\left(\quad\right)$
Bestimmung der QR-Zerlegung einer vollbesetzten Matrix	$\mathcal{O}\left(\quad\right)$
Komplexität des Lösen des LGSs $Ax = b$ mittels Gauß-Elimination	$\mathcal{O}\left(\quad\right)$
Sei A eine nichtsinguläre Matrix und b ein Vektor. Welche Komplexität hat dann die Bestimmung der LR-Zerlegung $A = PLR$?	$\mathcal{O}\left(\quad\right)$
Komplexität des Lösens eines Gleichungssystems, wenn die LR-Zerlegung bereits vorliegt?	$\mathcal{O}\left(\quad\right)$
Wie viele Jacobirotationen benötigt die Bestimmung der QR-Zerlegung für eine tridiagonale Matrix?	
Bestimmung der LR-Zerlegung einer vollbesetzten Matrix	$\mathcal{O}\left(\quad\right)$
Komplexität des Bestimmens der Determinante bei bereits gegebener QR-Zerlegung	$\mathcal{O}\left(\quad\right)$

Szenario	Komplexität
Komplexität des Bestimmens der Determinante bei bereits gegebener LR-Zerlegung	$\mathcal{O}\left(\quad\right)$
Komplexität des Bestimmens des Betrags der Determinante bei bereits gegebener QR-Zerlegung	$\mathcal{O}\left(\quad\right)$
Welche Komplexität hat das Lösen von $Ax = b$ bei einer k-diagonalen Matrix A?	$\mathcal{O}\left(\quad\right)$

Aufgabe 1.9

Teilaufgabe 1.9a

Welche der folgenden Verfahren können zum Lösen von linearen Gleichungssystemen verwendet werden?

ja	nein	
☐	☐	Gauß-Seidel-Verfahren
☐	☐	Quicksort
☐	☐	cg-Verfahren
☐	☐	QR-Zerlegung
☐	☐	Romberg-Verfahren

Teilaufgabe 1.9b

Welche der folgenden Methoden ist zum Invertieren von Matrizen sinnvoll?

ja	nein	
☐	☐	LR-Zerlegung
☐	☐	Newton-Cotes-Regel
☐	☐	SVD
☐	☐	Dijkstra-Algorithmus
☐	☐	Ich versuche das Invertieren von Matrizen wenn möglich zu vermeiden.

Aufgabe 1.10

Das Modul `Solver` stellt grundlegende Funktionen zum Lösen von linearen Gleichungssystemem bereit.Dabei sollen Sie die entsprechenden Methoden implementieren.

Teilaufgabe 1.10a

Implementieren Sie die Funktion `decomposeLU(M)`. Diese soll eine LR-Zerlegung einer übergebenen Matrix M (dargestellt als NumPy-Array) berechnen und die linke untere Dreiecksmatrix L und rechte obere Dreiecksmatrix R der LR-Zerlegung zurückgeben. **Hinweis:** *Sie sollen explizit überprüfen, ob eine LR-Zerlegung der übergebenen Matrix überhaupt möglich ist!*

Teilaufgabe 1.10b

Implementieren Sie nun die Funktion `forward_substitute(L, b)`, die eine untere Dreiecksmatrix L (als $n \times n$ NumPy-Array) imd einen n-Vektor b (als $n \times 1$ NumPy-Array) übergeben bekommt und das Gleichungssystem $Ly = b$ nach y löst und diesen zurückgibt. *Nutzen Sie dabei die besondere Struktur der Matrix* L.

Teilaufgabe 1.10c

Implementieren Sie nun die Funktion solveSystem(M, B). Diese soll das Gleichungssystem $M \cdot X = B$ noch X lösen. Verwenden Sie dazu bereits implementierte Methoden des Moduls. *Sie können davon ausgehen, dass das Modul `Solver` bereits die Funktion `backward_substitute(R, y)` kennt und die übergebenen Matrizen zueinander kompatibel sind.*

Teilaufgabe 1.10d

Implementieren Sie nun die Funktion calcDeterminant(M). Diese soll die Determinante der übergebenen Matrix M berechnen und zurückgeben. Verwenden Sie dazu bereits implementierte Methoden des Moduls.

Aufgabe 1.11

Wir betrachten nun noch einmal das Modul `Solver`, welches nun für die QR-Zerlegung erweitert werden soll. In dieser Aufgabe können Sie der Einfachheit halber davon ausgehen, dass die übergebenen Parameter **immer** kompatibel sind, Sie brauchen also *keine Fehlerbehandlung*.

Teilaufgabe 1.11a

Implementieren Sie die Funktion givens_rotation(A, i, j). Diese soll die Givensrotationsmatrix J zur Matrix A erstellen, so dass $J \cdot A$ an der Stelle (i, j) eine 0 besitzt.

Teilaufgabe 1.11b

Implementieren Sie anschließend die Funktion householder(A, j), welche die Householderspiegelungsmatrix einer übergebenen Matrix A in Spalte j berechnent und zurückgibt.

Teilaufgabe 1.11c

Implementieren Sie anschließend die Funktion decomposeQR(A), welche die QR-Zerlegung einer übergebenen Matrix A berechnent und zurückgibt. Zum Erzeugen der Nullen sollen Sie dabei Givensrotationsmatrizen verwenden.

Teilaufgabe 1.11d

Implementieren Sie nun die Funktion backward_substitute(R, y), die eine obere Dreiecksmatrix R (als $n \times n$ NumPy-Array) imd einen n-Vektor y (als $n \times 1$ NumPy-Array) übergeben bekommt und das Gleichungssystem $Rx = y$ nach x löst und diesen zurückgibt. *Nutzen Sie dabei die besondere Struktur der Matrix* R.

KAPITEL 2

Lineare Ausgleichsprobleme

Definition 2.1 (*Über- und unterbestimmte lineare Gleichungssysteme*)

Sei $A \in \mathbb{R}^{n \times m}, b \in \mathbb{R}^m$, so heißt das Gleichungssystem ...

- ... **unterbestimmt**, falls das Gleichungssystem ***weniger*** *Gleichungen als Unbekannte* hat.
- ... **überbestimmt**, falls das Gleichungssystem ***mehr*** *Gleichungen als Unbekannte* hat.

Dabei gehen linear abhängige Zeilen nur **einfach** in die Wertung als Gleichung mit ein.

Man stellt schnell fest, dass überbestimmte Gleichungenssysteme im Allgemeinen **keine Lösung** haben, weil sich einzelne Gleichungen widersprechen könnten. Vor allem bei aus mehrfachen Messwerten generierten Gleichungssystemen kommt dies wegen unvermeidlicher Ungenauigkeiten relativ häufig vor.
Wenn das Gleichungssystem nicht lösbar ist, so ist das Residuum $r(x) = b - Ax$ nichtverschwindend für allgemeine x, naheliegend ist deswegen die Suche nach einem $\arg\min r(x)$. Wir betrachten deswegen folgendes Problem:

Problem 2.1 (*Lineares Ausgleichsproblem*)

Gegeben seien $A \in \mathbb{R}^{n \times m}$ und $b \in \mathbb{R}^m$. Gesucht ist ein $x \in \mathbb{R}^n$, so dass das Residuum

$$r(x) = b - Ax$$

minimiert wird.
Wir suchen also ein $\widehat{x}$ mit

$$\widehat{x} = \arg\min_{x \in \mathbb{R}^n} r(x).$$

Verfahren 2.1 (*Methode der „kleinsten Quadrate“*)

Wir lösen das lineare Ausgleichsproblem, indem wir das Residuum $r(x)$ bzgl. der euklidischen Norm minimieren. Das heißt das lineare Ausgleichsproblem wird umformuliert zu:
Gegeben seien $A \in \mathbb{R}^{n \times m}$ und $b \in \mathbb{R}^m$. Gesucht ist ein $x \in \mathbb{R}^n$, so dass die Norm des Residuums

$$\|r(x)\|_2 = \|b - Ax\|_2 = \|Ax - b\|_2$$

minimiert wird.

Das Ausgleichsproblem heißt linear, weil der Parameter x nur linear in den Defekt $Ax - b$ eingeht. Während das Residuum immer ein Minimum hat, ist der Minimierer x nur dann eindeutig bestimmt, wenn $\operatorname{rang}(A) = m$, also die Spalten von A linear unabhängig sind.

Lösen mittels QR-Zerlegungen Ist die QR-Zerlegung bereits bekannt, so gilt:

Satz 2.1 (*Lineare Ausgleichsprobleme bei bekannter QR-Zerlegung*)

Ist die QR-Zerlegung einer Matrix $A \in \mathbb{R}^{n\times m}$ bekannt, so ist zum Verfahren der „kleinsten Quadrate“ äquivalent:

$$\arg\min \|Ax-b\|_2 = \arg\min \left\|Rx - Q^T b\right\|_2$$

Bei der QR-Zerlegung ist das Beste, das man erreichen kann, die oberen m Gleichungen zu erfüllen, der Rest spielt dabei dann „keine Rolle“ mehr. Man löst dann das „verkleinerte System“ per Rückwärtssubstitution. Der Rest bestimmt das Residuum, es verschwindet für kein x, ist aber zeitgleich ein Maßstab für die Erfüllbarkeit des Systems.

Normalengleichung

Satz 2.2 (*Lineare Ausgleichsprobleme bei bekannter QR-Zerlegung*)

Problem 2.1 ist äquivalent zum folgenden Optimierungsproblem.
Bestimme das Minimum des Funktionals

$$\P : x \mapsto \|b-Ax\|_2^2 = \sum_{i=1}^{n}\left(b_i - \sum_{j=1}^{m} a_{ij}x_j\right)^2.$$

Mathematisch gesehen ist ein Lösungsverfahren alle partiellen Ableitungen der Null gleichzusetzen, wir erhalten somit dann die Normalengleichung des linearen Ausgleichsproblems mit

$$A^T Ax = A^T b,$$

dies entspricht einem quadratischen $m\times m$-Gleichungssystem, wobei $A^T A$ eine symmetrische $m\times m$-Matrix ist, die positiv definit und invertierbar ist, sofern A vollen Rang hat ($\operatorname{rang}(A) = m$). Zu Lösen ist dieses Gleichungssystem dann über Verfahren wie Cholesky.

Ausgleichsgerade Gegeben sei eine Reihe von Messwerten (x_i, y_i) mit $1 \leqslant i \leqslant n$, deren Ausgleichsgerade $y = a_1 \cdot x + a_0$ gesucht ist. Wir wissen, dass die Gleichung

$$\begin{pmatrix} 1 & x_1 \\ \vdots & \vdots \\ 1 & x_n \end{pmatrix} \cdot \begin{pmatrix} a_0 \\ a_1 \end{pmatrix} = \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix}$$

ideal ist. Mit der Normalengleichung ergibt sich dann als Lösung:

$$A^T A = \begin{pmatrix} n & \sum_{i=1}^{n} x_i \\ \sum_{i=1}^{n} x_i & \sum_{i=1}^{n} x_i^2 \end{pmatrix} = n \cdot \begin{pmatrix} 1 & \overline{X} \\ \overline{X} & \overline{X^2} \end{pmatrix}$$

$$A^T b = n \cdot \begin{pmatrix} \overline{Y} \\ \overline{XY} \end{pmatrix}.$$

Mittels Cramer’scher Regel folgt dann die Lösung als

$$a_0 = \frac{\overline{Y}\cdot\overline{X^2} - \overline{X}\cdot\overline{XY}}{\overline{X^2} - \overline{X}^2} \quad \text{und} \quad a_1 = \frac{\overline{Y}\cdot\overline{X^2} - \overline{X}\cdot\overline{XY}}{\overline{X^2} - \overline{X}^2}.$$

Anwendung 1 – Bestimmung einer Ausgleichsfunktion

Problem 2.2 (*Bestimmung einer Ausgleichsfunktion*)

Gegeben seien n Messwerte (x_i, y_i). Suche dann unter den Polynomen $f_a(x) = \sum_{i=0}^{m} a_i \cdot x^i$ – mit $a \in \mathbb{R}^m$ – dasjenige, welches zu den Messwerten am besten passt.

Anwendung 2 – Bestimmung einer Ausgleichsebene

Problem 2.3 (*Bestimmung einer Ausgleichsebene*)

Gegeben seien n Messwerte (x_i, y_i) mit jeweiligen Messwerten z_i. Suche dann unter den Ebenen $z = a + bx + cy$, welche die Messdaten am besten *im quadratischen Mittel* approximiert.

Aufgaben

Aufgabe 2.1

Gegeben sei die QR-Zerlegung der Matrix A_4 mit

$$A_4 = \underbrace{\frac{1}{3} \cdot \begin{pmatrix} 2 & 0 & 1 & -2 \\ 2 & 0 & -2 & 1 \\ 1 & 0 & 2 & 2 \\ 0 & 3 & 0 & 0 \end{pmatrix}}_{=: Q_4} \cdot \underbrace{\begin{pmatrix} 3 & -1 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}}_{=: R_4}.$$

Bestimmen Sie die Lösung des linearen Ausgleichsproblem der kleinsten Quadrate ($\min_x \|A_4 x - b\|_2$) mit Hilfe der QR-Zerlegung für $b = \begin{pmatrix} 2 & -2 & 3 & -1 \end{pmatrix}^T$.

Aufgabe 2.2

Teilaufgabe 2.2a

Wie lautet die Normalengleichung des linearen Ausgleichsproblems?

Teilaufgabe 2.2b

Gegeben sind die Datenpunkte x_i und zugehörige Datenwerte y_i mit

x_i	-1	0	2	3
y_i	0	0	2	2

Bestimmen Sie die Gerade $y = m \cdot x + t$, die diese Daten im *least squares* Sinne am besten approximiert.

Aufgabe 2.3

Gegeben seien die folgenden 2D-Datenpunkte $p_i = \begin{pmatrix} x_i & y_i \end{pmatrix}^T$:

$$p_0 = \begin{pmatrix} -1 \\ 1 \end{pmatrix}, p_1 = \begin{pmatrix} 0 \\ -2 \end{pmatrix}, p_2 = \begin{pmatrix} 1 \\ -1 \end{pmatrix}, p_3 = \begin{pmatrix} 2 \\ 2 \end{pmatrix}$$

Teilaufgabe 2.3a

Bestimmen Sie das lineare überbestimmte Gliehcungssystem $Au = b$, welches die Bedingungen $f(x_i) = y_i$ für eine quadratische Funktion $f(x) = ax^2 + bx + c$ repräsentiert. Geben Sie dazu A, b und u explizit an.

Teilaufgabe 2.3b

Nennen Sie ein weiteres Verfahren neben der Gaußchen Normalengleichung um überbestimmte lineare Gleichungssysteme im *least squares* Sinn zu lösen.

Aufgabe 2.4

Bestimmen Sie ob folgende Gleichungssysteme über-, unterbestimmt oder eindeutig lösbar sind:

unterbestimmt	eindeutig lösbar	überbestimmt	
☐	☐	☐	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{pmatrix} \cdot x = \begin{pmatrix} 3 \\ 2 \\ 1 \end{pmatrix}$
☐	☐	☐	$\begin{pmatrix} 1 & 2 \\ 0 & 2 \\ 0 & 3 \end{pmatrix} \cdot x = \begin{pmatrix} 3 \\ 2 \\ 1 \end{pmatrix}$
☐	☐	☐	$\begin{pmatrix} 1 & 2 & 0 \\ 0 & 2 & 3 \\ 2 & 4 & 0 \end{pmatrix} \cdot x = \begin{pmatrix} 2 \\ 1 \\ 4 \end{pmatrix}$
☐	☐	☐	$\begin{pmatrix} 1 & 2 \\ 0 & 2 \\ 2 & 4 \end{pmatrix} \cdot x = \begin{pmatrix} 1 \\ 4 \\ 2 \end{pmatrix}$
☐	☐	☐	$\begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot x = \begin{pmatrix} 3 \\ 2 \\ 1 \end{pmatrix}$
☐	☐	☐	$\begin{pmatrix} 1 & 3 \\ 2 & 0 \\ 1 & 0 \end{pmatrix} \cdot x = \begin{pmatrix} 2 \\ 3 \\ 4 \end{pmatrix}$

KAPITEL 3

Matrizen - Datenstrukturen und praktische Verfahren

Wir überlegen uns nun wie man besonders *dünn besetzte Matrizen* möglichst effizient speichert. Dazu wollen wir möglichst **keine Nullwerte** speichern. Bezeichne nun im Folgenden A eine dünnbesetzte $m \times n$-Matrix auf den reellen Zahlen. Wir definieren dann die Anzahl an Nichtnullwerten einer Matrix A als

$$nn_A = \#\{x \mid \exists 1 \leqslant i \leqslant m, 1 \leqslant j \leqslant n.\ x = A_{ij} \wedge x \neq 0\},$$

die Anzahl an Nichtnullwerten einer Zeile z der Matrix A als

$$nnr_z = \#\{x \mid \exists 1 \leqslant j \leqslant n.\ x = A_{zj} \wedge x \neq 0\}$$

und die Anzahl an Nichtnullwerten einer Spalte s der Matrix A als

$$nnc_s = \#\{x \mid \exists 1 \leqslant i \leqslant m.\ x = A_{is} \wedge x \neq 0\}.$$

! Im Folgenden gilt, dass die Indizierung im Allgemeinen bei 1 beginnt.

3.1 Row And Column Storage (RaCS)

Wir speichern in einem nn_A großen Feld val die Nichtnullwerte der Matrix A und in den Feldern col_ind und row_ind die zugehörigen Spalten- respektive Zeilenindizes. Allgemein gilt also:

$$A_{ij} = \mathsf{val}_k \quad \Leftrightarrow \quad \mathsf{row\_ind}_k = i \wedge \mathsf{col\_ind}_k = j.$$

Der Speicheraufwand beträgt nur noch $3 \cdot nn_A << m \cdot n$, da $nn_A << m \cdot n$.

3.2 Compressed Row Storage (CRS)

Wir verbessern nun das obige Verfahren, indem wir statt row_ind nur noch Pointer speichern, die die jeweiligen Zeilenanfänge markieren. Statt row_ind gibt es dann ein Feld row_ptr, welches die Zeiger enthält. Die Elemente aus row_ptr zeigen dann auf die Elemente der anderen Arrays. Allgemein gilt hier also:

$$A_{ij} = \mathsf{val}_k \quad \Leftrightarrow \quad (\mathsf{row\_ptr}_i \leqslant k < \mathsf{row\_ptr}_{i+1}) \wedge \mathsf{col\_ind}_k = j.$$

Wie signalisiert man das Ende? Der letzte Eintrag in row_ptr ist immer $nn_A + 1$.

Wie signalisiert man leere Zeilen? Die leeren Zeilen machen kein Problem mit der induktiven Definition des Feldes row_ptr:

- $\mathsf{row\_ptr}_1 = 1$
- $\mathsf{row\_ptr}_i = \mathsf{row\_ptr}_{i-1} + nnr_{i-1}$

Vorteile Der Speicheraufwand beläuft sich auf $2 \cdot nn_A + m + 1$. Ebenfalls erlaubt das CRS einen besonders effizienten Zeilenzugriff, sowie eine besonders effiziente Matrix-Vektor-Multiplikation:

```
def matMulVek_CRS(val, col_ind, row_ptr, x):
    n = x.shape()[0]
    y = np.zeros((n,1))
    for i in range(0, n):
        y[i] = 0 #Eigentlich irrelevant, da np.zeros oben
        for j in range(row_ptr[i], row_ptr[i+1]):
            y[i] = y[i] + val[j] * x[col_ind[j]]
```

Mit CRS gestaltet sich allerdings das Einfügen und Löschen von Werten als besonders trickreiche Aufgabe. Die Arrays müssten dafür nämlich umstrukturiert werden, eventuell könnte hier eine Implementierung mit verketteten Listen Abhilfe schaffen, allerdings ist das dann mit einem höheren Suchaufwand verbunden.
Ebenfalls gibt es keinen wahlfreien Zugriff auf einzelne Elemente, denn obwohl sich der Zeilenzugriff als besonders effizient herausstellt, ist der Spaltenzugriff sehr ineffizient zu gestalten. Man sucht deswegen auf sortiertem Teilarray in der jeweiligen Zeile, was dann im besten Fall logarithmischen Aufwand mit sich zieht (*binary search*).

Block Compressed Row Storage (BCRS) Als Unterart der CRS gibt es die BCRS-Methodik, bei der dicht besetzte Gebiete in einzelnen Blöcken zusammengefasst werden. Jeder nichtnull Block wird dann intern normal gespeichert, da bei dicht besetzten Blöcken eine normale Speicherung effizienter ist. Diese Blöcke werden dann als Elemente für die CRS verwendet, womit dann auch null-Gebiete nicht mehr gespeichert werden.
Vor allem für große Blockgrößen ist das BCRS deutlich effizienter als das reine CRS.

3.3 Compressed Column Storage (CCS)

Ein Manko, welches wir erkennen können, ist, dass vor allem für ein $m >> n$ die CRS ineffizienter wäre, als wenn man über die Zeilen adressiert. Wir definieren deswegen das CCS-Verfahren analog zu CRS, erhalten damit dieselben Vor- wie auch Nachteile. Statt dem row_ptr gibt es jetzt damit dann ein Feld col_ptr. Allgemein gilt dann also:

$$A_{ij} = \mathrm{val}_k \quad \Leftrightarrow \quad (\mathrm{col\_ptr}_j \leqslant k < \mathrm{col\_ptr}_{j+1}) \wedge \mathrm{row\_ind}_k = i.$$

Wie signalisiert man das Ende? Der letzte Eintrag in col_ptr ist immer $nn_A + 1$.

Wie signalisiert man leere Spalten? Die leeren Spalten machen kein Problem mit der induktiven Definition des Feldes col_ptr:

- $\mathrm{col\_ptr}_1 = 1$
- $\mathrm{col\_ptr}_i = \mathrm{col\_ptr}_{i-1} + nnc_{i-1}$

3.4 Matrizen und Graphen

Im Allgemeinen kann die Besetzungsstruktur einer allgemeinen $n \times n$-Matrix auch als Graph dargestellt werden. Sei $A \in \mathbb{R}^{n \times n}$, so definieren wir mit

$$V = \{P_i \mid 1 \leqslant i \leqslant n\} \quad \text{und} \quad E = \{(P_i, P_j) \mid a_{ij} \neq 0\}$$

einen **gerichteten** Graphen $G = (V, E)$. Per Zusammenhangsanalyse des Graphen kann man dann feststellen, ob sich das Gleichungssystem in mehrere **unabhängige Teilprobleme** unterteilen lässt, welche sich dann seperat lösen lassen.

3.5 Blockmatrizen

Sei M eine Matrix der Größe $m \times n$. Wir zerlegen nun die Zahl der Spalten und Zeilen ganzzahlig in $m = \sum_{i=1}^{q} m_i$ und $n = \sum_{i=1}^{r} n_i$, wobei q und r die Anzahl an Summanden beschreiben. Dann lässt sich M darstellen als

$$M = \begin{pmatrix} M_{11} & \cdots & M_{1r} \\ \vdots & \ddots & \vdots \\ M_{q1} & \cdots & M_{qr} \end{pmatrix}, \text{ mit } M_{ij} \in \mathbb{R}^{m_i \times n_j}.$$

Gründe für Blockmatrizen

Besonders gerne wird eine Matrix in Blockmatrizen zerlegt, wenn ein Problem *nicht mehr komplett in den Speicher passt*. Man passt dann die Blockgröße an den verfügbaren Speicher an, ein Analogon findet sich in der Optimierung für Cachearchitekturen.

Ein weiterer Grund für Blockmatrizen ist die *effizientere Bereitstellung der Daten*, man passt dabei dann die Blockgröße dem gegebenen Cache an, was das *Nutzen von Pipelines* erlaubt. Besonders bei GPGPUs ist diese Herangehensweise entscheidend, man erhält somit bis zu mehreren Größenordnungen an Performancegewinn.

3.6 Aufgaben

Aufgabe 3.1

Speichern Sie die folgende Matrix A im CRS-Format ab. Die Indizierung beginnt dabei bei 1.

$$A = \begin{pmatrix} 1 & 2 & 7 & 0 \\ -2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 4 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Aufgabe 3.2

Sei ein Vektor $b = \begin{pmatrix} 1 & 1 & -1 & 1 & 0 \end{pmatrix}^T$ und eine Matrix B im CCS-Format mit

```
    val = [1, 4, 7, -2, 5]
col_ind = [2, 3, 4,  2, 3]
row_ptr = [1, 1, 2,  4, 6]
```

gegeben. Bestimmen Sie den Vektor $b^T B$ ohne die Matrix zu rekonstruieren. *Hinweis: Zwischenschritte müssen erkennbar sein! Die Indizierung soll bei 1 beginnen.*

Aufgabe 3.3

Hinweis: In dieser Aufgabe gilt stets, dass die Indizierung bei 0 beginnen soll. **Teilaufgabe 3.3a**

Wie ergibt sich die Dimension einer $m \times n$ Matrix aus dem CRS-Format?

Teilaufgabe 3.3b

Wie ergibt sich die Dimension einer $m \times n$ Matrix aus dem CCS-Format?

Teilaufgabe 3.3c

Wie kann man eine $n \times n$-Tridiagonalmatrix möglichst effizient speichern?

Teilaufgabe 3.3d

Sei eine Matrix A im CCS-Format gegeben. Wie lässt sich die Transponierte von A unter Ausnutzung der Struktur möglichst effizient berechnen und abspeichern?

Aufgabe 3.4

Sei nun eine Matrix M mit Zeilenanzahl 4 im CCS-Format

$$\begin{aligned} \texttt{val} &= [4,\ 3,\ 7,\ -2,\ 8,\ 4] \\ \texttt{col\_ind} &= [1,\ 2,\ 3,\ 1,\ 2,\ 3] \\ \texttt{row\_ptr} &= [0,\ 0,\ 1,\ 3,\ 5,\ 6] \end{aligned}$$

gegeben. Rekonstruieren Sie die Matrix. Die Indizierung beginnt weiterhin bei 0.

Aufgabe 3.5

Im Folgenden soll je eine Klasse für die CCS/CRS-Matrixspeicherformen erstellt werden. Fehlerbehandlungen sind **nicht erforderlich**! Für beide Klassen sollten folgende Funktionen implementiert werden:

`__init__(self, other)` – **Konstruktor** Der Konstruktor soll eine gegebene Matrix `other` (dargestellt als $m \times n$ NumPy-Array) in das jeweilige Format übertragen. Die zu füllenden Instanzvariablen sind:

- `_rowPtr` oder `_colPtr`
- `_colIndices` oder `_rowIndices`
- `_values`
- `_width` und `_height`: Die Breite und Höhe der Matrix
- `_nonZeroElements`: Die Anzahl an Elementen, die nicht 0 sind.

`toMatrix(self)` Erstellt aus einer im jeweiligen Format gespeicherten Matrix wieder eine „normale“ Matrix (Darzustelen als NumPy-Array).

`getEntry(self, i, j)` Bestimmt den Eintrag der Matrix in Zeile `i` und Spalte `j`

`mul(self, other)` Multipliziert eine Matrix im gegebenen Format mit einer anderen „normalen“ Matrix. *Achtung: Bei CRS ist `self @ other`, bei CCS `other @ self` zu berechnen. Sie können davon ausgehen, dass die Ihnen übergebenen Matrizen zueinander kompatibel sind!*

`transpose(self)` Berechnet das Transponierte der Matrix. Sie können hier auch die Klassenmethode `fromFinal(cls, hoehe, breite, values, rowIndices, colPtr)` oder `fromFinal(cls, hoehe, breite, values, colIndices, rowPtr)` verwenden. Diese erstellt eine Matrix mit den gegebenen Einträgen.

`toCCS(self)` **und** `toCRS(self)` Wandelt eine gegebene Matrix in das jeweils komplementäre Schema um. ***Achtung**: Es ist **nicht gewünscht** die Matrix in irgendeinem Zwischenschritt zu rekonsturieren.*

KAPITEL 4

DISKRETISIERUNG UND QUANTISIERUNG

4.1 Diskretisierung

4.1.1 Faltung

Zielsetzung ist die Beantwortung der Frage, wie sich ein Signal bei der Übertragung über ein System verhält. Hierzu werden einfache, in ihrem Verlauf vollständig bekannte Signale angegeben und der zeitliche Verlauf der Ausgangssignale berechnet.
Seien also Funktionen $f, g : D \subseteq \mathbb{R}^n \to \mathbb{C}$ gegeben, so definieren wir die Faltung als

$$(f * g)(x) := \int_D f(\tau) \cdot g(x - \tau) d\tau,$$

wobei das Integral auf der rechten Seite für fast alle x wohldefiniert sein soll. Im besonderen Fall, dass $f, g \in \mathcal{L}^1(\mathbb{R}^n)$ – womit f und g integrierbare Funktionen mit endlichem uneigentlichem Betragsintegral sind – ist diese Bedingung immer erfüllt.

Faltung von periodischen Funktionen Seien f, g *periodische* Funktionen der Periode T **einer Veränderlichen**, so definiert man die Faltung von f und g als

$$(f * g)(x) := \frac{1}{T} \int_a^{a+T} f(\tau) \cdot g(x - \tau) d\tau,$$

wobei $f * g$ wieder eine T-periodische Funktion und die Faltung auf einem beliebigen Intervall mit Periodenlänge T definiert ist.

Diskrete Faltung Seien $f, g \colon D \to \mathbb{C}$ Funktionen mit dem diskreten Definitionsbereich $D \subseteq \mathbb{Z}$, so ist die diskrete Faltung von f und g definiert als

$$(f * g)(n) = \sum_{k \in D} f(k) g(n - k).$$

Rechengesetze Die Menge $\mathcal{L}^1(R^n)$ bildet zusammen mit der Addition und Faltung einen kommutativen Ring. En detail heißt das also:

- Kommutativität — $f * g = g * f$
- Assoziativität — $f * (g * h) = (f * g) * h$
- Distributivität — $f * (g + h) = (f * g) + (f * h)$
- Assoziativität mit der skalaren Multiplikation — $\lambda(f * g) = (\lambda f) * g = f * (\lambda g)$

Glättungsfilter Tiefpass-, Bartlettfilter oder die diskrete Gaussfunktion

Kantendetektion Solbeloperator

Separabilität Ein Filter heißt **separierbar** genau dann, wenn seine Matrixdarstellung – die *Filtermaske* — den Rang 1 hat.

4.1.2 Fourier-Transformation

Die Fouriertransformatiion baut auf der harmonischen Schwingungslehre in Zusammenhang mit einem mathematischen Theorem auf.
Im Allgemeinen werden harmonische Schwingungen mittels

$$\exp(i \cdot kx) = \cos(kx) + i \cdot \sin(kx)$$

dargestellt, wobei k die Wellenzahl, $\lambda = \frac{2\pi}{k}$ die Wellenlänge und $\nu = \frac{k}{2\pi}$ die Frequenz ist.

Satz 4.1 (*Superpositionsdarstellung*)

Eine jede (quadratisch integrierbare) Funktion $f \in \mathrm{Abb}(\mathbb{R}, \mathbb{C})$ ist durch Superposition von harmonischen Schwingungen darstellbar:

$$f(x) = \frac{1}{(2\pi)^n} \int_{\mathbb{R}^n} (\mathcal{F}f)(y)\, e^{iy \cdot x}\, dy\,.$$

Die Funktion $\mathcal{F}f$ heißt **Spektralfunktion** oder auch **Fouriertransformierte** von f und es gilt ebenso:

$$(\mathcal{F}f)(y) = \frac{1}{(2\pi)^{n/2}} \int_{\mathbb{R}^n} f(x)\, e^{-iy \cdot x}\, dx.$$

Satz 4.2 (*Faltungssatz*)

Seien f, g faltbare Funktionen, so gilt

$$\mathcal{F}(f * g) = \sqrt{2\pi} \cdot \mathcal{F}f \cdot \mathcal{F}g$$

beziehungsweise

$$\mathcal{F}f * \mathcal{F}g = \sqrt{2\pi} \cdot \mathcal{F}(f \cdot g)$$

4.1.3 Abtasttheorem

Satz 4.3 (*Abtasttheorem*)

Ist das Spektrum $\widetilde{g}(k)$ einer kontinuierlichen Funktion $g(x)$ bandbegrenzt, sprich gilt

$$\widetilde{g}(k) = 0 \quad \textit{für} \quad |k| \geqslant k_{\max},$$

und wählt man eine Schrittweite Δx mit $\Delta x \cdot k_{\max} \leqslant \pi$, so kann $g(x)$ aus den Abtastwerten $(g(j \cdot \Delta x))_{j \in \mathbb{Z}}$ ***exakt rekonstruiert werden****.*

Grobe Faustregel: Abtastrate ist die doppelte maximale Frequenz.

4.1.4 Aliasing und Anti-Aliasing

Aliasing tritt immer dann auf, wenn das zu messende Signal **hochfrequente Anteile** enthält, welche gemäß Abtasttheorem *nicht reproduzierbar* sind. Durch Unterabtasten der hohen Frequenzen wird dann eine nicht vorhandene, niedere Frequenz vorgetäuscht.

Beim Antialiasing *entfernt* man nun **vor dem Abtasten** die höheren Frequenzen im Frequenzraum durch eine Multiplikation mit einem Tiefpass, im Ortsraum durch eine Faltung mit der sinc-Funktion.

4.2 Quantisierung

Ziel ist es ein $r \in \mathbb{R}$ mit $r_{min} \leqslant r \leqslant r_{max}$ im Wertebereich $W := [r_{min}, r_{max}]$ zu approximieren. Haben wir *nahezu* ***gleich****verteilte* Punkte, so können wir das Intervall in *äquidistante* Teilintervalle aufteilen und jeweils deren *Mittelpunkt* als Repräsentanten nehmen. (UNIFORME QUANTISIERUNG)
Sind die Punkte ungleichverteilt, so gibt es unterschiedliche Strategien von Maschinenzahlen bis zum Median-Cut-Verfahren, man spricht von einer *nicht uniformen Quantisierung*.

4.2.1 Gleitpunktzahlen und Maschinenzahlen

Die Menge der **normalisierten** t**-stelligen Gleitpunktzahlen** $FP_{B,t}$ zur Basis B sind definiert als

$$FP_{B,t} = \left\{ \pm M \cdot B^E \;\middle|\; M, E \in \mathbb{Z}, E \text{ beliebig}, B^{t-1} \leqslant M < B^t \text{ oder } M = 0 \right\}.$$

M beschreibt hierbei die Mantisse, E den Exponenten. Die Normalisierung der Zahlen erzwingt man durch die Bedingung $B^{t-1} \leqslant M$.
Die Menge der **Maschinenzahlen** ist dann definiert als

$$MFP_{B,t,\alpha,\beta} = \left\{ g \in FP_{B,t} \;\middle|\; \alpha \leqslant E \leqslant \beta \right\}.$$

Dabei sind B, t, α und β durch die jeweilige Implementierung festgelegt und werden **niemals** explizit gespeichert. Für jede Maschinenzahl wird dann Vorzeichen, Mantisse und Exponent gespeichert, heutzutage vorzugsweise im IEEE-754-Format.
Die Gleitpunktzahlen sind dann die Repräsentanten, die Quantisierung erfolgt dann durch das Runden. Aus den vorherigen Semestern sollte ebenfalls bekannt sein, dass das Rechnen mit Gleitpunktzahlen bei naiver Implementierung **immer** zu **Rundungsfehlern** führen kann. Der **relative Fehler** ist dabei beschränkt durch die Auflösung (es gilt $\rho = 1/B^{t-1}$), der *absolute Fehler* aber hingegen kann aufgrund der *logarithmischen* Verteilung sehr **groß** werden. Neben Rundungsfehlern kann es selbstverständlich ebenfalls zu *Diskretisierungsfehlern* oder *Iterationsfehlern* kommen.
Der Wunsch ist es, dass das Ergebnis einer Gleitpunktoperation $*_{\mathcal{FP}}$ zu dem Prozess der exakten Resultatsberechnung mit anschließender Rundung nach Modell äquivalent ist. Die Auflösung ρ ist dabei dann eine Schranke für den relativen Rundungsfehler. Wir stellen somit dann die starke Hypothese auf:

i Es existiert ein $\widetilde{\varepsilon} \in \mathcal{O}(\rho)$, so dass

$$a *_{\mathcal{FP}} b = (a * b) \cdot (1 + \varepsilon),$$

wobei $|\varepsilon| = |\varepsilon(*, a, b)| \leqslant \widetilde{\varepsilon}|$.

Wichtige Informationen

Kommutativität gilt auch weiterhin bei Fließkommaoperationen, Assoziativität und Distributivität hingegen **nicht**!

Hinsichtlich des **relativen Fehlers** sind Multiplikation und Divisionen gutartig, wohingegen die Adddition und Subtraktion sehr gefährlich werden können, unter anderem wegen des Auslöschungseffektes.

! Durch die Verwendung von Gleitkommazahlen, welche dem Informatiker aus verschiedenen Veranstaltungen bereits bekannt sein sollten, werden Zahlen aufgrund der **festen** Mantisse mit einem gewissen relativen **Rundungsfehler** gespeichert. Der Fehler ist ursprünglich die Anzahl an Mantissenziffern, kann aber durch gewisse Rechenoperationen **drastisch erhöht werden**. Man nennt diesen Effekt dann **Auslöschung**. Wir betrachten dazu folgendes **Beispiel 4.1:**16-ziffrige Mantisse

$$0,\underbrace{7948236243749276}_{\text{Fehler von } 10^{-16}} - 0,\underbrace{7948236243749241}_{\text{Fehler von } 10^{-16}} = \underbrace{0,35}_{\text{Fehler von } 10^{-2}} \cdot 10^{-14}$$

※

Kondition und Stabilität von Verfahren In der Numerik werden durch die Konzepte der *Kondition* und *Stabilität* die Anfälligkeit eines Verfahrens gegenüber Störungen beschrieben. Die *Kondition* eines **Problems** ist vom verwendeten Verfahren **unabhängig**, die *Stabilität* eines **Verfahrens** hängt hingegen maßgeblich vom **Verfahren** ab.
Ein Problem heißt dann **gut konditioniert**, wenn – *exakte Rechnung vorausgesetzt* – die Fehler der Eingabedaten *nicht verstärkt* werden. Damit gilt dann:

$$\frac{|F(\tilde{x}) - F(x)|}{|F(x)|} \approx \frac{|\tilde{x} - x|}{|x|}.$$

Ein Verfahren heißt **(vorwärts-)stabil**, wenn die Rechenfehler des Verfahrens sich nicht akkumulieren, wenn also gilt, dass

$$\frac{\left|\tilde{F}(y) - F(y)\right|}{|F(y)|} << 1.$$

Stabilität von Operationen

! Während die arithmetischen Grundoperationen immer **stabil** sind, sind es die **Hintereinanderausführung** von *stabilen* Operationen **nicht automatisch**.

Akzeptable Ergebnisse Die obige Definition der Vorwärtsstabilität ist vor allem bei schlecht konditionierten Problemen verbesserungswürdig. Wir wollen deswegen ein Ergebnis y' **akzeptabel**, wenn es als exakt betrachtetes Ergebnis zu nur leicht gestörten Eingabewerten verstanden werden kann. Wenn also x die normale Eingabe, $F(x)$ das exakte Ergebnis und x' der leicht gestörte Eingabewert mit dem gestörten Ergebnis $F(x')$ ist, so muss der Zusammenhang $\|x' - x\| \leqslant \varepsilon$ erfüllt sein mit ε sehr klein. Man nennt ein solches Verfahren dann auch *rückwärtsstabil*.
Der Nutzen von akzeptablen Ergebnissen hängt dann aber auch wieder von der Konditionierung des Problems ab, denn akzeptable Ergebnisse müssen nicht unbedingt sehr genau sein.

Gute Kondition (+)

- Kleine Eingabestörungen sind **unkritisch** zu betrachten, da sie nur zu **kleinen** Störungen der Ergebnisse führen
- Es lohnt sich einen ***guten*** Algorithmus zu entwickeln, welcher die *gute Kondition* ausnutzt.

Schlechte Kondition (−)

- Lösungen reagieren **empfindlich** auf **kleinste Störungen** der Eingabe.
- Eine Verbesserung des genutzten Verfahrens ist hier **sinnlos** und **verschwendete Zeit**, da das Verfahren nichts an der Kondition des Problems ändert.
- Aus Sicht des Anwenders ist die Aufgabenstellung hier **besonders kritisch zu hinterfragen**.

Definition der Kondition Wir definieren die *(relative)* **Kondition** κ eines Problems $y = F(x)$, mit $F \in \mathrm{Abb}(\mathbb{R}^m, \mathbb{R}^n)$ als die kleinste Zahl $\kappa \geqslant 0$, für die gilt, dass ein $\delta > 0$ existiert, so dass für alle $\widetilde{x}$ mit $0 < \|\widetilde{x} - x\| < \delta$ gilt, dass

$$\frac{\|F(\widetilde{x}) - F(x)\|}{\|F(x)\|} \leqslant \kappa \cdot \frac{\|\widetilde{x} - x\|}{\|x\|}.$$

Konditionszahl einer Matrix Wir definieren die Kondition einer *(invertierbaren)* Matrix $A \in \mathbb{R}^{n \times n}$ als

$$\kappa(A) = \frac{\max\limits_{\|x\|=1} \|Ax\|}{\min\limits_{\|x\|=1} \|Ax\|}.$$

Es gilt dann auch

$$\kappa(AB) \leqslant \kappa(A) \cdot \kappa(B).$$

i Von den in Kapitel 1 vorgestellten Verfahren ist **nur** die LR-Zerlegung **mit** Pivotsuche und die QR-Zerlegung stabil. Bei der QR-Zerlegung gilt dann sogar:

$$\kappa(A) = \kappa(Q \cdot R) = \kappa(R)$$

4.2.2 Quantisierung von Farbwerten mit dem *median-cut-algorithm*

Verfahren 4.1 (*Verfahren der mittleren Schnitte (median-cut-algorithm)*)

Eingabe: Punktewolke $\mathfrak{P}$ in Gebiet $\mathcal{G}$, Anzahl an Schritten N.
Globale Variablen: s aktuelle Anzahl an Schritten, vorinitialisiert mit 0.

(1) Finde eine am besten passende umgebende Box der Wolke $\mathfrak{P}$ (*best-fitting bounding box*), dies liefert dann ein Rechteck oder auch einen Quader.

(2) Teil den Quader an der längsten Kante, so dass **beide Teile** (in etwa) *gleich viele* Punkte enthalten. Damit bildet man den *Median* der Werte.

(3) **Falls** $s \geqslant N$, so halte an, **andernfalls** gehe zu Schritt (1).

Die Wahl der Repräsentanten erfolgt dann über beispielsweise den Mittelpunkt der Quader oder aber auch den Mittelwert der enthaltenen Puntke.

4.2.3 Vektorquantisierung

Minimierer der geringsten Quadrate (*least square minimizer*) im *eindimensionalen* Der *least square minimizer* ist eine Alternative bei nicht gleichverteilten Werten mit nicht äquidistanter Schrittweite und einem Repräsentanten aus einem Schrittweitenintervall.
Wir nehmen nun an, dass die Werte gemäß einer Wahrscheinlichkeitsdichtefunktion $p(r)$ verteilt sind und dass damit dann

$$P(r \leqslant R) = \int_{-\infty}^{R} p(\rho)\, d\rho$$

gilt. Ist r dann eine Realisierung einer Zufallsvariablen mit Dichte $p(\rho)$, welche die Wahrscheinlichkeitsverteilung der Punkte beschreibt, also

$$p(\rho) = 0, \text{ wenn } r_{max} < \rho < r_{min}$$

gilt, so ist der mittlere quadratische Fehler durch $\varepsilon = \sum_{j=0}^{L-1} \int_{w_j}^{w_{j+1}} (\rho - r_j)^2 p(\rho)\, d\,\rho$ gegeben.

Satz 4.4

Das obige Funktional ε wird minimiert, falls die Bedingung der nächsten Nachbarn (nearest neighbor)

$$w_j = \frac{r_{j-1} + r_j}{2}$$

und die Schwerpunktsbedingung

$$r_j = \frac{\int_{w_j}^{w_{j+1}} \rho \cdot p(\rho)\, d\rho}{\int_{w_j}^{w_{j+1}} p(\rho)\, d\rho}$$

gelten.

Mehrdimensionale Approximation Wir wollen nun ein $r \in \mathbb{R}^n$ mit $w_{i,min} \leqslant r_i \leqslant w_{i,max}$ und $1 \leqslant i \leqslant n$ approximieren. Der gesamte Wertebereich spannt sich dann auf als

$$W := \bigtimes_{i=1}^{n} \left[w_{i,min}, w_{i,max} \right].$$

Wir haben dazu dann L Vektoren r_i, welche die Repräsentanten der disjunkten Teilgebiete W_i darstellen, womit sich dann ein Quantisierungsfehler von

$$\varepsilon = \sum_{i=1}^{L} \int_{W_i} \|r - r_i\|^2 \cdot \rho(r)\, dr,$$

wobei $\rho(r)$ eine r dimensionale Dichte darstellt. Wir stellen einen ähnlichen Satz zu Optimalitätskriterien wie oben auf und erkennen

Satz 4.5

Das obige Fehlerfunktional ε wird für gegebene Repräsentanten r_i mit $1 \leqslant i \leqslant L$ minimiert, wenn für $r \in W_i$ gilt, dass

$$\forall\, j \neq i.\, \|r - r_i\| \leqslant \left\| r - r_j \right\|.$$

Ist eine Zerlegung von W in disjunkte W_i gegeben, so gilt dann für die Repräsentanten

$$r_i = \frac{\int\limits_{W_i} r \cdot \rho(r)\, dr}{\int\limits_{W_i} \rho(r)\, dr}.$$

Iteratives Verfahren zur Bestimmung der optimalen Vektorquantisierung Seien nun jeweils Mengen von M n-dimensionalen Trainingsvektoren x_i und L n-dimensionalen Kodevektoren r_i gegeben. Zudem sei eine zu den Kodevektoren gehörige Partition P des $\mathbb{R}^n$ ebenfalls wie eine jedem Trainingsvektor einen Kodevektor zuordnende Funktion $Q \in \mathrm{Abb}(\mathbb{R}^n, \mathbb{R}^n)$ bekannt. Liegt ein Trainingsvektor x_i dann in P_j, so wird er dann auch durch $Q(x_i) = r_j$ approximiert. Unser Ziel ist es den Fehler, gegeben durch

$$\varepsilon = \frac{1}{M \cdot n} \cdot \sum_{i=1}^{M} \|x_i - Q(x_i)\|^2,$$

zu minimieren. Auch hier gibt es wieder Optimalitätskriterien, welche denen der vorausgegangenen Sätze ähneln. So gibt es auch hier die Bedingung der nähesten Nachbarn (*nearest neighbor condition*), welche die Partition P auf Voronoiregionen aufspaltet mit

$$P_i = \left\{ x \;\middle|\; \forall\, 1 \leqslant k \leqslant L.\ \|x - r_i\|^2 \leqslant \|x - r_k\|^2 \right\},$$

sowie die Schwerpunktsbedingung (*centroid condition*), sprich die Repräsentanten werden mit

$$r_i = \frac{1}{|M_i|} \cdot \sum_{m \in M_i} x_m,$$

wobei $M_i = \left\{ m \;\middle|\; x_m \in P_i \right\}$ gilt, gewählt.

Verfahren 4.2 (*Lloyds-Algorithmus (k-means clustering)*)

(1) Berechne mit

$$r_1^* = \frac{1}{M} \cdot \sum_{i=1}^{M} x_i \quad \text{und} \quad \varepsilon^* = \frac{1}{M \cdot n} \cdot \sum_{i=1}^{M} \|x_i - r_1^*\|$$

unter der Annahme, dass $P_1 = \mathbb{R}^n$ und $M_1 = \left\{ 1, \ldots, M \right\}$, eine Startlösung.

(2) Spalte die Kodevektoren auf in

$$r_i^{(0)} - (1 + \beta) \cdot r_i^* \quad \text{und} \quad r_{i+L}^{(0)} = (1 - \beta) \cdot r_i^*.$$

(3) Ordne die Samples zu den neuen Kodevektoren zu, sprich bestimme P_i und M_i.

(4) Berechne die neuen Kodevektoren, bestimme also die r_i.

(5) **Solange** der Abstand der alten zu den neuen Kodevektoren größer ist als eine *vorher festgelegte* Schranke, gehe zu Schritt (3)

(6) **Falls** die gewünschte Anzahl der Kodevektoren noch nicht erreicht ist gehe zu Schritt (2).

4.3 Aufgaben

Aufgabe 4.1

Die Filterung eines diskreten Signals f mit der Filtermaske g ist definiert durch die diskrete Faltung:

$$f^{\text{filtered}}(n) = \sum_k f(k) \cdot g(n-k)$$

Teilaufgabe 4.1a

Die unendliche periodische Zahlenfolge

$$[\ldots, -1, 0, 1, -1, 0, 1, -1, 0, 1, \ldots]$$

wird (i) mit einem Tiefpassfilter der Größe 3 ($[g_{-1}, g_0, g_1] = 1/3 \cdot [1,1,1]$) und (ii) mit einem Sobel-Filter ($[g_{-1}, g_0, g_1] = [-1,0,1]$) gefiltert. Beide Filterungen sind unabhängig voneinander zu betrachten! Bestimmen Sie jeweils die Ergebnisse.

Teilaufgabe 4.1b

Beschreiben Sie mit je einem Stichwort, welche Auswirkungen ein Sobelfilter und ein Tiefpassfilter auf ein Graustufenbild haben.

Teilaufgabe 4.1c

Welche der folgenden 2D-Filter sind separierbar? Geben Sie im Falle der Separierbarkeit jeweils auch die beiden 1D-Filter an.

$$F_1 = \frac{1}{9} \cdot \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} \quad F_2 = \frac{1}{8} \cdot \begin{pmatrix} 0 & 1 & 2 \\ 1 & 0 & 1 \\ 2 & 1 & 0 \end{pmatrix} \quad F_3 = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} \quad F_4 = \frac{1}{6} \cdot \begin{pmatrix} 1 & 2 & 2 & 1 \\ 2 & 4 & 4 & 2 \\ 1 & 2 & 2 & 1 \end{pmatrix}$$

Teilaufgabe 4.1d

Was sind die Vorteile von separierbaren Filtern?

Teilaufgabe 4.1e

Filtern Sie das nachfolgende 5×5 Bild mit der Filtermaske $G = \frac{1}{4} \cdot \begin{pmatrix} 1 & -2 & 1 \\ 2 & -4 & 2 \\ 1 & -2 & 1 \end{pmatrix}$. Dabei sollen die Randpunkte *nicht* berechnet werden.

$$\text{Input} = \begin{pmatrix} 12 & 8 & 4 & 8 & 12 \\ 16 & 12 & 8 & 8 & 12 \\ 20 & 16 & 12 & 12 & 12 \\ 16 & 12 & 8 & 8 & 12 \\ 12 & 8 & 4 & 8 & 12 \end{pmatrix}$$

Teilaufgabe 4.1f

Ein (Grau-)bild mit $N \times N$ Pixeln soll mit einem **nicht** separierbaren Filter der Größe M gefiltert werden. Dabei werden nur die inneren $(N-k) \times (N-k)$ Pixel berechnet. Berechnen Sie die Anzahl an aritmetischen Operationen, sowie den absoluten Anteil der Multiplikationen daran. Was ändert sich wenn ein separierbarer Filter verwendet wird?

Aufgabe 4.2

Eine Funktion f soll nun nacheinander mit h_1 und h_2 gefaltet werden.

Teilaufgabe 4.2a

Ist die Reihenfolge der Operanden hierbei wichtig?

Teilaufgabe 4.2b

Die zweimalige Faltung $f \mapsto f * h_1 \mapsto (f * h_1) * h_2$ kann man auch durch **eine** Faltung mit f erreichen. Mit welcher Funktion gelingt das?

Teilaufgabe 4.2c

Was geschieht, wenn man eine Funktion f mit der Diracfunktion δ_a faltet?

Teilaufgabe 4.2d

Prüfen Sie die folgenden Behauptungen auf Korrektheit und kreuzen Sie das entsprechende Kästchen an. Hierbei seien die Funktionen f, g, h jeweils so gewählt, dass die Faltungen existieren.

richtig	falsch	
☐	☐	$\forall\, f, g, h.\ f * (g + h) = (f * g) + (f * h)$
☐	☐	$\forall\, f, g.\ f * g = g * f$
☐	☐	$\forall\, f.\ f * f = f$
☐	☐	$\forall\, f, g, h.\ f * (g * h) = (h * g) * f$

Aufgabe 4.3

Gegeben sind die Funktionen

$$g(x) := \begin{cases} 1 & \text{, falls } 0 \leqslant x \leqslant 1 \\ 0 & \text{sonst} \end{cases} \quad \text{und} \quad h(x) := \begin{cases} 1 + x & \text{, falls } -1 \leqslant x \leqslant 0 \\ 1 - x & \text{, falls } 0 \leqslant x \leqslant 1 \\ 0 & \text{sonst} \end{cases}.$$

Beantworten Sie die folgenden Fragen zum Faltungsprodukt $f = g * h$:

Teilaufgabe 4.3a

Auf welchem Intervall ist $f(x)$ ungleich Null?

Teilaufgabe 4.3b

An welcher Stelle x_0 nimmt die Funktion $f(x)$ das Maximum an?

Teilaufgabe 4.3c

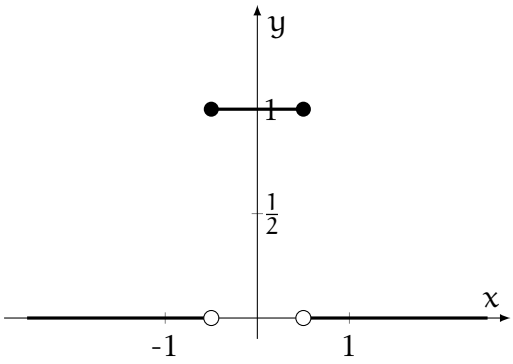
Welches ist der maximale Funktionswert $f(x_0)$?

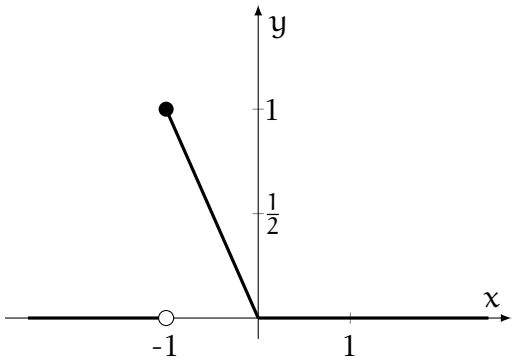
Aufgabe 4.4

Gegeben sind nun die Funktionen

$$h_1(x) := \begin{cases} 1 & \text{, falls } |x| \leqslant 1/2 \\ 0 & \text{sonst} \end{cases} \quad \text{und} \quad h_2(x) := \begin{cases} -x & \text{, falls } -1 \leqslant x \leqslant 0 \\ 0 & \text{sonst} \end{cases},$$

sowie die Funktionsgraphen der Funktionen:





Zeichnen Sie das Ergebnis der Faltung $h_1 * h_2$.

Aufgabe 4.5

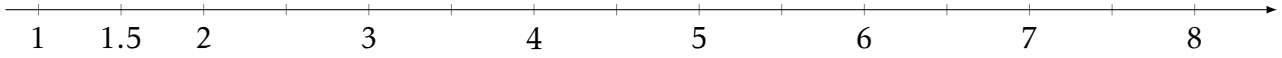
Gegeben sind die Funktionen

$$h_3(x) := \begin{cases} 0 & \text{, falls } x < 0 \\ x & \text{sonst} \end{cases} \quad \text{und} \quad h_4(x) := \begin{cases} 2 & \text{, falls } 0 < x < 1 \\ 0 & \text{sonst} \end{cases}.$$

Geben Sie das Ergebnis der Faltung $h_3 * h_4$ als mathematischen Ausdruck an, indem Sie das Faltungsintegral lösen.

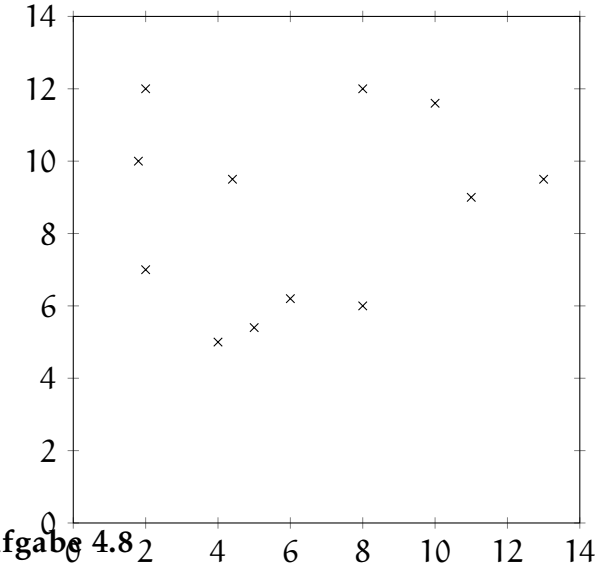
Aufgabe 4.6

Betrachten Sie $FP_{2,3}$, die Menge der Gleitpunktzahlen zur Basis $B = 2$ mit Mantissenlänge $t = 3$. Markieren Sie in der Abbildung (jeweils durch ein Kreuz) alle $a \in FP_{2,3}$ mit $1 \leqslant a \leqslant 8$:



Aufgabe 4.7

Gegeben ist eine Punktwolke mit 12 Punkten. Führen Sie zwei Schritte des Median-Cut-Verfahrens durch, wählen Sie nach dem zweiten Schritt einen jeweiligen Repräsentanten als Mittelpunkt der jeweiligen Punktwolke und benutzen Sie dabei die folgenden Vorlagen:



Aufgabe 4.8

Prüfen Sie die folgenden Behauptungen auf Korrektheit und kreuzen Sie das entsprechende Kästchen an.

Teilaufgabe 4.8a

Bei welchen Operationen kann die Auslöschung zweier positiver Gleitpunktzahlen fast gleicher Größe auftreten?

richtig	falsch	
☐	☐	Addition
☐	☐	Multiplikation
☐	☐	Divison
☐	☐	Subtraktion

Teilaufgabe 4.8b

Welche Rechengesetze sind bei Rechenoperationen mit Gleitpunktzahlen erfüllt?

richtig	falsch	
☐	☐	Assoziativgesetz
☐	☐	Kommutativgesetz
☐	☐	Distributivgesetz

Teilaufgabe 4.8c

richtig	falsch	
☐	☐	Die erreichbare Genauigkeit von Gleitpunktzahlen wird durch die Anzahl verwendeter Bits für den Exponenten, der Wertebereich durch die Anzahl Bits für die Mantisse bestimmt.
☐	☐	Die erreichbare Genauigkeit von Gleitpunktzahlen wird durch die Anzahl verwendeter Bits für die Mantisse, der Wertebereich durch die Anzahl Bits für den Exponenten bestimmt.

KAPITEL 5

Rekonstruktion kontinuierlicher Daten — Interpolation und Approximation

Interpolation (+)

Bekannt seinen n ($n \in \mathbb{N}$) Punkte im Raum. Soll die rekonstruierte Funktion nun **exakt** durch die Punkte verlaufen, so sprechen wir von **Interpolation**. Es ist *nicht in jedem Fall* das bessere Verfahren (*Messfehler*).

Approximation (−)

Bekannt seinen n ($n \in \mathbb{N}$) Punkte im Raum. Soll die rekonstruierte Funktion nun nur *näherungsweise* durch die Punkte verlaufen, so sprechen wir von **Approximation**. Vorteil: Besserer Umgang mit *Messfehlern*.

5.1 Interpolation im eindimensionalen Raum

Problem 5.1 (*Interpolation*)

Von einer unbekannten Funktion[1] $f \in \mathrm{Abb}(\mathbb{R},\mathbb{R})$ seien die Werte an endlich vielen Stellen, sowie gewisse Grundeigenschaften bekannt. Wir wollen die Stützstellen als x_i, die Werte der Funktion an den Stützstellen (*Stützwerte*) als y_i mit $1 \leqslant i \leqslant n$ auffassen. Wir wollen dann f aus den gegebenen endlich vielen Punkten *näherungsweise* rekonstruieren.

Wir wählen dazu eine Klasse K von Funktionen mittels derer wir die Rekonstruktion starten wollen. Wir setzen als Kriterium an K voraus, dass …

… f in K gut approximierbar sein muss.

… K auf Computern **gut** *darstellbar* und *manipulierbar* sein muss.

5.1.1 Lokale Verfahren

Bei **lokalen Verfahren** hängt der interpolierte Wert $p(x)$ **nur** von den unmittelbar „benachbarten" Werten ab. Gilt beispielsweise $x \in [x_i, x_{i+1}]$, so gehen in die Betrechnung von $p(x)$ nur y_i und y_{i+1} mit ein.
Seien nun im Folgenden also eine Menge $\mathfrak{A}_S$ von n Stützstellen x_i und eine Menge $\mathfrak{O}_S$ von n Stützwerten y_i von f gegeben und eine kontinuierliche Repräsentation $p(x)$ von f gesucht.

Stückweise konstante Interpolation (*nearest neighbor*) Um den Wert an der Stelle x anzunähern, suche den nähesten Nachbarn, sprich die nächst gelegene Stützstelle x_j. Der interpolierte Wert ist dann mit y_j gegeben. Die Repräsentation p lässt sich dann auch allgemein durch

$$p(x) = \begin{cases} y_1 & x_1 \leqslant x \leqslant \frac{1}{2}(x_1 + x_2) \\ y_i & \frac{1}{2}(x_{i-1} + x_i) < x \leqslant \frac{1}{2}(x_i + x_{i+1}) \quad \text{für alle } 2 \leqslant i \leqslant n-1 \\ y_n & \frac{1}{2}(x_{n-1} + x_n) < x \leqslant x_n \end{cases}$$

[1] Selbstverständlich sind Daten im Allgemeinen (*in der Realität*) durch Relationen beschrieben, wir bleiben der Einfachheit halber zuerst bei Funktionen.

bestimmen. Der einzige Rechenaufwand bei dieser Methode beläuft sich auf die Suche der Nachbarn, welche bei äquidistanten Stützstellen mit Schrittweite h in $\mathcal{O}(1)$, bei bereits geordneten Stützstellen in $\mathcal{O}(\log(n))$ und andernfalls in $\mathcal{O}(n\log(n))$ bestimmen lässt.
Problematisch ist die Ungenauigkeit, welche mit einer konstanten Interpolation einherget. Als obere Schranke für den Fehler bei glattem f lässt sich

$$|p(x)-f(x)| \leqslant \frac{\overbrace{\max_{1\leqslant i\leqslant n}\left\{x_{i+1}-x_i\right\}}^{=:h}\cdot \max_{x_1\leqslant\xi\leqslant x_n}\left\{|f'(\xi)|\right\}}{2}$$

Stückweise lineare Interpolation Die stückweise lineare Interpolation ist das am weitesten verbreitete Interpolationsverfahren. Um den Wert an der Stelle x hierbei anzunähern, suchen wir zuerst die *nächst gelegene* linke und rechte Stützstelle x_i und x_{i+1}, so dass $x_i \leqslant x \leqslant x_{i+1}$ gilt, und interpolieren dann im Intervall $[x_i, x_{i+1}]$ linear. Wir stellen dann für $1 \leqslant i < n$ die Funktion

$$p_i(x) = m_i \cdot (x-x_i) + y_i = \underbrace{\frac{y_{i+1}-y_i}{x_{i+1}-x_i}}_{=:m_i}\cdot(x-x_i)+y_i$$

auf, erhalten damit dann insgesamt die Interpolationsfunktion

$$p(x) = \sum_{i=1}^{n-1} p_i(x)\cdot \mathbb{1}_{[x_i,x_{i+1}]}.$$

Alternativ kann für das p_i auch das gewichtete Mittel von y_i und y_{i+1} gewählt werden, es ergibt sich somit dann

$$p_i(x) = \omega\cdot y_i + (1-\omega)\cdot y_{i+1} = \underbrace{\frac{x_{i+1}-x}{x_{i+1}-x_i}}_{=:\omega}\cdot y_i + \underbrace{\frac{x-x_{i+1}}{x_{i+1}-x_i}}_{=:1-\omega}\cdot y_{i+1}.$$

Catmull-Rom — Stückweise kubische Interpolation Wir schätzen hierbei an jeder Stützstelle x_i die erste Ableitung y_i' und finden dann auf jedem Teilintervall $T_i := [x_i, x_{i+1}]$ das (*eindeutige*) kubische Polynom, welches in den Endpunkten neben den Stützwerten auch noch die geschätzten Ableitungen interpoliert. Wir können Ableitungen im Allgemeinen über Vorwärts-, Rückwärts- sowie der zentralen Differenz abschätzen, dabei gilt, dass letztere den geringsten Fehler macht. Damit berechnet sich y_i' im einfachen Fall durch

$$y_i' = \frac{y_{i+1}-y_{i-1}}{x_{i+1}-x_{i-1}},$$

will man die zentrale Differenz exakt betrachten, so nimmt man das gewichtete Mittel von Vorwärts- und Rückwärtsdifferenz, damit ist dann y_i' berechenbar durch

$$y_i' = \frac{x_i-x_{i-1}}{x_{i+1}-x_{i-1}}\cdot y_{i,fw}' + \frac{x_{i+1}-x_i}{x_{i+1}-x_{i-1}}\cdot y_{i,bw}' = \frac{x_i-x_{i-1}}{x_{i+1}-x_{i-1}}\cdot \underbrace{\frac{y_{i+1}-y_i}{x_{i+1}-x_i}}_{=y_{i,fw}'} + \frac{x_{i+1}-x_i}{x_{i+1}-x_{i-1}}\cdot \underbrace{\frac{y_i-y_{i-1}}{x_i-x_{i-1}}}_{=y_{i,bw}'}.$$

Das kubische Polynom p bestimmt sich dann mit

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

wobei sich die a_i durch

$$a_0 = \frac{y_i}{(x_{i+1}-x_i)^3},\quad a_1 = 3\cdot a_0 + \frac{y_i'}{(x_{i+1}-x_i)^2},\quad a_2 = 3\cdot a_3 - \frac{y_{i+1}'}{(x_{i+1}-x_i)^2}\quad\text{und}\quad a_3 = \frac{y_{i+1}}{(x_{i+1}-x_i)^3}$$

bestimmen lassen.

Fehlerabschätzungen lokaler Interpolanten Wir wollen nun kurz annehmen, dass die Stützstellen äquidistant mit Schrittweite h verteilt sind, sowie die zu interpolierende Funktion f genügend oft differenzierbar ist.[2] Dann lassen sich folgende Fehlerabschätzungen erkennen:

Verfahrensname	obere Fehlerschranke	„Fehlerkomplexität“
Konstante Interpolation (*nearest neighbor*)	$\leqslant \frac{\lvert f'(\xi)\rvert}{2 \cdot h}$	$\in \mathcal{O}(h)$
Lineare Interpolation	$\leqslant \frac{\lvert f''(\xi)\rvert}{8 \cdot h^2}$	$\in \mathcal{O}(h^2)$
Kubische Interpolation (*catmull rom*)	$\leqslant \frac{\lvert f'''(\xi)\rvert}{24 \cdot h^3}$	$\in \mathcal{O}(h^3)$

5.1.2 Globale Verfahren — Polynominterpolation

Bei **globalen Verfahren** hängt der interpolierte Wert $p(x)$ von allen verfügbaren Werten ab. Wir rufen uns aus den Mathematikveranstaltungen in Erinnerung, dass die Polynome bis zum Grad $n-1$ einen n-dimensionalen Vektorraum aufspannen:

$$\mathbb{P}_n = \operatorname{span}\left\{ x^i \;\middle|\; i \in \{0, \ldots, n-1\} \right\}$$

Man nennt diese Basis auch **Monom**- oder **Taylorbasis**.

Lagrangepolynominterpolation Wie im lokalen Fall ist auch bei der Lagrangepolynominterpolation eine Menge $\mathcal{A}_S$ an n Stützstellen x_i, sowie eine Menge $\mathcal{O}_S$ an n Stützwerten y_i gegeben. Ein Polynom $p \in \mathbb{P}_n$ interpoliert diese Werte genau dann, wenn für alle i gilt, dass $p(x_i) = y_i$. Setzen wir eine „Polynomvorlage“ in die n Interpolationsbedingungen ein, erhalten wir ein lineares Gleichungssystem mit n Gleichungen für die n unbekannten Koeffizienten:

$$\underbrace{\begin{pmatrix} 1 & x_1 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & \cdots & x_n^{n-1} \end{pmatrix}}_{A} \cdot \underbrace{\begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix}}_{c} = \underbrace{\begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix}}_{y}$$

Dabei ist A die Vandermondematrix des Problems, dessen Determinante nicht 0 ist und sich über

$$\det(A) = \prod_{1 \leqslant j < k \leqslant n} (x_j - x_k)$$

berechnen lässt. Damit ist sie ebenfalls quadratisch, voll besetzt und nicht singulär, solange alle Stützstellen paarweise verschienden sind. Das Problem mit der Vandermondematrix ist die schlechte Konditionieren bei schon mäßig gewähltem n. Dennoch können wir aus der Nichtsingularität der Matrix die Existenz eines eindeutig bestimmten Interpolationspolynoms vom Grad $n-1$ folgern. Wegen der **schlechten Konditionierung** der Matrix ist dieser Weg allerdings der **falsche**!

Rekursionsformel von Aitken-Neville

$$p_{i,k}(x) = \begin{cases} y_i & \text{, wenn } k = 0 \\ p_{i,k-1}(x) \cdot \dfrac{x_{i+k} - x}{x_{i+k} - x_i} + p_{i+1,k-1}(x) \cdot \dfrac{x - x_i}{x_{i+k} - x_i} & \text{, sonst} \end{cases}$$

[2] Die Forderung der Äquidistanz kann auch weggelassen werden, statt der globalen Schrittweite h tritt dann allerdings wie oben der maximale Abstand aufeinanderfolgender Stützstellen auf. Das neue h berechnet sich dann durch $h = \max\limits_{1 \leqslant i \leqslant n} \left\{ x_{i+1} - x_i \right\}$.

Wir stellen fest, dass diese Rekursionsformel das gewichtete Mittel der beiden Polynome niedrigeren Grades berechnet. Diese „Ursprungspolynome“ vom Grad $k-1$ haben dann jeweils k Interpolationspunkte, von denen sich $k-1$ überlappen. Das Polynom am Ende eines Schrittes hat alle Interpolationspunkte der beiden Polynome des Schrittes vorher und ist vom Grad k.
Dieses Rekursionsschema darf ebenso wenig wie das der Fibonaccizahlen „naiv“ implementiert werden, da dann für den Polynomgrad n der Aufwand in $\mathcal{O}(2^n)$ liegt. Stattdessen verwendet man gerne einen Ansatz der dynamischen Programmierung, was dann auf ein Dreiecksschema führt. Dieses werden wir im Zusammenhang mit der Newtoninterpolation näher kennenlernen.

Alternative Polynombasen I — Bernsteinpolynombasen Bernsteinpolynombasen sind sehr beliebt bei Graphikern für geometrische Modellierungsaufgaben, für Polynominterpolationen eher weniger geeignet. Wir werden sie wieder bei den Bézierkurven antreffen.

$$B_i^n(x) = \binom{n}{i} \cdot (1-x)^{n-i} \cdot x^i$$

Mit obigen Funktionen gilt dann für den Raum $\mathbb{P}_{n+1}$:

$$\mathbb{P}_{n+1} = \operatorname{span}\left\{ B_i^n(x) \;\middle|\; 0 \leqslant i \leqslant n \right\}$$

Alternative Polynombasen II — Lagrangepolynombasen Zu den gegebenen Stützstellen x_i definieren wir die Polynome

$$L_i(x) = \frac{\prod\limits_{\substack{j=1 \\ j\neq i}}^{n} (x - x_j)}{\prod\limits_{\substack{j=1 \\ j\neq i}}^{n} (x_i - x_j)}$$

für alle $1 \leqslant i \leqslant n$. Dabei gilt, dass die L_i linear unabhängig sind und damit eine Basis von $\mathbb{P}_n$ bilden, sowie dass $L_i(x_j) = \delta_{ij}$.
Das (*eindeutige*) Interpolationspolynom ist dann durch

$$p(x) = \sum_{i=1}^{n} y_i \cdot L_i(x)$$

gegeben.

Alternative Polynombasen III — Newtonpolynombasen Zu den gegebenen Stützstellen x_i definieren wir die Polynome

$$N_i(x) = \prod_{j=1}^{i-1} (x - x_j)$$

für alle $1 \leqslant i \leqslant n$. Das (*eindeutige*) Interpolationspolynom ist dann durch

$$p(x) = \sum_{i=1}^{n} a_{i-1} \cdot N_i(x)$$

gegeben. Die a_j berechen sich über den Algorithmus von **Aitken-Neville**. In dieser Darstellung ist dann auch das Interpolationspolynom effizient für alle x auswertbar. Wir betrachten dazu:

```
1 def evaluateNewton(x, a, xi, n): #a, xi: np.array(n)
2     eva = a[n-1]
3     for i in range(n-2, -1, -1):
4         eva = eva * (x - xi[i]) + a[i]
5     return eva
```

Der Algorithmus von Aitken-Neville

Verfahren 5.1 (*Algorithmus von Aitken-Neville*)

Eingabe: Stützstellen $x_i \in \mathfrak{A}_S$, Stützwerte $y_i \in \mathfrak{O}_S$.
Ausgabe: Koeffizienten a_i mit $0 \leqslant i \leqslant n-1$.

(1) $P = (p_{ik})_{\substack{1 \leqslant i \leqslant n-k \\ 0 \leqslant k \leqslant n-1}}$ und $k \leftarrow 1$.

(2) Für $1 \leqslant i \leqslant n$ mache $p_{i,0} \leftarrow y_i$.

(3) **Solange** $k < n$ **tue** Für $1 \leqslant i \leqslant n-k$ mache $p_{i,k} = \dfrac{p_{i+1,k-1} - p_{i,k-1}}{x_{i+k} - x_i}$

(4) $a \leftarrow p_1$.

Aufwand $\in \mathcal{O}(n^2)$

Das Verfahren lässt sich auch in Dreiecksform darstellen.

5.2 Multivariate Interpolation

Wir wollen nun Funktionen mehrerer Verändlicher betrachten. Wie im univariaten Fall sind wieder Stützstellen $x^i = (x_1^i, \ldots, x_n^i) \in \mathfrak{A}_S$ und Stützwerte $f_i = f(x^i) \in \mathfrak{O}_S$ gegeben und wir suchen Näherungswerte von f an den dazwischenliegenden Stellen. Wir wollen uns dabei auf maximal trivariate Funktionen einschränken, ähnliche Verfahren sind allerdings durchaus auch für allgemein multivariate Funktionen durchführbar.
Während wir im eindimensionalen Fall Punkte und Intervalle betrachtet haben, betrachten wir im zweidimensionalen Gitter mit *Kanten und Zellen* sowie **Ecken**. Die **Ecken** werden als **geometrische Eigenschaft** klassifiziert, sie geben die Lage an. *Kanten und Zellen* werden als *topologische Eigenschaften* klassifiziert, sie geben Nachbarschaftsbeziehungen an. Es gibt eine Formel für planare Gitter ohne Löcher, dass

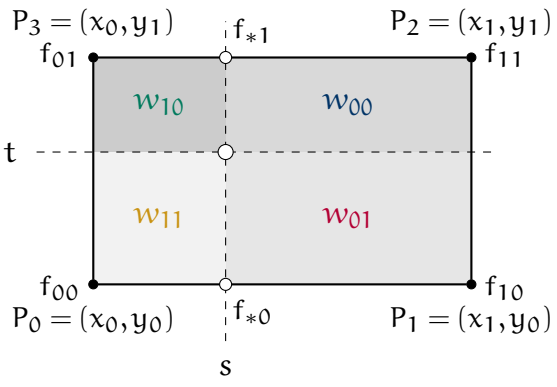
$$|\#\text{Zellen}| - |\#\text{Kanten}| + |\#\text{Ecken}| = 1.$$

5.2.1 Lokale Interpolationsverfahren

Auch hier berücksichtigen wir **ausschließlich** die Daten in der **Umgebung** des jeweiligen Punktes. Dies wird vorallem **zellenweise** deutlich, wenn wir **nur** Werte in den Eckpunkten der Zelle betrachten.

Bilineare Interpolation Wir verstehen hierunter ein zweistufiges Verfahren, bei welchem zuerst zwei Interpolationen in x-Richtung und anschließend eine Interpolation in y-Richtung mit den Ergebnissen des ersten Schrittes stattfindet. Wir erhalten somit in der bilinearen Interpolation das Ergebnis

$$\begin{aligned} BL(s,t) &= (1-\beta) \cdot \underbrace{\left((1-\alpha) \cdot f_{00} + \alpha \cdot f_{10}\right)}_{=f_{*0}} \\ &+ \beta \cdot \underbrace{\left((1-\alpha) \cdot f_{01} + \alpha \cdot f_{11}\right)}_{=f_{*1}} \end{aligned}$$



mit

$$\alpha = \frac{s - x_0}{x_1 - x_0} \quad \text{und} \quad \beta = \frac{t - y_0}{y_1 - y_0}.$$

Eine andere Sichtweise geht über die eingezeichneten Flächen**anteile**, wir erhalten somit

$$\begin{aligned} BL(s,t) &= \frac{(y_1 - t)\cdot(x_1 - s)}{(y_1 - y_0)\cdot(x_1 - x_0)} \cdot f_{00} + \frac{(y_1 - t)\cdot(s - x_0)}{(y_1 - y_0)\cdot(x_1 - x_0)} \cdot f_{10} + \frac{(t - y_0)\cdot(s - x_0)}{(y_1 - y_0)\cdot(x_1 - x_0)} \cdot f_{01} \\ &\quad + \frac{(t - y_0)\cdot(x_1 - s)}{(y_1 - y_0)\cdot(x_1 - x_0)} \cdot f_{11} \\ &= w_{00}\cdot f_{00} + w_{10}\cdot f_{10} + w_{01}\cdot f_{01} + w_{11}\cdot f_{11} \end{aligned}$$

Bivariate Polynome auf Tensorprodukten

Liegen die Stützstellen auf einem „Tensorprodukt"gitter, so kann die Interpolation zuerst in x-Richtung und dann in y-Richtung – *oder umgekehrt* – jeweils als **eindimensionale Interpolation** ausgeführt werden.

Multilineare Interpolation im n-dimensionalen Raum

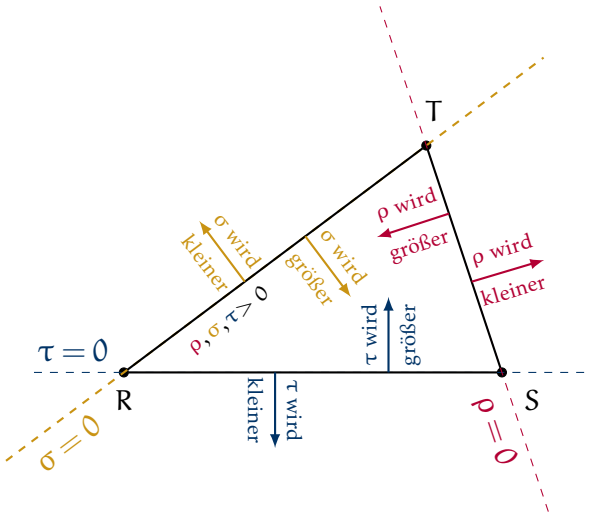
Es gibt

$$2^{n-1} + 2^{n-2} + \cdots + 2 + 1$$

lineare Interpolationen in **alle** n Raumrichtungen (*auch hier ist die Reihenfolge egal*).

Baryzentrische Koordinaten Seien drei Punkte R, S, T in der Ebene gegeben, die **nicht** auf einer Geraden liegen. Dann lässt sich ein jeder Punkt P der Ebene *eindeutig* darstellen durch

$$P = \rho\cdot P + \sigma\cdot S + \tau\cdot T,$$



wobei $\rho + \sigma + \tau = 1$ gelten muss. Das Tupel (ρ, σ, τ) heißt dann **baryzentrische Koordinate** von P bezüglich $\Delta(R, S, T)$. Die Koordinaten werden über das Lösen des Gleichungssystems

$$\begin{pmatrix} 1 & 1 & 1 \\ x_R & x_S & x_T \\ y_R & y_S & y_T \end{pmatrix} \cdot \begin{pmatrix} \rho \\ \sigma \\ \tau \end{pmatrix} = \begin{pmatrix} 1 \\ x_P \\ y_P \end{pmatrix}$$

mittels LR-Zerlegung.
Zudem gilt:

$$\rho = \frac{\mathbf{area}(\Delta(P,S,T))}{\mathbf{area}(\Delta(P,S,T))}, \quad \sigma = \frac{\mathbf{area}(\Delta(R,P,T))}{\mathbf{area}(\Delta(P,S,T))} \quad \text{und} \quad \tau = \frac{\mathbf{area}(\Delta(R,S,P))}{\mathbf{area}(\Delta(P,S,T))}$$

n-dimensionale baryzentrische Koordinaten

Statt eines Dreiecks betrachtet man nun einen Simplex bestehend aus n linear unabhängigen Vektoren und $n+1$ Punkten X_i. Damit ist jeder Punkt P des Raumes darstellbar durch

$$P = \sum_{i=1}^{n+1} \xi_i \cdot X_i.$$

Die Berechnung findet genauso wie im zweidimensionalen statt, man stellt wieder die Matrix $A = (a_ij)$ mit

$$a_{0j} = 1, a_{ij} = (X_j)_i$$

auf und löst das so entstehende LGS durch LR-/QR-Zerlegung oder einem iterativen Verfahren.

Lineare Interpolation Seien nun in den Eckpunkten des Dreiecks $\Delta(R,S,T)$ die Dtaen f_R, f_S und f_T gegeben, so erhält man den Wert des linearen Interpolanten an einer Stelle $P \in \Delta(R,S,T)$ durch

$$f_P = \rho \cdot f_R + \sigma \cdot f_S + \tau \cdot f_T,$$

wobei ρ, σ, τ die baryzentrischen Koordinaten von P bezüglich $\Delta(R,S,T)$ sind.

lineare Interpolation im n-dimensionalen Raum

Gegeben seien die baryzentrischen Koordinaten ξ des Punktes P im n-dimensionalen Raum zum Simplex $(X_i)_{1 \leqslant i \leqslant n+1}$ und die Daten der Funktion an den Randpunkten, so berechnet sich der Wert des linearen Interpolanten durch

$$f_P = \sum_{i=1}^{n+1} \xi_i \cdot f_{X_i}.$$

5.2.2 Globale Interpolationsverfahren im zweidimensionalen Raum

Tensorproduktansatz Wir setzten hierfür nunmehr voraus, dass die Stützstellen ein rechteckiges Feld bilden, also sich die Menge an Stützstellen durch

$$\mathfrak{A}_S = \left\{ (x_i, y_j) \;\middle|\; 1 \leqslant i \leqslant n, 1 \leqslant j \leqslant m \right\}$$

darstellen lässt. Seien dann zu jeder Stützstelle ein Funktionswert gegeben und ein Polynom

$$p(x,y) = \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} c_{pq} x^i y^j$$

gesucht, welches die Interpolationsbedingungen $p(x_i, y_j) = f_{ij}$ erfüllt.
Effiziente Berechnungen werden durch das Verwenden der Newtonpolynome garantiert, es ist das Gleichungssystem $XDY^T = F$ zu lösen, wobei X und Y die Newtonpolynomdreiecksmatrizen ausgewertet in den jeweiligen Stützstellen x_i oder y_i sind und F die Funktionswertematrix. Man löst dann die jeweils $n+m$ auftretenenden ***eindimensionalen*** *Interpolationsprobleme* möglichst effizient durch Aitken-Neville.

Probleme der Polynominterpolation

Im allgemeinen Fall ist eine solche Polynominterpolation allerdings nicht möglich. Denn nicht nur die Zahl der Freiheitsgrade ist problematisch (*Welchen Polynomgrad wählen wir für eine gewisse Anzahl an Punkten?*) auch die (theoretische) **Lösbarkeit** ist nicht garantiert. Ebenso können interpolierte Funktionen **mehrdeutig** sein.

Radiale Basisfunktionen Die Idee der *radialen Basisfunktionen* ist es eine eindimensionale Funktion $\widetilde{h}(r)$ *radialsymmetrisch* fortzusetzen. Eine Funktion h ist genau dann radialsymmetrich, wenn gilt, dass

$$h(x) = h(\|x\|).$$

Wir setzen dann den Interpolanten an als

$$g(x,y) = \sum_{i=1}^{n} d_i \cdot h_i(x,y),$$

wobei die d_i die noch zu bestimmenden Koeffizienten darstellen und h_i die um die Stützstelle (x_i, y_i) *zentrierte radiale Basisfunktion* ist. Sei $\widetilde{h}(r)$ die eindimensionale Funktion, welche mit

$$h_0(x,y) = \widetilde{h}(\sqrt{x^2+y^2})$$

radialsymmetrisch fortgesetzt wurde, für die um die Stützstelle (x_i, y_i) zentrierte RBF bestimmt man dann über

$$h_i(x,y) = h_0(x-x_i, y-y_i).$$

Die Koeffizienten werden dann durch die n Interpolationsbedingungen

$$g(x_j,y_j) = \sum_{i=1}^{n} d_i \cdot h_i(x_j,y_j) = f_j$$

bestimmt.

Radiale Basisfunktionen mit linearer Präzision Der Unterschied ist hierbei, dass lineare Funktionen möglichst exakt rekonstruiert werden. Wir unterscheiden deswegen im Ansatz und setzten

$$f(x,y) = a + bx + cy + \sum_{i=1}^{n} d_i h_i(x,y)$$

an. Damit verschwinden die Momente aller Ordnungen kleiner gleich 1. Für das Bestimmen der Koeffizienten lösen wir daraufhin folgendes $(n+3) \times (n+3)$ Gleichungssystem:

$$\begin{pmatrix} 0 & 0 & 0 & \sum_j h_1(x_j,y_j) & \cdots & \sum_j h_n(x_j,y_j) \\ 0 & 0 & 0 & \sum_j x_j h_1(x_j,y_j) & \cdots & \sum_j x_j h_n(x_j,y_j) \\ 0 & 0 & 0 & \sum_j y_j h_1(x_j,y_j) & \cdots & \sum_j y_j h_n(x_j,y_j) \\ 1 & x_1 & y_1 & h_1(x_1,y_1) & \cdots & h_n(x_1,y_1) \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & y_n & h_1(x_n,y_n) & \cdots & h_n(x_n,y_n) \end{pmatrix} \cdot \begin{pmatrix} a \\ b \\ c \\ d_1 \\ \vdots \\ d_n \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ f_1 \\ \vdots \\ f_n \end{pmatrix}$$

5.3 Freiformmodellierung und Bézierkurven

Lokalität Eine lokale Veränderung der Eingabedaten (*von beispielsweise einem Stützwert*) darf *nach Möglichkeit* nur **lokale** Auswirkungen haben, idealerweise ändert sich die Kurvengestalt auch nur in einem kleinen Bereich um den Stützpunkt.

Bernsteinpolynome

$$B_i^n(x) = \binom{n}{i} \cdot (1-x)^{n-i} \cdot x^i$$

Die oben definierten Polynome spannen dann den Raum $\mathbb{P}_{n+1}$ auf:

$$\mathbb{P}_{n+1} = \operatorname{span}\left\{ B_i^n(x) \;\middle|\; 0 \leqslant i \leqslant n \right\}$$

Für die Funktionen gilt dann auch allgemein, dass für $t \in [0,1]$ $0 \leqslant B_i^n(t) \leqslant 1$ und $B_i^n(t)$ eine i-fache Nullstelle in $t = 0$, sowie eine $(n-i)$-fache in $t = 1$ hat. Ebenfalls gilt für alle t, dass $\sum_{i=0}^{n} B_i^n(t) = 1$.

Bézierkurven Gegeben seien eine Menge an **Kontrollpunkten** $\mathfrak{B} = \left\{ b_0, \ldots, b_n \right\} \subset \mathbb{R}^d$, so heißt die Kurve C im $\mathbb{R}^d$ **Bézierkurve** vom Grad n, wenn

$$C(t) = \sum_{i=0}^{n} b_i \cdot B_i^n(t),$$

wobei $t \in [0,1]$. Verbindet man die Kontrollpunkte durch einen Polygonzug, so erhält man das der Bézierkurve zugehörige **Kontrollpolygon**.

Formeigenschaften von Bézierkurven Die Geometrie des Kontrollpolygons spiegelt dann auch (grob) die Geometrie der Kurve wider. Wir erhalten

(1) **Interpolation der Endpunkte** Es muss gelten, dass

$$C(0) = b_0 \quad \text{und} \quad C(1) = b_n.$$

(2) **Tangentiale Tangentenbedingung** In den Endpunkten nähert sich die Kurve tangential an das Kontrollpolygon an, es gilt also

$$C'(0) = n(b_1 - b_0) \quad \text{und } C'(1) = n(b_n - b_{n-1})$$

(3) **Konvexe Hülle** Die Bézierkurve liegt in der **konvexen Hülle** der Kontrollpunkte. Die **konvexe Hülle** einer Menge M ist die kleinste Obermenge M' von M, welche ebenfalls konvex ist.

(4) **Affine Invarianz** Eine affine Abbildung ist mit $\Phi(x) = Ax + b$ gegeben, um eine Bézierkurve affin zu transformieren müssen **bloß die Kontrollpunkte affin transformiert** werden.

(5) **Variationsreduzierend** Für jede Gerade g gilt, dass die Anzahl an Schnittpunkten von g mit der Kurve kleiner oder gleich der Anzahl an Schnittpunkten von g mit dem Kontrollpolygon ist.

```
1 def de_casteljau(P, t):
2     # You will do this in exercise 7 ...
3     pass
```

Auswertung Wie oben bereits festgestellt ist die naive Auswertung der Bernsteinpolynome teuer und ineffizient; Abhilfe wird durch **Horner-Bézier** (*etwas effizienter*) oder durch **fortgesetzte lineare Interpolation mit *de Casteljau*** (*stabil, siehe rechts*) geschaffen.

5.4 Aufgaben

Aufgabe 5.1

Gegeben seien n Punkte (x_i, y_i) und ein Polynom $p(x)$ vom Grad $n-1$, welches die n Punkte interpoliert.

Teilaufgabe 5.1a

Wie lauten zu den gegebenen Stützstellen x_i gehörenden Newtonpolynome $q_j(x)$?

Teilaufgabe 5.1b

Das Polynom $p(x)$ kann man mit Hilfe der so angegebenen Newtonbasis darstellen. Geben Sie dabei eine Rekursionsformel zur Berechnung der Koeffizienten a_j an.

Teilaufgabe 5.1c

Bestimmen Sie die Koeffizienten des Newtoninterpolationspolynoms zu den folgenden Punkten und geben Sie anschließend das Newtoninterpolationspolynom an und werten Sie jenes Polynom an der Stelle $x = 2$ *effizient* aus.

x_i	0	1	3
y_i	1	-1	7

Aufgabe 5.2

Nennen Sie jeweils zwei Eigenschaften des Catmull-Rom- und Nearest-Neighbor-Interpolationsverfahrens.

Aufgabe 5.3

Gegeben seien nun folgende Punkte:

x_i	0	1	2	4
y_i	1	-1	1	2

Teilaufgabe 5.3a

Zeichnen Sie die Funktion $n(x) : [0.4] \to \mathbb{R}$, welche obige Werte, gemäß der *nearest-neighbor-interpolation* stückweise konstant interpoliert.

Teilaufgabe 5.3b

Berechnen Sie nun die Funktion $l(x) : [0,4] \to \mathbb{R}$, welche obige Werte stückweise linear interpoliert.

Aufgabe 5.4

Seien nun folgende Punkte gegeben:

x_i	0	2	4	6
y_i	-1	-3	-1	3

Geben Sie die Ableitungen m_1 und m_2 des Catmull-Rom-Interpolanten an den Stellen x_1 und x_2 an und skizzieren Sie den Interpolanten im Intervall $[x_1, x_2]$.

Aufgabe 5.5

Seien nun folgende Punkte gegeben:

i	0	1	2	3
x_i	0	1	2	4
y_i	3	5	-1	3

Bestimmen Sie die Lagrangepolynome und geben Sie die Koeffizienten des Interpolationspolynoms an.

Aufgabe 5.6

Gegeben sind die Eckpunkte eines Dreiecks $\Delta(R,S,T)$ mit $R = \begin{pmatrix}3 & 0\end{pmatrix}^T$, $S = \begin{pmatrix}3 & 6\end{pmatrix}^T$ und $T = \begin{pmatrix}0 & 3\end{pmatrix}^T$. Die Zuordnung der baryzentrischen Koordinaten zu den Eckpunkte ist mit $\rho \leftrightarrow R$, $\sigma \leftrightarrow S$ und $\tau \leftrightarrow T$ gegeben.

Teilaufgabe 5.6a

Berechnen Sie die zu den baryzentrischen Koordinaten ρ_i, σ_i und τ_i gehörenden Punkte P_i:

i	ρ_i	σ_i	τ_i
0	0	0	1
1	$2/3$	$1/3$	0
2	$1/3$	$1/2$	$1/6$

Teilaufgabe 5.6b

Die Funktionswerte an den Eckpunkten seien mit $f_R = 3$, $f_S = 6$ und $f_T = 4$ bekannt. Berechnen Sie die interpolierten Funktionswerte an den in Teilaufgabe 5.6a bestimmten Punkten!

Teilaufgabe 5.6c

Berechnen Sie nun die baryzentrischen Koordinaten $\begin{pmatrix}\rho_Q & \sigma_Q & \tau_Q\end{pmatrix}^T$ des Punktes $Q_1 = \begin{pmatrix}2 & 2\end{pmatrix}^T$ bzgl. des Dreiecks $\Delta(R,S,T)$.

Aufgabe 5.7

Betrachten Sie das Prisma mit dreieckigen Deckflächen $\Delta(R_0,S_0,T_0)$ und $\Delta(R_1,S_1,T_1)$, wobei

$$R_0 = \begin{pmatrix}0\\2\\-1\end{pmatrix}, \quad S_0 = \begin{pmatrix}3\\0\\-1\end{pmatrix}, \quad T_0 = \begin{pmatrix}3\\2\\-1\end{pmatrix}, \quad R_1 = \begin{pmatrix}0\\2\\2\end{pmatrix}, \quad S_1 = \begin{pmatrix}3\\0\\2\end{pmatrix} \text{ und } T_1 = \begin{pmatrix}3\\2\\2\end{pmatrix}$$

bekannt sind. Ebenso sind die Temperaturwerte in den Eckpunkten mit

$$f_{R_0} = 12, \quad f_{S_0} = 48, \quad f_{T_0} = 36, \quad f_{R_1} = 12, \quad f_{S_1} = 72, \quad f_{T_1} = 72$$

bekannt. Diese sollen ins Innere interpoliert werden. Ermitteln Sie den interpolierten Wert im Punkt $P = \begin{pmatrix}2 & 1 & 0\end{pmatrix}^T$, indem Sie zunächst entlang der senkrechten Kanten $[R_0R_1], [S_0S_1]$ und $[T_0T_1]$ linear interpolieren und dann die erhaltenen Werte f_R, f_S und f_T mit Hilfe baryzentrischer Koordinaten linear interpolieren.

Aufgabe 5.8

Teilaufgabe 5.8a

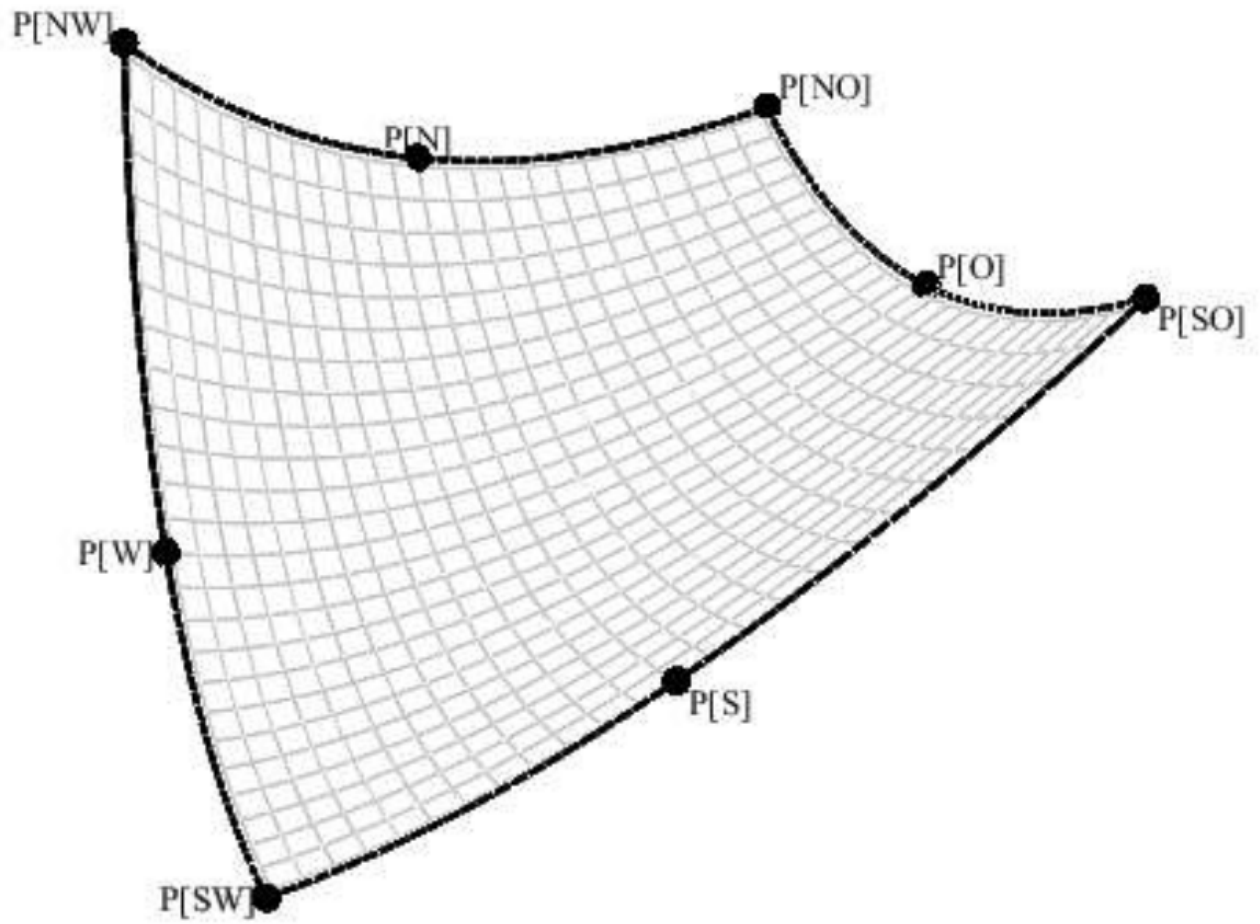
In den Ecken eines Recktecks $P_{0,0} = \begin{pmatrix}1 & 1\end{pmatrix}^T$, $P_{1,0} = \begin{pmatrix}5 & 1\end{pmatrix}^T$, $P_{1,1} = \begin{pmatrix}5 & 4\end{pmatrix}^T$ und $P_{0,1} = \begin{pmatrix}1 & 4\end{pmatrix}^T$ sind vier Werte $f_{00} = 10$, $f_{10} = 2$, $f_{11} = 3$ und $f_{01} = 1$ vorgegeben. Wir wollen diese nun bilinear interpolieren. Die Werte des Interpolanten in einem Punkt P kann man als gewichtete Summe schreiben

$$f_P = w_{00}^P \cdot f_{00} + w_{10}^P \cdot f_{10} + w_{11}^P \cdot f_{11} + w_{01}^P \cdot f_{01}.$$

Bestimmen Sie für den Mittelpunkt $M = \begin{pmatrix}3 & 5/2\end{pmatrix}^T$ und den Punkt $Q = \begin{pmatrix}2 & 2\end{pmatrix}^T$ die Gewichte und den bilinearen Interpolationswert.

Teilaufgabe 5.8b

Man betrachte das Coonspatch zu vier gegebenen Randkurven $C_S(s)$, $C_O(t)$, $C_N(s)$ und $C_W(t)$:



Der Mittelpunkt des Coonspatches, das heißt der Punkt $F(0.5,0.5)$, kann als Linearkombination der vier Kurvenmittelpunkte

$$P_S = C_S(1/2), P_O = C_)(1/2), P_N = C_N(1/2), P_W = C_W(1/2),$$

mit den vier Eckpunkten

$$P_{SW} = C_S(0) = C_W(0), P_{SO} = C_S(1) = C_O(0), P_{NO} = C_O(1) = C_N(1), P_{NW} = C_W(1) = C_N(0)$$

beschrieben werden. Es gilt also:

$$F(1/2, 1/2) = w_{SW}P_{SW} + w_SP_S + w_{SO}P_{SO} + w_OP_O + w_{NO}P_{NO} + w_NP_N + w_{NW}P_{NW} + w_WP_W$$

Bestimmen Sie nun die acht Gewichte $w_{SW}, ..., w_W$.

KAPITEL 6

Singulärwertzerlegung und Hauptkomponentenanalyse

6.1 Elemente der linearen Algebra

Definition 6.1 (*Norm — Wiederholung aus C1*)

Sei V ein $\mathbb{K}$-Vektorraum. Eine Abbildung $\|\cdot\| \in \mathrm{Abb}(V, \mathbb{K})$ heiße **Norm** genau dann, wenn für alle $v, u \in V$ und $\lambda \in \mathbb{K}$ die folgenden Eigenschaften erfüllt sind:

1. $\|v\| \geqslant 0$
2. $\|v\| = 0 \Leftrightarrow v = 0$
3. $\|\lambda v\| = |\lambda| \, \|v\|$
4. $\|v + u\| \leqslant \|v\| + \|u\|$

B Auf dem $\mathbb{R}^n$ gilt, dass für $1 \leqslant p < \infty$

$$\|x\|_p := \left(\sum_i |x_i|^p \right)^{1/p} \quad \text{und} \quad \|x\|_\infty := \max_i |x_i|$$

Normen definieren.

Definition 6.2 (*Matrixnorm*)

Seien V, W normierte Vektorräume und die Abbildung $F \in \mathrm{Abb}(V, W)$ linear (also $F \in L(V, W)$. Dann heiße

$$\|F\| = \sup_{v \in V \setminus \{0\}} \frac{\|Fv\|_W}{\|v\|_V} = \sup_{\|v\|_V = 1} \|Fv\|_W$$

die **Operator-** oder auch **Matrixnorm** von F. In endlichdimensionalen Gebieten mit kompakter Einheitskugel kann das obige Supremum auch durch ein Maximum ersetzt werden.

Definition 6.3 (*Spektralradius*)

Sei $A \in \mathbb{R}^{m \times m}$, so heiße

$$\rho(A) := \max_{i=1,\ldots,m} |\lambda_i(A)|,$$

wobei $\lambda_i(A)$ der i-te Eigenwert von A sei, der **Spektralradius** von A.

B

$\|\cdot\|_1$ — **Spaltensummennorm**

$$\|A\|_1 = \max_{\|x\|_1=1} \|Ax\|_1 = \max_{j=1,\dots,m} \sum_{i=1}^{m} |a_{ij}|$$

$\|\cdot\|_2$ — **Spektralnorm**

$$\|A\|_2 = \sup_{\|x\|_2=1} \|Ax\|_2 = \sqrt{\rho(A^H A)}$$

Dabei ergibt sich für ein invertierbares A der Zusammenhang $\|A\|_2 = \rho(A)$

$\|\cdot\|_\infty$ — **Zeilensummennorm**

$$\|A\|_\infty = \max_{i=1,\dots,m} \sum_{j=1}^{m} |a_{ij}|$$

Wir definieren noch eine weitere Eigenschaft von Matrizen, deren Diagonaldominanz.

Definition 6.4 (*Diagonaldominanz*)

Sei $A \in \mathbb{R}^{n\times n}$, so heiße A …

… **(zeilenweise) diagonaldominant** genau dann, wenn für alle $i \in \{1,\dots,n\}$

$$|a_{ii}| > \sum_{\substack{j=1 \\ j\neq i}}^{n} |a_{ij}|$$

gilt.

… **(zeilenweise) *schwach* diagonaldominant** genau dann, wenn für alle $i \in \{1,\dots,n\}$

$$|a_{ii}| \geqslant \sum_{\substack{j=1 \\ j\neq i}}^{n} |a_{ij}|$$

gilt.

Definition 6.5 (*Kondition*)

Sei A eine nicht singuläre Matrix, so ist die Kondition von A bzgl. einer Norm definiert als

$$\kappa(A) = \frac{\max\limits_{\|x\|=1} \|Ax\|}{\min\limits_{\|x\|=1} \|Ax\|}.$$

Ist A zudem noch quadratisch so folgt

$$\kappa(A) = \|A\| \cdot \left\|A^{-1}\right\|,$$

stets gilt damit, dass $\kappa(A) \geqslant \left|\frac{\lambda_{max}}{\lambda_{min}}\right|$.

Satz 6.1 (*Spektralsatz*)

Sei $A \in \mathbb{R}^{n\times n}$ eine reelle **symmetrische** Matrix, so gibt es eine **Orthonormalbasis** aus **Eigenvektoren**. Gleichwertig dazu ist die Existenz einer **Faktorisierung** $A = VDV^T$ mit $D = \text{diag}(\lambda_1, \ldots, \lambda_n)$, den Spalten der orthogonalen Matrix V als normierte Eigenvektoren von A.

6.2 Singular-value decomposition – SVD

Satz 6.2 (*Singulärwertzerlegung*)

Sei $A \in \mathbb{R}^{m\times n}$ eine beliebige Matrix, so gibt es eine Faktorisierung $A = U\Sigma V^T$.
Dabei gilt:

- $U \in \mathbb{R}^{m\times m}$ und $V \in \mathbb{R}^{n\times n}$ sind orthogonal
- Σ ist eine Diagonalmatrix mit $\Sigma_{ij} = 0$ für $i \neq j$ und $\Sigma_{11} \geqslant \Sigma_{22} \geqslant \cdots \geqslant 0$
- Die Spalten von U bzw. V sind die Eigenvektoren von AA^T bzw. A^TA
- $\sigma_i = \Sigma_{ii} \neq 0$ sind die Singulärwerte von A, genauer $\sigma_i = \sqrt{\lambda_i}$ wobei λ_i die Eigenwerte von AA^T oder A^TA sind.

Durch die Singulärwertzerlegung lassen sich folgende Werte der Matrix A einfach bestimmen:

- Der **Rang** entspricht genau der Anzahl der zu Null verschiedenen Singulärwerten.
  $\text{rang}(A) = r$ aus $\sigma_1 \geqslant \cdots \geqslant \sigma_r > 0$
- Der **Kern** enstpricht dem aufgespannten Raum aus den Zeilen V_i mit $i > r$.
  $\ker(A) = \text{span}\left\{V_{r+1}^T, \cdots, V_n^T\right\}$
- Das **Bild** entspricht dem aufgespannten Raum aus den Spalten U_i mit $i \leqslant r$.
  $\text{im}(A) = \text{span}\{U_1, \cdots, U_r\}$
- Die **Euklidsche Matrixnorm** entspricht dem größten Singulärwert.
  $|||A|||_2 = \sigma_1 = \sigma_{max}$
- Die **Konditionszahl** entspricht dem Quotienten aus größtem und kleinstem Singulärwert.
  $\kappa_2 = \frac{\sigma_1}{\sigma_r} = \frac{\sigma_{max}}{\sigma_{min}}$

Pseudo-Inverse Die Pseudoinverse A^{-1} ermöglicht die Annäherung an die tatsächliche Inverse von A falls diese nicht einfach zu bestimmen ist.

$$A^{-1} = V\Sigma^{-1}U^T \qquad \text{wobei} \quad \Sigma_{ii}^{-1} = 1/\Sigma_{ii}$$

Wichtig: A^{-1} und Σ^{-1} sind $n \times m$-Matrizen wie Σ^T und A^T!

$$x = A^{-1}b = \sum_{i=1}^{r} \frac{1}{\sigma_i}(u_i \circ b)\, v_i$$

$$A = \left(\begin{array}{c|c|c} u_1 & u_2 & u_3 \end{array}\right) \begin{pmatrix} \sigma_1 & 0 & 0 \\ 0 & \sigma_2 & 0 \\ 0 & 0 & \sigma_3 \end{pmatrix} \left(\begin{array}{c} v_1 \\ \hline v_2 \\ \hline v_3 \end{array}\right)$$

low-rank-approximation

Mit der SVD kann man eine Matrix durch Matrizen von niedrigerem Rang approximieren.
Dafür wird A_k vom Rang k gesucht, mit

$$A_k = \min_{X:\mathrm{rang}(X)=k} \|A - X\|_F$$

wobei A_k und $X \in \mathbb{R}^{m\times n}$ Matrizen.
Für die Frobenius-Norm gilt $\|A\|_F = \sqrt{\sum_{i=1}^{r} \sigma_i^2}$ (σ_i sind Singulärwerte von A)
mit Hilfe der SVD:

$$A_k = U\ \mathrm{diag}(\sigma_1, \cdots,\ \sigma_k,\ \underbrace{0, \cdots,\ 0}_{\substack{\text{setze } \sigma_{k+1} \\ \text{bis } \sigma_r \text{ Null}}})\ V^T$$

Was bringt die low-rank-approximation?

- Speichereffizienz: Nur $k \cdot (m+n)$ Werte statt $m \cdot n$
- Schnelle Berechnung von Matrix-Vektor-Multiplikation:
  $A_k x = \sum_{i=1}^{k} \sigma_i (v_i \circ x) u_i$ ergibt $k \cdot (2m+n)$ statt $2n \cdot m$ arithmetische Operationen

Approximationsfehler Es ist die beste Approximation bezüglich der Frobenius-Norm $\|\cdot\|_F$.
Es gelten:

$$\min_{X:\mathrm{rang}(X)=k} \|A - X\|_F^2 = \|A - A_k\|_F^2 = \sum_{j>k} \sigma_j^2$$
$$\min_{X:\mathrm{rang}(X)=k} \|A - X\|_2 = \|A - A_k\|_2 = \sigma_{k+1}$$

6.3 Principal component analysis – PCA

Problem 6.1

Große Datensätze sind oft schwierig analysierbar und haben nur selten hohe Aussagekraft. Die Hauptachsentransformation wirkt dem entgegen, da sie unter Beibehaltung der wesentlichen Merkmale hoch-dimensionale auf niedrig-dimensionale Probleme reduziert.

Schritte zur Bestimmung der PCA Seien Daten der Form $\{X_i\}_{1\leqslant i\leqslant N} \in \mathbb{R}^n$ gegeben.

- Balanciere die Daten um Mittelwert
  $\overline{X} = \frac{1}{N}\sum_i X_i \qquad \overline{X_i} = X_i - \overline{X}$
- Bilde Kovarianzmatrix (symmetrisch positiv semi-definit)
  $C = cov(X,X) = \frac{1}{N-1}\sum_i \overline{X_i} \cdot \overline{X_i}^T$
- Diagonalisierung der Kovarianzmatrix
  $$C = Q \begin{pmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & & \\ \vdots & & \ddots & \vdots \\ 0 & \lambda_2 & \cdots & \lambda_n \end{pmatrix} Q^T$$
  wobei Q orthogonal ist und $\lambda_1 \geqslant \lambda_2 \geqslant \cdots \geqslant \lambda_n \geqslant 0$
- Die Spalten q_i von Q sind die Eigenvektoren. Diese bezeichnet man als Hauptachsen.

6.4 Aufgaben

Aufgabe 6.1

Entscheiden Sie, ob die Division zweier Zahlen a und b (a/b) gut oder schlecht konditioniert ist und begründen Sie Ihre Aussage.

Aufgabe 6.2

Gegeben seien zwei Geraden durch die Gleichungen

$$G = \{(x,y) \mid 7x - 6y = 1\} \quad \text{und} \quad H = \{(x,y) \mid 6x - 5y = 1\},$$

welche sich im Punkt $P = (1,1)$ schneiden.

Teilaufgabe 6.2a

Ist das Schnittproblem bei Störung der Koeffizienten der Geradengleichung gut oder schlecht konditioniert?

Teilaufgabe 6.2b

Sei A die Matrix zur Bestimmung des Schnittpunktes $G \cap H$. Bestimmen Sie dann die Konditionszahlen dieser 2×2-Matrix bzgl. der Zeilensummennorm.

Teilaufgabe 6.2c

Sei $D = \operatorname{diag}(\lambda_1, \ldots, \lambda_n)$ eine gegebene Diagonalmatrix mit $|\lambda_{i-1}| \geqslant |\lambda_i| > 0$ für alle i. Bestimmen Sie die Konditionszahl der Matrix für eine von Ihnen gewählte Matrixnorm und geben Sie dabei vor allem an, welche Sie verwendet haben.

Aufgabe 6.3

Betrachten Sie die beiden folgenden Funktionen

$$f, g : \mathbb{R}^+ \to \mathbb{R}^+, f(x) = x^\pi \quad g(x) = x^{1/\pi},$$

welche beide mittels eines nich näher spezifizierten numerischen Verfahrens in der Nähe von 1 ausgewertet werden. Für welche Funktion ist dieses Problem besser konditioniert? Begründen Sie Ihre Antwort.

Aufgabe 6.4

Gegeben sei die Singulärwertzerlegung (SVD) der Matrix $A = U\Sigma V^T$:

$$A = \frac{1}{25} \cdot \begin{pmatrix} 36 & 27 & -20 \\ 30 & -40 & 0 \\ -48 & -36 & -15 \end{pmatrix} = \underbrace{\frac{1}{5} \cdot \begin{pmatrix} 3 & 0 & -4 \\ 0 & 5 & 0 \\ -4 & 0 & -3 \end{pmatrix}}_{=: U} \cdot \underbrace{\begin{pmatrix} 3 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{pmatrix}}_{=: \Sigma} \cdot \underbrace{\frac{1}{5} \cdot \begin{pmatrix} 4 & 3 & 0 \\ 3 & -2 & 0 \\ 0 & 0 & 5 \end{pmatrix}^T}_{=: V^T}$$

Teilaufgabe 6.4a

Bestimmen Sie für die Matrix A die Singulärwerte, den Rang, das Bild, den Kern, sowie die Konditionszahl bzgl. der Spektralnorm ($\|\cdot\|_2$)

Teilaufgabe 6.4b

Geben Sie die Definitionen der Spektralnorm, der Zeilensummennorm, der Spaltensummennorm und der Frobeniusnorm an und bestimmen Sie $\|A\|_2, \|A\|_\infty, \|A\|_1$ und $\|A\|_F$.

Teilaufgabe 6.4c

Bestimmen Sie x unter Verwendung der SVD, so dass gilt

$$A \cdot x = \begin{pmatrix} 9 \\ 0 \\ -12 \end{pmatrix}.$$

Teilaufgabe 6.4d

Bestimmen Sie die Matrix vom Rang 1, die A im Sinne der Frobeniusnorm am besten approximiert.

KAPITEL 7

ITERATIVE VERFAHREN

7.1 Allgemeine Theorie von iterativen Verfahren

Definition 7.1 (*Fixpunkt*)

Sei $M \neq \emptyset$ und $\Phi \in \mathrm{Abb}(M, M)$ eine Abbildung. Ein $x \in M$, das den Zusammenhang

$$\Phi(x) = x \tag{7.1}$$

erfülle heiße **Fixpunkt** von Φ.

Wir betrachten also eine Abbildung $\Phi : M \to M$. Sei $M \subseteq V$, wobei V ein normierter Vektorraum ist. Wir betrachten für einen Startwert $x_0 \in M$ eine rekursiv definierte Folge

$$x_{n+1} := \Phi(x_n), \tag{$*$}$$

und stellen uns die Frage, was der Grenzwert der Folge sei. Aussage darüber verschafft der folgende Satz:

Satz 7.1 (*Fixpunkt*)

Falls die oben definierte Folge $(x_n)_{n\in\mathbb{N}}$ konvergiert mit $\lim_n x_n := x_*$ und Φ stetig sei, so ist der Grenzwert x_* **immer ein Fixpunkt** von Φ.

$$\Phi(x_*) = x_*$$

Definition 7.2 (*Fixpunktiteration*)

Sei $\Phi \in \mathrm{Abb}(M, M)$ mit $M \subseteq V$, wobei V ein normierter Vektorraum ist und $x_0 \in M$. Die Berechnung

$$x_{n+1} := \Phi(x_n) \tag{$*$}$$

heiße **Fixpunktiteration**.

Die Fixpunktiteration hat ihren Namen daher, dass wenn Φ stetig ist und die Fixpunktiteration konvergiert, so der Grenzwert immer ein Fixpunkt von Φ ist. Wir wollen uns jetzt mit der Frage beschäftigen, wann Fixpunktiterationen überhaupt konvergieren. Ein wichtiges Kriterium dafür ist der Banach'sche Fixpunktsatz, auf den dieses Kapitel hinzielen möchte. Eine Voraussetzung dafür sind sogenannte Banachräume. Diese sollten aus C1 bekannt sein, wir wiederholen sie an dieser Stelle aber nocheinmal.

Definition 7.3 (*Vollständigkeit, Banachraum*)

Ein normierter $\mathbb{K}$-VR $(V, \|\cdot\|)$ heiße **vollständig** genau dann, wenn jede Cauchyfolge in V gegen ein Element in V konvergiert.
Ein Vektorraum heiße **Banachraum** genau dann, wenn er normiert und vollständig ist.
Ein Banachraum, dessen Norm durch ein Skalarprodukt erzeugt wird, heiße **Hilbertraum**.

Nun zurück zur Frage, wann und unter welchen Bedingungen Fixpunktiterationen konvergieren. Man kann sich *graphisch* überlegen, dass die „Steilheit“ von Φ dabei eine Rolle spielt. Wir definieren:

Definition 7.4 (*Kontraktion*)

Sei $(V, \|\cdot\|)$ ein $\mathbb{R}$-Vektorraum, $M \subseteq V$ und $\Phi \in \mathrm{Abb}(M, V)$. Wir sagen Φ heiße **Kontraktion** genau dann, wenn eine Konstante $0 \leqslant k < 1$ existiert, so dass für alle $x, y \in M$ gelte, dass

$$\|\Phi(x) - \Phi(y)\| \leqslant k\|x - y\|. \tag{K}$$

Eine Konstante k, welche diesen Zusammenhang erfüllt, heißt auch **Kontraktionskonstante** von Φ.

Anschaulich bedeutet es, wenn Φ eine Kontraktion ist, dass die Bilder von Φ näher beieinander liegen als die Urbilder von Φ.

Korrolar 7.4.1 (*Stetigkeit*)

Eine jede Kontraktion ist stetig.

Das in Definition 7.4 definierte Kriterium für Kontraktionen (K) ist nur sehr umständlich zu überprüfen. Folgender Satz vereinfacht dieses Kriterium, schränkt aber gleichzeitig auch unsere Auswahlmenge an Funktionen ein:

Satz 7.2 (*Kontraktionskriterium*)

Im Fall $V = \mathbb{R}$ ist für die Kontraktionseigenschaft hinreichend, dass f differenzierbar ist mit $k_* := \sup_{x \in M} |f'| < 1$. k_* ist dann **Kontraktionskonstante** von Φ.

Es ist sicherlich auch möglich Satz 7.2 auch auf $V = \mathbb{R}^n$ anzupassen, dann hieße die Bedingung $\sup_{x \in M} \|\mathcal{J}f(x)\| < 1$. In diesem Fall ist es aber notwendig zu spezifizieren, welche Matrixnorm man verwendet.

Wir kommen nun zu einer Art „Hauptsatz“ in unserem Kapitel über Fixpunktiterationen, dem Fixpunktsatz von Banach. Mit ihm wollen wir alles, was wir bislang über Fixpunkte und ihre Iterationen gelernt haben zusammenfassen um eine möglichst allgemeine Aussage über die Konvergenz von Fixpunktiterationen zu treffen.

Satz 7.3 (*Fixpunktsatz von* Banach)

Sei $(V, \|\cdot\|)$ ein **Banachraum** und sei $\emptyset \neq M \subseteq V$ eine **abgeschlossene Teilmenge** von V. Sei $\Phi \in \mathrm{Abb}(M, V)$ **selbstabbildend**, sprich $\Phi(M) \subseteq M$, sowie eine **Kontraktion**. Dann hat Φ genau einen Fixpunkt $x_* \in M$ und x_* ist Grenzwert der Folge $(*)$, wobei $x_0 \in M$ **beliebig** ist.
Sei k zudem noch eine Kontraktionskonstante von Φ, so fällt der Approximationsfehler in jedem Iterationsschritt um mindestens den Faktor k, also

$$\|x_{n+1} - x_*\| \leqslant k\|x_n - x_*\| \text{ damit auch } \|x_n - x_*\| \leqslant k^n \|x_0 - x_*\|$$

Satz 7.3 ist „**konstruktiv**“, damit erklärt er insbesondere auch den Weg zum Fixpunkt und nicht nur Existenz oder Eindeutigkeit.

Falls $M = V$, so ist sowohl die Eigenschaft der Abgeschlossenheit, als auch die Selbstabbildungseigenschaft **trivialerweise** erfüllt.

7.1.1 Newtonverfahren

Verfahren 7.1 (*Das Newton-Verfahren als Fixpunktiteration*)

Das Newton-Verfahren

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

zur Bestimmung einer Nullstelle x_* von f ist eine Fixpunktiteration, und zwar für die Funktion

$$\Phi(x) := x - \frac{f(x)}{f'(x)}.$$

Die asymptotische Kontraktionskonstante ist hierbei 0.

i Das Newton-Verfahren hat eine Fehlerreduktion der Form $\|x_{n+1} - x_*\| \approx c \cdot \|x_n - x_*\|^2$, sollte man $f \in \mathcal{C}^2$ und $f'(x_*) \neq 0$ voraussetzen.

Auch diese Erkenntnis haben wir im zweiten Semester aus dem Newton-Verfahren gezogen. Wir definieren deswegen ebenso:

Definition 7.5 (*Lineare und quadratische Konvergenz*)

Sei $(x_n) \subset (V, \|\cdot\|)$ eine Folge mit Grenzwert $x_* \in V$ und V sei normiert. Das Verfahren heiße …

… **linear konvergent** genau dann, wenn für ein $k < 1$ der Zusammenhang $|x_n - x_*| \leqslant k \cdot |x_{n-1} - x_*|$ erfüllt ist.

… **quadratisch konvergent** genau dann, wenn für ein $c > 0$ der Zusammenhang $\|x_n - x_*\| \approx c \cdot \|x_{n-1} - x_*\|^2$ erfüllt ist.

An dieser Stelle folgt nun ein abschließender Satz dieses Kapitels, hier an dieser Stelle allerdings ohne Beweis, eine Beweisidee sei mit dem Satz von Taylor genannt:

Satz 7.4 (*Hinreichendes Kriterium für quadratisches Konvergenzverhalten*)

Eine Fixpunktiteration sei mindestens **quadratisch konvergent** genau dann, wenn die asymptotische Kontraktionskonstante k_* gleich null it.

i Fixpunktiterationen sind also im Allgemeinen – sofern die Voraussetzungen des Satzes 7.3 erfüllt sind – **linear konvergent**, das heißt der Fehler reduziert sich pro Iterationsschritt mindestens um die Kontraktionskonstante k. Als gute Näherung für die Fehlerabschätzung kann – bei hinreichender Nähe zu x_* – auch die asymptotische Kontraktionskonstante k_* verwendet werden.

Verfahren 7.2 (*Newtonverfahren im* $\mathbb{R}^n$)

Sei $f \in \mathrm{Abb}(\mathbb{R}^n, \mathbb{R}^n) \in \mathcal{C}^2(D)$ und es sei die Jacobimatrix $\mathcal{J}f(x)$ invertierbar für alle $x \in \mathbb{R}^n$. So lautet das **Newtonverfahren** zur Bestimmung einer Nullstelle von f:

$$x_{n+1} := x_n - [\mathcal{J}f(x_n)]^{-1} \cdot f(x_n)$$

Das Verfahren ist – wie im skalaren Fall – **lokal quadratisch konvergent**.

Definition 7.6 (*Lokale quadratische Konvergenz*)

Es existiere eine Umgebung $U_\varepsilon(x_*)$ um x_* derart, dass wenn der Startwert $x_0 \in U_\varepsilon(x_*)$ aus ebendieser Umgebung gewählt sei, das Verfahren dann **quadratisch konvergent** heiße.
Lokal ist in diesem Fall dann tatsächlich als Umgebung zu verstehen.

Verfahren 7.3 (*„Effizienteres“ Newtonverfarhren im* $\mathbb{R}^n$)

(1) Löse das lineare Gleichungssystem $(\mathcal{J}f)(x_n) \cdot \Delta x = -f(x_n)$

(2) Setze $x_{n+1} := x_n + \Delta x$

7.1.2 Sekantenverfahren

Im Gegensatz zum Newtonverfahren ist hierbei keine Kenntnis der Ableitungen notwendig.

Verfahren 7.4 (*Sekantenverfahren*)

Sei $f \in \mathrm{Abb}(\mathbb{R}, \mathbb{R})$ und zwei Startwerte x_0 und x_1. Bestimme dann den Schnittpunkt der Sekante durch $(x_0, f(x_0))$ und $(x_1, f(x_1))$ mit der x-Achse:

$$x_{n+1} := \frac{x_{i-1} \cdot f(x_i) - x_i \cdot f(x_{i-1})}{f(x_i) - f(x_{i-1})}$$

Das Verfahren hat die Konvergenzordnung $p = \frac{\sqrt{5}+1}{2}$.

7.1.3 Bisektionsverfahren

Verfahren 7.5 (*Bisektionsverfahren*)

Sei $f \in \mathrm{Abb}(\mathbb{R}, \mathbb{R})$ und zwei Startwerte x_0 und x_1 mit $f(x_0) \cdot f(x_1) < 0$ gegeben, so gibt es im Intervall $[x_0, x_1]$ mindestens eine Nullstelle.
Bestimme damit den Mittelpunkt $x_{i+1} = \frac{1}{2} \cdot (x_i + x_{i-1})$ und betrachte danach das Intervall $[x_{i-1}, x_{i+1}]$ falls $f(x_{i-1}) \cdot f(x_{i+1}) < 0$ oder $[x_{i+1}, x_i]$ falls $f(x_{i+1}) \cdot f(x_i) < 0$.
Das Verfahren hat gerade noch so eine lineare Konvergenzordnung.

7.1.4 Regula falsi

Im Prinzip eine Mischung aus Bisektions- und Sekantenverfahren, bei dem man wie beim Bisektionsverfahren mit zwei Punkten startet, danach aber *wie beim Sekantenverfahren* den *Schnittpunkt der Sekante mit der x-Achse* bestimmt um anschließend wieder das neue Intervall zu bestimmen. Konvergenzordnung ähnlich des Bisektionsverfahrens.

7.2 Jacobi-Verfahren

Verfahren 7.6 (*Jacobi-Verfahren / Gesamtschrittverfahren*)

Gegeben sei ein lineares Gleichungssystem $Ax = b$ mit $A = (a_ij) \in \mathbb{R}^{\ltimes} \times \ltimes$ mit $n \in \mathbb{N}$. Seien alle Diagonaleinträge a_{ii} ungleich 0. Wir konstruieren uns gemäß dem Beispiel oben (**??**) ein $\Phi(x) := -D^{-1}Rx + D^{-1}b$. Die Fixpunktiteration $x_{m+1} := \Phi(x_m)$ mit dem so konstruierten Φ heiße dann **Jacobi**- oder auch **Gesamtschrittverfahren**. Ein Iterationsschritt sehe dabei wie folgt für alle $i = 1, \ldots, n$ aus:

$$(x_{m+1})_i := \frac{1}{a_{ii}} \left(b_i - \sum_{j \in \{1,\ldots,n\} \setminus \{i\}} a_{ij} (x_m)_j \right)$$

Satz 7.5 (*Hinreichendes Konvergenzkriterium*)

Sei Φ wie in Definition 7.1, dann ist Φ eine Kontraktion bezüglich der $\|\cdot\|_\infty$-Norm auf ganz $\mathbb{R}^n$, falls die Systemmatrix A **diagonaldominant** ist. Damit konvergiert Verfahren 7.6.
Sei A unzerlegbar und schwach diagonaldominant, so konvergieren ebenfalls alle Verfahren. Eine Matrix A heißt unzerlegbar, wenn ihr Graph **stark zusammenhängend** ist.

Sei A in eine strikte untere Dreiecksmatrix L, eine Diagonalmatrix D und eine strikte obere Dreiecksmatrix R aufgeteilt, so ist die Iterationsmatrix von A bestimmt durch

$$V_J = -D^{-1}(L+R).$$

! Hierbei werden neuere, bessere Werte ignoriert, das Verfahren ist dafür allerdings **besser** parallelisierbar.

7.3 Gauss-Seidel-Verfahren

Verfahren 7.7 (*Gauß-Seidel-Verfahren*)

Gegeben sei ein lineares Glechungssystem $Ax = b$ mit $A = (a_ij) \in \mathbb{R}^{\kappa} \times \kappa$ mit $n \in \mathbb{N}$. Seien des Weiteren alle Diagonaleinträge a_{ii} ungleich 0. Ein Verfahren heiße **Gauß-Seidel**- oder auch **Einzelschrittverfahren** genau dann, wenn es zur Berechnung der i-ten Komponente des neuen Vektors die Iterationsvorschrift

$$x_{m+1,i} := \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} \mathbf{x_{m+1,j}} - \sum_{j=i+1}^{n} a_{ij} x_{m,j} \right),$$

bei der der Unterschied zum Jacobiverfahren (7.6) **blau** hervorgehoben wurde, verwendet.

Sei A wieder in eine strikte untere Dreiecksmatrix L, eine Diagonalmatrix D und eine strikte obere Dreiecksmatrix R aufgeteilt, so ist die Iterationsmatrix von A bestimmt durch

$$V_{GS} = -(L+D)^{-1}R.$$

Anmerkungen

i Das Gauß-Seidel-Verfahren konvergiert im Allgemeinen (bei tridiagonalen Matrizen) um den Faktor **zwei schneller** als das Jacobiverfahren. Einen Beweis dieses Satzes findet sich in Plato, Numerische Mathematik kompakt mit Korollar 10.42.

Das Gauß-Seidel-Verfahren lässt sich platzsparender implementieren, bei der Berechnung von $x_{m+1,i}$ ist $x_{m,i}$ überschreibbar, der Wert wird nicht mehr gebraucht.

7.4 SOR-Verfahren

Das Gauß-Seidel-Verfahren kann man relaxiert auffassen, man nennt dies dann sukzessiv überrelaxiert „*successive overrelaxion*“ (SOR). Die Idee hierbei ist den Zielwert entweder **bewusst** zu „überschießen“ oder **bewusst** das „Überschießen“ abzuschwächen. Auch hier gilt dann $\omega \in (0,2)$, sonst divergiert das Verfahren. Man erhält dann:

$$x_n = H_\omega x_{n-1} + \omega(D+\omega L)^{-1} b,$$

wobei H_ω definiert ist als

$$H_\omega := (D + \omega L)^{-1} [(1-\omega)D - \omega R] = E_n - \omega(D + \omega L)^{-1} A.$$

Verfahren 7.8 (*SOR-Verfahren*)

Gegeben sei ein lineares Gleichungssystem $Ax = b$ mit $A = (a_ij) \in \mathbb{R}^{\ltimes} \times \ltimes$ mit $n \in \mathbb{N}$. Seien des Weiteren alle Diagonaleinträge a_{ii} ungleich 0. Ein Verfahren heiße **SOR-Verfahren** genau dann, wenn es zur Berechnung der i-ten Komponente des neuen Vektors die Iterationsvorschrift

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

verwendet.

Sei A wieder in eine strikte untere Dreiecksmatrix L, eine Diagonalmatrix D und eine strikte obere Dreiecksmatrix R aufgeteilt, so ist die Iterationsmatrix von A bestimmt durch

$$V_{SOR} = -\left(\frac{1}{\omega} D + L \right)^{-1} \left(\frac{\omega - 1}{\omega} D + R \right).$$

Im Vergleich zu der Komplexität von $\mathcal{O}(n)$ bei Jacobi und Gauß-Seidel kommt es bei SOR-Verfahren mit einem optimalen Parameter zu einer $\mathcal{O}(\sqrt{n})$ Komplexität.

KAPITEL 8

NICHTLINEARE OPTIMIERUNG

Satz 8.1 (*Lagrange-Multiplikatoren bei mehreren Nebenbedingungen*)

Sei $f : D \subseteq \mathbb{R}^n \to \mathbb{R}$, $g : D \to \mathbb{R}^m$ mit $m \leqslant n$. Seien $f, g \in \mathcal{C}^1(D)$. Ferner besitze f in ξ unter der Nebenbedingung $g(\xi) = 0$ eine Extremstelle. Dann ist eine **notwendige** Bedingung für die Existenz dieser Extremstelle, dass ...

(I) $\nabla f(x_0) = \sum_{i=1}^{m} \lambda_i \cdot \nabla g_i(\xi)$

(II) $g(\xi) = 0$

$\lambda \in \mathbb{R}^m$ beschreibt wieder den Lagrangemultiplikator, das dazugehörende Funktional ist $\mathcal{L}(\xi, \lambda) := f(\xi) - \langle \lambda, g(\xi) \rangle$.

8.1 Optimierung mittels Newtonverfahren

Verfahren 8.1 (*Newtonverfahren zur Optimierung*)

Sei $f \in \mathrm{Abb}(\mathbb{R}^n, \mathbb{R})$ und es sei die Hessematrix $\mathcal{H}f(x)$ invertierbar für alle $x \in \mathbb{R}^n$. So lautet das **Newtonverfahren** zur Bestimmung eines Optimums von f:

$$x_{n+1} := x_n - [\mathcal{H}f(x_n)]^{-1} \cdot \nabla f(x_n)$$

Das Problem ist der hohe Aufwand beim Berechnen der $n^2/2$ vielen zweiten Ableitungen.

8.2 Abstiegsverfahren

Verfahren 8.2 (*Abstiegsverfahren*)

Sei ein Startwert $x_0 \in \mathbb{R}^n$ sowie eine Startsuchrichtung $s_0 \in \mathbb{R}^n$ gegeben. Dann heißt die Iterationsvorschrift

$$x_{n+1} = x_n + \alpha_n \cdot s_n,$$

mit $\alpha_n > 0$ als Suchweite und s_0 als Suchrichtung **Abstiegsverfahren**, falls $s_n \circ \nabla f(x_n) < 0$ gilt.

Die optimale Schrittweite kann man durch das näherungsweise Lösen des eindimensionalen Minimierungsproblems

$$t_n = \arg\min_{t>0} \left\{ f(x_n + t \cdot s_n) \right\}$$

bestimmen, bei einem Newtonverfahren ist $t_n = 1$ zu wählen, will man es besser dämpfen so auch $t_n < 1$. Ansonsten ist das nachfolgende Verfahren gut für Schrittweitenwahlen geeignet.

8.2.1 Abstiegsverfahren des goldenen Schnittes

Verfahren 8.3 (*Verfahren des goldenen Schrittes*)

Sei $f \in \mathrm{Abb}(\mathbb{R},\mathbb{R})$ und drei Startwerte x_0, x_1 und x_2 mit $f(x_0) > f(x_1) < f(x_2)$ gegeben. Wähle dann einen neuen Wert x_n mit $x_{n-3} < x_n < x_{n-1}$. Gilt nun $f(x_n) < f(x_2)$, so bildet man das neue Tripel (x_{n-2}, x_n, x_{n-1}), andernfalls mit (x_n, x_{n-2}, x_{n-1}).
d sollte dabei im **goldenen Schnitt** gewählt worden sein. $(\frac{\sqrt{5}+1}{2})$

8.2.2 Auffinden der Suchrichtung

Newtonverfahren Das Newtonverfahren wird zum Abstiegsverfahren, wenn die Hessematrix positiv definit ist.

Gradientenverfahren

Verfahren 8.4 (*Gradienten-Verfahren*)

Sei f „glatt“ und x_n eine hinreichende Näherung an die Minimalstelle von f. Dann ist

$$x_{n+1} := x_n - \alpha_n \nabla f(x_n) \text{ mit bspw. } \alpha_n := \frac{\langle \nabla f(x_n), \nabla f(x_n)\rangle}{\langle A\nabla f(x_n), \nabla f(x_n)\rangle}$$

im Allgemeinen eine bessere Näherung. Es gilt dann also $s_n = -\nabla f(x_n)$.

Verfahren der konjugierten Gradienten (*cg*-Verfahren) Ein **quadratisches Funktional** ist von der Form

$$F(x) = x^T A x + 2b^T x + c,$$

wobei A eine **positiv definite** quadratische Matrix, b und x Vektoren und c ein Skalar ist. Man nennt dann zwei Vektoren u und v A-***konjungiert***, wenn

$$u^T A v = 0.$$

Satz 8.2 (*Schritteanzahl beim cg-Verfahren*)

Wählt man als Suchrichtungen (s_i) eine Folge an paarweisen konjungierter Richtungen und dazu optimale Schrittweiten t_i, so ist man nach (spätestens) n Schritten im Minimum.

Verfahren 8.5 (*Verfahren der konjugierten Gradienten*)

(1) Wähle x^1 beliebig und berechne

$$g^1 = b + Ax^1 \quad , \quad s^1 = -g^1 \quad \text{und} \quad \alpha^1 = \frac{g^1 \circ g^1}{s^1 \circ As^1}.$$

(2) **Für** $k \in \{1,2,\ldots,n\}$ **tue**

(2.1) **Falls** $\|g^{k+1}\| < \varepsilon$ **halte an.**

(2.2) Bestimme von x^k in Richtung s^k das Minimum $x^{k+1} = x^k + \alpha^k s^k$.

(2.3) Aktualisiere dann den Gradienten durch $g^{k+1} = g^k + \alpha^k A s^k$

(2.4) Aktualisiere dann die Suchrichtung, so dass die neue Suchrichtung A-konjugiert zu allen bisherigen ist:

$$s^{k+1} = -g^{k+1} + \beta^k \cdot s^k \quad \text{mit} \quad \beta^k = \frac{g^{k+1} \circ g^{k+1}}{g^k \circ g^k}$$

(2.5) Bestimme die neue optimale Schrittweite α^{k+1} mit

$$\alpha^{k+1} = \frac{g^{k+1} \circ g^{k+1}}{s^{k+1} \circ A s^{k+1}}$$

Dieses Verfahren kann auch zur Lösung linearer Gleichungssysteme mit **positiv definiter Koeffizientenmatrix** verwendet werden.

Verfahren 8.6 (*Verfahren der konjugierten Gradienten zum Lösen allgemeiner LGS*)

(1) Wähle x^1 beliebig und berechne

$$g^1 = \nabla f(x^0) \quad , \quad s^1 = -g^1 \quad \text{und} \quad \alpha^1 = \arg\min_t \left\{ f\left(x^0 + t s^0\right) \right\}.$$

(2) **Für** $k \in \left\{1, 2, \ldots, n\right\}$ **tue**

(2.1) **Falls** $\left\|g^{k+1}\right\| < \varepsilon$ **halte an.**

(2.2) Bestimme von x^k in Richtung s^k das Minimum $x^{k+1} = x^k + \alpha^k s^k$.

(2.3) Aktualisiere dann den Gradienten durch $g^{k+1} = \nabla f(x^{k+1})$

(2.4) Aktualisiere dann die Suchrichtung, so dass die neue Suchrichtung A-konjugiert zu allen bisherigen ist: $s^{k+1} = -g^{k+1} + \beta^k \cdot s^k$ mit

$$\beta^k = \frac{g^{k+1} \circ g^{k+1}}{g^k \circ g^k} (\textit{Fletcher/Reeves}) \quad \text{oder} \quad \beta^k = \frac{g^{k+1} \circ (g^{k+1} - g^k)}{g^k \circ g^k} (\textit{Pollack/Ribière})$$

(2.5) Bestimme die neue optimale Schrittweite α^{k+1} mit

$$\alpha^{k+1} = \arg\min_t \left\{ f\left(x^{k+1} + t s^{k+1}\right) \right\}$$

Quasinewtonverfahren Wir wählen an der Hessematrix statt eine Folge positiv definiter Matrizen $(H_k)_{k \in \mathbb{N}}$, mit einer jeder die Bedingung

$$x_k - x_{k-1} = H_k(\nabla f(x_k) - \nabla f(x_{k-1}))$$

erfüllt ist. Diese Bedingung heißt dann **Quasinewtonbedingung**. Der Vorteil liegt in der erhaltenen superlinearen Konvergenz des Verfahrens.

KAPITEL 9

Numerische Integration

9.1 Monte-Carlo

9.2 Newton-Cotes-Formeln

Das Kapitel über die Newton-Cotes-Formeln befasst sich mit dem Sachverhalt, dass sich Polynome in Abhängigkeit dessen Grades exakt integrieren lassen.

Die grundsätzliche Idee dahinter sehe wie folgt aus:
Man bestimme zu einer gegebenen Funktion ***f(x)*** das Polynom ***p(x)*** durch die folglich vorgestellten Regeln, welches die Funktionswerte an den äquidistanten Stellen interpoliert und nehme somit als Näherungswert

$$\int_a^b p(x)dx \text{ anstelle vom ursprünglichen } \int_a^b f(x)dx$$

Hierbei gilt es zu beachten, dass zu einer Funktion ***f(x)*** mit **n+1** gegebenen Punkten

$$(x_0, f(x_0)), (x_1, f(x_1)), \ldots, (x_n, f(x_n))$$

genau ein interpolierendes Polynom vom Grad **n** existiert.

9.2.1 Mittelpunktsregel

Es sei ein Punkt $(x_0, f(x_0))$ mit $x_0 = \frac{a+b}{2}$ gegeben.

$$\hookrightarrow \textbf{\#Punkte = 1} \Leftrightarrow \textbf{Polynom Grad = 0}$$

Somit gilt für unser zu interpolierende Polynom:

$$p_0(x) = f(x_0) = \text{const}$$

$$\hookrightarrow \int_a^b f(x)dx \approx \int_a^b p_0(x)dx = (b-a)f(x_0)$$

Dies ist auch als Rechtecksregel bekannt, da wir im Prinzip das Integral durch ein Rechteck approximieren.

Zusammengefasst:

- $x_0 = a$: **linke Rechtecksregel**
- $x_0 = \frac{a+b}{2}$: **Mittelpunktsregel**
- $x_0 = b$: **rechte Rechtecksregel**

9.2.2 Trapezregel

Es seien zwei Punkte $(x_0, f(x_0))$, $(x_1, f(x_1))$ mit $x_0 = a$, $x_1 = b$ gegeben.

$$\hookrightarrow \textbf{\#Punkte = 2} \Leftrightarrow \textbf{Polynom Grad = 1}$$

Somit gilt für unser zu interpolierende Polynom:

$$p_1(x) = \frac{b-x}{b-a} f(x_0) + \frac{x-a}{b-a} f(x_1)$$

$$\hookrightarrow \int_a^b f(x)dx \approx \int_a^b p_1(x)dx = (b-a)\frac{f(a)+f(b)}{2}$$

9.2.3 Simpsonregel

Es seien drei Punkte $(x_0, f(x_0))$, $(x_1, f(x_1))$, $(x_2, f(x_2))$ mit $x_0 = a$, $x_1 = \frac{a+b}{2}$, $x_2 = b$ gegeben.

$$\hookrightarrow \textbf{\#Punkte = 3} \Leftrightarrow \textbf{Polynom Grad = 2}$$

Somit gilt für unser zu interpolierende Polynom (analog zu Lagrange):

$$p_2(x) = \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} f(x_0) + \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)} f(x_1) + \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)} f(x_2)$$

$$\hookrightarrow \int_a^b f(x)dx \approx \int_a^b p_2(x)dx = (b-a)(\frac{1}{6}f(x_0) + \frac{4}{6}f(x_1) + \frac{1}{6}f(x_2))$$

ANHANG A

Tabelle der Algorithmenaufwände und Fehlerabschätzungen

Im Folgenden sei dann nur noch eine kleine Übersicht über die verschiedenen Aufwände und Fehlerabschätzungen einiger Verfahren und Algorithmen gegeben.

A.1 Aufwände

	Algorithmus/Verfahren	Komplexität
Allgemeine Lineare Algebra	Matrix-Vektor-Multiplikation Ab mit vollbesetzter Matrix A	$\mathcal{O}(n^2)$
	Matrix-Vektor-Multiplikation Ab mit m-diagonaler Matrix A	$\mathcal{O}(m \cdot n)$
	Matrix-Vektor-Multiplikation Ab mit Rang-1-Matrix und gegebenem $A = uv^T$	$\mathcal{O}(n)$
	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 vollbesetzter Matrix A und m-diagonaler Matrix B	$\mathcal{O}(m \cdot n^2)$
	Berechnung der Inversen einer invertiarbaren $(n \times n)$-Diagonalmatrix.	$\mathcal{O}(n)$
	Rang einer invertierbaren $n \times n$-Matrix	n
Direkte Verfahren	Bestimmung der LR- bzw. QR-Zerlegung	$\mathcal{O}(n^3)$
	Bestimmung LR-Zerlegung von m-diagonaler Matrix	$\mathcal{O}(m^2 \cdot n)$
	Unterscheidungsfaktor zwischen LR-Zerlegung und Householderspiegelungen	$q_{HR} \approx 2 \cdot q_{LR}$
	Unterscheidungsfaktor zwischen LR-Zerlegung und Givensrotationen	$q_{GR} \approx 4 \cdot q_{LR}$
	Unterscheidungsfaktor zwischen LR-Zerlegung und Cholesky	$q_{Ch} \approx \frac{1}{2} \cdot q_{LR}$
	Lösen des Gleichungssystems mit geg. LR/QR/SVD-Zerlegung	$\mathcal{O}(n^2)$
	Lösen des Gleichungssystems mit geg. LR/QR-Zerlegung bei m-diagonaler Matrix	$\mathcal{O}(n \cdot m)$
	Vorwärts- und Rückwärtssubstitutionen	$\mathcal{O}(n^2)$
	Bestimmung der Determinante bei geg. LR-Zerlegung	$\mathcal{O}(n)$
	Bestimmung der Betragsdeterminante bei geg. PLR-Zerlegung	$\mathcal{O}(n)$
	Bestimmung der Determinante bei geg. PLR-Zerlegung	$\mathcal{O}(n^2)$
	Bestimmung der Betragsdeterminante bei geg. QR-Zerlegung	$\mathcal{O}(n)$

(*Fortgesetzt auf nächster Seite* ↪)

(*Aufwandstabelle von vorheriger Seite fortgesetzt*)

	Algorithmus/Verfahren	Komplexität
Faltung	Anzahl der arithmetischen Operationen bei einem nicht separierbaren $M \times M$ Filter bei einem $N \times N$ Bildausschnitt	$2 \cdot N^2 \cdot M^2$
	Anzahl der arithmetischen Operationen bei einem separierbaren $M \times M$ Filter bei einem $N \times N$ Bildausschnitt	$4 \cdot N^2 \cdot M$
	Berechnung der Fourier-Transformation *ohne* FFT	$\mathcal{O}\left(n^2\right)$
	Berechnung der Fourier-Transformation *mit* FFT	$\mathcal{O}\left(n \cdot \log n\right)$
Bézier	Auswerten eines Punktes auf einer Bézierkurve mittels De Casteljau	$\mathcal{O}\left(n^2\right)$
	Anzahl der Kontrollkurve einer Bézierkurve mit Grad n	$n+1$
	Grad einer Bézierkurve mit n Kontrollpunkten	$n-1$
Optimierung	Aufwand eines Iterationsschrittes bei Jacobi,Gauss-Seidel oder SOR	$\mathcal{O}\left(n^2\right)$
	Anzahl der Iterationsschritte bei Jacobi oder Gauss-Seidel	$\mathcal{O}\left(n\right)$
	Anzahl der Iterationsschritte bei SOR	$\mathcal{O}\left(\sqrt{n}\right)$
	Konvergenzordnung bei Jacobi,Gauss-Seidel oder SOR	$\rho = 1$
	Bestimmung der Koeffizienten bei Interpolation mit Newtonbasis	$\mathcal{O}\left(n^2\right)$
	Sei A eine dünnbesetzte $(n \times n)$-Matrix mit höchstens m von Null verschiedenen Einträgen pro Zeile. Welche Komplexität hat die Berechnung von k Iterationsschritten des Jacobi/Gauß-Seidel/SOR-Verfahrens?	$\mathcal{O}\left(k \cdot (n \cdot m)\right)$

A.2 Fehlerabschätzungen

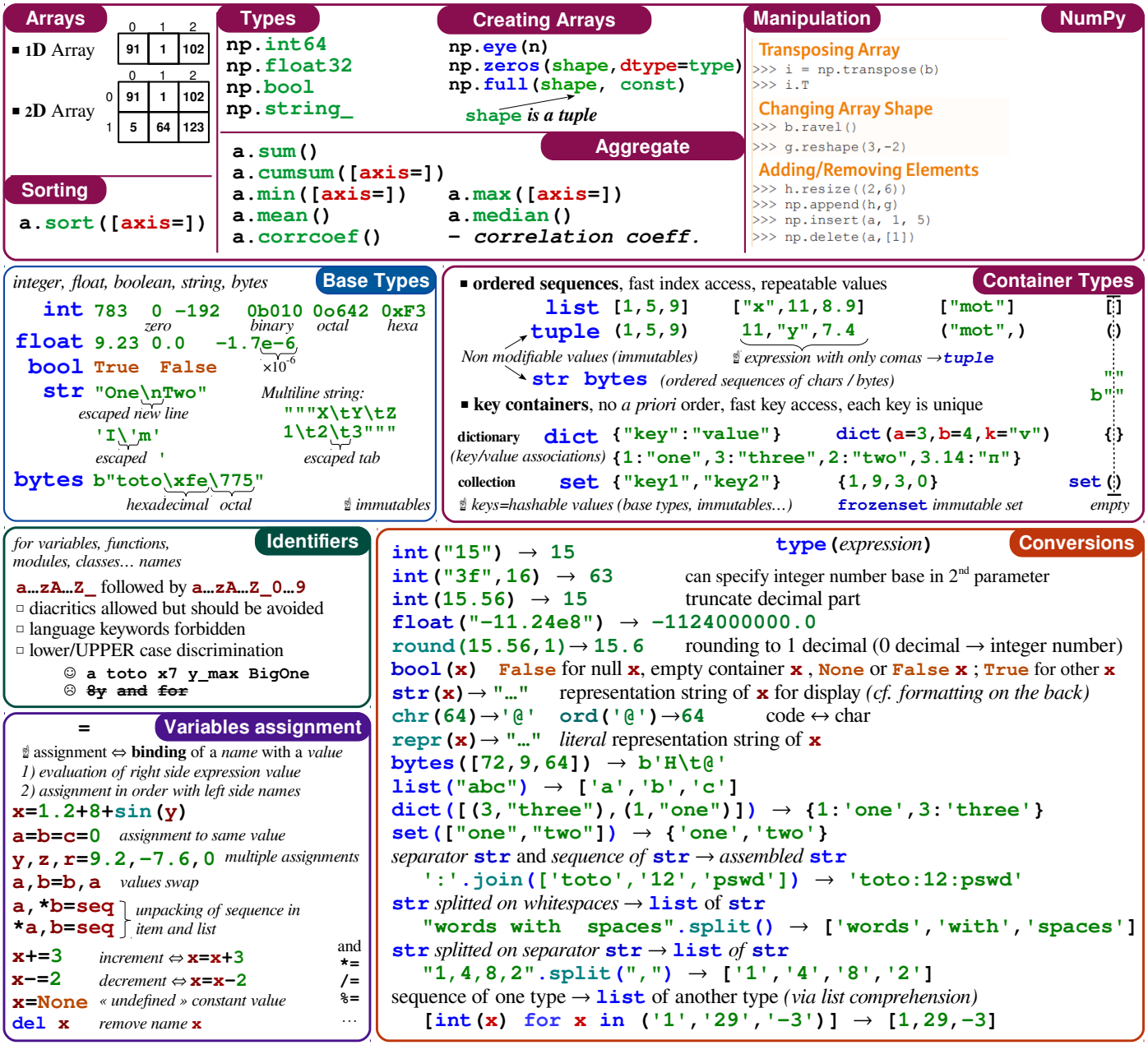
Verfahren	Fehlerabschätzung
Stückweise konstante Interpolation (Nearest Neigbor)	$\mathcal{O}\left(h\right)$
Stückweise lineare Interpolation	$\mathcal{O}\left(h^2\right)$
Stückweise kubische Interpolation (Catmull-Rom)	$\mathcal{O}\left(h^3\right)$
Midpoint Subdivision	$\mathcal{O}\left(\frac{1}{(\text{levelzahl})^4}\right)$
(Iterierte) Trapezregel	$\mathcal{O}\left(h^2\right)$
(Iterierte) Simpsonregel	$\mathcal{O}\left(h^4\right)$

ANHANG B

Python 3 — Zusammenfassung und Kodeschnipsel

B.1 Zusammenfassung der wichtigsten Befehle

Nachfolgend kommt eine kleiner Zusammenschnitt der elementarsten und wichtigsten Befehlskombination, welche Python 3 zu bieten hat.



Sequence Containers Indexing

for lists, tuples, strings, bytes…

```
negative index   -5  -4  -3  -2  -1
positive index    0   1   2   3   4
lst=[10,  20,  30,  40,  50]
positive slice    0   1   2   3   4   5
negative slice   -5  -4  -3  -2  -1
```

Items count
`len(lst)→5`

index from 0
(here from 0 to 4)

Individual access to **items** via `lst[`*index*`]`
`lst[0]→10` ⇒ *first one* `lst[1]→20`
`lst[-1]→50` ⇒ *last one* `lst[-2]→40`
On mutable sequences (`list`), remove with `del lst[3]` *and modify with assignment* `lst[4]=25`

Access to **sub-sequences** via `lst[`*start slice*`:`*end slice*`:`*step*`]`
`lst[:-1]→[10,20,30,40]` `lst[::-1]→[50,40,30,20,10]` `lst[1:3]→[20,30]` `lst[:3]→[10,20,30]`
`lst[1:-1]→[20,30,40]` `lst[::-2]→[50,30,10]` `lst[-3:-1]→[30,40]` `lst[3:]→[40,50]`
`lst[::2]→[10,30,50]` `lst[:]→[10,20,30,40,50]` *shallow copy of sequence*
Missing slice indication → from start / up to end.
On mutable sequences (`list`), remove with `del lst[3:5]` *and modify with assignment* `lst[1:4]=[15,25]`

Boolean Logic

Comparisons : `<` `>` `<=` `>=` `==` `!=` (*boolean results*) ≤ ≥ = ≠

`a and b` logical and — *both simultaneously*
`a or b` logical or — *one or other or both*

pitfall : `and` *and* `or` *return value of* `a` *or of* `b` *(under shortcut evaluation).*
⇒ *ensure that* `a` *and* `b` *are booleans.*

`not a` logical not

`True` `False` True and False constants

Statements Blocks

parent statement:
→ *statement block 1…*
⋮
parent statement:
→ *statement block2…*
⋮
next statement after block 1

indentation !

configure editor to insert 4 spaces in place of an indentation tab.

Modules/Names Imports

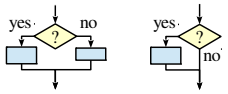
module `truc` ⇔ *file* `truc.py`

```
from monmod import nom1,nom2 as fct
```
→*direct access to names, renaming with* `as`
`import monmod` →*access via* `monmod.nom1` …
modules and packages searched in *python path* (cf `sys.path`)

Conditional Statement

statement block executed only ***if*** *a condition is true*

`if` *logical condition*`:`
→ *statements block*



Can go with several *elif*, *elif*... and only one final *else*. Only the block of first true condition is executed.

```
if age<=18:
    state="Kid"
elif age>65:
    state="Retired"
else:
    state="Active"
```

with a var `x`:
`if bool(x)==True:` ⇔ `if x:`
`if bool(x)==False:` ⇔ `if not x:`

Maths

floating numbers… approximated values — *angles in radians*

Operators: `+ - * / // % **`
Priority `(…)` × ÷ ↑ ↑ a^b
integer ÷ ÷ remainder
`@` → matrix × *python3.5+*`numpy`

```
(1+5.3)*2→12.6
abs(-3.2)→3.2
round(3.57,1)→3.6
pow(4,3)→64.0
```

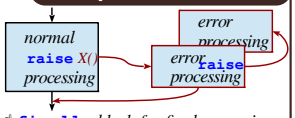
usual order of operations

```
from math import sin,pi…
sin(pi/4)→0.707…
cos(2*pi/3)→-0.4999…
sqrt(81)→9.0 ✓
log(e**2)→2.0
ceil(12.5)→13
floor(12.5)→12
```

modules `math`, `statistics`, `random`, `decimal`, `fractions`, `numpy`, *etc. (cf. doc)*

Exceptions on Errors

Signaling an error:
`raise` *ExcClass(…)*
Errors processing:
`try:`
→ *normal procesising block*
`except` *Exception* `as e:`
→ *error processing block*



`finally` *block for final processing in all cases.*

Conditional Loop Statement

statements block executed ***as long as*** *condition is true*

`while` *logical condition*`:`
→ *statements block*

beware of infinite loops!

```
s = 0      } initializations before the loop
i = 1      }
while i <= 100:   } condition with a least one variable value (here i)
    s = s + i**2
    i = i + 1     # make condition variable change !
print("sum:",s)
```

Loop Control

`break` *immediate exit*
`continue` *next iteration*
`else` *block for* ***normal*** *loop exit.*

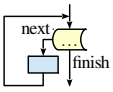
Algo:

$$s = \sum_{i=1}^{i=100} i^2$$

Iterative Loop Statement

statements block executed ***for each*** *item of a container or iterator*

`for var in` *sequence*`:`
→ *statements block*



Go over sequence's **values**

```
s = "Some text"   } initializations before the loop
cnt = 0
for c in s:       } loop variable, assignment managed by for statement
    if c == "e":
        cnt = cnt + 1
print("found",cnt,"'e'")
```

Algo: count number of e *in the string.*

loop on dict/set ⇔ loop on keys sequences
use *slices* to loop on a subset of a sequence

Go over sequence's **index**
- modify item at index
- access items around index (before / after)

```
lst = [11,18,9,12,23,4,17]
lost = []
for idx in range(len(lst)):
    val = lst[idx]
    if val > 15:
        lost.append(val)
        lst[idx] = 15
print("modif:",lst,"-lost:",lost)
```

Algo: limit values greater than 15, memorizing of lost values.

Go simultaneously over sequence's **index** and **values**:
`for idx,val in enumerate(lst):`

good habit : don't modify loop variable

Display

```
print("v=",3,"cm :",x,",",y+4)
```

items to display : literal values, variables, expressions

`print` options:
- `sep=" "` items separator, default space
- `end="\n"` end of print, default new line
- `file=sys.stdout` print to file, default standard output

Input

```
s = input("Instructions:")
```

`input` always returns a **string**, convert it to required type (cf. boxed *Conversions* on the other side).

Generic Operations on Containers

`len(c)` → items count
`min(c)` `max(c)` `sum(c)`
`sorted(c)` → `list` sorted *copy*
`val in c` → boolean, membership operator `in` (absence `not in`)
`enumerate(c)` → *iterator* on (index, value)
`zip(c1,c2…)` → *iterator* on tuples containing c_i items at same index
`all(c)` → `True` if **all** `c` items evaluated to true, else `False`
`any(c)` → `True` if **at least one** item of `c` evaluated true, else `False`

Note: For dictionaries and sets, these operations use ***keys****.*

Specific to ***ordered sequences containers*** *(lists, tuples, strings, bytes…)*
`reversed(c)` → *inversed iterator* `c*5` → duplicate `c+c2` → concatenate
`c.index(val)` → *position* `c.count(val)` → *events count*

Integer Sequences

`range([`*start*`,] `*end*` [,`*step*`])`
start default 0, *end* not included in sequence, *step* signed, default 1

`range(5)` → 0 1 2 3 4 `range(2,12,3)` → 2 5 8 11
`range(3,8)` → 3 4 5 6 7 `range(20,5,-5)` → 20 15 10
`range(len(`*seq*`))` → *sequence of index of values in seq*
range provides an immutable sequence of int constructed as needed

Operations on Lists

modify original list

```
lst.append(val)          add item at end
lst.extend(seq)          add sequence of items at end
lst.insert(idx, val)     insert item at index
lst.remove(val)          remove first item with value val
lst.pop([idx]) →value    remove & return item at index idx (default last)
lst.sort()  lst.reverse()  sort / reverse liste in place
```

Operations on Dictionaries

```
d[key]=value          d.clear()
d[key] → value        del d[key]
d.update(d2)          update/add associations
d.keys()
d.values()            →iterable views on keys/values/associations
d.items()
d.pop(key[,default]) → value
d.popitem() → (key,value)
d.get(key[,default]) → value
d.setdefault(key[,default]) →value
```

Operations on Sets

Operators:

- `|` → union (vertical bar char)
- `&` → intersection
- `-` `^` → difference/symmetric diff.
- `<` `<=` `>` `>=` → inclusion relations

Operators also exist as methods.

```
s.update(s2)  s.copy()
s.add(key)  s.remove(key)
s.discard(key)  s.clear()
s.pop()
```

Files

storing data on disk, and reading it back

```
f = open("file.txt","w",encoding="utf8")
```

- file **variable** for operations
- **name** of file on disk (+path…)
- opening **mode**
  - `'r'` read
  - `'w'` write
  - `'a'` append
  - …`'+'` `'x'` `'b'` `'t'`
- **encoding** of chars for *text files*: `utf8` `ascii` `latin1` …

cf. modules `os`, `os.path` and `pathlib`

writing

```
f.write("coucou")
f.writelines(list of lines)
```

reading

read empty string if end of file

```
f.read([n])          → next chars
                       if n not specified, read up to end !
f.readlines([n])     → list of next lines
f.readline()         → next line
```

*text mode `t` by default (read/write `str`), possible binary mode `b` (read/write `bytes`). **Convert from/to required type** !*

```
f.close()
```

dont forget to **close the file** after use !

```
f.flush()  write cache
f.truncate([size])  resize
```

reading/writing progress sequentially in the file, modifiable with:

```
f.tell() →position
f.seek(position[,origin])
```

Very common: opening with a guarded block (automatic closing) and reading loop on lines of a text file:

```
with open(…) as f:
    for line in f :
        # processing of line
```

Function Definition

function name (identifier), named parameters

```
def fct(x,y,z):
    """documentation"""
    # statements block, res computation, etc.
    return res
```

`fct`

result value of the call, if no computed result to return: `return None`

parameters and all variables of this block exist only *in* the block and *during* the function call (think of a "black box")

Advanced: `def fct(x,y,z,*args,a=3,b=5,**kwargs):`

**args variable positional arguments (→`tuple`), default values,*
***kwargs variable named arguments (→`dict`)*

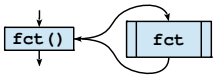
Function Call

```
r = fct(3,i+2,2*i)
```

storage/use of returned value — *one argument per parameter*

this is the use of function name *with parentheses* which does the call

Advanced: **sequence* **dict*



Operations on Strings

```
s.startswith(prefix[,start[,end]])
s.endswith(suffix[,start[,end]])  s.strip([chars])
s.count(sub[,start[,end]])  s.partition(sep) → (before,sep,after)
s.index(sub[,start[,end]])  s.find(sub[,start[,end]])
s.is…()  tests on chars categories (ex. s.isalpha())
s.upper()  s.lower()  s.title()  s.swapcase()
s.casefold()  s.capitalize()  s.center([width,fill])
s.ljust([width,fill])  s.rjust([width,fill])  s.zfill([width])
s.encode(encoding)  s.split([sep])  s.join(seq)
```

Formatting

formating directives — values to format

```
"modele{} {} {}".format(x,y,r) → str
"{selection:formatting!conversion}"
```

- **Selection** :
  - `2`
  - `nom`
  - `0.nom`
  - `4[key]`
  - `0[2]`
- **Formatting** :

fill char *alignment* *sign* *mini width*.*precision~maxwidth* *type*

`< > ^ =` `+ -` *space* `0` at start for filling with 0

integer: `b` binary, `c` char, `d` decimal (default), `o` octal, `x` or `X` hexa…
float: `e` or `E` exponential, `f` or `F` fixed point, `g` or `G` appropriate (default),
string: `s` … `%` percent

- **Conversion** : `s` (readable text) or `r` (literal representation)

Examples:

```
"{:+2.3f}".format(45.72793)
→'+45.728'
"{1:>10s}".format(8,"toto")
→'      toto'
"{x!r}".format(x="I'm")
→'"I\'m"'
```