Algorithmik kontinuierlicher Systeme

30. Januar 2017

Zusammenfassung der wichtigen Verfahren anhand von Vorlesungsfolien, Übungen, Skript und „Summary“. Fehler oder fehlende Inhalte bitte melden: he29heri@stud.informatik.uni-erlangen.de
Ergänzt für das SS 2015 von Andreas Hammer (andreas.hammer@fau.de)

1 Was sind kontinuierliche Daten?

Fluch der Dimensionen Speicherkomplexität von $\mathbb{R}^k \to \mathbb{R}^l$ ist $n^k l$ (n diskrete Werte).

2 Fehler

x exakter Wert, $\hat{x}$ gemessener Wert

$$\epsilon_{\text{abs}} = \hat{x} - x \qquad \text{absoluter Fehler}$$
$$\epsilon_{\text{rel}} = \left| \frac{x - \hat{x}}{x} \right| \qquad \text{relativer Fehler}$$

Bei Vektoren $||\hat{x} - x||$ bzw. $\frac{||\hat{x}-x||}{||x||}$ benutzen.
ϵ_x Eingabefehler von x, ϵ_y Eingabefehler von y; betrachtet wird jeweils $x \circ y$.

$$\epsilon_{\text{abs }+} = \epsilon_x + \epsilon_y$$
$$\epsilon_{\text{rel }+} = \left| \frac{\epsilon_x + \epsilon_y}{x + y} \right|$$
$$\epsilon_{\text{abs }\cdot} = x \cdot \epsilon_y + y \cdot \epsilon_x$$
$$\epsilon_{\text{rel }\cdot} = \left| \frac{x \cdot \epsilon_y + y \cdot \epsilon_x}{x \cdot y} \right| = \left| \frac{\epsilon_y}{y} + \frac{\epsilon_x}{x} \right|$$

Somit addieren sich bei der Addition die absoluten Fehler, bei der Multiplikation die relativen Fehler.

2.1 Kondition

2.1.1 Kondition von einfachen Funktionen

Verhältnis von relativen Eingabe- zu relativem Ausgabefehler; unabhängig vom Berechnungsverfahren.

$$\kappa_f(x) := \left| \frac{x \cdot f'(x)}{f(x)} \right| \qquad \text{Kondition}$$
$$\kappa_{f \circ g} = \kappa_f \cdot \kappa_g \qquad \text{Verkettung}$$

2.1.2 Kondition von Matrizen

Bei einer Matrix A ist die Kondition der Matrix

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

2.1.3 Anwendung

Zusammenhang zwischen Eingabedaten x, einer Matrix A und Ausgabedaten b:

$$\frac{||\hat{x} - x||}{x} \leq \kappa(A) \cdot \frac{||\hat{b} - b||}{||b||}$$

Allgemein:

$$\kappa(A \cdot B) \leq \kappa(A) \cdot \kappa(B)$$

2.2 Stabilität

Verstärkung von Rechenfehlern in den Eingabedaten durch das Berechnungsverfahren.

3 Falten und Filtern

Falten von kontinuierlichen Daten, *Filtern* von diskreten Daten.

3.1 Separierbarkeit

Eine Matrix ist separierbar, wenn ihr Rang gleich 1 ist.
Anschaulich: Wenn nur eine „sinnvolle“ Zeile existiert, und die anderen nur Vielfache davon sind.

3.2 Separieren von Filtern

Für jede Spalte und jede Zeile der Matrix Faktoren finden, die das jeweilige Element erzeugen. Diese Faktoren in Vektoren schreiben.
Beispiel:

$$\begin{array}{c} \\ 1 \\ 2 \\ 1 \end{array}\!\!\begin{array}{c} \begin{array}{ccc} 1 & 2 & 1 \end{array} \\ \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \end{array} \Rightarrow \begin{bmatrix} 1 \cdot 1 & 1 \cdot 2 & 1 \cdot 1 \\ 2 \cdot 1 & 2 \cdot 2 & 2 \cdot 1 \\ 1 \cdot 1 & 1 \cdot 2 & 1 \cdot 1 \end{bmatrix} \Rightarrow \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix}$$

3.3 Filtern

Der Filter wird für jedes zu filternde Element auf die Zielmatrix „gelegt“ und die einzelnen Werte, die übereinander liegen, multipliziert. Die Summe dieser Produkte ist dann das neue Element an der entsprechenden Stelle.
Bei separierten Filtern wird jede Zeile und jede Spalte einzeln multipliziert.
Wichtig: Die Ergebnisse müssen in einer neuen, temporären Matrix gespeichert werden!

3.4 Aufwand

Anzahl der Multiplikationen für das Filtern einer $N \times N$-Matrix mit einem $k \times k$-Filter:
Separierbar: $2 \cdot k \cdot N^2$
Nicht separierbar: $k^2 \cdot N^2$
Anzahl *aller* nötigen Operationen (zusätzlich Aufsummieren der Werte):
Separierbar: $2 \cdot \left(2 \cdot k \cdot N^2\right)$
Nicht separierbar: $2 \cdot \left(k^2 \cdot N^2\right)$

3.5 Falten

Die Verknüpfung von zwei Faltungsfunktionen $f(x)$ und $g(x)$ ist definiert als

$$(f * g)(x) = \int f(\tau)g(x - \tau)d\tau$$

wobei τ eine Hilfsvariable ist. Anschaulich: Die erste Funktion wird an der y-Achse gespiegelt und über die zweite Funktion geschoben. Die überdeckte Fläche in Abhängigkeit der Verschiebung ist die Ergebnisfunktion.
Für diese Verknüpfung gilt das Kommutativ- (Faktoren vertauschen), Assoziativ- (Klammerung) und Distributivgesetz (Ausklammern). *Vorgehen:* Man stellt einzelne Integrale auf, die die jeweiligen Definitionsgrenzen der Funktionen haben. Bei konstanten Funktionen kommt ein lineares Integral heraus, bei linearen Funktionen ein quadratisches, etc.

4 Arithmetik

4.1 IEEE-Gleitkommazahlen

Speicherung von V Vorzeichen, M Mantisse (1 wird nicht gespeichert) und E Exponent (mit B Bias, $B = 2^{k-1} - 1$, k Stellen des Exponents).

$$x = (-1)^V \cdot 1, M \cdot 2^{E-B}$$

Obere Schranke für den relativen Fehler (Maschinengenauigkeit), t sind die Stellen der Mantisse:

$$|f_{\text{rel}}(x)| \leq 2^{-t} =: \epsilon$$

4.1.1 Rechenregeln

Das Kommutativgesetz gilt für Maschinenzahlen, Assoziativgesetz und Distributivgesetz allerdings nicht.

5 Lineare Gleichungssysteme

5.1 Vorwärtseinsetzen

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

```
unsigned int n = l.getHeight(); // l = lower matrix
Matrix y = Matrix(n, 1); // result

unsigned int i, j;
for (i = 0; i < n; i++) {
    double value = b.getEntry(i, 0);
    for (j = 0; j < i; j++) {
        value -= l.getEntry(i, j) * y.getEntry(j, 0);
    }
    value /= l.getEntry(i, i); // not necessary for LU
    y.setEntry(i, 0, value);
}
```

5.2 Rückwärtseinsetzen

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

```
unsigned int n = u.getHeight(); // u = upper matrix
Matrix x = Matrix(n, 1); // result

int i, j;
```

```
for (i = n - 1; i >= 0; i--) {
    assert(u.getEntry(i, i) != 0);

    double value = y.getEntry(i, 0);
    for (j = n - 1; j > i; j--) {
        value -= u.getEntry(i, j) * x.getEntry(j, 0);
    }
    x.setEntry(i, 0, value / u.getEntry(i, i));
}
```

5.3 Gauß'sches Eliminationsverfahren

Aufwand $\mathcal{O}(n^3)$

Code analog zur LU-Zerlegung, l wird nicht verwendet.

5.4 LU-Zerlegung

Aufwand $\mathcal{O}(n^3)$, falls L und U bekannt sind $\mathcal{O}(n^2)$

Analog zu Gauß, Multiplikationsfaktor wird in L gespeichert, U ist Ergebnis des Gauß-Verfahrens.

$$\begin{aligned} A &= L \cdot U && \text{LU-Zerlegung} \\ Ly &= b && \text{Lösen durch Vorwärtseinsetzen} \\ Ux &= y && \text{Lösen durch Rückwärtseinsetzen} \end{aligned}$$

$$\underbrace{\begin{pmatrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{pmatrix}}_{A} = \underbrace{\begin{pmatrix} 1 & 0 & 0 \\ \cdot & 1 & 0 \\ \cdot & \cdot & 1 \end{pmatrix}}_{L} \cdot \underbrace{\begin{pmatrix} \cdot & \cdot & \cdot \\ 0 & \cdot & \cdot \\ 0 & 0 & \cdot \end{pmatrix}}_{U}$$

```
unsigned int n = matrix.getHeight(); // matrix to decompose

Matrix l = Matrix(n, n); // lower matrix
l.setIdentity();
Matrix u = matrix; // upper matrix

unsigned int m, k, j;
for (m = 0; m < n - 1; m++) {
    for (k = m + 1; k < n; k++) {
        assert(u.getEntry(m,m) != 0);

        float factor = u.getEntry(k, m)/u.getEntry(m,m);
        u.setEntry(k, m, 0.f);
        for (j = m + 1; j < n; j++) {
            u.setEntry(k, j, u.getEntry(k, j) - factor * u.getEntry(m, j));
        }
        l.setEntry(k, m, factor);
    }
}
```

5.5 Thomas-Algorithmus

Aufwand $\mathcal{O}(n)$, LU-Zerlegung für tridiagonale Matrizen.

$$\underbrace{\begin{pmatrix} b_1 & c_1 & 0 \\ a_1 & b_2 & c_2 \\ 0 & a_2 & b_3 \end{pmatrix}}_{A} = \underbrace{\begin{pmatrix} 1 & 0 & 0 \\ l_1 & 1 & 0 \\ 0 & l_2 & 1 \end{pmatrix}}_{L} \cdot 2 \underbrace{\begin{pmatrix} r_1 & c_1 & 0 \\ 0 & r_2 & c_2 \\ 0 & 0 & r_3 \end{pmatrix}}_{R}$$

mit $r_1 = b_1, l_1 = a_1/r_1, l_{n-1} = a_{n-1}/r_{n-1}, r_n = b_n - l_{n-1} * c_{n-1}$

6 Lineare Ausgleichsprobleme

Ziel: Residuum ($r = Ax - b$) eines überbestimmten Gleichungssystems zu minimieren; Minimierung von $||Ax - b||_2$ (quadratische Abweichung im Sinne der euklidischen Norm). Bestimmung der Ausgleichsgerade $f(x) = ax + b$ durch die Punkte (x_n, y_n).

6.1 Least Square Methode

Aufwand $\mathcal{O}(n^3)$, im Gegensatz zu QR-Zerlegung schlecht konditioniert.

$$\begin{pmatrix} x_1 & 1 \\ x_2 & 1 \\ \vdots & 1 \\ x_k & 1 \end{pmatrix} \cdot \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_k \end{pmatrix}$$

$$2 \cdot (A^T Ax - A^T b) = 0) \Leftrightarrow A^T Ax = A^T b \qquad \text{Normalengleichung}$$

Berechnung von $A^T A$ und $A^T b$ und Lösen von $A^T A\ x = A^T b$.

6.1.1 Alternative Normen

- Gewichtete Norm:

$$|||r||| := ||Dr||$$
$$||r||^2 := d_1^2 r_1^2 + d_2^2 r_2^2 + \cdots + d_n^2 r_n^2$$

  Sinn: Länger zurückliegende Werte weniger stark gewichten als aktuelle.

6.2 QR-Zerlegung

Aufwand $\mathcal{O}(n^3)$, n^2 Multiplikationen, $n(n-1)$ Additionen. Doppelt so hoch wie LR.

Matrix muss anders als bei LR nicht quadratisch sein. Stabil.

$$A = Q \cdot R$$

$$\underbrace{\begin{pmatrix} \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \end{pmatrix}}_{A} = \underbrace{\begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}}_{Q} \cdot \underbrace{\begin{pmatrix} \cdot & \cdot \\ 0 & \cdot \\ 0 & 0 \\ 0 & 0 \end{pmatrix}}_{R}$$

Das Ausgleichsproblem kann nun gelöst werden:

$$\begin{aligned} QRx &= b \\ z &= Q^T b \\ R'x &= z \qquad R' \text{ ist } R \text{ ohne Nullzeilen} \end{aligned}$$

6.2.1 Jacobi Rotationen

Multiplikation von Givens Matrizen G_{ij} an die Ausgangsmatrix A, die jeweils eine Null an der Stelle ij in der Matrix erzeugen.

$-s$ steht dabei an der Stelle ij in G, c jeweils links und oberhalb auf der Diagonalen, s oben rechts vervollständigt das Rechteck.

$$\underbrace{G_{nm} \cdot \ldots \cdot G_{31} \cdot G_{21}}_{Q^T} \cdot A = R$$

$$G_{52} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & c & 0 & 0 & s \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & -s & 0 & 0 & c \end{pmatrix}$$

$$c = \frac{a_{lo}}{\sqrt{a_{lo}^2 + a_{lu}^2}} \quad s = \frac{a_{lu}}{\sqrt{a_{lo}^2 + a_{lu}^2}}$$

a_{lo} ist dabei der Eintrag in A an der Stelle vom links oberen c, a_{lu} an der Stelle von $-s$.

6.2.2 Householder Spiegelungen

Multiplikation von Householder Matrizen H_i an A, die jeweils eine Nullspalte (unterhalb der Diagonalelemente) in der i-ten Spalte erzeugen.

$$\underbrace{H_m \cdot \ldots \cdot H_2 \cdot H_1}_{Q^T} \cdot A = R$$

$$H_i = E - \frac{2 \cdot u_i u_i^T}{u_i^T u_i} \qquad ||u||_2 = \sqrt{u_1^2 + u_2^2 + \ldots \cdot u_n^2}$$

$$u_i = v_i \pm ||v_i||_2 \cdot e_i \qquad + \text{ oder } - \text{ ist beliebig}$$

e_i ist der i-te Einheitsvektor, v_i ist der i-te Spaltenvektor von A bei dem alle Einträge oberhalb dem i-ten auf Null gesetzt sind.

7 Lineare Algebra

Inverse einer 2×2 Matrix:

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

Definitheit Falls alle Eigenwerte der Matrix ... sind, ist die Matrix ...

- > 0: positiv definit
- ≥ 0: positiv semidefinit
- ≤ 0: negativ semidefinit
- < 0: negativ definit
- sonst indefinit

Alternativ: Betrachtung von $x^T Ax \; \forall x \neq 0$ anstelle der Eigenwerte.

Aufwände

- Matrix-Matrix Multiplikation $\mathcal{O}(n^3)$
- Matrix-Vektor Multiplikation $\mathcal{O}(n^2)$
- Matrix-Skalar Multiplikation $\mathcal{O}(n^2)$

8 Normen

8.1 Vektornorm

$$\text{Summennorm: } ||x||_1 = \sum_i |x_i|$$

$$\text{Euklidische Norm: } ||x||_2 = \sqrt{\left(\sum_{i=1}^{n} x_i^2\right)}$$

$$\text{Maximumsnorm: } ||x||_\infty = \max\{|x_i|\}$$

8.2 Matrixnorm

$$\text{Summennorm: } |||A|||_1 = \text{maximale Spaltensumme}$$

$$\text{Euklidische Norm: } |||A|||_2 = \sqrt{\lambda_{\max}}, \quad \lambda_{\max} \text{ größter Eigenwert von } AA^T$$

$$\text{Maximumsnorm: } |||A|||_\infty = \text{maximale Zeilensumme}$$

Der Spektralradius $r(A)$ ist der betragsmäßig größte Eigenwert von A.

Konditionszahl der Matrix A: $\kappa(A) = |||A||| \cdot |||A^{-1}|||$ oder auch $\kappa(A) = \frac{\lambda_{\max}}{\lambda_{\min}}$ (Eigenwerte dabei jeweils von der Matrix AA^T). Für orthogonale Matrizen: $\kappa = 1$

9 Compressed Row Storage (CRS)

Speicherung der Matrix in drei Arrays: values, colIndices, rowPtr. values und colIndices sind nur so lang wie es Einträge in der Matrix gibt, rowPtr ist immer so lang wie die Matrix Zeilen hat + 1. values speichert den Wert, colIndices die Spalte des Werts. rowPtr zeigt jeweils auf den Anfang der Spalte in values.

Analog Compressed Collumn Storage (CCS): values, rowIndices, colPtr.

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

```
CRSMatrix::CRSMatrix(Matrix &other) {
    unsigned int i, j;

    _height      = other.getHeight();
    _width       = other.getWidth();
    _values      = NULL;
    _colIndices = NULL;
    _rowPtr      = new unsigned int[_height + 1];

    _nonZeroElements = 0;
    for (i = 0; i < _height; i++) {
        for (j = 0; j < _width; j++) {
            if (other.getEntry(i, j) != 0) {
                _nonZeroElements++;
            }
        }
```

```
    }
    if (_nonZeroElements == 0) {
        return;
    }

    unsigned int index = 0;
    _values     = new float[_nonZeroElements];
    _colIndices = new unsigned int[_nonZeroElements];

    for (i = 0; i < _height; i++) {
        for (j = 0; j < _width; j++) {
            float entry = other.getEntry(i, j);
            if (entry != 0) {
                _values[index] = entry;
                _colIndices[index] = j;

                index++;
            }
        }
        _rowPtr[i+1] = index;
    }
}
```

getEntry() Aufwand $\mathcal{O}(n)$ (bei schneller Implementierung $\mathcal{O}(\log n)$)

```
float CRSMatrix::getEntry(unsigned int i, unsigned int j) const {
    if (_nonZeroElements == 0) {
        return 0.0f;
    }

    unsigned int k;
    // Search column in all elements in this row.
    for (k = _rowPtr[i]; k < _rowPtr[i+1]; k++) {
        if (_colIndices[k] == j) {
            return _values[k];
        }
    }

    return 0.0f;
}
```

Effiziente Matrixmultiplikation

```
Matrix m(_height, other.getWidth()); // result
m.setZero();

unsigned int row = 0; // current row

unsigned int i, j;
for (i = 0; i < _nonZeroElements; i++) {
    // We are at the end of the current row, move to next row.
    // Also jump over empty rows.
    while (_rowPtr[row] == _rowPtr[row+1]
            || i == _rowPtr[row+1]) {
        row++;
    }

    for (j = 0; j < other.getWidth(); j++) {
        m.setEntry(row, j, m.getEntry(row, j) +
                _values[i] * other.getEntry(_colIndices[i], j));
```

```
    }
}
```

10 Singulärwertzerlegung (SVD)

$$\underbrace{A}_{m\times n} = \underbrace{\overbrace{U}^{A^TA}}_{m\times m} \cdot \underbrace{\Sigma}_{m\times n} \cdot \underbrace{\overbrace{V^T}^{AA^T}}_{n\times n}$$

$$\underbrace{\begin{pmatrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{pmatrix}}_{A} = \underbrace{\begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}}_{U} \cdot \underbrace{\begin{pmatrix} \sigma_1 & 0 & 0 \\ 0 & \sigma_2 & 0 \\ 0 & 0 & \sigma_3 \\ 0 & 0 & 0 \end{pmatrix}}_{\Sigma} \cdot \underbrace{\begin{pmatrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{pmatrix}}_{V^T}$$

Berechnung der Eigenwerte und Eigenvektoren von A^TA und Speicherung der Singulärwerte (σ, Wurzeln der Eigenwerte) absteigend in Σ ($\sigma_1 > \sigma_2 > \ldots$). V (nicht V^T!) besteht aus den Eigenvektoren als Spalten (alternativ die Zeilen von V^T). U kann nun aus V bestimmt werden, dabei sind die Spaltenvektoren durch $u_i = \frac{1}{\sigma_i} A v_i$ bestimmt. Fehlende Vektoren müssen ergänzt werden, dabei muss U eine Orthonormalbasis aufspannen (z. B. durch Kreuzprodukt im $\mathbb{R}^3$). Geometrische Interpretation: U = Drehung, Σ = Streckung, V^T = Drehung

10.1 Eigenschaften ablesen

Der Rang r von A ist die Anzahl der Singulärwerte (Werte in Σ), das Bild sind die ersten r Spaltenvektoren von U, der Kern die letzten $n - r$ Spaltenvektoren von V. Die Kondition von A bezüglich der Euklidischen Norm ist $\kappa_2 = \frac{\max\limits_n \sigma_n}{\min\limits_n \sigma_n}$. Die Euklidische Norm ($|||A|||_2$) ist der größte Singulärwert.

10.2 Pseudoinverse $A^{\sim 1}$

Mit der Pseudoinverse kann man über- (Lösen eines Ausgleichsproblems) und unterbestimmte (kleinster Abstand zum Ursprung) LGS lösen.

$$A^{\sim 1} = V \Sigma^{\sim 1} U^T$$

$$x = A^{\sim 1} b = V(\Sigma^{\sim 1}(U^T b))$$

$\Sigma^{\sim 1}$ erhält man, indem man die Kehrwerte der Hauptdiagonale bildet und das Ergebnis transponiert. Alternativ kann x auch wie folgt berechnet werden:

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

Hierbei ist r der Rang der Matrix A, σ_i der Singulärwert an der Stelle i, u_i die i-te Spalte von U und v_i die i-te Spalte von V.

10.3 Low Rank Approximation

Approximieren der Matrix A durch eine Matrix A_k mit niedrigerem Rang. Dabei einfach die letzten (unteren) Singulärwerte auf Null setzen und die Matrix neu berechnen.

10.4 Frobenius-Norm

Die Frobenius-Norm einer Matrix A ist

$$||A||_F = \sqrt{\sigma_0^2, \ldots, \sigma_n^2}$$

Die Matrix mit Rang k, die A im Sinne der Frobenius-Norm am besten approximiert, ist

$$A_k = \sum_{i=1}^{k} \sigma_i u_i v_i^T$$

11 Hauptkomponentenanalyse

Ziel: Reduzierung der Dimension eines Datensatz unter Erhalt der charakteristischen Eigenschaften.
Gegeben: n Punkte (x_i, y_i).

Zuerst Berechnung des Mittelwertvektors (Mittelwerte der x- und y-Koordinaten):

$$\overline{x} = \frac{1}{n} \begin{pmatrix} x_1 + x_2 + \ldots + x_n \\ y_1 + y_2 + \ldots + y_n \end{pmatrix}$$

Nun muss das Koordinatensystem so verschoben werden, dass der Mittelwertvektor im Ursprung liegt:

$$z_i = \begin{pmatrix} x_i \\ y_i \end{pmatrix} - \overline{x} = \begin{pmatrix} x_i' \\ y_i' \end{pmatrix}$$

Mit den erhaltenen Vektoren wird nun A aufgestellt:

$$A = \begin{pmatrix} x_1' & x_2' & \ldots & x_n' \\ y_1' & y_2' & \ldots & y_n' \end{pmatrix}$$

Nun folgt das Aufstellen der Kovarianzmatrix C:

$$C = \frac{1}{n-1} A A^T$$

Nun müssen die Eigenwerte und Eigenvektoren von C berechnet werden. Damit kann C wie folgt zerlegt werden:

$$C = QWQ^T$$

$$\underbrace{\begin{pmatrix} \cdot & \cdot \\ \cdot & \cdot \end{pmatrix}}_{C} = \underbrace{\begin{pmatrix} \cdot & \cdot \\ \cdot & \cdot \end{pmatrix}}_{Q} \cdot \underbrace{\begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix}}_{W} \cdot \underbrace{\begin{pmatrix} \cdot & \cdot \\ \cdot & \cdot \end{pmatrix}}_{Q^T}$$

Dabei besteht Q aus den Eigenvektoren als Spalten, W besteht aus den Eigenwerte auf der Diagonale (nach Größe der Eigenwerte sortiert). Nun können die einzelnen Punkte in dem neuen Koordinatensystem dargestellt werden:

$$y_i = Q^T z_i$$

Alternativ lässt sich die PCA auch durch die SVD berechnen. Dabei sind die Hauptkomponenten (Q) die Spalten von U der SVD. Die Eigenwerte lassen aus den Singulärwerten berechnen:

$$\lambda_i = \frac{1}{n-1} \sigma_i^2$$

12 Median Cut

12.1 Vorgehen

1. Konvexe Hülle um gegebene Punkte ziehen
2. Längste Kante durch 2 teilen ⇒ es sollten auf beiden Seiten ca. gleichviele Punkte vorkommen
3. Schritte wiederholen, bis gewünschte Iterationszahl erreicht ist

12.2 Wahl der Repräsentanten

Entweder Mittelpunkt der konvexen Hülle oder Mittelwert der enthaltenen Punkte.

13 Interpolation

13.1 Lokale Interpolation

13.1.1 Nearest Neighbor

Der y-Wert der Stützstelle x_i, die am nächsten liegt, wird als Interpolationswert für das gesuchte x verwendet. Graphisch: Man zieht bei jedem Punkt gerade (konstante) Linien zwischen den Mitten der x-Werte der Punkte.

13.1.2 Lineare Interpolation

Benachbarte Stützstellen werden jeweils durch eine Gerade verbunden, somit ergibt sich $p(x)$ aus $n-1$ linearen Funktionen p_i:

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

Anschaulich: Die Steigung ist der Unterschied des Ergebnisses pro Änderung der Eingabe, also y durch x. X und Y müssen vom Nullpunkt zum Startpunkt des Abschnittes verschoben werden.

13.1.3 Catmull-Rom Interpolation

Benachbarte Stützstellen werden jeweils durch ein Polynom verbunden. Dazu muss an der Stützstelle x_i mit Funktionswert y_i die Ableitung y_i' geschätzt werden. Das Polynom für dieses Teilintervall berechnet sich dann durch:

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

$$a_0 = \frac{y_i}{(x_{i+1} - x_i)^3}$$

$$a_1 = 3 \cdot a_0 + \frac{y_i'}{(x_{i+1} - x_i)^2}$$

$$a_2 = 3 \cdot a_3 - \frac{y_{i+1}'}{(x_{i+1} - x_i)^2}$$

$$a_3 = \frac{y_{i+1}}{(x_{i+1} - x_i)^3}$$

Die Ableitungen können auf unterschiedliche Weise geschätzt werden:

- Vorwärtsdifferenz: $y_i' = \frac{y_{i+1} - y_i}{x_{i+1} - x_i}$
- Rückwärtsdifferenz: $y_i' = \frac{y_i - y_{i-1}}{x_i - x_{i-1}}$
- Zentrale Differenz: $y_i' = \frac{y_{i+1} - y_{i-1}}{x_{i+1} - x_{i-1}}$
  (Diese in der Klausur benutzen, falls nicht anders angegeben)

13.1.4 Fehler

- Nearest Neighbor: $\mathcal{O}(h)$
- Linear: $\mathcal{O}(h^2)$
- Catmull-Rom: $\mathcal{O}(h^3)$

13.2 Globale Interpolation

13.2.1 Vandermonde Matrix

Aufwand: $\mathcal{O}(n^3)$, schlecht konditioniert.

Berechnet ein Polynom vom Grad $n-1$ durch alle n Punkte.

$$\begin{pmatrix} 1 & x_1 & \dots & x_1^{n-1} \\ 1 & x_2 & \dots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \vdots \\ 1 & x_n & \dots & x_n^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$$

13.2.2 Lagrange Polynome

Zuerst müssen die Lagrangepolynome $L_i(x)$ berechnet werden, dabei sind x_i, x_j die Stützstellen:

$$L_i(x) = \prod_{\substack{j=1 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j} = \frac{x - x_1}{x_i - x_1} \cdot \frac{x - x_2}{x_i - x_2} \cdot \dots$$

Das Interpolationspolynom $p(x)$ ergibt sich somit durch:

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

13.2.3 Newton Polynome

Zuerst müssen die Newtonpolynome $N_i(x)$ berechnet werden, dabei sind x_i, x_j die $n{+}1$ Stützstellen, n ist der Grad des Interpolationspolynoms:

$$N_i(x) = \prod_{j=0}^{i-1} (x - x_j) = (x - x_0)(x - x_1) \dots (x - x_{i-1}) \qquad N_0(x) = 1$$

Das Interpolationspolynom $p(x)$ ergibt sich somit durch:

$$p(x) = \sum_{i=0}^{n} c_i \cdot N_i(x) = c_0 + c_1(x - x_0) + c_2(x - x_0)(x - x_1) + \dots c_n(x - x_0)(x - x_1) \dots (x - x_{n-1})$$

Die Koeffizienten c_i lassen sich durch den Algorithmus von Aitken-Neville ($\mathcal{O}(n^2)$) berechnen:
Formal: $p_{i,k} = \frac{p_{i+1,k-1} - p_{i,k-1}}{x_{i+k} - x_i}$

Beispiel bei 4 y-Werten:

$$\begin{array}{lllll} p_{0,0} = y_0 & \longrightarrow & p_{0,1} = \frac{p_{1,0} - p_{0,0}}{x_1 - x_0} & \longrightarrow & p_{0,2} = \frac{p_{1,1} - p_{0,1}}{x_2 - x_0} \longrightarrow p_{0,3} = \frac{p_{1,2} - p_{0,2}}{x_3 - x_0} \\ & \nearrow & & \nearrow & \\ p_{1,0} = y_1 & \longrightarrow & p_{1,1} = \frac{p_{2,0} - p_{1,0}}{x_2 - x_1} & \longrightarrow & p_{1,2} = \frac{p_{2,1} - p_{0,1}}{x_3 - x_1} \\ & \nearrow & & \nearrow & \\ p_{2,0} = y_2 & \longrightarrow & p_{2,1} = \frac{p_{3,0} - p_{2,0}}{x_3 - x_2} & & \\ & \nearrow & & & \\ p_{3,0} = y_3 & & & & \end{array}$$

$p(x)$ lässt sich nun effizient mit dem erweiterten Horner-Schema in $\mathcal{O}(n)$ auswerten.

```
float result = coeff(n - 1, 0); // c_0 ... c_{n-1}
while (n --> 1) {
    result = result * (x - m_controlPoints[n - 1].x) + coeff(n - 1, 0);
}
```

Von Hand:

$$p(x) = a_0 + a_1(x - x_1) + a_2(x - x_1)(x - x_2) + \cdots + a_{n-1}(x - x_1)(x - x_2)\ldots(x - x_{n-1})$$

13.2.4 Bilineare Interpolation

Grundlage ist die lineare Interpolation auf einer Strecke. x_0 und x_1 sind die Stützstellen mit den Stützwerten f_0 und f_1. An der Interpolationsstelle p wird der Interpolationswert f_p berechnet.

$$\alpha = \frac{p - x_0}{x_1 - x_0} \qquad \text{Streckenverhältnis}$$
$$f_p = (1 - \alpha)f_0 + \alpha f_1$$

Bei der bilineare Interpolation sollen nun Werte in einem Rechteck interpoliert werden. Dabei wird zuerst für die obere und untere Kante zwei mal in x-Richtung linear interpoliert, dann mit den berechneten Werten in y-Richtung (oder andersrum).

13.2.5 Baryzentrische Koordinaten

Die baryzentrische Koordinaten eines Dreiecks RST für einen Punkt P ergeben sich durch:

$$\rho = \frac{area(\Delta(PST))}{area(\Delta(RST))} = \frac{det(S - P, T - P)}{det(S - R, T - R)}$$
$$\sigma = \frac{area(\Delta(RPT))}{area(\Delta(RST))} = \frac{det(P - R, T - R)}{det(S - R, T - R)}$$
$$\tau = \frac{area(\Delta(RSP))}{area(\Delta(RST))} = \frac{det(S - R, P - R)}{det(S - R, T - R)}$$

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

Alternativ kann ρ, σ und τ auch durch Lösen eines LGS (ausgehend von der Formel für P und da $\rho + \sigma + \tau = 1$ gilt) berechnet werden:

$$\begin{aligned} \rho + \sigma + \tau &= 1 \\ R_x \cdot \rho + S_x \cdot \sigma + T_x \cdot \tau &= P_x \\ R_y \cdot \rho + S_y \cdot \sigma + T_y \cdot \tau &= P_y \end{aligned}$$

Zur Interpolation dient folgende Formel:

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

Merksatz für Aufgaben der Art „Wo ist $\rho > 0$?“:
Rho = 0 gegenüber von R, Tau gegenüber von T, Sigma gegenüber von S. In das Dreieck hinein positiv, aus dem Dreieck heraus negativ.

14 Bézierkurven

Bernsteinpolynome sind definiert durch:

$$B_i^n(t) := \binom{n}{i}(1-t)^{n-i}t^i$$

Bézierkurven vom Grad n sind wie folgt definiert:

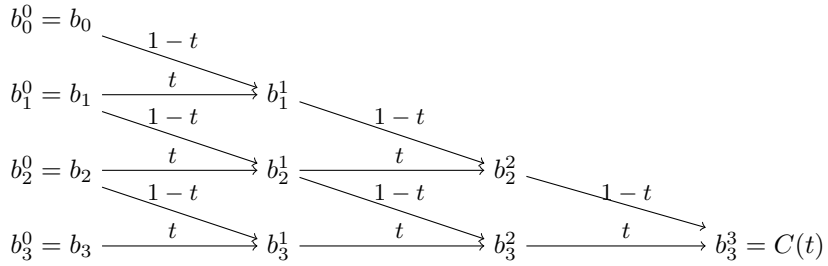
$$C(t) = \sum_{i=0}^{n} b_i B_i^n(t) \qquad t \in [0,1]$$

14.1 Eigenschaften

- Interpolation der Eckpunkte
- In den Endpunkten tangential an das Kontrollpolygon
- Bézierkurve liegt in der konvexen Hülle des Kontrollpolygons
- Affine Invarianz (Verschiebung/Drehung/Streckung der Bézierkurve durch Operation auf dem Kontrollpolygon)
- Variationsreduzierend (jede Gerade schneidet die Bézierkurve maximal so oft wie das Kontrollpolygon)

14.2 Algorithmus von de Casteljau

Schnelle Auswertung von Bézierkurven (Beispiel mit 4 Punkten):




```
// evaluate for parameter t
unsigned int n = m_controlPoints.size();
std::vector<Point3D> points = m_controlPoints; // create a copy!

for (unsigned int i = 1; i < n; i++) {
    for (unsigned int j = n - 1; j >= i; j--) {
        points[j] = points[j - 1] * (1 - t) + points[j] * t;
    }
}

return points[n-1]; // = C(t)
```

Geometrisch gesehen werden jeweils zwei Kontrollpunkte verbunden und dann im Verhältnis $t : 1-t$ geteilt. Es werden alle Strecken im Kontrollpolygon in diesem Verhältnis geteilt und die entstehenden Punkte verbunden. Dies wiederholt man nun mit den neuen Strecken.

14.3 Midpoint Subdivision

Ermöglicht es schnell Bézierkurven zu zeichnen, ähnlich zum Verfahren von de Casteljau mit $t = 0.5$. In jedem Schritt werden die aktuellen Strecken halbiert und die entstehenden Punkte zu den neuen Linien verbunden.

Falls Programmieraufgabe:

1. deCasteljau ausführen
2. linke Teilkurve: von links oben Diagonal runter
   rechte Teilkurve: von links unten gerade nach rechts
3. Für diese Teilkurven rekursieren, bis maximale Rekursionstiefe erreicht
4. Ergebnisse zusammenführen: links normal, rechts in umgekehrter Reihenfolge

14.4 Coons-Patch

$$
\begin{aligned}
F_s(s,t) &= (1-t)\cdot C_S(s) + t\cdot C_N(s) \\
F_t(s,t) &= (1-s)\cdot C_W(t) + s\cdot C_O(t) \\
[\ F_{st}(s,t) &= (1-s)\cdot(1-t)\cdot C_W(0) + s\cdot(1-t)\cdot C_O(0) + (1-s)\cdot t\cdot C_W(1) + st\cdot C_O(1)\ ]
\end{aligned}
$$

Einfachere Formel:

$$
\begin{aligned}
F_{st}(s,t) &= (1-t)\cdot F_t(s,0) + t\cdot F_t(s,1) \\
\Rightarrow F(s,t) &= F_s(s,t) + F_t(s,t) - F_{st}(s,t)
\end{aligned}
$$

15 Flächenberechnung

15.1 Monte-Carlo

Erzeugen von Zufallspunkten und Zählen der Treffer. Konvertiert recht langsam: $\mathcal{O}(N^{-\frac{1}{2}})$

15.2 Newton-Cotes-Formeln

15.2.1 Prinzip

$\int_a^b f(x)$ soll angenähert werden. Dazu werden $n+1$ äquidistante Stützstellen in diesem Intervall gewählt. Die dazugehörigen Stützpunkte werden durch Polynominterpolation (Lagrangepolynome) approximiert und das erhaltene Polynom integriert um die Fläche zu berechnen.

15.2.2 Rechtecksregel ($n = 0$)

Stützstelle x_0 in $[a, b]$, Interpolationspolynom ist eine konstante Funktion auf Höhe $f(x_0)$. Somit gilt:

$$\int_a^b f(x)\,\mathrm{d}x \approx (b-a)\cdot f(x_0)$$

x_0 kann links ($x_0 = a$), rechts ($x_0 = b$) oder in der Mitte ($x_0 = \frac{a+b}{2}$; Mittelpunktsregel) des Intervalls gewählt werden.

15.2.3 Trapezregel ($n = 1$)

Annäherung durch ein Trapez mit den Eckpunkten $(a, 0)$, $(b, 0)$, $(a, f(a))$, $(b, f(b))$:

$$\int_a^b f(x)\,\mathrm{d}x \approx \frac{b-a}{2}\cdot(f(a)+f(b))$$

15.2.4 Simpsonregel ($n = 2$)

Annäherung durch ein „Rechteck mit Parabel oben drauf“:

$$\int_a^b f(x)\,\mathrm{d}x \approx \frac{b-a}{6}\cdot\left(f(a) + 4\cdot f\left(\frac{a+b}{2}\right) + f(b)\right)$$

15.2.5 Newtons 3/8 Regel ($n = 3$)

$$\int_a^b f(x)\,\mathrm{d}x \approx \frac{b-a}{8} \cdot (f(a) + 3f(x_1) + 3f(x_2) + f(b))$$

$$x_i = a + i \cdot h \qquad h = \frac{b-a}{3}$$

15.2.6 Fehlerabschätzung

- Mittelpunktsregel: $\mathcal{O}(h^3)$, $(h = b - a)$
- Trapezregel: $\mathcal{O}(h^3)$, $(h = b - a)$
- Simpsonregel: $\mathcal{O}(h^5)$, $(h = \frac{b-a}{2})$
- Newtons 3/8 Regel: $\mathcal{O}(h^5)$, $(h = \frac{b-a}{3})$

15.3 Iterierte Newton-Cotes-Formeln

Da Newton-Cotes-Formeln höheren Grades problematisch sind, wird das Integrationsintervall in mehrere gleich große Teilintervalle unterteilt und auf jedem Intervall eine Newton-Cotes-Formel niedrigen Grades angewendet (z. B. Trapezregel oder Simpsonregel).

15.3.1 Trapezsumme

$$\int_a^b f(x)\,\mathrm{d}x \approx \mathcal{T}^h_{[a,b]} := h\left(\frac{f(a)}{2} + f(a+h) + f(a+2h) + \ldots + f(a+(n-1)h) + \frac{f(b)}{2}\right)$$

15.3.2 Simpsonsumme

$$\int_a^b f(x)\,\mathrm{d}x \approx \mathcal{S}^h_{[a,b]} := \frac{h}{3}\left(f(a) + 4f(a+h) + 2f(a+2h) + 4f(a+3h) + \ldots + 2f(a+(n-1)h) + f(b)\right)$$

15.3.3 Fehlerabschätzung

- Trapezsumme: $\mathcal{O}(h^2)$, $(h = \frac{b-a}{n})$
- Simpsonsumme: $\mathcal{O}(h^4)$, $(h = \frac{b-a}{n})$

15.4 Romberg-Quadratur

Erhöht durch Extrapolation die Genauigkeit der numerischen Integration. Dabei werden aus zwei berechneten Approximationen durch Trapezsumme (mit unterschiedlicher Schrittweiten) eine neue bessere Approximationen berechnet.

$$\begin{array}{lllllllll}
h = 1 & \mathcal{T}^1_{[0,1]} = T^0_1 & & & & & & & \\
 & & \searrow & & & & & & \\
h = 0.5 & \mathcal{T}^{0.5}_{[0,1]} = T^0_{0.5} & \rightarrow & T^0_{0.5} + \frac{T^0_{0.5} - T^0_1}{2^2-1} = T^1_{0.5} & & & & & \\
 & & \searrow & & \searrow & & & & \\
h = 0.25 & \mathcal{T}^{0.25}_{[0,1]} = T^0_{0.25} & \rightarrow & T^0_{0.25} + \frac{T^0_{0.25} - T^0_{0.5}}{2^2-1} = T^1_{0.25} & \rightarrow & T^1_{0.25} + \frac{T^1_{0.25} - T^1_{0.5}}{2^4-1} = T^2_{0.25} & & \\
 & & \searrow & & \searrow & & \searrow & \\
h = 0.125 & \mathcal{T}^{0.125}_{[0,1]} = T^0_{0.125} & \rightarrow & T^0_{0.125} + \frac{T^0_{0.125} - T^0_{0.25}}{2^2-1} = T^1_{0.125} & \rightarrow & T^0_{0.125} + \frac{T^1_{0.125} - T^1_{0.25}}{2^4-1} = T^2_{0.125} & \rightarrow & \ldots \\
\vdots & \vdots & & \vdots & & \vdots & & \ddots
\end{array}$$

Allgemein gilt folgende Formel zum Berechnen von T_h^k (h Schrittweite, k Anzahl der Extrapolationen):

$$T_h^k = T_h^{k-1} + \frac{T_h^{k-1} - T_{2h}^{k-1}}{2^{2k} - 1}$$

16 Iterative Verfahren

16.1 Banach'scher Fixpunktsatz

Wenn eine Funktion $\varphi(x)$ in einem Intervall selbstabbildend und eine Kontraktion ist, dann gibt es in diesem Intervall einen Fixpunkt $\overline{x}$ und für jeden Startwert in diesem Intervall konvergiert $x_{i+1} = \varphi(x_i)$ gegen $\overline{x}$.

16.2 Nullstellenbestimmung

16.2.1 Newtonverfahren

Bestimmung der Tangente am Punkt $(x_i, f(x_i))$, Schnittpunkt der Tangente mit der x-Achse ergibt x_{i+1}.

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

Lokal quadratisch konvergent.

16.2.2 Sekantenverfahren

Bestimmung der Sekante durch zwei Punkte $(x_i, f(x_i))$ und $(x_{i+1}, f(x_{i+1}))$, Schnittpunkt der Sekante mit der x-Achse ergibt x_{i+2}.

$$x_{i+2} = \frac{x_i \cdot f(x_{i+1}) - x_{i+1} \cdot f(x_i)}{f(x_{i+1}) - f(x_i)}$$

Lokal konvergent (bei Stetigkeit und einer Nullstelle im Intervall).

16.2.3 Bisektionsverfahren

Analog zur Binärsuche: Auswerten der Funktion am linken Rand (a), in der Mitte ($\frac{a+b}{2}$) und am rechten Rand (b) des Intervalls. Suche wird in dem Teilintervall fortgesetzt, bei dem die Funktionswerten an den Rändern unterschiedliche Vorzeichen haben.

Konvergiert immer gegen eine Nullstelle.

16.2.4 Regula falsi

Sekantenverfahren bei dem jeweils das Intervall mit Vorzeichenwechsel gewählt wird.

Konvergiert immer, aber langsamer als Sekantenverfahren.

16.2.5 Abbruchkriterien

- $i = i_{\max}$ (feste Anzahl von Iterationen)
- $||x_{i+1} - x_i|| < \epsilon$ (Genauigkeitsschranke für x_i)
- $|F(x_i)| < \epsilon$ (für die Nullstellensuche)
- $|\Phi(x) - x| < \epsilon$ (für die Fixpunktsuche)

16.2.6 Konvergenzordnung

Iterationsfolge x_i, exakter Wert $\overline{x}$

Lineare Konvergenz ($p = 1$) Fehler verkleinert sich pro Iteration linear

$$\exists c < 1 : |x_{i+1} - \overline{x}| \leq c \cdot |x_i - \overline{x}|$$

Höhere Konvergenzordnungen Fehler verkleinert sich pro Iteration um eine Potenz

$$\exists c < 1 : |x_{i+1} - \overline{x}| \leq c \cdot |x_i - \overline{x}|^p$$

- Fixpunktiteration: mindestens linear konvergent
- Newtonverfahren: quadratisch konvergent für einfache Nullstellen; linear konvergent für mehrfache Nullstellen
- Sekantenverfahren: $p = 1.62$
- Bisektionsverfahren und regula falsi: etwa linear konvergent

16.3 Lineare Gleichungssysteme

16.3.1 Vektoriteration

Die Vektoriteration ist eine Fixpunktiteration für eine lineare Abbildung:

$$\Phi(x) = Vx + d$$

$$x_{i+1} = Vx_i + d \quad \text{Iterationsvorschrift}$$

Konvergiert falls alle Eigenwerte von V betragsmäßig kleiner 1 sind.
Zum Lösen eines LGS wird $x^* = Vx + d$ auf $Ax = b$ angewendet und es gibt sich die Fixpunktiteration.

$$x_{i+1} = (E - B^{-1}A)x_i + B^{-1}b$$

16.3.2 Jakobi-Verfahren

Das Jakobi-Verfahren verwendet für B den Diagonalanteil von A:

$$B = diag(A) =: D$$

Dies ergibt mit $A = L + D + R$ (L untere Dreiecksmatrix, D Diagonalmatrix, R obere Dreiecksmatrix) folgende Iterationsvorschrift (V_J ist die Iterationsmatrix des Jakobi-Verfahrens):

$$\begin{aligned} x_{i+1} &= (E - D^{-1}A) \cdot x_i + D^{-1}b \\ x_{i+1} &= \underbrace{-D^{-1}(L + R)}_{V_j} \cdot x_i + D^{-1}b \end{aligned}$$

Konvergiert falls der Spektralradius r kleiner 1 ist: $r(V_j) < 1$. Es gelten allerdings vereinfachende hinreichende Kriterien für die Konvergenz:

- $|||V_j||| < 1$ (mit beliebiger Norm)
- Starkes Zeilensummenkriterium: Diagonalelement in jeder Zeile betragsmäßig größer als Summe der Beträge der anderen Einträge
- Starkes Spaltensummenkriterium: analog
- Schwaches Zeilen- und schwaches Spaltensummenkriterium

Zur Berechnung verwendet man nicht die Matrixdarstellung, sondern folgende Vorschrift (i Iteration, j Zeile):

$$x_j^{i+1} = \frac{b_j - \sum_{k \neq j} a_{jk} x_k^i}{a_{jj}}$$

Es wird dabei also jeweils die Werte in der Matrix in einer Zeile mit dem entsprechendem Wert von x_i multipliziert, von b abgezogen und durch den Wert in der Matrix auf der Diagonalen geteilt.

16.3.3 Gauß-Seidel-Verfahren

Ähnlich zum Jakobi-Verfahren, allerdings wird $B = L + D$ verwendet:

$$x_{i+1} = (E - (L+D)^{-1}A) \cdot x_i + (L+D)^{-1}b$$
$$x_{i+1} = \underbrace{-(L+D)^{-1}R}_{V_{GS}} \cdot x_i + (L+D)^{-1}b$$

Gleiche Konvergenzkriterien wie beim Jakobi-Verfahren mit einem Zusatz: Falls A positiv definit ist konvergiert es ebenfalls. Auch gilt für den Spektralradius $r(V_{GS}) = r(V_J)^2$, somit konvergiert das Gauß-Seidel Verfahren schneller als das Jakobi-Verfahren.

Zur Berechnung verwendet folgende Vorschrift (i Iteration, j Zeile):

$$x_j^{i+1} = \frac{b_j - \sum_{k=1}^{j-1} a_{jk} x_k^{i+1} - \sum_{k=j+1}^{n} a_{jk} x_k^i}{a_{jj}}$$

Im Unterschied zum Jakobi-Verfahren werden neu berechnete Werte für x_j^{i+1} zur Berechnung der folgenden x_k^{i+1}-Werte ($k = j+1$, $k = j+2$, ...) verwendet, und nicht wie beim Jakobi-Verfahren erst für x_j^{i+2}.

Bei der Programmierung braucht man somit keine Kopie des Arrays für die x-Werte sondern kann auf einem Array arbeiten.

16.4 SOR-Verfahren

Das SOR-Verfahren sorgt für eine beschleunigte Konvergenz, indem man mithilfe eines *Relaxationsparameters* ω die „Richtung“ des Verfahrens beeinflusst. Hier wird mithilfe des gegebenen Parameters der neue und alte x-Wert gewichtet.

$$x_j^{i+1} = (1-\omega) \cdot x_j^i + \omega \cdot \left(\frac{b_j - \sum_{k=1}^{j-1} a_{jk} x_k^{i+1} - \sum_{k=j+1}^{n} a_{jk} x_k^i}{a_{jj}} \right)$$

16.5 Partielle DGL

Laplace-Gleichung: $\Delta u = 0$ (mit $\Delta = \sum_{i=1}^{d} \frac{\partial^2}{\partial x_i^2}$); für 2 Dimensionen gilt $\Delta u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}$

Poisson-Gleichung: $\Delta u = f$ (mit beliebiger Funktion f)

Ω eingeschränkter Definitionsbereich auf dem die DGL gilt, $\partial\Omega$ Rand des Definitionsbereichs. Nebenbedingung kann z. B. sein, dass eine Funktion g auf dem Rand gilt: $u|_{\partial\Omega} = g|_{\partial\Omega}$ (Dirichlet-Nebenbedingung)

Diskretisierung von Ω mit Maschenabstand h.

Die Ableitungen lassen sich im $\mathbb{R}^2$ folgendermaßen approximieren (diskretisiert):

$$\frac{\partial^2 u}{\partial x^2} = \frac{u(x-1,y) - 2u(x,y) + u(x+1,y)}{h^2}$$
$$\frac{\partial^2 u}{\partial y^2} = \frac{u(x,y-1) - 2u(x,y) + u(x,y+1)}{h^2}$$

Insgesamt (diskretisiert):

$$\Delta u = \frac{-4u(x,y) + u(x-1,y) + u(x+1,y) + u(x,y-1) + u(x,y+1)}{h^2}$$

Damit kann die Systemmatrix aufgestellt werden, die dann z. B. mit dem Gauß-Seidel gelöst werden kann.

17 Nichtlineare Optimierung

17.1 Lagrange Multiplikatoren

Ziel: Minima von $f : \mathbb{R}^n \mapsto \mathbb{R}$ unter m Nebenbedingungen $g_1(x) = 0, g_2(x) = 0, \ldots, g_m(x) = 0$ finden.

Am Extremum sind die Gradienten der Funktion F und die Nebenbedingungen g_i parallel zueinander. Wegen ihrer unterschiedlichen Länge führt man noch Skalare λ_i ein (die Lagrange Multiplikatoren):

$$\nabla F(x) = \lambda_1 \nabla g_1(x) + \ldots + \lambda_m \nabla g_m(x)$$

$$\nabla F(x) + \sum_{i=1}^{m} \lambda_i g_i(x) = 0$$

17.2 Sekantenverfahren

Man hat 2 Punkte x_0 und x_1 als Startwerte gegeben.
Graphisch:

- Verbinde $f(x_0)$ und $f(x_1)$
- Am Schnittpunkt mit der x-Achse ist der neue Punkt.

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

17.3 Newton-Verfahren im $\mathbb{R}^n$

Zur Nullstellenbestimmung von $F : \mathbb{R}^n \mapsto \mathbb{R}^n$, mit Jakobi-Matrix J von F.

$$x_{i+1} = x_i - J^{-1}(x_i) F(x_i)$$

17.4 Abstiegsverfahren

Startwert $x_0 \in \mathbb{R}^n$, Schrittweite t_0, Suchrichtung $s_0 \in \mathbb{R}^n$. Falls $\tau_i > 0$ und $s_i \cdot \nabla F(x_i) < 0$ so spricht man von einem Abstiegsverfahren. Iterationsvorschrift:

$$x_{i+1} = x_i + \tau_i \cdot s_i$$

Schrittweite und Suchrichtung muss für jeden Schritt neu berechnet werden.

17.5 Gradientenverfahren

In jedem Schritt muss zuerst Suchrichtung $s_i = -\text{grad}(Q)$ und dann Schrittweite τ_i bestimmt werden.

$$\tau_i = -\frac{(Ax_i + b)^T s_i}{s_i^T A\ s_i}$$

Der Gradient der Matrix ist $\text{grad}(Q) = 2(Ax + b)$. Zur Berechnung des neuen Wertes wird die Formel des Abstiegsverfahrens benutzt.

Goldener-Schnitt-Verfahren

- Starte mit 3 Werten $a < b < c$ für die gilt: $f(a) > f(b) < f(c)$.
- Wähle neuen Punkt d: $a < d < c$.
- Ist $f(d) < f(b)$ wählt man das Tripel mit d als mittleren Punkt, sonst mit b als mittleren Punkt.

d wird so gewählt, dass es die Strecke im Goldenen-Schnitt teilt.

Quadratisches Funktional Für ein quadratisches Funktional der Form $F(x) = x^T Ax + 2b^T x + c$, positiv definite $n \times n$-Matrix A, n-Vektor b, Konstante c gilt exakt:

$$s_i = -2(Ax_i + b)$$
$$t_i = -\frac{s_i^T b + s_i^T Ax_i}{s_i^T As_i}$$

17.6 cg-Verfahren

u und v sind konjugierte Richtungen falls gilt: $u^T Av = 0$

Das cg-Verfahren (conjugate gradients) findet theoretisch in n Schritten (im $\mathbb{R}^n$) das Minimum, wegen Rundungsfehler wird es aber häufig iterativ angewendet.

Erster Schritt mit dem Gradientenverfahren, dann Bestimmung von s_1 durch Lösen der Gleichung $s_1^T\ A\ s_0 = 0$, also $s_{neu} = s_{alt}$ konjugiert zu A.

18 Komplexitäten

18.1 Rechenaufwand

Anmerkung: Bezogen immer auf $(n \times n)$-Matrizen bzw. n-Vektoren.

- Matrix mal Vektor: $\mathcal{O}(n^2)$
- Matrix mal Matrix: $\mathcal{O}(n^3)$
- Matrix mal Vektor, wenn Matrix von Rang 1: $\mathcal{O}(3n)$
- Zwei tridiagonale Matrizen: $\mathcal{O}(n^2)$
- Skalarprodukt/Inneres Produkt zweier Vektoren: $\mathcal{O}(2n)$
- Vektorprodukt/Kreuzprodukt/Äußeres Produkt zweier Vektoren: $\mathcal{O}(3n)$
- Gauß'sches Eliminationsverfahren: $\mathcal{O}(n^3)$
- LR-Lösen einer Matrix: $\mathcal{O}(n^3)$ (mit Rückwärtseinsetzen)
- LR-Lösen einer m-diagonalen Matrix: $\mathcal{O}(n \cdot m + n^2)$
- QR-Lösen: $\mathcal{O}(n^3)$, n^2 Multiplikationen, $n(n-1)$ Additionen.

- Least Square: $\mathcal{O}(n^3)$
- Vorwärts-/Rückwärtseinsetzen: $\mathcal{O}(n^2)$
- Lösen für Tridiagonalmatrizen: $\mathcal{O}(n)$
- Lösung mittels THOMAS-Algorithmus: $\mathcal{O}(n)$ (für tridiagonale Matrizen)
- Berechnen des Betrags der Determinante, wenn LR-/QR-Zerlegung bekannt ist: $\mathcal{O}(n)$
- Ein JACOBI-Schritt: $\mathcal{O}(2n^2)$, ein GAUSS-SEIDEL-Schritt: $\mathcal{O}(2n^2)$
- Auswerten eines einzelnen DE CASTELJAU-Schrittes mit n Kontrollpunkten: $\mathcal{O}(\frac{n^2}{2})$
- Midpoint-Subdivision mit n Kontrollpunkten pro Rekursionsschritt: $\mathcal{O}(\frac{n^2}{2})$
  Vorsicht! Bei jedem Rekursionsschritt entstehen $n-1$ zusätzliche Punkte!
- AITKEN-NEVILLE: $\mathcal{O}(n^2)$
- Filtern mit $(k \times k)$-Filter:
  Multiplikationen: separierbar: $\mathcal{O}(2 \cdot k \cdot n^2)$, nicht separierbar: $\mathcal{O}(k^2 \cdot n^2)$
  Alle Operationen: separierbar: $\mathcal{O}(2 \cdot (2 \cdot k \cdot n^2))$, nicht separierbar: $\mathcal{O}(2 \cdot (k^2 \cdot n^2))$
- Auswerten eines Polynoms vom Grad n mittels HORNER-Schema: $\mathcal{O}(n)$

18.2 Fehler

Anmerkung: Bezogen immer auf äquidistante Schrittweite h.

- Nearest Neighbor: $\mathcal{O}(h)$
- Stückweise lineare Interpolation: $\mathcal{O}(h^2)$
- Mittelpunktsregel: $\mathcal{O}(h^3)$
- Trapezregel: $\mathcal{O}(h^3)$
- SIMPSON-Regel: $\mathcal{O}(h^5)$
- Newtons 3/8-Regel: $\mathcal{O}(h^5)$
- Trapezsumme: $\mathcal{O}(h^2)$
- Simpsonsumme: $\mathcal{O}(h^4)$
- Catmull-Rom: $\mathcal{O}(h^3)$

18.3 Konvergenzordung

- Monte-Carlo: $\mathcal{O}(n^{-\frac{1}{2}})$
- Fixpunktiteration: mindestens linear konvergent
- NEWTON: quadratisch konvergent für einfache, linear konvergeht für mehrfache Nullstellen
- Sekantenverfahren: $p = 1.62$
- Bisektionsverfahren und regula falsi: etwa linear konvergent