Numerische Mathematik für Maschinenbauer und Umwelttechniker

Vorlesungsskriptum WS 2006/07

R. Verfürth

Fakultät für Mathematik, Ruhr-Universität Bochum

Inhaltsverzeichnis

Einleitung 5

Kapitel I. Lineare Gleichungssysteme 11
I.1. Lineare Gleichungssysteme und Matrizen 11
I.2. Das Gaußsche Eliminationsverfahren 12
I.3. Aufwand des Gaußschen Eliminationsverfahrens 16
I.4. Die LR-Zerlegung 18
I.5. Symmetrische positiv definite Matrizen 21
I.6. Das Cholesky-Verfahren 23
I.7. Große Gleichungssysteme 26
I.8. Störungsrechnung 28

Kapitel II. Nicht lineare Gleichungssysteme 33
II.1. Das Bisektionsverfahren 33
II.2. Das Sekantenverfahren 34
II.3. Das Newtonverfahren für Funktionen einer Variablen 36
II.4. Eigenschaften des Newtonverfahrens 38
II.5. Das Newtonverfahren für Funktionen mehrerer Variabler 40
II.6. Dämpfung 42

Kapitel III. Interpolation 45
III.1. Lagrange-Interpolation 45
III.2. Die Newtonsche Darstellung des Lagrangeschen Interpolationspolynomes 45
III.3. Genauigkeit der Lagrange-Interpolation 48
III.4. Hermite-Interpolation 51
III.5. Kubische Spline-Interpolation 53
III.6. Bézier-Darstellung von Polynomen und Splines 58

Kapitel IV. Integration 63
IV.1. Motivation 63
IV.2. Quadraturformeln 64
IV.3. Newton-Cotes-Formeln 65
IV.4. Gauß-Formeln 65
IV.5. Zusammengesetzte Quadraturformeln 65
IV.6. Das Romberg-Verfahren 68
IV.7. Spezielle Integranden 70
IV.8. Mehrdimensionale Integrationsbereiche 71

Kapitel V. Eigenwertprobleme 75
V.1. Übersicht 75
V.2. Die Potenzmethode 75
V.3. Rayleigh-Quotienten 77
V.4. Inverse Iteration von Wielandt 78
V.5. Inverse Rayleigh-Quotienten-Iteration 80
V.6. Berechnung eines Eigenvektors 81

Kapitel VI. Gewöhnliche Differentialgleichungen 83
VI.1. Motivation 83
VI.2. Einschrittverfahren 86
VI.3. Verfahrensfehler und Ordnung 87
VI.4. Runge-Kutta-Verfahren 88
VI.5. Stabilität 91
VI.6. Steife Differentialgleichungen 94
VI.7. Schrittweitensteuerung 96
VI.8. Mehrschrittverfahren 101

Zusammenfassung 103

Index 107

Einleitung

Die numerische Mathematik entwickelt und untersucht Methoden zur numerischen, d.h. computerunterstützten Lösung praktischer Probleme. Dabei besteht eine beständige Interaktion zwischen dem Anwender, der z.B. seine Kenntnisse über das Anwendungsproblem und seine (mathematische) Modellierung einfließen lässt, anderen Bereichen der Mathematik, die z.B. Aussagen über Existenz, Eindeutigkeit und Eigenschaften der Lösungen des mathematischen Problems liefern, und der Numerik. Wesentliche Gesichtspunkte bei der Entwicklung oder Auswahl eines numerischen Verfahrens sind z.B. Effizienz und Stabilität des Verfahrens, d.h. die Frage, welchen Aufwand – computer- und manpower – das Verfahren erfordert und wie empfindlich die Lösung von Fehlern bei der Bestimmung der Eingabedaten und den durchzuführenden Rechenoperationen abhängt. Bei der Beurteilung eines Verfahrens spielt natürlich die ursprüngliche Fragestellung eine erhebliche Rolle. So ist es z.B. sinnlos, mit einem numerischen Verfahren die Temperatur im Innern eines Stahlblockes auf 1/1000 Grad genau zu bestimmen, wenn die gemessene Oberflächentemperatur nur auf 1 – 10 Grad genau bekannt ist!

Eine wesentliche Rolle bei der Entwicklung und Beurteilung eines numerischen Verfahrens spielt die Tatsache, dass man in einem Computer stets nur mit Zahldarstellungen mit endlich vielen Ziffern arbeiten kann. Daher wollen wir im Folgenden kurz auf diese endlichen Zahldarstellungen und die daraus resultierenden Rundungsfehler eingehen.

Zahlen werden in NORMALISIERTER FORM

$$\sigma\, 0.a_1 \dots a_t\, b^c$$

dargestellt. Dabei ist

$b \in \mathbb{N}^*$	die BASIS,
$t \in \mathbb{N}^*$	die MANTISSENLÄNGE,
$a_i \in \mathbb{N}_{b-1} = \{0, \dots, b-1\}$	die ZIFFERN, $a_1 \neq 0$,
$c \in \mathbb{Z}$	der EXPONENT,
$\sigma \in \{-1, +1\}$	das VORZEICHEN.

Gebräuchliche Basen sind

$b = 10$	DEZIMALSYSTEM,
$b = 2$	DUALSYSTEM,
$b = 16$	HEXADEZIMALSYSTEM.

Übliche Mantissenlängen bezogen auf die Basis 10 sind:

$$t = 8 \quad \text{oder} \quad t = 15.$$

Der Einfachheit halber beschränken wir uns im Folgenden auf das Dezimalsystem. Mit $\mathrm{rd}(x)$ bezeichnen wir die endliche Darstellung der Zahl x. Für

$$x = \sigma 0.a_1 \dots a_t a_{t+1} \dots 10^c$$

ist also

$$\mathrm{rd}(x) = \begin{cases} \sigma 0.a_1 \dots a_t \, 10^c & \text{falls } a_{t+1} < 5, \\ \sigma 0.a_1 \dots (a_t + 1) \, 10^c & \text{falls } a_{t+1} \geq 5. \end{cases}$$

Mithin ist

$$|x - \mathrm{rd}(x)| \leq 5 \cdot 10^{c-t-1}$$

und

$$\frac{|x - rd(x)|}{|x|} \leq 5 \cdot 10^{-t}.$$

Die Zahl

$$\mathrm{eps} = 5 \cdot 10^{-t}$$

heißt Maschinengenauigkeit.

Im Folgenden bezeichnen wir mit $\cong$ Ergebnisse, die bis auf Terme der Ordnung eps^2 oder höher genau sind. Für die Addition und Multiplikation von zwei Zahlen ergibt sich damit z.B.

$$\begin{aligned} \mathrm{rd}(x + y) &= [x(1 + \mathrm{eps}) + y(1 + \mathrm{eps})](1 + \mathrm{eps}) \\ &= (x + y)(1 + \mathrm{eps})^2 \\ &\cong (x + y)(1 + 2\,\mathrm{eps}) \end{aligned}$$

und

$$\begin{aligned} \mathrm{rd}(x \cdot y) &= [x(1 + \mathrm{eps}) \cdot y(1 + \mathrm{eps})](1 + \mathrm{eps}) \\ &= xy(1 + \mathrm{eps})^3 \\ &\cong xy(1 + 3\,\mathrm{eps}), \end{aligned}$$

d.h., der relative Fehler beträgt $2\,\mathrm{eps}$ bzw. $3\,\mathrm{eps}$.

Beispiel .1. Für $b = 10$ und $t = 8$ erhält man mit

$$\begin{aligned} a &= 0.23371258 \cdot 10^{-4} \\ b &= 0.33678429 \cdot 10^{2} \\ c &= -0.33677811 \cdot 10^{2} \end{aligned}$$

die Ergebnisse

$$\begin{aligned} a + (b + c) &= 0.23371258 \cdot 10^{-4} + 0.61800000 \cdot 10^{-3} \\ &= 0.64137126 \cdot 10^{-3} \end{aligned}$$

und

$$\begin{aligned}(a+b)+c &= 0.33678452 \cdot 10^2 - 0.33677811 \cdot 10^2 \\ &= 0.64100000 \cdot 10^{-3}\end{aligned}$$

gegenüber dem exakten Ergebnis

$$0.641371258 \cdot 10^{-3}.$$

Der relative Fehler beträgt also $2 \cdot 10^{-9}$ bzw. $3 \cdot 10^{-4}$. Dieses Beispiel zeigt, dass es ratsam ist, Zahlen möglichst gleicher Größenordnung zu addieren, um Auslöschung zu vermeiden.

Beispiel .2. Zu lösen ist das LGS

$$\begin{aligned}0.780x_1 + 0.563x_2 &= 0.217 \\ 0.913x_1 + 0.659x_2 &= 0.254\end{aligned}$$

mit $b = 10$ und $t = 3$. Die exakte Lösung lautet

$$x = (1, -1).$$

Wir erhalten mit

- Cramerscher Regel: LGS nicht lösbar, da $\det A = 0$,
- Gaußelimination: $x = (0.278, 0.0)$,
- Gaußelimination mit Spalten- oder totaler Pivotisierung: LGS nicht lösbar.

Diese völlig unzureichenden Ergebnisse liegen an der Kondition von ca. $2 \cdot 10^6$ der Matrix. Daher werden die Ergebnisse – unabhängig vom verwendeten Lösungsalgorithmus – erst besser, wenn man zu einer 8- oder 9-stelligen Arithmetik übergeht.

Beispiel .3. Zu berechnen ist $\ln(2)$. Aus

$$-\ln(1-x) = \sum_{n=1}^{\infty} \frac{x^n}{n} \quad \forall x \in [-1, 1)$$

folgt

$$\begin{aligned}\ln(2) &= \sum_{n=1}^{\infty} \frac{(-1)^{n-1}}{n} \\ &= \sum_{n=1}^{m} \frac{(-1)^{n-1}}{n} + R_m\end{aligned}$$

mit

$$\frac{1}{2m} \le |R_m| \le \frac{1}{m}.$$

Damit folgt für $b = 10$ und $t = 8$

$$\begin{aligned}\left|\ln(2) - \mathrm{rd}\left(\sum_{n=1}^{m}\frac{(-1)^{n-1}}{n}\right)\right| &\leq \underbrace{|R_m|}_{\text{Abbruchfehler}} \\ &\quad + \underbrace{\left|\sum_{n=1}^{m}\frac{(-1)^{n-1}}{n} - \mathrm{rd}\left(\sum_{n=1}^{m}\frac{(-1)^{n-1}}{n}\right)\right|}_{\text{Rundungsfehler}} \\ &\leq \frac{1}{m} + m \cdot 10^{-8}.\end{aligned}$$

Die rechte Seite wird für $m = 10^4$ minimal und liefert den Gesamtfehler $2 \cdot 10^{-4}$. Dieses Beispiel zeigt das typische Verhalten, dass mit größer werdendem m der Abbruchfehler abnimmt, während der Rundungsfehler zunimmt. Insgesamt kann der Gesamtfehler nicht unter eine von der verwendeten Arithmetik abhängigen Schranke – hier $2 \cdot 10^{-4}$ – gedrückt werden.
Andererseits folgt aus

$$\frac{1}{2}\ln(\frac{1+x}{1-x}) = \sum_{n=1}^{\infty}\frac{x^{2n-1}}{2n-1} \quad \forall x \in [-1, 1)$$

für $x = \frac{1}{3}$

$$\begin{aligned}\ln(2) &= 2\sum_{n=1}^{\infty}\frac{3^{-(2n-1)}}{2n-1} \\ &= 2\sum_{n=1}^{m}\frac{3^{-(2n-1)}}{2n-1} + \tilde{R}_m\end{aligned}$$

mit

$$\begin{aligned}|\tilde{R}_m| &\leq \frac{2}{(2m+1)3^{2m+1}}\sum_{k=0}^{\infty}3^{-2k} \\ &= \frac{2}{3(2m+1)}\frac{9}{8} \cdot 9^{-m} \\ &\leq 10^{-7} \qquad\qquad \text{für } m \geq 6.\end{aligned}$$

Also erhalten wir mit diesem Verfahren schon mit 6 Summanden einen Abbruch- und Gesamtfehler von 10^{-7}. Dieses Verfahren ist also offensichtlich viel genauer und effizienter als das erste.

Zusammenfassend kann man folgende Forderungen an ein gutes numerisches Verfahren aufstellen:

- ÖKONOMIE: Der Aufwand sollte möglichst gering sein. Ein Maß hierfür ist die Zahl der benötigten arithmetischen Operationen.
- STABILITÄT: Ein Verfahren sollte möglichst unempfindlich gegen Rundungsfehler sein und a priori und a posteriori Abschätzungen des Fehlers liefern.
- ANWENDUNGSBEREICH: Er sollte möglichst groß sein.

Im Folgenden geben wir zu vielen Algorithmen entsprechende `Java`-Programme an. Unter der Adresse

`http://www.ruhr-uni-bochum.de/num1/demo/index.html`

findet man das Java-Applet `Numerics` mitsamt Benutzeranleitung, das die im Folgenden vorgestellten (und einige zusätzliche) Algorithmen demonstriert.

KAPITEL I

Lineare Gleichungssysteme

I.1. Lineare Gleichungssysteme und Matrizen

Ein LINEARES GLEICHUNGSSYSTEM MIT n GLEICHUNGEN UND n UNBEKANNTEN $x_1, \ldots, x_n$, kurz LGS, hat die Form

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n &= b_2 \\ &\vdots \\ a_{n1}x_1 + a_{m2}x_2 + \ldots + a_{nn}x_n &= b_n \end{aligned} \tag{I.1}$$

mit den KOEFFIZIENTEN a_{ij} und den ABSOLUTGLIEDERN b_i. (Kommt eine Unbekannte in einer Gleichung nicht vor, hat sie dort den Koeffizienten 0.) Für ein solches LGS schreibt man

$$\begin{pmatrix} a_{11} & \ldots & a_{1n} \\ \vdots & & \vdots \\ a_{n1} & \ldots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}$$

oder kurz

$$A\mathbf{x} = \mathbf{b}$$

mit der KOEFFIZIENTENMATRIX $A = (a_{ij})_{n \times n}$, dem Spaltenvektor $\mathbf{x} \in \mathbb{R}^n$ mit den unbekannten Komponenten x_i und dem Spaltenvektor $\mathbf{b} \in \mathbb{R}^n$ der RECHTEN SEITE.

Ein LGS heißt HOMOGEN, wenn $\mathbf{b} = 0$ ist. Andernfalls heißt es INHOMOGEN.

Ein Spaltenvektor $\mathbf{c}$ mit den Komponenten $c_1, \ldots, c_n$ heißt eine Lösung des LGS (I.1), wenn für $x_i = c_i (i = 1, \ldots, n)$ alle n Gleichungen erfüllt sind, d.h. wenn $A\mathbf{c} = \mathbf{b}$ gilt.

Ein homogenes LGS $A\mathbf{x} = \mathbf{0}$ besitzt stets mindestens eine Lösung, nämlich den Nullvektor. Diese Lösung nennt man auch die TRIVIALE LÖSUNG.

Nicht jedes LGS besitzt eine Lösung. Es tritt stets einer der folgenden drei Fälle auf:

- Das Gleichungssystem besitzt KEINE Lösung.
- Das Gleichungssystem besitzt GENAU EINE Lösung.

- Das Gleichungssystem besitzt UNENDLICH VIELE Lösungen.

BEISPIEL I.1. (zwei parallele Geraden)

$$\begin{aligned} 2x_1 + x_2 &= 1 \\ 2x_1 + x_2 &= 2. \end{aligned}$$

Dieses LGS ist nicht lösbar.

BEISPIEL I.2. (zwei nicht parallele Geraden)

$$\begin{aligned} 2x_1 + x_2 &= 1 \\ x_1 + x_2 &= 2 \end{aligned}$$

Hier ist $x_1 = -1$, $x_2 = 3$ die eindeutige Lösung.

BEISPIEL I.3.

$$\begin{aligned} 2x_1 + x_2 &= 1 \\ 4x_1 + 2x_2 &= 2. \end{aligned}$$

Hier ist $x_1 = a$, $x_2 = 1 - 2a$ für jedes $a \in \mathbb{R}$ eine Lösung.

I.2. Das Gaußsche Eliminationsverfahren

Das bekannteste und wichtigste Verfahren zur Lösung linearer Gleichungssysteme $A\mathbf{x} = \mathbf{b}$ ist das Gaußsche Eliminationsverfahren. Zu seiner Beschreibung ist es zweckmäßig, die Koeffizientenmatrix A und die rechte Seite $\mathbf{b}$ zur erweiterten Koeffizientenmatrix $(A, \mathbf{b})$ zusammenzufassen, indem man $\mathbf{b}$ als $(n + 1)$-ten Spaltenvektor hinzufügt.

Das Gaußsche Eliminationsverfahren beruht auf folgenden Grundoperationen:

- Vertauschen zweier Zeilen von $(A, \mathbf{b})$.
- Multiplikation einer Zeile von $(A, \mathbf{b})$ mit einer von Null verschiedenen Zahl.
- Addition bzw. Subtraktion des α-fachen einer Zeile von $(A, \mathbf{b})$ von einer anderen Zeile.

Diese Operationen ändern die Lösungsmenge des LGS nicht, d.h. geht $(B, \mathbf{c})$ durch solche Operationen aus $(A, \mathbf{b})$ hervor, so besitzt das LGS $B\mathbf{x} = \mathbf{c}$ genau die gleichen Lösungen wie das LGS $A\mathbf{x} = \mathbf{b}$.

Das Gaußsche Eliminationsverfahren besteht aus zwei Etappen:

- dem ELIMINATIONSTEIL,
- dem RÜCKLÖSUNGSTEIL.

Am Ende des ELIMINATIONSTEILS hat die erweiterte Koeffizientenmatrix folgende Struktur

$$
\begin{matrix} r \left\{ \begin{matrix} \\ \\ \\ \\ \\ \end{matrix} \right. \\ n-r \left\{ \begin{matrix} \\ \\ \\ \end{matrix} \right. \end{matrix}
\underbrace{\begin{pmatrix}
\bullet & * & * & \dots & \dots & \dots & \dots & \dots & * & * \\
0 & \bullet & * & \dots & \dots & \dots & \dots & \dots & * & * \\
0 & 0 & \bullet & \dots & \dots & \dots & \dots & \dots & * & * \\
\vdots & \vdots & & \ddots & \vdots & \vdots & & & \vdots & \vdots \\
\vdots & \vdots & & & \bullet & * & \dots & \dots & * & * \\
0 & \dots & & & \dots & 0 & * & \dots & * & * \\
\vdots & & & & & \vdots & \vdots & & \vdots & \vdots \\
0 & \dots & & & \dots & 0 & * & \dots & * & *
\end{pmatrix}}_{n+1} .
$$

Dabei steht $\bullet$ für eine von Null verschiedene Zahl und $*$ für eine beliebige Zahl.

Das LGS ist genau dann eindeutig lösbar, wenn $r = n$ ist.

Im Eliminationsteil wird kontrolliert, ob $r = n$ ist. Falls $r < n$ ist, wird das Verfahren mit einer Fehlermeldung abgebrochen.

Im RÜCKLÖSUNGSTEIL wird die Lösung des LGS ermittelt.

ALGORITHMUS I.4. GAUSSSCHES ELIMINATIONSVERFAHREN *für* $A\mathbf{x} = \mathbf{b}$.

(1) ELIMINATIONSTEIL:
   (a) *Setze* $i = 1$.
   (b) (PIVOTSUCHE) *Bestimme einen Index* i_0 *mit* $i \le i_0 \le n$ *und* $a_{i_0 i} \neq 0$. *Falls kein derartiger Index existiert, beende das Verfahren mit einer entsprechenden Fehlermeldung, andernfalls fahre mit Schritt* (c) *fort.*
   (c) (ZEILENTAUSCH) *Falls* $i_0 \neq i$ *ist, vertausche die Zeilen* i *und* i_0 *von* $(A, \mathbf{b})$.
   (d) (ELIMINATION) *Für* $k = i+1, \dots, n$ *subtrahiere das* $\frac{a_{ki}}{a_{ii}}$-*fache der* i-*ten Zeile von* $(A, \mathbf{b})$ *von der* k-*ten Zeile von* $(A, \mathbf{b})$.
   (e) *Falls* $i < n-1$ *ist, erhöhe* i *um* 1 *und gehe zu Schritt* (b) *zurück. Im Fall* $i = n-1$ *prüfe, ob* $a_{nn} = 0$ *ist. Falls* $a_{nn} = 0$ *ist, beende das Verfahren mit einer entsprechenden Fehlermeldung. Andernfalls fahre mit dem Rücklösungsteil fort.*

(2) RÜCKLÖSUNGSTEIL: *Setze*

$$x_n = \frac{b_n}{a_{nn}}$$

und berechne für $i = n-1, n-2, \dots, 1$ sukzessive

$$x_i = \frac{1}{a_{ii}}\{b_i - a_{in}x_n - a_{in-1}x_{n-1} - \ldots - a_{ii+1}x_{i+1}\}.$$

Der Vektor

$$\mathrm{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}$$

ist die eindeutige Lösung des LGS. Beende das Verfahren.

Bemerkung I.5. (1) Bei der Implementierung des Gaußschen Eliminationsverfahrens sollte man bei der Pivotsuche den Index i_0 so bestimmen, dass

$$|a_{i_0 i}| = \max\{|a_{ki}| : i \le k \le n\}$$

ist. Die Abfrage „$a_{i_0 i} = 0$?“ ist zu ersetzen durch „$|a_{i_0 i}| \le \varepsilon$?“, wobei ε eine kleine positive Zahl ist. Die Wahl von ε hängt von der Rechengenauigkeit ab. Typische Werte sind $\varepsilon = 10^{-6}$ oder $\varepsilon = 10^{-8}$.
(2) Das LGS

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \mathbf{x} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

mit der exakten Lösung

$$\mathbf{x} = \begin{pmatrix} 2 \\ 1 \end{pmatrix}$$

zeigt, dass Algorithmus I.4 ohne die Pivotsuche in Schritt (b) auch bei exakter Arithmetik versagt.

Das folgende Java-Programm realisiert Algorithmus I.4:

```
// Gaussian elimination
    public void gaussElimination() throws LinearAlgebraException {
        elimination();
        backSolve();
    }    // end of gaussElimination
// elimination part of Gaussian elimination
    public void elimination() throws LinearAlgebraException {
        for ( int i = 0; i < dim-1; i++) {
            int jp = pivot(i);          // find pivot, j is pivotelement
            if ( Math.abs( a[jp][i] ) < EPS )         // pivot = 0 ??
                throw new LinearAlgebraException(
                " WARNING: zero pivot in "+i+"-th elimination step");
            if ( jp > i ) {                 // swap equations i and j
                swap(b, i, jp);
                swap(a, i, jp);
            }
            for ( int k = i+1; k < dim; k++ )         // elimination
                substractEquations(k, i, a[k][i]/a[i][i]);
        }
        if ( Math.abs( a[dim-1][dim-1] ) < EPS )
            throw new LinearAlgebraException(
            " WARNING: zero pivot in "+(dim-1)+"-th elimination step");
    }    // end of elimination
// find pivot
```

```
    public int pivot(int i) {
        int pivotElement = i;
        double pivotValue = Math.abs( a[i][i] );
        for ( int j = i+1; j < dim; j++ ) {
            if ( Math.abs( a[j][i] ) > pivotValue ) {
                pivotElement = j;
                pivotValue = Math.abs( a[j][i] );
            }
        }
        return pivotElement;
    }    // end of pivot
// substract factor times equation i from equation k
    public void substractEquations(int k, int i, double factor) {
        b[k] -= b[i]*factor;
        for( int l = i+1; l < dim; l++ )
            a[k][l] -= a[i][l]*factor;
    }    // end of substractEquations
// solution part of Gaussian elimination
    public void backSolve() {
        x[dim-1] = b[dim-1]/a[dim-1][dim-1];
        for ( int i = dim-2; i >= 0; i-- )
            x[i] = (b[i] - innerProduct(a[i], x, i+1, dim))/a[i][i];
    }    // end of backSolve
// swap components i and j of vector u
    public void swap(double[] u, int i, int j) {
        double v = u[i];
        u[i] = u[j];
        u[j] = v;
    } // end of swap
// inner product of sections first, ..., last-1 of two vectors
    public double innerProduct(double[] u, double[] v, int first,
        int last) {
        double prod = 0;
        for( int i = first; i < last; i++ )
            prod += u[i]*v[i];
        return prod;
    } // end of inner product
```

Die folgenden beiden Beispiele verdeutlichen Algorithmus I.4.

Beispiel I.6. Betrachte das LGS

$$\begin{aligned} x_1 + 2x_2 + 3x_3 &= 4 \\ 4x_1 + 5x_2 + 6x_3 &= 0 \\ 7x_1 + 8x_2 + 10x_3 &= 4. \end{aligned}$$

Jede dieser Gleichungen beschreibt eine Ebene im Raum. Daher besteht die Lösungsmenge aus keinem Punkt, genau einem Punkt, einer Geraden oder aus einer Ebene. Es ist

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 10 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 4 \\ 0 \\ 4 \end{pmatrix}.$$

Das Gaußsche Eliminationsverfahren liefert

$$\begin{array}{ccc|c} 1 & 2 & 3 & 4 \\ 4 & 5 & 6 & 0 \\ 7 & 8 & 10 & 4 \end{array} \quad \rightarrow \quad \begin{array}{ccc|c} 1 & 2 & 3 & 4 \\ 0 & -3 & -6 & -16 \\ 0 & -6 & -11 & -24 \end{array} \quad \rightarrow \quad \begin{array}{ccc|c} 1 & 2 & 3 & 4 \\ 0 & -3 & -6 & -16 \\ 0 & 0 & 1 & 8 \end{array}$$

und

$$\begin{aligned} x_3 &= 8 \\ x_2 &= -\frac{1}{3}\{-16 + 6 \cdot 8\} \\ &= -\frac{32}{3} \\ x_1 &= 4 - 2 \cdot (-\frac{32}{3}) - 3 \cdot 8 \\ &= \frac{4}{3}. \end{aligned}$$

Die eindeutige Lösung des LGS ist

$$\begin{pmatrix} \frac{4}{3} \\ -\frac{32}{3} \\ 8 \end{pmatrix}.$$

Beispiel I.7. Wir ersetzen in Beispiel I.6 den Koefffizienten 10 von x_3 in der dritten Gleichung durch 9:

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 4 \\ 0 \\ 4 \end{pmatrix}.$$

Das Gaußsche Eliminationsverfahren liefert jetzt

$$\begin{array}{ccc|c} 1 & 2 & 3 & 4 \\ 4 & 5 & 6 & 0 \\ 7 & 8 & 9 & 4 \end{array} \quad \rightarrow \quad \begin{array}{ccc|c} 1 & 2 & 3 & 4 \\ 0 & -3 & -6 & -16 \\ 0 & -6 & -12 & -24 \end{array} \quad \rightarrow \quad \begin{array}{ccc|c} 1 & 2 & 3 & 4 \\ 0 & -3 & -6 & -16 \\ 0 & 0 & 0 & 8 \end{array}$$

und bricht wegen $a_{33} = 0$ ab. (Wegen $b_3 = 8$ hat das LGS keine Lösung.)

I.3. Aufwand des Gaußschen Eliminationsverfahrens

Algorithmus I.4 (S. 13) benötigt im i-ten Eliminationsschritt

$$\begin{aligned} (n-i)(n-i+1) &\text{ Additionen} \\ (n-i)(n-i+1) &\text{ Multiplikationen} \\ n-i &\text{ Divisionen} \end{aligned}$$

insgesamt also

$$\frac{1}{6}(n-1)n(2n-1) + \frac{1}{2}(n-1)n \text{ Additionen}$$

$$\frac{1}{6}(n-1)n(2n-1)+\frac{1}{2}(n-1)n \text{ Multiplikationen}$$
$$\frac{1}{2}(n-1)n \text{ Divisionen,}$$

wovon

$$\frac{1}{2}(n-1)n \text{ Additionen und Multiplikationen}$$

auf die Umformung der rechten Seite entfallen.

Algorithmus I.4 (S. 13) benötigt im i-ten Rücklösungsschritt zur Berechnung von x_{n-i+1}, $1 \leq i \leq n$

$$i-1 \text{ Additionen}$$
$$i-1 \text{ Multiplikationen}$$
$$1 \text{ Division}$$

insgesamt also

$$\frac{1}{2}(n-1)n \text{ Additionen}$$
$$\frac{1}{2}(n-1)n \text{ Multiplikationen}$$
$$n \text{ Divisionen.}$$

Insgesamt ergibt sich somit:

> Das Gaußsche Eliminationsverfahren benötigt $O(n^3)$ Operationen für den Eliminationsteil und $O(n^2)$ Operationen für den Rücklösungsteil.

Der Aufwand von $O(n^3)$ Operationen für das Gaußsche Eliminationsverfahren ist mit dem Aufwand von $O(n!) \approx O(n^n)$ Operationen für die Cramersche Regel zu vergleichen. Wie die Tabelle I.1 zeigt, ist die Cramersche Regel schon für kleine Werte von n dem Gaußschen Verfahren hoffnungslos unterlegen.

Tabelle I.1. Aufwand und Rechenzeit des Gaußschen Eliminationsverfahrens und der Cramerschen Regel

	Gauß-Elimination		Cramersche Regel	
n	Operationen	Rechenzeit	Operationen	Rechenzeit
10	1000	< 1 μsec	$3.6 \cdot 10^6$	3.6 μsec
15	3375	3.3 μsec	$1.3 \cdot 10^{12}$	$>$ 21 Minuten
20	8000	8 μsec	$2.4 \cdot 10^{18}$	$>$ 77 Jahre

Bei der Rechenzeit sind wir von einem Gigaflop-Rechner ausgegangen.

I.4. Die LR-Zerlegung

In der Praxis tritt häufig das Problem auf, dass man mehrere LGS $A\mathbf{x} = \mathbf{b}$ mit gleicher Matrix A und verschiedenen rechten Seiten $\mathbf{b}$ lösen muss. Da der Eliminationsteil des Gaußschen Verfahrens, Algorithmus I.4 (S. 13), $O(n^3)$-Operationen, der Rücklösungsteil aber nur $O(n^2)$-Operationen erfordert, ist es nicht sinnvoll, diesen Algorithmus auf jedes LGS separat anzuwenden.

Die LR-Zerlegung ist auf diese Problematik zugeschnitten. Die Idee besteht darin, in Algorithmus I.4 (S. 13) die Behandlung der Matrix A und der rechten Seite $\mathbf{b}$ zu entkoppeln. Hierzu merkt man sich die Elementaroperationen, die im Eliminationsteil ausgeführt werden, um sie später auf die rechte Seite anzuwenden.

Der Algorithmus zerfällt in den Zerlegungs- und den Lösungsteil.

Im Zerlegungsteil werden drei Matrizen L, R und P mit folgenden Eigenschaften berechnet:

- P ist eine Permutationsmatrix, d.h. P entsteht durch Zeilenvertauschungen aus der $n \times n$ Einheitsmatrix I_n.
- R ist eine obere Dreiecksmatrix mit von Null verschiedenen Diagonalelementen, d.h. R hat die Form

$$\begin{pmatrix} \bullet & * & * & \dots & * \\ 0 & \bullet & * & \dots & * \\ 0 & 0 & \bullet & \dots & * \\ \vdots & \vdots & & \ddots & \vdots \\ 0 & 0 & \dots & 0 & \bullet \end{pmatrix}.$$

- L ist eine untere Dreiecksmatrix mit Diagonalelementen 1, d.h. L hat die Form

$$\begin{pmatrix} 1 & 0 & 0 & \dots & \dots & 0 \\ * & 1 & 0 & \dots & \dots & 0 \\ * & * & 1 & \dots & \dots & 0 \\ \vdots & \vdots & & \ddots & & \vdots \\ \vdots & \vdots & & & \ddots & 0 \\ * & * & \dots & \dots & * & 1 \end{pmatrix}.$$

- Es ist

$$LR = PA.$$

Im Lösungsteil wird bei gegebener rechter Seite $\mathbf{b}$ das LGS $A\mathbf{x} = \mathbf{b}$ in folgenden Schritten gelöst:

- Berechne $\mathbf{z} = P\mathbf{b}$.
- Löse das LGS $L\mathbf{y} = \mathbf{z}$.
- Löse das LGS $R\mathbf{x} = \mathbf{y}$.

Dabei können der zweite und dritte Schritt wegen der Dreiecksstruktur der Matrizen L und R analog zum Rücklösungsteil von Algorithmus I.4 (S. 13) ausgeführt werden.

Die folgenden beiden Algorithmen realisieren diese Ideen.

ALGORITHMUS I.8. *Bestimmung der* LR-ZERLEGUNG.

(0) *Setze* $\mathbf{p} = (1, 2, \ldots, n)$ *und* $i = 1$.

(1) *Bestimme einen Index* i_0 *mit* $i \le i_0 \le n$ *und* $a_{i_0 i} \neq 0$. *Falls kein solcher Index existiert, ist die Matrix* A *nicht zerlegbar und damit auch nicht invertierbar; beende das Verfahren mit einer entsprechenden Fehlermeldung.*

(2) *Vertausche die* i*-te und die* i_0*-te Zeile von* A *und die* i*-te und die* i_0*-te Komponente von* $\mathbf{p}$.

(3) *Für* $k = i+1, i+2, \ldots, n$ *führe folgende Schritte aus:*

   (a) *Ersetze* a_{ki} *durch* $\dfrac{a_{ki}}{a_{ii}}$.

   (b) *Für* $j = i+1, i+2, \ldots, n$ *ersetze* a_{kj} *durch* $a_{kj} - a_{ij}a_{ki}$.

(4) *Falls* $i < n-1$ *ist, erhöhe* i *um* 1 *und gehe zu Schritt* 1 *zurück. Falls* $i = n-1$ *und* $a_{nn} = 0$ *ist, ist die Matrix* A *nicht zerlegbar und damit auch nicht invertierbar; beende das Verfahren mit einer entsprechenden Fehlermeldung. Ansonsten ist das Verfahren erfolgreich abgeschlossen.*

Nach erfolgreicher Durchführung von Algorithmus I.8 enthält die obere Hälfte der Matrix A einschließlich ihrer Diagonalen die obere Hälfte der Matrix R. Die untere Hälfte von A ohne Diagonale enthält die untere Hälfte von L ohne die Diagonale, die durch 1 ergänzt werden muss. Die Matrix P erhält man aus dem Vektor $\mathbf{p}$ wie folgt: In der i-ten Zeile $(1 \le i \le n)$ von P steht in der p_i-ten Spalte eine 1, alle anderen Elemente der Zeile sind 0.

ALGORITHMUS I.9. *Lösen des LGS* $A\mathbf{x} = \mathbf{b}$ *bei bekannter LR-Zerlegung.*

(1) *Für* $i = 1, \ldots, n$ *setze*

$$z_i = b_{p_i}.$$

(2) *Setze*

$$y_1 = z_1$$

und berechne für $i = 2, \ldots, n$ *sukzessive*

$$y_i = z_i - a_{i1}y_1 - \ldots - a_{ii-1}y_{i-1}.$$

(3) *Setze*

$$x_n = \frac{y_n}{a_{nn}}$$

und berechne für $i = n-1, n-2, \ldots, 1$ *sukzessive*

$$x_i = \frac{1}{a_{ii}}\{y_i - a_{ii+1}x_{i+1} - \ldots - a_{in}x_n\}.$$

Bemerkung I.10. Der Vektor **z** kann auf dem Speicherplatz für **x** und der Vektor **y** auf demjenigen für **b** abgespeichert werden.

Das folgende `Java`-Programm realisiert die Algorithmen I.8 und I.9. Die Methoden `pivot`, `swap`, `innerProduct` und `backsolve` sind identisch mit den in Paragraph I.2 angegebenen Methoden gleichen Namens.

```
// LR decomposition
    public void lrDecomposition() throws LinearAlgebraException {
        lrElimination();
        permutation();
        forSolve();
        backSolve();
    }    // end of lrDecomposition
// elimination part of LR-decomposition
    public void lrElimination() throws LinearAlgebraException {
        perm = new int[dim];
        for( int i = 0; i < dim; i++ )
            perm[i] = i;
        for ( int i = 0; i < dim-1; i++) {
            int jp = pivot(i);     // find pivot, j is pivotelement
            if ( Math.abs( a[jp][i] ) < EPS )    // pivot = 0 ??
                throw new LinearAlgebraException(
                " WARNING: zero pivot in "+i+"-th elimination step");
            if ( jp > i ) {             // swap equations i and j
                swap(perm, i, jp);
                swap(a, i, jp);
            }
            for ( int k = i+1; k < dim; k++ ) {  // elimination
                a[k][i] /= a[i][i];
                substractRows(k, i, a[k][i]);
            }
        }
        if ( Math.abs( a[dim-1][dim-1] ) < EPS )
            throw new LinearAlgebraException(
            " WARNING: zero pivot in "+(dim-1)+"-th elimination step");
    }    // end of lrElimination
// substract factor times row i from row k of a
    public void substractRows(int k, int i, double factor) {
        for( int l = i+1; l < dim; l++ )
            a[k][l] -= a[i][l]*factor;
    }    // end of substractRows
// forward solution part of LR-algorithm (rhs is x, result is b)
    public void forSolve() {
        b[0] = x[0];
        for( int i = 1; i < dim; i++ )
            b[i] = x[i] - innerProduct(a[i], b, 0, i);
    }    // end of forSolve
// permute right-hand side according to vector perm and store on x
    public void permutation() {
        for( int i = 0; i < dim; i++ )
            x[i] = b[ perm[i] ];
    }    // end of permutation
```

Mit den gleichen Überlegungen wie im vorigen Paragraphen ergibt sich für den Aufwand der Algorithmen I.8 und I.9:

> Algorithmus I.8 benötigt $O(n^3)$ arithmetische Operationen. Algorithmus I.9 benötigt pro rechter Seite $O(n^2)$ Operationen.

Das folgende Beispiel verdeutlicht die Vorgehensweise der Algorithmen I.8 und I.9.

BEISPIEL I.11. Für die Matrix aus Beispiel I.6 (S. 15) liefert Algorithmus I.8 die LR-Zerlegung:

$$\mathbf{p} = (1, 2, 3)$$

und

$$\begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 10 \end{matrix} \quad \longrightarrow \quad \begin{matrix} 1 & 2 & 3 \\ 4 & -3 & -6 \\ 7 & -6 & -11 \end{matrix} \quad \longrightarrow \quad \begin{matrix} 1 & 2 & 3 \\ 4 & -3 & -6 \\ 7 & 2 & 1 \end{matrix}$$

Wir erhalten $A = LR$ mit

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ 7 & 2 & 1 \end{pmatrix}, \quad R = \begin{pmatrix} 1 & 2 & 3 \\ 0 & -3 & -6 \\ 0 & 0 & 1 \end{pmatrix}.$$

Für die rechte Seite

$$\mathbf{b} = \begin{pmatrix} 4 \\ 0 \\ 4 \end{pmatrix}$$

ergibt sich $\mathbf{z} = \mathbf{b}$ und

$$\mathbf{y} = \begin{pmatrix} 4 \\ -16 \\ 8 \end{pmatrix}, \quad \mathbf{x} = \begin{pmatrix} \frac{4}{3} \\ -\frac{32}{3} \\ 8 \end{pmatrix}.$$

I.5. Symmetrische positiv definite Matrizen

Eine $n \times n$ Matrix A heißt SYMMETRISCH, wenn für ihre Elemente a_{ij} gilt $a_{ij} = a_{ji}$ für alle $1 \leq i, j \leq n$. Eine symmetrische Matrix A ist POSITIV DEFINIT, kurz S.P.D., wenn für alle vom Nullvektor verschiedenen Vektoren $\mathbf{x} \in \mathbb{R}^n$ gilt $\mathbf{x}^T A \mathbf{x} > 0$.

> Es gelten die folgenden äquivalenten Charakterisierungen für s.p.d. Matrizen:
>
> - A ist s.p.d.
> - Alle Eigenwerte von A sind reell und positiv.

- Die Determinanten der n Hauptmatrizen

$$\begin{aligned} H_1 &= a_{11}, \\ H_2 &= \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}, \\ &\vdots \\ H_k &= \begin{pmatrix} a_{11} & \dots & a_{1k} \\ \vdots & & \\ a_{k1} & \dots & a_{kk} \end{pmatrix}, \\ &\vdots \\ H_n &= A \end{aligned}$$

sind alle positiv.

Beispiel I.12. Die Matrix

$$A = \begin{pmatrix} 5 & -2 & 2 \\ -2 & 6 & -1 \\ 2 & -1 & 4 \end{pmatrix}$$

ist symmetrisch. Die Hauptmatrizen und ihre Determinanten lauten

$$\begin{aligned} H_1 &= 5, & \det H_1 &= 5, \\ H_2 &= \begin{pmatrix} 5 & -2 \\ -2 & 6 \end{pmatrix}, & \det H_2 &= 26, \\ H_3 &= A, & \det H_3 &= 83. \end{aligned}$$

Also ist A positiv definit.

Beispiel I.13. Die Matrix

$$A = \begin{pmatrix} 5 & -3 & 9 \\ -3 & 3 & -3 \\ 9 & -3 & 5 \end{pmatrix}$$

ist symmetrisch. Die Hauptmatrizen und ihre Determinanten sind

$$\begin{aligned} H_1 &= 5, & \det H_1 &= 5, \\ H_2 &= \begin{pmatrix} 5 & -3 \\ -3 & 3 \end{pmatrix}, & \det H_2 &= 6, \\ H_3 &= A, & \det H_3 &= -96. \end{aligned}$$

Das charakteristische Polynom ist

$$\chi_A(\lambda) = -(4+\lambda)(\lambda^2 - 17\lambda + 24).$$

Also sind die Eigenwerte -4, $\frac{17-\sqrt{193}}{2}$, $\frac{17+\sqrt{193}}{2}$; die Matrix ist nicht positiv definit.

I.6. Das Cholesky-Verfahren

Für s.p.d. Matrizen kann man bei der LR-Zerlegung auf die Pivotisierung verzichten. Es gilt nämlich:

> Sei $A \in \mathbb{R}^{n \times n}$ s.p.d. Dann gibt es genau eine untere Dreiecksmatrix L mit Diagonalelementen 1 und genau eine Diagonalmatrix D mit positiven Diagonalelementen, so dass gilt $A = LDL^T$.

Sei nun $1 \leq i \leq j \leq n$. Dann folgt aus $A = LDL^T$ durch Koeffizientenvergleich

$$\begin{aligned} a_{ij} &= \sum_{k,l=1}^{n} \ell_{ik} d_{kl} \ell_{lj}^T \\ &= \sum_{k=1}^{n} \ell_{ik} d_{kk} \ell_{jk} \\ &= \sum_{k=1}^{i-1} \ell_{ik} d_{kk} \ell_{jk} + d_{ii} \ell_{ji} \end{aligned}$$

und somit

$$\begin{aligned} d_{ii} &= a_{ii} - \sum_{k=1}^{i-1} \ell_{ik}^2 d_{kk} \,, \\ \ell_{ji} &= \frac{1}{d_{ii}} \Big[a_{ji} - \sum_{k=1}^{i-1} \ell_{ik} d_{kk} \ell_{jk} \Big] \quad , j > i. \end{aligned}$$

Dies führt auf folgenden Algorithmus zur Berechnung von L und D. Dabei wird von der s.p.d. Matrix A nur der Teil unterhalb der Diagonalen gespeichert und durch D und den Unterdiagonalteil von L überschrieben.

Algorithmus I.14. Cholesky-Zerlegung *einer s.p.d. Matrix.*

(0) *Gegeben: A (Matrix, s.p.d., nur Teil unter der Diagonalen, Diagonale einschließlich)*
Gesucht: L, D (Zerlegung $A = LDL^T$ mit $\ell_{ii} = 1$)

(1) *Für $j = 2, ..., n$ berechne*

$$a_{j1} = \frac{a_{j1}}{a_{11}}.$$

(2) *Für $i = 2, ..., n$ führe folgende Schritte aus*

(a) *Berechne*

$$a_{ii} = a_{ii} - \sum_{k=1}^{i-1} a_{ik}^2 a_{kk}.$$

(b) *Falls $i < n$ ist, berechne für $j = i+1, ..., n$*

$$a_{ji} = \frac{1}{a_{ii}} \Big[a_{ji} - \sum_{k=1}^{i-1} a_{ik} a_{kk} a_{jk} \Big].$$

Der folgende Algorithmus löst das LGS $A\mathbf{x} = \mathbf{b}$ bei bekannter Cholesky-Zerlegung $A = LDL^T$.

Algorithmus I.15. *Lösung eines LGS bei bekannter Cholesky-Zerlegung.*

(0) *Gegeben:*

A *(unterer Teil einer Matrix, enthält D und L)*

$\mathbf{b}$ *(rechte Seite)*

Gesucht:

$\mathbf{x}$ *(Lösung von $A\mathbf{x} = \mathbf{b}$ mit $A = LDL^T$)*

(1) *Setze*

$$z_1 = b_1$$

und berechne für $i = 2, ..., n$

$$z_i = b_i - \sum_{k=1}^{i-1} a_{ik} z_k.$$

(2) *Berechne für $i = 1, ..., n$*

$$y_i = \frac{z_i}{a_{ii}}.$$

(3) *Setze*

$$x_n = y_n$$

und berechne für $i = n-1, n-2, ..., 1$

$$x_i = y_i - \sum_{k=i+1}^{n} a_{ki} x_k.$$

Das folgende Beispiel verdeutlicht die Vorgehensweise von Algorithmus I.14.

Beispiel I.16. Wir betrachten die Matrix

$$A = \begin{pmatrix} 5 & -2 & 2 \\ -2 & 6 & -1 \\ 2 & -1 & 4 \end{pmatrix}$$

aus Beispiel I.12 (S. 22). Für $i = 1$ liefert Algorithmus I.14

$$d_{11} = 5$$

$$\ell_{21} = -\frac{2}{5}$$

$$\ell_{31} = \frac{2}{5}.$$

Für $i = 2$ erhalten wir analog

$$\begin{aligned} d_{22} &= 6 - \left(-\frac{2}{5}\right)^2 5 \\ &= \frac{26}{5} \\ \ell_{32} &= \frac{5}{26}\left[-1 - \left(-\frac{2}{5}\right) 5 \frac{2}{5}\right] \\ &= -\frac{1}{26}. \end{aligned}$$

Für $i = 3$ ergibt sich schließlich

$$\begin{aligned} d_{33} &= 4 - \left(\frac{2}{5}\right)^2 5 - \left(-\frac{1}{26}\right)^2 \frac{26}{5} \\ &= \frac{83}{26}. \end{aligned}$$

Also ist

$$L = \begin{pmatrix} 1 & 0 & 0 \\ -\frac{2}{5} & 1 & 0 \\ \frac{2}{5} & -\frac{1}{26} & 1 \end{pmatrix},$$

$$D = \begin{pmatrix} 5 & 0 & 0 \\ 0 & \frac{26}{5} & 0 \\ 0 & 0 & \frac{83}{26} \end{pmatrix}.$$

Bemerkung I.17. (1) Wie bei Algorithmus I.9 (S. 19) kann $\mathbf{z}$ auf dem Speicherplatz für $\mathbf{x}$ und $\mathbf{y}$ auf demjenigen für $\mathbf{b}$ abgespeichert werden.
(2) Für allgemeine s.p.d. Matrizen haben Algorithmen I.14 und I.15 den gleichen Aufwand wie die Algorithmen I.8 (S. 19) und I.9 (S. 19) und erfordern $O(n^3)$ bzw. $O(n^2)$ Operationen.
(3) Algorithmen I.14 und I.15 sind besonders einfach für s.p.d. Tridiagonalmatrizen, d.h. Matrizen mit

$$a_{ij} = 0 \quad \text{falls} \quad |i - j| > 1.$$

In diesem Fall treten in den Schritten 2 bzw. 1 und 3 nur die Terme mit $k = i - 1$ oder $k = i + 1$ in Erscheinung. Dementsprechend erfordert die Cholesky Zerlegung nur $O(n)$ Operationen.

Das folgende Java-Programm realisiert die Algorithmen I.14 und I.15:

```
// Cholesky decomposition
    public void cholesky() throws LinearAlgebraException {
        choleskyDecomposition();
```

```
        choleskyForSolve();
        choleskyDiagonal();
        choleskyBackSolve();
    }    // end of cholesky
// decomposition part of Cholesky decomposition
    public void choleskyDecomposition() throws LinearAlgebraException {
        if ( Math.abs( a[0][0] ) < EPS )
            throw new LinearAlgebraException(
            " WARNING: zero diagonal element in Cholesky decomposition");
        for ( int j = 1; j < dim; j++ )
            a[j][0] /= a[0][0];
        for ( int i = 1; i < dim; i++ ) {
            for ( int k = 0; k < i; k++ )
                a[i][i] -= a[i][k]*a[i][k]*a[k][k];
            if ( Math.abs( a[i][i] ) < EPS )
                throw new LinearAlgebraException(
                " WARNING: zero diagonal element in Cholesky
                decomposition");
            if ( i < dim-1 ) {
                for ( int j = i+1; j < dim; j++ ) {
                    for ( int k = 0; k < i; k++ )
                        a[j][i] -= a[i][k]*a[k][k]*a[j][k];
                    a[j][i] /= a[i][i];
                }
            }
        }
    }    // end of choleskyDecomposition
// forward solution part of Cholesky decomposition (result is stored on x)
    public void choleskyForSolve() {
        x[0] = b[0];
        for ( int i = 1; i < dim; i++ )
            x[i] = b[i] - innerProduct(a[i], x, 0, i);
    }    // end of choleskyForSolve
// division by diagonal in Cholesky decomposition (result is stored on b)
    public void choleskyDiagonal() {
        for ( int i = 0; i < dim; i++ )
            b[i] = x[i]/a[i][i];
    }    // end of choleskyDiagonal
// backward solution part of Cholesky decomposition
    public void choleskyBackSolve() {
        x[dim-1] = b[dim-1];
        for ( int i = dim-2; i >= 0; i-- ) {
            x[i] = b[i];
            for ( int j = i+1; j < dim; j++ )
                x[i] -= a[j][i]*x[j];
        }
    }    // end of choleskyBackSolve
```

I.7. Große Gleichungssysteme

Bei der Diskretisierung partieller Differentialgleichungen sind sehr grosse LGS zu lösen. Diese sind allerdings dünn besetzt, d.h. pro Zeile sind nur sehr wenige Matrixelemente von Null verschieden.

Im allgemeinen kann man aber bei dem Gaußschen Eliminationsverfahren und seinen Verwandten hieraus keinen Vorteil ziehen. Dies

zeigt die Matrix

$$\begin{pmatrix} 1 & 1 & 1 & 1 & \dots & 1 & 1 \\ 1 & 1 & 0 & 0 & \dots & 0 & 0 \\ 1 & 0 & 1 & 0 & \dots & 0 & 0 \\ \vdots & \vdots & & \ddots & & & \vdots \\ \vdots & \vdots & & & \ddots & & \vdots \\ \vdots & \vdots & & & & \ddots & \vdots \\ 1 & 0 & 0 & 0 & \dots & 0 & 1 \end{pmatrix}.$$

Nach einem Eliminationsschritt hat sie nämlich die Form

$$\begin{pmatrix} 1 & 1 & 1 & 1 & \dots & 1 & 1 \\ 0 & 0 & -1 & -1 & \dots & -1 & -1 \\ 0 & -1 & 0 & -1 & \dots & -1 & -1 \\ \vdots & \vdots & & \ddots & & & \vdots \\ \vdots & \vdots & & & \ddots & & \vdots \\ \vdots & \vdots & & & & \ddots & \vdots \\ 0 & -1 & -1 & -1 & \dots & -1 & 0 \end{pmatrix}$$

und ist voll besetzt. Diesen Effekt nennt man AUFFÜLLEN (engl. FILL-IN).

Glücklicherweise ist die Situation bei der Diskretisierung partieller Differentialgleichungen günstiger. Die dort auftretenden Matrizen sind nämlich BANDMATRIZEN, d.h. es gibt eine Zahl $b \ll n$, genannt BANDBREITE, so dass $a_{ij} = 0$ ist für alle Indexpaare i, j mit $|i - j| > b$. Obige Matrix ist keine Bandmatrix. Die Bandstruktur bleibt beim Gaußschen Eliminationsverfahren und seinen Verwandten erhalten. Man muss daher nur b Speicherplätze pro Zeile und Spalte zur Verfügung stellen. Entsprechend benötigt der Eliminationsteil nur $O(nb^2)$ Operationen und der Rücklösungsteil $O(nb)$ Operationen. Bei partiellen Differentialgleichungen in zwei bzw. drei Raumdimensionen ist typischerweise $b = n^{1/2}$ bzw. $b = n^{2/3}$.

Zur Illustration geben wir in Tabelle I.2 für eine Differenzendiskretisierung der POISSONGLEICHUNG $-\Delta u = f$ auf einem äquidistanten Gitter folgende Größen an:

- d die Raumdimension,
- h die Gitterweite in jeder Raumdimension,
- N_h die Zahl der Unbekannten,
- e_h die Zahl der von Null verschiedenen Matrixelemente pro Zeile,
- b_h die Bandbreite,
- s_h die Zahl der benötigten Speicherplätze,
- z_h die Zahl der benötigten arithmetischen Operationen.

TABELLE I.2. Speicherbedarf und Rechenaufwand für die Lösung einer Differenzendiskretisierung der Poissongleichung mit dem Cholesky-Verfahren

n	h	N_h	e_h	b_h	s_h	z_h
2	$\frac{1}{16}$	225	$1.1 \cdot 10^3$	15	$3.3 \cdot 10^3$	$7.6 \cdot 10^5$
	$\frac{1}{32}$	961	$4.8 \cdot 10^3$	31	$2.9 \cdot 10^4$	$2.8 \cdot 10^7$
	$\frac{1}{64}$	$3.9 \cdot 10^3$	$2.0 \cdot 10^4$	63	$2.5 \cdot 10^5$	$9.9 \cdot 10^8$
	$\frac{1}{128}$	$1.6 \cdot 10^4$	$8.0 \cdot 10^4$	127	$2.0 \cdot 10^6$	$3.3 \cdot 10^{10}$
3	$\frac{1}{16}$	$3.3 \cdot 10^3$	$2.4 \cdot 10^4$	225	$7.6 \cdot 10^5$	$1.7 \cdot 10^8$
	$\frac{1}{32}$	$3.0 \cdot 10^4$	$2.1 \cdot 10^5$	961	$2.8 \cdot 10^7$	$2.8 \cdot 10^{10}$
	$\frac{1}{64}$	$2.5 \cdot 10^5$	$1.8 \cdot 10^6$	$3.9 \cdot 10^3$	$9.9 \cdot 10^8$	$3.9 \cdot 10^{12}$
	$\frac{1}{128}$	$2.0 \cdot 10^6$	$1.4 \cdot 10^7$	$1.6 \cdot 10^4$	$3.3 \cdot 10^{10}$	$5.3 \cdot 10^{14}$

Tabelle I.2 belegt eindrücklich, dass das Gaußsche Eliminationsverfahren und seine Verwandten zur Lösung derartiger großer LGS ungeeignet sind. Ein Giga-Flop-Rechner würde z.B. zur Lösung der Poissongleichung auf dem Einheitswürfel mit der Schrittweite $h = \frac{1}{128}$ mehr als 6 Tage und einen Speicherplatz von 33 Gigabyte erfordern. Man beachte, dass in diesem Beispiel zur Speicherung eines Vektors „nur“ 2 Megabyte und zur Speicherung der von Null verschiedenen Matrixelemente „nur“ 14 Megabyte erforderlich sind.

Für derartige große LGS muss man stattdessen auf moderne iterative Lösungsverfahren wie VORKONDITIONIERTE KONJUGIERTE GRADIENTEN (PCG-) VERFAHREN oder MEHRGITTERVERFAHREN zurückgreifen. Wir können hier nicht näher auf diese Verfahren eingehen, geben aber in Tabelle I.3 den Aufwand an, den diese Verfahren für die Lösung einer Differenzendiskretisierung der Poissongleichung benötigen. Ein Vergleich mit der letzten Spalte von Tabelle I.2 belegt die Überlegenheit der iterativen Verfahren.

I.8. Störungsrechnung

Aufgrund von Rundungsfehlern liefern die Verfahren der vorigen Abschnitte nicht die exakte Lösung $\mathbf{x}^*$ eines LGS $A\mathbf{x} = \mathbf{b}$ sondern nur eine Näherungslösung $\hat{\mathbf{x}}$. Wie groß ist die Euklidische Norm $\|\mathbf{x}^* - \hat{\mathbf{x}}\|_2$ des Fehlers?

Da wir $\mathbf{x}^*$ nicht kennen, können wir diese Größe nicht berechnen. Was wir berechnen können, ist das RESIDUUM

$$\mathbf{r} = A\hat{\mathbf{x}} - \mathbf{b}.$$

TABELLE I.3. Aufwand des PCG-Verfahrens mit SSOR-Vorkonditionierung und des Mehrgitterverfahrens zur Lösung einer Differenzendiskretisierung der Poissongleichung

n	h	PCG-SSOR	Mehrgitter
2	$\frac{1}{16}$	$16'200$	$11'700$
	$\frac{1}{32}$	$86'490$	$48'972$
	$\frac{1}{64}$	$500'094$	$206'988$
	$\frac{1}{128}$	$3'193'542$	$838'708$
3	$\frac{1}{16}$	$310'500$	$175'500$
	$\frac{1}{32}$	$3'425'965$	$1'549'132$
	$\frac{1}{64}$	$4.0 \cdot 10^7$	$1.3 \cdot 10^7$
	$\frac{1}{128}$	$5.2 \cdot 10^8$	$1.1 \cdot 10^8$

Wir wollen daher $\|\mathbf{x}^* - \hat{\mathbf{x}}\|_2$ durch $\|\mathbf{r}\|_2$ abschätzen. Wie zuverlässig ist eine solche Abschätzung?

Um dies zu beantworten, benötigen wir den Begriff der Matrixnorm und der Kondition. Die zur Euklidischen Norm gehörige MATRIXNORM ist definiert durch

$$\text{(I.2)} \qquad |||A|||_2 = \max_{\mathbf{x} \in \mathbb{R}^n \setminus \{0\}} \frac{\|A\mathbf{x}\|_2}{\|\mathbf{x}\|_2}.$$

Die zur Euklischen Norm gehörige KONDITION einer Matrix ist definiert durch

$$\text{(I.3)} \qquad \kappa_2(A) = \begin{cases} \infty & \text{falls } A \text{ nicht invertierbar,} \\ |||A|||_2 |||A^{-1}|||_2 & \text{falls } A \text{ invertierbar.} \end{cases}$$

Es gilt folgende Charakterisierung:

Für jede invertierbare Matrix A ist

$$\kappa_2(A) = \{\lambda_{\max}(A^T A)/\lambda_{\min}(A^T A)\}^{\frac{1}{2}}.$$

Für jede invertierbare, symmetrische Matrix A ist

$$\kappa_2(A) = \lambda_{\max}(A)/\lambda_{\min}(A).$$

Dabei bezeichnet $\lambda_{\max}(B)$ bzw. $\lambda_{\min}(B)$ den betragsmäßig größten bzw. kleinsten Eigenwert einer Matrix B.

Man beachte, dass für jede Matrix A gilt $\kappa_2(A) \geq 1$.

Aus der Definition der Matrixnorm folgt

$$\begin{aligned} \|\mathbf{r}\|_2 &= \|A\hat{\mathbf{x}} - \mathbf{b}\|_2 \\ &= \|A(\hat{\mathbf{x}} - \mathbf{x}^*)\|_2 \\ &\leq |||A|||_2 \|\hat{\mathbf{x}} - \mathbf{x}^*\|_2 \end{aligned}$$

und

$$\begin{aligned} \|\hat{\mathbf{x}} - \mathbf{x}^*\|_2 &= \|A^{-1}A(\mathbf{x}^* - \hat{\mathbf{x}})\|_2 \\ &= \|A^{-1}\mathbf{r}\|_2 \\ &\leq |||A^{-1}|||_2 \|\mathbf{r}\|_2. \end{aligned}$$

Analog ergibt sich

$$\begin{aligned} \|\mathbf{b}\|_2 &= \|A\mathbf{x}^*\|_2 \\ &\leq |||A|||_2 \|\mathbf{x}^*\|_2 \end{aligned}$$

und

$$\begin{aligned} \|\mathbf{x}^*\|_2 &= \|A^{-1}\mathbf{b}\|_2 \\ &\leq |||A^{-1}|||_2 \|\mathbf{b}\|_2. \end{aligned}$$

Damit ergibt sich insgesamt

$$\frac{1}{\kappa_2(A)} \frac{\|\mathbf{r}\|_2}{\|\mathbf{b}\|_2} \leq \frac{\|\mathbf{x}^* - \hat{\mathbf{x}}\|_2}{\|\mathbf{x}^*\|_2} \leq \kappa_2(A) \frac{\|\mathbf{r}\|_2}{\|\mathbf{b}\|_2}.$$

Der relative Fehler $\|\mathbf{x}^* - \hat{\mathbf{x}}\|_2 / \|\mathbf{x}^*\|_2$ ist also in beide Richtungen abschätzbar durch die relative Größe des Residuums $\|\mathbf{r}\|_2 / \|\mathbf{b}\|_2$. Die Kondition $\kappa_2(A)$ der Matrix gibt dabei den Verstärkungsfaktor an. Je größer die Kondition ist, umso ungenauer ist diese Abschätzung. Bei einer großen Kondition $\kappa_2(A)$ können kleine Störungen der rechten Seite $\mathbf{b}$ zu großen Änderungen der Lösung $\mathbf{x}^*$ führen.

Beispiel I.18. Für die Matrix

$$A = \begin{pmatrix} 5000 & 4999 \\ 4999 & 5000 \end{pmatrix}$$

erhalten wir

$$\begin{aligned} \lambda_{\max}(A) &= 9999 \\ \lambda_{\min}(A) &= 1 \end{aligned}$$

$$\kappa_2(A) = 9999.$$

Für die rechte Seite

$$\mathbf{b} = \begin{pmatrix} 9999 \\ 9999 \end{pmatrix}$$

lautet die exakte Lösung

$$\mathbf{x}^* = \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

Für die Näherungslösung

$$\hat{\mathbf{x}} = \begin{pmatrix} 2 \\ 0 \end{pmatrix}$$

erhalten wir das Residuum

$$\mathbf{r} = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$$

und

$$\begin{aligned} \frac{\|\mathbf{r}\|_2}{\|\mathbf{b}\|_2} &= \frac{1}{9999} \\ &\approx 10^{-4} \\ \frac{\|\mathbf{x}^* - \hat{\mathbf{x}}\|_2}{\|\mathbf{x}^*\|_2} &= \frac{1}{\sqrt{2}} \\ &\approx 0.7071. \end{aligned}$$

BEMERKUNG I.19. Man kann die Matrixnorm und Kondition für beliebige Vektornormen definieren. Dazu muss man auf der rechten Seite von (I.2) in Zähler und Nenner die Euklidische Norm $\|\cdot\|_2$ durch die aktuelle Vektornorm $\|\cdot\|$ ersetzen. In (I.3) muss man dann $|||\cdot|||_2$ durch die so berechnete neue Matrixnorm ersetzen. Für die Maximumnorm

$$\|\mathbf{x}\|_\infty = \max_{1 \le i \le n} |x_i|$$

als Vektornorm erhält man z.B. als Matrixnorm die ZEILENSUMMENNORM

$$\boxed{|||A|||_\infty = \max_{1 \le i \le n} \sum_{j=1}^{n} |a_{ij}|.}$$

Für die ℓ_1-Norm

$$\|\mathbf{x}\|_1 = \sum_{i=1}^{n} |x_i|$$

als Vektornorm erhält man z.B. als Matrixnorm die SPALTENSUMMENNORM

$$|||A|||_1 = \max_{1 \le j \le n} \sum_{i=1}^{n} |a_{ij}|.$$

KAPITEL II

Nicht lineare Gleichungssysteme

II.1. Das Bisektionsverfahren

Gesucht ist eine Nullstelle x^* einer stetigen Funktion f einer reellen Variablen, die auf einem beschränkten oder unbeschränkten Intervall I definiert ist. Wir nehmen an, dass wir zwei Punkte x_0 und y_0 mit $x_0 < y_0$ kennen, in denen f Funktionswerte mit unterschiedlichem Vorzeichen hat, d.h. $f(x_0)f(y_0) < 0$. Dann folgt aus dem Zwischenwertsatz (Mathematik für Maschinenbauer, Bauingenieure und Umwelttechniker I, §III.3.3, Satz 4), dass f eine Nullstelle in dem Intervall (x_0, y_0) hat. Wir berechnen nun den Mittelpunkt $\hat{x} = \frac{1}{2}(x_0 + y_0)$ dieses Intervalles. Falls $f(\hat{x}) = 0$ ist, sind wir fertig. Andernfalls gilt entweder $f(x_0)f(\hat{x}) < 0$ oder $f(\hat{x})f(y_0) < 0$. Im ersten Fall setzen wir $x_1 = x_0$ und $y_1 = \hat{x}$, im zweiten Fall $x_1 = \hat{x}$ und $y_1 = y_0$. Das neue Intervall (x_1, y_1) hat die halbe Länge des alten Intervalles (x_0, y_0) und enthält wieder eine Nullstelle von f. Wenn wir diesen Prozess wiederholen, erhalten wir eine Folge immer kleiner werdender Intervalle, die eine Nullstelle von f enthalten.

Diese Überlegungen führen auf den folgenden Algorithmus:

Algorithmus II.1. Bisektionsverfahren *zur Bestimmung einer Nullstelle einer Funktion einer Veränderlichen.*

(0) *Gegeben eine stetige Funktion f einer reellen Veränderlichen und zwei Punkte x_0 und y_0 aus dem Definitionsbereich von f mit $x_0 < y_0$ und $f(x_0)f(y_0) < 0$ und eine Toleranz $\varepsilon > 0$ sowie eine obere Schranke N für die Zahl der Iterationen.*

(1) *Setze $i = 0$.*

(2) *Falls $y_i - x_i < \varepsilon$ ist, gebe $\hat{x} = \frac{1}{2}(x_i + y_i)$ als Näherung für die gesuchte Nullstelle aus und beende das Verfahren. Andernfalls gehe zu Schritt 3.*

(3) *Berechne $\hat{x} = \frac{1}{2}(x_i + y_i)$. Falls $|f(\hat{x})| < \varepsilon$ ist, gebe $\hat{x}$ als Näherung für die gesuchte Nullstelle aus und beende das Verfahren. Andernfalls gehe zu Schritt 4.*

(4) *Setze*

$$\begin{aligned} x_{i+1} &= x_i, & y_{i+1} &= \hat{x}, & &\text{falls } f(x_i)f(\hat{x}) < 0 \\ x_{i+1} &= \hat{x}, & y_{i+1} &= y_i, & &\text{falls } f(x_i)f(\hat{x}) > 0 \end{aligned}$$

erhöhe i um 1. Falls $i = N$ ist, beende das Verfahren mit einer entsprechenden Meldung, sonst gehe zu Schritt 2 zurück.

Wie man sich leicht überlegt, gilt nach i Schritten des Bisektionsverfahrens $y_i - x_i \leq 2^{-i}(y_0 - x_0)$, so dass das Verfahren spätestens nach $\frac{\ln(y_0 - x_0) - \ln(\varepsilon)}{\ln(2)} + 1$ Schritten abbricht.

Beispiel II.2. Zur Berechnung von $\sqrt{2}$ wenden wir Algorithmus II.1 auf die Funktion $f(x) = x^2 - 2$ mit den Startwerten $x_0 = 1$ und $y_0 = 2$ an. Das Ergebnis der ersten 10 Iterationen ist in Tabelle II.1 wiedergegeben.

Tabelle II.1. Bisektionsverfahren zur Berechnung von $\sqrt{2}$

i	x_i	y_i
0	1	2
1	1	1.5
2	1.25	1.5
3	1.375	1.5
4	1.375	1.4375
5	1.40625	1.4375
6	1.40625	1.421875
7	1.4140625	1.421875
8	1.4140625	1.41796875
9	1.4140625	1.416015625

II.2. Das Sekantenverfahren

Wieder gehen wir davon aus, dass wir zwei Näherungen für die gesuchte Nullstelle von f geraten haben. Allerdings nennen wir sie diesmal x_0 und x_1 und verzichten auf die Forderung, dass die Funtionswerte unterschiedliches Vorzeichen haben. Wir ersetzen nun den Graphen von f durch die Gerade (Sekante) durch die Punkte $(x_0, f(x_0))$ und $(x_1, f(x_1))$ und hoffen, dass der Schnittpunkt x_2 dieser Geraden mit der x-Achse (sofern er exisitiert!) eine bessere Näherung für die gesuchte Nullstelle ist (vgl. Abbildung II.1).

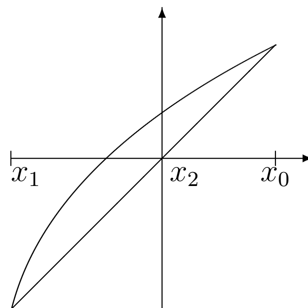


Abbildung II.1. Sekantenverfahren

Die Gleichung der Sekante lautet

$$y = f(x_0) + \frac{x - x_0}{x_1 - x_0}(f(x_1) - f(x_0)).$$

Daher muss der Punkt x_2 die Gleichung

$$0 = f(x_0) + \frac{x_2 - x_0}{x_1 - x_0}(f(x_1) - f(x_0))$$

erfüllen. Diese Gleichung kann genau dann nach x_2 aufgelöst werden, wenn $f(x_1) \neq f(x_0)$ ist. In diesem Fall erhalten wir

$$\begin{aligned} x_2 &= x_0 - \frac{f(x_0)(x_1 - x_0)}{f(x_1) - f(x_0)} \\ &= x_1 - \frac{f(x_1)(x_1 - x_0)}{f(x_1) - f(x_0)}. \end{aligned}$$

Wenn wir diesen Prozess iterieren, erhalten wir den folgenden Algorithmus:

Algorithmus II.3. Sekantenverfahren *zur Bestimmung einer Nullstelle einer Funktion einer Veränderlichen.*

(0) *Gegeben eine stetige Funktion f einer reellen Veränderlichen und zwei Punkte x_0 und x_1 aus dem Definitionsbereich von f und eine Toleranz $\varepsilon > 0$ sowie eine obere Schranke N für die Zahl der Iterationen.*

(1) *Setze $i = 1$.*

(2) *Falls $|x_i - x_{i-1}| < \varepsilon$ ist, gebe $\hat{x} = \frac{1}{2}(x_i + x_{i-1})$ als Näherung für die gesuchte Nullstelle aus und beende das Verfahren. Andernfalls gehe zu Schritt 3.*

(3) *Falls $|f(x_i) - f(x_{i-1})| < \varepsilon$ ist, liegt eine waagerechte Tangente vor. Beende das Verfahren mit einer entsprechenden Fehlermeldung. Andernfalls gehe zu Schritt 4.*

(4) *Berechne*

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

Falls $|f(x_{i+1})| < \varepsilon$ ist, gebe x_{i+1} als Näherung für die Nullstelle aus und beende das Vefahren. Andernfalls erhöhe i um 1. Falls $i = N$ ist, beende das Verfahren mit einer entsprechenden Meldung, sonst gehe zu Schritt 2 zurück.

Das folgende `Java`-Programm realisiert das Sekantenverfahren. Dabei ist `fct` eine abstrakte Klasse, die für eine stetige Funktion den Funktionswert `f` liefert.

```
// secant rule
    public void secantRule() throws ZeroException {
        int it = 0;
        fx = fct.f(x);
        fy = fct.f(y);
```

```
        dfx = (fy - fx)/(y - x);
        it++;
        while( it < maxit && Math.abs(fy) > tol ) {
            x = y;
            if( Math.abs(dfx) <= EPS )
                throw new ZeroException(
                " WARNING: vanishing derivative");
            y -= fy/dfx;
            fx = fy;
            fy = fct.f(y);
            dfx = (fy - fx)/(y - x);
            it++;
        }
    }   // end of secantRule
```

Beispiel II.4. Zur Berechnung von $\sqrt{2}$ wenden wir Algorithmus II.3 auf die Funktion $f(x) = x^2 - 2$ mit den Startwerten $x_0 = 1$ und $x_1 = 2$ an. Die Ergebnisse sind in Tabelle II.2 wiedergegeben.

Tabelle II.2. Sekantenverfahren zur Berechnung von $\sqrt{2}$

i	x_i	$f(x_i)$
0	1	-1
1	2	2
2	$1.\overline{3}$	$-0.\overline{2}$
3	1.4	-0.4
4	1.41463414	0.00118976
5	1.41421143	-0.00000601

II.3. Das Newtonverfahren für Funktionen einer Variablen

Das Newtonverfahren und seine Varianten sind die verbreitesten und effizientesten Verfahren zur Nullstellenberechnung bei differenzierbaren Funktionen.

Zur Beschreibung des Verfahrens nehmen wir an, dass wir eine Näherung x_0 für die gesuchte Nullstelle $\overline{x}$ der Funktion f geraten haben. Dann wird f in der Nähe von x_0 durch die Tangente an die Kurve $y = f(x)$ im Punkt $(x_0, f(x_0))$ approximiert. Der Schnittpunkt x_1 dieser Tangente mit der x-Achse sollte daher eine passable Näherung für die gesuchte Nullstelle sein. (Dies gilt natürlich nur, sofern die Tangente nicht waagerecht ist!) Die Gleichung der Tangente ist

$$y = f(x_0) + f'(x_0)(x - x_0).$$

Also lautet die Bestimmungsgleichung für x_1:

$$0 = f(x_0) + f'(x_0)(x_1 - x_0) \quad \Longrightarrow \quad x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}.$$

Diese Vorgehensweise wird in Abbildung II.2 verdeutlicht.

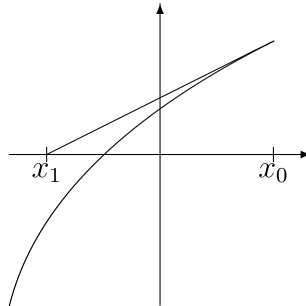


ABBILDUNG II.2. Geometrische Interpretation des Newtonverfahrens

Wenn wir diese Vorschrift iterieren erhalten wir den folgenden Algorithmus:

ALGORITHMUS II.5. NEWTONVERFAHREN *zur Bestimmung einer Nullstelle einer Funktion einer Veränderlichen.*

(0) *Gegeben eine differenzierbare Funktion f einer reellen Veränderlichen, ein Punkt x_0 aus dem Definitionsbereich von f und eine Toleranz $\varepsilon > 0$ sowie eine obere Schranke N für die Zahl der Iterationen.*
(1) *Setze $i = 0$.*
(2) *Falls $|f(x_i)| < \varepsilon$ ist, gebe x_i als Näherung für die Nullstelle aus und beende das Vefahren. Andernfalls gehe zu Schritt 3.*
(3) *Falls $|f'(x_i)| < \varepsilon$ ist, liegt eine waagerechte Tangente vor. Beende das Verfahren mit einer entsprechenden Fehlermeldung. Andernfalls gehe zu Schritt 4.*
(4) *Berechne*

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

Erhöhe i um 1. Falls $i = N$ ist, beende das Verfahren mit einer entsprechenden Meldung, sonst gehe zu Schritt 2 zurück.

Das folgende `Java`-Programm realisiert das Newtonverfahren. Dabei ist `fct` jetzt eine abstrakte Klasse, die für eine stetig differenzierbare Funktion den Funktionswert `f` und den Wert der Ableitung `df` liefert.

```
// Newton algorithm
    public void newton() throws ZeroException {
        int it = 0;
        fx = fct.f(x);
        dfx = fct.df(x);
        while( it < maxit && Math.abs(fx) > tol ) {
            if( Math.abs(dfx) <= EPS )
                throw new ZeroException(
                " WARNING: vanishing derivative");
            x -= fx/dfx;
            fx = fct.f(x);
            dfx = fct.df(x);
        }
```

```
}    // end of newton
```

Beispiel II.6. Zur Berechnung von $\sqrt{2}$ wenden wir Algorithmus II.5 auf die Funktion $f(x) = x^2 - 2$ mit dem Startwert $x_0 = 2$ an. Die Ergebnisse sind in Tabelle II.3 wiedergegeben.

Tabelle II.3. Newtonverfahren zur Berechnung von $\sqrt{2}$

i	x_i	$f(x_i)$
0	2	2
1	1.5	0.25
2	$1.41\overline{6}$	$0.0069\overline{4}$
3	1.41421568	0.00000601

II.4. Eigenschaften des Newtonverfahrens

Das Newtonverfahren bricht natürlich zusammen, wenn die Tangente waagerecht ist, d.h. wenn $f'(x_i) = 0$ wird für ein i. Bei ungünstiger Wahl des Startwertes kann es auch divergieren, d.h. $|x_i| \to \infty$ für $i \to \infty$, oder in einen Kessel geraten, d.h. es gibt ein $m \in \mathbb{N}^*$ mit $x_i = x_{i+m}$ für alle i.

Beispiel II.7. Betrachte das Polynom $f(x) = x^4 - 3x^2 - 2$. Dann lautet das Newtonverfahren mit $x_0 = 1$,

$$\begin{aligned} x_1 &= 1 - \frac{-4}{-2} \\ &= -1, \\ x_2 &= -1 - \frac{-4}{2} \\ &= 1 \\ &= x_0. \end{aligned}$$

Das Newtonverfahren gerät also in einen Kessel.

Beispiel II.8. Das Newtonverfahren angewandt auf die Funktion $f(x) = \arctan(x)$ liefert für den Startwert $x_0 = 2$ die in Tabelle II.4 angegebenen Ergebnisse. Offensichtlich divergiert das Verfahren.

Ist aber der Startwert hinreichend nahe bei der Nullstelle, konvergiert das Newtonverfahren quadratisch, d.h. es gibt eine Konstante c mit

$$\boxed{|x_{i+1} - \overline{x}| \le c|x_i - \overline{x}|^2}$$

für alle hinreichend großen i.

TABELLE II.4. Newtonverfahren für $f(x) = \arctan(x)$

i	x_i	$f(x_i)$
0	2.0	1.107148
1	-3.535743	-1.295169
2	13.950959	1.499239
3	-279.344066	-1.567216
4	122016.998918	1.570788

Numerisch äußerst sich die quadratische Konvergenz in einer Verdoppelung der korrekten Nachkommastellen mit jedem Iterationsschritt.

BEISPIEL II.9. Wir betrachten das Polynom

$$p(x) = x^3 + x^2 + 2x + 1.$$

Wegen $p(-1) = -1$ und $p(0) = 1$ hat es eine Nullstelle im Intervall $(-1, 0)$. Für die Ableitung gilt in $(-1, 0)$

$$p'(x) = 3x^2 + 2x + 2 > 3x^2 > 0.$$

Also ist das Newtonverfahren durchführbar. Für den Startwert $x_0 = -0.5$ erhalten wir die in Tabelle II.5 angegebenen Ergebnisse.

TABELLE II.5. Newtonverfahren für $f(x) = x^3 + x^2 + 2x + 1$

i	x_i
0	-0.500000
1	-0.571429
2	-0.569841
3	-0.569840

BEISPIEL II.10. Seien $k \geq 2$ und $a > 0$. Dann ist $\sqrt[k]{a}$ die Nullstelle des Polynoms $x^k - a$. Die Iterationsvorschrift des Newtonverfahrens lautet in diesem Fall

$$\begin{aligned} x_{i+1} &= x_i - \frac{x_i^k - a}{k x_i^{k-1}} \\ &= \frac{1}{k}\left[(k-1)x_i + \frac{a}{x_i^{k-1}}\right]. \end{aligned}$$

Dieses Verfahren, das die Berechnung von Wurzeln mit den Grundrechenarten erlaubt, war schon den alten Griechen bekannt und wird als VERFAHREN VON HERON bezeichnet.

Beispiel II.11. Betrachte die Funktion $\frac{1}{x} - a$ mit $a > 0$. Die Iterationsvorschrift des Newtonverfahrens lautet in diesem Fall

$$\begin{aligned} x_{i+1} &= x_i - \frac{\frac{1}{x_i} - a}{-\frac{1}{x_i^2}} \\ &= 2x_i - ax_i^2 \\ &= x_i(2 - ax_i). \end{aligned}$$

Dieses Verfahren zur divisionsfreien Berechnung von Reziproken war schon den alten Ägyptern bekannt.

II.5. Das Newtonverfahren für Funktionen mehrerer Variabler

Das Newtonverfahren mit seinen Modifikationen ist das wichtigste Verfahren zur numerischen Lösung nicht linearer Gleichungssysteme mit n Gleichungen und n Unbekannten. Ein solches Gleichungssystem kann stets in die Form gebracht werden

$$\mathbf{f}(\mathbf{x}) = 0 \tag{II.1}$$

mit einer Funktion $\mathbf{f} : \mathbb{R}^n \supset D \to \mathbb{R}^n$. Die Idee des Verfahrens ist die gleiche wie bei Gleichungen in einer Unbekannten:
Wir nehmen an, dass wir eine Näherungslösung $\mathbf{x}_0$ für (II.1) „geraten" haben. Falls $\mathbf{f}$ differenzierbar ist, wird $\mathbf{f}$ in der Nähe von $\mathbf{x}_0$ durch die lineare Funktion $\mathbf{x} \mapsto \mathbf{f}(\mathbf{x}_0) + D\mathbf{f}(\mathbf{x}_0)(\mathbf{x} - \mathbf{x}_0)$ approximiert. Wir ersetzen $\mathbf{f}$ in (II.1) durch diese Approximation und erhalten das lineare Gleichungssystem

$$\mathbf{f}(\mathbf{x}_0) + D\mathbf{f}(\mathbf{x}_0)(\mathbf{x} - \mathbf{x}_0) = 0$$

für den unbekannten Vektor $\mathbf{x}$. Es ist genau dann lösbar, wenn die Jacobi-Matrix $D\mathbf{f}(\mathbf{x}_0)$ invertierbar ist. In diesem Fall lautet die Lösung des linearen Gleichungssystems

$$\mathbf{x}_1 = \mathbf{x}_0 - D\mathbf{f}(\mathbf{x}_0)^{-1}\mathbf{f}(\mathbf{x}_0).$$

Wir nehmen $\mathbf{x}_1$ als neue, (hoffentlich) bessere Näherung für die Lösung von (II.1) und wiederholen das Verfahren mit $\mathbf{x}_1$ an Stelle von $\mathbf{x}_0$.

> Achtung: Selbstverständlich wird bei der Berechnung von $\mathbf{x}_1$ die Matrix $D\mathbf{f}(\mathbf{x}_0)$ nicht invertiert. Stattdessen wird das lineare Gleichungssystem
>
> $$D\mathbf{f}(\mathbf{x}_0)(\mathbf{x}_1 - \mathbf{x}_0) = -\mathbf{f}(\mathbf{x}_0)$$
>
> mit dem Gaußschen Eliminationsverfahren oder der LR-Zerlegung gelöst.

Zusammenfassend erhalten wir folgenden Algorithmus, wobei $\|\mathbf{x}\|_2$ die Euklidische Norm eines Vektors $\mathbf{x}$ bezeichnet:

ALGORITHMUS II.12. NEWTONVERFAHREN *zur Lösung eines nicht linearen Gleichungssystems.*

(0) *Gegeben eine differenzierbare Funktion* $\mathbf{f}$ *mehrerer reeller Variabler, ein Vektor* $\mathbf{x}_0$ *aus dem Definitionsbereich von* $\mathbf{f}$ *und eine Toleranz* $\varepsilon > 0$ *sowie eine obere Schranke* N *für die Zahl der Iterationen.*
(1) *Setze* $i = 0$.
(2) *Falls* $\|\mathbf{f}(\mathbf{x}_i)\|_2 < \varepsilon$ *ist, gebe den Vektor* $\mathbf{x}_i$ *als Näherung für die Nullstelle aus und beende das Vefahren. Andernfalls gehe zu Schritt 3.*
(3) *Berechne mit dem Gaußschen Eliminationsverfahren oder mit der LR-Zerlegung die Lösung* $\mathbf{z}$ *des linearen Gleichungssystems*

$$D\mathbf{f}(\mathbf{x}_i)\mathbf{z} = -\mathbf{f}(\mathbf{x}_i).$$

Falls das Lösungsverfahren für dieses LGS mit der Fehlermeldung „das LGS hat keine eindeutige Lösung“ abbricht, beende das Verfahren mit einer entsprechenden Fehlermeldung. Andernfalls gehe zu Schritt 4.
(4) *Berechne*

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \mathbf{z}.$$

Erhöhe i um 1. *Falls* $i = N$ *ist, beende das Verfahren mit einer entsprechenden Meldung, sonst gehe zu Schritt 2 zurück.*

Wie in einer Dimension, $n = 1$, konvergiert das Newtonverfahren nicht für jeden Startwert. Falls aber $\mathbf{x}_0$ hinreichend nahe bei einer Lösung $\mathbf{x}^*$ des Gleichungssystems liegt, $\mathbf{f}$ zweimal stetig differenzierbar ist und $D\mathbf{f}(\mathbf{x}^*)$ invertierbar ist, kann man zeigen, dass das Newtonverfahren QUADRATISCH KONVERGIERT. D.h., es gibt eine Konstante $c > 0$ mit

$$\boxed{\|\mathbf{x}^* - \mathbf{x}_{i+1}\|_2 \le c\|\mathbf{x}^* - \mathbf{x}_i\|_2^2}$$

für alle i.

BEISPIEL II.13. Zur Darstellung der Zahnflanken von Stirnradgetrieben verwendet man die Kreisevolvente γ mit der Parameterdarstellung

$$\mathbf{x}(t) = \begin{pmatrix} r\sin t - rt\cos t \\ r\cos t + rt\sin t - r \end{pmatrix}.$$

Soll γ durch die Punkte $(0, 0)$ und (a, b) gehen, führt dies auf die beiden folgenden nicht linearen Gleichungen für r und t:

$$f_1(r, t) = r\sin t - rt\cos t - a = 0$$
$$f_2(r, t) = r\cos t + rt\sin t - r - b = 0.$$

Die Jacobi-Matrix von $\mathbf{f}$ im Punkt (r_0, t_0) lautet

$$\begin{pmatrix} \sin t_0 - t_0 \cos t_0 & r_0 t_0 \sin t_0 \\ \cos t_0 + t_0 \sin t_0 - 1 & r_0 t_0 \cos t_0 \end{pmatrix}.$$

Die Determinante der Matrix ist $r_0 t_0 (\sin t_0 - t_0)$. Sie ist ungleich Null sofern $r_0 t_0 \neq 0$ ist.
Für $(a, b) = (1, 1)$ und die Startwerte $(r_0, t_0) = (2, 1.2)$ liefert das Newtonverfahren dann z.B. folgende Werte:

$$\begin{aligned} r_0 &= 2 & t_0 &= 1.2 \\ r_1 &= 2.12598 & t_1 &= 1.17449 \\ r_2 &= 2.12891 & t_2 &= 1.17504 \\ r_3 &= 2.12891 & t_3 &= 1.17504. \end{aligned}$$

II.6. Dämpfung

Bei ungünstiger Wahl des Startwertes $\mathbf{x}_0$ divergiert das Newtonverfahren, weil es zu große Schritte macht (vgl. Beispiel II.8 (S. 38)). Daher versucht man, die Newton-Schritte zu dämpfen. Zur Beschreibung der Idee betrachten wir eine Funktion $\mathbf{f}$ mehrerer Veränderlicher und einen Vektor $\mathbf{x}$, so dass die Jacobi-Matrix $D\mathbf{f}(\mathbf{x})$ invertierbar ist. Setze $\mathbf{z} = -D\mathbf{f}(\mathbf{x})^{-1}\mathbf{f}(\mathbf{x})$ und betrachte die Funktion

$$\varphi(t) = \|\mathbf{f}(\mathbf{x} + t\mathbf{z})\|_2^2$$

der reellen Veränderlichen t. Dann gilt

$$\begin{aligned} \varphi(0) &= \|\mathbf{f}(\mathbf{x})\|_2^2 \\ \varphi'(0) &= 2\mathbf{f}(\mathbf{x})^T D\mathbf{f}(\mathbf{x})\mathbf{z} \\ &= -2\|\mathbf{f}(\mathbf{x})\|_2^2. \end{aligned}$$

Daher gibt es ein $\delta > 0$ mit

$$\begin{aligned} \|\mathbf{f}(\mathbf{x} + t\mathbf{z})\|_2^2 &\leq (1 - t)\|\mathbf{f}(\mathbf{x})\|_2^2 \\ &< \|\mathbf{f}(\mathbf{x})\|_2^2 \qquad \text{für alle } 0 < t \leq \delta. \end{aligned}$$

Dies führt auf den folgenden Algorithmus:

Algorithmus II.14. Gedämpftes Newtonverfahren *zur Lösung nicht linearer Gleichungssysteme.*

(0) *Gegeben eine differenzierbare Funktion $\mathbf{f}$ mehrerer reeller Veränderlicher, ein Vektor $\mathbf{x}_0$ aus dem Definitionsbereich von $\mathbf{f}$ und eine Toleranz $\varepsilon > 0$ sowie obere Schranken K und N für die Zahl der Halbierungsschritte bzw. der Iterationen.*
(1) *Setze $i = 0$.*
(2) *Falls $\|\mathbf{f}(\mathbf{x}_i)\|_2 < \varepsilon$ ist, gebe den Vektor $\mathbf{x}_i$ als Näherung für die Nullstelle aus und beende das Vefahren. Andernfalls gehe zu Schritt 3.*

(3) *Berechne mit dem Gaußschen Eliminationsverfahren oder mit der LR-Zerlegung die Lösung* $\mathbf{z}$ *des linearen Gleichungssystems*

$$Df(\mathbf{x}_i)\mathbf{z} = -\mathbf{f}(\mathbf{x}_i).$$

Falls das Lösungsverfahren für dieses LGS mit der Fehlermeldung „das LGS hat keine eindeutige Lösung“ abbricht, beende das Verfahren mit einer entsprechenden Fehlermeldung. Andernfalls gehe zu Schritt 4.

(4) *Ausgehend von* $t_i = 1$ *bestimme durch sukzessives Halbieren ein* $t_i > 0$ *mit*

$$\|\mathbf{f}(\mathbf{x}_i + t_i\mathbf{z})\|_2^2 \leq (1 - \frac{1}{2}t_i)\|\mathbf{f}(\mathbf{x}_i)\|_2^2.$$

Falls nach K *Schritten kein solches* t_i *gefunden wurde, beende das Verfahren mit einer entsprechenden Meldung. Sonst setze*

$$\mathbf{x}_{i+1} = \mathbf{x}_i + t_i\mathbf{z}.$$

Erhöhe i *um* 1*. Falls* $i = N$ *ist, beende das Verfahren mit einer entsprechenden Meldung, sonst gehe zu Schritt 2 zurück.*

BEMERKUNG II.15. Die Dämpfung greift in der Regel nur in den ersten zwei bis drei Iterationen. Danach ist stets $t_i = 1$, so dass das gedämpfte Newtonverfahren dann mit dem ungedämpften Newtonverfahren identisch ist. Insbesondere bleibt die quadratische Konvergenz des Verfahrens erhalten.

BEISPIEL II.16. Wenden wir das gedämpfte Newtonverfahren auf die Funtion $f(x) = \arctan(x)$ mit dem Startwert $x_0 = 2$ an (vgl. Beispiel II.8 (S. 38) und Tabelle II.4 (S. 39)), erhalten wir die in Tabelle II.6 angegebenen Ergebnisse. Sie belegen nachdrücklich den Vorteil der Dämpfung.

TABELLE II.6. Gedämpftes Newtonverfahren für $\arctan(x)$

i	x_i	$f(x_i)$
0	2.000000	1.107148
1	-0.767871	-0.654841
2	0.273081	0.266581
3	-0.013380	-0.013379
4	0.000001	0.000001

Das folgende `Java`-Programm realisiert das gedämpfte Newtonverfahren in einer Dimension. Dabei ist `fct` wieder eine abstrakte Klasse, die für eine stetig differenzierbare Funktion den Funktionswert `f` und den Wert der Ableitung `df` liefert.

```
// Newton algorithm with damping
    public void dampedNewton() throws ZeroException {
        int it = 0;
        fx = fct.f(x);
        dfx = fct.df(x);
        while( it < maxit && Math.abs(fx) > EPS ) {
            if( Math.abs(dfx) <= EPS )
                throw new ZeroException(
                " WARNING: vanishing derivative");
            int count = 0;
            double factor = 1.0;
            double step = fx/dfx;
            while( count < MAXREF &&
                   Math.abs(fct.f(x - factor*step)) >
                   Math.sqrt(1.0 - factor/2.0)*Math.abs(fx) ) {
                factor /= 2.0;
                count++;
            }
            if( count >= MAXREF &&
                    Math.abs( fct.f(x - factor*step) ) >
                    Math.sqrt(1.0 - factor/2.0)*Math.abs(fx) )
                throw new ZeroException(" WARNING: damping failed");
            else {
                x -= factor*step;
                fx = fct.f(x);
                dfx = fct.df(x);
                it++;
            }
        }
    }    // end of dampedNewton
```

KAPITEL III

Interpolation

III.1. Lagrange-Interpolation

Wir betrachten die folgende Aufgabe:

> LAGRANGESCHES INTERPOLATIONSPROBLEM:
> Gegeben sind ein Intervall $[a, b] \subset \mathbb{R}$, eine stetige Funktion $f : [a, b] \to \mathbb{R}$, eine natürliche Zahl n und $n + 1$ Punkte (genannt KNOTEN) $a \leq x_0 < x_1 < \ldots < x_n \leq b$. Gesucht ist ein Polynom p vom Grade $\leq n$ mit
>
> $$p(x_i) = f(x_i) \tag{III.1}$$
>
> für alle $0 \leq i \leq n$.

BEMERKUNG III.1. In der Praxis sind in der Regel nur die Funktionswerte (genannt DATEN) $f(x_0), \ldots, f(x_n)$ bekannt. Die Algorithmen, die wir im Folgenden betrachten, benötigen nur die Knoten $x_0, \ldots, x_n$ und die zugehörigen Daten $f(x_0), \ldots, f(x_n)$.

Zuerst überlegen wir uns, dass das Lagrangesche Interpolationsproblem höchstens eine Lösung hat. Wären nämlich p_1 und p_2 zwei Polynome vom Grade $\leq n$, die die Interpolationsaufgabe (III.1) lösen, wäre ihre Differenz q ein Polynom vom Grade $\leq n$ mit den $n+1$ Nullstellen $x_0, \ldots, x_n$. Das geht nur, wenn q das Nullpolynom ist, also p_1 und p_2 übereinstimmen.

Eine Lösung der Interpolationsaufgabe (III.1) kann explizit angegeben werden:

> Das LAGRANGESCHE INTERPOLATIONSPOLYNOM
>
> $$L_n f(x) = \sum_{i=0}^{n} f(x_i) \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j} \tag{III.2}$$
>
> löst das Interpolationsproblem (III.1).

III.2. Die Newtonsche Darstellung des Lagrangeschen Interpolationspolynomes

Die Darstellung (III.2) des Lagrangeschen Interpolationpolynomes ist für praktische Rechnungen aus folgenden Gründen ungeeignet:

- Die Auswertung von (III.2) erfordert für *jedes* Argument $O(n^2)$ Operationen.
- Die Auswertung von (III.2) ist anfällig für Rundungsfehler, da durch kleine Zahlen dividiert werden muss.

Aus diesen Gründen wollen wir eine andere Form des Lagrangeschen Interpolationspolynomes angeben, die diese Schwierigkeiten vermeidet. Hierzu benötigen wir die DIVIDIERTEN DIFFERENZEN $f[x_i, \dots, x_{i+k}]$ von f zu den Punkten $x_0, \dots, x_n$. Sie sind rekursiv definiert durch die Vorschrift

DIVIDIERTE DIFFERENZEN:

$$f[x_i] = f(x_i)$$

$$\text{für } 0 \le i \le n,$$

$$f[x_i, \dots, x_{i+k+1}] = \frac{f[x_{i+1}, \dots, x_{i+k+1}] - f[x_i, \dots, x_{i+k}]}{x_{i+k+1} - x_i}$$

$$\text{für } 0 \le i \le n, 0 \le k \le n - i - 1.$$

Mit Hilfe der dividierten Differenzen ergibt sich folgende Darstellung des Lagrangeschen Interpolationspolynomes:

NEWTONSCHE DARSTELLUNG DES LAGRANGESCHEN INTERPOLATIONSPOLYNOMES:

$$L_n f(x) = f(x_0) + \sum_{i=1}^{n} f[x_0, ..., x_i] \prod_{j=0}^{i-1} (x - x_j). \tag{III.3}$$

Der folgende Algorithmus berechnet die dividierten Differenzen:

ALGORITHMUS III.2. *Berechnung dividierter Differenzen.*

(0) *Gegeben:*

$$x_0, \dots, x_n \text{ (KNOTEN)}$$

$$y_0 = f(x_0), \dots, y_n = f(x_n) \text{ (DATEN)}$$

Gesucht:

$$\triangle_{i,k} = f[x_i, ..., x_{i+k}] \quad 0 \le i \le n, 0 \le k \le n - i.$$

(1) *Für* $i = 0, 1, ..., n$ *setze*

$$\triangle_{i,0} = y_i.$$

(2) *Für* $k = 1, ..., n$ *und* $i = 0, 1, ..., n - k$ *berechne*

$$\triangle_{i,k} = \frac{\triangle_{i+1,k-1} - \triangle_{i,k-1}}{x_{i+k} - x_i}.$$

Der folgende Algorithmus wertet bei gegebenen dividierten Differenzen das Interpolationspolynom $L_n f$ in einem gegebenem Punkt aus und benutzt dabei die Newtonsche Darstellung:

Algorithmus III.3. Interpolationsformel von Newton.

(0) *Gegeben*

$x_0, \ldots, x_n$ *(Knoten)*

$\triangle_{0,0}, \ldots, \triangle_{0,n}$ *(zugehörige dividierte Differenzen)*

x^* *(Auswertungspunkt)*

Gesucht:

$$y^* = L_n f(x^*).$$

(1) *Setze*

$$y = \triangle_{0,n}.$$

(2) *Für* $k = n-1, n-2, ..., 0$ *berechne*

$$y = \triangle_{0,k} + (x^* - x_k)y.$$

(3) *Setze*

$$y^* = y.$$

Die Berechnung des Lagrangeschen Interpolationspolynomes mit Hilfe der Newtonschen Darstellung und der Algorithmen III.2 und III.3 hat folgende Vorteile:

- Sie erfordert *einmalig* $O(n^2)$ Operationen zur Berechnung der dividierten Differenzen und $O(n)$ Operationen für *jede* Auswertung des Polynomes.
- Sie ist nicht anfällig gegen Rundungsfehler.

Folgende `Java`-Programme realisieren die Algorithmen III.2 und III.3. Die arrays `nodes` und `data` enthalten die Knoten $x_0, \ldots, x_n$ und die Daten $f(x_0), \ldots, f(x_n)$; das array `y` speichert die dividierten Differenzen $f[x_0], f[x_0, x_1], \ldots, f[x_0, x_1, \ldots, x_n]$.

```
// initialize Newton polynomial
    public void NewtonPolynomial( double[] nodes, double[] data ) {
        dim = nodes.length;
        x = new double[dim];
        y = new double[dim];
        for( int i = 0; i < dim; i++ ) {
            x[i] = nodes[i];
            y[i] = data[i];
        }
        for( int k = 1; k < dim; k++ )
            for( int j = dim-1; j >= k; j-- )
                y[j] = (y[j] - y[j-1])/(x[j] - x[j-k]);
    }    // end of initialization
// evaluate Newton polynomial at point z
    public double f(double z) {
        double u = y[dim-1];
```

```
        for( int k = dim-2; k >= 0; k-- ) {
            u *= (z - x[k]);
            u += y[k];
        }
        return u;
    }   // end of evaluation
```

Beispiel III.4. Für $x_0 = -1$, $x_1 = -\frac{1}{3}$, $x_2 = \frac{1}{3}$, $x_3 = 1$ und $f(x) = |x|$ erhalten wir folgendes Schema

$$\begin{array}{r|llll}
-1 & 1 & & & \\
 & & -1 & & \\
-\frac{1}{3} & \frac{1}{3} & & \frac{3}{4} & \\
 & & 0 & & 0 \\
\frac{1}{3} & \frac{1}{3} & & \frac{3}{4} & \\
 & & 1 & & \\
1 & 1 & & &
\end{array}$$

und

$$\begin{aligned}
L_n f(x) &= 1 - (x+1) + \frac{3}{4}(x+1)(x+\frac{1}{3}) \\
&= -x + \frac{3}{4}(x^2 + \frac{4}{3}x + \frac{1}{3}) \\
&= \frac{3}{4}x^2 + \frac{1}{4}.
\end{aligned}$$

III.3. Genauigkeit der Lagrange-Interpolation

Für die Genauigkeit der Lagrange Interpolation gilt:

Die Funktion f sei $(n+1)$-mal stetig differenzierbar auf dem Intervall $[a,b]$. Dann gibt es zu jedem $x^* \in I$ ein $\eta^* = \eta^*(x^*) \in I$ mit

$$f(x^*) - L_n f(x^*) = \frac{1}{(n+1)!} f^{(n+1)}(\eta^*) \prod_{i=0}^{n} (x^* - x_i).$$

Insbesondere gilt

$$\max_{a \le x \le b} |f(x) - L_n f(x)| \le \frac{(b-a)^{n+1}}{(n+1)!} \max_{a \le x \le b} |f^{(n+1)}(x)|$$

und

$$\max_{a \le x \le b} |f(x) - L_n f(x)| \le h^{n+1} \max_{a \le x \le b} |f^{(n+1)}(x)|$$

mit

$$h = \max_{0 \le i \le n+1} |x_i - x_{i-1}|$$

und $x_{-1} = a$, $x_{n+1} = b$.

Der Fehler der Lagrange-Interpolation nimmt also mit wachsendem Polynomgrad n ab, wenn die Funktion f beliebig oft differenzierbar und

$$\sup_{n\in\mathbb{N}} \max_{a\le x\le b} |f^{(n+1)}(x)| < \infty$$

ist. Die einfachen Beispiele $f(x) = e^{2x}$, $a = 0$, $b = 1$ und $f(x) = \cos(2x)$, $a = 0$, $b = \pi$ zeigen aber, dass diese Bedingung sehr restriktiv ist.

Es gilt sogar das folgende negative Resultat:

Satz von Faber: Zu jeder Knotenmatrix $a \le x_0^{(n)} < \ldots < x_n^{(n)} \le b$, $n \in \mathbb{N}$, gibt es ein $f \in C(I, \mathbb{R})$, so dass für die Folge der zugehörigen Lagrangeschen Interpolationspolynome gilt

$$\lim_{n\to\infty} \max_{a\le x\le b} |f(x) - L_n f(x)| = \infty.$$

Der Fehler der Lagrange-Interpolation wächst also unter Umständen mit zunehmendem Polynomgrad. Daher ist dringend davon abzuraten, die Knotenzahl und damit den Polynomgrad zu groß zu wählen. Stattdessen sollte man das Intervall $[a, b]$ in kleine Teilintervalle zerlegen und dort separat mit einem relativ niedrigen Grad interpolieren. Dieser Ansatz liegt der Spline-Interpolation aus Paragraph III.4 zugrunde.

Beispiel III.5. Zur Illustration des Satzes von Faber geben wir in Tabelle III.1 den maximalen Fehler des Lagrangeschen Interpolationspolynoms der Funktion $|x|$ auf dem Intervall $[-1, 1]$ in n äquidistanten Punkten bzw. in n Čebyšev Punkten $-\cos(\frac{i\pi}{n-1})$, $0 \le i \le n - 1$, an. Zum Vergleich geben wir auch den maximalen Fehler des interpolierenden Splines aus Paragraph III.5 für äquidistante Knoten an. Diese Werte zeigen die Überlegenheit der Spline-Interpolation. Die Abbildungen III.1 – III.4 zeigen für einige Fälle die Funktion, die Interpolierende und die Fehlerkurve. Man beachte die unterschiedliche Skalierung der Abbildungen.

Tabelle III.1. Maximaler Fehler der Lagrange- und Spline-Interpolation der Funktion $|x|$ auf $[-1, 1]$ in n Knoten

n	Lagrange		Spline
	äquidistant	Čebyšev	äquidistant
5	0.1472	0.1422	0.0858
9	0.3157	0.0737	0.0425
17	11.1371	0.0372	0.0213
33	105717.8079	0.0186	0.0106

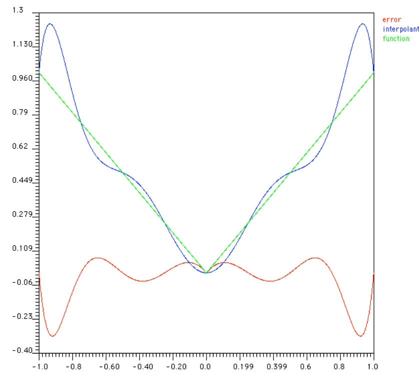


Abbildung III.1. Lagrange-Interpolation von $|x|$ in 9 äquidistanten Knoten

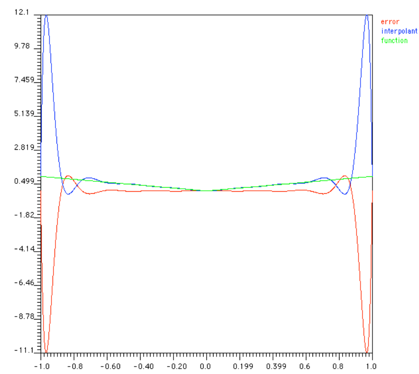


Abbildung III.2. Lagrange-Interpolation von $|x|$ in 17 äquidistanten Knoten

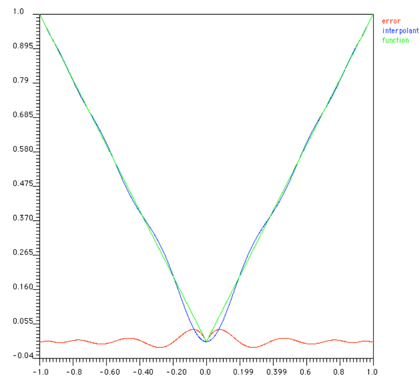


Abbildung III.3. Lagrange-Interpolation von $|x|$ in 17 Čebyšev-Knoten

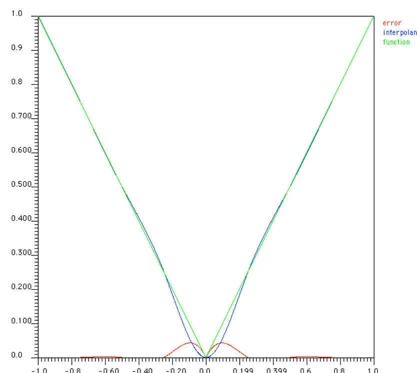


ABBILDUNG III.4. Spline-Interpolation von $|x|$ in 9 äquidistanten Knoten

III.4. Hermite-Interpolation

Insbesondere beim Computer Aided Design (CAD) stößt man auf Interpolationsaufgaben, bei denen Funktionswerte und Ableitungen zu interpolieren sind. Wir betrachten den folgenden Spezialfall:

HERMITESCHES INTERPOLATIONSPROBLEM:
Gegeben sind ein Intervall $[a, b] \subset \mathbb{R}$, eine differenzierbare Funktion $f : [a, b] \to \mathbb{R}$, eine natürliche Zahl n und $n + 1$ Knoten $a \le x_0 < x_1 < \ldots < x_n \le b$. Gesucht ist ein Polynom p vom Grade $\le 2n + 1$ mit

$$\begin{aligned} p(x_i) &= f(x_i) \\ p'(x_i) &= f'(x_i) \end{aligned} \tag{III.4}$$

für alle $0 \le i \le n$.

BEMERKUNG III.6. Wie bei der Lagrange-Interpolation sind in der Regel nur die Werte $f(x_0), f'(x_0), \ldots, f(x_n), f'(x_n)$ bekannt.

Man kann zeigen, dass die Interpolationsaufgabe (III.4) eine eindeutige Lösung besitzt und dass das Interpolationspolynom eine zu (III.3) analoge Darstellung besitzt. Der „Trick“ ist einfach, Knoten mehrfach aufzuführen und bei den dividierten Differenzen Differenzenquotienten mit verschwindendem Nenner durch geeignete Ableitungen zu ersetzen.

Um diese Vorgehensweise zu präzisieren, definieren wir Punkte $\hat{x}_0, \ldots, \hat{x}_{2n+1}$ durch die Vorschrift

$$\hat{x}_{2i} = \hat{x}_{2i+1} = x_i \text{ für } i = 0, 1, \ldots, n.$$

Dann definieren wir DIVIDIERTE DIFFERENZEN $f[\hat{x}_i, \ldots, \hat{x}_{i+k}]$ durch

$$f[\hat{x}_i] = f(\hat{x}_i)$$
$$\text{für } 0 \le i \le 2n+1$$
$$f[\hat{x}_i, \hat{x}_{i+1}] = \begin{cases} \dfrac{f(\hat{x}_{i+1}) - f(\hat{x}_i)}{\hat{x}_{i+1} - \hat{x}_i} & \text{falls } \hat{x}_{i+1} \ne \hat{x}_i, \\ f'(\hat{x}_i) & \text{falls } \hat{x}_{i+1} = \hat{x}_i \end{cases}$$
$$\text{für } 0 \le i \le 2n$$
$$f[\hat{x}_i, \ldots, \hat{x}_{i+k+1}] = \frac{f[\hat{x}_{i+1}, \ldots, \hat{x}_{i+k+1}] - f[\hat{x}_i, \ldots, \hat{x}_{i+k}]}{\hat{x}_{i+k+1} - \hat{x}_i}$$
$$\text{für } 0 \le i \le 2n+1, 0 \le k \le 2n-i.$$

Damit ergibt sich folgende Lösung der Hermiteschen Interpolationsaufgabe:

Das Polynom (HERMITESCHES INTERPOLATIONSPOLYNOM)

$$H_n f(x) = f(\hat{x}_0) + \sum_{i=1}^{2n+1} f[\hat{x}_0, \ldots, \hat{x}_i] \prod_{j=0}^{i-1} (x - \hat{x}_j)$$

löst die Hermitesche Interpolationsaufgabe (III.4).

BEISPIEL III.7. Für die Daten

$$\begin{aligned} x_0 &= -1, & x_1 &= 0, & x_2 &= 1 \\ f(x_0) &= 1, & f(x_1) &= 0, & f(x_2) &= 1 \\ f'(x_0) &= -1, & f'(x_1) &= 0, & f'(x_2) &= 1 \end{aligned}$$

erhalten wir folgendes Schema für die dividierten Differenzen

$$\begin{array}{r|cccccccccc}
-1 & 1 & & & & & & & & & \\
 & & -1 & & & & & & & & \\
-1 & 1 & & 0 & & & & & & & \\
 & & -1 & & 1 & & & & & & \\
0 & 0 & & 1 & & -\frac{1}{2} & & & & & \\
 & & 0 & & 0 & & 0 & & & & \\
0 & 0 & & 1 & & -\frac{1}{2} & & & & & \\
 & & 1 & & -1 & & & & & & \\
1 & 1 & & 0 & & & & & & & \\
 & & 1 & & & & & & & & \\
1 & 1 & & & & & & & & &
\end{array}$$

und das Hermitesche Interpolationspolynom

$$H_n f(x) = 1 - (x+1) + (x+1)^2 x - \frac{1}{2}(x+1)^2 x^2$$

$$= -\frac{1}{2}x^4 + \frac{3}{2}x^2.$$

III.5. Kubische Spline-Interpolation

Wegen des Satzes von Faber ist es nicht sinnvoll, den Polynomgrad bei der Interpolation zu hoch zu wählen. Wegen der Fehlerabschätzung für die Lagrange-Interpolation aus Abschnitt III.3 kann andererseits bei festem Grad die Genauigkeit der Polynominterpolation verbessert werden, indem der Abstand zwischen den Knoten verringert wird. Man betrachtet daher die Interpolation durch Splines. Das sind Funktionen, die stückweise Polynome sind und die global gewisse Differenzierbarkeitseigenschaften haben. Der Einfachheit halber beschränken wir uns hier auf den wichtigen Spezialfall der kubischen Splines.

Wir wählen $n+1$ Knoten $a = x_0 < x_1 < \ldots < x_n = b$ und halten diese im Folgenden fest. Eine Funktion u heißt KUBISCHER SPLINE (zu den Knoten $x_0, \ldots, x_n$), wenn u zweimal stetig differenzierbar ist und auf jedem Teilintervall $[x_k, x_{k+1}]$, $0 \le k \le n-1$, ein kubisches Polynom ist. Auf jedem dieser n Teilintervalle hat u vier Freiheitsgrade. Da u mitsamt seinen ersten beiden Ableitungen in den $n-1$ inneren Punkten $x_1, \ldots, x_{n-1}$ stetig sein soll, muss es $3(n-1)$ Zwangsbedingungen erfüllen. Insgesamt hat daher ein kubischer Spline $n+3$ Freiheitsgrade. Wenn wir eine gegebene Funktion f in den $n+1$ Punkten $x_0, \ldots, x_n$ interpolieren wollen, fehlen uns daher zwei Bedingungen. Diese müssen wir zusätzlich fordern. Zwei geläufige Bedingungen sind

$$u''(x_0) = 0, \qquad u''(x_n) = 0$$

und

$$u'(x_0) = f'(x_0), \qquad u'(x_n) = f'(x_n).$$

Wir beschränken uns auf die erste Variante und betrachten das folgende Interpolationsproblem:

KUBISCHE SPLINE-INTERPLATION:
Finde zu den Knoten $a = x_0 < \ldots < x_n = b$ und den Daten $f(x_0), \ldots, f(x_n)$ einen kubischen Spline u, der die Bedingungen

$$u(x_0) = f(x_0), \ldots, u(x_n) = f(x_n)$$

und

$$u''(x_0) = 0,\ u''(x_n) = 0$$

erfüllt.

Zur Lösung dieses Interpolationsproblems führen wir folgende Abkürzungen ein:

$$I_k = [x_k, x_{k+1}] \qquad , 0 \le k \le n-1,$$

$$\begin{aligned}
h_k &= x_{k+1} - x_k && ,0 \le k \le n-1, \\
\mu_k &= \frac{h_{k-1}}{h_{k-1} + h_k} && ,1 \le k \le n-1, \\
\lambda_k &= \frac{h_k}{h_{k-1} + h_k} && ,1 \le k \le n-1, \\
m_k &= u''(x_k) && ,0 \le k \le n.
\end{aligned}$$

Betrachten wir ein Intervall I_k. Da u'' dort ein Polynom ersten Grades ist, gilt

$$(u_{|I_k})''(x) = m_k + \frac{1}{h_k}(x - x_k)(m_{k+1} - m_k).$$

Wir integrieren diese Darstellung zweimal und erhalten

$$\begin{aligned}
(u_{|I_k})'(x) &= \alpha_k + (x - x_k)m_k + \frac{1}{2h_k}(x - x_k)^2(m_{k+1} - m_k) \\
u_{|I_k}(x) &= \beta_k + (x - x_k)\alpha_k + \frac{1}{2}(x - x_k)^2 m_k \\
&\quad + \frac{1}{6h_k}(x - x_k)^3(m_{k+1} - m_k)
\end{aligned}$$

mit noch unbekannten Integrationskonstanten α_k, β_k. Aus der Interpolationsbedingung $u(x_k) = f(x_k)$ erhalten wir sofort

$$\beta_k = f(x_k).$$

Hieraus und aus der Interpolationsbedingung $u(x_{k+1}) = f(x_{k+1})$ erhalten wir die Bedingung

$$f(x_{k+1}) = f(x_k) + \alpha_k h_k + \frac{1}{2}m_k h_k^2 + \frac{1}{6}(m_{k+1} - m_k)h_k^2$$

und damit die Bestimmungsgleichung

$$\alpha_k = \frac{f(x_{k+1}) - f(x_k)}{h_k} - \frac{1}{3}m_k h_k - \frac{1}{6}m_{k+1}h_k.$$

Damit sind die Integrationskonstanten α_k und β_k festgelegt. Aber die Werte $m_1, \ldots, m_{n-1}$ sind nach wie vor unbekannt. Diese erhalten wir aus der Forderung, dass u' in den Punkten $x_1, \ldots, x_{n-1}$ stetig sein soll. Aus den bisherigen Ergebnissen folgt

$$\begin{aligned}
(u_{|I_k})'(x_k) &= \alpha_k \\
&= \frac{f(x_{k+1}) - f(x_k)}{h_k} - \frac{1}{3}m_k h_k - \frac{1}{6}m_{k+1}h_k
\end{aligned}$$

und

$$\begin{aligned}
(u_{|I_k})'(x_{k+1}) &= \alpha_k + m_k h_k + \frac{1}{2}(m_{k+1} - m_k)h_k \\
&= \frac{f(x_{k+1}) - f(x_k)}{h_k} + \frac{1}{6}m_k h_k + \frac{1}{3}m_{k+1}h_k.
\end{aligned}$$

Wegen der Stetigkeit von u' im Punkt x_k muss

$$(u_{|I_k})'(x_k) = (u_{|I_{k+1}})'(x_k)$$

sein. Dies liefert die Bedingung

$$\begin{aligned}&\frac{f(x_{k+1}) - f(x_k)}{h_k} - \frac{1}{3}m_k h_k - \frac{1}{6}m_{k+1}h_k \\ &= \frac{f(x_k) - f(x_{k-1})}{h_{k-1}} + \frac{1}{6}m_{k-1}h_{k-1} + \frac{1}{3}m_k h_{k-1}\end{aligned}$$

bzw. nach elementarer Umformung

$$\begin{aligned}&\mu_k m_{k-1} + 2m_k + \lambda_k m_{k+1} \\ &= \frac{6}{h_k + h_{k-1}}\left[\frac{f(x_{k+1}) - f(x_k)}{h_k} - \frac{f(x_k) - f(x_{k-1})}{h_{k-1}}\right].\end{aligned}$$

Wenn wir beachten, dass aufgrund der Interpolationsbedingungen $m_0 = m_n = 0$ ist, sind dies $n-1$ Gleichungen für die Unbekannten $m_1, \ldots, m_{n-1}$. Man kann zeigen, dass dieses Gleichungssystem eindeutig lösbar ist.

Zusammenfassend erhalten wir damit den folgenden Algorithmus:

Algorithmus III.8. Kubische Spline-Interpolation.

(0) *Gegeben: Knoten $x_0 < \ldots < x_n$ und Daten $f(x_0), \ldots, f(x_n)$. Gesucht: kubischer Spline u mit $u(x_0) = f(x_0), \ldots, u(x_n) = f(x_n)$ und $u''(x_0) = 0$ sowie $u''(x_n) = 0$.*

(1) *Berechne folgende Größen*

$$\begin{aligned}h_k &= x_{k+1} - x_k &&, 0 \le k \le n-1, \\ \mu_k &= \frac{h_{k-1}}{h_{k-1} + h_k} &&, 1 \le k \le n-1, \\ \lambda_k &= \frac{h_k}{h_{k-1} + h_k} &&, 1 \le k \le n-1, \\ \delta_k &= \frac{f(x_{k+1}) - f(x_k)}{h_k} &&, 0 \le k \le n-1, \\ d_k &= \frac{6}{h_k + h_{k-1}}[\delta_k - \delta_{k-1}] &&, 1 \le k \le n-1.\end{aligned}$$

(2) *Setze*

$$m_0 = m_n = 0$$

und löse das LGS

$$\begin{pmatrix} 2 & \lambda_1 & & & & 0 \\ \mu_2 & 2 & \lambda_2 & & & \\ & \ddots & \ddots & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & & & \mu_{n-2} & 2 & \lambda_{n-2} \\ 0 & & & & \mu_{n-1} & 2 \end{pmatrix} \begin{pmatrix} m_1 \\ m_2 \\ \vdots \\ \vdots \\ m_{n-2} \\ m_{n-1} \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ \vdots \\ \vdots \\ d_{n-2} \\ d_{n-1} \end{pmatrix}.$$

(3) *Der gesuchte kubische Spline u hat auf dem Intervall* $I_k = [x_k, x_{k+1}]$, $0 \le k \le n-1$, *die Darstellung*

$$\begin{aligned} u_{|I_k} &= f(x_k) + (x - x_k)\{\delta_k - \frac{1}{3}m_k h_k - \frac{1}{6}m_{k+1}h_k\} \\ &\quad + \frac{1}{2}m_k(x - x_k)^2 \\ &\quad + \frac{1}{6h_k}(m_{k+1} - m_k)(x - x_k)^3. \end{aligned}$$

Die folgende `Java`-Klasse realisiert obigen Algorithmus. Dabei ist `Function` eine abstrakte Klasse, die eine Methode `f` bereitstellt, die zu einem gegebenen Punkt `z` einen Funktionswert `f(z)` liefert.

```
public class CubicSpline extends Function {
// items relevant for cubic spline
    public int dim;                 // number of nodes
    public double[] x;              // nodes
    public double[] y;              // data
    public int intnodes;            // number of moments = dim-2
    public double[] moments;        // spline moments
    public double[][] a;            // matrix for computing moments
    public double[] b;              // rhs for computing moments
    public double[] mm;             // auxiliary array for spline evaluation
// initialize
    CubicSpline( double[] nodes, double[] data ) {
        dim = nodes.length;
        x = new double[dim];
        y = new double[dim];
        for( int i = 0; i < dim; i++ ) {
            x[i] = nodes[i];
            y[i] = data[i];
        }
        intnodes = dim - 2;
        moments = new double[intnodes];
        a = new double[intnodes][3];
        b = new double[intnodes];
        initMatrix();
        initRhs();
        computeMoments();
        mm = new double[dim];
        for ( int i = 1; i < dim-1; i++ )
            mm[i] = moments[i-1];
    }    // end of initialization
// initialize matrix
    private void initMatrix() {
        for ( int i = 0; i < intnodes; i++ )
            a[i][1] = 2.0;                          // diagonal elements
        for ( int i = 1; i < intnodes; i++ ) {
            a[i][0]   = (x[i+1] - x[i])/(x[i+2] - x[i]);
            a[i-1][2] = (x[i+1] - x[i])/(x[i+1] - x[i-1]);
        }
    }
// initialize right-hand side
    private void initRhs() {
        for ( int i = 0; i < intnodes; i++ )
```

```
            b[i] = 6.0*(  (y[i+2] - y[i+1])/(x[i+2] - x[i+1])
                    - (y[i+1] - y[i])/(x[i+1] - x[i]))/(x[i+2] - x[i]);
    }
// compute the moments
    private void computeMoments() {
        double faktor;
        for ( int i = 0; i < intnodes-1; i++ ) {    // elimination
            faktor = a[i+1][0]/a[i][1];
            a[i+1][1] -= a[i][2]*faktor;
            b[i+1] -= b[i]*faktor;
        }
        moments[intnodes-1] = b[intnodes-1]/a[intnodes-1][1];
        for ( int i = intnodes-2; i >= 0; i-- ) {
            moments[i] = b[i] - a[i][2]*moments[i+1];
            moments[i] /= a[i][1];
        }
    }
// find interval to which evaluation point z belongs
    private int findInterval(double z) {
        int k = -1;
        while ( x[k+1] <= z && k < dim-2 ) k++;
        return k;
    }
// return function value at point z
    public double f(double z) {
        double u = y[0];
        if( z >= x[dim-1] )
            u = y[dim-1];
        if( z > x[0] && z < x[dim-1] ) {
            int k = findInterval(z);
            double zz = z - x[k];
            double xx = x[k+1] - x[k];
            u = y[k]
                  + zz*((y[k+1] - y[k])/xx - mm[k]*xx/3.0 - mm[k+1]*xx/6.0
                  + zz*(mm[k]/2.0 + zz*(mm[k+1] - mm[k])/(6.0*xx) ) );
        }
        return u;
    }    // end of evaluation
}    // end of CubicSpline
```

Für den Fehler der Spline-Interpolation gilt:

> Die Funktion f sei viermal stetig differenzierbar und erfülle die Bedingung
> $$f''(a) = f''(b) = 0.$$
> Dann gilt für den Fehler der kubischen Spline-Interpolation
> $$\max_{a \le x \le b} |f(x) - u(x)| \le h^4 \max_{a \le x \le b} |f^{(4)}(x)|$$
> mit $h = \max_{0 \le k \le n-1} |x_{k+1} - x_k|$.

Beispiel III.9. Interpolieren wir die Funktion $|x|$ auf dem Intervall $[-1, 1]$ in n äquidistanten Punkten erhalten wir für den Fehler der kubischen Spline-Interpolation die in Tabelle III.1 (S. 49) wiedergegebenen

Werte (vgl. Abbildung III.4 (S. 51)). Sie zeigen einerseits die Überlegenheit der Spline-Interpolation über die Lagrange-Interpolation, andererseits den Einfluss der mangelnden Differenzierbarkeit der Funktion f.

III.6. Bézier-Darstellung von Polynomen und Splines

In vielen Anwendungen wie z.B. dem Computer Aided Design oder der Seitenbeschreibungssprache Postscript wird die Bézier-Darstellung von Polynomen und Splines benutzt. Sie beruht auf den BERNSTEIN-POLYNOMEN

$$B_{n,k}(x) = \binom{n}{k} x^k (1-x)^{n-k}, \quad 0 \le k \le n, n \ge 0.$$

Für jedes n bilden die Bernstein Polynome $B_{n,0}, \ldots, B_{n,n}$ eine Basis des Raumes der Polynome vom Grad $\le n$. Jedes Polynom p vom Grad $\le n$ kann daher eindeutig dargestellt werden in der Form

$$p(x) = \sum_{k=0}^{n} b_k B_{n,k}(x).$$

Dies ist die sogenannte BÉZIER-DARSTELLUNG von p. Die Koeffizienten $b_0, \ldots, b_n$ heißen die BÉZIER-PUNKTE von p.

Die Auswertung eines Polynoms in Bézier-Darstellung erfolgt mit dem Algorithmus von de Casteljau. Zu seiner Beschreibung definieren wir ausgehend von den Bézier-Punkten $b_0, \ldots, b_n$ die Hilfspolynome

$$b_{r,s}(x) = \sum_{k=r}^{s} b_k B_{s-r,k-r}(x), \quad 0 \le r \le s \le n.$$

Dann ist offensichtlich $b_{0,n}(x) = p(x)$ und $b_{r,r}(x) = b_r$, $0 \le r \le n$. Aus den Identitäten

$$\begin{aligned} B_{j+1,0}(x) &= (1-x)B_{j,0}(x), \\ B_{j+1,i}(x) &= (1-x)B_{j,i}(x) + xB_{j,i-1}(x), \\ B_{j+1,j+1}(x) &= xB_{j,j}(x), \end{aligned}$$

erhält man dann die Rekursionsformel

$$b_{r,s}(x) = (1-x)b_{r,s-1}(x) + xb_{r+1,s}(x).$$

Sie erlaubt die rekursive Berechnung von $b_{0,n}(x)$ aus den Bézier-Punkten $b_k = b_{k,k}(x)$, $0 \le k \le n$. Damit ergibt sich folgender Algorithmus:

ALGORITHMUS III.10. ALGORITHMUS VON DE CASTELJAU *zur Auswertung eines Polynomes in Bézier-Darstellung.*

(0) *Gegeben:* n *der Grad des Polynoms* p, $b_0, \ldots, b_n$ *die Bézier-Punkte des Polynoms,* $x \in [0,1]$ *der Auswertungspunkt*
Gesucht: $p(x)$ *der Wert des Polynoms* p *im Punkt* x

(1) *Für* $k = 0, 1, \ldots, n$ *setze*

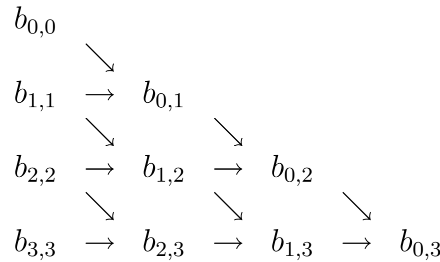
$$b_{k,k} = b_k.$$

(2) *Für* $j = 1, 2, \ldots, n$ *und für* $k = 0, 1, \ldots, n-j$ *berechne*

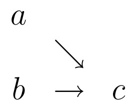
$$s = k + j$$
$$b_{k,s} = (1-x)b_{k,s-1} + xb_{k+1,s}.$$

(3) *Setze* $p(x) = b_{0,n}$.

Für die praktische Rechnung ordnet man die Werte $b_{r,s}$ in einem Dreiecksschema an. Die Werte mit gleichem ersten Index stehen in derselben Diagonalen; die Werte mit dem gleichen zweiten Index stehen in derselben Zeile.



Das Symbol



steht dabei für die Rechenvorschrift

$$c = (1-x)a + xb.$$

Beispiel III.11. Für das kubische Polynom p mit den Bézier-Punkten $b_0 = 2$, $b_1 = 10$, $b_2 = 7$, $b_3 = 0$ und den Auswertungspunkt $x = 0.4$ erhalten wir folgendes Schema

2			
10	5.2		
7	8.8	6.64	
0	4.2	6.96	6.768

und den Funktionswert $p(0.4) = 6.768$.

Wir betrachten nun die Bézier-Darstellung eines kubischen Splines u zu der Unterteilung $a = x_0 < x_1 < \ldots < x_n = b$. Auf dem Teilintervall $I_k = [x_k, x_{k+1}]$, $0 \le k \le n-1$, der Länge $h_k = x_{k+1} - x_k$ schreiben wir dazu

$$u_{|I_k}(x) = \sum_{\ell=0}^{3} b_{k,\ell} B_{3,\ell}(\frac{x - x_k}{h_k}).$$

Dann gilt

$$\begin{aligned}
(u_{|I_k})(x_k) &= b_{k,0},\\
(u_{|I_k})(x_{k+1}) &= b_{k,3},\\
(u_{|I_k})'(x_k) &= \frac{3}{h_k}(b_{k,1} - b_{k,0}),\\
(u_{|I_k})'(x_{k+1}) &= \frac{3}{h_k}(b_{k,3} - b_{k,2}),\\
(u_{|I_k})''(x_k) &= \frac{6}{h_k^2}(b_{k,2} - 2b_{k,1} + b_{k,0}),\\
(u_{|I_k})''(x_{k+1}) &= \frac{6}{h_k^2}(b_{k,3} - 2b_{k,2} + b_{k,1}).
\end{aligned}$$

Anders als im vorigen Abschnitt betrachten wir jetzt die Interpolationsaufgabe:

> Finde zu den Knoten $x_0 < \ldots < x_n$ und den Daten $f(x_0)$, $\ldots, f(x_n)$ und $f'(x_0)$, $f'(x_n)$ einen kubischen Spline u mit
>
> $$u(x_k) = f(x_k), \quad k = 0, 1, \ldots, n$$
>
> und
>
> $$\begin{aligned} u'(x_0) &= f'(x_0),\\ u'(x_n) &= f'(x_n). \end{aligned}$$

(Beachte, dass wir jetzt in den Randpunkten $x_0 = a$ und $x_n = b$ Bedingungen an u' statt an u'' im vorigen Abschnitt stellen!)

Wir bezeichnen jetzt mit $M_k = u'(x_k)$, $0 \le k \le n$, die Werte der ersten Ableitung von u in den Knoten. Aus der Interpolationsbedingung $u(x_k) = f(x_k)$ folgt für jedes $k \in \{0, \ldots, n-1\}$

$$b_{k,0} = f(x_k), \quad b_{k,3} = f(x_{k+1}).$$

Aus den Formeln für $(u_{|I_k})'$ erhalten wir dann

$$\begin{aligned}
b_{k,1} &= b_{k,0} + \frac{h_k}{3} M_k\\
&= f(x_k) + \frac{h_k}{3} M_k,\\
b_{k,2} &= b_{k,3} - \frac{h_k}{3} M_{k+1}\\
&= f(x_{k+1}) - \frac{h_k}{3} M_{k+1}.
\end{aligned}$$

Damit sind die Bézier-Punkte $b_{k,i}$, $0 \le k \le n-1$, $0 \le i \le 3$, durch die Daten $f(x_0), \ldots, f(x_n)$ und die Werte $M_0, \ldots, M_n$ ausgedrückt. Aus

der Stetigkeit von u'' in den Punkten $x_1, \ldots, x_{n-1}$ erhalten wir schließlich für $k = 1, \ldots, n-1$ die Bedingung

$$\begin{aligned}\frac{6}{h_{k-1}^2}(b_{k-1,3} - 2b_{k-1,2} + b_{k-1,1}) &= (u_{|I_{k-1}})''(x_k) \\ &= (u_{|I_k})''(x_k) \\ &= \frac{6}{h_k^2}(b_{k,2} - 2b_{k,1} + b_{k,0}).\end{aligned}$$

Dies liefert nach einigen Umformungen die Bestimmungsgleichungen

$$\begin{aligned}&\lambda_k M_{k-1} + 2M_k + \mu_k M_{k+1} \\ =& \frac{3}{h_k + h_{k-1}}\left[h_{k-1}\frac{f(x_{k+1}) - f(x_k)}{h_k} + h_k \frac{f(x_k) - f(x_{k-1})}{h_{k-1}}\right], \\ &1 \le k \le n-1.\end{aligned}$$

Dabei ist zu beachten, dass $M_0 = f'(x_0)$ und $M_n = f'(x_n)$ durch die Interpolationsbedingungen festgelegt sind.

Insgesamt erhalten wir damit folgenden Algorithmus:

ALGORITHMUS III.12. *Kubische Spline-Interpolation in Bézier-Darstellung.*

(0) *Gegeben: Knoten* $x_0 < \ldots < x_n$ *und Daten* $f(x_0), \ldots, f(x_n)$ *und* $f'(x_0)$, $f'(x_n)$.
Gesucht: Kubischer Spline u *in Bézier-Darstellung mit* $u'(x_0) = f'(x_0)$, $u(x_0) = f(x_0)$, $u(x_1) = f(x_1)$, …, $u(x_n) = f(x_n)$, $u'(x_n) = f'(x_n)$

(1) *Berechne folgende Größen*

$$\begin{aligned}h_k &= x_{k+1} - x_k && ,0 \le k \le n-1, \\ \mu_k &= \frac{h_{k-1}}{h_k + h_{k-1}} && ,1 \le k \le n-1, \\ \lambda_k &= \frac{h_k}{h_k + h_{k-1}} && ,1 \le k \le n-1, \\ \delta_k &= \frac{f(x_{k+1}) - f(x_k)}{h_k} && ,0 \le k \le n-1, \\ D_k &= \frac{3}{h_k + h_{k-1}}[h_{k-1}\delta_k + h_k\delta_{k-1}] && ,1 \le k \le n-1.\end{aligned}$$

(2) *Setze*

$$M_0 = f'(x_0), \quad M_n = f'(x_n)$$

und löse das lineare Gleichungssystem

$$\begin{pmatrix} 2 & \mu_1 & & & & 0 \\ \lambda_2 & 2 & \mu_2 & & & \\ & \ddots & \ddots & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & & & \lambda_{n-2} & 2 & \mu_{n-2} \\ 0 & & & & \lambda_{n-1} & 2 \end{pmatrix} \begin{pmatrix} M_1 \\ M_2 \\ \vdots \\ M_{n-2} \\ M_{n-1} \end{pmatrix}$$

$$= \begin{pmatrix} D_1 - \lambda_1 f'(x_0) \\ D_2 \\ \vdots \\ D_{n-2} \\ D_{n-1} - \mu_{n-1} f'(x_n) \end{pmatrix}.$$

(3) *Für* $k = 0, \ldots, n-1$ *setze*

$$\begin{aligned} b_{k,0} &= f(x_k), \\ b_{k,1} &= f(x_k) + \frac{h_k}{3} M_k, \\ b_{k,2} &= f(x_{k+1}) - \frac{h_k}{3} M_{k+1}, \\ b_{k,3} &= f(x_{k+1}). \end{aligned}$$

Dann hat der gesuchte kubische Spline u *auf dem Intervall* $I_k = [x_k, x_{k+1}]$ *die Bézier-Darstellung*

$$u_{|I_k}(x) = \sum_{\ell=0}^{3} b_{k,\ell} B_{3,\ell}\left(\frac{x - x_k}{h_k}\right).$$

Die Auswertung des Splines auf den Teilintervallen erfolgt mit dem Algorithmus von de Casteljau, Algorithmus III.10.

KAPITEL IV

Integration

IV.1. Motivation

Viele bestimmte Integrale kann man nicht explizit bestimmen. Man muss sie numerisch näherungsweise berechnen. Ein Ansatz besteht darin, den Integranden durch ein Polynom zu interpolieren und das Interpolationspolynom zu integrieren. Wir wollen diesen Ansatz an zwei einfachen Beispielen demonstrieren.

Bezeichne mit p das lineare Interpolationspolynom zu der Funktion f und den Knoten a und b, $a < b$. Gleichung (III.2) aus Paragraph III.1 liefert

$$p(x) = f(a)\frac{b-x}{b-a} + f(b)\frac{x-a}{b-a}.$$

Hieraus ergibt sich

$$\begin{aligned}\int_a^b f(x)dx &\approx \int_a^b p(x)dx \\ &= f(a)\frac{1}{2}(b-a) + f(b)\frac{1}{2}(b-a).\end{aligned}$$

Geometrisch ist dies die Fläche des Trapezes mit Grundseite $[a, b]$ und Höhen $f(a)$ und $f(b)$.

Bezeichne mit q das quadratische Interpolationspolynom zu f und den Knoten a, $\frac{a+b}{2}$, b. Gleichung (III.2) aus Paragraph III.1 liefert jetzt nach elementaren Umformungen

$$\begin{aligned}q(x) = f(a)&\frac{a+b-2x}{b-a}\frac{b-x}{b-a} \\ &+ f(\frac{a+b}{2})4\frac{x-a}{b-a}\frac{b-x}{b-a} \\ &+ f(b)\frac{x-a}{b-a}\frac{2x-a-b}{b-a}.\end{aligned}$$

Damit erhalten wir

$$\begin{aligned}\int_a^b f(x)dx &\approx \int_a^b q(x)dx \\ &= f(a)\frac{1}{6}(b-a) + f(\frac{a+b}{2})\frac{2}{3}(b-a) + f(b)\frac{1}{6}(b-a).\end{aligned}$$

IV.2. Quadraturformeln

Zur näherungsweisen numerischen Berechnung bestimmter Integrale benutzt man QUADRATURFORMELN; das sind Ausdrücke der Form

$$Q(f) = \sum_{k=0}^{n} c_k f(x_k).$$

Die Zahlen $c_0, c_1, \ldots, c_n$ heißen die GEWICHTE der Quadraturformel; $x_0, x_1, \ldots, x_n \in [a,b]$ sind die KNOTEN ($[a,b]$ ist der Integrationsbereich). Die Güte einer Quadraturformel wird durch ihre ORDNUNG gemessen. Dies ist die maximale Zahl K, sodass alle Polynome vom Grad $\leq K$ durch die Quadraturformel exakt integriert werden.

Die wichtigsten Quadraturformeln sind:

$$(b-a)f(\frac{a+b}{2}) \quad \text{Mittelpunktsregel, Ordnung 1}$$

$$\frac{b-a}{2}[f(a)+f(b)] \quad \text{Trapezregel, Ordnung 1}$$

$$\frac{b-a}{6}[f(a)+4f(\frac{a+b}{2})+f(b)] \quad \text{Simpsonregel, Ordnung 3.}$$

Geometrisch beschreibt die Trapezregel die Fläche des Trapezes mit Grundseite $[a,b]$ und Höhen $f(a)$ und $f(b)$.

Betrachte eine beliebige Quadraturformel

$$Q(f) = \sum_{k=0}^{n} c_k f(x_k)$$

mit Knoten $a \leq x_0 < \ldots < x_n \leq b$ und das Polynom

$$p(x) = (x-x_0)^2 \cdot \ldots \cdot (x-x_n)^2$$

vom Grad $2n+2$. Dann ist

$$Q(p) = 0 \quad \text{und} \quad \int_a^b p(x)dx > 0.$$

Daher gilt:

Die Ordnung einer Quadraturformel mit $n+1$ Knoten ist höchstens $2n+1$.

IV.3. Newton-Cotes-Formeln

Bei fest vorgegebenen Knoten (z.B. ÄQUIDISTANT: $x_k = a + \frac{b-a}{n} k$, $0 \le k \le n$) kann man die Gewichte so bestimmen, dass man eine Quadraturformel der Ordnung n erhält. Die entsprechenden Gewichte sind gegeben durch

$$c_k = \int_a^b \prod_{\substack{i=0 \\ i \neq k}}^{n} \frac{x - x_i}{x_k - x_i} \qquad (0 \le k \le n).$$

Man erhält diese Gewichte durch Integration des Lagrangeschen Interpolationspolynoms zu f und den Knoten $x_0, \ldots, x_n$ (vgl. (III.2) in Paragraph III.1). Die entsprechenden Quadraturformeln heißen NEWTON-COTES-FORMELN. Die Trapezregel ist ein Beispiel für eine Newton-Cotes-Formel.

IV.4. Gauß-Formeln

Lässt man auch die Wahl der Knoten frei, kann man Quadraturformeln der Ordnung $2n + 1$ konstruieren. Derartige Formeln heißen GAUSS-FORMELN. Die Mittelpunktsregel ist ein Beispiel für eine Gauß-Formel. Weitere Beispiele sind:

$$\frac{b-a}{2}\Big[f(\frac{3+\sqrt{3}}{6}a + \frac{3-\sqrt{3}}{6}b) + f(\frac{3-\sqrt{3}}{6}a + \frac{3+\sqrt{3}}{6}b)\Big]$$

Ordnung 3,

$$\frac{b-a}{18}\Big[5f(\frac{5+\sqrt{15}}{10}a + \frac{5-\sqrt{15}}{10}b) + 8f(\frac{a+b}{2}) + 5f(\frac{5-\sqrt{15}}{10}a + \frac{5+\sqrt{15}}{10}b)\Big]$$

Ordnung 5.

Für beliebiges n sind die Knoten der entsprechenden Gauß-Formel die auf das Intervall $[a, b]$ transformierten Nullstellen des $(n + 1)$-ten Legendre-Polynoms; die Gewichte sind durch obige Formel für Newton-Cotes-Formeln gegeben. Knoten und Gewichte können numerisch stabil und effizient durch Lösen eines geeigneten Eigenwertproblems bestimmt werden.

IV.5. Zusammengesetzte Quadraturformeln

Man kann offensichtlich versuchen, ein zu berechnendes Integral immer genauer numerisch anzunähern, indem man Quadraturformeln mit immer mehr Knoten und entsprechend höherer Ordnung benutzt. Man

kann jedoch zeigen, dass dieser einfache Weg nicht funktioniert: Egal wie man die Folge der Knoten wählt, es gibt immer eine Funktion, dessen Integral mit wachsender Knotenzahl *immer schlechter* approximiert wird. Dies ist eine Folge des Satzes von Faber aus Abschnitt III.3.

Einen Ausweg aus dieser Zwickmühle bieten ZUSAMMENGESETZTE QUADRATURFORMELN. Die Idee ist einfach: Das Integrationsintervall $[a, b]$ wird in viele gleich große Intervalle unterteilt, auf jedem Teilintervall wird eine feste Quadraturformel, z.B. die Trapezregel, angewandt und die Ergebnisse werden aufaddiert. Zur Verdeutlichung wähle ein $n \geq 2$ und setze $h = \frac{b-a}{n}$, dann lauten die ZUSAMMENGESETZTE MITTELPUNKTSREGEL, TRAPEZREGEL und SIMPSONREGEL:

$$h \sum_{i=1}^{n} f(a + \frac{2i-1}{2} h),$$

$$\frac{h}{2} \left[f(a) + 2 \sum_{i=1}^{n-1} f(a + ih) + f(b) \right],$$

$$\frac{h}{6} \left[f(a) + 2 \sum_{i=1}^{n-1} f(a + ih) + 4 \sum_{i=1}^{n} f(a + \frac{2i-1}{2} h) + f(b) \right].$$

Der Fehler dieser drei Näherungen zu dem Integral $\int_a^b f(x)dx$ verhält sich wie ch^2 für die Mittelpunkts- und Trapezregel und wie ch^4 für die Simpsonregel.

Generell gilt:

> Basiert die zusammengesetzte Quadraturformel auf einer Formel der Ordnung K, so hat sie bei hinreichend oft differenzierbaren Integranden f einen Fehler von ch^{K+1}.

Dabei ist c eine Konstante, die nur von der betreffenden Quadraturformel und dem Integranden f abhängt. Ein Fehler von ch^2 bzw. ch^4 bedeutet, dass sich der Fehler bei Halbieren von h um den Faktor 4 bzw. 16 reduziert.

Die folgenden `Java`-Programme realisieren die zusammengesetzte Mittelpunkts-, Trapez- und Simpsonregel sowie die zusammengesetzten Gauß-Formeln der Ordnungen 3, 5 und 7. Dabei ist `fct` eine abstrakte Klasse, die für eine stetige Funktion den Funktionswert `f` liefert.

```
// midpoint rule
    private double midpoint( int nn ) {
        double integral = 0.0;
        double h = (rightBoundary - leftBoundary)/nn;
        double x = leftBoundary + h/2;
```

```
        for( int i = 0; i < nn; i++ ) {
            integral += fct.f( x );
            x += h;
        }
        return integral*h;
    }    // end of midpoint
// trapezoidal rule
    private double trapezoidal( int nn ) {
        double integral;
        double h = (rightBoundary - leftBoundary)/nn;
        integral = (fct.f( leftBoundary ) + fct.f( rightBoundary ))/2;
        double x = leftBoundary + h;
        for( int i = 1; i < nn; i++ ) {
            integral += fct.f( x );
            x += h;
        }
        return integral*h;
    }    // end of trapezoidal
// simpson rule
    private double simpson( int nn ) {
        return (trapezoidal( nn ) + 2.0*midpoint( nn ))/3.0;
    }    // end of simpson
// 2 point Gauss rule
    private double Gauss2( int nn ) {
        double[] x = new double[2];
        double[] w = new double[2];
        double integral = 0.0;
        double a = leftBoundary;
        double h = (rightBoundary - leftBoundary)/nn;
        w[0] = w[1] = 0.5;
        x[0] = (3.0 - Math.sqrt(3.0))/6.0;
        x[1] = 1.0 - x[0];
        x[0] *= h,    x[1] *= h;
        for( int i = 0; i < nn; i++ ) {
            integral += w[0]*fct.f( a + x[0] ) + w[1]*fct.f( a + x[1] );
            a += h;
        }
        return integral*h;
    }    // end of Gauss2
// 3 point Gauss rule
    private double Gauss3( int nn ) {
        double[] x = new double[3];
        double[] w = new double[3];
        double integral = 0.0;
        double a = leftBoundary;
        double h = (rightBoundary - leftBoundary)/nn;
        w[0] = w[2] = 5.0/18.0,    w[1] = 4.0/9.0;
        x[0] = (5.0 - Math.sqrt(15.0))/10.0;
        x[2] = 1.0 - x[0],    x[1] = 0.5;
        x[0] *= h,    x[1] *= h,    x[2] *= h;
        for( int i = 0; i < nn; i++ ) {
            integral += w[0]*fct.f( a + x[0] ) + w[1]*fct.f( a + x[1] )
                        + w[2]*fct.f( a + x[2] );
            a += h;
        }
        return integral*h;
    }    // end of Gauss3
```

```
// 4 point Gauss rule
    private double Gauss4( int nn ) {
        double[] x = new double[4];
        double[] w = new double[4];
        double integral = 0.0;
        double a = leftBoundary;
        double h = (rightBoundary - leftBoundary)/nn;
        w[0] = w[3] = (18.0 - Math.sqrt(30.0))/72.0;
        w[1] = w[2] = 0.5 - w[0];
        x[0] = (1.0 - Math.sqrt( (15.0 + 2.0*Math.sqrt(30.0))/35.0 ))/2.0;
        x[1] = (1.0 - Math.sqrt( (15.0 - 2.0*Math.sqrt(30.0))/35.0 ))/2.0;
        x[2] = 1.0 - x[1],    x[3] = 1.0 - x[0];
        x[0] *= h,    x[1] *= h,    x[2] *= h,    x[3] *= h;
        for( int i = 0; i < nn; i++ ) {
            integral += w[0]*fct.f( a + x[0] ) + w[1]*fct.f( a + x[1] );
            integral += w[2]*fct.f( a + x[2] ) + w[3]*fct.f( a + x[3] );
            a += h;
        }
        return integral*h;
    }   // end of Gauss4
```

Beispiel IV.1. Tabelle IV.1 zeigt die Ergebnisse der zusammengesetzten Mittelpunkts-, Trapez- und Simpsonregel für das Integral

$$\int_0^1 \frac{3}{2}\sqrt{x}dx = 1.$$

Für die zusammengesetzten Gauß-Formeln mit 2, 3 und 4 Knoten (Ordnungen 3, 5 und 7) sind die entsprechenden Ergebnisse in Tabelle IV.2 wiedergegeben.

Tabelle IV.1. Mittelpunkts-, Trapez- und Simpsonregel angewandt auf das Integral $\int_0^1 \frac{3}{2}\sqrt{x}$

h	Mittelpunkt	Trapez	Simpson
1	1.06066	0.75000	0.95711
$\frac{1}{2}$	1.02452	0.99530	0.98479
$\frac{1}{4}$	1.00947	0.96493	0.99462
$\frac{1}{8}$	1.00355	0.98720	0.99810
$\frac{1}{16}$	1.00131	0.99537	0.99933
$\frac{1}{32}$	1.00047	0.99834	0.99976
$\frac{1}{64}$	1.00017	0.99941	0.99992
$\frac{1}{128}$	1.00006	0.99979	0.99997

IV.6. Das Romberg-Verfahren

Es soll $\int_a^b f(x)dx$ näherungsweise berechnet werden. Wir bezeichnen dazu mit $T_{0,k}$, $k = 0, 1, \ldots, N$, das Ergebnis der zusammengesetzten

TABELLE IV.2. Gaußsche Formeln mit 2, 3 und 4 Knoten angewandt auf das Integral $\int_0^1 \frac{3}{2}\sqrt{x}$

h	2 Knoten	3 Knoten	4 Knoten
1	1.01083	1.00377	1.00174
$\frac{1}{2}$	1.00386	1.00133	1.00062
$\frac{1}{4}$	1.00137	1.00047	1.00022
$\frac{1}{8}$	1.00048	1.00017	1.00008
$\frac{1}{16}$	1.00017	1.00006	1.00003
$\frac{1}{32}$	1.00006	1.00002	1.00001
$\frac{1}{64}$	1.00002	1.00001	1.00000
$\frac{1}{128}$	1.00001	1.00000	1.00000

Trapezregel für dieses Integral zu $h = (b-a)2^{-k}$, d.h. zur Unterteilung in 2^k Teilintervalle. Wir wählen ein $K \in \{1, \dots, N\}$ und berechnen nun rekursiv

ROMBERG-VERFAHREN:

$$T_{i+1,k} = \frac{4^{i+1}T_{i,k+1} - T_{i,k}}{4^{i+1} - 1}$$
$$i = 0, 1, \dots, K-1, \ k = 0, 1, \dots, N-i-1.$$

Jedes $T_{i,k}$ ist eine Näherung für $\int_a^b f(x)dx$ mit einem Fehler $c_i \left(\frac{b-a}{4^k}\right)^{i+1}$, d.h.

$$\int_a^b f(x)dx - T_{i,k} = O(4^{-k(i+1)}).$$

Dabei hängen die Faktoren c_i nur von i und dem Integranden f ab. Der Übergang von $T_{i,k}$ zu $T_{i,k+1}$ bedeutet also eine Reduktion des Fehlers um den Faktor 4^{i+1}.

Das folgende `Java`-Programm realisiert das Romberg-Verfahren. Die Methode `trapezoidal` ist die im vorigen Abschnitt angegebene zusammengesetzte Trapezregel. Die Variable `numberOfIntervals` gibt die ursprüngliche Zahl der Teilintervalle für die zusammengesetzte Trapezregel an.

```
// Romberg scheme
    private void romberg() {
        double factor = 4.0;
        int ni = numberOfIntervals;
        for( int level = 0; level <= numberOfRefinements; level++ ) {
            quad[0][level] = trapezoidal( ni );
            ni *=2;
```

```
        }
        for( int column = 1; column <= numberOfRefinements; column++ ) {
            for( int level = 0; level <= numberOfRefinements - column;
                 level++ )
                quad[column][level] = (factor*quad[column-1][level+1]
                                      - quad[column-1][level])
                                      /(factor - 1.0);
            factor *= 4.0;
        }
    }   // end of romberg
```

Beispiel IV.2. Tabelle IV.3 zeigt die Ergebnisse des Romberg-Verfahrens angewandt auf das Integral $\int_0^1 \frac{3}{2}\sqrt{x} = 1$.

Tabelle IV.3. Romberg-Verfahren für $\int_0^1 \frac{3}{2}\sqrt{x} = 1$

k	$T_{0,k}$	$T_{1,k}$	$T_{2,k}$	$T_{3,k}$	$T_{4,k}$
0	0.750000	0.957107	0.986635	0.995411	0.998389
1	0.905330	0.984789	0.995274	0.998378	0.999431
2	0.964925	0.994619	0.998329	0.999426	0.999799
3	0.987195	0.998097	0.999409	0.999797	0.999929
4	0.995372	0.999327	0.999791	0.999928	
5	0.998338	0.999762	0.999926		
6	0.999406	0.999916			
7	0.999788				

IV.7. Spezielle Integranden

In diesem Paragraphen gehen wir stichwortartig auf mögliche Modifikationen der bisher betrachteten Techniken ein, wenn der Integrand nicht genügend glatt oder der Integrationsbereich unbeschränkt ist.
1: Ist f auf den Teilintervallen $[a_i, a_{i+1}]$, $0 \le i \le m-1$, $a = a_0 < \ldots < a_m = b$ hinreichend oft differenzierbar, aber global nicht glatt, so zerlege man $\int_a^b f dx$ in die Teilintegrale $\int_{a_i}^{a_{i+1}} f dx$ und approximiere diese separat.
2: Besitzt f in einer der Intervallgrenzen eine Singularität, so hilft häufig eine geeignete Substitution weiter; z.B.

$$\int_0^b \frac{\cos x}{\sqrt{x}} dx = \int_0^{\sqrt{b}} 2\cos t^2 dt.$$

3: Eine andere Möglichkeit den Fall 2 zu behandeln, besteht in einer Aufspaltung des Integrals verbunden mit einer schnell konvergenten Reihenentwicklung des Integranden; z.B.

$$\int_0^b \frac{\cos x}{\sqrt{x}} dx = \int_\varepsilon^b \frac{\cos x}{\sqrt{x}} dx + \int_0^\varepsilon \frac{\cos x}{\sqrt{x}} dx$$

$$= \int_{\varepsilon}^{b} \frac{\cos x}{\sqrt{x}} dx + \sum_{k=0}^{\infty} (-1)^k \frac{1}{(2k)!(2k+\frac{1}{2})} \varepsilon^{2k+\frac{1}{2}}$$

wobei die Reihe für kleines ε sehr schnell konvergiert.
4: Eine andere Möglichkeit im Falle 2 ist die Subtraktion einer Funktion, die die gleiche Singularität aufweist und die exakt integrierbar ist; z.B.

$$\begin{aligned} \int_0^b \frac{\cos x}{\sqrt{x}} dx &= \int_0^b \frac{\cos x - 1}{\sqrt{x}} dx + \int_0^b \frac{1}{\sqrt{x}} dx \\ &= \int_0^b \underbrace{\frac{\cos x - 1}{\sqrt{x}}}_{\in C^1} dx + 2\sqrt{b}. \end{aligned}$$

5: Bei uneigentlichen Integralen der Form $\int_0^\infty f dx$ oder $\int_{-\infty}^\infty f dx$ hilft häufig die Verwendung einer Gaußschen Formel zur Gewichtsfunktion e^{-x} oder e^{-x^2}.
6: Durch geeignete Transformationen der Form $x = \frac{t}{1-t}$ oder $x = -\ln(t)$ oder $t = \frac{e^x-1}{e^x+1}$ können Integrale der Form 5 auf solche der Form $\int_a^b g(t)dt$ zurückgeführt werden. Allerdings handelt man sich dabei häufig Singularitäten des transformierten Integranden ein.
7: Bei Integralen der Form 5 hilft häufig auch das Abschneiden des Intervalls. So ist z.B.

$$\int_0^\infty e^{-x^2} dx = \int_0^a e^{-x^2} dx + R_a$$

mit

$$\begin{aligned} 0 < R_a &= \int_a^\infty e^{-x^2} dx \\ &< \int_a^\infty e^{-ax} dx \\ &= \frac{1}{a} e^{-a^2} \end{aligned}$$

und z.B.

$$0 < R_3 < 5 \cdot 10^{-5}.$$

IV.8. Mehrdimensionale Integrationsbereiche

Mehrdimensionale Integrationsbereiche zerlegt man in einfache Teilgebiete wie Dreiecke, Tetraeder, Rechtecke oder Würfel. Für diese Teilgebiete leitet man dann Quadraturformeln her. Das Integral über den

ursprünglichen Integrationsbereich wird dann durch eine zusammengesetzte Quadraturformel approximiert, deren Bausteine die Quadraturformeln für die entsprechenden Teilbereiche sind.

Besonders einfach ist die Situation für Rechtecke und Würfel. Sind nämlich

$$Q_1(f) = \sum_{i=0}^{n} c_i f(x_i)$$
$$Q_2(f) = \sum_{j=0}^{m} d_j f(y_j)$$

zwei Quadraturformeln der Ordnung K für $\int_a^b f(x)dx$ bzw. $\int_c^d f(y)dy$, ist

$$Q(f) = \sum_{i=0}^{n} \sum_{j=0}^{m} c_i d_j f(x_i, y_j)$$

eine Quadraturformel der Ordnung K für $\int_a^b \int_c^d f(x,y)dxdy$. Analog erhält man aus drei Formeln

$$Q_1(f) = \sum_{i=0}^{n} c_i f(x_i)$$
$$Q_2(f) = \sum_{j=0}^{m} d_j f(y_j)$$
$$Q_3(f) = \sum_{k=0}^{l} e_k f(z_k)$$

der Ordnung K für eindimensionale Integrale mit

$$Q(f) = \sum_{i=0}^{n} \sum_{j=0}^{m} \sum_{k=0}^{l} c_i d_j e_k f(x_i, y_j, z_k)$$

eine Formel der Ordnung K für ein Dreifachintegral über einen Würfel.

Dreiecke und Tetraeder bildet man zuerst durch eine affine Transformation $\mathbf{x} \mapsto A\mathbf{x} + \mathbf{b}$ auf das Referenzdreieck $\{\mathbf{x} \in \mathbb{R}^2 : x_1 \geq 0, x_2 \geq 0, x_1 + x_2 \leq 1\}$ bzw. das Referenztetraeder $\{\mathbf{x} \in \mathbb{R}^3 : x_1 \geq 0, x_2 \geq 0, x_3 \geq 0, x_1 + x_2 + x_3 \leq 1\}$ ab. Dann leitet man Quadraturformeln für Integrale über diesen Referenzmengen her und transformiert auf das ursprüngliche Dreieck bzw. Tetraeder zurück. Bei dieser Transformation werden die Knoten der Quadraturformeln mit der inversen Transformation $\mathbf{y} \mapsto A^{-1}(\mathbf{y} - \mathbf{b})$ abgebildet und die Gewichte mit der Determinante $\det A$ multipliziert. So erhält man z.B. mit den Punkten

$$x_0 = (\frac{1}{3}, \frac{1}{3}), x_1 = (\frac{1}{2}, \frac{1}{2}), x_2 = (0, \frac{1}{2}), x_3 = (\frac{1}{2}, 0)$$

und den Vorschriften

$$Q_0(f) = \frac{1}{2} f(x_0)$$
$$Q_1(f) = \frac{1}{6} \sum_{i=1}^{3} f(x_i)$$

zwei Quadraturformeln der Ordnung 1 bzw. 2 für Integrale über das Referenzdreieck.

Für ein allgemeines Dreieck T mit der Fläche A, den Eckpunkten $\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3$, dem Schwerpunkt $\mathbf{m} = \frac{1}{3}(\mathbf{p}_1 + \mathbf{p}_2 + \mathbf{p}_3)$ und den Kantenmittelpunkten $\mathbf{q}_1 = \frac{1}{2}(\mathbf{p}_1 + \mathbf{p}_2)$, $\mathbf{q}_2 = \frac{1}{2}(\mathbf{p}_2 + \mathbf{p}_3)$, $\mathbf{q}_3 = \frac{1}{2}(\mathbf{p}_3 + \mathbf{p}_1)$ ergeben sich die Quadraturformeln

$$Q_0(f) = A f(\mathbf{m}),$$
$$Q_1(f) = \frac{1}{3} A \sum_{i=1}^{3} f(\mathbf{q}_i).$$

KAPITEL V

Eigenwertprobleme

V.1. Übersicht

Die wichtigsten Verfahren zur numerischen Berechnung von Eigenwerten und Eigenvektoren sind:

(1) Die Potenzmethode und das Verfahren der Rayleigh-Quotienten für die Berechnung des betragsmäßig größten Eigenwertes.
(2) Das inverse Verfahren von Wielandt und das Verfahren der inversen Rayleigh-Quotienten für die Berechnung des betragsmäßig kleinsten Eigenwertes einer invertierbaren Matrix.
(3) Das QR-Verfahren zur Berechnung aller Eigenwerte.

Wir werden hier nur die Verfahren aus (1) und (2) besprechen.

V.2. Die Potenzmethode

Zur Beschreibung der Idee nehmen wir an, dass die Matrix A die reellen Eigenwerte $\mu_1, \ldots, \mu_n$ mit zugehörigen Eigenvektoren $\mathbf{y}_1, \ldots, \mathbf{y}_n$ hat und dass gilt $|\mu_1| > |\mu_2| \geq \ldots \geq |\mu_n|$. Weiter sei $\mathbf{z} = \alpha_1 \mathbf{y}_1 + \ldots + \alpha_n \mathbf{y}_n$ ein gegebener Vektor mit $\alpha_1 \neq 0$. Dann gilt für alle $m \in \mathbb{N}$

$$\begin{aligned} A^m \mathbf{z} &= \alpha_1 \mu_1^m \mathbf{y}_1 + \ldots + \alpha_n \mu_n^m \mathbf{y}_n \\ &= \mu_1^m \left\{ \alpha_1 \mathbf{y}_1 + \alpha_2 \left(\frac{\mu_2}{\mu_1} \right)^m \mathbf{y}_2 + \ldots + \alpha_n \left(\frac{\mu_n}{\mu_1} \right)^m \mathbf{y}_n \right\}. \end{aligned}$$

Wegen

$$\left| \frac{\mu_n}{\mu_1} \right| \leq \ldots \leq \left| \frac{\mu_2}{\mu_1} \right| < 1$$

konvergiert der Ausdruck in den geschweiften Klammern für $m \to \infty$ gegen $\alpha_1 \mathbf{y}_1$. Daher gilt

$$\lim_{m \to \infty} \frac{\|A^m \mathbf{z}\|_2}{\|A^{m-1} \mathbf{z}\|_2} = |\mu_1|.$$

Diese Idee wird in folgendem Algorithmus konkretisiert. Hierbei bezeichnet $(\mathbf{u}, \mathbf{v}) = \mathbf{u}^T \mathbf{v}$ das Euklidische Skalarprodukt zweier Vektoren und $\|\mathbf{u}\|_2 = (\mathbf{u}, \mathbf{u})^{\frac{1}{2}} = (\mathbf{u}^T \mathbf{u})^{\frac{1}{2}}$ die Euklidische Norm eines Vektors.

Algorithmus V.1. Potenzmethode *zur Berechnung des betragsmäßig größten Eigenwertes.*

(0) *Gegeben:*

A *(Matrix)*

$\mathbf{x}_0$ *(Startnäherung für einen Eigenvektor mit* $\|\mathbf{x}_0\|_2 = 1$*)*

(1) *Für* $m = 0, 1, 2, \ldots$ *führe folgende Schritte aus:*

$$\begin{aligned}
\mathbf{u}_{m+1} &= A\mathbf{x}_m, \\
\sigma_{m+1} &= \operatorname{sgn}((\mathbf{u}_{m+1}, \mathbf{x}_m)), \\
\lambda_{m+1} &= \sigma_{m+1}\|\mathbf{u}_{m+1}\|_2, \\
\mathbf{x}_{m+1} &= \sigma_{m+1}\frac{1}{\|\mathbf{u}_{m+1}\|_2}\mathbf{u}_{m+1}.
\end{aligned}$$

Hat A die Eigenwerte $\mu_1, \ldots, \mu_n$ (mehrfache Eigenwerte werden mehrfach aufgeführt!) und gilt $|\mu_1| = \ldots = |\mu_p| > |\mu_{p+1}| \geq \ldots \geq |\mu_n|$ sowie $\mu_1 = \ldots = \mu_p$, kann man zeigen, dass die Zahlen λ_m gegen den Eigenwert μ_1 und die Vektoren $\mathbf{x}_m$ gegen einen Eigenvektor zum Eigenwert μ_1 konvergieren. Dabei wird der Fehler jeweils pro Iteration um den Faktor $\frac{|\mu_{p+1}|}{|\mu_1|}$ reduziert. Die Konvergenzgeschwindigkeit hängt also ab vom Abstand zwischen dem betragsmäßig größten und zweitgrößten Eigenwert.
Falls $|\mu_1| = \ldots = |\mu_p| > |\mu_{p+1}| \geq \ldots \geq |\mu_n|$ aber $\mu_1 \neq \ldots \neq \mu_p$ ist, konvergieren die Zahlen λ_m nach wie vor gegen den Eigenwert μ_1, aber die Vektoren $\mathbf{x}_m$ müssen nicht mehr gegen einen zugehörigen Eigenvektor konvergieren.

Das folgende `Java`-Programm realisiert Algorithmus V.1:

```
// power method
    public void power() throws LinearAlgebraException {
        double ew = 0.0, diff = 1.0, norm = 1.0;
        int s = 1, iter = 0;
        b = new double[dim];
        norm = Math.sqrt( innerProduct(x, x)/dim );
        if ( norm > EPS )
            multiply(x, x, 1.0/norm);
        else
            throw new LinearAlgebraException(
            " ERROR: zero vector after "+iter+" iterations");
        while ( iter < maxit && diff > EPS ) {
            b = multiply(a, x);
            s = ( innerProduct(b, x) >= 0 ) ? 1 : -1;
            diff = ew;
            norm = Math.sqrt( innerProduct(b, b)/dim );
            if ( norm < EPS )
                throw new LinearAlgebraException(
                " ERROR: zero vector after "+iter+" iterations");
            ew = s*norm;
            residuals[iter] = ew;
            multiply(x, b, 1.0/ew);
            diff = Math.abs( ew - diff );
            iter++;
        }
```

```
        iterations = Math.max(iter, 1);
    }    // end of power
// inner product of two vectors
    public double innerProduct(double[] u, double[] v) {
        double prod = 0;
        for( int i = 0; i < dim; i++ )
            prod += u[i]*v[i];
        return prod;
    }    // end of inner product
// multiply matrix c with vector y
    public double[] multiply(double[][] c, double[] y) {
        double[] result = new double[dim];
        for( int i = 0; i < dim; i++ )
            result[i] = innerProduct(c[i], y);
        return result;
    }    // end of multiply
// multiply vector v with double c and store on u
    public void multiply(double[] u, double[] v, double c) {
        for( int i = 0; i < dim; i++ )
            u[i] = v[i]*c;
    }    // end of multiply
```

V.3. Rayleigh-Quotienten

Wir nehmen nun an, dass die Matrix A symmetrisch ist. Dann kann man die Eigenvektoren paarweise orthogonal und normiert wählen, d.h. $(\mathbf{y}_i, \mathbf{y}_j) = 0$ für $i \neq j$ und $(\mathbf{y}_i, \mathbf{y}_i) = 1$. Dabei bezeichnet $(\cdot, \cdot)$ das Euklidische Skalarprodukt.

Damit erhalten wir für alle $m \in \mathbb{N}$ und $k \in \{m, m+1\}$

$$\begin{aligned}(A^k\mathbf{z}, A^m\mathbf{z}) &= (\alpha_1\mu_1^k\mathbf{y}_1 + \ldots + \alpha_n\mu_n^k\mathbf{y}_n, \alpha_1\mu_1^m\mathbf{y}_1 + \ldots + \alpha_n\mu_n^m\mathbf{y}_n) \\ &= \alpha_1^2\mu_1^{m+k} + \ldots + \alpha_n^2\mu_n^{m+k} \\ &= \mu_1^{m+k}\left\{\alpha_1^2 + \alpha_2^2\left(\frac{\mu_2}{\mu_1}\right)^{m+k} + \ldots + \alpha_n^2\left(\frac{\mu_n}{\mu_1}\right)^{m+k}\right\}.\end{aligned}$$

Der Ausdruck in den geschweiften Klammern konvergiert für $m \to \infty$ gegen α_1^2. Daraus folgt

$$\lim_{m\to\infty} \frac{(A^{m+1}\mathbf{z}, A^m\mathbf{z})}{(A^m\mathbf{z}, A^m\mathbf{z})} = \mu_1.$$

Diesmal hängt die Konvergenzgeschwindigkeit aber von $(\frac{\mu_2}{\mu_1})^2$ statt von $\frac{\mu_2}{\mu_1}$ ab, so dass die Iteration schneller konvergiert.

Diese Idee wird in folgendem Algorithmus umgesetzt:

Algorithmus V.2. Rayleigh-Quotienten-Iteration *zur Bestimmung des betragsmäßig größten Eigenwertes einer symmetrischen Matrix.*

(0) *Gegeben:* $\mathbf{x}_0$ *(Startwert mit* $\|\mathbf{x}_0\|_2 = 1$*)*
(1) *Für* $m = 0, 1, \ldots$ *führe folgende Schritte aus:*

$$\mathbf{u}_{m+1} = A\mathbf{x}_m,$$

$$\mathbf{x}_{m+1} = \frac{1}{\|\mathbf{u}_{m+1}\|_2}\mathbf{u}_{m+1},$$
$$\lambda_{m+1} = (\mathbf{x}_m, \mathbf{u}_{m+1}).$$

Unter den gleichen Voraussetzungen wie bei Algorithmus V.1 (S. 75), kann man zeigen, dass die Zahlen λ_m gegen den betragsmäßig größten Eigenwert von A konvergieren. Dabei wird der Fehler pro Iteration um den Faktor $(\frac{|\mu_{p+1}|}{|\mu_1|})^2$ reduziert. Algorithmus V.2 konvergiert also doppelt so schnell wie Algorithmus V.1 (S. 75).

Das folgende `Java`-Programm realisiert Algorithmus V.2. Die Methoden `innerProduct` und `multiply` sind die selben wie bei der Potenzmethode.

```
// Rayleigh quotient iteration
    public void rayleigh() throws LinearAlgebraException {
        double ew = 0.0, diff = 1.0, norm = 1.0;
        int iter = 0;
        b = new double[dim];
        norm = Math.sqrt( innerProduct(x, x)/dim );
        if ( norm > EPS )
            multiply(x, x, 1.0/norm);
        else
            throw new LinearAlgebraException(
            " ERROR: zero vector after "+iter+" iterations");
        while ( iter < maxit && diff > EPS ) {
            b = multiply(a, x);
            diff = ew;
            ew = innerProduct(b, x)/dim;
            norm = Math.sqrt( innerProduct(b, b)/dim );
            if ( norm < EPS )
                throw new LinearAlgebraException(
                " ERROR: zero vector after "+iter+" iterations");
            residuals[iter] = ew;
            multiply(x, b, 1.0/norm);
            diff = Math.abs( ew - diff );
            iter++;
        }
        iterations = Math.max(iter, 1);
    }    // end of rayleigh
}
```

V.4. Inverse Iteration von Wielandt

Wendet man Algorithmus V.1 (S. 75) auf die Matrix A^{-1} an und bildet die Kehrwerte der entsprechenden Größen, erhält man einen Algorithmus zur Berechnung des betragsmäßig kleinsten Eigenwertes. Aus Effizienzgründen wird dabei die Matrix A^{-1} *nicht* berechnet, sondern $\mathbf{z} = A^{-1}\mathbf{x}$ durch Lösen des Gleichungssystems $A\mathbf{z} = \mathbf{x}$ bestimmt.

ALGORITHMUS V.3. INVERSE ITERATION VON WIELANDT *zur Berechnung des betragsmäßig kleinsten Eigenwertes.*

(0) *Gegeben:* $\mathbf{x}_0$ *(Startwert mit* $\|\mathbf{x}_0\|_2 = 1$*)*
(1) *Für* $m = 0, 1, \ldots$ *führe folgende Schritte aus:*

(a) *Löse das LGS*

$$A\mathbf{u}_{m+1} = \mathbf{x}_m.$$

(b) *Berechne*

$$\begin{aligned}
\sigma_{m+1} &= \operatorname{sgn}((\mathbf{u}_{m+1}, \mathbf{x}_m)),\\
\rho_{m+1} &= \frac{\sigma_{m+1}}{\|\mathbf{u}_{m+1}\|_2},\\
\mathbf{x}_{m+1} &= \sigma_{m+1}\frac{1}{\|\mathbf{u}_{m+1}\|_2}\mathbf{u}_{m+1}.
\end{aligned}$$

Hat die Matrix A die Eigenwerte $\mu_1, \ldots, \mu_n$ und gilt $|\mu_1| \geq \ldots \geq |\mu_{n-r-1}| > |\mu_{n-r}| = \ldots = |\mu_n| > 0$ sowie $\mu_{n-r} = \ldots = \mu_n$, kann man zeigen, dass die Zahlen ρ_m gegen den Eigenwert μ_n und die Vektoren $\mathbf{x}_m$ gegen einen Eigenvektor von A zum Eigenwert μ_n konvergieren.

Das folgende `Java`-Programm realisiert Algorithmus V.3. Die Methoden `innerProduct` und `multiply` sind die selben wie bei der Potenzmethode. Die Methoden `lrElimination`, `permutation`, `forSolve` und `backSolve` lösen ein lineares Gleichungssystem mit der LR-Zerlegung und sind in Paragraph I.4 wiedergegeben.

```
// inverse power method
    public void inversePower() throws LinearAlgebraException {
        double ew = 0.0, diff = 1.0, norm = 1.0;
        int s = 1, iter = 0;
        b = new double[dim];
        d = new double[dim];
        norm = Math.sqrt( innerProduct(x, x)/dim );
        if ( norm > EPS )
            multiply(x, x, 1.0/norm);
        else
            throw new LinearAlgebraException(
            " ERROR: zero vector after "+iter+" iterations");
        lrElimination();
        while ( iter < maxit && diff > EPS ) {
            copy(b, x);     // methods of permutation, forSolve, backSolve
            copy(d, x);       // take rhs b and give solution x
            permutation();
            forSolve();
            backSolve();
            copy(b, x);            // store solution on b
            copy(x, d);            // retrieve old x
            s = ( innerProduct(b, x) >= 0 ) ? 1 : -1;
            diff = ew;
            norm = Math.sqrt( innerProduct(b, b)/dim );
            if ( norm < EPS )
                throw new LinearAlgebraException(
                " ERROR: zero vector after "+iter+" iterations");
            ew = s/norm;
            residuals[iter] = ew;
            multiply(x, b, ew);
            diff = Math.abs( ew - diff );
            iter++;
        }
```

```
        iterations = Math.max(iter, 1);
    }    // end of inversePower
}
// copy vector v to vector u
    public void copy(double[] u, double[] v) {
        for( int i = 0; i < dim; i++ )
            u[i] = v[i];
    }    // end of copy
```

V.5. Inverse Rayleigh-Quotienten-Iteration

Die Idee des vorigen Abschnittes kann auch auf Algorithmus V.2 (S. 77) übertragen werden. Dies führt auf folgenden Algorithmus:

Algorithmus V.4. Inverse Rayleigh-Quotienten-Iteration *zur Bestimmung des betragsmäßig kleinsten Eigenwertes einer symmetrischen Matrix.*

(0) *Gegeben:* $\mathbf{x}_0$ *(Startwert mit* $\|\mathbf{x}_0\|_2 = 1$*)*

(1) *Für* $m = 0, 1, \ldots$ *führe folgende Schritte aus:*

(a) *Löse das LGS*

$$A\mathbf{u}_{m+1} = \mathbf{x}_m.$$

(b) *Berechne*

$$\mathbf{x}_{m+1} = \frac{1}{\|\mathbf{u}_{m+1}\|_2}\mathbf{u}_{m+1},$$
$$\rho_{m+1} = (\mathbf{x}_m, \mathbf{u}_{m+1})^{-1}.$$

Unter den gleichen Voraussetzungen wie bei Algorithmus V.3 (S. 78), kann man zeigen, dass die Zahlen ρ_m gegen den Eigenwert μ_n von A konvergieren. Allerdings konvergiert Algorithmus V.4 doppelt so schnell wie Algorithmus V.3 (S. 78).

Das folgende `Java`-Programm realisiert Algorithmus V.4. Die Methoden `innerProduct` und `multiply` sind die selben wie bei der Potenzmethode. Die Methoden `lrElimination`, `permutation`, `forSolve` und `backSolve` lösen ein lineares Gleichungssystem mit der LR-Zerlegung und sind in Paragraph I.4 wiedergegeben.

```
// inverse Rayleigh quotient iteration
    public void inverseRayleigh() throws LinearAlgebraException {
        double ew = 0.0, diff = 1.0, norm = 1.0;
        int iter = 0;
        b = new double[dim];
        d = new double[dim];
        norm = Math.sqrt( innerProduct(x, x)/dim );
        if ( norm > EPS )
            multiply(x, x, 1.0/norm);
        else
            throw new LinearAlgebraException(
            " ERROR: zero vector after "+iter+" iterations");
        lrElimination();
        while ( iter < maxit && diff > EPS ) {
            copy(b, x);      // methods of permutation, forSolve, backSolve
```

```
            copy(d, x);         // take rhs b and give solution x
            permutation();
            forSolve();
            backSolve();
            copy(b, x);            // store solution on b
            copy(x, d);            // retrieve old x
            diff = ew;
            ew = dim/innerProduct(b, x);
            norm = Math.sqrt( innerProduct(b, b)/dim );
            if ( norm < EPS )
                throw new LinearAlgebraException(
                " ERROR: zero vector after "+iter+" iterations");
            residuals[iter] = ew;
            multiply(x, b, 1.0/norm);
            diff = Math.abs( ew - diff );
            iter++;
        }
        iterations = Math.max(iter, 1);
    }    // end of inverseRayleigh
}
```

V.6. Berechnung eines Eigenvektors

Wir nehmen an, dass wir eine Näherung $\tilde{\mu}$ für einen Eigenwert μ_i der Matrix A bestimmt haben, und wollen einen zugehörigen Eigenvektor $\tilde{\mathbf{y}}$ bestimmen. Wir setzen voraus, dass $\tilde{\mu}$ dichter an μ_i liegt als an jedem anderen Eigenwert von A, d.h.

$$|\tilde{\mu} - \mu_i| < |\tilde{\mu} - \mu_j| \text{ für jeden Eigenwert } \mu_j \text{ von } A \text{ mit } \mu_j \neq \mu_i.$$

Dann ist $\mu_i - \tilde{\mu}$ der betragsmäßig kleinste Eigenwert der Matrix $A - \tilde{\mu}I$ und jeder andere Eigenwert dieser Matrix hat einen echt größeren Betrag. Außerdem ist jeder Eigenvektor von $A - \tilde{\mu}I$ zum Eigenwert $\mu_i - \tilde{\mu}$ ein Eigenvektor von A zum Eigenwert μ_i. Daher können wir Algorithmus V.3 (S. 78) auf die Matrix $A - \tilde{\mu}I$ anwenden und erhalten eine Folge von Vektoren $\mathbf{x}_m$, die gegen einen Eigenvektor von A zum Eigenwert μ_i konvergiert.

Im allgemeinen liefern schon wenige Iterationen eine hinreichend genaue Approximation für einen solchen Eigenvektor.

Man beachte, dass in jeder Iteration des Verfahrens ein lineares Gleichungssystem der Form

$$(A - \tilde{\mu}I)\mathbf{u}_{m+1} = \mathbf{x}_m$$

gelöst werden muss.

KAPITEL VI

Gewöhnliche Differentialgleichungen

VI.1. Motivation

Viele der in der Praxis auftretenden gewöhnlichen Differentialgleichungen (im Folgenden kurz: gDgl) können nicht analytisch gelöst werden. Stattdessen muss man ihre Lösung numerisch approximieren. In diesem Kapitel wollen wir einen kurzen Überblick über die hierfür gebräuchlichsten Verfahren und die bei ihrer Anwendung zu berücksichtigenden Aspekte geben.

Für die Motivation beschränken wir uns auf den einfachsten Fall und betrachten das ANFANGSWERTPROBLEM (im Folgenden kurz: AWP)

$$\begin{aligned} y' &= f(t, y(t)) \\ y(t_0) &= y_0 \end{aligned}$$

mit einer hinreichend glatten Funktion f. Für die numerische Lösung führen wir Gitterpunkte $t_0 < t_1 < \ldots$ ein und bezeichnen die numerische Approximation für $y(t_i)$ mit η_i oder $\eta(t_i, h_i)$. Dabei bezeichnet $h_i = t_i - t_{i-1}$ die Schrittweite. Zunächst betrachten wir der Einfachheit halber nur äquidistante Gitterpunkte, d.h. $h_i = h$ für alle i. In der Praxis muss man aber veränderliche Schrittweiten h_i vorsehen und diese mittels Schrittweitenkontrolle zusammen mit der Lösung adaptiv bestimmen.

Angenommen wir kennen die Lösung y des AWP im Punkte t. Wegen $y'(t) = f(t, y(t))$ gibt $f(t, y(t))$ die Steigung der Tangente an die Lösungskurve im Punkt t an. Daher sollte $y(t) + hf(t, y(t))$ eine passable Näherung an $y(t + h)$ sein. Diese Überlegung führt auf:

EXPLIZITES EULERVERFAHREN

$$\begin{aligned} \eta_0 &= y_0 \\ \eta_{i+1} &= \eta_i + hf(t_i, \eta_i) \\ t_{i+1} &= t_i + h. \end{aligned}$$

Genauso gut hätten wir eine Näherung η_{i+1} für $y(t + h)$ aus der Bedingung herleiten können, dass die Tangente durch die Lösungskurve

im Punkte $t_{i+1} = t_i + h$ durch den Punkt (t_i, y_i) gehen soll, d.h. $y_i = \eta_{i+1} - hf(t_i + h, \eta_{i+1})$. Dies führt auf:

IMPLIZITES EULERVERFAHREN

$$\begin{aligned} \eta_0 &= y_0 \\ \eta_{i+1} &= \eta_i + hf(t_{i+1}, \eta_{i+1}) \\ t_{i+1} &= t_i + h. \end{aligned}$$

Im Gegensatz zum expliziten Eulerverfahren muss man beim impliziten Eulerverfahren in jedem Schritt ein nicht lineares Gleichungssystem der Form $u = z + hf(t, u)$ mit bekannten Größen z und t lösen. Falls f (als Funktion von u bei festem t) differenzierbar ist, folgt aus dem Satz über implizite Funktionen (vgl. Mathematik für Maschinenbauer, Bauingenieure und Umwelttechniker I §IV.3), dass dieses Gleichungssystem für hinreichend kleines h eine Lösung in der Nähe von z hat. Diese Lösung kann mit wenigen Iterationen des Newtonverfahrens aus den Paragraphen II.3 und II.5 berechnet werden.

Ein drittes Verfahren erhalten wir durch folgende Überlegung: Für die exakte Lösung des AWP gilt

$$\begin{aligned} y(t+h) &= y(t) + \int_t^{t+h} y'(s)ds \\ &= y(t) + \int_t^{t+h} f(s, y(s))ds. \end{aligned}$$

Das Integral wird beim expliziten Eulerverfahren durch $hf(t, y(t))$ approximiert und beim impliziten Eulerverfahren durch $hf(t+h, y(t+h))$. Ebenso könnten wir es aber auch durch die Trapezregel

$$\int_t^{t+h} f(s, y(s))ds \approx \frac{h}{2}[f(t, y(t)) + f(t+h, y(t+h))]$$

aus Paragraph IV.1 annähern. Dies führt auf:

TRAPEZREGEL oder
VERFAHREN VON CRANK-NICOLSON

$$\begin{aligned} \eta_0 &= y_0 \\ \eta_{i+1} &= \eta_i + \frac{h}{2}[f(t_i, \eta_i) + f(t_{i+1}, \eta_{i+1})] \\ t_{i+1} &= t_i + h. \end{aligned}$$

Wie beim impliziten Eulerverfahren ist in jedem Schritt ein nicht lineares Gleichungssystem zu lösen. Wieder ist dies für hinreichend kleines

h möglich und kann mit wenigen Schritten des Newtonverfahrens aus den Paragraphen II.3 und II.5 geschehen.

Das folgende `Java`-Programm realisiert die drei beschriebenen Verfahren. Dabei steuert der Parameter `theta` die Verfahrenswahl:

$$\texttt{theta} = \begin{cases} 0 & \text{explizites Eulerverfahren,} \\ 1 & \text{implizites Eulerverfahren,} \\ \frac{1}{2} & \text{Crank-Nicolson-Verfahren.} \end{cases}$$

Die Methode `gaussElimination` löst ein lineares Gleichungssystem mit dem Gaußschen Eliminationsverfahren und ist in Paragraph I.2 wiedergegeben. `force` ist eine abstrakte Klasse, die eine differenzierbare Funktion mehrerer Veränderlicher realisiert und unter `f` bzw. `df` den Funktionswert bzw. die Jacobi-Matrix bereitstellt.

```
// linear single step method
    public void lssm( double theta ) throws LinearAlgebraException {
        double[] ff = new double[dim];
        double alpha = -theta*dt;
        double beta = (1.0 - theta)*dt;
        t = initialTime;
        copy(eta, x0);
        for( int step = 1; step <= steps; step++ ) {
            ff = add(eta, force.f(t, eta), 1.0, beta);
            t += dt;
            eta = ivpNewton(alpha, t, ff);
        }
    }    // end of lssm
// Newton method for the solution of equations of the form z + a*f(s,z) = g
    public double[] ivpNewton(double a, double s, double[] g)
        throws LinearAlgebraException {
        double[] y = new double[dim];            // solution
        double[] dy = new double[dim];           // increment
        double[][] jf = new double[dim][dim];    // Jacobian
        double resf;                             // norm of f
        double resy;                             // norm of dy
        copy(y, g);
        int iter = 0;
        b = add(g, force.f(s, y), 1.0, -a);
        accumulate(b, y, -1.0);
        resf = Math.sqrt( innerProduct(b, b)/dim );
        resy = 1.0;
        while( iter < MAXIT && resf > TOL && resy > TOL ) {
            jf = force.df(s, y);
            for( int i = 0; i < dim; i++ )
                for( int j = 0; j < dim; j++ ) {
                    a[i][j] = a*jf[i][j];
                    if( j == i ) a[i][j] += 1.0;
                }
            gaussElimination();
            copy(dy, x);
            resy = Math.sqrt( innerProduct(dy,dy)/dim );
            accumulate(y, dy, 1.0);
            b = add(g, force.f(s, y), 1.0, -a);
            accumulate(b, y, -1.0);
            resf = Math.sqrt( innerProduct(b, b)/dim );
```

```
            iter++;
        }
        return y;
    }    // end of ivpNewton
// copy vector v to vector u
    public void copy(double[] u, double[] v) {
        for( int i = 0; i < dim; i++ )
            u[i] = v[i];
    }    // end of copy
// add s times vector v to vector u
    public void accumulate(double[] u, double[] v, double s) {
        for( int i = 0; i <dim; i++ )
            u[i] += s*v[i];
    }    // end of accumulate
// return s*u+t*v
    public double[] add(double[] u, double[] v, double s, double t) {
        double[] w = new double[dim];
        for( int i = 0; i < dim; i++ )
            w[i] = s*u[i] + t*v[i];
        return w;
    }    // end of add
// inner product of two vectors
    public double innerProduct(double[] u, double[] v) {
        double prod = 0;
        for( int i = 0; i < dim; i++ )
            prod += u[i]*v[i];
        return prod;
    }    // end of inner product
```

VI.2. Einschrittverfahren

Bei den drei Verfahren des vorigen Abschnittes wird die Näherung η_{i+1} für $y(t_{i+1})$ ausschließlich durch die letzte Näherung η_i für $y(t_i)$ bestimmt. Daher spricht man von einem EINSCHRITTVERFAHREN (kurz ESV). Die allgemeine Definition eines ESV lautet:

EINSCHRITTVERFAHREN

$$\eta_0 = y_0$$
$$\eta_{i+1} = \eta_i + h\Phi(t_i, \eta_i, h; f)$$
$$t_{i+1} = t_i + h.$$

Die Funktion Φ heißt die VERFAHRENSFUNKTION des ESV.

Für das explizite Eulerverfahren ist offensichtlich

$$\Phi(t, y, h; f) = f(t, y).$$

Die Verfahrensfunktionen für das implizite Eulerverfahren und die Trapezregel sind komplizierter. Für hinreichend kleines h erhalten wir für das implizite Eulerverfahren

$$\Phi(t, y, h; f) = \frac{1}{h}\Big[[Id - hf(t+h, \cdot)]^{-1}y - y\Big]$$

und für die Trapezregel

$$\Phi(x, y, h; f) = \frac{1}{h}\Big[[Id - \frac{h}{2}f(t+h,\cdot)]^{-1}[y + \frac{h}{2}f(t,y)] - y\Big],$$

wobei g^{-1} die Umkehrfunktion der Funktion g bezeichnet. Man beachte, dass diese Darstellung der Verfahrensfunktion nur für die theoretische Analyse benötigt wird. Für die praktische Rechnung benutzt man die Darstellung der Verfahren aus dem vorigen Abschnitt.

VI.3. Verfahrensfehler und Ordnung

Die Qualität eines ESV wird durch den LOKALEN VERFAHRENSFEHLER gemessen. Dieser ist definiert durch

$$\tau(x, y, h) = \frac{1}{h}\{z(x+h) - y\} - \Phi(x, y, h; f) \quad , h > 0.$$

Dabei ist z die Lösung des AWP

$$z' = f(t, z(t)),$$
$$z(x) = y.$$

Ein ESV hat die ORDNUNG p, falls für alle hinreichend glatten Funktionen f der Ausdruck $h^{-p}\tau(x, y, h)$ für $h \to 0$ beschränkt bleibt. Hierfür schreibt man auch $\tau(x, y, h) = O(h^p)$.

Der lokale Verfahrensfehler beschreibt die Fehlerfortpflanzung in einem einzelnen Schritt eines ESV. Man kann aber zeigen, dass er auch den globalen Fehler, d.h. $|y(t_i) - \eta_i|$ für alle i, beschreibt:

Hat das ESV die Ordnung p, gilt für alle $t \neq t_0$ und alle $n \in \mathbb{N}^*$

$$\max_{0 \le i \le n} \left| y(t_0 + i\frac{t - t_0}{n}) - \eta(t_0 + i\frac{t - t_0}{n}, \frac{t - t_0}{n}) \right| \le c(t, f)\left(\frac{|t - t_0|}{n}\right)^p.$$

Für das explizite Eulerverfahren erhalten wir durch Taylorentwicklung (vgl. Mathematik für Maschinenbauer, Bauingenieure und Umwelttechniker I §VII.3.1)

$$\begin{aligned}\tau(x, y, h) &= \frac{1}{h}\{z(x+h) - y\} - f(x, y) \\ &= z'(x) + \frac{1}{2}hz''(x) + 0(h^2) - \underbrace{f(x, y)}_{=z'(x)} \\ &= \frac{1}{2}hz''(x) + 0(h^2).\end{aligned}$$

Also hat das explizite Eulerverfahren die Ordnung 1. Mit etwas mehr Aufwand kann man zeigen, dass das implizite Eulerverfahren ebenfalls die Ordnung 1 hat und dass das Verfahren von Crank-Nicolson die Ordnung 2 hat.

VI.4. Runge-Kutta-Verfahren

Viele der für die Praxis relevanten Einschrittverfahren gehören dieser Verfahrensklasse an. Die allgemeine Form lautet:

RUNGE-KUTTA-VERFAHREN

$$\eta_0 = y_0$$

$$\eta_{i,j} = \eta_i + h \sum_{k=1}^{r} a_{jk} f(t_i + c_k h, \eta_{i,k}), \quad 1 \leq j \leq r$$

$$\eta_{i+1} = \eta_i + h \sum_{k=1}^{r} b_k f(t_i + c_k h, \eta_{i,k})$$

$$t_{i+1} = t_i + h.$$

Dabei ist $0 \leq c_1 \leq \ldots \leq c_r \leq 1$. Die Zahl r heißt STUFE des Runge-Kutta-Verfahrens. Das Verfahren heißt EXPLIZIT, wenn $a_{jk} = 0$ ist für alle $k \geq j$; ansonsten heißt es IMPLIZIT. Implizite Verfahren sind aufwändiger als explizite Verfahren, da bei ihnen in jedem Schritt nicht lineare Gleichungssysteme gelöst werden müssen. Dafür haben sie bessere Stabilitätseigenschaften und sind insgesamt den expliziten Verfahren überlegen (s. Abschnitt VI.5).

Der Übersichtlichkeit halber fasst man die Zahlen c_k, a_{jk}, b_k in einem Schema der Form

$$\begin{array}{c|cccc} c_1 & a_{11} & a_{12} & \ldots & a_{1r} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_r & a_{r1} & a_{r2} & \ldots & a_{rr} \\ \hline & b_1 & b_2 & \ldots & b_r \end{array}$$

zusammen.

Die Verfahren aus Abschnitt VI.1 sind alle Runge-Kutta-Verfahren. Dem expliziten Eulerverfahren entspricht das Schema

$$\begin{array}{c|c} 0 & 0 \\ \hline & 1 \end{array} \qquad r = 1.$$

Dem impliziten Eulerverfahren entspricht das Schema

$$\begin{array}{c|c} 1 & 1 \\ \hline & 1 \end{array} \qquad r = 1.$$

Dem Verfahren von Crank-Nicolson entspricht schließlich das Schema

$$\begin{array}{c|cc} 0 & 0 & 0 \\ 1 & \frac{1}{2} & \frac{1}{2} \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array} \qquad r = 2.$$

Das sog. KLASSISCHE RUNGE-KUTTA-VERFAHREN ist schließlich gegeben durch das Schema

$$\begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ \hline & \frac{1}{6} & \frac{2}{6} & \frac{2}{6} & \frac{1}{6} \end{array} \qquad r = 4.$$

Die entsprechende Verfahrensvorschrift lautet:

$$\begin{aligned} \eta_0 &= y_0 \\ \eta_{i,1} &= \eta_i \\ \eta_{i,2} &= \eta_i + \frac{h}{2} f(t_i, \eta_{i,1}) \\ \eta_{i,3} &= \eta_i + \frac{h}{2} f(t_i + \frac{h}{2}, \eta_{i,2}) \\ \eta_{i,4} &= \eta_i + h f(t_i + \frac{h}{2}, \eta_{i,3}) \\ \eta_{i+1} &= \eta_i + \frac{h}{6} \{ f(t_i, \eta_{i,1}) + 2 f(t_i + \frac{h}{2}, \eta_{i,2}) \\ &\qquad + 2 f(t_i + \frac{h}{2}, \eta_{i,3}) + f(t_i + h, \eta_{i,4}) \} \\ t_{i+1} &= t_i + h. \end{aligned}$$

Es hat die Stufe 4 und die Ordnung 4. Als explizites Verfahren hat es aber keine besonders guten Stabilitätseigenschaften.

Wegen ihrer hohen Ordnung und guten Stabilitätseigenschaften sind die STARK DIAGONAL IMPLIZITEN RUNGE-KUTTA-VERFAHREN (kurz SDIRK-VERFAHREN) für die praktische Rechnung besonders gut geeignet. Die einfachsten Verfahren dieser Klasse haben die Stufe 2 und die Ordnung 3 bzw. die Stufe 5 und die Ordnung 4. Sie sind bestimmt durch die Schemata

$$\begin{array}{c|cc} \frac{3+\sqrt{3}}{6} & \frac{3+\sqrt{3}}{6} & 0 \\ \frac{3-\sqrt{3}}{6} & \frac{-\sqrt{3}}{3} & \frac{3+\sqrt{3}}{6} \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array} \qquad \text{SDIRK2.}$$

und

$$\begin{array}{c|ccccc}
\frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 & 0 \\
\frac{3}{4} & \frac{1}{2} & \frac{1}{4} & 0 & 0 & 0 \\
\frac{11}{20} & \frac{17}{50} & -\frac{1}{25} & \frac{1}{4} & 0 & 0 \\
\frac{1}{2} & \frac{371}{1360} & -\frac{137}{2720} & \frac{15}{544} & \frac{1}{4} & 0 \\
1 & \frac{25}{24} & -\frac{49}{48} & \frac{125}{16} & -\frac{85}{12} & \frac{1}{4} \\
\hline
 & \frac{25}{24} & -\frac{49}{48} & \frac{125}{16} & -\frac{85}{12} & \frac{1}{4}
\end{array} \qquad \text{SDIRK5.}$$

Damit lautet die Verfahrensvorschrift z. B. für das SDIRK2-Verfahren

$$\begin{aligned}
\eta_0 &= y_0 \\
\eta_{i,1} &= \eta_i + \frac{3+\sqrt{3}}{6} h f(t_i + \frac{3+\sqrt{3}}{6} h, \eta_{i,1}) \\
\eta_{i,2} &= \eta_i - \frac{\sqrt{3}}{3} h f(t_i + \frac{3+\sqrt{3}}{6} h, \eta_{i,1}) \\
&\quad + \frac{3+\sqrt{3}}{6} h f(t_i + \frac{3-\sqrt{3}}{6} h, \eta_{i,2}) \\
\eta_{i+1} &= \eta_i + \frac{h}{2} \left\{ f(t_i + \frac{3+\sqrt{3}}{6} h, \eta_{i,1}) + f(t_i + \frac{3-\sqrt{3}}{6} h, \eta_{i,2}) \right\} \\
t_{i+1} &= t_i + h.
\end{aligned}$$

Das folgende `Java`-Programm realisiert ein allgemeines SDIRK-Verfahren. Dabei stellt die Klasse `RKParameters` die nötigen Parameter bereit. Insbesondere liefert `rk.d` die als konstant angenommene Diagonale der Matrix $(a_{ij})_{1\le i,j\le r}$. Die Methoden `accumulate`, `copy` und `ivpNewton` sind bereits oben wiedergegeben. `force` hat die gleiche Bedeutung wie oben.

```
// Runge-Kutta method
    public void sdirk( int type ) throws LinearAlgebraException {
        RKParameters rk = new RKParameters(type);
        // auxiliary quantities for runge-kutta levels
            double[][] rkf = new double[rk.l][dim];     // function values
            double[][] rketa = new double[rk.l][dim];   // intermediate eta's
            double[] tt = new double[rk.l];             // intermediate times
        // end of runge-kutta declarations
        double[] ff = new double[dim];
        double alpha = -rk.d*dt;
        t = initialTime;
        copy(eta, x0);
        for( int step = 1; step <= steps; step++ ) {
            for( int j = 0; j < rk.l; j++ )
                tt[j] = t + rk.c[j]*dt;
            int jj = 0;
            for( int j = 0; j < rk.l; j++ ) {
                copy(ff, eta);
                for( int k = 0; k < j; k++ ) {
                    accumulate(ff, rkf[k], rk.a[jj]*dt);
                    jj++;
```

```
            }
            rketa[j] = ivpNewton(alpha, tt[j], ff);
            rkf[j] = force.f(tt[j], rketa[j]);
        }
        t += dt;
        for( int j = 0; j < rk.l; j++ )
            accumulate(eta, rkf[j], rk.b[j]*dt);
    }
}   // end of sdirk
```

VI.5. Stabilität

Das explizite und das implizite Eulerverfahren haben beide die Ordnung 1; für $h \to 0$ verhält sich der Fehler bei beiden Verfahren gleich. Das implizite Verfahren ist aber aufwändiger als das explizite Verfahren, da in jedem Schritt ein nicht lineares Gleichungssystem gelöst werden muss. Macht sich dieser erhöhte Aufwand denn überhaupt bezahlt?

Um diese Frage zu beantworten, betrachten wir das AWP

$$\begin{aligned} y' &= -\lambda y \\ y(0) &= 1 \end{aligned}$$

mit $\lambda \gg 1$. Die exakte Lösung ist $y(t) = e^{-\lambda t}$ und klingt sehr schnell ab.

Wir wenden auf dieses AWP zuerst das explizite Eulerverfahren mit $\eta_0 = y_0 = 1$ an und erhalten

$$\begin{aligned} \eta_{i+1} &= \eta_i - h\lambda\eta_i \\ \Longrightarrow \quad \eta_i &= (1 - h\lambda)^i \quad \text{für alle } i. \end{aligned}$$

Die η_i konvergieren für $i \to \infty$ offensichtlich genau dann gegen Null, wenn gilt

$$|1 - h\lambda| < 1 \qquad \Longleftrightarrow \qquad h < \frac{2}{\lambda}.$$

Also gilt:

> Die numerische Lösung des expliziten Verfahrens hat nur dann das gleiche qualitative Verhalten wie die exakte Lösung, wenn die Schrittweite h hinreichend klein ist.

Wir betrachten nun das implizite Eulerverfahren mit $\eta_0 = y_0 = 1$ und erhalten

$$\begin{aligned} \eta_{i+1} &= \eta_i - h\lambda\eta_{i+1} \\ \Longrightarrow \qquad \eta_i &= \left(\frac{1}{1 + h\lambda}\right)^i \quad \text{für alle } i. \end{aligned}$$

Wegen $\lambda > 0$ konvergieren die η_i für $i \to \infty$ jetzt für jede Schrittweite h gegen Null, d.h.:

> Das implizite Verfahren liefert für jede Schrittweite eine numerische Lösung, die das gleiche qualitative Verhalten hat wie die exakte Lösung.

Der Mehraufwand für das implizite Verfahren macht sich also bezahlt!

Dieses Phänomen nennt man STABILITÄT. Um es genauer zu beschreiben, wenden wir ein beliebiges Runge-Kutta-Verfahren auf das obige AWP an. Mit ein wenig Rechnung sieht man dann, dass die numerische Lösung die Form hat

$$\eta_i = g(h\lambda)^i$$

mit

$$g(z) = 1 + zb^T(I - zA)^{-1}e$$

und

$$b = \begin{pmatrix} b_1 \\ \vdots \\ b_r \end{pmatrix}, \; A = \begin{pmatrix} a_{11} & \dots & a_{1r} \\ \vdots & & \vdots \\ a_{r1} & \dots & a_{rr} \end{pmatrix}, \; e = \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix}.$$

Die Menge

$$\boxed{\begin{aligned} S = \{\mu \in \mathbb{C} : I - \mu A &\text{ ist regulär und} \\ &|g(\mu)| = |1 + \mu b^T(I - \mu A)^{-1}e| \le 1\} \end{aligned}}$$

heißt das STABILITÄTSGEBIET des Verfahrens.

BEISPIEL VI.1. Für das explizite Eulerverfahren ist

$$g(z) = z + 1$$

und damit

$$S = \{\mu \in \mathbb{C} : |\mu + 1| \le 1\}.$$

Für das implizite Eulerverfahren ist

$$g(z) = \frac{1}{1 - z}$$

und damit

$$\begin{aligned} S &= \{\mu \in \mathbb{C} : \frac{1}{|1 - \mu|} \le 1\} \\ &= \{\mu \in \mathbb{C} : |1 - \mu| \ge 1\}. \end{aligned}$$

Für das Verfahren von Crank-Nicolson erhalten wir schließlich

$$g(z) = \frac{2 + z}{2 - z}$$

und

$$S = \{\mu \in \mathbb{C} : |\frac{2+\mu}{2-\mu}| \leq 1\}$$
$$= \{\mu \in \mathbb{C} : \operatorname{Re}\mu \leq 0\}.$$

Abbildung VI.1 zeigt die Stabilitätsgebiete dieser drei Verfahren.

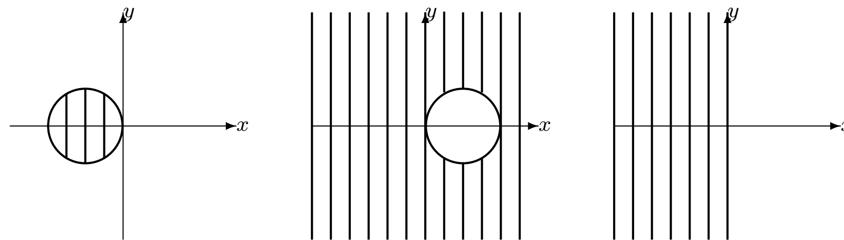


ABBILDUNG VI.1. Stabilitätsgebiete des expliziten und impliziten Eulerverfahrens und des Crank-Nicolson-Verfahrens (v.l.n.r.)

Je größer das Stabilitätsgebiet ist, um so größere Schrittweiten können verwendet werden. Im Idealfall ist $S \supset \{z \in \mathbb{C} : \operatorname{Re} z \leq 0\}$. Verfahren, die dieses Kriterium erfüllen, nennt man A-STABIL. Bei solchen Verfahren erhält man bei obigem AWP für jede Schrittweite eine qualitativ richtige numerische Lösung. Das implizite Eulerverfahren, das Crank-Nicolson-Verfahren und die in Abschnitt VI.4 genannten SDIRK-Verfahren sind A-stabil.

Bei expliziten Runge-Kutta-Verfahren ist die Funktion g ein Polynom. Da für jedes nicht konstante Polynom p gilt

$$\lim_{x \to -\infty} |p(x)| = \infty,$$

können explizite Runge-Kutta-Verfahren niemals A-stabil sein. Für obiges AWP liefern sie nur dann eine qualitativ richtige numerische Lösung, wenn die Schrittweite h hinreichend klein ist.

BEISPIEL VI.2. Wir führen jeweils 100 Schritte der beiden Eulerverfahren und des Crank-Nicolson-Verfahrens für das AWP

$$\mathbf{y}' = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \mathbf{y}$$
$$\mathbf{y}(0) = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

mit $h = 0.13$ aus. Die exakte Lösung lautet $\mathbf{y}(t) = (\cos t, \sin t)$ mit $0 \leq t \leq 13$. Abbildung VI.2 zeigt die entsprechenden Lösungskurven. Wir erkennen, dass das explizite Eulerverfahren explodiert. Dies ist eine Folge seiner fehlenden Stabilität. Das implizite Eulerverfahren dagegen dämpft die Lösung zu stark. Das Crank-Nicolson-Verfahren schließlich

liefert eine Lösungskurve, die auf der exakten Lösungskurve, dem Kreis um den Ursprung mit Radius 1, liegt.

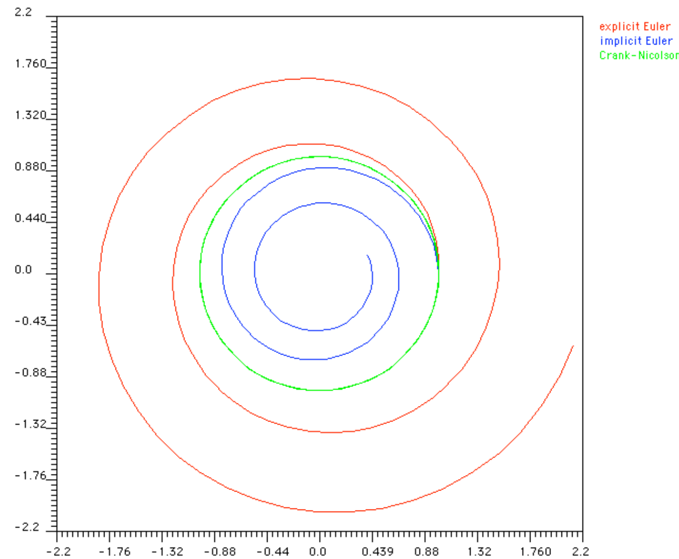


Abbildung VI.2. Lösungskurven der beiden Eulerverfahren und des Crank-Nicolson-Verfahrens für das AWP aus Beispiel VI.2

Beispiel VI.3. Wir führen jeweils 1000 Schritte des impliziten Eulerverfahrens, des Crank-Nicolson-Verfahrens und des klassischen Runge-Kutta-Verfahrens für das AWP

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} 4x - 8xy \\ -0.3y + 0.6xy \end{pmatrix}$$
$$\begin{pmatrix} x(0) \\ y(0) \end{pmatrix} = \begin{pmatrix} 0.9 \\ 0.1 \end{pmatrix}$$

mit $h = 0.1$ aus. Dieses AWP beschreibt ein Räuber-Beute-Modell. Die exakte Lösung ist eine geschlossene Kurve. Abbildung VI.3 zeigt, dass das implizite Eulerverfahren die Lösung zu stark dämpft. Die beiden anderen Verfahren liefern qualitativ gute Ergebnisse, wobei das Crank-Nicolson-Verfahren etwas besser ist.

VI.6. Steife Differentialgleichungen

Eine gDgl nennt man steif, wenn ihre Lösung Komponenten mit sehr unterschiedlichem Abklingverhalten (d.h. stark unterschiedlichen Zeitskalen) enthält. Derartige Probleme treten z.B. bei Diffusions- und Wärmeleitungsvorgängen oder bei chemischen Reaktionen auf. Für solche gDgl muss man Verfahren mit sehr gutem Stabilitätsverhalten verwenden, da ansonsten schnell abklingende Lösungskomponenten das

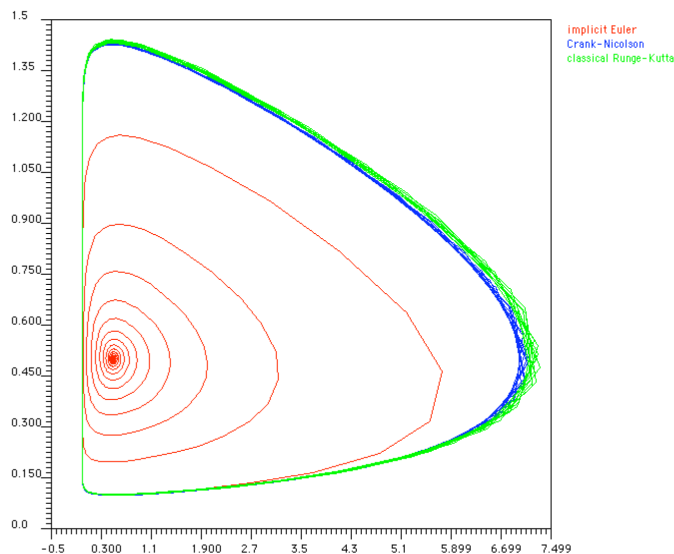


Abbildung VI.3. Lösungskurven des impliziten Eulerverfahrens, des Crank-Nicolson-Verfahrens und des klassischen Runge-Kutta-Verfahrens für das AWP aus Beispiel VI.3

Langzeitverhalten der numerischen Lösung dominieren und exorbitant kleine Schrittweiten erfordern oder bei zu großen Schrittweiten explodieren.

Zur Verdeutlichung der Problematik betrachten wir folgendes Beispiel.

Beispiel VI.4. Wir betrachten das AWP

$$\mathbf{y}' + \begin{pmatrix} 500 & 499 \\ 499 & 500 \end{pmatrix} \mathbf{y} = 0,$$

$$\mathbf{y}(0) = \begin{pmatrix} 2 \\ 0 \end{pmatrix}.$$

Die Matrix

$$\begin{pmatrix} 500 & 499 \\ 499 & 500 \end{pmatrix}$$

hat die Eigenwerte 999 und 1 mit den zugehörigen Eigenvektoren

$$\begin{pmatrix} 1 \\ 1 \end{pmatrix} \quad \text{bzw.} \quad \begin{pmatrix} 1 \\ -1 \end{pmatrix}.$$

Daher lautet die exakte Lösung des AWP

$$\mathbf{y}(t) = e^{-t} \begin{pmatrix} 1 \\ -1 \end{pmatrix} + e^{-999t} \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

Nach der Zeit $t = \frac{1}{100}$ ist die Komponente $e^{-999t}(1,1)^T$ um den Faktor $4.5 \cdot 10^{-5}$ abgeklungen. Wenn wir das explizite Eulerverfahren auf

dieses AWP anwenden, müssen wir aber dennoch für *alle* Zeiten eine Schrittweite kleiner als $\frac{1}{500}$ wählen, um eine qualitativ korrekte Lösung zu erhalten. Andernfalls explodiert die Komponente zum Eigenvektor $(1,1)^T$. Dies gilt auch, wenn wir einige Schritte mit sehr kleiner Schrittweite rechnen und dann auf eine Schrittweite größer als $\frac{1}{500}$ umschalten. Bei stabilen impliziten Verfahren, wie z.B. impliziter Euler, Crank-Nicolson, SDIRK, erhalten wir dagegen auch bei großen Schrittweiten eine qualitativ korrekte Lösung.

VI.7. Schrittweitensteuerung

Ziel ist es, ausgehend vom Punkt t_0 in möglichst wenigen Schritten eine Näherung der Lösung $y(t)$ im Punkt t mit einer gegebenen Genauigkeit ε zu berechnen. Typischerweise ist $\varepsilon = K \cdot \mathrm{eps}$, wobei eps die Maschinengenauigkeit und K eine obere Schranke für $\|y(t)\|$ ist.

Es gibt zwei Strategien zur Schrittweitensteuerung:

- Vergleich verschiedener Schrittweiten,
- Vergleich verschiedener Verfahren.

Zur Darstellung der ersten Strategie bezeichne mit $\eta(t;h)$ die mit einem ESV der Ordnung p und Schrittweite h gewonnene Näherung für $y(t)$. Dann ist unter Vernachlässigung von Termen höherer Ordnung

$$\begin{aligned}\eta(t;h) &= y(t) + e_p(t)h^p \\ \eta(t;\frac{h}{2}) &= y(t) + e_p(t)\Big[\frac{h}{2}\Big]^p\end{aligned}$$

und somit

$$e_p(t) = \frac{1}{h^p(1-2^{-p})}\Big\{\eta(t;h) - \eta(t;\frac{h}{2})\Big\}.$$

Also gilt bei Fortführen des Verfahrens mit der neuen Schrittweite H unter Vernachlässigung von Termen höherer Ordnung

$$\begin{aligned}e(t+H;H) &= \eta(t+H;H) - y(t+H) \\ &= e_p(t+H)H^p \\ &= e_p(t)H^p \\ &= \frac{1}{1-2^{-p}}\Big\{\eta(t;h) - \eta(t;\frac{h}{2})\Big\}(\frac{H}{h})^p.\end{aligned}$$

Mithin erhalten wir als optimale Schrittweite

$$\boxed{H = h\Big\{(1-2^{-p})\frac{\varepsilon}{\|\eta(t;h)-\eta(t;\frac{h}{2})\|_2}\Big\}^{\frac{1}{p}}.}$$

Dies führt auf folgenden Algorithmus zur Schrittweitensteuerung:

Algorithmus VI.5. Schrittweitensteuerung durch Halbieren.

(0) *Gegeben* t_0, y_0 *und Startschrittweite* h.
(1) *Berechne*

$$\eta(t_0 + h; h),$$

$$\eta(t_0 + h; \frac{h}{2}),$$

$$H = h\Big\{(1 - 2^{-p})\frac{\varepsilon}{\|\eta(t_0 + h; h) - \eta(t_0 + h; \frac{h}{2})\|_2}\Big\}^{1/p}.$$

(2) *Falls* $H \le \frac{h}{4}$ *ist, setze* $h = 2H$ *und gehe zu 1 zurück.*
(3) *Setze*

$$t_0 = t_0 + h,$$

$$y_0 = \eta(t_0 + h; \frac{h}{2}),$$

$$h = 2H$$

und gehe zu 1 zurück.

Das folgende `Java`-Programm realisiert Algorithmus VI.5 für die beiden Eulerverfahren und das Crank-Nicolson-Verfahren. Dabei steuert der Parameter `theta` wieder die Verfahrenswahl:

$$\texttt{theta} = \begin{cases} 0 & \text{explizites Eulerverfahren,} \\ 1 & \text{implizites Eulerverfahren,} \\ \frac{1}{2} & \text{Crank-Nicolson-Verfahren.} \end{cases}$$

Die Methode `lssm( eta, theta, i )` führt ausgehend von der Näherung `eta` i Schritte des Verfahrens mit dem Parameter `theta` mit fester Schrittweite aus und speichert das Ergebnis auf der Variablen `eta`. Die Größe `order` gibt die Ordnung des Verfahrens an, also 2 für `theta` $= \frac{1}{2}$ und 1 sonst. Die nicht aufgeführten Methoden sind identisch mit den gleichnamigen Methoden bei den zuvor aufgeführten Verfahren ohne Schrittweitensteuerung.

```
// linear single step method with step-size control
    public void alssm( double theta ) throws LinearAlgebraException {
        double estErr;
        int tries = 0;
        double twop = Math.pow(2.0, order);
        step = 0;
        t = initialTime;
        dt = FIRST*(finalTime - initialTime)/steps;
        while( step < steps && t < finalTime ) {
            copy( eta, x0 );
            copy( etaa, eta );
            dt /= 2.0;
            lssm( eta, theta, 2 );
            dt *= 2.0;
            lssm( etaa, theta, 1 );
            accumulate( etaa, eta, -1.0 );
            estErr = norm( etaa )*(twop - 1.0)/twop;
            if( estErr > tolerance && tries < MAXTRIES ) {
```

```
                tries++;
                dt *= Math.min( DECREMENT,
                                Math.exp((Math.log(tolerance) -
                                Math.log(estErr))/order) );
                dtmax = Math.max( dtmax, dt );
                dtmin = Math.min( dtmin, dt );
            }
            else {
                tries = 0;
                t += dt;
                step++;
                dt *= Math.min( INCREMENT,
                                Math.exp((Math.log(tolerance) -
                                Math.log(estErr))/order) );
                dt = Math.min( dt, finalTime - t );
            }
        }
    }   // end of alssm
```

Zur Darstellung der zweiten Strategie bezeichnen wir mit $\tilde{\eta}(t;h)$ die mit einem Verfahren der Ordnung q, $q > p$, gewonnenen Näherungen. Dann ist bis auf Terme höherer Ordnung

$$\eta(t;h) = y(t) + e_p(t)h^p$$
$$\tilde{\eta}(t;h) = y(t) + \tilde{e}_q(t)h^q$$

und somit

$$e_p(t) = h^{-p}\Big\{\eta(t;h) - \tilde{\eta}(t;h)\Big\}.$$

Mithin ist für das Verfahren der Ordnung p die optimale Schrittweite gegeben durch

$$\boxed{H = h\Big\{\frac{\varepsilon}{\|\eta(t;h) - \tilde{\eta}(t;h)\|_2}\Big\}^{\frac{1}{p}}.}$$

Dies führt zu folgendem Algorithmus zur Schrittweitensteuerung:

Algorithmus VI.6. Schrittweitensteuerung durch Ordnungsvergleich.

(0) *Gegeben t_0, y_0 und Startschrittweite h.*
(1) *Berechne*

$$\eta(t_0 + h; h),$$
$$\tilde{\eta}(t_0 + h; h),$$
$$H = h\Big\{\frac{\varepsilon}{\|\eta(t_0 + h; h) - \tilde{\eta}(t_0 + h; h)\|_2}\Big\}^{1/p}.$$

(2) *Falls $H \leq \frac{h}{2}$ ist, gehe zu 1 zurück und setze $h = H$.*
(3) *Setze*

$$t_0 = t_0 + h,$$
$$y_0 = \eta(t_0 + h; h),$$

$$h = H$$

und gehe zu 1 zurück.

Bei den Runge-Kutta-Fehlberg-Verfahren kann man die Näherung $\tilde{\eta}$ ohne großen zusätzlichen Aufwand berechnen.

Beispiel VI.7. Das einfachste Runge-Kutta-Fehlberg-Verfahren ist gegeben durch

$$\begin{aligned}
\eta_0 &= y_0, \\
\tilde{\eta}_0 &= y_0 \\
\eta_{0,1} &= \eta_0 \\
\eta_{0,2} &= \eta_0 + \frac{h}{4} f(t_0, \eta_{0,1}) \\
\eta_{0,3} &= \eta_0 - \frac{189}{800} h f(t_0, \eta_{0,1}) + \frac{729}{800} h f(t_0 + \frac{1}{4} h, \eta_{0,2}) \\
\eta_{0,4} &= \eta_0 + \frac{214}{891} h f(t_0, \eta_{0,1}) + \frac{1}{33} h f(t_0 + \frac{1}{4} h, \eta_{0,2}) \\
&\quad + \frac{650}{891} h f(t_0 + \frac{27}{40} h, \eta_{0,3}) \\
\eta_1 &= \eta_{0,4} \\
\tilde{\eta}_1 &= \eta_0 + \frac{533}{2106} h f(t_0, \eta_{0,1}) + \frac{800}{1053} h f(t_0 + \frac{h}{4}, \eta_{0,2}) \\
&\quad - \frac{1}{78} h f(t_0 + h, \eta_{0,4}).
\end{aligned}$$

Dann ist η_1 eine Approximation zweiter Ordnung ($p = 2$) und $\tilde{\eta}_1$ eine solche dritter Ordnung ($q = 3$). Man beachte, daß die Berechnung von $\tilde{\eta}$ nur eine zusätzliche Funktionsauswertung erfordert.

Das folgende `Java`-Programm realisiert Algorithmus VI.6 für Runge-Kutta-Fehlberg-Verfahren. Dabei stellt die Klasse `RKParameters` wieder die nötigen Parameter bereit; `rk.d` liefert die als konstant angenommene Diagonale der Matrix $(a_{ij})_{1 \le i,j \le r}$. Die Größe `order` gibt wieder die Ordnung des Verfahrens an. Die nicht aufgeführten Methoden sind identisch mit den gleichnamigen Methoden bei den zuvor aufgeführten Verfahren ohne Schrittweitensteuerung.

```
// Runge-Kutta-Fehlberg method with step-size control
    public void arkf( int type ) throws LinearAlgebraException {
        rk = new RKParameters(type);
        rkf = new double[rk.l][dim];
        rketa = new double[rk.l][dim];
        rktt = new double[rk.l];
        alpha = -rk.d*dt;
        double estErr;
        int tries = 0;
        step = 0;
        t = initialTime;
        dt = FIRST*(finalTime - initialTime)/steps;
```

```
        copy( eta, x0 );
        while( step < steps && t < finalTime ) {
            copy( etaa, eta );
            for( int j = 0; j < rk.l; j++ )
                rktt[j] = t + rk.c[j]*dt;
            int jj = 0;
            for( int j = 0; j < rk.l; j++ ) {
                copy(ff, eta);
                for( int k = 0; k < j; k++ ) {
                    accumulate(ff, rkf[k], rk.a[jj]*dt);
                    jj++;
                }
                rketa[j] = ivpNewton(alpha, rktt[j], ff);
                rkf[j] = force.f(rktt[j], rketa[j]);
            }
            for( int j = 0; j < rk.l; j++ )
                accumulate(etaa, rkf[j], rk.b[j]*dt);
            copy( eta, rketa[rk.l-1] );
            accumulate( etaa, eta, -1.0 );
            estErr = norm( etaa );
            if( estErr > tolerance && tries < MAXTRIES ) {
                tries++;
                dt *= Math.min( DECREMENT,
                                Math.exp((Math.log(tolerance) -
                                Math.log(estErr))/order) );
            }
            else {
                tries = 0;
                t += dt;
                step++;
                dt *= Math.min( INCREMENT,
                                Math.exp( (Math.log(tolerance) -
                                Math.log(estErr))/order) );
                dt = Math.min( dt, finalTime - t );
            }
        }
    }   // end of arkf
```

Beispiel VI.8. Wir wenden die beiden Euler-Verfahren und das Crank-Nicolson-Verfahren mit der Schrittweitensteuerung von Algorithmus VI.5 und die Runge-Kutta-Fehlberg-Verfahren der Ordnung 2 (RKF2), 3 (RKF3) und 4 (RKF4) mit der Schrittweitensteuerung von Algorithmus VI.6 auf das AWP aus Beispiel VI.2 (S. 93) an. Alle Verfahren liefern die exakte Lösungskurve, den Kreis um den Ursprung mit Radius 1. Wie aus Tabelle VI.1 ersichtlich ist, unterscheiden sie sich aber deutlich bei der erreichten Endzeit, der maximalen Schrittweite und der Rechenzeit. Dabei bedeutet

- T die erreichte Endzeit,
- N die Zahl der Schritte,
- $h_{\max}$ die maximale Schrittweite,
- msec die Rechenzeit in Millisekunden auf einem MacIntosh G4 Powerbook.

Die Anfangszeit und die zu erreichende Endzeit ist jeweils 0 bzw. 13, die erste Schrittweite ist immer 0.013 und die Toleranz ist stets $\varepsilon = 10^{-6}$.

Tabelle VI.1. Verfahren mit Schrittweitensteuerung für das AWP aus Beispiel VI.2 (S. 93)

	T	N	$h_{\max}$	msec
expliziter Euler	1.299	1000	0.00615	247
impliziter Euler	1.317	1000	0.00615	525
Crank-Nicolson	10.987	1000	0.04590	423
RKF2	0.441	1000	0.00176	312
RKF3	13.000	400	0.13323	199
RKF4	13.000	151	0.35478	67

Beispiel VI.9. Wir wenden die Verfahren aus dem vorigen Beispiel auf das Räuber-Beute-Modell aus Beispiel VI.3 (S. 94) an. Anfangs- und Endzeit sind jeweils 0 bzw. 100, die erste Schrittweite ist immer 0.1 und die Toleranz ist stets $\varepsilon = 10^{-6}$. Alle Verfahren liefern die gleiche Lösungskurve und benutzen die minimale Schrittweite 0.0002. Die entsprechenden Ergebnisse sind in Tabelle VI.2 wiedergegeben.

Tabelle VI.2. Verfahren mit Schrittweitensteuerung für das AWP aus Beispiel VI.3 (S. 94)

	T	N	$h_{\max}$	msec
expliziter Euler	0.959	5000	0.00390	1148
impliziter Euler	0.889	5000	0.00451	2298
Crank-Nicolson	43.938	5000	0.24362	2299
RKF2	0.706	5000	0.00095	1615
RKF3	100.000	4011	0.60405	1615
RKF4	100.000	1593	1.45927	869

VI.8. Mehrschrittverfahren

Bei allen bisherigen Verfahren erfordert die Berechnung einer neuen Näherung η_{i+1} nur die Kenntnis der letzten Näherung η_i. Bei Mehrschrittverfahren benötigt man dagegen die Kenntnis weiterer älterer Näherungen $\eta_{i-1}, \ldots, \eta_{i-r}$.
Die wichtigsten derartigen Verfahren sind die Verfahren von Adams-Bashforth und Adams-Moulton und die BDF-Formeln (Backward Difference Formulae). Wir beschränken uns auf die BDF-Formeln, da sie im Gegensatz zu den Adams-Verfahren gute Stabilitätseigenschaften haben und auch zur Lösung steifer Differentialgleichungen geeignet sind.

Die Idee der BDF-Formeln besteht darin, die Ableitung im Punkt t_{i+1} durch einen rückwärtigen Differenzenquotienten zu den Punkten $t_{i+1}, t_i, \ldots, t_{i-r}$ zu approximieren und diese Approximation mit der

rechten Seite $f(t_{i+1}, \eta_{i+1})$ gleichzusetzen. Für die Werte $r = 1, 2, 3$ und äquidistante Punkte $t_i = t_0 + ih$ führt dieser Ansatz zu folgenden Verfahren:

BDF 2:

$$\eta_{i+1} - \frac{4}{3}\eta_i + \frac{1}{3}\eta_{i-1} = \frac{2}{3}hf(t_{i+1}, \eta_{i+1})$$

BDF 3:

$$\eta_{i+1} - \frac{18}{11}\eta_i + \frac{9}{11}\eta_{i-1} - \frac{2}{11}\eta_{i-2} = \frac{6}{11}hf(t_{i+1}, \eta_{i+1})$$

BDF 4:

$$\eta_{i+1} - \frac{48}{25}\eta_i + \frac{36}{25}\eta_{i-1} - \frac{16}{25}\eta_{i-2} + \frac{3}{25}\eta_{i-3} = \frac{12}{25}hf(t_{i+1}, \eta_{i+1})$$

Der Fehler dieser Verfahren ist von der Ordnung 2, 3 bzw. 4. Die ersten zwei, drei bzw. vier η-Werte müssen mit einem Einschrittverfahren entsprechender Ordnung berechnet werden (sog. Anlaufrechnung).

Eine Schrittweitensteuerung ist bei Mehrschrittverfahren möglich aber wesentlich aufwändiger als bei Einschrittverfahren, da bei jeder Schrittweitenänderung entweder eine neue Anlaufrechnung durchgeführt oder alte Näherungslösungen interpoliert werden müssen.

Zusammenfassung

I Lineare Gleichungssysteme

1. Lineare Gleichungssysteme und Matrizen
   Lineare Gleichungssysteme (LGS); Koeffizienten; Absolutglieder; Koeffizientenmatrix; rechte Seite; homogene und inhomogene LGS; LGS mit keiner, genau einer, unendlich vielen Lösungen
2. Das Gaußsche Eliminationsverfahren
   Idee des Verfahrens; Pivotsuche; Zeilentausch; Elimination; Rücklösung; Pivotsuche auch bei exakter Arithmetik erforderlich
3. Aufwand des Gaußschen Eliminationsverfahrens
   Aufwand $O(n^3)$ für Elimination; Aufwand $O(n^2)$ für Rücklösung; Aufwand $O(n^n)$ für Cramersche Regel
4. Die L-R-Zerlegung
   Idee des Verfahrens; Zerlegungsteil; Lösungsteil; Aufwand $O(n^3)$ für Zerlegungsteil; Aufwand $O(n^2)$ für Lösungsteil
5. Symmetrische positiv definite Matrizen
   Symmetrische Matrizen; symmetrische positiv definite (s.p.d.) Matrizen; A s.p.d. $\iff$ alle Eigenwerte sind reell und positiv $\iff$ die Determinanten aller Hauptmatrizen sind positiv
6. Das Cholesky-Verfahren
   Idee des Verfahrens; Berechnung der Zerlegung; Lösen eines LGS bei bekannter Cholesky-Zerlegung; Aufwand
7. Große Gleichungssysteme
   Auffüllen; Bandmatrizen; Bandbreite; Aufwand $O(nb^2)$ für Elimination und $O(nb)$ für Rücklösung; Gaußsches Eliminationsverfahren und Verwandte ungeeignet für sehr große LGS; Ausweg: vorkonditionierte konjugierte Gradienten- und Mehrgitterverfahren
8. Störungsrechnung
   Fehler; Residuum; Matrixnorm; Kondition; Abschätzung des Fehlers durch das Residuum; Bedeutung einer großen Kondition; Matrixnorm und Kondition bzgl. anderer Vektornormen

II Nicht lineare Gleichungssysteme

1. Das Bisektionsverfahren
   Idee des Verfahrens; Abschätzung der Zahl der benötigten Iterationen
2. Das Sekantenverfahren
   Idee des Verfahrens; Abbruch bei waagerechter Sekante
3. Das Newtonverfahren für Funktionen einer Variablen
   Idee des Verfahrens; Abbruch bei waagerechter Tangente

4. Eigenschaften des Newtonverfahrens
   Kessel; Divergenz bei ungünstigem Startwert; quadratische Konvergenz bei Startwert hinreichend nahe bei Nullstelle; quadratische Konvergenz = Verdoppelung der Zahl der korrekten Nachkommastellen mit jeder Iteration; Verfahren von Heron; divisionsfreie Berechnung von Reziproken
5. Das Newtonverfahren für Funktionen mehrerer Variablen
   Idee des Verfahrens; Lösen eines LGS in jeder Iteration; quadratische Konvergenz
6. Dämpfung
   Notwendigkeit der Dämpfung; Durchführung der Dämpfung; Erhalt der quadratischen Konvergenz

III Interpolation

1. Lagrange-Interpolation
   Problemstellung; es gibt höchstens ein Interpolationspolynom; Lagrangesche Darstellung des Interpolationspolynomes
2. Die Newtonsche Darstellung des Lagrangeschen Interpolationspolynomes
   Nachteile der Lagrangeschen Darstellung; dividierte Differenzen; Newtonsche Darstellung des Interpolationspolynomes; Aufwand $O(n^2)$ zur Berechnung der dividierten Differenzen; Aufwand $O(n)$ zur Auswertung in einem Punkt
3. Genauigkeit der Lagrange-Interpolation
   Fehlerabschätzung; Satz von Faber; zu hoher Polynomgrad kontraproduktiv
4. Hermite-Interpolation
   Problemstellung; dividierte Differenzen bei mehrfachen Knoten; Darstellung des Hermiteschen Interpolationspolynomes mittels dividierter Differenzen
5. Kubische Splineinterpolation
   Kubische Splines; Interpolationsaufgabe; Lösung der Interpolationsaufgabe; Fehlerabschätzung; Vorteile gegenüber Lagrange-Interpolation
6. Bézier-Darstellung von Polynomen und Splines
   Bernstein-Polynome; Bézier-Darstellung eines Polynomes und Bézier-Punkte; Algorithmus von de Casteljau zur Berechnung des Polynomwertes aus den Bézier-Punkten; Bézier-Darstellung eines kubischen Splines; Berechnung eines interpolierenden kubischen Splines in Bézier-Darstellung

IV Integration

1. Motivation
   Idee: Interpoliere den Integranden und integriere das Interpolationspolynom exakt
2. Quadraturformeln
   Form; Gewichte; Knoten; Mittelpunktsregel; Trapezregel; Simpsonregel; maximale Ordnung $2n + 1$
3. Newton-Cotes-Formeln
   Bestimmung der Gewichte; Ordnung n; Beispiele
4. Gauß-Formeln
   Bestimmung der Knoten und Gewichte; Ordnung $2n + 1$; Beispiele

5. Zusammengesetzte Quadraturformeln
   Erhöhung der Knotenzahl bringt i.a. keine Verbesserung; allgemeine Form zusammengesetzter Quadraturformeln; zusammengesetzte Mittelpunkts-, Trapez- und Simpsonregel; Fehler zusammengesetzter Quadraturformeln
6. Romberg Verfahren
   Idee des Verfahrens; Genauigkeit
7. Spezielle Integranden
   Aufspalten des Integrationsbereiches; geeignete Substitution; Abspalten der Singularität; geeignete Gewichtsfunktionen; Reihenentwicklungen
8. Mehrdimensionale Integrationsbereiche
   Zerlegen in einfache Teilgebiete; Quadraturformeln für Rechtecke und Würfel; Quadraturformeln für Dreiecke

V Eigenwertprobleme

1. Übersicht
   Potenzmethode und Rayleigh-Quotienten zur Berechnung des betragsmäßig größten Eigenwertes; inverses Verfahren von Wielandt und Verfahren der inversen Rayleigh-Quotienten für die Berechnung des betragsmäßig kleinsten Eigenwertes einer invertierbaren Martrix
2. Potenzmethode
   Idee des Verfahrens; Konvergenzgeschwindigkeit
3. Rayleigh-Quotienten
   Idee des Verfahrens; Konvergenzgeschwindigkeit; Rayleigh-Quotienten-Iteration doppelt so schnell wie Potenzmethode
4. Inverse Iteration von Wielandt
   Idee des Verfahrens; Konvergenzgeschwindigkeit
5. Inverse Rayleigh-Quotienten-Iteration
   Idee des Verfahrens; Konvergenzgeschwindigkeit; inverse Rayleigh-Quotienten-Iteration doppelt so schnell wie inverse Iteration von Wielandt
6. Berechnung eines Eigenvektors
   Idee des Verfahrens

VI Gewöhnliche Differentialgleichungen

1. Motivation
   Anfangswertprobleme; explizites Eulerverfahren; implizites Eulerverfahren; Trapezregel oder Verfahren von Crank-Nicolson
2. Einschrittverfahren
   Allgemeine Form eines Einschrittverfahrens; Verfahrensfunktion; Verfahrensfunktionen des expliziten und impliziten Eulerverfahrens und des Verfahrens von Crank-Nicolson
3. Verfahrensfehler und Ordnung
   Verfahrensfehler; Ordnung; Ordnung $p \iff$ lokaler und globaler Fehler $O(h^p)$; Eulerverfahren haben Ordnung 1; Verfahren von Crank-Nicolson hat Ordnung 2
4. Runge-Kutta-Verfahren
   Allgemeine Form; Runge-Kutta-Schema; explizite und implizite Verfahren; Eulerverfahren und Verfahren von Crank-Nicolson als Runge-Kutta-Verfahren; klassisches Runge-Kutta-Verfahren; SDIRK-Verfahren

5. Stabilität

   Verhalten der Eulerverfahren für die DGL $y' = -\lambda y$ mit $\lambda \gg 1$; das implizite Verfahren ist dem expliziten überlegen; Stabilität; Stabilitätsgebiet; Stabilitätsgebiete der Eulerverfahren und des Crank-Nicolson-Verfahrens; A-stabil; implizite Eulerverfahren, Crank-Nicolson-Verfahren und SDIRK-Verfahren sind A-stabil

6. Steife Differentialgleichungen

   Steif $\iff$ Lösungskomponenten mit stark unterschiedlichem Abklingverhalten $\iff$ stark unterschiedliche Zeitskalen; explizite Verfahren liefern unsinnige Lösungen; nur A-stabile implizite Verfahren geeignet

7. Schrittweitensteuerung

   Schrittweitensteuerung durch Halbieren und Ordnungsvergleich; Runge-Kutta-Fehlberg-Verfahren

8. Mehrschrittverfahren

   Verfahren von Adams-Bashforth und Adams-Moulton; BDF-Formeln; Ordnung der BDF-Formeln; Anlaufrechnung

Index

eps, 6
η_i, 83
$\eta(t_i, h_i)$, 83
$|||\cdot|||_1$, 32
$|||\cdot|||_2$, 29
$|||\cdot|||_\infty$, 31
$\mathrm{rd}(x)$, 6

A-stabil, 93
Absolutglied, 11
Adams-Bashforth-Verfahren, 101
Adams-Moulton-Verfahren, 101
äquidistante Knoten, 65
Algorithmus von de Casteljau, 58
Anfangswertproblem, 83
Anlaufrechnung, 102
Auffüllen, 27
AWP, 83

Bézier-Darstellung, 58
Bézier-Punkte, 58
Backward Difference Formulae, 101
Bandbreite, 27
Bandmatrix, 27
Basis, 5
BDF-Formeln, 101
Bernstein-Polynome, 58
Bisektionsverfahren, 33

Cholesky-Zerlegung, 23

Daten, 45
Dezimalsystem, 5
dividierte Differenzen, 46, 51
Dualsystem, 5

Einschrittverfahren, 86
Elimination, 13
Eliminationsteil, 13
ESV, 86
explizites Eulerverfahren, 83
explizites Runge-Kutta-Verfahren, 88

Exponent, 5

fill-in, 27

Gaußsches Eliminationsverfahren, 13
Gauß-Formeln, 65
gDgl, 83
gedämpftes Newtonverfahren, 42
Gewichte einer Quadraturformel, 64
gewöhnliche Differentialgleichung, 83

Hauptmatrix, 22
Hermitesches Interpolationspolynom, 52
Hermitesches Interpolationsproblem, 51
Hexadezimalsystem, 5
homogenes Gleichungssystem, 11

implizites Eulerverfahren, 84
implizites Runge-Kutta-Verfahren, 88
inhomogenes Gleichungssystem, 11
inverse Iteration von Wielandt, 78
inverse Rayleigh-Quotienten-Iteration, 80

Kessel, 38
klassisches Runge-Kutta-Verfahren, 89
Knoten, 45
Knoten einer Quadraturformel, 64
Koeffizient, 11
Koeffizientenmatrix, 11
Kondition, 29
kubische Spline-Interplation, 53
kubischer Spline, 53

Lagrangesches Interpolationspolynom, 45

Lagrangesches
Interpolationsproblem, 45
LGS, 11
lineares Gleichungssystem, 11
lokaler Verfahrensfehler, 87

Mantissenlänge, 5
Maschinengenauigkeit, 6
Matrixnorm, 29
Mehrgitterverfahren, 28
Mittelpunktsregel, 64

Newton-Cotes-Formeln, 65
Newtonsche Darstellung des
Lagrangeschen
Interpolationspolynomes, 46
Newtonverfahren, 37, 41

obere Dreiecksmatrix, 18
Ordnung, 87
Ordnung einer Quadraturformel, 64

PCG-Verfahren, 28
Permutationsmatrix, 18
Pivotsuche, 13
Poissongleichung, 27
Potenzmethode, 75

quadratische Konvergenz, 38, 41
Quadraturformel, 64

Rayleigh-Quotienten-Iteration, 77
rechte Seite, 11
Residuum, 28
Romberg-Verfahren, 69
Rücklösungsteil, 13
Runge-Kutta-Fehlberg-Verfahren, 99
Runge-Kutta-Verfahren, 88

Satz von Faber, 49
Schrittweitensteuerung durch
Halbieren, 96
Schrittweitensteuerung durch
Ordnungsvergleich, 98
SDIRK-Verfahren, 89
Sekantenverfahren, 35
Simpsonregel, 64
Spaltensummennorm, 32
s.p.d. Matrix, 21
Stabilität, 92
Stabilitätsgebiet, 92
stark diagonal implizite
Runge-Kutta-Verfahren, 89
steife Differentialgleichung, 94
Stufe eines Runge-Kutta-Verfahrens,
88
symmetrisch positiv definite Matrix,
21
symmetrische Matrix, 21

Trapezregel, 64, 84
triviale Lösung, 11

untere Dreiecksmatrix, 18

Verfahren von Crank-Nicolson, 84
Verfahren von Heron, 39
Verfahrensfunktion, 86
vorkonditionierte konjugierte
Gradienten Verfahren, 28
Vorzeichen, 5

Zahldarstellung in normalisierter
Form, 5
Zeilensummennorm, 31
Zeilentausch, 13
zusammengesetzte
Mittelpunktsregel, 66
zusammengesetzte Quadraturformel,
66
zusammengesetzte Simpsonregel, 66
zusammengesetzte Trapezregel, 66