Numerische Mathematik

für Naturwissenschaftler, Ingenieure und Informatiker

Prof. Dr. Lars Grüne
und
Dr. Michael Heinrich Baumann

Sommersemester 2023
Stand: 11. Oktober 2023

Inhaltsverzeichnis

Kapitel 1

Was ist Numerische Mathematik?

Die *Numerische Mathematik* oder kurz *Numerik* beschäftigt sich mit der numerischen (= mit konkreten Zahlenwerten) Auswertung oder Auflösung mathematischer Formeln durch geeignete Algorithmen, die typischerweise auf digitalen Rechnern implementiert werden.[1] Dies umfasst auch geeignete Näherungsstrategien, da sich viele mathematische Formeln gar nicht exakt, sondern nur angenähert lösen lassen.

Schematisch lässt sich dies wie folgt darstellen:

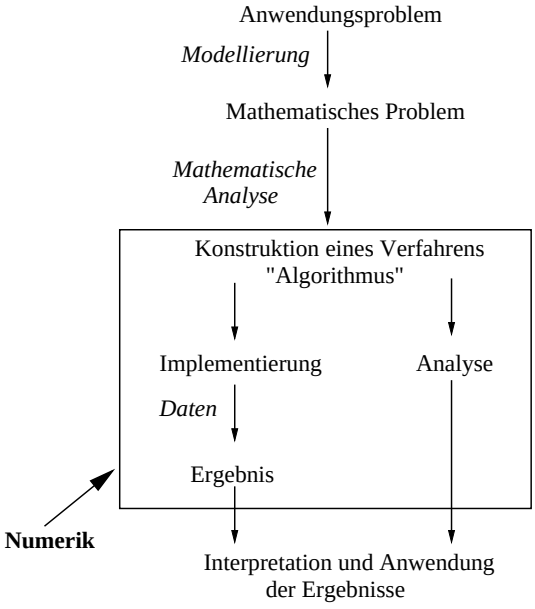

Abbildung 1.1: Numerik: Schematische Darstellung in Anlehnung an [Bollhöfer und Mehrmann, 2004]

[1] Letzteres klingt zwar heutzutage selbstverständlich, tatsächlich ist die Numerik aber viel älter als die modernen Computer. Vor der Erfindung des Computers wurde bereits viele Jahrhunderte „per Hand“ (oder vielleicht treffender „per Kopf“) numerisch gerechnet.

Bei der Konstruktion eines geeigneten Algorithmus stehen dabei insbesondere drei Aspekte im Mittelpunkt:

- Genauigkeit
- Geschwindigkeit
- Numerische Stabilität, d. h. Robustheit gegenüber Fehlern

Die ersten beiden Punkte sind dabei selbsterklärend. Der dritte Punkt bedeutet, dass Fehler, die in den Daten vorhanden sind oder in den einzelnen Rechenschritten entstehen, im Endergebnis *nicht mehr als unvermeidbar* verstärkt werden. Was „unvermeidbar“ hier bedeutet, werden wir später genauer untersuchen.

Wir werden das in Abbildung 1.1 dargestellte Vorgehen an einem sehr einfachen Beispiel veranschaulichen und dabei insbesondere aufzeigen, warum man Fehler nicht einfach vermeiden kann und der dritte Punkt deshalb wichtig ist.

Beispiel 1.1 Wir betrachten die folgende Wurfsituation:

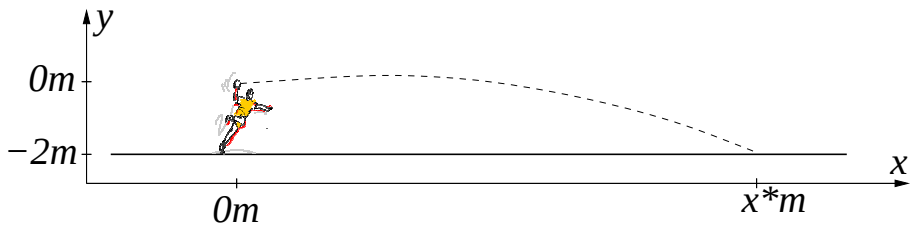

Abbildung 1.2: Wurfsituation

Anwendungsproblem: Bestimme den Punkt x^*, in dem der Ball bei gegebenem Abwurfwinkel β und gegebener Abwurfgeschwindigkeit v_0 auftrifft. □

Um dies zu berechnen, benötigen wir zunächst ein mathematisches Modell des Wurfs. Unter `http://de.wikipedia.org/wiki/Wurfparabel`[2] findet man für die Wurfparabel prinzipiell die Formel

$$y(x) = x \tan \beta - \frac{g}{2v_0^2 \cos^2 \beta} x^2, \tag{1.1}$$

in der x die horizontale und y die vertikale Position des Balles bezeichnet. Diese Formel geht wegen $y(0) = 0$ davon aus, dass der Abwurfpunkt in $(x, y) = (0, 0)$ liegt, was gerade zu unserer Wahl des Koordinatensystems passt.

Formel (1.1) ist nun unser *mathematisches Modell*. Es gibt uns an, in welcher Höhe y sich der Ball in Abhängigkeit von der horizontalen Position x aufhält.

Wie genau ist dieses Modell? Auf der angebenen Webseite ist erläutert, dass in dieser Formel der Luftwiderstand vernachlässigt wurde; die Formel stimmt also exakt nur im Vakuum.

[2] Aufgerufen am 8. April 2021.

Wenn wir das Modell für einen Wurf in der „realen Welt“ verwenden, machen wir also einen *Modellfehler*, der für kleinere Geschwindigkeiten zwar recht klein aber nichtsdestotrotz vorhanden ist.

In die Formel (1.1) gehen drei Parameter ein: β, v_0 und g. Die Parameter β und v_0 sind durch den Werfer bestimmt und können im Prinzip gemessen werden, allerdings in der Praxis sicherlich nicht mit beliebiger Genauigkeit. Der Parameter g ist die Erdbeschleunigung, die zwar mit hoher Genauigkeit berechnet werden kann, in der Praxis aber oft einfach mit $g \approx 9.81 m/s^2$ genähert wird. Durch die ungenaue Messung von β und v_0 sowie die Näherung von g machen wir *Datenfehler*.

Um nun den Punkt x^*, in dem der Ball auftrifft, zu berechnen, müssen wir die Gleichung

$$y(x^*) = -2 \tag{1.2}$$

lösen. Dies ist unser **Mathematisches Problem**. Wir suchen also eine Lösung der quadratischen Gleichung

$$ax^2 + bx + c = 0$$

mit

$$a = -\frac{g}{2v_0^2 \cos^2 \beta}, \quad b = \tan \beta \quad \text{und} \quad c = 2 \tag{1.3}$$

und benötigen folglich einen Algorithmus zur Lösung quadratischer Gleichungen. Wir geben hier einen einfachen Algorithmus an, der die bekannte allgemeine Lösungsformel für quadratische Gleichungen („Mitternachtsformel“)

$$x_{1/2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

verwendet.

Algorithmus 1.2 (Lösen einer quadratischen Gleichung)
Eingabe: Koeffizienten `a, b, c`

(1) Berechne `x1 = (-b + sqrt(b*b-4*a*c))/(2*a)`

(2) Berechne `x2 = (-b - sqrt(b*b-4*a*c))/(2*a)`

Ausgabe: Lösungen `x1, x2` □

Die Lösung des ursprünglichen Problems erhalten wir dann, indem wir Algorithmus 1.2 mit den Koeffizienten `a, b` und `c` aus (1.3) aufrufen und x^* als die größere der beiden Lösung `x1, x2` auswählen. Hierbei entstehen *Rechenfehler*, denn kein digitaler Rechner kann beliebig genau rechnen. Dies liegt daran, dass im Rechner jeweils nur eine feste Anzahl von führenden Ziffern ungleich Null pro Zahl gespeichert wird, deren letzte kaufmännisch gerundet wird:[3] Verwendet der Rechner z. B. eine Darstellung mit vier Ziffern, so werden die Zahlen

$$66.66666666, \quad 987654.321, \quad 0.0011223344$$

[3] Kaufmännisch Runden heißt, dass die Ziffern $0, 1, 2, 3, 4$ ab- und die Ziffern $5, 6, 7, 8, 9$ aufgerundet werden.

im Rechner gerundet als

$$66.67 = 0.6667 \cdot 10^2, \qquad 987700 = 0.9877 \cdot 10^6, \qquad 0.001122 = 0.1122 \cdot 10^{-2}$$

dargestellt. Beachte, dass sich hierbei die Größenordnung nicht ändert, da wegfallende Ziffern zwischen erster bzw. letzter relevanter Ziffer und dem Dezimalpunkt[4] durch 0 ersetzt werden.

Diese Rundung wird nun nicht nur für die Eingabedaten und das Endergebnis durchgeführt. Intern wird z. B. die Formel aus Algorithmus 1.2 (2) in elementare Rechenoperationen zerlegt und wie folgt ausgeführt:

```
z0 = -b;
z1 = b*b;
z2 = a*c;
z3 = 4*z2;
z4 = z1-z3;
z5 = sqrt(z4);
z6 = z0 - z5;
z7 = 2*a;
x2 = z6/z7;
```

In jedem dieser Schritte wird das Zwischenergebnis `z0, z1, z2, ...` dabei gemäß der obigen Konvention gerundet. Glücklicherweise macht dies bei unserem Wurfproblem keine Schwierigkeiten, da Algorithmus 1.2 sowohl für `x1` als auch für `x2` für die entstehende quadratische Gleichung auch bei Rundung in jedem Schritt sinnvolle Ergebnisse liefert.

Dies gilt aber nicht allgemein: Beispielsweise für die Gleichung

$$\frac{1}{1000}x^2 + 100x + 100 = 0$$

kommt es bei der Ausführung von Algorithmus 1.2 (1) mit Rundung zu Rechenfehlern, die deutlich größer als die Rundungsfehler der einzelnen Schritte sind [→ Übungsaufgabe]. Algorithmus 1.2 ist also tatsächlich *nicht* numerisch stabil.

Wir bezeichnen mit $\mathcal{R}_n : \mathbb{R} \to \mathbb{R}$ den eben beschriebenen Rundungsoperator[5], also den, der jeder Zahl die Zahl zuordnet, die nur noch n führende Stellen hat (und die letzte kaufmännisch rundet). Im obigen Beispiel war $n = 4$. Wir müssen jedoch beachten, dass im Allgemeinen nicht mehr alle unserer „normalen“ Rechengesetze gelten. So ist

$$\mathcal{R}_n(a + \mathcal{R}_n(b + c)) \overset{i.\,A.}{\neq} \mathcal{R}_n(\mathcal{R}_n(a + b) + c)$$

und

$$\mathcal{R}_n(\mathcal{R}(a \cdot b) + \mathcal{R}_n(a \cdot c))) \overset{i.\,A.}{\neq} \mathcal{R}_n(a \cdot \mathcal{R}_n(b + c))$$

[4] Beachte, dass in diesem Skript als Dezimaltrennzeichen der Punkt, statt wie im Deutschen üblich das Komma, verwendet wird, da in Programmiersprachen normalerweise der Punkt benutzt wird. Um keine weitere Verwirrung zu schaffen, wird auf Tausendertrennzeichen verzichtet.

[5] Beachte, dass $\mathcal{R}_n(x)$ im Allgemeinen nicht dem Runden auf n Nachkommastellen entspricht.

[→ Übungsaufgabe].

Abbildung 1.3 ergänzt nun die Abbildung 1.1 um die einzelnen Fehlerquellen. Beachte, dass sich dort eine weitere Fehlerquelle findet, nämlich der sogenannte *Diskretisierungsfehler*, die in unserem einfachen Beispiel nicht auftritt.

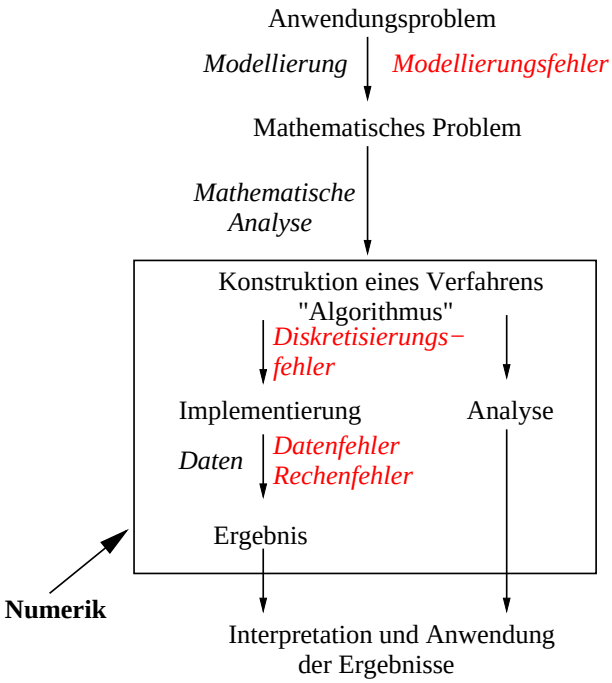

Abbildung 1.3: Numerik: Schematische Darstellung mit Fehlerquellen (Grundstruktur in Anlehnung an [Bollhöfer und Mehrmann, 2004])

Kapitel 2

Lineare Gleichungssysteme

Das Lösen linearer Gleichungssysteme ist eines der wichtigsten Teilprobleme der Numerik. Viele numerische Algorithmen, die wir im Verlauf dieser Vorlesung kennen lernen werden, führen am Ende auf die Lösung eines linearen Gleichungssystems.

Ausführlich aufgeschrieben besteht das Problem darin, Zahlen $x_1, \ldots, x_n \in \mathbb{R}$ zu bestimmen, für die das Gleichungssystem

$$\begin{array}{ccccccccc} a_{1\,1}x_1 & + & a_{1\,2}x_2 & + & \ldots & + & a_{1\,n}x_n & = & b_1 \\ & & & & & & & \vdots & \\ a_{m\,1}x_1 & + & a_{m\,2}x_2 & + & \ldots & + & a_{m\,n}x_n & = & b_m \end{array} \tag{2.1}$$

erfüllt ist. Die ausführliche Schreibweise in (2.1) ist etwas unhandlich, weswegen wir lineare Gleichungsysteme in der üblichen Matrix-Vektor-Form schreiben werden, nämlich als

$$Ax = b, \tag{2.2}$$

mit

$$A = \begin{pmatrix} a_{1\,1} & a_{1\,2} & \ldots & a_{1\,n} \\ \vdots & \vdots & & \vdots \\ a_{m\,1} & a_{m\,2} & \ldots & a_{m\,n} \end{pmatrix}, \quad x = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \quad \text{und} \quad b = \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix}. \tag{2.3}$$

Einige kurze Bemerkungen zur Notation in diesem Kapitel: Wir werden Matrizen typischerweise mit großen Buchstaben (z. B. A) und Vektoren mit kleinen (z. B. b) bezeichnen. Ihre Einträge werden wir mit indizierten Kleinbuchstaben bezeichnen, wie in (2.3). Mit einem hochgestellten „T“ bezeichnen wir transponierte Matrizen und Vektoren, für A und x aus (2.3) also z. B.

$$x^T = (x_1, \ldots, x_n), \quad A^T = \begin{pmatrix} a_{1\,1} & a_{2\,1} & \ldots & a_{m\,1} \\ \vdots & \vdots & & \vdots \\ a_{1\,n} & a_{2\,n} & \ldots & a_{n\,m} \end{pmatrix}.$$

Bevor wir zu Lösungsalgorithmen kommen, werden wir erst noch ein wichtiges Anwendungsproblem betrachten, das auf die Lösung eines linearen Gleichungssystems führt.

2.1 Lineare Ausgleichsprobleme

Das erste unserer Anwendungsbeispiele ist für viele praktische Zwecke besonders wichtig, weswegen wir es etwas genauer untersuchen wollen. Nehmen wir an, wir haben ein Experiment durchgeführt, bei dem wir für verschiedene Eingabewerte $t_1, t_2, \ldots, t_m$ Messwerte $z_1, z_2, \ldots, z_m$ erhalten. Aufgrund von theoretischen Überlegungen (z. B. aufgrund eines zugrundeliegenden physikalischen Gesetzes) kennt man eine Funktion $f(t)$, für die $f(t_i) = z_i$ gelten sollte. Diese Funktion wiederum hängt aber nun von unbekannten Parametern $x_1, \ldots, x_n$ ab; wir schreiben $f(t; x)$ für $x = (x_1, \ldots, x_n)^T$, um dies zu betonen. Zum Beispiel könnte $f(t; x)$ durch

$$f(t; x) = x_1 + x_2 t \quad \text{oder} \quad f(t; x) = x_1 + x_2 t + x_3 t^2$$

gegeben sein. Im ersten Fall beschreibt die gesuchte Funktion eine Gerade, im zweiten eine Parabel. Wichtig ist hierbei, dass die Funktion linear von den x_i abhängt. Wenn wir annehmen, dass die Funktion f das Experiment wirklich exakt beschreibt und keine Messfehler vorliegen, so könnten wir die Parameter x_i durch Lösen des linearen Gleichungssystems

$$\begin{aligned} f(t_1; x) &= z_1 \\ &\vdots \\ f(t_m; x) &= z_m \end{aligned} \tag{2.4}$$

nach x bestimmen. Wir müssen also $\widetilde{A}x = z$ lösen, wobei $\widetilde{A}$ und z für die obigen Beispiele gegeben sind durch

$$\widetilde{A} = \begin{pmatrix} 1 & t_1 \\ \vdots & \vdots \\ 1 & t_m \end{pmatrix} \quad \text{bzw.} \quad \widetilde{A} = \begin{pmatrix} 1 & t_1 & t_1^2 \\ \vdots & \vdots & \vdots \\ 1 & t_m & t_m^2 \end{pmatrix} \quad \text{und} \quad z = \begin{pmatrix} z_1 \\ \vdots \\ z_m \end{pmatrix}$$

Diese linearen Gleichungssysteme haben m Gleichungen (eine für jedes Wertepaar (t_i, z_i)) und n Unbekannte (nämlich gerade die unbekannten Parameter x_i), wobei m üblicherweise sehr viel größer als n ist. Man sagt, dass das Gleichungsystem *überbestimmt* ist. Da Messwerte eines Versuchs praktisch immer mit Fehlern behaftet sind, ist es sicherlich zu optimistisch, anzunehmen, dass das Gleichungssystem $\widetilde{A}x = z$ lösbar ist (überbestimmte Gleichungssysteme haben oft keine Lösung!). Statt also den (vermutlich vergeblichen) Versuch zu machen, eine exakte Lösung x dieses Systems zu finden, wollen wir probieren, eine möglichst gute Näherungslösung zu finden, d. h., wenn $\widetilde{A}x = z$ nicht lösbar ist, wollen wir zumindest ein x finden, sodass $\widetilde{A}x$ möglichst nahe bei z liegt. Dazu müssen wir ein Kriterium für „möglichst nahe“ wählen, das sowohl sinnvoll ist als auch eine einfache Lösung zulässt (das ist ein Kompromiss, den man in der Numerik sehr oft eingeht). Hier bietet sich das sogenannte *Ausgleichsproblem* (auch *Methode der kleinsten Quadrate* genannt) an:

Finde $x = (x_1, \ldots, x_n)^T$, sodass $\varphi(x) := \|\widetilde{A}x - z\|^2$ minimal wird.

Hierbei bezeichnet $\|y\|$ die euklidische Norm eines Vektors $y \in \mathbb{R}^n$, also

$$\|y\| = \sqrt{\sum_{i=1}^{n} y_i^2} \quad \text{und damit} \quad \|y\|^2 = \sum_{i=1}^{n} y_i^2.$$

Um die Funktion φ zu minimieren, setzen wir den Gradienten

$$\nabla\varphi(x) = \left(\frac{\partial\varphi}{\partial x_1}(x), \ldots, \frac{\partial\varphi}{\partial x_n}(x)\right)^T$$

gleich Null. Beachte dazu, dass der Gradient der Funktion $g(x) := \|f(x)\|^2 = \sum_{i=1}^n f_i(x)^2$ für beliebiges $f : \mathbb{R}^n \to \mathbb{R}^n$ durch

$$\nabla g(x) = 2Df(x)^T f(x)$$

gegeben ist, weil $(\nabla g(x))_j = \frac{\partial g}{\partial x_j}(x) = \sum_{i=1}^n 2 \cdot f_i(x) \cdot \frac{\partial f_i}{\partial x_j}(x)$ und $Df(x) = \left(\frac{f_i}{x_j}(x)\right)_{i=1,\ldots,n;j=1,\ldots,n}$. Wir erhalten also

$$\nabla\varphi(x) = 2\widetilde{A}^T(\widetilde{A}x - z) = 2\widetilde{A}^T\widetilde{A}x - 2\widetilde{A}^T z$$

Falls $\widetilde{A}$ vollen Spaltenrang besitzt, ist die zweite Ableitung $D^2\varphi(x) = 2\widetilde{A}^T\widetilde{A}$ positiv definit (vgl. Definition 2.5), womit jede Nullstelle des Gradienten $\nabla\varphi$ ein Minimum von φ ist. Folglich minimiert ein Vektor x die Funktion φ genau dann, wenn gilt:

$$0 = \nabla\varphi(x) = 2\widetilde{A}^T\widetilde{A}x - 2\widetilde{A}^T z.$$

Diese Gleichung nennt man *Normalengleichung*. Das Ausgleichsproblem wird also wie folgt gelöst:

$$\text{löse} \quad Ax = b \quad \text{mit} \quad A = \widetilde{A}^T\widetilde{A} \text{ und } b = \widetilde{A}^T z.$$

Das zunächst recht kompliziert scheinende Minimierungsproblem für φ wird also auf die Lösung eines linearen Gleichungssystems zurückgeführt.

2.2 Das Gauß'sche Eliminationsverfahren

Wir werden nun ein erstes Verfahren zur Lösung linearer Gleichungssysteme kennenlernen. Das *Gauß'sche Eliminationsverfahren* ist ein sehr anschauliches Verfahren, das zudem recht leicht implementiert werden kann. Es beruht auf der einfachen Tatsache, dass ein lineares Gleichungssystem $Ax = b$ leicht lösbar ist, falls die Matrix A in *oberer Dreiecksform* vorliegt, d. h.,

$$A = \begin{pmatrix} a_{1\,1} & a_{1\,2} & \ldots & a_{1\,n} \\ 0 & a_{2\,2} & \ldots & a_{2\,n} \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \ldots & 0 & a_{n\,n} \end{pmatrix}.$$

In diesem Fall kann man $Ax = b$ leicht mittels der rekursiven Vorschrift

$$x_n = \frac{b_n}{a_{n\,n}}, \quad x_{n-1} = \frac{b_{n-1} - a_{n-1\,n}x_n}{a_{n-1\,n-1}}, \ldots, \quad x_1 = \frac{b_1 - a_{1\,2}x_2 - \ldots - a_{1\,n}x_n}{a_{1\,1}}$$

oder, kompakt geschrieben,

$$x_i = \frac{b_i - \sum_{j=i+1}^{n} a_{i\,j}x_j}{a_{i\,i}}, \qquad i = n, n-1, \ldots, 1 \tag{2.5}$$

(mit der Konvention $\sum_{j=n+1}^{n} a_{ij}x_j = 0$) lösen. Dieses Verfahren wird als *Rückwärtseinsetzen* bezeichnet.

Die Idee des Gauß'schen Eliminationsverfahrens liegt nun darin, das Gleichungssystem $Ax = b$ in ein Gleichungssystem $\widetilde{A}x = \tilde{b}$ so umzuformen, dass die Matrix $\widetilde{A}$ in oberer Dreiecksform vorliegt. Dazu werden Vielfache einer Zeile des Gleichungssystems von einer anderen Zeile subtrahiert. Man überlegt sich leicht, dass dies an den Lösungen des Gleichungssystems nichts ändert und das Gleichungssystem mit der umgeformten Matrix folglich die gleiche Lösung wie das System mit der ursprünglichen Matrix besitzt. Zu beachten ist dabei lediglich, dass die „rechte Seite" des Gleichungssystem – in Matrix-Vektor-Schreibseise also der Vektor b – entsprechend mit umgeformt werden muss.

Wir formulieren den Algorithmus für allgemeine quadratische Matrizen A.

Algorithmus 2.1 (Gauß-Elimination, Grundversion)
Gegeben sei eine Matrix $A \in \mathbb{R}^{n\times n}$ und ein Vektor $b \in \mathbb{R}^n$.

(1) für j von 1 bis $n-1$ (Spaltenzähler)
    für i von n bis $j+1$ (Zeilenzähler)

(2) falls $a_{ij} \neq 0$:

(3) subtrahiere a_{ij}/a_{jj}-mal die j-te Zeile von der i-ten Zeile,
d. h., führe (3a), (3b) und (3c) aus:

(3a) Setze $\alpha := a_{ij}/a_{jj}$ und $a_{ij} := 0$ und berechne:

(3b) $a_{ik} := a_{ik} - \alpha\, a_{jk}$ für alle $k = j+1, \ldots, n$ mit $a_{jk} \neq 0$

(3c) $b_i := b_i - \alpha\, b_j$

(4) Ende der i-Schleife
Ende der j-Schleife

□

Wir wollen dieses Verfahren an einem Beispiel veranschaulichen.

Beispiel 2.2 Gegeben sei das lineare Gleichungssystem (2.2) mit

$$A = \begin{pmatrix} 1 & 5 & 6 \\ 7 & 9 & 6 \\ 2 & 3 & 4 \end{pmatrix} \quad \text{und} \quad b = \begin{pmatrix} 29 \\ 43 \\ 20 \end{pmatrix}.$$

Um die Matrix A auf obere Dreiecksgestalt zu bringen, müssen wir die drei Einträge 7, 2 und 3 unterhalb der Diagonalen auf 0 bringen („eliminieren"). Wir beginnen mit der 2. Hierzu subtrahieren wir 2-mal die erste Zeile von der letzten und erhalten so

$$A_1 = \begin{pmatrix} 1 & 5 & 6 \\ 7 & 9 & 6 \\ 0 & -7 & -8 \end{pmatrix}$$

Das Gleiche machen wir mit dem Vektor b. Dies liefert

$$b_1 = \begin{pmatrix} 29 \\ 43 \\ -38 \end{pmatrix}.$$

Nun fahren wir fort mit der 7: Wir subtrahieren 7-mal die erste Zeile von der zweiten, sowohl in A_1 als auch in b_1, und erhalten

$$A_2 = \begin{pmatrix} 1 & 5 & 6 \\ 0 & -26 & -36 \\ 0 & -7 & -8 \end{pmatrix} \quad \text{und} \quad b_2 = \begin{pmatrix} 29 \\ -160 \\ -38 \end{pmatrix}.$$

Im dritten Schritt „eliminieren“ wir die -7, die jetzt an der Stelle der 3 steht, indem wir $7/26$-mal die zweite Zeile von der dritten subtrahieren:

$$A_3 = \begin{pmatrix} 1 & 5 & 6 \\ 0 & -26 & -36 \\ 0 & 0 & \frac{22}{13} \end{pmatrix} \quad \text{und} \quad b_3 = \begin{pmatrix} 29 \\ -160 \\ \frac{66}{13} \end{pmatrix}.$$

Hiermit sind wir fertig und setzen $\widetilde{A} = A_3$ und $\tilde{b} = b_3$. Rückwärtseinsetzen gemäß (2.5) liefert dann

$$x_3 = \frac{\frac{66}{13}}{\frac{22}{13}} = 3, \quad x_2 = \frac{-160 - 3 \cdot (-36)}{-26} = 2 \text{ und } x_1 = \frac{29 - 2 \cdot 5 - 3 \cdot 6}{1} = 1.$$

□

Es kann passieren, dass dieser Algorithmus zu keinem Ergebnis führt, obwohl das Gleichungssystem lösbar ist. Der Grund dafür liegt in Schritt (3a), in dem durch das Diagonalelement a_{jj} geteilt wird. Hierbei wurde stillschweigend angenommen, dass dieses ungleich Null ist, was aber im Allgemeinen nicht der Fall sein muss. Glücklicherweise gibt es eine Möglichkeit, dieses zu beheben:

Nehmen wir an, dass wir Schritt (3a-c) für gegebene Indizes i und j ausführen möchten und $a_{jj} = 0$ ist. Nun gibt es zwei Möglichkeiten: Im ersten Fall ist $a_{ij} = 0$. In diesem Fall brauchen wir nichts zu tun, da das Element a_{ij}, das auf 0 gebracht werden soll, bereits gleich 0 ist. Im zweiten Fall gilt $a_{ij} \neq 0$. In diesem Fall können wir die i-te und j-te Zeile der Matrix A sowie die entsprechenden Einträge im Vektor b vertauschen, wodurch wir die gewünschte Eigenschaft $a_{ij} = 0$ erreichen, nun allerdings nicht durch Elimination sondern durch Vertauschung. Dieses Verfahren nennt man *Pivotierung* und der folgende Algorithmus bringt nun tatsächlich jedes lineare Gleichungssystem in Dreiecksform.

Algorithmus 2.3 (Gauß-Elimination mit Pivotierung)
Gegeben sei eine Matrix $A \in \mathbb{R}^{n \times n}$ und ein Vektor $b \in \mathbb{R}^n$.

(1) für j von 1 bis $n-1$ (Spaltenzähler)
    für i von n bis $j+1$ (Zeilenzähler)

(2) falls $a_{ij} \neq 0$:

(2*) falls $a_{jj} = 0$, vertausche a_{jk} und a_{ik}, $k = j, \ldots, n$, sowie b_i und b_j

(3) falls (2*) für diese i und j nicht ausgeführt wurden,
subtrahiere a_{ij}/a_{jj}-mal die j-te Zeile von der i-ten Zeile,
d. h., führe (3a), (3b) und (3c) aus:

(3a) Setze $\alpha := a_{ij}/a_{jj}$ und $a_{ij} := 0$ und berechne

(3b) $a_{ik} := a_{ik} - \alpha\, a_{jk}$ für alle $k = j+1, \ldots, n$ mit $a_{jk} \neq 0$

(3c) $b_i := b_i - \alpha\, b_j$

(4) Ende der i-Schleife
Ende der j-Schleife

□

Das Element a_{ij}, das in Schritt (2*) mit a_{jj} getauscht wird, nennt man *Pivotelement*. Wir werden später im Abschnitt 2.4 eine Strategie kennen lernen, bei der – auch wenn $a_{jj} \neq 0$ ist – gezielt ein Element $a_{kj} \neq 0$ als Pivotelement ausgewählt wird, mit dem man ein besseres Robustheitsverhalten des Algorithmus' erhalten kann.

Wie viele elementare Rechenoperationen benötigt dieser Algorithmus?

Alle Rechenoperationen finden in den Schritten (3a)–(3c) statt:

- Schritt (3a): 1 Operation
- Schritt (3b): $2(n-j)$ Operationen, falls alle $a_{jk} \neq 0$
- Schritt (3c): 2 Operationen

Dies macht insgesamt $2(n-j)+3 = 2(n-j)+3$ Operationen. Für jedes j wird Schritt (3) von Schritt (2) aus im schlechtesten Fall für $i = n, \ldots, j+1$, also $n-j$ mal aufgerufen, was auf

$$2(n-j)^2 + 3(n-j) = 2n^2 - 4nj + 2j^2 + 3n - 3j = 2j^2 + (-4n-3)j + 2n^2 + 3n$$

Operationen für die j-te Spalte führt. Dies geschieht für die Spalten $j = 1, \ldots, n-1$, wodurch wir insgesamt

$$\begin{aligned}
&\sum_{j=1}^{n-1} \left(2j^2 + (-4n-3)j + (2n^2+3n)\right) \\
&= 2 \underbrace{\sum_{j=1}^{n-1} j^2}_{=\frac{n^3}{3}-\frac{n^2}{2}+\frac{n}{6}} + (-4n-3) \underbrace{\sum_{j=1}^{n-1} j}_{=\frac{n(n-1)}{2}} + \sum_{j=1}^{n-1} (2n^2+3n) \\
&= \frac{2}{3}n^3 - n^2 + \frac{1}{3}n - (4n+3)\frac{n(n-1)}{2} + (n-1)(2n^2+3n) \\
&= \frac{2}{3}n^3 + \frac{1}{2}n^2 - \frac{7}{6}n
\end{aligned}$$

Operationen erhalten. Sind wir nur an einer Abschätzung des Aufwands interessiert, so betrachten wir nur die höchste Potenz und sagen, dass der Algorithmus den Aufwand der Ordnung $\mathcal{O}(n^3)$ besitzt.

Beachte, dass wir hier zwei mal den schlechtesten Fall angenommen haben:

(i) In Schritt (3b) haben wir $a_{j\,k} \neq 0$ für alle $k = j, \ldots, n$ angenommen

(ii) In Schritt (2) haben wir $a_{i\,j} \neq 0$ für alle $i = n, \ldots, j+1$ angenommen

Wenn viele dieser Elemente bereits Null sind, wird sich die Anzahl der Rechenoperationen deutlich verringern. Dies ist z. B. der Fall für sogenannte Bandmatrizen, also Matrizen, bei denen nur die Diagonale und Nebendiagonalen jeweils gleichmäßig ober und unterhalb der Diagonalen Einträge ungleich 0 besitzen, insgesamt m (Neben-)Diagonalen. z. B. ist eine Bandmatrix mit $m = 3$ von der Form

$$\begin{pmatrix} a_1 & b_1 & 0 & \ldots & \ldots & 0 \\ c_2 & a_2 & b_2 & \ddots & \ddots & \vdots \\ 0 & c_3 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & b_{n-2} & 0 \\ 0 & \ldots & 0 & c_{n-1} & a_{n-1} & b_{n-1} \\ 0 & \ldots & \ldots & 0 & c_n & a_n \end{pmatrix}.$$

Für Matrizen dieser Form verringert sich der Rechenaufwand der Gauß-Elimination of $\mathcal{O}(n)$, weil für jede Spalte nur eine von n unabhängige Anzahl von Rechnungen nötig ist.

2.3 LR-Faktorisierung und das Choleski-Verfahren

In der obigen Version des Gauß-Verfahrens haben wir die Matrix A auf obere Dreiecksform gebracht und zugleich alle dafür notwendigen Operationen auch auf den Vektor b angewendet. Dies ist ineffizient, wenn man dasselbe lineare Gleichungssystem für verschiedene rechte Seiten b lösen will (z. B. wenn man die Finite Differenzenmethode aus Modellprojekt 2 für verschiedene Temparaturen $g(x)$ oder verschiedene Koeffizienten α lösen will).

Eine Lösung für dieses Problem liegt darin, den Vektor b unverändert zu lassen und statt dessen eine Zerlegung

$$A = LR$$

zu berechnen, wobei R in der bereits bekannten oberen Dreiecksform und L in *unterer Dreiecksform*

$$L = \begin{pmatrix} l_{1\,1} & 0 & \ldots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ l_{n-1\,1} & \ldots & l_{n-1\,n-1} & 0 \\ l_{n\,1} & l_{n\,2} & \ldots & l_{n\,n} \end{pmatrix}.$$

vorliegt. Die Zerlegung $A = LR$ wird als *LR-Faktorisierung* oder *LR-Zerlegung* bezeichnet.

Um $Ax = b$ zu lösen, kann man dann $LRx = b$ wie folgt in zwei Schritten lösen: Zunächst löst man das Gleichungssystem $Ly = b$. Dies kann, ganz analog zum Rückwärtseinsetzen (2.5), durch *Vorwärtseinsetzen* geschehen:

$$y_i = \frac{b_i - \sum_{j=1}^{i-1} l_{i\,j} y_j}{l_{i\,i}}, \qquad i = 1, 2, \ldots, n. \tag{2.6}$$

Im zweiten Schritt löst man dann durch Rückwärtseinsetzen das Gleichungssystem $Rx = y$. Dann gilt

$$Ax = LRx = Ly = b,$$

womit das gewünschte System $Ax = b$ gelöst ist.

Das Gauß'sche Eliminationsverfahren läßt sich so erweitern, dass damit – bis auf die Zeilenvertauschungen in A – eine LR-Zerlegung einer invertierbaren Matrix A möglich ist.

Die Subtraktion „i-te Zeile Minus α Mal j-te Zeile“ lässt sich durch Multiplikation von links mit der Matrix

$$\begin{pmatrix} 1 & & & & & \\ & \ddots & & & & \\ & & \ddots & & & \\ & -\alpha & & \ddots & & \\ & & & & \ddots & \\ & & & & & 1 \end{pmatrix}$$

erzielen, hierbei steht α in der i-ten Zeile und j-ten Spalte.

Wenn sich die Gauß-Elimination ohne Zeilenvertauschung durchführen lässt, so werden nur Operationen mit solchen Matrizen benötigt. Wenn man die dabei benötigten Matrizen F_1, $F_2, \ldots,$ F_k zu einer Matrix $F = F_k \cdot F_{k-1} \cdots F_1$ aufmultipliziert, so erhält man $A = LR$ mit $L = F^{-1}$, also die LR-Faktorisierung (beachte, dass die F-Matrix als Produkt unterer Dreiecksmatrizen eine untere Dreiecksmatrix ist, weswegen auch ihre Inverse eine untere Dreiecksmatrix ist).

Sind allerdings wegen der Pivotierung Zeilenvertauschungen nötig, so ist lässt sich i. A. keine LR-Zerlegung erhalten.[1] Die Vertauschung der i-ten und der j-ten Zeile in A lässt

[1] tatsächlich existiert für allgemeine invertierbare Matrizen im Allgemeinen keine LR-Zerlegung

sich durch die Multiplikation von links mit Matrizen der Form

$$\begin{pmatrix} 1 & & & & & & & & & \\ & \ddots & & & & & & & & \\ & & 1 & & & & & & & \\ & & & 0 & & & 1 & & & \\ & & & & 1 & & & & & \\ & & & & & \ddots & & & & \\ & & & & & & 1 & & & \\ & & & 1 & & & & 0 & & \\ & & & & & & & & 1 & \\ & & & & & & & & & \ddots \\ & & & & & & & & & & 1 \end{pmatrix}$$

erreichen, wobei die 0-Einträge gerade an der i-ten und j-ten Stelle stehen. Sind solche Zeilenvertauschungen notwendig, so kann man die benötigten Matrizen $P_1, \ldots, P_l$ ebenfalls zu einer Matrix $P = P_l \cdots P_1$ aufmultipizieren und erhält dann eine Zerlegung der Form $PA = LR$ mit $L = F^{-1}$ von oben (P nennt man *Permutationsmatrix*).

Bei der Invertierung von F und der Berechnung von L fällt uns auf, dass in L genau die αs aus der Gauß-Elimination stehen. In der Tat kann man sich die Multiplikationen $F_k \cdots F_1$ und die Invertierung sparen und L direkt bestimmen.

Algorithmus 2.4 (LR-Zerlegung)
Gegeben sei eine Matrix $A \in \mathbb{R}^{n \times n}$

(0) $R := A, L := Id_n$

(1) für j von 1 bis $n-1$ (Spaltenzähler)
für i von n bis $j+1$ (Zeilenzähler)

(2) falls $r_{ij} \neq 0$:

(3a) Setze $l_{ij} := r_{ij}/r_{jj}$ und $r_{ij} := 0$ und berechne:

(3b) $r_{ik} := r_{ik} - l_{ij}\, r_{jk}$ für alle $k = j+1, \ldots, n$ mit $r_{jk} \neq 0$

(4) Ende der i-Schleife
Ende der j-Schleife

□

Mit dem sogenannten In-Place-LR-Verfahren könnte man sogar noch viel Speicherplatz sparen. Wir widmen uns nun jedoch der Frage, warum es genügt, die αs, die jetzt l_{ij} heißen, direkt in L zu schreiben, anstatt F^{-1} zu berechnen. Wir zeigen für den Fall $n = 3$ exemplarisch, dass das L aus dem LR-Algorithmus dem aus den theoretischen Überlegungen entspricht:

$$
\begin{aligned}
&\begin{pmatrix} 1 & & \\ l_{2\,1} & 1 & \\ l_{3\,1} & l_{3\,2} & 1 \end{pmatrix} \cdot F = \\
&= \begin{pmatrix} 1 & & \\ l_{2\,1} & 1 & \\ l_{3\,1} & l_{3\,2} & 1 \end{pmatrix} \cdot F_3 F_2 F_1 \\
&= \begin{pmatrix} 1 & & \\ l_{2\,1} & 1 & \\ l_{3\,1} & l_{3\,2} & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & & \\ & 1 & \\ & -l_{3\,2} & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & & \\ -l_{2\,1} & 1 & \\ & & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & & \\ & 1 & \\ -l_{3\,1} & & 1 \end{pmatrix} \\
&= \begin{pmatrix} 1 & & \\ l_{2\,1} & 1 & \\ l_{3\,1} & & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & & \\ -l_{2\,1} & 1 & \\ & & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & & \\ & 1 & \\ -l_{3\,1} & & 1 \end{pmatrix} \\
&= \begin{pmatrix} 1 & & \\ & 1 & \\ l_{3\,1} & & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & & \\ & 1 & \\ -l_{3\,1} & & 1 \end{pmatrix} \\
&= \begin{pmatrix} 1 & & \\ & 1 & \\ & & 1 \end{pmatrix}
\end{aligned}
$$

Eine andere Methode zur Berechnung einer LR-Zerlegung ist das *Choleski-Verfahren*. Dieses funktioniert zwar nicht für allgemeine Matrizen, sondern nur für *symmetrische, positiv definite* Matrizen, liefert dafür aber eine besonders schöne Form der LR-Faktorisierung.

Definition 2.5 (i) Eine Matrix $A \in \mathbb{R}^{n\times n}$ heißt *symmetrisch*, falls $A^T = A$ ist.

(ii) Eine Matrix $A \in \mathbb{R}^{n\times n}$ heißt *positiv definit*, falls $x^T Ax > 0$ ist für alle Vektoren $x \in \mathbb{R}^n$ mit $x \neq (0, \ldots, 0)^T$. □

Diese Bedingung ist zwar einschränkend aber in vielen Anwendungen erfüllt. So führt z. B. das Ausgleichsproblem stets auf ein Gleichungssystem mit symmetrischer und positiv definiter Matrix A. Später werden wir die Spline-Interpolation und das Finite Differenzenverfahren behandeln, für die das ebenso ist. Für symmetrische und positiv definite Matrizen kann man zeigen, dass immer eine LR-Zerlegung existiert, die zusätzlich die schöne Form $R = L^T$ besitzt, insbesondere genügt es also, die untere Dreiecksmatrix L zu berechnen. Der Beweis folgt dabei konstruktiv aus dem folgenden Algorithmus.

Algorithmus 2.6 (Choleski-Verfahren) Gegeben sei eine Matrix $A \in \mathbb{R}^{n\times n}$.

(1) Für j von 1 bis n (Spaltenzähler)
Für i von 1 bis n (Zeilenzähler)

(2a) Falls $i > j$ setze $\quad l_{i\,j} := \dfrac{a_{i\,j} - \sum_{k=1}^{j-1} l_{i\,k} l_{j\,k}}{l_{j\,j}}$

(2b) Falls $i = j$ setze $\quad l_{i\,i} := \sqrt{a_{i\,i} - \sum_{k=1}^{i-1} l_{i\,k}^2}$

(2c) Falls $i < j$ setze $\quad l_{i\,j} := 0$

(3) Ende der i-Schleife
Ende der j-Schleife

□

Offenbar ist dieser Algorithmus nicht so anschaulich wie die Gauß-Elimination. Um zu beweisen, dass dieser Algorithmus das richtige Ergebnis liefert, schreiben wir die Gleichung $A = LL^T$ explizit auf und lösen nach L auf. Dies ist jedoch direkt für beliebige Dimensionen sehr unübersichtlich, weswegen wir per Induktion über die Dimension der Matrix A vorgehen. Zum **Induktionsanfang** betrachten wir zunächst 2×2 Matrizen. Hier ergibt sich

$$\begin{pmatrix} a_{1\,1} & a_{1\,2} \\ a_{2\,1} & a_{2\,2} \end{pmatrix} = \begin{pmatrix} l_{1\,1} & 0 \\ l_{2\,1} & l_{2\,2} \end{pmatrix} \begin{pmatrix} l_{1\,1} & l_{2\,1} \\ 0 & l_{2\,2} \end{pmatrix} = \begin{pmatrix} l_{1\,1}^2 & l_{1\,1} l_{2\,1} \\ l_{2\,1} l_{1\,1} & l_{2\,1}^2 + l_{2\,2}^2 \end{pmatrix}. \tag{2.7}$$

Daraus erhalten wir

$$\begin{aligned} l_{1\,1} &= \sqrt{a_{1\,1}} \\ l_{2\,1} &= a_{2\,1}/l_{1\,1} \\ l_{2\,2} &= \sqrt{a_{2\,2} - l_{2\,1}^2} \end{aligned}$$

was genau den Berechnungen im Algorithmus 2.6 für $i = 1, 2$ und $j = 1, 2$ entspricht. Die erste und letzte Gleichung sind dabei reell lösbar, denn aus der positiven Definitheit von A folgt $a_{1\,1} > 0$ und

$$a_{2\,2} - l_{2\,1}^2 = a_{2\,2} - \frac{a_{2\,1}^2}{l_{1\,1}^2} = a_{2\,2} - \frac{a_{2\,1}^2}{a_{1\,1}} = \frac{\det(A)}{a_{1\,1}} > 0.$$

Für größere Matrizen gehen wir per Induktion vor. Wir schreiben

$$A = \begin{pmatrix} A_{n-1} & \bar{a}_n \\ \bar{a}_n^T & a_{n\,n} \end{pmatrix}$$

Falls A symmetrisch und positiv definit ist, so hat auch die Matrix A_{n-1} diese Eigenschaft. Als **Induktionsannahme** nehmen wir an, dass der Choleski-Algorithmus 2.6 für die $(n-1) \times (n-1)$-Matrix A_{n-1} die LR Faktorisierung der Form $A_{n-1} = L_{n-1} L_{n-1}^T$ korrekt berechnet.

Wir schreiben die gesuchte untere Dreiecksmatrix L nun als

$$L = \begin{pmatrix} L_{n-1} & 0 \\ \bar{l}_n^T & l_{n\,n} \end{pmatrix}$$

mit $\bar{l}_n = (l_{n\,1}, \ldots, l_{n\,n-1})^T$.

Zum **Induktionsschluss** müssen wir nun die Gleichungen im Choleski-Algorithmus für $i = n$ und $j = 1, \ldots, n$ überprüfen (die Gleichungen für $i \leq n-1$ liefern nach der Induktionsannahme bereits die richtige Matrix L_{n-1}). Dazu betrachten wir die Gleichung $LL^T = A$, die sich mit der obigen Notation als

$$\begin{pmatrix} A_{n-1} & \bar{a}_n \\ \bar{a}_n^T & a_{n\,n} \end{pmatrix} = \begin{pmatrix} L_{n-1} & 0 \\ \bar{l}_n^T & l_{n\,n} \end{pmatrix} \begin{pmatrix} L_{n-1}^T & \bar{l}_n \\ 0 & l_{n\,n} \end{pmatrix} = \begin{pmatrix} L_{n-1}L_{n-1}^T & L_{n-1}\bar{l}_n \\ (L_{n-1}\bar{l}_n)^T & \bar{l}_n^T\bar{l}_n + l_{n\,n}^2 \end{pmatrix} \tag{2.8}$$

schreiben lässt. Bestimmt man nun die Einträge im Vektor $\bar{l}_n$ durch Vorwärtseinsetzen aus dem Gleichungssystem $L_{n-1}\bar{l}_n = \bar{a}_n$, so erhält man gerade die Gleichungen für $l_{n,j}$, $j = 1, \ldots, n-1$ aus Algorithmus 2.6. Löst man dann noch die Gleichung $\bar{l}_n^T\bar{l}_n + l_{n\,n}^2 = a_{n\,n}$, so ergibt sich auch die Gleichung für $j = n$ in Algorithmus 2.6. Dass diese Gleichung reell lösbar ist, folgt aus $\det(A) = \det(L)^2$ (wegen $A = LL^T$), $\det(A_{n-1}) = \det(L_{n-1})^2$ (wegen $A_{n-1} = L_{n-1}L_{n-1}^T$) und $\det(L)^2 = \det(L_{n-1})^2 l_{n\,n}^2$ (wegen der Form von L), womit

$$l_{n\,n}^2 = \det(L)^2 / \det(L_{n-1})^2 = \det(A)/\det(A_{n-1})$$

wegen der positiven Definitheit von A (und damit auch von A_{n-1}) reell und positiv ist.

Die Choleski-Zerlegung benötigt im ungünstigsten Fall

$$\frac{1}{3}n^3 + \frac{1}{2}n^2 + \frac{1}{6}n$$

elementare Rechenoperationen. Für große n, also wenn der n^3-Term dominant ist, ist dies nur etwa die Hälfte der Operationen der Gauß-Elimination. Ebenso wie bei der Gauß-Elimination reduziert sich die Anzahl der Operationen auf $\mathcal{O}(n)$ im Falle von Bandmatrizen.

2.4 Fehlerabschätzungen und Kondition

Wir haben bereits am einfachen Beispiel der Lösung von quadratischen Gleichungen gesehen, dass Rundungsfehler das Ergebnis stark verfälschen können. Um zu verstehen, warum und wann das passiert, ist der Begriff der *Kondition* nützlich. Die Kondition ist eine Eigenschaft eines mathematischebn Problems und hat zunächst gar nichts mit dem konkreten Lösungsalgorithmus zu tun. Wir werden aber am Ende dieses Abschnitts sehen, wie man mit diesem Konzept auch das Verhalten von Algorithmen analysieren kann.

Die Kondition eines mathematischen Problems ist eine Zahl, die angibt, wie sehr Fehler in den Eingabedaten das Ergebnis verfälschen können. Speziell wollen wir hier untersuchen, wie sich Fehler im Vektor b auf das Ergebnis x des Gleichungssystems auswirken. Nehmen wir also an, dass wir das Gleichungssystem

$$Ax = b$$

lösen wollen, der Vektor b aber durch einen Fehler Δb gestört wird. Tatsächlich lösen wir also das gestörte Gleichungssystem

$$A\tilde{x} = b + \Delta b$$

und erhalten die Lösung $\tilde{x} = x + \Delta x$. Betrachten wir die zugehörigen relativen Fehler

$$\delta x = \frac{\|\Delta x\|}{\|x\|}, \quad \delta b = \frac{\|\Delta b\|}{\|b\|}$$

so stellt sich die Frage, wie groß δx in Abhängigkeit von δb ist.

Um dies zu beantworten, benötigen wir das Konzept der induzierten Matrixnorm.

Definition 2.7 Sei $\mathbb{R}^{n\times n}$ die Menge der $n \times n$-Matrizen und sei $\|\cdot\|$ eine Vektornorm im $\mathbb{R}^n$. Dann definieren wir für $A \in \mathbb{R}^{n\times n}$ die zu $\|\cdot\|$ gehörige *induzierte Matrixnorm* $\|A\|$ als

$$\|A\| := \max_{\substack{x\in\mathbb{R}^n \\ \|x\|=1}} \|Ax\|.$$

□

Die für uns wichtige Eigenschaft der Matrixnorm ist die Ungleichung

$$\|Ax\| \leq \|A\|\,\|x\|,$$

die für alle Vektoren x gilt.

Wie berechnet man die Matrixnorm? Wenn die Vektornorm die übliche euklidische Norm $\|\cdot\|_2$ ist, ist $\|A\|_2$ relativ kompliziert zu berechnen, denn es gilt

$$\|A\|_2 = \sqrt{\rho(A^T A)}$$

wobei $\rho(A^T A)$ den maximalen Eigenwert der symmetrischen Matrix $A^T A$ bezeichnet. Diese Matrixnorm heißt auch *Spektralnorm*.

Einfacher ist die Berechnung von $\|A\|$, wenn wir als Vektornorm die 1-*Norm*

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

oder die *Maximums-* (oder ∞*)-Norm*

$$\|x\|_\infty = \max_{i=1,\ldots,n} |x_i|$$

verwenden. Dann gilt nämlich

$$\|A\|_1 = \max_{j=1,\ldots,n} \sum_{i=1}^{n} |a_{i\,j}|$$

bzw.

$$\|A\|_\infty = \max_{i=1,\ldots,n} \sum_{j=1}^{n} |a_{i\,j}|$$

weswegen diese Matrixnormen auch Spalten- bzw. Zeilensummennorm genannnt werden.

Allgemeiner können mithilfe der p-Normen für Vektoren

$$\|x\|_p := \sqrt[p]{\sum_{i=1}^{n} |x_i|^p}$$

aus p-Matrixnormen definiert werden. Alle diese verschiedenen Matrixnormen sind tatsächlich äquivalent in dem Sinne, dass für je zwei Normen $\|\cdot\|_p$, $\|\cdot\|_q$ immer Konstanten $C_2 \geq C_1 > 0$ existieren, sodass für alle Matrizen A gilt

$$C_1\|A\|_p \leq \|A\|_q \leq C_2\|A\|_p.$$

Betrachten wir nun die Fehlerterme, so gilt

$$A\Delta x = A(\tilde{x} - x) = A\tilde{x} - Ax = \tilde{b} - b = \Delta b.$$

Daraus folgt

$$\Delta x = A^{-1}\Delta b$$

und damit

$$\|\Delta x\| = \|A^{-1}\Delta b\| \leq \|A^{-1}\|\,\|\Delta b\|.$$

Andererseits gilt natürlich $b = Ax$ und damit

$$\|b\| = \|Ax\| \leq \|A\|\,\|x\| \quad \Rightarrow \quad \frac{1}{\|x\|} \leq \frac{\|A\|}{\|b\|}.$$

Damit erhalten wir

$$\delta x = \frac{\|\Delta x\|}{\|x\|} = \|\Delta x\|\frac{1}{\|x\|} \leq \|A^{-1}\|\,\|\Delta b\|\frac{\|A\|}{\|b\|} = \|A^{-1}\|\,\|A\|\delta b.$$

Diese zur Abschätzung des relativen Fehlers δx wesentliche Größe $\|A^{-1}\|\,\|A\|$ nennen wir die Kondition von A. Formal:

$$\text{cond}(A) := \|A\|\,\|A^{-1}\| \qquad \text{und damit} \qquad \delta x \leq \text{cond}(A)\,\delta b. \tag{2.9}$$

Die Kondition $\text{cond}(A)$ gibt also an, wie anfällig die Genauigkeit der Lösung x gegenüber Fehlern in b und damit typischerweise auch gegenüber Rundungsfehlern ist. Sie kann z. B. mit der MATLAB Anweisung `cond(A,p)` ausgerechnet werden, wobei `p=1,2,inf` für die zu verwendende Norm steht.

Mit Hilfe der Kondition können wir nun zeigen, dass der Gauß-Algorithmus unter Umständen nicht sehr robust gegenüber Rundungsfehlern ist. Wir veranschaulichen dies mit der Matrix

$$A = \begin{pmatrix} 0.0001 & 1 \\ 1 & 2 \end{pmatrix}.$$

Deren Kondition in der euklidischen Norm beträgt $\text{cond}_2(A) \approx 5.8297$, was bedeutet, dass die relativen Fehler in b im Ergebnis höchstens um diesen relativ kleinen Faktor verstärkt werden. Das ist unkritisch; man sagt, die Matrix ist *gut konditioniert*.

Der Gauß-Algorithmus transformiert diese Matrix in die Form

$$R = \begin{pmatrix} 0.0001 & 1 \\ 0 & -9998 \end{pmatrix}.$$

Diese Matrix besitzt nun die Kondition $\mathrm{cond}_2(R) \approx 10^7$, was bedeutet, dass der relative Fehler in b beim Lösen (also beim Rückwärtseinsetzen) um einen Faktor von etwa 10 Millionen(!) verstärkt werden kann! Eine solche Matrix nennt man *schlecht konditioniert*.

Zwar kann man dieses Problem der Konditionsverschlechterung im Gauß-Algorithmus nicht prinzipiell lösen, man kann es aber abmildern, indem man auch dann Zeilenvertauschungen durchführt, wenn das Diagonalelement nicht exakt gleich Null sondern nur nahe bei Null ist. Dies ist in unserem Beispiel gerade der Fall. Vertauschen wir hier zunächst die Zeilen von A zu

$$A' = \begin{pmatrix} 1 & 2 \\ 0.0001 & 1 \end{pmatrix}$$

und führen dann den Eliminationsschritt durch, so erhalten wir

$$R' = \begin{pmatrix} 1 & 2 \\ 0 & 0.9998 \end{pmatrix}.$$

Diese Matrix besitzt die gleiche Kondition wie das ursprüngliche A. Allerdings wird bei solchen Vertauschungen i. A. die eventuell vorhandene Bandstruktur einer Matrix zerstört, weswegen der Algorithmus für Bandmatrizen deutlich ineffizienter werden kann.

Die Idee des Zeilenvertauschens, auch wenn keine Null vorliegt, kann weiter verallgemeinert werden. Die numerische Stabilität des Gauß-Algorithmus kann erhöht werden, indem man bei jeder neuen Spalte das betragsmäßig größte Element der Spalte, das auf oder unter der Diagonalen liegt, auf die Diagonale tauscht, indem man die entsprechenden Zeilen in A und b tauscht. Als Algorithmus formuliert entspricht diese Idee:

Algorithmus 2.8 (Gauß-Elimination mit Pivotsuche)
Gegeben sei eine Matrix $A \in \mathbb{R}^{n\times n}$ und ein Vektor $b \in \mathbb{R}^n$.

(1) für j von 1 bis $n-1$ (Spaltenzähler)

(1*) $\ell := argmax_{\ell=j,\ldots,n}|a_{\ell\,j}|$ (mit Tie-Breaking)

(1**) vertausche $a_{j\,k}$ und $a_{\ell\,k}$, $k = j,\ldots,n$, sowie b_ℓ und b_j

(1***) für i von n bis $j+1$ (Zeilenzähler)

(2) falls $a_{i\,j} \neq 0$:

(3) subtrahiere $a_{i\,j}/a_{j\,j}$-mal die j-te Zeile von der i-ten Zeile,
d. h., führe (3a), (3b) und (3c) aus:

(3a) Setze $\alpha := a_{i\,j}/a_{j\,j}$ und $a_{i\,j} := 0$ und berechne

(3b) $a_{i\,k} := a_{i\,k} - \alpha\, a_{j\,k}$ für alle $k = j+1,\ldots,n$ mit $a_{j\,k} \neq 0$

(3c) $b_i := b_i - \alpha\, b_j$

(4) Ende der i-Schleife
Ende der j-Schleife

□

Dieses Verfahren wird auch Spaltenpivotsuche genannt. Tie-Breaking bedeutet, dass man im Falle eines nichteindeutigen Minimierers einen auswählen muss. Zum Beispiel könnte man den mit dem höchsten Index benutzen.

Eine andere Methode, die Verschlechterung der Kondition zu vermeiden liegt darin, die LR-Zerlegung durch eine ganz andere Methode zu ersetzen, nämlich durch die QR-Zerlegung, wobei Q eine orthogonale Matrix, also eine Matrix mit $Q^TQ = 1$ ist. Für diese Methode kann man leicht zeigen, dass $\mathrm{cond}_2(Q) = 1$ und $\mathrm{cond}_2(R) = \mathrm{cond}_2(A)$ gilt (in der euklidischen Norm). Die Kondition kann sich also nicht verschlechtern. Allerdings benötigen Algorithmen für eine solche Zerlegung mehr Rechenoperationen als die Gauß-Elimination und auch hier bleibt eine eventuelle Bandstruktur von A i. A. nicht erhalten.

2.5 Iterative Verfahren

Wir haben im letzten Abschnitt gesehen, dass die bisher betrachteten Verfahren – die sogenannten *direkten Verfahren* – für Matrizen ohne besondere Struktur die Ordnung $\mathcal{O}(n^3)$ besitzen: Wenn sich also n verzehnfacht, so vertausendfacht sich die Anzahl der Operationen und damit die Rechenzeit. Für große Gleichungssysteme mit mehreren 100 000 Unbekannten, die in der Praxis durchaus auftreten, führt dies zu unakzeptabel hohen Rechenzeiten.

Eine alternative Klasse von Verfahren, die oft mit weniger Rechenaufwand eine Lösung liefern, sind die *iterativen Verfahren*. Allerdings zahlt man für den geringeren Aufwand einen Preis: Man kann bei diesen Verfahren nicht mehr erwarten, eine (bis auf Rundungsfehler) exakte Lösung zu erhalten, sondern muss von vornherein eine gewisse Ungenauigkeit im Ergebnis in Kauf nehmen.

Das Grundprinzip iterativer Verfahren funktioniert dabei wie folgt:

Ausgehend von einem Startvektor $x^{(0)}$ berechnet man mittels einer Rechenvorschrift $\Phi : \mathbb{R}^n \to \mathbb{R}^n$ iterativ eine Folge von Vektoren

$$x^{(i+1)} = \Phi(x^{(i)}), \qquad i = 0, 1, 2, \ldots,$$

die für $i \to \infty$ gegen die Lösung x^* des Gleichungssystems $Ax = b$ konvergieren, also $\lim_{i\to\infty} \|x^{(i)} - x^*\| = 0$. Wenn die gewünschte Genauigkeit erreicht ist, wird die Iteration abgebrochen und der letzte Wert $x^{(i)}$ als Näherung des Ergebnisses verwendet.

2.5.1 Gauß-Seidel- und Jacobi-Verfahren

Wir wollen nun zwei klassische iterative Verfahren kennen lernen, die beide nach dem gleichen Prinzip funktionieren: Man zerlegt die Matrix A in eine Differenz zweier Matrizen

$$A = M - N,$$

wobei M^{-1} leicht (d. h. mit sehr wenig Aufwand) zu berechnen ist. Dann wählt man einen Startvektor $x^{(0)}$ (z. B. den Nullvektor) und berechnet iterativ

$$x^{(i+1)} = M^{-1}Nx^{(i)} + M^{-1}b, \quad i = 0, 1, 2, \ldots. \tag{2.10}$$

Wenn die Zerlegung (unter passenden Annahmen an A) geeignet gewählt wurde, kann man erwarten, dass die Vektoren x_i gegen die gesuchte Lösung konvergieren.

Formal versucht man dazu, eine Norm $\|\cdot\|$ zu finden, sodass die Abschätzung

$$k = \|M^{-1}N\| < 1 \tag{2.11}$$

gilt. Dann kann man mit Hilfe des Banach'schen Fixpunktsatzes aus der Analysis und der geometrischen Reihe die Abschätzungen

$$\|x^{(i)} - x^*\| \leq \frac{k}{1-k}\|x^{(i)} - x^{(i-1)}\| \leq \frac{k^i}{1-k}\|x^{(1)} - x^{(0)}\| \tag{2.12}$$

beweisen.

Bei iterativen Algorithmen brauchen wir noch ein Abbruchkriterium, um zu entscheiden, wann wir die Iteration stoppen. Hier gibt es mehrere Möglichkeiten; ein einfaches aber trotzdem effizientes Kriterium ist es, sich ein $\varepsilon > 0$ vorzugeben, und die Iteration dann abzubrechen, wenn die Bedingung

$$\|x^{(i+1)} - x^{(i)}\| < \varepsilon \tag{2.13}$$

für eine vorgegebene Norm $\|\cdot\|$ erfüllt ist. Wenn wir hier die Norm aus (2.12) nehmen, so ist mit diesem Kriterium die Genauigkeit

$$\|x^{(i+1)} - x^*\| \leq \frac{k}{1-k}\varepsilon$$

gewährleistet. Analog kann man hier auch den relativen Fehler

$$\frac{\|x^{(i+1)} - x^{(i)}\|}{\|x^{(i+1)}\|} < \varepsilon'$$

verwenden. Will man bis zum Erreichen der maximal möglichen Rechengenauigkeit iterieren, so wählt man im relativen Abbruchkriterium ε gleich der Maschinengenauigkeit.

Beispiel 2.9 Wir illustrieren ein solches Verfahren an dem dreidimensionalen linearen Gleichungssystem mit

$$A = \begin{pmatrix} 15 & 3 & 4 \\ 2 & 17 & 3 \\ 2 & 3 & 21 \end{pmatrix} \quad \text{und} \quad b = \begin{pmatrix} 33 \\ 45 \\ 71 \end{pmatrix}.$$

Als Zerlegung $A = M - N$ wählen wir

$$M = \begin{pmatrix} 15 & 0 & 0 \\ 0 & 17 & 0 \\ 0 & 0 & 21 \end{pmatrix} \quad \text{und } N = -\begin{pmatrix} 0 & 3 & 4 \\ 2 & 0 & 3 \\ 2 & 3 & 0 \end{pmatrix},$$

d. h., wir zerlegen A in ihren Diagonalanteil M und den Nicht-Diagonalanteil $-N$. Diagonalmatrizen sind sehr leicht zu invertieren: Man muss einfach jeden Diagonaleintrag durch seinen Kehrwert ersetzen, also

$$M^{-1} = \begin{pmatrix} 1/15 & 0 & 0 \\ 0 & 1/17 & 0 \\ 0 & 0 & 1/21 \end{pmatrix}.$$

Damit erhalten wir

$$M^{-1}N = \begin{pmatrix} 0 & -1/5 & -4/15 \\ -2/17 & 0 & -3/17 \\ -2/21 & -1/7 & 0 \end{pmatrix} \quad \text{und} \quad M^{-1}b = \begin{pmatrix} 11/5 \\ 45/17 \\ 71/21 \end{pmatrix}.$$

Offenbar erfüllt die Matrix $M^{-1}N$ die Bedingung 2.11 sowohl in der Zeilen- als auch in der Spaltensummennorm.

Wir berechnen nun gemäß der Vorschrift (2.10) die Vektoren $x^{(1)}, \ldots, x^{(10)}$, wobei wir $x^{(0)} = (0\,0\,0)^T$ setzen. Es ergeben sich (jeweils auf vier Nachkommastellen gerundet)

$$\begin{pmatrix} 2.2000 \\ 2.6471 \\ 3.3810 \end{pmatrix}, \begin{pmatrix} 0.7690 \\ 1.7916 \\ 2.7933 \end{pmatrix}, \begin{pmatrix} 1.0968 \\ 2.0637 \\ 3.0518 \end{pmatrix}, \begin{pmatrix} 0.9735 \\ 1.9795 \\ 2.9817 \end{pmatrix}, \begin{pmatrix} 1.0090 \\ 2.0064 \\ 3.0055 \end{pmatrix},$$

$$\begin{pmatrix} 0.9973 \\ 1.9980 \\ 2.9982 \end{pmatrix}, \begin{pmatrix} 1.0009 \\ 2.0006 \\ 3.0005 \end{pmatrix}, \begin{pmatrix} 0.9997 \\ 1.9998 \\ 2.9998 \end{pmatrix}, \begin{pmatrix} 1.0001 \\ 2.0001 \\ 3.0001 \end{pmatrix}, \begin{pmatrix} 1.0000 \\ 2.0000 \\ 3.0000 \end{pmatrix}.$$

□

Je nach Wahl von M und N erhält man verschiedene Verfahren. Hier wollen wir zwei Verfahren genauer beschreiben und die Iteration (2.10) nicht mit Matrix-Multiplikationen sondern ausführlich für die Einträge $x_j^{(i+1)}$ der Vektoren $x^{(i+1)}$ aufschreiben, sodass die Verfahren dann direkt implementierbar sind. Das erste Verfahren ist das, welches wir auch im Beispiel 2.9 verwendet haben.

Algorithmus 2.10 (Jacobi-Verfahren oder **Gesamtschrittverfahren)**
Wir wählen M als Diagonalmatrix

$$M = \begin{pmatrix} a_{1\,1} & 0 & \ldots & 0 \\ 0 & a_{2\,2} & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \ldots & 0 & a_{n\,n} \end{pmatrix}$$

und $N = M - A$. Dann ergibt sich (2.10) zu

$$x_j^{(i+1)} = \frac{1}{a_{j\,j}} \left(b_j - \sum_{\substack{k=1 \\ k \neq j}}^{n} a_{j\,k} x_k^{(i)} \right), \quad \text{für } j = 1, \ldots, n.$$

□

Eine etwas andere Zerlegung führt zu dem folgenden Verfahren.

Algorithmus 2.11 (Gauß-Seidel-Verfahren oder **Einzelschrittverfahren)**
Wir wählen M als untere Dreiecksmatrix

$$M = \begin{pmatrix} a_{1\,1} & 0 & \dots & 0 \\ a_{2\,1} & a_{2\,2} & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ a_{n\,1} & \dots & a_{n\,n-1} & a_{n\,n} \end{pmatrix}$$

und $N = M - A$. Dieses M lässt sich zwar nicht direkt invertieren, wir können (2.10) aber von links mit M multiplizieren und erhalten so für $x^{(i+1)}$ das lineare Gleichungssystem

$$Mx^{(i+1)} = Nx^{(i)} + b.$$

Daraus können wir die Komponenten $x_j^{(i+1)}$ mittels Vorwärtseinsetzen bestimmen und erhalten so

$$x_j^{(i+1)} = \frac{1}{a_{j\,j}} \left(b_j - \sum_{k=1}^{j-1} a_{j\,k} x_k^{(i+1)} - \sum_{k=j+1}^{n} a_{j\,k} x_k^{(i)} \right), \quad \text{für } j = 1, \dots, n.$$

□

Beide Verfahren funktionieren, wenn die Matrix A strikt diagonaldominant ist, was bedeutet, dass die Ungleichung

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

für alle $i = 1, \dots, n$ erfüllt. In diesem Fall ist (2.11) für die Zeilensummennorm $\| \cdot \|_\infty$ mit

$$k = \max_{i=1,\dots,n} \left(\sum_{\substack{j=1 \\ j \neq i}}^{n} \frac{|a_{i\,j}|}{|a_{i\,i}|} \right) < 1$$

erfüllt und es folgt (2.12).

Das Gauß-Seidel Verfahren konvergiert zudem für symmetrische, positiv definite Matrizen. In diesem Fall ist (2.11) und damit auch (2.12) für die Spektralnorm erfüllt.

Für die Aufwandsabschätzung nehmen wir der Einfachheit halber an, dass die Anzahl der Iterationen bis zum Erreichen einer gewünschten Genauigkeit nicht von der Dimension n des Problems abhängt. Dann ist der Aufwand proportional zum Aufwand eines Iterationsschrittes. Dieser liegt für voll besetzte Matrizen bei $\mathcal{O}(n^2)$ und für Matrizen mit einer von n unabhängigen Anzahl von Einträgen $\neq 0$ pro Zeile bei $\mathcal{O}(n)$. In diesem Fall liegt der große Vorteil dieser iterativen Verfahren gegenüber den direkten Verfahren wie z. B. der Gauß-Elimination darin, dass die iterativen Verfahren keine Bandstruktur von A benötigen, um diese bessere Ordnung $\mathcal{O}(n)$ zu erzielen.

2.5.2 Das konjugierte Gradientenverfahren

Die bisherigen Verfahren basieren alle auf einer additiven Zerlegung der Matrix A. Eine weitere Klasse iterativer Verfahren folgt einer ganz anderen Idee, sie basieren nämlich auf Optimierungsmethoden. Zum Abschluss dieses Abschnitts wollen wir einen einfachen Vertreter dieser Klasse, das *konjugierte Gradientenverfahren* oder *CG-Verfahren* (CG=„conjugate gradient“), kurz betrachten. Dies ist wiederum ein Verfahren für symmetrische und positiv definite Matrizen A.[2]

Statt das Gleichungssystem $Ax = b$ zu lösen, löst man das Minimierungsproblem

$$\text{minimiere} \quad f(x) = \frac{1}{2}x^T Ax - b^T x.$$

Für eine Lösung dieses Minimierungsproblems gilt

$$0 = \nabla f(x) = Ax - b,$$

weswegen dies eine Lösung des ursprünglichen linearen Gleichungssystems liefert. Das CG-Verfahren ist eigentlich ein direktes Verfahren, da es (zumindest in der Theorie, also ohne Rundungsfehler) nach endlich vielen Schritten ein exaktes Ergebnis liefert. Trotzdem zählt man es zu den iterativen Verfahren, da die Zwischenergebnisse des Verfahrens bereits Näherungslösungen darstellen, so dass man das Verfahren in der Praxis vor dem Erreichen der exakten Lösung abbricht. Die Näherung $x^{(i+1)}$ wird hierbei aus der vorhergehenden bestimmt, indem eine Suchrichtung $d^{(i)} \in \mathbb{R}^n$ und eine Schrittweite $\alpha^{(i)} \in \mathbb{R}$ ermittelt wird, und dann

$$x^{(i+1)} = x^{(i)} + \alpha^{(i)} d^{(i)}$$

gesetzt wird, wobei $d^{(i)}$ und $\alpha^{(i)}$ so gewählt werden, dass $f(x^{(i+1)})$ möglichst klein wird und $f(x^{(i+1)}) < f(x^{(i)})$ gilt.

Zur Wahl der Schrittweite: Für eine gegebene Suchrichtung $d^{(i)}$ soll die Schrittweite $\alpha^{(i)}$ so gewählt werden, dass $h(\alpha) = f(x^{(i)} + \alpha d^{(i)})$ minimal wird. Dies ist ein eindimensionales Optimierungsproblem, da h eine Abbildung von $\mathbb{R}$ nach $\mathbb{R}$ ist. Bedingt durch die Struktur von h bzw. f kann man ausrechnen, dass das Minimum für

$$\alpha^{(i)} = \frac{(b - Ax^{(i)})^T d^{(i)}}{{d^{(i)}}^T A d^{(i)}}$$

angenommen wird.

Zur Wahl der Suchrichtung: Die Suchrichtung wird in verschiedenen Schritten auf unterschiedliche Weise gewählt. Im ersten Schritt wird die Richtung des steilsten Abstiegs verwendet. Da der Gradient ∇f in Richtung des steilsten Anstiegs zeigt, wählt man

$$d^{(0)} = -\nabla f(x^{(0)}) = -(Ax^{(0)} - b) = b - Ax^{(0)}.$$

Die weiteren Suchrichtungen für $i \geq 1$ werden nun so gewählt, dass sie bzgl. des Skalarproduktes $\langle x, y \rangle_A = x^T Ay$ orthogonal zu der vorhergehenden Richtung liegen, womit

[2] Ähnliche Verfahren für allgemeine Matrizen existieren ebenfalls, z. B. das CGS- oder das BiCGstab-Verfahren, sind aber komplizierter.

sichergestellt ist, dass „gleichmäßig“ in alle Richtungen des $\mathbb{R}^n$ gesucht wird. Formal wählt man dazu $d^{(i)}$ so, dass

$$\langle d^{(i)}, d^{(i-1)} \rangle_A = 0$$

ist. Zusätzlich zu dieser Bedingung (die von vielen Vektoren $d^{(i)}$ erfüllt ist) wird der Ansatz

$$d^{(i)} = r^{(i)} + \beta^{(i)} d^{(i-1)} \quad \text{mit} \quad r^{(i)} = -\nabla f(x^{(i)}) = b - Ax^{(i)}$$

gemacht. Wir gehen also in Richtung des negativen Gradienten (der zugleich das Residuum des linearen Gleichungssystems ist), modifizieren diese Richtung aber durch Addition von $\beta_i d^{(i-1)}$. Diese spezielle Korrektur erlaubt eine einfache Berechnung von $\beta^{(i)}$ als

$$\beta^{(i)} = -\frac{{r^{(i)}}^T A d^{(i-1)}}{{d^{(i-1)}}^T A d^{(i-1)}}.$$

Der so konstruierte Vektor $d^{(i)}$ steht tatsächlich auf allen vorhergehenden Suchrichtungen $d^{(0)}, \ldots, d^{(i-1)}$ senkrecht, was nicht direkt zu sehen ist, aber mit etwas Aufwand bewiesen werden kann. Beachte, dass in all diesen Berechnungen der Nenner der auftretenden Brüche ungleich Null ist, da A positiv definit ist.

Man kann beweisen, dass das Verfahren (in der Theorie, also ohne Rundungsfehler) nach spätestens n Schritten eine exakte Lösung des Problems findet. Bei großem n wird man die Iteration typischerweise bereits früher, d. h. nach Erreichen einer vorgegebenen Genauigkeit, abbrechen wollen, was möglich ist, da die Folge $x^{(i)}$ schon während des Iterationsprozesses gegen x^* konvergiert. Als Abbruchkriterium wird hier üblicherweise die Ungleichung $\|r^{(i)}\| \leq \varepsilon$ verwendet. Wegen $r^{(i)} = b - Ax^{(i)}$ gilt

$$Ax^{(i)} = b - r^{(i)},$$

sodass wir mittels (2.9) mit $\Delta b = -r^{(i)}$ die Abschätzung

$$\delta x = \frac{\|x^{(i)} - x^*\|}{\|x^*\|} \leq \operatorname{cond}(A) \cdot \frac{\|r^{(i)}\|}{\|b\|}$$

für den relativen Fehler erhalten.

Kapitel 3

Modellprojekt 1: Die LKW-Kabine

Informationen zu diesem Modellprojekt erhalten Sie in der Vorlesung oder in dem Buch [Bollhöfer und Mehrmann, 2004, Kapitel 2.1].

Kapitel 4

Interpolation

Die Interpolation von Funktionen oder Daten ist ein häufig auftretendes Problem sowohl in der Mathematik als auch in vielen Anwendungen.

Das allgemeine Problem, die sogenannte *Dateninterpolation*, entsteht, wenn wir eine Menge von Daten (x_i, f_i) für $i = 0, \ldots, n$ gegeben haben (z. B. Messwerte eines Experiments). Die Problemstellung ist nun wie folgt: Gesucht ist eine Funktion F, für die die Gleichung

$$F(x_i) = f_i \quad \text{für } i = 0, 1, \ldots, n \tag{4.1}$$

gilt.

Ein Beispiel für so ein Problem ist die Straßenoberfläche im Modellprojekt 1, für die wir aus den Messdaten eine Funktion gewinnen wollen. Ein weiteres Beispiel ist die Computergrafik, bei der z. B. in Zeichenprogrammen vorgegebene Punkte durch eine Kurve verbunden werden sollen.

Ein Spezialfall dieses Problems ist die *Funktionsinterpolation*: Nehmen wir an, dass wir eine reellwertige Funktion $f : \mathbb{R} \to \mathbb{R}, x \mapsto f(x)$ gegeben haben, die aber (z. B. weil keine explizite Formel bekant ist) sehr kompliziert auszuwerten ist. Das Ziel der Interpolation liegt nun darin, eine Funktion F zu bestimmen, die leicht auszuwerten ist, und die für vorgegebene *Stützstellen* $x_0, x_1, \ldots, x_n$ die Gleichungen

$$F(x_i) = f(x_i) \quad \text{für } i = 0, 1, \ldots, n \tag{4.2}$$

erfüllt. Mit der Schreibweise

$$f_i = f(x_i),$$

erhalten wir hier wieder die Bedingung (4.1), weswegen (4.2) tatsächlich ein Spezialfall von (4.1) ist. Wir werden in diesem Kapitel Verfahren zur Lösung von (4.1) entwickeln, die dann selbstverständlich auch auf den Spezialfall (4.2) anwendbar sind.

4.1 Polynominterpolation

Die grundlegende Methode zur Interpolation ist die Wahl von F als Polynom, also als Funktion der Form

$$P(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_m x^m. \tag{4.3}$$

Hierbei werden die Werte a_i, $i = 0, \ldots, m$, die *Koeffizienten* des Polynoms genannt. Die höchste auftretende Potenz (hier m) heißt der *Grad* des Polynoms, falls $a_m \neq 0$. Um zu betonen, dass wir hier Polynome verwenden, schreiben wir in diesem Abschnitt „P“ statt „F“ für die Interpolationsfunktion.

Das Problem der Polynominterpolation liegt nun darin, die Koeffizienten $a_0, a_1, \ldots, a_m$ so zu bestimmen, dass (4.1) erfüllt ist. Zunächst einmal müssen wir uns dazu überlegen, welchen Grad das gesuchte Polynom haben soll. Hier hilft uns der folgende Satz.

Satz 4.1 Sei $n \in \mathbb{N}$ und seien Daten $(x_i; f_i)$ für $i = 0, \ldots, n$ gegeben, sodass für die Stützstellen $x_i \neq x_j$ gilt für alle $i \neq j$. Dann gibt es genau ein Polynom vom Grad $m \leq n$, das die Bedingung

$$P(x_i) = f_i \quad \text{für } i = 0, 1, \ldots, n$$

erfüllt ist.

Eine anschauliche Begründung dafür ist, dass wir im Polynom gerade $n+1$ zu bestimmende Koeffizienten $a_0, a_1, \ldots, a_n$ haben und deswegen $n+1$ Datenpunkte (mit verschiedenen x_i) brauchen, um diese zu bestimmen.

Der formale Beweis geht so: Man überlegt sich, dass die Koeffizienten a_i durch das lineare Gleichungssystem

$$\begin{pmatrix} 1 & x_0 & \ldots & x_0^n \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & \ldots & x_n^n \end{pmatrix} \begin{pmatrix} a_0 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} f_0 \\ \vdots \\ f_n \end{pmatrix}. \tag{4.4}$$

bestimmt sind. Da die Determinante dieser Matrix gerade

$$\prod_{i=1}^{n} \left(\prod_{j=i+1}^{n} (x_i - x_j) \right) \neq 0$$

ist, falls die x_i paarweise verschieden sind, ist das Gleichungssystem eindeutig lösbar. □

Für $n+1$ gegebene Datenpunkte $(x_i; f_i)$ „passt“ also gerade ein Polynom vom Grad n. Wie berechnet man dies?

4.1.1 „Naive“ Polynominterpolation

Eine simple Art, ein Interpolationspolynom zu bestimmen, ist die, das lineare Gleichungssystem (4.4) zu lösen. Das Lösen eines solchen LGS hat grundsätzlich Ordnung $\mathcal{O}(n^3)$, die Auswertung des Polynoms für ein $x(\neq x_i \ \forall i)$ $n-1$ viele Potenzbildungen, n Multiplikationen und n Additionen, also $3n - 1 = \mathcal{O}(n)$. Wir werden aber sehen, dass es effizientere Möglichkeiten gibt, Polynominterpolationen zu gebrauchen.

4.1.2 Lagrange–Polynome

Die Idee der Lagrange–Polynome beruht auf einer geschickten Schreibweise für ein Polynom. Für die vorgegebenen Stützstellen $x_0, x_1 \ldots, x_n$ definieren wir für $i = 0, \ldots, n$ die *Lagrange–Polynome* L_i als

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

Man rechnet leicht nach, dass diese Polynome alle vom Grad n sind, und darüberhinaus die Gleichung

$$L_i(x_k) = \begin{cases} 1 & \text{für } i = k \\ 0 & \text{für } i \neq k \end{cases}$$

erfüllen. Unser gesuchtes Polynom ist damit gegeben durch

$$P(x) = \sum_{i=0}^{n} f_i L_i(x),$$

denn es gilt

$$P(x_k) = \sum_{i=0}^{n} \underbrace{f_i L_i(x_k)}_{\substack{=0 \text{ falls } i \neq k \\ = f_k \text{ falls } i = k}} = f_k,$$

also gerade die gewünschte Bedingung (4.1).

Beispiel 4.2 Betrachte die Daten (3; 68), (2; 16), (5; 352). Die zugehörigen Lagrange–Polynome sind gegeben durch

$$L_0(x) = \frac{x-2}{3-2}\,\frac{x-5}{3-5} = -\frac{1}{2}(x-2)(x-5),$$

$$L_1(x) = \frac{x-3}{2-3}\,\frac{x-5}{2-5} = \frac{1}{3}(x-3)(x-5),$$

$$L_2(x) = \frac{x-2}{5-2}\,\frac{x-3}{5-3} = \frac{1}{6}(x-2)(x-3).$$

Damit erhalten wir

$$P(x) = 68 \cdot \left(-\tfrac{1}{2}\right)(x-2)(x-5) + 16 \cdot \tfrac{1}{3}(x-3)(x-5) + 352 \cdot \tfrac{1}{6}(x-2)(x-3).$$

Für $x = 3$ ergibt sich $P(3) = 68 \cdot \left(-\frac{1}{2}\right)(3-2)(3-5) = 68$, für $x = 2$ berechnet man $P(2) = 16 \cdot \frac{1}{3}(2-3)(2-5) = 16$ und für $x = 5$ erhalten wir $P(5) = 352 \cdot \frac{1}{6}(5-2)(5-3) = 352$. □

4.1.3 Auswertung mittels Baryzentrischer Koordinaten

Durch Abzählen der notwendigen elementaren Rechenoperationen sieht man, dass die Auswertung des Polynoms P in der oben hergeleiteten Form gerade $\mathcal{O}(n^2)$ elementare Rechenoperationen benötigt (wir haben $n+1$ Summanden, in jedem Summanden muss $L_i(x)$ ausgewertet werden, was $2n$ Subtraktionen, n Divisionen und $n-1$ Multiplikationen benötigt, zudem muss in jedem Summanden eine weitere Multiplikation $f_i L_i(x)$ ausgeführt werden, was pro Summand $4n$ Operationen und damit insgesamt auf $(n+1)4n = 4n^2+4n = \mathcal{O}(n^2)$ Operationen führt).

Will man ein Polynom P für große Stützstellenzahl n – z. B. zum Erstellen eines Plots – an vielen Stellen x auswerten, kann dies bereits recht lange dauern. Wie betrachten deswegen eine alternative Darstellung, die eine schnellere Auswertung ermöglicht.

Dazu schreiben wir den Zähler von

$$L_i(x) = \prod_{\substack{j=0 \\ j\neq i}}^{n} \frac{x - x_j}{x_i - x_j}.$$

als

$$\frac{\ell(x)}{x - x_i} \quad \text{mit} \quad \ell(x) := \prod_{j=0}^{n} (x - x_j).$$

Der Nenner schreiben wir mittels der sogenannten *baryzentrischen Koordinaten*

$$w_i := \prod_{\substack{j=0 \\ j\neq i}}^{n} \frac{1}{x_i - x_j}.$$

Dann gilt

$$L_i(x) = \ell(x) \cdot \frac{w_i}{x - x_i}$$

und damit

$$P(x) = \sum_{i=0}^{n} L_i(x) f_i = \sum_{i=0}^{n} \ell(x) \cdot \frac{w_i}{x - x_i} \cdot f_i = \ell(x) \sum_{i=0}^{n} \frac{w_i}{x - x_i} \cdot f_i.$$

Beispiel 4.3 Betrachte wiederum die Daten (3; 68), (2; 16), (5; 352). Das zugehörige ℓ ist gegeben durch

$$\ell(x) = (x-2)(x-3)(x-5)$$

und die w_i berechnen sich zu

$$w_0 = \frac{1}{3-2}\,\frac{1}{3-5} = -\frac{1}{2},$$

$$w_1 = \frac{1}{2-3}\,\frac{1}{2-5} = \frac{1}{3},$$

$$w_3 = \frac{1}{5-2}\,\frac{1}{5-3} = \frac{1}{6}.$$

Damit erhalten wir

$$\begin{aligned} P(x) &= \ell(x)\left(\frac{-\frac{1}{2}}{x-3}\cdot 68 + \frac{\frac{1}{3}}{x-2}\cdot 16 + \frac{\frac{1}{6}}{x-5}\cdot 352\right) \\ &= -\frac{1}{2}(x-2)(x-5)\cdot 68 + \frac{1}{3}(x-3)(x-5)\cdot 16 + \frac{1}{6}(x-2)(x-3)\cdot 352, \end{aligned}$$

also – wie zu erwarten – das gleiche Polynom wie oben. □

Wie viele Operationen benötigt diese Darstellung? Um dies zu berechnen, teilen wir die Berechnung in zwei Algorithmen auf.

Algorithmus 4.4 (Berechnung der baryzentrischen Koordinaten)
Eingabe: Stützstellen $x_0,\ldots,x_n$

(1) für i von 0 bis n:

(2) setze $w_i := 1$

(3) für j von 0 bis n:

(4) falls $j \neq i$, setze $w_i := w_i/(x_i - x_j)$

(5) Ende der Schleifen

Ausgabe: baryzentrische Koordinaten $w_0,\ldots,w_n$ □

Durch Abzählen der Operationen sieht man leicht, dass die Berechnung der w_i gerade $2(n+1)n = 2n^2 + 2n = \mathcal{O}(n^2)$ Operationen benötigt. Dies entspricht der Ordnung des Aufwandes der Auswertung von P im vorhergehenden Abschnitt. Der Trick liegt nun aber darin, die w_i einmal vorab zu berechnen und die gespeicherten Werte in der Auswertung von P zu verwenden.

Algorithmus 4.5 (Auswertung des Interpolationspolynoms)
Eingabe: Stützstellen $x_0,\ldots,x_n$, Werte $f_0,\ldots,f_n$, baryzentrische Koordinaten $w_0,\ldots,w_n$, Auswertungsstelle x

(0) setze $l := 1$, $s := 0$ (Variablen für ℓ und $\sum_{i=0}^{n} \frac{w_i}{x-x_i} f_i$)

(1) für i von 0 bis n

(2) setze $y := x - x_i$

(3) falls $y = 0$ ist, setze $P := f_i$ und beende den Algorithmus

(4) setze $l := l * y$

(5) setze $s := s + w_i * f_i/y$

(6) Ende der Schleife

(7) Setze $P := l * s$

Ausgabe: Polynomwert $P = P(x)$ □

Durch Abzählen der Operationen sieht man: die Auswertung benötigt gerade $5(n+1)+1 = 5n+6 = \mathcal{O}(n)$ Operationen. Sind also die w_i einmal berechnet, so ist dies deutlich weniger aufwändig als die direkte Auswertung von P.

4.1.4 Fehlerabschätzungen

Wir betrachten in diesem Abschnitt das Problem der Funktionsinterpolation (4.2) für eine gegebene Funktion $f : \mathbb{R} \to \mathbb{R}$. Wir wollen abschätzen, wie groß der Abstand des interpolierenden Polynoms P von der Funktion f ist. Hierbei bezeichnen wir mit I das von den Stützstellen $x_0, \ldots, x_n$ definierte Intervall $I = [\min\limits_{i=0,\ldots,n} x_i, \max\limits_{i=0,\ldots,n} x_i]$.

Den Abstand zwischen P und f definieren wir nun punktweise als

$$|f(x) - P(x)| \quad \text{für } x \in I.$$

Wir betrachten nur $x \in I$, da $x \notin I$ keine Inter-, sondern eine Extrapolation wäre, die i. A. nicht sinnvoll ist. Der folgende Satz gibt eine Abschätzung für den Abstand in Abhängigkeit von den Stützstellen an. Hierbei bezeichnet $f^{(k)}$ die *k–te Ableitung* der Funktion f. Das Ausrufezeichen „!“ bezeichnet die übliche *Fakultät* für natürliche Zahlen k, d. h. $k! = 1 \cdot 2 \cdot 3 \cdots k$.

Satz 4.6 Sei f $(n+1)$–fach stetig differenzierbar und sei P das Interpolationspolynom zu den paarweise verschiedenen Stützstellen $x_0, \ldots, x_n$. Dann gelten die folgenden Ausagen.

(i) Für alle $x \in I$ gibt es ein $\xi \in I$, sodass die Gleichung

$$f(x) - P(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!} \cdot (x - x_0)(x - x_1) \cdots (x - x_n)$$

gilt.

(ii) Es gilt die Abschätzung

$$|f(x) - P(x)| \leq \max_{y \in I} \left| \frac{f^{(n+1)}(y)}{(n+1)!} \cdot (x - x_0)(x - x_1) \cdots (x - x_n) \right|.$$

(iii) Wenn wir das Intervall I als $I = [a, b]$ schreiben, gilt die Abschätzung

$$|f(x) - P(x)| \leq \max_{y \in I} \left| f^{(n+1)}(y) \right| \cdot \frac{(b-a)^{n+1}}{(n+1)!}.$$

Beweis: (i) Wähle ein $x \in I$ und setze

$$c(x) = \frac{f(x) - P(x)}{(x - x_0) \cdots (x - x_n)}.$$

Mit diesem c definieren wir die Funktion

$$h(y) := f(y) - P(y) - c(x) \cdot (y - x_0) \cdots (y - x_n).$$

Dann ist $h(y)$ wieder $(n+1)$ mal stetig differenzierbar und hat (mindestens) die $n+2$ Nullstellen $y = x_0, \ldots, x_n$ und $y = x$. Wir nummerieren diese Nullstellen aufsteigend mit der Bezeichnung $y_0^{(0)} < y_1^{(0)} < \ldots < y_{n+1}^{(0)}$. Nach dem Satz von Rolle aus der Analysis

gilt: Zwischen je zwei Nullstellen der Funktion $h(y)$ liegt (mindestens) eine Nullstelle ihrer Ableitung $h'(y)$ (hier wie im Folgenden leiten wir nach der Variablen „y“ ab). Also liegt für jedes $i = 0, \ldots, n$ zwischen den Werten $y_i^{(0)}$ und $y_{i+1}^{(0)}$ ein Wert $y_i^{(1)}$ mit $h'(y_i^{(1)}) = 0$, und wir erhalten $n+1$ Nullstellen $y_0^{(1)} < y_1^{(1)} < \ldots < y_n^{(1)}$ für die Ableitung $h'(y) = h^{(1)}(y)$. Indem wir induktiv fortfahren, erhalten wir $n+2-k$ Nullstellen $y_0^{(k)} < y_1^{(k)} < \ldots < y_{n+1-k}^{(k)}$ für die k-te Ableitung $h^{(k)}$ und damit für $k = n+1$ eine Nullstelle $\xi = y_0^{(n+1)}$ für die Funktion $h^{(n+1)}$.

Aus den Ableitungsregeln für Polynome folgt, dass $P^{(n+1)}(y) = 0$ und $[(y-x_0)\cdots(y-x_n)]^{(n+1)} = (n+1)!$ ist für alle $y \in \mathbb{R}$. Damit erhalten wir

$$0 = h^{(n+1)}(\xi) = f^{(n+1)}(\xi) - c(x)\cdot(n+1)! = f^{(n+1)}(\xi) - \frac{f(x)-P(x)}{(x-x_0)\cdots(x-x_n)}\cdot(n+1)!,$$

also

$$f^{(n+1)}(\xi) = \frac{f(x)-P(x)}{(x-x_0)\cdots(x-x_n)}\cdot(n+1)!.$$

Auflösen nach $f(x)-P(x)$ liefert die Gleichung in (i).

(ii) Diese Abschätzung folgt aus (i) wegen

$$\begin{aligned} |f(x)-P(x)| &= \left|\frac{f^{(n+1)}(\xi)}{(n+1)!}\cdot(x-x_0)(x-x_1)\cdots(x-x_n)\right| \\ &\leq \max_{y\in I}\left|\frac{f^{(n+1)}(y)}{(n+1)!}\cdot(x-x_0)(x-x_1)\cdots(x-x_n)\right|, \end{aligned}$$

da ξ aus I ist.

(iii) Wenn wir das Intervall I als $I = [a,b]$ schreiben, folgt für alle x und x_i aus I die Abschätzung

$$|x - x_i| \leq b - a.$$

Damit erhalten wir aus Satz 4.6 (ii) die Abschätzung

$$|f(x)-P(x)| \leq \max_{y\in I}\left|f^{(n+1)}(y)\right|\cdot\frac{(b-a)^{n+1}}{(n+1)!},$$

also gerade die Behauptung. □

Wir illustrieren diese Abschätzung an zwei Beispielen.

Beispiel 4.7 Betrachte die Funktion $f(x) = \sin(x)$ auf dem Intervall $I = [0, 2\pi]$. Die Ableitungen von f sind

$$f^{(1)}(x) = \cos(x),\ \ f^{(2)}(x) = -\sin(x),\ \ f^{(3)}(x) = -\cos(x),\ \ f^{(4)}(x) = \sin(x),\ \ \ldots$$

Für alle diese Funktionen gilt $|f^{(k)}(x)| \leq 1$ für alle $x \in \mathbb{R}$. Es ergibt sich die Abschätzung

$$|f(x)-P(x)| \leq \max_{y\in I}\left|f^{(n+1)}(y)\right|\cdot\frac{(b-a)^{n+1}}{(n+1)!} \leq \frac{(2\pi)^{n+1}}{(n+1)!}.$$

Dieser Term konvergiert für wachsende n sehr schnell gegen 0, weswegen man schon für kleine n eine sehr gute Übereinstimmung der Funktionen erwarten kann. □

Beispiel 4.8 Betrachte die sogenannte *Runge–Funktion* $f(x) = 1/(1+x^2)$ auf dem Intervall $[-5, 5]$. Die exakten Ableitungen führen zu ziemlich komplizierten Termen, man kann aber nachrechnen, dass für gerade n die Gleichung

$$\max_{y\in[a,b]} |f^{(n)}(y)| = |f^{(n)}(0)| = n!$$

gilt, für ungerade n gilt zumindest approximativ

$$\max_{y\in[a,b]} |f^{(n)}(y)| \approx n!.$$

Damit ergibt sich

$$|f(x) - P(x)| \leq \max_{y\in[a,b]} \left|f^{(n+1)}(y)\right| \cdot \frac{(b-a)^{n+1}}{(n+1)!} \approx (n+1)! \cdot \frac{(b-a)^{n+1}}{(n+1)!} = 10^{n+1}.$$

Dieser Term wächst für große n gegen unendlich, weswegen die Abschätzung hier keine brauchbare Fehlerschranke liefert, und tatsächlich zeigen sich bei dieser Funktion für äquidistante Stützstellen bei numerischen Tests große Probleme; insbesondere lässt sich für wachsende n keine Konvergenz erzielen, stattdessen stellt man für große n starke Schwankungen („Oszillationen") des interpolierenden Polynoms fest. □

Tatsächlich kann es aber auch bei „gutartigen" Funktionen wie der Sinusfunktion Probleme geben, wenn man sehr viele äquidistante Stützstellen verwendet. Grund dafür ist die Kondition des Problems, die sich für die Polynominterpolation wie folgt ergibt:

n	Kondition für äquidistante Stützstellen	Kondition für Tschebyscheff-Stützstellen
5	3.11	2.10
10	29.89	2.49
15	512.05	2.73
20	10986.53	2.90
60	$2.97 \cdot 10^{15}$	3.58
100	$1.76 \cdot 10^{27}$	3.90

Da die Kondition einen guten Anhaltspunkt für die Anfälligkeit gegenüber Rundungsfehlern gibt, ist es nicht verwunderlich, dass die schlechte Kondition zu schlechten Interpolationspolynomen führt, welche sich wiederum durch starke Oszillationen bemerkbar machen.

Wie kann man diese Probleme nun umgehen? Eine Möglichkeit ist die Wahl nicht äquidistanter Stützstellen. Mit den sogenannten Tschebyscheffstützstellen kann man tatsächlich sowohl die Fehlerabschätzung als auch die Kondition – und damit auch das tatsächliche numerische Ergebnis – deutlich verbessern. Die Idee dabei ist, die Stützstellen so zu wählen, dass der Ausdruck

$$\max_{x\in[a,b]} |(x-x_0)(x-x_1)\cdots(x-x_n)|$$

möglichst klein wird. Dies führt auf die Stützstellen

$$x_i = \cos\left(\frac{2i+1}{2n+2} \cdot \pi\right) \cdot \frac{b-a}{2} + \frac{a+b}{2}.$$

Allerdings ist dies nur eine Option, wenn wir einen Einfluss auf die Wahl der Stützstellen haben. Falls diese – wie bei der Fahrbahninterpolation aus dem Modellproblem – aus

gegebenen Daten stammen, bieten die Tschebyscheff-Stützstellen keine Lösung. In diesem Fall gibt es eine andere Möglichkeit, die sogenannte Splineinterpolation, die Thema des folgenden Abschnitts ist.

4.2 Splineinterpolation

Wir betrachten in diesem Abschnitt wiederum das Interpolationsproblem (4.1), nehmen aber jetzt der einfacheren Schreibweise wegen an, dass die Stützstellen aufsteigend angeordnet sind, also $x_0 < x_1 < \ldots < x_n$ gilt.

Die Grundidee der *Splineinterpolation* liegt darin, die interpolierende Funktion nicht global, sondern nur auf jedem Teilintervall $[x_i, x_{i+1}]$ als Polynom zu wählen. Diese Teilpolynome sollten dabei an den Intervallgrenzen nicht beliebig sondern möglichst glatt zusammenlaufen. Eine solche Funktion, die aus glatt zusammengefügten stückweisen Polynomen besteht, nennt man *Spline*.

Kubische Splines werden in Anwendungen wie z. B. der Computergrafik bevorzugt verwendet, und wir wollen als nächstes den Grund dafür erläutern. Ein Kriterium zur Wahl der Ordnung eines Splines – speziell bei grafischen Anwendungen, aber auch bei „klassischen“ Interpolationsproblemen – ist, dass die Krümmung der interpolierenden Kurve möglichst klein sein soll. Die Krümmung einer Kurve $y(x)$ in einem Punkt x ist gerade gegeben durch die zweite Ableitung $y''(x)$. Die Gesamtkrümmung für alle $x \in [x_0, x_n]$ kann nun auf verschiedene Arten gemessen werden, hier verwenden wir die L_2–Norm $\|\cdot\|_2$ für quadratisch integrierbare Funktionen, die für $g : [x_0, x_n] \to \mathbb{R}$ durch

$$\|g\|_2 := \left(\int_{x_0}^{x_n} g^2(x) dx \right)^{\frac{1}{2}}$$

gegeben ist. Die Größe $\|y''\|_2$ misst gerade die Gesamtkrümmung einer zweimal stetig differenzierbaren Funktion $y : [x_0, x_n] \to \mathbb{R}$ über dem gesamten Intervall.

Kubische Splines haben nun die Eigenschaft, dass sie unter allen interpolierenden Funktionen diejenigen mit der geringsten Krümmung sind, oder formal:

Sei $S : [x_0, x_n] \to \mathbb{R}$ ein die Daten (x_i, f_i), $i = 0, \ldots, n$ interpolierender kubischer Spline mit $S''(x_0) = S''(x_n) = 0$.[1] Sei $y : [x_0, x_n] \to \mathbb{R}$ eine zweimal stetig differenzierbare Funktion, die ebenfalls das Interpolationsproblem löst und $y''(x_0) = y''(x_n) = 0$ erfüllt. Dann gilt

$$\|S''\|_2 \leq \|y''\|_2.$$

Diese Eigenschaft erklärt auch den Namen Spline: Ein „Spline“ ist im Englischen eine dünne Holzlatte. Wenn man diese so verbiegt, dass sie vorgegebenen Punkten folgt (diese also „interpoliert”), so ist auch bei dieser Latte die Krümmung, die hier näherungsweise die notwendige „Biegeenergie“ beschreibt, minimal – zumindest für kleine Auslenkungen der Latte.

Um die Berechnung solcher kubischer Splines zu erläutern, müssen wir zunächst die obige umgangssprachliche Definition in mathematische Ausdrücke fassen.

[1] Diese Bedingung bedeutet, dass die Krümmung an den Rändern des Interpolationsintervalls gerade 0 ist. Wir werden diese Bedingung später noch einmal antreffen.

Definition 4.9 Seien $x_0 < x_1 < \ldots < x_n$ Stützstellen. Eine stetige Funktion $S : [x_0, x_n] \to \mathbb{R}$ heißt *kubischer Spline*, falls die folgenden zwei Bedingungen erfüllt sind.

(i) Auf jedem Intervall $I_k = [x_{k-1}, x_k]$ mit $k = 1, \ldots, n$ ist S gegeben durch ein Polynom S_k dritten Grades, d. h. für $x \in I_k$ gilt

$$S(x) = S_k(x) = a_k + b_k(x - x_{k-1}) + c_k(x - x_{k-1})^2 + d_k(x - x_{k-1})^3.$$

(ii) Die Ableitungen der Polynome S_k an den Stützstellen erfüllen die Bedingungen

$$S_k'(x_k) = S_{k+1}'(x_k) \quad \text{und} \quad S_k''(x_k) = S_{k+1}''(x_k)$$

für alle $k = 1, \ldots, n-1$. □

Bedingung (i) sagt einfach, dass S aus Polynomen zusammengesetzt ist, während Bedingung (ii) das „glatte Zusammenstoßen“ präzisiert: Die Bedingungen an die Ableitungen garantieren, dass die Splines an den „Nahtstellen“ keine „Knicke“ haben und ihre Krümmungungen stetig ineinander übergehen.

Ein solcher kubischer Spline aus Definition 4.9 löst dann das Interpolationsproblem, falls zusätzlich die Bedingung (4.1) erfüllt ist, also $S(x_i) = f_i$ für alle $i = 0, \ldots, n$ gilt.

Wir müssen uns zunächst Gedanken darüber machen, ob das Problem der Splineinterpolation so wohldefiniert ist, d. h. ob ein interpolierender Spline immer existiert und ob er eindeutig ist. Zur Erinnerung: Nach den Ableitungsregeln für Polynome gilt

$$S_k'(x) = b_k + 2c_k(x - x_{k-1}) + 3d_k(x - x_{k-1})^2 \quad \text{und} \quad S_k''(x) = 2c_k + 6d_k(x - x_{k-1}).$$

Zur Bestimmung von S müssen wir $4n$ Werte a_i, b_i, c_i und d_i für $i = 1, \ldots, n$ berechnen.

Aus Definition 4.9 (ii) erhalten wir hierbei $2n-2$ lineare Gleichungen und aus (4.1) erhalten wir weitere $2n$ lineare Gleichungen.

Insgesamt ergeben sich so $4n - 2$ lineare Gleichungen für $4n$ Unbekannte; dieses lineare Gleichungssystem ist damit lösbar, allerdings gibt es unendlich viele Lösungen. Der Grund für die fehlenden Gleichungen liegt in den Randpunkten x_0 und x_n, in denen wir keine Glattheitsbedingungen fordern müssen.

Um eine eindeutige Lösung zu erhalten, müssen wir zwei weitere Gleichungen festlegen, welche sich üblicherweise aus *Randbedingungen* in den Punkten x_0 und x_n ergeben. Wir wollen hierbei die sogenannten *natürlichen Randbedingungen* betrachten. Hier wird gefordert, dass die zweiten Ableitungen am Rand gleich Null sein sollen, also $S''(x_0) = S''(x_n) = 0$. Dies führt auf zwei zusätzlichen Gleichungen, womit das Gleichungssystem dann eindeutig lösbar ist (tatsächlich muss man hier natürlich noch nachrechnen, dass das entstehende System lösbar ist, was wir hier nicht explizit durchführen werden). Natürlich kann man viele andere Randbedingungen festlegen, die dann auf andere zusätzliche Gleichungen führen.

Um die Koeffizienten a_k, b_k, c_k und d_k für $k = 1, \ldots, n$ tatsächlich numerisch zu berechnen empfiehlt es sich, zunächst die Werte

$$f_k'' = S''(x_k) \quad \text{und} \quad h_k = x_k - x_{k-1}$$

für $k = 0, \ldots, n$ bzw. $k = 1, \ldots, n$ zu definieren. Löst man dann die 4 Gleichungen

$$S_k(x_{k-1}) = f_{k-1}, \;\; S_k(x_k) = f_k, \;\; S_k''(x_{k-1}) = f_{k-1}'', \;\; S_k''(x_k) = f_k'' \tag{4.5}$$

– unter Ausnutzung der Ableitungsregeln für Polynome – nach a_k, b_k, c_k und d_k auf, so erhält man

$$\begin{aligned} a_k &= f_{k-1} \\ b_k &= \frac{f_k - f_{k-1}}{h_k} - \frac{h_k}{6}(f_k'' + 2f_{k-1}'') \\ c_k &= \frac{f_{k-1}''}{2} \\ d_k &= \frac{f_k'' - f_{k-1}''}{6h_k}. \end{aligned}$$

Da die Werte h_k und f_k ja direkt aus den Daten verfügbar sind, müssen lediglich die Werte f_k'' berechnet werden. Da aus den natürlichen Randbedingungen sofort $f_0'' = 0$ und $f_n'' = 0$ folgt, brauchen nur die Werte $f_1'', \ldots, f_{n-1}''$ berechnet werden.

Beachte, dass wir in (4.5) bereits die Bedingungen an S_k und S_k'' in den Stützstellen verwendent haben. Aus den noch nicht benutzten Gleichungen für die ersten Ableitungen erhält man nun die Gleichungen für die f_k'': Aus $S_k'(x_k) = S_{k+1}'(x_k)$ erhält man

$$b_k + 2c_k(x_k - x_{k-1}) + 3d_k(x_k - x_{k-1})^2 = b_{k+1}$$

für $k = 1, \ldots, n-1$. Indem man hier die Werte f_k'' und h_k gemäß den obigen Gleichungen bzw. Definitionen einsetzt erhält man

$$h_k f_{k-1}'' + 2(h_k + h_{k+1}) f_k'' + h_{k+1} f_{k+1}'' = 6\frac{f_{k+1} - f_k}{h_{k+1}} - 6\frac{f_k - f_{k-1}}{h_k} =: \delta_k$$

für $k = 1, \ldots, n-1$. Dies liefert genau $n-1$ Gleichungen für die $n-1$ Unbekannten $f_1'', \ldots, f_{n-1}''$. In Matrixform geschrieben erhalten wir so das Gleichungssystem

$$\begin{pmatrix} 2(h_1+h_2) & h_2 & 0 & \ldots & \ldots & 0 \\ h_2 & 2(h_2+h_3) & h_3 & \ddots & \ddots & \vdots \\ 0 & h_3 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \ldots & \ldots & 0 & h_{n-1} & 2(h_{n-1}+h_n) \end{pmatrix} \begin{pmatrix} f_1'' \\ f_2'' \\ \vdots \\ \vdots \\ f_{n-1}'' \end{pmatrix} = \begin{pmatrix} \delta_1 \\ \delta_2 \\ \vdots \\ \vdots \\ \delta_{n-1} \end{pmatrix}$$

Zur Berechnung des Interpolationssplines löst man also zunächst dieses Gleichungssystem und berechnet dann gemäß der obigen Formel die Koeffizienten a_k, b_k, c_k, d_k aus den f_k''.

Kapitel 5

Integration

Die Integration von Funktionen ist eine elementare mathematische Operation, die in vielen Formeln benötigt wird. Im Gegensatz zur Ableitung, die für praktisch alle mathematischen Funktionen explizit analytisch berechnet werden kann, gibt es viele Funktionen, deren Integrale man nicht explizit angeben kann. Verfahren zur numerischen Integration (man spricht auch von *Quadratur*[1]) spielen daher eine wichtige Rolle, sowohl als eigenständige Algorithmen als auch als Basis für andere Anwendungen wie z. B. der numerischen Lösung von Differentialgleichungen.

Das Problem lässt sich hierbei ganz einfach beschreiben: Für eine Funktion $f : \mathbb{R} \to \mathbb{R}$ soll das Integral

$$\int_a^b f(x)dx \tag{5.1}$$

auf einem Intervall $[a, b]$ berechnet werden.

Wir behandeln hier zwei einfache Algorithmen, die aber in vielen Anwendungen gute Ergebnisse erzielen, nämlich die *Newton–Cotes–Formeln* und die *zusammengesetzten Newton–Cotes–Formeln*, die auch als *iterierte* oder *aufsummierte* Newton–Cotes–Formeln bezeichnet werden.

5.1 Newton–Cotes–Formeln

Die Grundidee der numerischen Integration liegt darin, das Integral (5.1) durch eine Summe

$$\int_a^b f(x)dx \quad \approx \quad (b-a)\sum_{i=0}^{n} \alpha_i f(x_i) \tag{5.2}$$

zu approximieren. Hierbei heißen die x_i die *Stützstellen* und die α_i die *Gewichte* der Integrationsformel.

Die Stützstellen x_i können hierbei entweder vorgegeben werden oder sie sind durch Messwerte oder Ergebnisse vorhergehender numerischer Berechnungen bestimmt.

[1] Das Wort Quadratur wird sowohl für die Integration verwendet – so wie bei uns – als auch für die Aufgabe der Umformung einer geometrischen Figur in ein Quadrat mit gleichem Flächeninhalt durch Konstruktion (siehe die berühmte und unmögliche Quadratur des Kreises).

Beispiel 5.1 Im Modellprojekt der LKW-Kabine ist es aus Ingenieursicht interessant, die Energiemenge zu berechnen, die von den Dämpfern während der Fahrt absorbiert wird. Auf dem Zeitintervall $[t_0, t_0 + T]$ ist diese definiert durch das Integral

$$\int_{t_0}^{t_0+T} f(t)dt$$

mit (bei linearer Reibung)

$$\begin{aligned} f(t) &= d_1(\dot{x}_1(t) - \dot{x}_2(t))\dot{x}_1(t) \\ &\quad + (d_2(\dot{x}_2(t) - \dot{x}_3(t)) - d_1(\dot{x}_1(t) - \dot{x}_2(t)))\dot{x}_2(t) \\ &\quad + (d_3(\dot{x}_3(t) - \dot{s}(t)) - d_2(\dot{x}_2(t) - \dot{x}_3(t)))\dot{x}_3(t). \end{aligned}$$

Beachte, dass hier über t statt über x wie in der allgemeinen Formulierung integriert wird. Wenn wir $x_i(t)$ numerisch durch Lösung der Differentialgleichung berechnen, erhalten wir die Werte $f(t_i)$ an den Zeitgitterpunkten t_i. Andere Werte haben wir nicht zur Verfügung, so dass wir nur diese als Stützstellen verwenden können. □

Um die Integrationsformel anwenden zu können, benötigen wir eine Formel, mit der wir zu gegebenen (oder von uns gewählten) x_i sinnvolle Gewichte α_i berechnen können.

Die Idee der Newton–Cotes Formeln liegt nun darin, die Funktion F zunächst durch ein Interpolationspolynom P vom Grad n (zu den Daten $(x_i, f(x_i))$, $i = 0, \ldots, n$) zu approximieren und dann das Integral über dieses Polynom zu berechnen. Wir führen diese Konstruktion nun durch:

Da wir einen expliziten Ausdruck in den $f(x_i)$ erhalten wollen, bietet sich die Darstellung von P mittels der Lagrange–Polynome an, also

$$P(x) = \sum_{i=0}^{n} f(x_i)L_i(x)$$

mit

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

vgl. Abschnitt 4.1.2. Das Integral über P ergibt sich dann zu

$$\begin{aligned} \int_a^b P(x)dx &= \int_a^b \sum_{i=0}^{n} f(x_i)L_i(x)\,dx \\ &= \sum_{i=0}^{n} f(x_i) \int_a^b L_i(x)\,dx. \end{aligned}$$

Um die Gewichte α_i in (5.2) zu berechnen, setzen wir

$$(b-a)\sum_{i=0}^{n} \alpha_i f(x_i) = \sum_{i=0}^{n} f(x_i) \int_a^b L_i(x)dx.$$

Auflösen nach α_i liefert dann

$$\alpha_i = \frac{1}{b-a} \int_a^b L_i(x)dx. \tag{5.3}$$

Diese α_i können dann explizit berechnet werden, denn die Integrale über die Lagrange–Polynome L_i sind explizit lösbar. Hierbei hängen die Gewichte α_i von der Wahl der Stützstellen x_i ab, nicht aber von den Funktionswerten $f(x_i)$. Für äquidistante Stützstellen

$$x_i = a + \frac{i(b-a)}{n}$$

sind die Gewichte aus (5.3) in Tabelle 5.1 für $n = 1, \ldots, 7$ angegeben.

n	α_0	α_1	α_2	α_3	α_4	α_5	α_6	α_7
1	$\frac{1}{2}$	$\frac{1}{2}$						
2	$\frac{1}{6}$	$\frac{4}{6}$	$\frac{1}{6}$					
3	$\frac{1}{8}$	$\frac{3}{8}$	$\frac{3}{8}$	$\frac{1}{8}$				
4	$\frac{7}{90}$	$\frac{32}{90}$	$\frac{12}{90}$	$\frac{32}{90}$	$\frac{7}{90}$			
5	$\frac{19}{288}$	$\frac{75}{288}$	$\frac{50}{288}$	$\frac{50}{288}$	$\frac{75}{288}$	$\frac{19}{288}$		
6	$\frac{41}{840}$	$\frac{216}{840}$	$\frac{27}{840}$	$\frac{272}{840}$	$\frac{27}{840}$	$\frac{216}{840}$	$\frac{41}{840}$	
7	$\frac{751}{17280}$	$\frac{3577}{17280}$	$\frac{1323}{17280}$	$\frac{2989}{17280}$	$\frac{2989}{17280}$	$\frac{1323}{17280}$	$\frac{3577}{17280}$	$\frac{751}{17280}$

Tabelle 5.1: Gewichte der Newton–Cotes Formeln aus (5.3) für äquidistante Stützstellen x_i

Beachte, dass sich die Gewichte immer zu 1 aufsummieren und symmetrisch in i sind, d. h. es gilt $\alpha_i = \alpha_{n-i}$. Außerdem sind die Gewichte unabhängig von den Intervallgrenzen a und b.

Aus der Abschätzung des Interpolationsfehlers kann man eine Abschätzung für den Integrationsfehler

$$F_n[f] := \int_a^b f(x)dx - (b-a) \sum_{i=0}^{n} \alpha_i f(x_i)$$

ableiten. Hierbei müssen die Stützstellen x_i nicht unbedingt äquidistant liegen.

Satz 5.2 Seien α_i die Gewichte, die gemäß (5.3) zu den Stützstellen $a \leq x_0 < \ldots < x_n \leq b$ berechnet wurden und sei $h = (b-a)/n$. Dann gibt es von a, b und f unabhängige Konstanten c_n, so dass für alle $(n+1)$–mal differenzierbaren Funktionen f die Abschätzung

$$|F_n[f]| \leq c_n h^{n+2} \max_{y \in [a,b]} |f^{(n+1)}(y)|$$

gilt, wobei $f^{(n+1)}$ wie im Abschnitt 4.1.4 die $(n+1)$–te Ableitung der Funktion f bezeichnet.

Beweis: Aus dem Ansatz zur Berechnung der Gewichte α_i folgt direkt die Gleichung

$$(b-a)\sum_{i=0}^{n}\alpha_i f(x_i) = \int_a^b P(x)dx$$

und damit

$$F_n[f] = \int_a^b f(x)dx - \int_a^b P(x)dx = \int_a^b f(x) - P(x)\,dx.$$

Aus Satz 4.6(i) folgt daher

$$\begin{aligned} |F_n[f]| &= \left|\frac{1}{(n+1)!}\int_a^b f^{(n+1)}(\xi(x))\prod_{i=0}^{n}(x-x_i)\,dx\right| \\ &\leq \frac{1}{(n+1)!}\max_{y\in[a,b]}\left|f^{(n+1)}(y)\right|\int_a^b\prod_{i=0}^{n}|x-x_i|\,dx, \end{aligned}$$

mit Werten $\xi(x)\in(a,b)$. Daraus folgt die behauptete Abschätzung mit

$$\begin{aligned} c_n &= \frac{1}{(n+1)!}\frac{1}{h^{n+2}}\int_a^b\prod_{i=0}^{n}|x-x_i|\,dx = \frac{1}{(n+1)!}\left(\frac{n}{b-a}\right)^{n+2}\int_a^b\prod_{i=0}^{n}|x-x_i|\,dx \\ \left(\text{Substitution: } z = n\tfrac{x-a}{b-a},\ \ z_i = n\tfrac{x_i-a}{b-a}\right) &= \frac{1}{(n+1)!}\int_0^n\prod_{i=0}^{n}|z-z_i|\,dz \end{aligned}$$

□

Bemerkung 5.3 Für gerades n, $(n+2)$–mal differenzierbares f und symmetrisch verteilte Stützstellen x_i (z. B. äquidistante Stützstellen) lässt sich die bessere Abschätzung

$$|F_n[f]| \leq d_n h^{n+3}\max_{y\in[a,b]}|f^{(n+2)}(y)|$$

beweisen, wobei die d_n ähnlich wie die c_n in Satz 5.2 berechnet werden. □

Die Konstanten c_n und d_n können für gegebenes n explizit berechnet werden. In Tabelle 5.2 sind die darauf basierenden Fehlerabschätzungen für $n = 1,\ldots,7$ und äquidistante Stützstellen approximativ angegeben, wobei $M_n := \max_{y\in[a,b]}|f^{(n)}(y)|$ ist.

n	1	2	3	4	5	6	7
	$\frac{(b-a)^3M_2}{12}$	$\frac{(b-a)^5M_4}{2880}$	$\frac{(b-a)^5M_4}{6480}$	$\frac{5.2(b-a)^7M_6}{10^7}$	$\frac{2.9(b-a)^7M_6}{10^7}$	$\frac{6.4(b-a)^9M_8}{10^{10}}$	$\frac{3.9(b-a)^9M_8}{10^{10}}$

Tabelle 5.2: Fehlerabschätzungen der Newton–Cotes Formeln für äquidistante Stützstellen

Vereinfacht gesagt erhöht sich also die Genauigkeit mit wachsendem n, allerdings nur dann, wenn nicht zugleich die höheren Ableitungen $|f^{(n)}|$ zunehmen. Es tauchen also die gleichen prinzipiellen Probleme wie bei den Interpolationspolynomen auf, was nicht weiter verwunderlich ist, da diese ja dem Verfahren zu Grunde liegen. Hier kommt aber noch ein weiteres

Problem hinzu, nämlich kann man für $n = 8$ und $n \geq 10$ beobachten, dass einige der Gewichte α_i negative werden. Dies kann zu numerischen Problemen (z. B. Auslöschungen) führen, die man möglichst vermeiden möchte. Aus diesem Grunde ist es nicht ratsam, den Grad des zugrundeliegenden Polynoms immer weiter zu erhöhen. Statt dessen wählt man ein anderes Verfahren, das im folgenden Abschnitt beschrieben ist.

5.2 Zusammengesetzte Newton–Cotes–Formeln

Der Ausweg aus den Problemen mit immer höheren Polynomgraden ist bei der Integration mittels Newton–Cotes–Formeln ganz ähnlich wie bei der Interpolation—nur einfacher. Bei der Interpolation sind wir von Polynomen zu Splines, also stückweisen Polynomen übergegangen. Um dort weiterhin eine „schöne“ Approximation zu erhalten, mussten wir Bedingungen an den Nahtstellen festlegen, die eine gewisse Glattheit der approximierenden Funktion erzwingen, weswegen wir die Koeffizienten recht kompliziert über ein lineares Gleichungssystem herleiten mussten.

Bei der Integration fällt diese Prozedur weg. Wie bei den Splines verwenden wir zur Herleitung der zusammengesetzten Newton–Cotes–Formeln stückweise Polynome, verzichten aber auf aufwändige Bedingungen an den Nahtstellen, da wir ja nicht an einer schönen Approximation der Funktion, sondern „nur“ an einer guten Approximation des Integrals interessiert sind. In der Praxis berechnet man die zugrundeliegenden stückweisen Polynome nicht wirklich, sondern wendet die Newton–Cotes–Formeln wir folgt auf den Teilintervallen an:

Sei N die Anzahl von Teilintervallen, auf denen jeweils die Newton–Cotes–Formel vom Grad n verwendet werden soll. Wir setzen

$$x_i = a + ih, \;\; i = 0, 1, \ldots, nN, \quad h = \frac{b-a}{nN}$$

und zerlegen das Integral (5.1) mittels

$$\int_a^b f(x)dx \;\; = \;\; \int_{x_0}^{x_n} f(x)dx + \int_{x_n}^{x_{2n}} f(x)dx + \cdots + \int_{x_{(N-1)n}}^{x_{Nn}} f(x)dx.$$

Auf jedem Teilintervall $[x_{jn}, x_{(j+1)n}]$ wenden wir nun die Newton–Cotes–Formel an, d. h. wir approximieren

$$\int_{x_{jn}}^{x_{(j+1)n}} f(x)dx \;\; \approx \;\; nh \sum_{i=0}^{n} \alpha_i f(x_{jn+i})$$

und addieren die Teilapproximationen auf, also

$$\int_a^b f(x)dx \;\; \approx \;\; nh \sum_{j=0}^{N-1} \sum_{i=0}^{n} \alpha_i f(x_{jn+i}).$$

Der entstehende Approximationsfehler

$$F_{N,n}[f] := \int_a^b f(x)dx - nh \sum_{j=0}^{N-1} \sum_{i=0}^{n} \alpha_i f(x_{jn+i})$$

ergibt sich einfach als Summe der Fehler $F_n[f]$ auf den Teilintervallen, weswegen man aus Satz 5.2 die Abschätzung

$$\begin{aligned} |F_{N,n}[f]| &\leq \sum_{j=0}^{N-1} c_n h^{n+2} \max_{y\in[x_{jn},x_{(j+1)n}]} |f^{(n+1)}(y)| \\ &\leq N c_n h^{n+2} \max_{y\in[a,b]} |f^{(n+1)}(y)| \quad = \quad \frac{c_n}{n}(b-a)h^{n+1} \max_{y\in[a,b]} |f^{(n+1)}(y)| \end{aligned}$$

und aus Bemerkung 5.3 für gerades n die Abschätzung

$$|F_{N,n}[f]| \leq \frac{d_n}{n}(b-a)h^{n+2} \max_{y\in[a,b]} |f^{(n+2)}(y)|$$

erhält. Im Folgenden geben wir die zusammengesetzten Newton–Cotes–Formeln für $n = 1, 2, 4$ mitsamt ihren Fehlerabschätzungen an; in allen Formeln sind die Stützstellen x_i als $x_i = a + ih$ gewählt.

$n = 1$**, Trapez–Regel:**

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

$$|F_{N,1}[f]| \leq \frac{b-a}{12}h^2 \max_{y\in[a,b]} |f^{(2)}(y)|, \quad h = \frac{b-a}{N} \quad \Rightarrow \quad |F_{N,1}[f]| = \mathcal{O}\left(\frac{1}{N^2}\right)$$

$n = 2$**, Simpson–Regel:**

$$\int_a^b f(x)dx \approx \frac{h}{3}\left(f(a) + 2\sum_{i=1}^{N-1} f(x_{2i}) + 4\sum_{i=0}^{N-1} f(x_{2i+1}) + f(b)\right)$$

$$|F_{N,2}[f]| \leq \frac{b-a}{180}h^4 \max_{y\in[a,b]} |f^{(4)}(y)|, \quad h = \frac{b-a}{2N} \quad \Rightarrow \quad |F_{N,1}[f]| = \mathcal{O}\left(\frac{1}{N^4}\right)$$

$n = 4$**, Milne–Regel:**

$$\begin{aligned} \int_a^b f(x)dx \approx \frac{2h}{45}\Bigg(&7\Big(f(a) + f(b)\Big) + 14\sum_{i=1}^{N-1} f(x_{4i}) \\ &+ 32\sum_{i=0}^{N-1}\Big(f(x_{4i+1}) + f(x_{4i+3})\Big) + 12\sum_{i=0}^{N-1} f(x_{4i+2})\Bigg) \end{aligned}$$

$$|F_{N,4}[f]| \leq \frac{2(b-a)}{945}h^6 \max_{y\in[a,b]} |f^{(6)}(y)|, \quad h = \frac{b-a}{4N} \quad \Rightarrow \quad |F_{N,1}[f]| = \mathcal{O}\left(\frac{1}{N^6}\right)$$

Zum Abschluss wollen wir an einem Beispiel die praktischen Auswirkungen der Fehlerabschätzungen illustrieren.

Beispiel 5.4 Das Integral

$$\int_0^1 e^{-x^2/2} dx$$

soll mit einer garantierten Genauigkeit von $\varepsilon = 10^{-10}$ numerisch approximiert werden. Die Ableitungen der Funktion $f(x) = e^{-x^2/2}$ lassen sich leicht berechnen; es gilt

$$f^{(2)}(x) = (x^2-1)f(x), \;\; f^{(4)}(x) = (3-6x^2+x^4)f(x), \;\; f^{(6)}(x) = (-15+45x^2-15x^4+x^6)f(x).$$

Mit etwas Rechnung (oder aus der grafischen Darstellung) sieht man, dass all diese Funktionen ihr betragsmäßiges Maximum auf $[0,1]$ in $y = 0$ annehmen, woraus die Gleichungen

$$\max_{y\in[0,1]} |f^{(2)}(y)| = 1, \;\; \max_{y\in[0,1]} |f^{(4)}(y)| = 3 \;\; \text{und} \;\; \max_{y\in[0,1]} |f^{(6)}(y)| = 15$$

folgen.

Löst man die oben angegebenen Fehlerabschätzungen $|F_{N,n}[f]| \leq \varepsilon$ für die Trapez–, Simpson– und Milne–Regel nach h auf und setzt die Abschätzungen für $\max_{y\in[0,1]} |f^{(n)}(y)|$ ein, so erhält man die folgenden Bedingungen an h

$$h \;\leq\; \sqrt{\frac{12\varepsilon}{(b-a)|f^{(2)}(0)|}} \;\approx\; \frac{1}{28867.51} \qquad \text{(Trapez–Regel)}$$

$$h \;\leq\; \sqrt[4]{\frac{180\varepsilon}{(b-a)|f^{(4)}(0)|}} \;\approx\; \frac{1}{113.62} \qquad \text{(Simpson–Regel)}$$

$$h \;\leq\; \sqrt[6]{\frac{945\varepsilon}{2(b-a)|f^{(6)}(0)|}} \;\approx\; \frac{1}{10.59} \qquad \text{(Milne–Regel)}$$

Der Bruch auf der rechten Seite gibt dabei die maximal erlaubte Größe für h vor. Um diese zu realisieren, muss $1/(nN) \leq h$ gelten für die Anzahl $nN + 1$ der Stützstellen. Da nN ganzzahlig ist, braucht man also 28869 Stützstellen für die Trapez–Regel, 115 Stützstellen für die Simpson–Regel und 13 Stützstellen für die Milne–Regel. □

5.3 Integration mit Monte Carlo

Während die bisherigen Integrationsmethoden geometrisch anschaulich waren und „direkt“ die Fläche unter einem Graphen berechneten, betrachten wir nun eine Methode, die die Wahrscheinlichkeitsrechnung nutzt. Im ersten Moment mag es ungewohnt klingen, Integration mit Mitteln der Wahrscheinlichkeitsrechnung angehen zu wollen. Wenn wir uns jedoch in Erinnerung rufen, dass Erwartungswerte (z. B. für stetige Zufallsvariablen) durch Integrale berechnet werden, erscheint es vielleicht weniger abwegig. Wir nehmen an, dass $(\Omega, \mathcal{F}, \mathbb{P})$ ein Wahrscheinlichkeitsraum und $U : \Omega \to [a,b]$ eine gleichverteilte Zufallsvariable ist. Das heißt, U hat die Dichte $f_U(x) = \frac{1}{b-a}$ auf $[a,b]$ und 0 sonst. Wir wollen $\int_a^b f(x)dx =: I$ für eine integrierbare Funktion f berechnen. Es gilt:

$$I = (b-a)\int_a^b f(x) f_U(x) dx = (b-a)\mathbb{E}[f(U)].$$

Wenn wir nun unabhänig identisch verteilte ($u.i.v.$) Realisationen von U ziehen können[2] $(u_1, u_2, \ldots)$, definieren wir

$$S_N = \frac{1}{N} \sum_{i=1}^{N} f(u_i), \quad N = 1, 2, \ldots,$$

und es gilt nach dem starken Gesetz der großen Zahlen, dass

$$\lim_{N \to \infty} S_N = \mathbb{E}[f(U)].$$

Als Approximation setzen wir:

$$I \approx I_N = (b - a) S_N$$

Man kann zeigen, dass, wenn f eine nichtkonstante Funktion ist, S_N mit Ordnung $\mathcal{O}\left(\frac{1}{\sqrt{N}}\right)$ gegen $\mathbb{E}[f(U)]$ konvergiert (siehe [Philip, 2018]).[3] Vergleicht man diese Ordnung mit der Trapezregel ($\mathcal{O}\left(\frac{1}{N^2}\right)$), sieht man, dass Monte Carlo viel langsamer ist. Man kann jedoch zeigen, dass für Verallgemeinerungen der Trapezregel auf höhere Dimensionen ($d \in \mathbb{N}$), deren Ordnung $\mathcal{O}\left(\frac{1}{\sqrt[d]{N^2}}\right)$ ist. Die Ordnung von Monte Carlo hingegen ist unabhängig von d. Das heißt, Monte Carlo vermeidet den sogenannten „Fluch der Dimensionen“. Zu beachten ist, dass Monte Carlo zufällige Ergebnisse liefert (die mit beliebig hoher Wahrscheinlichkeit < 1 nahe an I liegen). Dennoch werden sich i. A. die Werte für zwei verschiedenen Berechnungen des gleichen Integrals (mit der gleichen Methode) unterscheiden.[4]

[2] Es spielt keine Rolle, ob es sich um echte Zufallszahlen oder um künstliche berechnete Zahlen, die Zufallszahlen ähnlich sind (sogenannte Pseudozufallszahlen), handelt. Pseudozufallszahlengeneratoren sind heutzutage in der Regel in Programmiersprachen verfügbar, wobei wir vor sogenannten LCGen warnen möchten.

[3] Wobei die Konvergenz in einem stochastischen Sinne, der auf Konfidenzintervallen basiert, zu verstehen ist.

[4] Bei Verwendung von Pseudozufallszahlen kann dies durch das Setzen eines `seed`s verhindert werden.

Kapitel 6

Gewöhnliche Differentialgleichungen

In diesem Kapitel behandeln wir numerische Algorithmen (und einige theoretische Grundlagen) von Gewöhnlichen Differentialgleichungen der Form

$$\dot{y}(t) := \frac{d}{dt}y(t) = f(t, y(t)), \tag{6.1}$$

wobei

$$y : \mathbb{R} \to \mathbb{R}^n, \qquad y(t) = \begin{pmatrix} y_1(t) \\ \vdots \\ y_n(t) \end{pmatrix}$$

die gesuchte vektorwertige Lösung der Differentialgleichung und

$$f : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n, \qquad f(t, y) = \begin{pmatrix} f_1(t, y) \\ \vdots \\ f_n(t, y) \end{pmatrix}$$

eine gegebene Funktion ist.

Gleichung (6.1) ist eine vektorwertige gewöhnliche Differentialgleichung erster Ordnung und stellt die „Standardform“ gewöhnlicher Differentialgleichungen dar: für fast alle numerischen Algorithmen wird vorausgesetzt, dass die Gleichung in dieser Form vorliegt.

6.1 Umformung in eine vektorwertige DGL erster Ordnung

Oft ist dies aber zunächst nicht der Fall. Die Modellgleichungen, die zu Modellprojekt 1 gehören, sind z. B. ein System von drei skalaren gewöhnlichen Differentialgleichungen zweiter Ordnung. Wie bringt man diese in die Standardform (6.1)? Wir illustrieren dies allgemein für ein System skalarer Differentialgleichungen zweiter Ordnung der Form

$$\ddot{x}_i(t) = g_i(t, x_1(t), \dot{x}_1(t), \ldots, x_d(t), \dot{x}_d(t)), \quad i = 1, \ldots, d.$$

Nun führt man den $n = 2d$–dimensionalen Vektor

$$y(t) = \begin{pmatrix} y_1(t) \\ y_2(t) \\ \vdots \\ y_{n-1}(t) \\ y_n(t) \end{pmatrix} = \begin{pmatrix} x_1(t) \\ \dot{x}_1(t) \\ \vdots \\ x_d(t) \\ \dot{x}_d(t) \end{pmatrix}$$

ein. Für $y_1, y_3, \ldots$ gilt dann

$$\dot{y}_i(t) = \dot{x}_{(i+1)/2}(t) = y_{i+1}(t)$$

und für $y_2, y_4, \ldots$ gilt

$$\dot{y}_i(t) = \ddot{x}_{i/2}(t) = g_{i/2}(t, x_1(t), \dot{x}_1(t), \ldots, x_d(t), \dot{x}_d(t)) = g_{i/2}(t, y_1(t), \ldots, y_n(t)).$$

Wir erhalten also (6.1) mit

$$f(t, y) = \begin{pmatrix} f_1(t, y) \\ f_2(t, y) \\ \vdots \\ f_{n-1}(t, y) \\ f_n(t, y) \end{pmatrix} = \begin{pmatrix} y_2 \\ g_1(t, y_1, \ldots, y_n) \\ \vdots \\ y_n \\ g_d(t, y_1, \ldots, y_n) \end{pmatrix}$$

6.2 Anfangswertaufgaben

Die Problemstellung „löse die Differentialgleichung (6.1)“ ist in dieser allgemeinen Form nicht lösbar, denn zu einer DGL (6.1) existieren üblicherweise unendlich viele verschiedene Lösungen.

Beispiel 6.1 Betrachte die skalare Differentialgleichung

$$\dot{y}(t) = 3y(t).$$

Man rechnet leicht nach, dass jede Funktion der Form $y(t) = C \exp(3t)$ für beliebiges $C \in \mathbb{R}$ eine Lösung dieser Differentialgleichung ist. □

Um also eine brauchbare Problemstellung zu erhalten, für die es nur eine Lösung gibt, müssen wir weitere Informationen zur Problemstellung hinzufügen. Hierzu wählen wir eine Anfangszeit t_0 und einen *Anfangswert* y_0 und betrachten diejenige Lösung $y(t)$ von (6.1), die die *Anfangsbedingung*

$$y(t_0) = y_0 \tag{6.2}$$

erfüllt. Unter geeigneten Bedingungen an f, die wir hier nicht weiter ausführen wollen, gilt:

Für jedes Paar (t_0, y_0) existiert genau eine Lösung $y(t)$ der Differentialgleichung (6.1), die die Anfangsbedingung (6.2) erfüllt.

Diese wollen wir im Folgenden numerisch berechnen. Zunächst aber einige Beispiele.

Beispiel 6.2 Im einfachen Beispiel 6.1 rechnet man leicht nach, dass die Lösung zu (t_0, y_0) gerade durch

$$y(t) = \exp(3(t - t_0))y_0$$

gegeben ist.

Hierbei stellt man fest: Die Lösungen zu verschiedenen Anfangswerten laufen sehr schnell weit auseinander! Insbesondere verursachen kleine Änderungen des Anfangswertes große Änderungen der Lösung zu späteren Zeiten. Differentialgleichungen mit diesem Verhalten nennt man *instabil*. □

Beispiel 6.3 Ein Beispiel einer noch viel stärker instabilen Differentialgleichung ist das – ebenfalls skalare – Anfangswertproblem

$$\dot{y}(t) = 10\left(y - \frac{t^2}{1+t^2}\right) + \frac{2t}{(1+t^2)^2}, \qquad y(0) = y_0.$$

Für $y_0 = 0$ erhält man hier die Lösung

$$y(t) = \frac{t^2}{1+t^2}$$

während man für $y_0 = -\varepsilon$ die Lösung

$$y(t) = -\varepsilon \exp(10\,t) + \frac{t^2}{1+t^2}$$

erhält. Selbst für sehr kleines ε (das z. B. einen Rundungsfehler im Anfangswert repräsentiert) laufen die Lösungen sehr schnell weit auseinander. □

Beachte: Wenn kleine Änderungen im Anfangswert große Änderungen in der Lösung hervorrufen, so werden i. A. auch numerische Fehler drastisch verstärkt → die DGL ist numerisch schwer zu lösen.

Beispiel 6.4 Die im Wurfbeispiel 1.1 zur Modellierung verwendete Wurfparabel gibt keine Informationen darüber, zu welchem Zeitpunkt sich der Ball wo befindet. Um diese Informationen zu erhalten, können wir die Wurfbewegung durch eine Differentialgleichung modellieren. Hierbei gelten die folgenden Bezeichnungen:

y_1 = horizontale Position
y_2 = horizontale Geschwindigkeit
y_3 = vertikale Position
y_4 = vertikale Geschwindigkeit

Dann gilt die Differentialgleichung

$$\dot{y}(t) = f(t, y(t)) := \begin{pmatrix} y_2(t) \\ 0 \\ y_4(t) \\ -g \end{pmatrix}.$$

Abwurfwinkel β und Abwurfgeschwindigkeit v_0 gehen in die Anfangsbedingung ein:

$$y(0) = y_0 = \begin{pmatrix} 0 \\ v_0 \cos\beta \\ 0 \\ v_0 \sin\beta \end{pmatrix}.$$

Diese Differentialgleichung ist so einfach, dass man sie analytisch lösen kann. Die Lösung ist gegeben durch

$$y(t) = \begin{pmatrix} tv_0 \cos\beta \\ v_0 \cos\beta \\ tv_0 \sin\beta - t^2 g/2 \\ v_0 \sin\beta - tg \end{pmatrix}.$$

□

Beispiel 6.5 Auch für die Simulation der LKW-Kabine aus unserem Modellprojekt (Kapitel 3) benötigen wir einen Anfangswert. Hier liegt es nahe, die bereits berechnete Ruhelage unter Schwerkraft zu verwenden. □

6.3 Zeitgitter

Wir wollen uns nun der Lösung des Anfangswertproblems (6.1), (6.2) widmen. Wir beginnen mit skalaren Problemen, also solchen, bei denen die gesuchte Funktion $y(t)$ von $\mathbb{R}$ nach $\mathbb{R}$ abbildet. Dies können wir hier guten Gewissens machen, weil die Verallgemeinerung auf Vektoren später sehr einfach sein wird.

Gesucht ist also eine Lösungsfunktion $y : \mathbb{R} \to \mathbb{R}$. Eine wesentliche Schwierigkeit hierbei ist, dass wir die Differentialgleichung (6.1) in den allermeisten Fällen nicht einfach formal nach $y(t)$ auflösen können. Unser Algorithmus muss also mehr leisten als die *Auswertung* einer Funktion: er muss die Gleichung (6.1) auch *auflösen*.

Dies geht nicht exakt, d. h. wir können nicht erwarten, einen numerischen Algorithmus zu finden, der uns die exakte Lösung $y(t)$ liefert. Statt dessen werden wir nur eine *Approximation*, also eine Annäherung der Funktion $y(t)$ berechnen können. Zudem werden wir diese nicht für alle möglichen Zeiten t berechnen, sondern nur für eine endliche Anzahl von Zeiten.

Formal bedeutet dies das Folgende: Wir wollen die Lösungefunktion $y(t)$ auf einem Intervall $I = [t_0, t_0 + T]$ ausrechnen, wobei t_0 gerade die Anfangszeit aus der Anfangsbedingung ist.

Hierzu definieren wir uns ein *Zeitgitter*

$$I_h = \{t_0, t_1, \ldots, t_m = t_0 + T\},$$

das aus den $m + 1$ *Zeitgitterpunkten* $t_i = t_0 + ih$ besteht, vgl. Abbildung 6.1. Die Zahl $m \in \mathbb{N}$ ist die *Schrittanzahl* des Gitters. Aus ihr berechnet sich die *Schrittweite* $h > 0$ mittels $h = T/m$.

Unser Algorithmus soll uns nun die (unbekannten) exakten Werte

$$y_i = y(t_i),$$

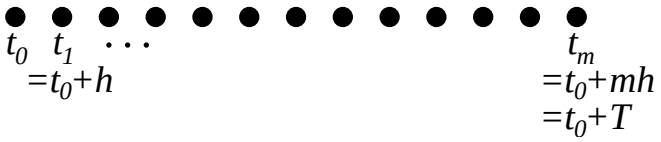

Abbildung 6.1: Zeitgitter

– also die Werte der Funktion $y(t)$ an den Zeitgitterpunkten t_i – durch numerisch berechenbare Näherungswerte

$$u_i \approx y_i$$

approximieren und das Problem der Berechnung von $y(t)$ so *näherungsweise* lösen.

6.4 Das explizite Euler-Verfahren

Unser erster einfacher Algorithmus beruht auf der Beobachtung, dass der Wert der Funktion $f(t, y(t))$ in (6.1) zu jedem Zeitpunkt t gerade die Steigung der Funktion $y(t)$ darstellt – denn nichts anderes ist ja die erste Ableitung $\dot{y}(t)$ geometrisch, vgl. Abbildung 6.2, in denen die Steigung durch die Tangente (also eine Gerade mit der Steigung $\dot{y}(t)$ durch den Punkt $y(t)$) symbolisiert wird.

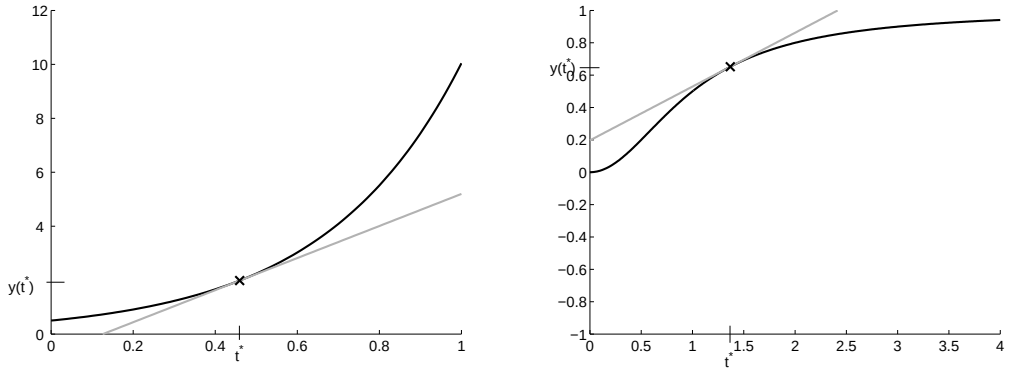

Abbildung 6.2: Lösungsfunktion und Steigung für Beispiele 6.2 und 6.3

Die hier skizzierten Tangenten an die Lösungsfunktionen sind für die Punkte $(t^*, y(t^*))$ gerade gegeben durch die Tangentengerade

$$t \mapsto y(t^*) + (t - t^*)\dot{y}(t^*) = y(t^*) + (t - t^*)f(t^*, y(t^*)). \tag{6.3}$$

Hierbei beobachtet man, dass die Tangente für t nahe t^* eine gute Näherung der Lösung darstellt, d. h. für $|t - t^*|$ klein gilt

$$y(t) \approx y(t^*) + (t - t^*)f(t^*, y(t^*)).$$

Auf dieser Beobachtung beruht das explizite Euler-Verfahren. In diesem Verfahren wird zunächst $u_0 = y_0$ gewählt. Dann werden die Näherungen u_1, u_2, ... sukzessive durch die Tangentengleichung (6.3) aus dem jeweiligen Vorgängerpunkt berechnet. Wir verwenden also die Tangentengleichung (6.3) mit $t^* = t_i$, $y(t^*) = u_i$, $t = t_{i+1}$ und erhalten damit unter Ausnutzung von $t_{i+1} - t_i = h$:

Algorithmus 6.6 (Explizites Euler-Verfahren)

Eingabe: Funktion f aus (6.1), t_0, y_0 aus (6.2), Schrittanzahl $m \in \mathbb{N}$, Schrittweite $h > 0$

(1) Setze $u_0 = y_0$

(2) für $i = 0, 1, \ldots, m-1$, setze

$$u_{i+1} = u_i + hf(t_i, u_i)$$

Ausgabe: Näherungswerte $u_i \approx y(t_i)$ □

Dies ist das einfachste Verfahren zur approximativen Lösung gewöhnlicher Differentialgleichungen.

Wir wollen an einem Beispiel verdeutlichen, dass dies Verfahren tatsächlich funktioniert.

Beispiel 6.7 Betrachte wiederum die Differentialgleichung

$$\dot{y}(t) = 3y(t).$$

Wir haben also $f(t, y) = 3y$. Wir wählen die Anfangsbedingung $t_0 = 0$, $y_0 = 1$. Die exakte Lösung zur Zeit $T = 1$ ist dann $y(T) = y_0 \exp(3(t - t_0)) = \exp(3) = 20.0855\ldots$.

Für das Euler-Verfahren wählen wir die Schrittweite $h = 1/m$, so dass wir gerade nach m Schritten die Zeit $t_m = t_0 + mh = 1$, also die gewünschte Endzeit erreichen. Die Iterationsvorschrift in Schritt (2) des Euler-Verfahrens ergibt sich also zu

$$u_{i+1} = u_i + hf(t, u_i) = u_i + \frac{1}{m} 3u_i = \left(1 + \frac{3}{m}\right) u_i.$$

Für $u_0 = 1$ errechnet man daraus die Werte

m	u_m
5	10.4858
10	13.7858
50	18.4202
100	19.2186
500	19.9063
1000	19.9955
5000	20.0675
10000	20.0765

Man erkennt: Die Approximation des exakten Wertes wird um so genauer, je mehr Schritte wir machen. Andererseits wird aber natürlich mit der Anzahl der Schritte auch der Rechenaufwand höher. Wir haben hier also eine für die Numerik typische Situation:
je höher die Rechengenauigkeit, desto länger wird die Rechenzeit. □

6.5 Allgemeine Einschrittverfahren

Das explizite Euler-Verfahren ist das einfachste Beispiel eines *allgemeinen expliziten Einschrittverfahrens*. Diese Verfahren funktionieren ganz genau wie Algorithmus 6.6, mit dem Unterschied, dass die Iterationsvorschrift in Schritt (2) durch

$$u_{i+1} = u_i + h\Phi(t_i, u_i, h)$$

ersetzt wird, wobei Φ nun eine geeignet gewählte Funktion ist. Mit

$$\Phi(t_i, u_i, h) = f(t_i, u_i)$$

erhalten wir den Spezialfall des Euler-Verfahrens, Ein anderes Verfahren ist das *Heun-Verfahren*, für das

$$\Phi(t_i, u_i, h) = \frac{1}{2}\Big(f(t_i, u_i) + f(t_i + h, u_i + hf(t_i, u_i))\Big)$$

gewählt wird. Das Heun-Verfahren kann man wie folgt aus der Trapezregel herleiten: Durch Integration von (6.1) erhält man die Gleichung

$$y(t_{i+1}) = y(t_i) + \int_{t_i}^{t_{i+1}} f(s, y(s))ds$$

Approximiert man das Integral durch die Trapezregel, so erhält man

$$y(t_{i+1}) = y(t_i) + \frac{h}{2}\Big(f(t_i, y(t_i)) + f(t_{i+1}, y(t_{i+1}))\Big)$$

und durch Ersetzen $u_i \approx y(t_i)$ und $u_{i+1} \approx y(t_{i+1})$ sowie $t_{i+1} = t_i + h$ ergibt sich

$$u_{i+1} = u_i + \frac{h}{2}\Big(f(t_i, u_i) + f(t_i + h, u_{i+1})\Big).$$

Diese Formel hat aber noch das Problem, dass der zu berechnende Wert u_{i+1} auch auf der rechten Seite im Argument von f auftaucht. Um dies zu vermeiden, ersetzen wir diesen Wert durch einen Schritt des Euler-Verfahrens. Damit erhalten wir gerade das Heun-Verfahren.

Für dieses Verfahren wiederholen wir die Berechnung aus Beispiel 6.7 (zur Erinnerung: der exakte Wert ist 20.0855 . . .).

m	u_m
5	17.8690
10	19.3742
50	20.0510
100	20.0767
500	20.0852
1000	20.0854

Wir sehen: Mit dem Heun-Verfahren können wir die Lösung mit 500 Zeitschritten auf drei Nachkommastellen genau berechnen, während das Euler-Verfahren selbst mit 10000 Zeitschritten nur eine korrekte Nachkommastelle liefert. Die Genauigkeit des Euler-Verfahrens mit 10000 Schritten erreicht das Heun-Verfahren bereits mit 100 Schritten.

Die Iterationsvorschrift (Schritt (2) des Algorithmus) lässt sich leicht auf vektorwertige Probleme verallgemeinern, in denen

$$y(t) = \begin{pmatrix} y_1(t) \\ \vdots \\ y_n(t) \end{pmatrix} \text{ und } f(t,y) = \begin{pmatrix} f_1(t,y) \\ \vdots \\ f_n(t,y) \end{pmatrix}$$

gilt. Hier werden u_i und Φ zu Vektoren

$$u_i = \begin{pmatrix} u_{i,1} \\ \vdots \\ u_{i,n} \end{pmatrix} \text{ und } \Phi(t,u,h) = \begin{pmatrix} \Phi_1(t,u,h) \\ \vdots \\ \Phi_n(t,u,h) \end{pmatrix}$$

und Schritt (2) wird nun einfach auf alle Komponenten des Vektors angewendet, also

(2) für $i = 0, 1, \ldots, m-1$, setze

$$u_{i+1,j} = u_{i,j} + h\Phi_j(t_i, u_i, h), \quad j = 1, \ldots, n$$

6.6 Fehlerbetrachtung

Warum ist das Heun-Verfahren so viel genauer als das Euler-Verfahren? Wie kann man die Genauigkeit eigentlich messen? Und warum funktionieren diese Verfahren überhaupt. Dies wollen wir in diesem und dem nächsten Abschnitt theoretisch untersuchen.

Hierzu überlegen wir uns, dass die numerische Lösung in jedem Schritt gerade die Gleichung

$$\frac{u_{i+1} - u_i}{h} - \Phi(t_i, u_i, h) = 0$$

erfüllt. Setzen wir nun die exakte Lösung in die linke Seite dieser Gleichung ein, so wird i. A. *nicht* 0 herauskommen. Wir definieren

$$\tau(t,h) := \frac{y(t+h) - y(t)}{h} - \Phi(t, y(t), h).$$

Die so definierte Größe τ heißt *lokaler Diskretisierungsfehler*.

Wie kann man diesen Fehler interpretieren? Nehmen wir an, dass wir bis zur Zeit t_i keine Fehler gemacht haben, dass also $u_i = y(t_i)$ gilt (eine Annahme, die natürlich nur für t_0 realistisch ist). Dann folgt

$$\tau(t_i,h)h = y(t_{i+1}) - y(t_i) - h\Phi(t_i, y(t_i), h) = y(t_{i+1}) \underbrace{- u_i - h\Phi(t_i, u_i, h)}_{=-u_{i+1}} = y(t_{i+1}) - u_{i+1}.$$

Der mit h skalierte Fehler $\tau(t_i,h)h$ misst also gerade den Fehler in einem Schritt unter der Annahme, dass bis zu diesem Schritt keine Fehler gemacht wurden. Wir werden im nächsten Abschnitt sehen, wie man aus τ den tatsächlichen Fehler (also ohne die unrealistische Annahme) berechnen kann. Zuvor aber noch ein paar Definitionen.

Definition 6.8 Ein Einschrittverfahren heißt *konsistent*, wenn für alle $t \in I = [t_0, t_0 + T]$ für alle differenzierbaren Funktionen f

$$\lim_{\substack{h \to 0 \\ h>0}} \tau(t,h) = 0$$

gilt.

Das Verfahren heißt *konsistent der Ordnung* $p \in \mathbb{N}$, falls

$$\tau(t,h) = \mathcal{O}(h^p)$$

gilt. □

Hierbei ist „$\tau(t,h) = \mathcal{O}(h^p)$“ eine Kurzschreibweise für:

Es existiert eine Konstante $C > 0$, so dass die Ungleichung $|\tau(t,h)| \leq Ch^p$ gilt.

Konsistenz ist sehr einfach zu testen: Es genügt, dass die Gleichung

$$f(t,y) = \Phi(t,y,0) \tag{6.4}$$

erfüllt ist.

Gleichung (6.4) sagt leider gar nichts über die Konsistenzordnung aus. Diese ist aber entscheidend für die Genauigkeit des Verfahrens, denn es gilt:
Je größer die Konsistenzordnung p, desto schneller konvergiert der lokale Diskretisierungsfehler $\tau(t,h)$ gegen 0 für $h \to 0$.

Für die zwei bisher betrachteten Verfahren gilt:

Satz 6.9 Das Euler-Verfahren besitzt Konsistenzordnung $p = 1$, das Heun-Verfahren Konsistenzordnung $p = 2$.

Beweis: Um die Konsistenzordnung des Euler-Verfahrens zu berechnen, muss man die *Taylor-Entwicklung* der Lösung anwenden:

$$y(t+h) = y(t) + h\dot{y}(t) + \frac{h^2}{2}\ddot{y}(\theta)$$

für ein (i. A. unbekanntes) θ mit $t \leq \theta \leq t + h$. Da $y(t)$ die Differentialgleichung erfüllt, gilt $\dot{y}(t) = f(t, y(t))$, also

$$y(t+h) = y(t) + hf(t,y(t)) + \frac{h^2}{2}\ddot{y}(\theta)$$

Zudem gilt für das Euler-Verfahren $\Phi(t,y,h) = f(t,y)$ und damit

$$\begin{aligned} \tau(t,h) &= \frac{y(t+h) - y(t)}{h} - \Phi(t,y(t),h) \\ &= \frac{hf(t,y(t)) + \frac{h^2}{2}\ddot{y}(\theta)}{h} - f(t,y(t)) = \frac{h}{2}\ddot{y}(\theta) = O(h), \end{aligned}$$

da $\ddot{y}(\theta)$ für $\theta \in [t, t+h]$ durch eine Konstante C abgeschätzt werden kann.

Für das Heun-Verfahren geht man ähnlich vor, indem man die Taylor-Entwicklung von y bis h^3 und die Taylor-Entwicklung von Φ verwendet. Hierbei fallen dann in der durch h geteilten Differenz alle Terme mit h heraus, so dass nur Terme der Ordnung $O(h^2)$ übrig bleiben.

6.7 Abschätzung des globalen Fehlers

Wir defineren den Fehler des Einschrittverfahrens als

$$e(t_i, h) := y(t_i) - u_i.$$

Definition 6.10 Ein Einschrittverfahren heißt *konvergent*, wenn für alle $i = 0, \ldots, m$ für alle differenzierbaren Funktionen f

$$\lim_{\substack{h \to 0 \\ h>0}} |e(t_i, h)| = 0$$

gilt.

Das Verfahren heißt *konvergent der Ordnung* $p \in \mathbb{N}$, falls

$$|e(t_i, h)| = \mathcal{O}(h^p)$$

gilt. □

Der lokale Diskretisierungsfehler $\tau(t, h)$ gibt nur eine Auskunft über den Fehler $e(t_i, h)$, wenn wir annehmen, dass wir bis zu diesem Schritt keinen Fehler gemacht haben, also dass $e(t_{i-1}, h) = 0$ ist. Das ist natürlich unrealistisch. Wie kann man nun zu einer Aussage über den wirklichen Fehler kommen? Dazu müssen wir Schritt für Schritt beobachten, wie sich der Betrag des Fehler $e(t_i, h)$ entwickelt. Wir illustrieren dies für das expliziten Euler-Verfahren und die Differentialgleichung $\dot{y}(t) = Ly(t)$, in der $L > 0$ eine feste Konstante ist. In der Rechnung nutzen wir aus, dass $\tau(t, h) = O(h)$ gilt, d. h., dass eine Konstante C existiert mit $|\tau(t, h)| \leq Ch$.

Zum Start des Algorithmus gelte $u_0 = y_0$, d. h. wir kennen den Anfangswert exakt und ohne Rundungs- bzw. Messfehler. Im ersten Schritt können wir daher ausnutzen, dass zu Beginn des Schrittes tatsächlich noch kein Fehler gemacht wurde. Es gilt daher nach der Rechnung aus dem letzten Abschnitt

$$|e(t_1, h)| = |y(t_1) - u_1| = \tau(t_1, h)h \leq Ch^2.$$

Im nächsten Schritt geht das nicht mehr, weswegen die Rechnung komplizierter wird:

$$\begin{aligned}
|e(t_2, h)| &= |y(t_2) - u_2| \\
&= |y(t_2) - u_1 - hf(t_1, u_1)| \\
&= |y(t_2) - y(t_1) - hf(t_1, y(t_1)) + y(t_1) + hf(t_1, y(t_1)) - u_1 - hf(t_1, u_1)| \\
&\leq |y(t_2) - y(t_1) - hf(t_1, y(t_1))| + \underbrace{|y(t_1) + hf(t_1, y(t_1)) - u_1 - hf(t_1, u_1)|}_{\substack{=|y(t_1)+hLy(t_1)-u_1-hLu_1| \\ =|(1+hL)(y(t_1)-u_1)|=(1+hL)|e(t_1,h)|}} \\
&\leq Ch^2 + (1 + hL)|e(t_1, h)| \leq Ch^2 + (1 + hL)Ch^2 \leq (1 + hL)2Ch^2.
\end{aligned}$$

Die gleiche Rechnung liefert im dritten Schritt

$$|e(t_3, h)| \leq (1 + hL)^2 3Ch^2$$

und nach i Schritten erhalten wir so

$$|e(t_i, h)| \leq (1 + hL)^{i-1} iCh^2.$$

Um diese Abschätzung noch etwas schöner zu machen, nutzen wir die Ungleichungen

$$(1 + hL)^{i-1} \leq \exp(ihL) = \exp(L(t_i - t_0)) \quad \text{und} \quad i \leq (t_i - t_0)/h$$

aus. Damit folgt dann

$$|e(t_i, h)| \leq \exp(L(t_i - t_0))(t_i - t_0)Ch. \tag{6.5}$$

bzw. im "schlimmsten" Fall, wenn $t_i = t_0 + T$ ist,

$$|e(t_i, h)| \leq \exp(LT)TCh.$$

Diese Abschätzung lässt sich auf allgemeine Einschrittverfahren und allgemeine Differentialgleichungen verallgemeinern. Es gilt:

Satz 6.11 Betrachte ein explizites Einschrittverfahren mit Iterationsvorschrift $u_{i+1} = u_i + h\Phi(t_i, u_i, h)$. Für Φ existiere ein $L > 0$, so dass die *Lipschitzbedingung*

$$|\Phi(t, y_1, h) - \Phi(t, y_2, h)| \leq L|y_1 - y_2|$$

für alle t, h, y_1, y_2 gilt. Dann gilt:

Ist das Verfahren konsistent mit Ordnung p, so ist es auch konvergent mit Ordnung p.

Genauer gilt die Abschätzung

$$|e(t_i, h)| \leq \exp(L(t_i - t_0))(t_i - t_0)Ch^p$$

für eine geeignete Konstante $C > 0$.

6.8 Diskussion der Fehlerabschätzung

(i) Die Berechnung der Konstanten C, die aus der Taylor-Entwicklung stammt, ist üblicherweise recht aufwändig. Oft stellt aber der $\exp(\ldots)$-Term die kritische Größe in dieser Abschätzung dar, so dass C für eine überschlagsmäßige Abschätzung des Fehlers in vielen Fällen vernachlässigt werden kann. Insbesondere wenn sowohl L als auch $t_i - t_0$ groß sind, kann der $\exp(\ldots)$-Term riesige Werte annehmen und damit zu großen Fehlern führen.

(ii) Die Konstante L hängt in erster Linie von der zu lösenden Gleichung und weniger vom gewählten Einschrittverfahren ab. Bei instabilen Differentialgleichungen ist diese Konstante groß, daher gilt: **instabile Differentialgleichungen sind auf längeren Zeitintervallen numerisch schwer zu lösen**. Dies ist anschaulich leicht einzusehen: Bei instabilen Differentialgleichungen, wie diejenige in Beispiel 6.3 laufen Lösungen mit nahe beieinander

liegenden Anfangswerten schnell auseinander. Genauso wie kleine Änderungen im Anfangswert große Änderungen in der Lösung hervorrufen können, so können auch kleine Fehler in den einzelnen Schritten des Einschrittverfahrens große Fehler in der Lösung hervorrufen.

(iii) Die Fehleranalyse in zwei Stufen (erst Konsistenz, dann Konvergenz) scheint auf den ersten Blick unnötig kompliziert, da am Ende ja doch „Konsistenzordnung = Konvergenzordnung“ herauskommt. Diese Vorgehen ist allerdings notwendig, da sich nur die Konsistenzordnung (über die Taylor-Entwicklung) direkt berechnen lässt.

(iv) Diese Fehlerabschätzung geht davon aus, dass bei der Auswertung des Einschrittverfahrens keine Fehler gemacht werden. In der Praxis haben wir es aber auch hier mit den bereits im ersten Kapitel angesprochenen Rundungsfehlern zu tun, weswegen man typischerweise beobachtet, dass der Fehler für sehr kleine Schrittweiten h nicht weiter ab-, sondern – im Gegenteil – wieder zunimmt.

6.9 Steife Differentialgleichungen

Nicht nur instabile sondern auch “besonders stabile” Differentialgleichungen führen zu numerischen Schwierigkeiten. Als Beispiel betrachten wir die Differentialgleichung

$$\dot{x}(t) = \lambda x(t)$$

mit $\lambda < 0$. Abbildung 6.3 zeigt die Lösungen für verschiedene λ in schwarz und die zugehörigen Euler-Approximationen (rot). Man sieht: während die exakte Lösung für immer größere λ immer schneller gegen 0 strebt, beginnt die numerische Näherungslösung zu oszillieren und wird sogar instabil.

Der Grund dafür ist, dass die Lipschitzkonstante L des Vektorfelds $f(x) = \lambda x$ gerade $L = |\lambda|$ ist und die Fehlerabschätzung daher wie erwartet schlecht wird. Ähnliche Phänomene treten bei höherdimensionalen linearen Differentialgleichungen $\dot{x} = Ax$ auf, wenn A Eigenwerte mit großem Betrag $|\lambda|$ aber negativem Realteil $\mathrm{Re}\lambda < 0$ besitzt. Auf Grund der mathematischen Analyse ist es also zu erwarten, dass die Euler-Approximation schlecht wird, aber anschaulich ist es trotzdem überraschend, denn eigentlich bewegt sich die Lösung ja nur am Anfang, danach aber praktisch nicht mehr. Sie hat also eine ganz einfache Struktur, die gut zu approximieren wäre. Gleichungen mit Lösungen dieser Art nenne man **steife Differentialgleichungen**.

Diese Beobachtung führt tatsächlich zu einer Abhilfe für dieses Problem. Man kann nämlich zeigen, dass gewisse Apprimationsverfahren diese Eigenschaft — schnelle Konvergenz gegen ein Gleichgewicht und dann praktisch konstantes Vert halten — von der exakten Gleichung “erben” und daher für steife Differentialgleichungen viel genauer sind. Ein Beispiel ist das **implizite Euler-Verfahren**, das gegeben ist durch die Vorschrift

$$u_{i+1} = u_i + hf(t_{i+1}, u_{i+1}).$$

Abbildung 6.4 zeigt die Lösungen dieses Verfahrens zusätzlich zu den Graphen aus Abbildung 6.3 in blau. Man sieht, dass diese viel genauer sind als das explizite Verfahren. Dies ist sogar für deutlich größere Zeitschritte der Fall.

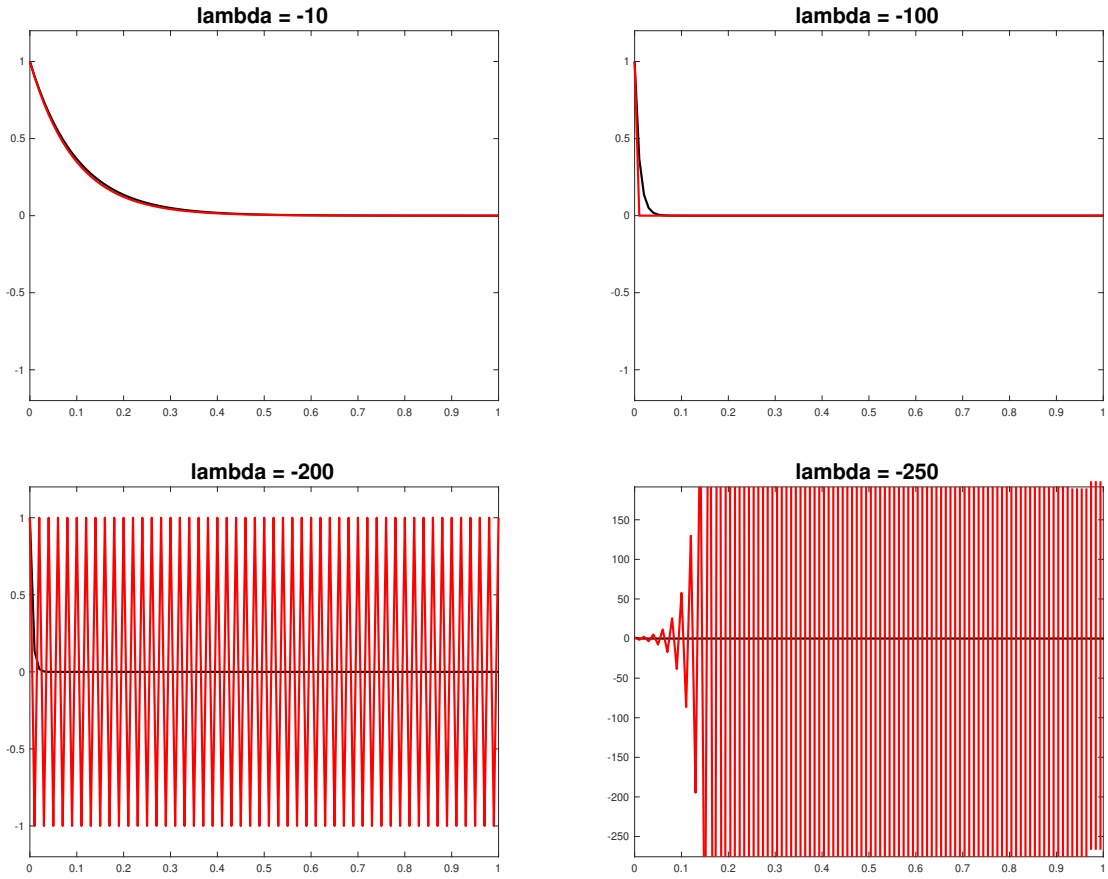

Abbildung 6.3: Lösungen (schwarz) und explizite Euler-Approximation (rot) für $\dot{x} = \lambda x$ mit variierendem λ

Das Problem bei den impliziten Verfahren (die es wie die expliziten Verfahren auch mit höherer Konsistenzordnung gibt) ist, dass de Gleichung erst nach u_{i+1} aufgelöst werden muss, bevor man u_{i+1} berechnen kann. Das wird i.A. numerisch gemacht. Verfahren dafür behandeln wir im nächsten Kapitel. Für einfache Gleichungen (wie unser Beispiel $\dot{x} = \lambda x$) kann die Auflösung aber auch analytisch gemacht werden:

$$\dot{x}(t) = \lambda x(t) \quad \rightsquigarrow \quad u_{i+1} = u_i + h\lambda u_{i+1} \quad \Rightarrow \quad u_{i+1} = \frac{u_i}{1 - h\lambda}.$$

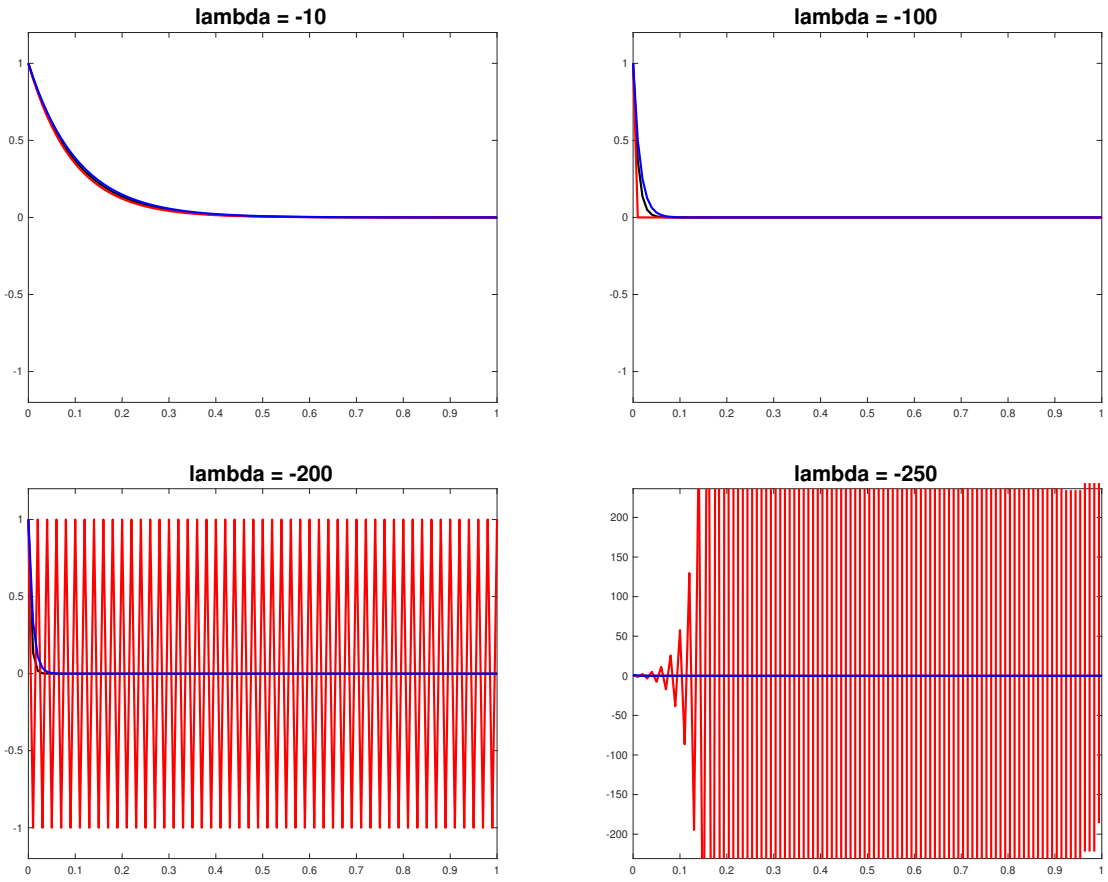

Abbildung 6.4: Lösungen (schwarz), explizite Euler-Approximation (rot) und implizite Euler-Approximation (blau) für $\dot{x} = \lambda x$ mit variierendem λ

Kapitel 7

Nichtlineare Gleichungen und Gleichungssysteme

Nichtlineare Gleichungen oder Gleichungssysteme müssen in vielen Anwendungen der Mathematik gelöst werden. Das „Prototyp“-problem dafür ist die Suche einer Nullstelle x^* einer nichtlinearen Funktion f, also eines x^* mit

$$f(x^*) = 0$$

Die Funktion f kann hierbei reell sein, also $f : \mathbb{R} \to \mathbb{R}$, und beschreibt dann eine einzelne nichtlineare Gleichung. Sie kann aber auch vektoriell sein, also $f : \mathbb{R}^n \to \mathbb{R}^n$, und beschreibt dann ein nichtlineares Gleichungssystem. Wichtig ist nur, dass x und $f(x)$ aus dem gleichen Raum stammen. Jede nichtlineare Gleichung und jedes nichtlineare Gleichungssystem kann in diese Nullstellenform gebracht werden.

Beispiel (Berechnung von Quadratwurzeln): Berechne $x^* > 0$ mit $f(x^*) = 0$ für $f(x) = 1 - x^2/2$. Die eindeutige positive Lösung ist $\sqrt{2} \approx 1.41421356237310$; ein numerisches Verfahren zur Berechnung von Nullstellen kann also insbesondere zur Berechnung von Wurzeln verwendet werden.

7.1 Das Bisektionsverfahren

Das Bisektionsverfahren ist eine einfache Methode für eindimensionale Probleme mit stetiger Funktion $f : \mathbb{R} \to \mathbb{R}$. Sie beruht auf dem Zwischenwertsatz (bzw. auf einem Spezialfall, der Nullstellensatz genannt wird) der Analysis, welcher besagt, dass für zwei Stellen $a < b$, in denen

$$\text{entweder } f(a) \leq 0 \text{ und } f(b) \geq 0 \qquad \text{oder } f(a) \geq 0 \text{ und } f(b) \leq 0$$

gilt, immer eine Stelle $x^* \in [a, b]$ existiert mit $f(x^*) = 0$.

Der Algorithmus wird gestartet mit zwei Stellen $a^{(0)} < b^{(0)}$, an denen f unterschiedliches Vorzeichen besitzt und berechnet dann für eine frei wählbare Fehlerschranke $\varepsilon > 0$ sukzessive

(1) $x^{(i)} = \frac{a^{(i)}+b^{(i)}}{2}$

(2) Falls $f(x^{(i)}) = 0$ oder $b^{(i)} - a^{(i)} < 2\varepsilon$ beende den Algorithmus mit $x^{(i)}$ als Ergebnis

(3) Falls $f(x^{(i)})f(a^{(i)}) < 0$ ist setze $a^{(i+1)} = a^{(i)}$, $b^{(i+1)} = x^{(i)}$,
sonst setze $a^{(i+1)} = x^{(i)}$, $b^{(i+1)} = b^{(i)}$;
setze $i = i + 1$ und gehe zu (1)

Der Fehler ist dabei maximal gleich der halben Intervallbreite, welche sich in jedem Schritt halbiert. Es gilt also

$$|x^{(i)} - x^*| \leq \left(\frac{1}{2}\right)^{i+1} |b^{(0)} - a^{(0)}|.$$

Im Unterschied zur Fixpunktiteration, die wir als nächstes kennen lernen werden, konvergiert der Fehler dabei im Allgemeinen nicht monoton, d. h. er kann im Schritt $i+1$ größer sein als im Schritt i, erfüllt aber immer die obige Abschätzung. Die Anwendung auf unsere Beispielfunktion $f(x) = 1 - x^2/2$ mit Startwerten $a = 1$ und $b = 2$ illustriert dies:

```
x( 0) = 1.50000000000000  (Fehler: 0.08578643762690)
x( 1) = 1.25000000000000  (Fehler: 0.16421356237310)
x( 2) = 1.37500000000000  (Fehler: 0.03921356237310)
x( 3) = 1.43750000000000  (Fehler: 0.02328643762690)
x( 4) = 1.40625000000000  (Fehler: 0.00796356237310)
x( 5) = 1.42187500000000  (Fehler: 0.00766143762690)
x( 6) = 1.41406250000000  (Fehler: 0.00015106237310)
x( 7) = 1.41796875000000  (Fehler: 0.00375518762690)
x( 8) = 1.41601562500000  (Fehler: 0.00180206262690)
x( 9) = 1.41503906250000  (Fehler: 0.00082550012690)
x(10) = 1.41455078125000  (Fehler: 0.00033721887690)
x(11) = 1.41430664062500  (Fehler: 0.00009307825190)
x(12) = 1.41418457031250  (Fehler: 0.00002899206060)
x(13) = 1.41424560546875  (Fehler: 0.00003204309565)
x(14) = 1.41421508789062  (Fehler: 0.00000152551753)
x(15) = 1.41419982910156  (Fehler: 0.00001373327153)
x(16) = 1.41420745849609  (Fehler: 0.00000610387700)
x(17) = 1.41421127319336  (Fehler: 0.00000228917974)
x(18) = 1.41421318054199  (Fehler: 0.00000038183110)
x(19) = 1.41421413421631  (Fehler: 0.00000057184321)
x(20) = 1.41421365737915  (Fehler: 0.00000009500606)
```

Großer Vorteil des Bisektionsverfahrens (gegenüber der Fixpunktiteration), dass es immer konvergiert, wenn man passende Startwerte $a^{(0)}$ und $b^{(0)}$ finden kann – man nennt so ein Verfahren *global konvergent*. Wesentlicher Nachteil ist, dass es nicht auf Funktionen im $\mathbb{R}^n$ verallgemeinert werden kann.

7.2 Fixpunktiteration

Die Fixpunktiteration ist eine recht einfache Methode, die auf der Idee beruht, die Lösung eines nichtlinearen Gleichungssystems als Fixpunktgleichung zu formulieren. Setzen wir

$$g(x) = f(x) + x,$$

so gilt $f(x^*) = 0$ genau dann, wenn $g(x^*) = x^*$ gilt. Ein Punkt $x^* \in \mathbb{R}^n$ mit $g(x^*) = x^*$ heißt *Fixpunkt* von g. Statt einer Nullstelle von f können wir alternativ also einen Fixpunkt von g suchen.

Der Algorithmus dazu ist nun – ausgehend von einem Startwert $x^{(0)}$ denkbar einfach

(1) Berechne $x^{(i+1)} = g(x^{(i)})$

(2) Falls $|x^{(i+1)} - x^{(i)}| < \varepsilon$, beende den Algorithmus,
ansonsten setzte $i = i + 1$ und gehe zu (1)

Angewendet auf unser Beispiel erhalten wir $g(x) = 1 - x^2/2 + x$. Mit dem Startwert $x^{(0)} = 1$ erhalten wir in 10 Iterationen

```
x( 0) = 1.00000000000000  (Fehler: 0.41421356237310)
x( 1) = 1.50000000000000  (Fehler: 0.08578643762690)
x( 2) = 1.37500000000000  (Fehler: 0.03921356237310)
x( 3) = 1.42968750000000  (Fehler: 0.01547393762690)
x( 4) = 1.40768432617188  (Fehler: 0.00652923620122)
x( 5) = 1.41689674509689  (Fehler: 0.00268318272380)
x( 6) = 1.41309855196381  (Fehler: 0.00111501040929)
x( 7) = 1.41467479318270  (Fehler: 0.00046123080961)
x( 8) = 1.41402240794944  (Fehler: 0.00019115442365)
x( 9) = 1.41429272285787  (Fehler: 0.00007916048478)
x(10) = 1.41418076989350  (Fehler: 0.00003279247959)
```

Offenbar funktioniert dieses Verfahren also. Funktioniert es immer? Dazu betrachten wir zwei Varianten. In der ersten Variante ändern wir den Startwert auf $x^{(0)} = 4$. Damit erhalten wir

```
x( 0) =            4.00000000000000  (Fehler:           2.58578643762691)
x( 1) =           -3.00000000000000  (Fehler:           4.41421356237309)
x( 2) =           -6.50000000000000  (Fehler:           7.91421356237309)
x( 3) =          -26.62500000000000  (Fehler:          28.03921356237310)
x( 4) =         -380.07031250000000  (Fehler:         381.48452606237311)
x( 5) =       -72605.79153442382812  (Fehler:       72607.20574798619782)
x( 6) = -2635873086.96163988113403  (Fehler: 2635873088.37585353851318)
```

also offenbar keine Konvergenz.

Das Verfahren funktioniert also nur dann, wenn der Startwert hinreichend nahe am Fixpunkt liegt. Man sagt, das Verfahren ist *lokal konvergent*.

Für eine zweite Variante ändern wir die Funktion $f(x) = 1 - x^2/2$ ab zu $f(x) = 2 - x^2$. Diese besitzt die gleichen Nullstellen, wie in der ersten Variante. Für $g(x) = 2 - x^2 + x$ erhalten wir

```
x( 0) = 1.00000000000000  (Fehler: 0.41421356237310)
x( 1) = 2.00000000000000  (Fehler: 0.58578643762690)
x( 2) = 0.00000000000000  (Fehler: 1.41421356237310)
x( 3) = 2.00000000000000  (Fehler: 0.58578643762690)
x( 4) = 0.00000000000000  (Fehler: 1.41421356237310)
x( 5) = 2.00000000000000  (Fehler: 0.58578643762690)
x( 6) = 0.00000000000000  (Fehler: 1.41421356237310)
x( 7) = 2.00000000000000  (Fehler: 0.58578643762690)
x( 8) = 0.00000000000000  (Fehler: 1.41421356237310)
x( 9) = 2.00000000000000  (Fehler: 0.58578643762690)
x(10) = 0.00000000000000  (Fehler: 1.41421356237310)
```

Das Verfahren konvergiert also nicht (Tests mit anderen Startwerten zeigen, dass es ebenfalls nicht konvergiert).

Warum funktioniert das Verfahren für $g(x) = 1-x^2/2+x$, versagt aber für $g(x) = 2-x^2+x$? Die Antwort liegt – ähnlich wie für die Iterationsverfahren für lineare Gleichungssysteme – im Banach'schen Fixpunktsatz. Da wir hier nichtlineare Funktionen betrachten, müssen wir allerdings nicht die Funktionen selbst, sondern ihre Ableitungen betrachten:

Im eindimensionalen konvergiert das Verfahren immer dann, wenn man eine Umgebung U von x^* finden kann, sodass für alle $x \in U$ die Ungleichung

$$|g'(x)| \leq k < 1$$

gilt. In diesem Fall gelten für alle Startwerte $x^{(0)} \in U$ die Abschätzungen

$$|x^{(i)} - x^*| \leq \frac{k}{1-k}|x^{(i)} - x^{(i-1)}|$$

und

$$|x^{(i)} - x^*| \leq \frac{k^i}{1-k}|x^{(1)} - x^{(0)}|$$

Existiert hingegen eine Umgebung U von x^*, sodass für alle $x \in U$ die Ungleichung

$$|g'(x)| > 1$$

gilt, so konvergiert das Verfahren nicht.

Testen wir dies an unseren beiden Beispielfunktionen: Für die erste Funktion $g(x) = 1 - x^2/2 + x$ gilt

$$g'(x) = -x + 1$$

und damit z. B. für $x \in U = (0.1, 1.9)$:

$$|g'(x)| = |-x + 1| \leq 0.9 < 1.$$

Für die zweite Funktion $g(x) = 2 - x^2 + x$ gilt

$$g'(x) = -2x + 1$$

und damit für $x \in U = [1.1, 2]$ (und damit auch für alle kleineren Umgebungen U um $x^* = \sqrt{2}$)

$$|g'(x)| = |-2x + 1| \geq 1.2 > 1.$$

Für vektorwertige Funktionen $g : \mathbb{R}^n \to \mathbb{R}^n$ funktioniert das Konvergenzkriterium ähnlich. In diesem Fall ist die Ableitung $\frac{d}{dx}g(x)$ eine $n \times n$–Matrix, weswegen die Bedingung $|g'(x)| \leq k < 1$ durch

$$\left\| \frac{d}{dx} g(x) \right\| \leq k < 1$$

ersetzt werden muss, wobei $\|\cdot\|$ eine beliebige induzierte Matrixnorm ist.

Diese Kriterien lassen sich verallgemeinern, wenn die Funktion g nicht differenzierbar ist, sind dann aber nicht mehr so schön auszurechnen.

In der Praxis wird man die Fixpunktiteration typischerweise zunächst einmal ohne große vorherige Analyse ausprobieren. Wenn sie zu keinem Ergebnis führt, kann man versuchen, durch Skalierung der Funktion f (wie in unserem Beispiel) eine Form zu finden, für die der Algorithmus konvergiert. Das heißt, man multipliziert die Funktion mit einem (kleinen) Skalar, sodass sie flacher durch die Nullstelle geht (wenn die Funktion dort differenzierbar ist). Es gibt aber auch Funktionen, für die dieses Vorgehen eben nicht geht, z. B. Nullstelle von $sgn(x)\sqrt[3]{|x|}$ mit U symmetrisch um $x^* = 0$.

Gemeinsamer Nachteil der beiden bisher vorgestellten Verfahren ist, dass sie vergleichsweise langsam konvergieren. Beide Verfahren besitzen Fehlerschranken der Form

$$|x^{(i)} - x^*| \leq Ck^i =: e_i$$

für ein $k \in (0, 1)$. Diese Art der Konvergenz nennt man *lineare Konvergenz*, weil die Fehlerschranken e_i gerade die lineare Gleichung

$$e_{i+1} = ke_i$$

erfüllen. Lineare Konvergenz kann man leicht erkennen, wenn man die Ergebnisfolge betrachtet: Die Anzahl der korrekten Stellen in $x^{(i)}$ nimmt mit konstanter Geschwindigkeit zu.

Ein deutlich schnelleres Verfahren ist das Newton–Verfahren, das im folgenden Abschnitt vorgestellt wird.

7.3 Das Newton–Verfahren

Die Idee des Newton–Verfahrens ist im eindimensionalen wie folgt: Berechne die Tangente $g(x)$ von f im Punkt $x^{(i)}$, d. h. die Gerade

$$g(x) = f(x^{(i)}) + f'(x^{(i)})(x - x^{(i)})$$

und wähle $x^{(i+1)}$ als Nullstelle von g, also

$$\begin{aligned} & f(x^{(i)}) + f'(x^{(i)})(x^{(i+1)} - x^{(i)}) = 0 \\ \Leftrightarrow \quad & f'(x^{(i)})x^{(i+1)} = f'(x^{(i)})x^{(i)} - f(x^{(i)}) \\ \Leftrightarrow \quad & x^{(i+1)} = x^{(i)} - \frac{f(x^{(i)})}{f'(x^{(i)})}. \end{aligned}$$

Die Idee ist in Abbildung 7.1 grafisch dargestellt.

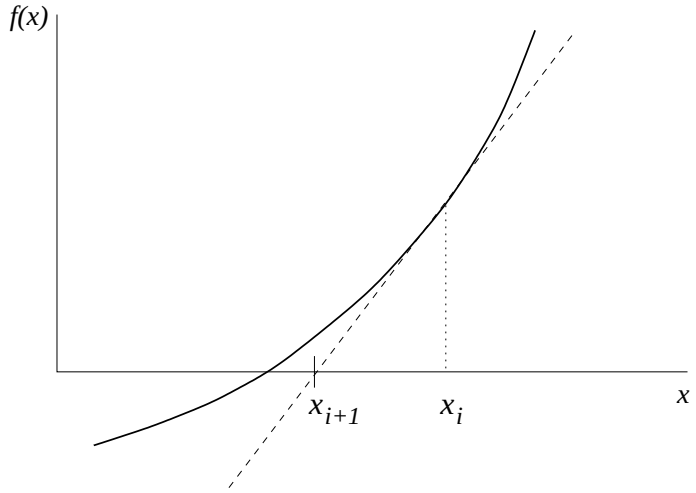

Abbildung 7.1: Newton–Verfahren

Der Algorithmus ergibt sich ausgehend von einem Startwert $x^{(0)}$ zu:

(1) Berechne $x^{(i+1)} = x^{(i)} - f(x^{(i)})/f'(x^{(i)})$

(2) Falls $|x^{(i+1)} - x^{(i)}| < \varepsilon$, beende den Algorithmus,
ansonsten setzte $i = i + 1$ und gehe zu (1)

Für die bereits bekannte Funktion $f(x) = 1 - x^2/2$ erhalten wir ausgehend von $x^{(0)} = 1$:

```
x( 0) = 1.00000000000000  (Fehler: 0.41421356237310)
x( 1) = 1.50000000000000  (Fehler: 0.08578643762690)
x( 2) = 1.41666666666667  (Fehler: 0.00245310429357)
x( 3) = 1.41421568627451  (Fehler: 0.00000212390141)
x( 4) = 1.41421356237469  (Fehler: 0.00000000000159)
x( 5) = 1.41421356237310  (Fehler: 0.00000000000000)
```

Dies geht offenbar viel schneller als die anderen Verfahren. Tatsächlich kann man beweisen, dass die Fehlerschranke für zweimal stetig differenzierbare Funktionen f mit $f'(x^*) \neq 0$ eine Gleichung der Form

$$e_{i+1} = Ce_i^2$$

erfüllt – man spricht von quadratischer Konvergenz. Auch quadratische Konvergenz kann man in der Ergebnisfolge leicht erkennen: die Anzahl der korrekten Stellen verdoppelt sich dabei in jedem Schritt.

Wie die Fixpunktiteration konvergiert auch das Newton-Verfahren nur lokal, d. h. für Startwerte $x^{(0)}$, die nahe der gesuchten Nullstelle x^* liegen – das Newton–Verfahren ist *lokal quadratisch konvergent*.

Warum besitzt das Newton–Verfahren diese schnelle Konvergenz? Dies liegt an der Tatsache, dass der Abstand der Tangente g zur Funktion f quadratisch von $x - x^{(i)}$ abhängt, d. h. im i–ten Schritt haben wir

$$g(x) = f(x) + \mathcal{O}((x - x^{(i)})^2)$$

also

$$|g(x^*)| \leq |f(x^*)| + C_1(x^* - x^{(i)})^2 = C_1(x^* - x^{(i)})^2.$$

Daraus folgt

$$|g(x^{(i+1)}) - g(x^*)| = |g(x^*)| \leq C_1(x^* - x^{(i)})^2.$$

Da g eine Gerade mit Steigung $f'(x^{(i)})$ ist, ist die Umkehrfunktion g^{-1} von g Lipschitz stetig mit der Lipschitzkonstante $L = 1/f'(x^{(i)})$. Also folgt

$$|x^{(i+1)} - x^*| = |g^{-1}(g(x^{(i+1)})) - g^{-1}(g(x^*))| \leq L|g(x^{(i+1)}) - g(x^*)| \leq LC_1(x^* - x^{(i)})^2,$$

also gerade die quadratische Konvergenz mit $C = LC_1$.

Das Newton–Verfahren lässt sich auch auf Nullstellenprobleme im $\mathbb{R}^n$ verallgemeinern. Wenn wir die Iterationsvorschrift in Schritt (1) des Newton–Verfahrens für $f : \mathbb{R} \to \mathbb{R}$ betrachten, stellt sich die Frage, wie eine geeignete Verallgemeinerung aussehen kann.

Sei dazu $f : \mathbb{R}^n \to \mathbb{R}^n$ nun eine vektorwertige Funktion. Die Ableitung $\frac{df}{dx}(x)$ an einer Stelle $x \in \mathbb{R}^n$ ist jetzt keine reelle Zahl mehr, sondern eine Matrix.

Nun können wir die Ableitung $\frac{df}{dx}(x^{(i)})$ nicht einfach in die Iterationsvorschrift für $x^{(i+1)}$ einsetzen, da man ja durch eine Matrix nicht teilen kann. Man kann also nicht einfach $f(x^{(i)})/f'(x^{(i)})$ durch $f(x^{(i)})/\frac{df}{dx}(x^{(i)})$ ersetzen, sondern muss, um denselben Effekt zu erzielen, die entsprechende Operation für Matrizen verwenden. Dem eindimensionalen Teilen durch $f'(x^{(i)})$ entspricht für Matrizen die Multiplikation mit $[\frac{df}{dx}(x^{(i)})]^{-1}$, der Iterationsschritt (2) wird also zu

$$x^{(i+1)} = x^{(i)} - \left[\frac{df}{dx}(x^{(i)})\right]^{-1} f(x^{(i)}).$$

Da das Invertieren von Matrizen aber numerisch sehr aufwändig ist, wird der Schritt nicht in der Form (7.1) implementiert. Statt dessen löst man das lineare Gleichungssystem $\frac{df}{dx}(x^{(i)})\Delta x^{(i)} = f(x^{(i)})$, dessen Lösung gerade den Vektor $\Delta x^{(i)} = [\frac{df}{dx}(x^{(i)})]^{-1} f(x^{(i)})$ auf der rechten Seite von (7.1) liefert.

Das Newton-Verfahren im $\mathbb{R}^n$ mit Startwert $x^{(0)}$ lautet also

(1) Löse das lineare Gleichungssystem $\frac{d}{dx} f(x^{(i)})\Delta x^{(i)} = f(x^{(i)})$
und berechne $x^{(i+1)} = x^{(i)} - \Delta x^{(i)}$

(2) Falls $\|\Delta x^{(i)}\| < \varepsilon$, beende den Algorithmus,
ansonsten setzte $i = i + 1$ und gehe zu (1)

Es gilt also: Um ein nichtlineares Gleichungssystem mit Hilfe des Newton–Verfahrens zu lösen, muss eine Folge von linearen Gleichungssystemen iterativ gelöst werden.

Kapitel 8

Optimierung

In diesem Kapitel wollen wir uns damit beschäftigen, wie wir numerisch Extremstellen und -werte berechnen können. Wir beschränken uns dabei auf Minimierungsprobleme (denn o. B. d. A. kann ein Maximierungsproblem durch Multiplikation mit „-1“ in ein Minimierungsproblem umgeschrieben werden). Teilweise nehmen wir an, dass es eine eindeutige Minimalstelle gibt und ebenso nehmen wir manchmal die (strenge) Konvexität der Funktion an.

8.1 Liniensuchverfahren

In Abschnitt 2.5.2 haben wir ein lineares Gleichungssystem gelöst, indem wir ein Optimierungsproblem gefunden haben, dessen Lösung die Lösung des ursprünglichen Gleichungssystems war. Für allgemeine stetig differenzierbare Funktionen $f : \mathbb{R}^n \to \mathbb{R}$, die eine eindeutige Minimalstelle haben, können wir ähnlich wie in Abschnitt 2.5.2 vorgehen. Wir beginnen mit einem Startwert, bestimmen eine Suchrichtung und entlang dieser Linie (weswegen diese Verfahren „Liniensuchverfahren“ heißen) das Minimum der Funktion von $\mathbb{R}$ nach $\mathbb{R}$. Wenn die Suchrichtung mithilfe des negativen Gradienten bestimmt wird, spricht man von Gradientenverfahren. Anders als in Abschnitt 2.5.2, in dem die zu minimierende Funktion eine mehrdimensionale Parabel war, können wir hier die Schrittweite nicht analytisch bestimmen. Stattdessen benötigen wir einen Algorithmus, der uns auf den Linien die reellen Funktionen (approximativ) minimiert.

8.2 Intervallschachtelung

Wir betrachten eine zweimal stetig differenzierbare Funktion $f : \mathbb{R} \to \mathbb{R}$. Um ein Minimum zu finden, könnten wir die Funktion zweimal ableiten und überprüfen, wo $f'(x) = 0$ gilt. Wenn zudem $f''(x) > 0$ gilt, ist x eine Minimalstelle und $f(x)$ das Minimum, bei $f''(x) < 0$ ist x keine Minimalstelle, bei $f''(x) = 0$ wären weitere Untersuchungen nötig. Das berechnen der Richtungsableitungen kann u. U. aber sehr aufwändig sein, weswegen wir ein anderes Verfahren besprechen, das sogar für nicht-stetige eindimensionale Funktionen funktioniert, wenn diese unimodal sind.

Definition 8.1 Eine nicht notwendigerweise stetige Funktion $f : I \to \mathbb{R}$ mit $I \subset \mathbb{R}$ ein Intervall von Länge > 0 heißt *unimodal*, wenn sie eine eindeutige Minimalstelle x^* besitzt und auf $I \cap \{x|x < x^*\}$ streng monoton fallend sowie auf $I \cap \{x|x > x^*\}$ streng monoton steigend ist. Wir nehmen an, dass x^* im Inneren von I liegt. □

Für den Algorithmus braucht man als Eingabe eine Stelle l, die links des Minimums, und eine Stelle r, die rechts des Minimums liegt. Er werden nun zwei Stellen (nach einer noch zu bestimmenden Vorschrift) r' und l' gewählt, sodass $l < l' < r' < r$ gilt. Nun wird $f(l')$ mit $f(r')$ verglichen. Wenn $f(l') < f(r')$ gilt, so ist $x^* \in (l, r')$ (wobei wir nicht wissen, ob $x^* \in (l, l')$, $x^* \in (l', r')$ oder $x^* = l'$ gilt). Wenn $f(l') > f(r')$ gilt, so ist $x^* \in (l', r)$ (wobei wir nicht wissen, ob $x^* \in (l', r')$, $x^* \in (r', r)$ oder $x^* = r'$ gilt). Und wenn $f(l') = f(r')$ gilt, so ist $x^* \in (l', r')$. Im ersten Fall wählt man nun r' als neues r, im zweiten l' als neues l und im dritten beides. Dann startet man den Schritt erneut. Der dritte Fall ist dabei in der Praxis sehr selten und deshalb eher akademisch.

Grundsätzlich können die Stellen r' und l' frei gewählt werden. Für die algorithmische Berechnung des Verfahrens und vor allem für die Konvergenzgeschwindigkeitsanalyse ist es vorteilhaft, wenn die Stellen r, r', l', l immer im gleichen Verhältnis im betrachteten Intervall liegen. Dies funktioniert genau für den *goldenen Schnitt*[1] $\Phi = \frac{1+\sqrt{5}}{2}$. Wir teilen das Intervall (r, l) in jedem Schritt im Verhältnis $\Phi : 1 : \Phi = 1 : \varphi : 1$ (mit $\varphi = \Phi - 1 = \frac{1}{\Phi}$). Dies führt zu folgendem Algorithmus:

Algorithmus 8.2 (Finden der Minimalstelle einer unimodalen Funktion)
Eingabe: Startstellen `l<x*, r>x*`, Abbruch ε,

(1) Berechne `l' = l + (1-φ)(r-l)` und `r' = l + φ(r-l)`

(2) Wenn $|r' - l'| \geq \varepsilon$, berechne `a = f(l')-f(r')`, sonst setze `r = r', l=l'` und gehe zu (5).

(3a) Wenn $a < 0$: `r = r', r'=l', l' = l + (1-φ)(r-l)`

(3b) Wenn $a > 0$: `l = l', l'=r', r' = l + φ(r-l)`

(3c) Wenn $a = 0$: `r = r', l=l', l' = l + (1-φ)(r-l), r' = l + φ(r-l)`

(4) Gehe zu (2).

(5) Setze `x = (r+l):2`

Ausgabe: Minimalstelle `x`. □

Durch geschickte Speicherung von Auswertungen von f kann der Algorithmus deutlich schneller werden. Mit jeder Auswertung von f verkürzt sich das Intervall um den Faktor $\varphi \approx 0.618$. Der Algorithmus liefert für $\varepsilon \to 0$ x^* und wenn f in x^* stetig ist, kann damit das Minimum bestimmt werden.

[1] Den goldenen Schnitt kennt man nicht nur aus der Kunst, sondern auch in der Mathematik vom goldenen Rechteck, der goldenen Spirale und dem Pentagramm.

8.3 Das Gauß-Newton-Verfahren für nichtlineare Ausgleichsprobleme

Zum Abschluss dieses Kapitels wollen wir noch einmal zum Ausgleichsproblem zurück kommen, das wir in Abschnitt 2.1 eingeführt haben. Wir haben dort das lineare Ausgleichsproblem betrachtet, das wir hier mit leicht anderer Notation zunächst noch einmal zusammenfassen wollen:

Zu einer Matrix $A \in \mathbb{R}^{m\times n}$ mit $m > n$ und einem Vektor $z \in \mathbb{R}^m$ finde den Vektor $x \in \mathbb{R}^n$, der die (quadrierte) Norm

$$\|Ax - z\|_2^2$$

minimiert.

In der theoretischen Betrachtung haben wir gesehen, dass dieser Vektor x gerade die Lösung der Normalengleichungen

$$A^T Ax = A^T z$$

ist, die ein „gewöhnliches“ lineares Gleichungssystem im $\mathbb{R}^n$ darstellen, das z. B. mit dem Choleski-Verfahren gelöst werden kann.

Das **nichtlineare Ausgleichsproblem** ist, wie im linearen Fall, als ein Minimierungsproblem gegeben. Hierzu betrachten wir für $D \subseteq \mathbb{R}^n$ und $m > n$ eine zweimal stetig differenzierbare Abbildung

$$f : D \to \mathbb{R}^m$$

und wollen für diese Abbildung das Problem

$$\text{minimiere } g(x) := \|f(x)\|_2^2 \text{ über } x \in D \tag{8.1}$$

lösen.

Wie beim linearen Ausgleichsproblem ist die Interpretation und Anwendung dieses Problems wie folgt: Seien z_i Messwerte zu Parametern t_i für $i = 1, \ldots, m$. Auf Grund theoretischer Überlegungen (z. B. eines zu Grunde liegenden physikalischen Gesetzes) weiß man, dass eine Funktion $h : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}$ existiert, sodass die (idealen) Messwerte für einen geeigneten Parametervektor $x^* \in \mathbb{R}^n$ die Gleichung $h(t_i, x^*) = z_i$ erfüllen. Mit

$$f(x) = \begin{pmatrix} h(t_1, x) - z_1 \\ \vdots \\ h(t_m, x) - z_m \end{pmatrix}$$

würde also $f(x^*) = 0$ gelten. Da wir hier Messfehler einkalkulieren müssen, wird diese Gleichung üblicherweise nicht exakt sondern nur approximativ erfüllt sein, weswegen wir also nach einer Lösung des Ausgleichsproblems (8.1) suchen.

Ein Beispiel für eine solche Anwendung ist z. B. durch Daten für ein Populationswachstum gegeben. Hier ist (unter idealen Bedingungen) ein theoretisches Wachstum der Form $z = h(t, x) = x_1 e^{x_2 t}$ zu erwarten, also eine Funktion, die nichtlinear in $x = (x_1, x_2)^T$ ist.

Wir leiten den Algorithmus zur Lösung nichtlinearer Ausgleichsprobleme nun informell her. Wir wollen bei der Lösung des Problems (8.1) nur lokale Minima im Inneren von D

betrachten, also Minimalstellen auf dem Rand ∂D nicht berücksichtigen. Darüberhinaus wollen wir uns auf lokale Minimalstellen $x^* \in D$ von $g : \mathbb{R}^n \to \mathbb{R}$ beschränken, die die hinreichenden Bedingungen

$$Dg(x^*) = 0 \quad \text{und} \quad D^2 g(x^*) \text{ ist positiv definit} \tag{8.2}$$

erfüllen. Die Ableitung von $g(x) = \|f(x)\|_2^2 = f(x)^T f(x)$ erfüllt

$$Dg(x)^T = 2Df(x)^T f(x),$$

also müssen wir zum Finden von Kandidaten von Minimalstellen das $n \times n$-nichtlineare Gleichungssystem

$$G(x) := Df(x)^T f(x) = 0 \tag{8.3}$$

lösen. Wenn wir hierfür das Newton-Verfahren einsetzen, so erhalten wir mit der dortigen Schreibweise die Iteration $x^{(i+1)} = x^{(i)} - \Delta x^{(i)}$ mit den iterativ zu lösenden Gleichungssystemen

$$DG(x^{(i)})\Delta x^{(i)} = G(x^{(i)}), \quad i = 0, 1, 2, \ldots, \tag{8.4}$$

Falls ein lokales Minimum x^* mit (8.2) existiert, so ist

$$\frac{1}{2} D^2 g(x) = DG(x) = D^2 f(x)^T f(x) + Df(x)^T Df(x)$$

in $x = x^*$ positiv definit, also auch für alle x in einer Umgebung von x^*, weswegen $DG(x)$ insbesondere invertierbar und das Newton-Verfahren anwendbar ist.

Falls das Ausgleichsproblem tatsächlich ein lösbares Gleichungssystem ist (wenn also in der obigen Interpretation keine Messfehler vorliegen), so gilt $f(x^*) = 0$; das Problem heißt dann *kompatibel*. Im diesem Fall gilt

$$DG(x^*) = Df(x^*)^T Df(x^*).$$

Auch wenn Kompatibilität ein Idealfall ist und in der Praxis kaum auftritt, so werden wir doch vereinfachend annehmen, dass $f(x^*)$ für x nahe bei x^* nahe bei Null liegt, also

$$DG(x) \approx Df(x)^T Df(x)$$

gilt. Auf Basis dieser informellen Überlegung ersetzen wir in der Iterationsvorschrift (8.4) die Ableitung $DG(x^{(i)})$ durch $Df(x^{(i)})^T Df(x^{(i)})$. Damit vermeiden wir die Verwendung der zweiten Ableitung und erhalten

$$Df(x^{(i)})^T Df(x^{(i)})\Delta x^{(i)} = G(x^{(i)}) = Df(x^{(i)})^T f(x^{(i)}), \quad i = 0, 1, 2, \ldots \tag{8.5}$$

Dies sind gerade die Normalengleichungen zu dem linearen Ausgleichsproblem

$$\text{minimiere } \|Df(x^{(i)})\Delta x^{(i)} - f(x^{(i)})\|_2^2,$$

die wir für die Iteration nun verwenden. Insgesamt führt dies auf den folgenden Algorithmus, das sogenannte **Gauß-Newton-Verfahren**:

Gegeben sei eine Funktion $f : \mathbb{R}^n \to \mathbb{R}^m$, ihre Ableitung $Df : \mathbb{R}^n \to \mathbb{R}^{m \times n}$ sowie ein Anfangswert $x^{(0)} \in \mathbb{R}^n$ und eine gewünschte Genauigkeit $\varepsilon > 0$. Setze $i := 0$.

(1) Löse das lineare Ausgleichsproblem $\min \|Df(x^{(i)})\Delta x^{(i)} - f(x^{(i)})\|_2^2$
und berechne $x^{(i+1)} = x^{(i)} - \Delta x^{(i)}$

(2) Falls $\|\Delta x^{(i)}\| < \varepsilon$, beende den Algorithmus,
ansonsten setze $i := i + 1$ und gehe zu (1)

Bemerkung 8.3 Ebenso, wie wir im Newton-Verfahren eine Folge von linearen Gleichungssystemen zur Lösung des nichtlinearen Gleichungssystems lösen müssen, lösen wir hier eine Folge linearer Ausgleichsprobleme zur Lösung des nichtlinearen Ausgleichsproblems. □

Man kann beweisen, dass das Verfahren trotz des „Fehlers“ in der Herleitung lokal gegen die korrekte Lösung konvergiert. Allerdings ist die Konvergenz i. A. nicht mehr quadratisch und wird um so langsamer, je größer $f(x^*)$ ist, d. h. je weniger kompatibel das Problem ist.

Kapitel 9

Modellprojekt 2: Die Kühlrippe

Informationen zu diesem Modellprojekt erhalten Sie in der Vorlesung oder in dem Buch [Bollhöfer und Mehrmann, 2004, Kapitel 2.2].

Kapitel 10

Finite Differenzen

In diesem Kapitel behandeln wir die Finite Differenzen-Methode zur Lösung der stationären Wärmeleitungsgleichung

$$w\Delta T(x,y,t) = 0, \tag{10.1}$$

also einer parabolischen partiellen Differentialgleichung, mit den Dirichlet-Randbedingungen

$$T(x,0,t) = g(x,t) \tag{10.2}$$

und den (gemischten oder) Cauchy-Randbedingung

$$\frac{\partial T(x,y,t)}{\partial \nu} = -\alpha(T(x,y,t) - T_U). \tag{10.3}$$

Um die Grundprinzipien zu veranschaulichen, vereinfachen wir das Problem zunächst noch etwas: wir betrachten einen vertikalen eindimensionalen Schnitt durch das zweidimensionale Modell. Die x-Variable wird dadurch überflüssig (da sie ja konstant ist), ebenso fallen die Randbedingungen am rechten und linken Rand weg. Es bleibt also die Gleichung

$$w \cdot \frac{\partial^2 T(y)}{\partial y^2} = 0. \tag{10.4}$$

mit der Randbedingung am unteren Rand

$$T(0) = g, \tag{10.5}$$

(g ist jetzt nur noch eine reelle Zahl) und der Randbedingung am oberen Rand

$$\frac{\partial T(y)}{\partial y} = -\alpha(T(y) - T_U) \quad \text{in } y = y^*. \tag{10.6}$$

Physikalisch ist dies nun kein besonders sinnvolles Modell mehr. Es ist aber nützlich, die Grundprinzipien unseres numerischen Schemas zu entwickeln.

10.1 Finite Differenzen in 1d

Das Finite Differenzen-Verfahren beruht auf der Approximation aller Ableitungen durch Differenzenquotienten in endlich vielen Gitterpunkten. Statt einer Gleichung, die Ableitungen in unendlich vielen Punkten enthält, erhält man so ein ein System von Gleichungen, das endlich viele Differenzen enthält, daher Finite (=endlich viele) Differenzen. In den Gleichungen (10.4)–(10.6) treten erste und zweite Ableitungen der Funktion T auf, weswegen wir Approximationen für diese betrachten müssen.

Für die erste Ableitung existieren dabei drei verschiedene Möglichkeiten, nämlich

$$\begin{aligned}
\text{Vorwärtsdifferenz:} \quad & \frac{\partial T(y)}{\partial y} \approx \frac{T(y+h)-T(y)}{h} \\
\text{Rückwärtsdifferenz:} \quad & \frac{\partial T(y)}{\partial y} \approx \frac{T(y)-T(y-h)}{h} \\
\text{zentrale Differenz:} \quad & \frac{\partial T(y)}{\partial y} \approx \frac{T(y+h)-T(y-h)}{2h}
\end{aligned}$$

wobei $h > 0$ jeweils eine kleine Zahl, die sogenannte Schrittweite ist.

Für die zweite Ableitung verwenden wir die zentrale Differenz zweiter Ordnung

$$\frac{\partial^2 T(y)}{\partial y^2} \approx \frac{T(y-h) - 2T(y) + T(y+h)}{h^2}.$$

Mit Hilfe der Taylor-Formel kann man zeigen, dass Vorwärts- und Rückwärtsdifferenz eine Genauigkeit der Ordnung $\mathcal{O}(h)$ besitzen, während die beiden zentralen Differenzen die Ordnung $\mathcal{O}(h^2)$ erreichen.

Wir wollen (10.4)–(10.6) nun für $y \in [0, y^*]$ lösen. Die Idee der Finiten Differenzen liegt nun darin, Gitterpunkte $y_i = ih$, $i = 0, \ldots, n$ mit Schrittweite $h = y^*/n$ einzuführen und in $y = y_i$ eine Approximation $u_i \approx T(y_i)$ zu berechnen, indem man alle auftretenden Ableitungen durch entsprechende Differenzen ersetzt, bei denen $y_i \pm h$ gerade benachbarten Gitterpunkten $y_{i\pm1}$ entsprechen. Wie beim Euler-Verfahren für gewöhnliche Differentialgleichungen erhalten wir numerisch also "lediglich" eine approximative Lösung unseres Problems über die Werte in den Gitterpunkten.

Wir beginnen mit der Wärmeleitungsgleichung (10.4). Die Ersetzung der Ableitung liefert

$$0 = w \cdot \frac{\partial^2 T(y)}{\partial y^2} \approx w \cdot \frac{T(y-h) - 2T(y) + T(y+h)}{h^2}.$$

Setzen wir hier nun die Gitterpunkte y_i, $i = 0, \ldots, n$ und die Approximationen $u_i \approx T(y_i)$ ein und setzen statt der exakten Formel die Approximation gleich 0, so erhalten wir

$$w \cdot \frac{u_{i-1} - 2u_i + u_{i+1}}{h^2} = 0, \qquad i = 0, \ldots, n, \tag{10.7}$$

was ein System von $n+1$ (linearen) Gleichungen für die Werte u_i darstellt. Beachte, dass in diesen Gleichungen die "künstlichen" Werte u_{-1} (in der Gleichung für $i = 0$) und u_{n+1} (in der Gleichung für $i = n$) auftreten. Diese werden wir nun mit Hilfe der Randbedingungen wieder entfernen.

Die Randbedingung (10.5) lässt sich einfach einfügen: wir setzen $u_0 = g$, so dass wir die Gleichung für $i = 0$ in (10.7) entfernen (womit wir insbesondere u_{-1} loswerden) und die Gleichung für $i = 1$ durch

$$w \cdot \frac{g - 2u_1 + u_2}{h^2} = 0$$

ersetzen können.

Die Randbedingung (10.6) ist etwas komplizierter zu behandeln, denn dort tritt eine erste Ableitung auf, die wir wieder durch eine Differenz ersetzen müssen. Da die zentrale Differenz die beste Approximation liefert, verwenden wir diese und erhalten als Diskretisierung

$$\frac{u_{n+1} - u_{n-1}}{2h} = -\alpha(u_n - T_U).$$

Dies können wir nach u_{n+1} auflösen und erhalten so

$$u_{n+1} = -2h\alpha(u_n - T_U) + u_{n-1}.$$

Dies können wir in die Gleichung (10.7) für $i = n$ einsetzen und erhalten so

$$0 = w \cdot \frac{u_{n-1} - 2u_n + u_{n+1}}{h^2} = w \cdot \frac{2u_{n-1} - 2u_n - 2h\alpha(u_n - T_U)}{h^2}.$$

Zusammengefasst, skaliert und so sortiert, dass alle u_i–Terme links und alle anderen Terme rechts stehen, erhalten wir so das vollständige Gleichungssystem

$$\begin{aligned} w \cdot \frac{2u_1 - u_2}{h^2} &= w \cdot \frac{g}{h^2} \\ w \cdot \frac{-u_{i-1} + 2u_i - u_{i+1}}{h^2} &= 0, \qquad i = 2, \ldots, n-1 \\ w \cdot \frac{-u_{n-1} + (1 + h\alpha)u_n}{h^2} &= w\frac{\alpha}{h}T_U. \end{aligned} \tag{10.8}$$

Wenn wir dieses lineare Gleichungssystem lösen, haben wir die gesuchte Approximation berechnet. Um dieses Gleichungssystem am Rechner zu lösen, muss es in der Matrix-Vektor-Form

$$\underbrace{\frac{w}{h^2}\begin{pmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -1 & 1 + h\alpha \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_{n-1} \\ u_n \end{pmatrix}}_{u} = \underbrace{w\begin{pmatrix} \frac{g}{h^2} \\ 0 \\ \vdots \\ 0 \\ \frac{\alpha}{h}T_U \end{pmatrix}}_{b} \tag{10.9}$$

geschrieben werden (leere Einträge in der Matrix bedeuten "0"). In dieser "$Au = b$"-Form kann das Gleichungssystem z. B. in MAPLE oder MATLAB gelöst werden.

10.2 Finite Differenzen in 2d

Für unser zweidimensionales Rechengebiet $[-x^*, x^*] \times [0, y^*]$ müssen wir die Gleichung

$$w\left(\frac{\partial^2 T(x,y)}{\partial x^2} + \frac{\partial^2 T(x,y)}{\partial y^2}\right) = 0. \tag{10.10}$$

mit den Randbedingungen

$$T(x,0) = g(x) \tag{10.11}$$

am unteren Rand und

$$\frac{\partial T(x,y)}{\partial \nu} = -\alpha(T(x,y) - T_U) \tag{10.12}$$

an den übrigen Rändern lösen.

Wenn wir in x und y–Richtung die Schrittweiten h_x und h_y verwenden, sind die Approximationen von Ableitungen in 2d durch Differenzen völlig analog zu denen in 1d:

$$\begin{aligned}
\frac{\partial T(x,y)}{\partial x} &\approx \frac{T(x+h_x,y) - T(x-h_x,y)}{2h_x} \\
\frac{\partial T(x,y)}{\partial y} &\approx \frac{T(x,y+h_y) - T(x,y-h_y)}{2h_y} \\
\frac{\partial^2 T(x,y)}{\partial x^2} &\approx \frac{T(x-h_x,y) - 2T(x,y) + T(x+h_x,y)}{h_x^2} \\
\frac{\partial^2 T(x,y)}{\partial y^2} &\approx \frac{T(x,y-h_y) - 2T(x,y) + T(x,y+h_y)}{h_y^2}
\end{aligned}$$

Dies führt analog zum 1d Fall auf die approximative Gleichung

$$\begin{aligned}
0 &= w\left(\frac{\partial^2 T(x,y)}{\partial x^2} + \frac{\partial^2 T(x,y)}{\partial y^2}\right) \\
&\approx w\Bigg(\frac{T(x-h_x,y) - 2T(x,y) + T(x+h_x,y)}{h_x^2} \\
&\qquad + \frac{T(x,y-h_y) - 2T(x,y) + T(x,y+h_y)}{h_y^2}\Bigg).
\end{aligned}$$

Diese Gleichung enthält jeweils den zentralen Punkt (x,y) und vier um diesen herum sternförmig angeordnete weitere Punkte. Man spricht daher von einem Differenzenstern.

Um zu einem am Rechner implementierbaren Schema zu kommen, überdecken wir das Rechengebiet mit einem zweidimensionalen Gitter mit Punkten

$$x_j = jh_x - x^*,\ j = 0,\ldots,n_x = 2x^*/h_x, \qquad y_k = kh_yk,\ k = 0,\ldots,n_y = y^*/h_y$$

in das wir den Differenzenstern "einbetten". Schreiben wir die Approximationen als $u_{j,k} \approx T(x_j, y_k)$ und setzen wiederum die approximative Gleichung gleich 0, so erhalten wir die Gleichungen

$$w\left(\frac{u_{j-1,k} - 2u_{j,k} + u_{j+1,k}}{h_x^2} + \frac{u_{j,k-1} - 2u_{j,k} + u_{j,k+1}}{h_y^2}\right) = 0 \tag{10.13}$$

für $j = 0,\ldots,n_x$, $k = 0,\ldots,n_y$.

Genau wie im eindimensionalen Fall erhalten wir hier "überstehende" Gitterpunkte in den Gleichungen für $j = 0$, $j = n_x$, $k = 0$, $k = n_y$, die wir mit Hilfe der Randbedingungen (10.11), (10.12) wieder entfernen können.

Hierzu betrachten wir die zwei Summanden in (10.13) separat. Da die Werte $u_{j,0}$ (also für $k = 0$) durch die Randbedingung (10.11) bestimmt sind, können wir diese Gleichungen weglassen und in den Gleichungen für $k = 1$ den zweiten Summanden ersetzen durch

$$w \cdot \frac{u_{j,0} - 2u_{j,1} + u_{j,2}}{h_y^2} = w \cdot \frac{g(x_j) - 2u_{j,1} + u_{j,2}}{h_y^2}.$$

Am oberen Rand ($k = n_y$) erhalten wir mit $\nu = y$ aus der Randbedingung (10.12) diskretisiert mit dem zentralen Differenzenquotienten die Gleichung

$$\frac{u_{j,n_y+1} - u_{j,n_y-1}}{2h_y} = -\alpha(u_{j,n_y} - T_U).$$

Aufgelöst nach u_{j,n_y+1} ergibt dies

$$u_{j,n_y+1} = -2h_y\alpha(u_{j,n_y} - T_U) + u_{j,n_y-1},$$

was eingesetzt in die Gleichungen für $k = n_y$ für den zweiten Summanden

$$w \cdot \frac{u_{j,n_y-1} - 2u_{j,n_y} + u_{j,n_y+1}}{h_y^2} = w \cdot \frac{2u_{j,n_y-1} - 2u_{j,n_y} - 2h_x\alpha(u_{j,n_y} - T_U)}{h_y^2}$$

liefert. Am rechten Rand erhalten wir analog (durch Vertauschen von x und y sowie j und k) mit $\nu = x$

$$u_{n_x+1,k} = -2h_x\alpha(u_{n_x,k} - T_U) + u_{n_x-1,k},$$

was für den ersten Summanden auf

$$w \cdot \frac{u_{n_x-1,k} - 2u_{n_x,k} + u_{n_x+1,k}}{h_x^2} = w \cdot \frac{2u_{n_x-1,k} - 2u_{n_x,k} - 2h_x\alpha(u_{n_x,k} - T_U)}{h_x^2}$$

führt. Am linken Rand erhalten wir wegen $\nu = -x$ in (10.12) ein anderes Vorzeichen, also

$$-\frac{u_{1,k} - u_{-1,k}}{2h_x} = -\alpha(u_{0,k} - T_U)$$

und damit

$$u_{-1,k} = -2h_x\alpha(u_{0,k} - T_U) + u_{1,k}.$$

Damit erhalten wir den neuen ersten Summanden

$$w \cdot \frac{u_{-1,k} - 2u_{0,k} + u_{1,k}}{h_x^2} = w \cdot \frac{-2h_x\alpha(u_{0,k} - T_U) - 2u_{0,k} + 2u_{1,k}}{h_x^2}.$$

Insgesamt führt dies nach Herauskürzen von w auf das komplette Gleichungssystem

$$
\begin{array}{lrcll}
(1) & \frac{-u_{j-1,k}+2u_{j,k}-u_{j+1,k}}{h_x^2} + \frac{-u_{j,k-1}+2u_{j,k}-u_{j,k+1}}{h_y^2} & = & 0, & \begin{array}{l} j=1,\ldots,n_x-1 \\ k=2,\ldots,n_y-1 \end{array} \\
(2) & \frac{-u_{j-1,1}+2u_{j,1}-u_{j+1,1}}{h_x^2} + \frac{2u_{j,1}-u_{j,2}}{h_y^2} & = & \frac{1}{h_y^2} g(x_j), & j=1,\ldots,n_x-1 \\
(3) & \frac{(2+2h_x\alpha)u_{0,k}-2u_{1,k}}{h_x^2} + \frac{-u_{0,k-1}+2u_{0,k}-u_{0,k+1}}{h_y^2} & = & \frac{2\alpha}{h_x} T_U, & k=2,\ldots,n_y-1 \\
(4) & \frac{-2u_{n_x-1,k}+(2+2h_x\alpha)u_{n_x,k}}{h_x^2} + \frac{-u_{n_x,k-1}+2u_{n_x,k}-u_{n_x,k+1}}{h_y^2} & = & \frac{2\alpha}{h_x} T_U, & k=2,\ldots,n_y-1 \\
(5) & \frac{-u_{j-1,n_y}+2u_{j,n_y}-u_{j+1,n_y}}{h_x^2} + \frac{-2u_{j,n_y-1}+(2+2h_y\alpha)u_{j,n_y}}{h_y^2} & = & \frac{2\alpha}{h_y} T_U, & j=1,\ldots,n_x-1 \\
(6) & \frac{(2+2h_x\alpha)u_{0,1}-2u_{1,1}}{h_x^2} + \frac{2u_{0,1}-u_{0,2}}{h_y^2} & = & \frac{1}{h_y^2} g(x_0) + \frac{2\alpha}{h_x} T_U & \\
(7) & \frac{-2u_{n_x-1,1}+(2+2h_x\alpha)u_{n_x,1}}{h_x^2} + \frac{2u_{n_x,1}-u_{n_x,2}}{h_y^2} & = & \frac{1}{h_y^2} g(x_{n_x}) + \frac{2\alpha}{h_x} T_U & \\
(8) & \frac{(2+2h_x\alpha)u_{0,n_y}-2u_{1,n_y}}{h_x^2} + \frac{-2u_{0,n_y-1}+(2+2h_y\alpha)u_{0,n_y}}{h_y^2} & = & \frac{2\alpha}{h_x} T_U + \frac{2\alpha}{h_y} T_U & \\
(9) & \frac{-2u_{n_x-1,n_y}+(2+2h_x\alpha)u_{n_x,n_y}}{h_x^2} + \frac{-2u_{n_x,n_y-1}+(2+2h_y\alpha)u_{n_x,n_y}}{h_y^2} & = & \frac{2\alpha}{h_x} T_U + \frac{2\alpha}{h_y} T_U &
\end{array}
$$

Hierbei stehen die Gleichungen für die folgenden Gitterpunkte:

(1): innere Punkte
(2): unterer Rand ohne Eckpunkte
(3): linker Rand ohne Eckpunkte
(4): rechter Rand ohne Eckpunkte
(5): oberer Rand ohne Eckpunkte
(6): linker unterer Eckpunkt
(7): rechter unterer Eckpunkt
(8): linker oberer Eckpunkt
(9): rechter oberer Eckpunkt

Um dieses Gleichungssystem im Rechner lösbar zu machen, muss es in die Form

$$AU = B \tag{10.14}$$

gebracht werden, wobei A eine Matrix und U und B Vektoren sind. Hierzu definieren wir den Vektor

$$U = \begin{pmatrix} U_1 \\ U_2 \\ \vdots \\ U_{N-1} \\ U_N \end{pmatrix} = \begin{pmatrix} u_{0,1} \\ u_{1,1} \\ \vdots \\ u_{n_x-1,n_y} \\ u_{n_x,n_y} \end{pmatrix}.$$

Im Vektor U stehen also die Werte der Gitterpunkte zeilenweise von unten nach oben angeordnet. Damit kann die Matrix A und der Vektor B einfach bestimmt werden, indem man in jede Zeile der Matrix A die Koeffizienten der entsprechenden Gleichung und in den Vektor B die rechten Seiten der entsprechenden Gleichungen einträgt.

Wir führen dies für $h_x = h_y = h$ durch und teilen dazu Gleichungen (3)–(7) durch 2 und Gleichungen (8)–(9) durch 4. Das ergibt

$$
\begin{array}{lrcll}
(1) & \frac{-u_{j-1,k}-u_{j,k-1}+4u_{j,k}-u_{j+1,k}-u_{j,k+1}}{h^2} & = & 0, & \begin{array}{l} j=1,\ldots,n_x-1 \\ k=2,\ldots,n_y-1 \end{array} \\
(2) & \frac{-u_{j-1,1}+4u_{j,1}-u_{j+1,1}-u_{j,2}}{h^2} & = & \frac{1}{h^2}g(x_j), & j=1,\ldots,n_x-1 \\
(3) & \frac{-u_{0,k-1}/2+(2+h\alpha)u_{0,k}-u_{1,k}-u_{0,k+1}/2}{h^2} & = & \frac{\alpha}{h}T_U, & k=2,\ldots,n_y-1 \\
(4) & \frac{-u_{n_x-1,k}-u_{n_x,k-1}/2+(2+h\alpha)u_{n_x,k}-u_{n_x,k+1}/2}{h^2} & = & \frac{\alpha}{h}T_U, & k=2,\ldots,n_y-1 \\
(5) & \frac{-u_{j-1,n_y}/2-u_{j,n_y-1}+(2+h\alpha)u_{j,n_y}-u_{j+1,n_y}/2}{h^2} & = & \frac{\alpha}{h}T_U, & j=1,\ldots,n_x-1 \\
(6) & \frac{(2+h\alpha)u_{0,1}-u_{1,1}-u_{0,2}/2}{h^2} & = & \frac{1}{2h^2}g(x_0)+\frac{\alpha}{h}T_U & \\
(7) & \frac{-u_{n_x-1,1}+(2+h\alpha)u_{n_x,1}-u_{n_x,2}/2}{h^2} & = & \frac{1}{2h^2}g(x_{n_x})+\frac{\alpha}{h}T_U & \\
(8) & \frac{-u_{0,n_y-1}/2+(1+h\alpha)u_{0,n_y}-u_{1,n_y}/2}{h^2} & = & \frac{\alpha}{h}T_U & \\
(9) & \frac{-u_{n_x-1,n_y}/2-u_{n_x,n_y-1}/2+(1+h\alpha)u_{n_x,n_y}}{h^2} & = & \frac{\alpha}{h}T_U &
\end{array}
$$

Damit erhalten wir eine $(n_x+1)n_y \times (n_x+1)n_y$–Matrix

$$
A = \frac{1}{h^2}\begin{pmatrix}
A_1 & D & & & & & & \\
D & A_2 & D & & & & & \\
 & D & A_3 & D & & & & \\
 & & \ddots & \ddots & \ddots & & & \\
 & & & D & A_{n_y-2} & D & & \\
 & & & & D & A_{n_y-1} & D & \\
 & & & & & D & A_{n_y} &
\end{pmatrix},
$$

in der A_k, $k=1,\ldots,n_y$ und D jeweils $(n_x+1)\times(n_x+1)$–Matrizen sind mit

$$
A_1 = A_{n_y} = \begin{pmatrix}
2+h\alpha & -1/2 & & & \\
-1/2 & 2+h\alpha & -1/2 & & \\
 & \ddots & \ddots & \ddots & \\
 & & -1/2 & 2+h\alpha & -1/2 \\
 & & & -1/2 & 1+h\alpha
\end{pmatrix},
$$

$$
A_2 = A_3 = \ldots = A_{n_y-1} = \begin{pmatrix}
2+h\alpha & -1 & & & \\
-1 & 4 & -1 & & \\
 & \ddots & \ddots & \ddots & \\
 & & -1 & 4 & -1 \\
 & & & -1 & 2+h\alpha
\end{pmatrix},
$$

und der Diagonalmatrix

$$D = \begin{pmatrix} -1/2 & & & & \\ & -1 & & & \\ & & \ddots & & \\ & & & -1 & \\ & & & & -1/2 \end{pmatrix}$$

(in all diesen Matrizen sind nicht dargestellte Einträge immer gleich 0). Beachte, dass die Matrix A symmetrisch ist (weil wir die Gleichungen (3)–(9) durch 2 bzw. 4 geteilt haben) und viele Einträge gleich Null besitzt — beides Eigenschaften, die man bei der numerischen Lösung des Gleichungssystems ausnutzen kann.

Der Vektor B auf der rechten Seite ist ähnlich aufgebaut. Es gilt

$$B = \begin{pmatrix} B_1 \\ B_2 \\ \vdots \\ B_{n_y} \end{pmatrix}$$

wobei die B_k, $k = 1, \ldots, n_y$, $n_x + 1$–dimensionale Vektoren sind mit

$$B_1 = \begin{pmatrix} \frac{g(x_0)}{2h^2} + \frac{\alpha}{h} T_U \\ \frac{g(x_1)}{h^2} \\ \frac{g(x_2)}{h^2} \\ \vdots \\ \frac{g(x_{n_x-1})}{h^2} \\ \frac{g(x_{n_x})}{2h^2} + \frac{\alpha}{h} T_U \end{pmatrix}, \quad B_2 = \ldots = B_{n_y-1} = \begin{pmatrix} \frac{\alpha}{h} T_U \\ 0 \\ 0 \\ \vdots \\ 0 \\ \frac{\alpha}{h} T_U \end{pmatrix}, \qquad B_{n_y} = \begin{pmatrix} \frac{\alpha}{h} T_U \\ \frac{\alpha}{h} T_U \\ \frac{\alpha}{h} T_U \\ \vdots \\ \frac{\alpha}{h} T_U \\ \frac{\alpha}{h} T_U \end{pmatrix}.$$

Beachte: Für $n_x + 1$ Knoten in x–Richtung und $n_y + 1$ Knoten in y–Richtung besitzt das Gleichungssystem $(n_x + 1)n_y$ Unbekannte.

10.3 Fehlerabschätzung

Wie genau ist die mit dieser Methode erhaltene Approximation? Genau wie bei den gewöhnlichen Differentialgleichungen betrachtet man dazu zunächst die *Konsistenz*, d. h. die Genauigkeit, mit der wir die Ableitungen approximiert haben. Formal sagen wir, dass das Schema konsistent mit Ordnung $O(h^p)$ ist, wenn die exakte Lösung T die Gleichungen des Schemas mit Ordnung $O(h^p)$ erfüllt, d. h. wenn

$$A \begin{pmatrix} T(x_0, y_1) \\ T(x_1, y_1) \\ \vdots \\ T(x_{n_x-1}, y_{n_y}) \\ T(x_{n_x}, y_{n_y}) \end{pmatrix} = B + O(h^p)$$

gilt.

Da wir in unserem Schema alle Ableitungen von T mit $O(h^2)$ approximiert haben, gilt:

Satz 10.1 Das Schema hat die Konsistenzordnung $O(h^2)$.

Für die Qualität der Abschätzung ist allerdings die Konvergenz entscheidend. Wir sagen, dass das Schema konvergent mit Ordnung $O(h^p)$ ist, wenn

$$e_{j,k} = O(h^p) \quad \text{gilt für} \quad e_{j,k} := T(x_j, y_k) - u_{j,k}.$$

Genau wie bei den gewöhnlichen Differentialgleichungen erhofft man sich, dass die Konsistenzordnung sich auf die Konvergenzordnung überträgt. Hierzu benötigt man allerdings Bedingungen an die Koeffizienten des Schemas, d. h. an die Einträge der Matrix A. Der Grund dafür ist wie folgt: Für den Fehlervektor $e = (e_{0,1}, e_{1,1}, \ldots, e_{n_x,n_y})^T$ folgt mit $\widehat{T} = (T(x_0, y_1), T(x_1, y_1), \ldots, T(x_{n_x}, y_{n_y})^T$ aus der Konsistenz die Gleichung

$$Ae = A\widehat{T} - AU = B + O(h^p) - B = O(h^p).$$

Anders geschrieben gibt es also einen Vektor $r = O(h^p)$ mit $Ae = r$, also

$$e = A^{-1}r.$$

Mit kleiner werdenden Schrittweiten h wird nun einerseits r immer kleiner, andererseits aber A^{-1} immer größer. Das Produkt könnte daher mit kleiner werdenden Schrittweiten immer größer werden. Daher brauchen wir eine Bedingung an die Matrix A, die dieses verhindert. Eine solche Bedingung verwenden wir im folgenden Satz.

Satz 10.2 In jeder Zeile der Matrix sei der Diagonaleintrag > 0 und alle anderen Einträge ≤ 0. Dann ist das Schema konvergent mit Ordnung $O(h^p)$, wenn es konsistent mit Ordnung $O(h^p)$.

Da diese Bedingung bei unserem Schema offenbar erfüllt ist, ist unser Schema folglich konvergent mit Ordnung $O(h^2)$.

Zum Abschluss sollte erwähnt werden, dass es neben den Finiten Differenzen eine weitere wichtige numerische Methode zur Lösung partieller Differentialgleichungen gibt, nämlich die Finiten Elemente. Anstatt wie bei den Finiten Differenzen die Ableitungen zu diskretisieren, wird die partielle Differentialgleichung bei den Finiten Elementen zunächst (oft mit nicht unerheblichem mathematischen Aufwand) in eine Integralgleichung transformiert, bei der nur noch Integrale und keine Ableitungen auftreten. Diese Integrale werden dann mit einem numerischen Integrationsverfahren approximiert, was letzendlich wieder auf ein lineares Gleichungssystem führt, was dann numerisch zu lösen ist.

Im Eindimensionalen geht dies mit den zusamengesetzten Newton-Cotes Formeln, die wir in Kapitel 5 besprochen haben. In höheren Dimensionen verwendet man statt der Teilintervalle der zusamengesetzten Newton-Cotes Formeln allgemeinere Mengen (in 2d oft Dreiecke oder Vierecke), auf denen die Integrale approximiert werden.

Die Vorteile der Finiten Elemente sind zum einen, dass man z. B. mit Dreiecken deutlich kompliziertere Rechengebiete als mit den Differenzensternen flexibel behandeln kann. Zum anderen ist die Konvergenz dieser Methode für eine Reihe von Typen partieller Differentialgleichungen mathematisch besser verstanden. Für viele Typen allerdings ist die hier vorgestellte Finite Differenzenmethode durchaus gleichwertig — und mit geringeren mathematischen Kenntnissen zu verstehen.

Literaturverzeichnis

[Bollhöfer und Mehrmann, 2004] M. Bollhöfer, V. Mehrmann: Numerische Mathematik. Eine projektorientierte Einführung für Ingenieure, Mathematiker und Naturwissenschaftler. Vieweg, Wiesbaden, 2004.

[Philip, 2018] P. Philip: Numerical Methods for Mathematical Finance. Lecture Notes. LMU, München, 2018.