Kapitel 1

Einführung

1.1 Was ist numerische Mathematik?

Gegenstand der numerischen Mathematik (oder einfach *Numerik*) ist die näherungsweise Lösung mathematischer Probleme durch Zahlenwerte. Die Berechnung der Lösung erfolgt dabei durch einen *Algorithmus*, d.h. eine Folge von elementaren Anweisungen und Rechenoperationen, die sich auf einem Computer ausführen lassen. Ein solcher Algorithmus stützt sich oft auf Ergebnisse der reinen Mathematik und reflektiert mathematische Eigenschaften des Problems. Die zu behandelnden Probleme stammen oft aus den Ingenieurs- und Naturwissenschaften.
Ein typisches Problem der numerischen Mathematik ist das folgende:

Beispiel: Gesucht ist eine 2x diff'bare Funktion $y : [0,1] \to \mathbb{R}$ mit $y(0) = 0$, $y(1) = 1$, welche die Differentialgleichung $y''(x)+y(x) = 0$ erfüllt. Die eindeutige Lösung ist gegeben durch $y(x) = \sin(x)/\sin(1)$. Wie erhält man eine numerische Lösung?

Ansatz: Unterteilung von $[0,1]$ in n Teilintervalle der Länge $h := \frac{1}{n}$. Wir suchen Zahlenwerte für y an den Stellen $x = 0, h, 2h, ..., nh = 1$. Diesen Ansatz nennt man *Diskretisierung* des Problems.
Nach der Formel von Taylor gilt (falls $y(x)$ genügend oft diff'bar)

$$y(x+h) = y(x) + hy'(x) + \frac{h^2}{2}y''(x) + \frac{h^3}{6}y'''(x) + \frac{h^4}{24}y''''(\xi_1), \quad x \le \xi_1 \le x+h,$$

$$y(x-h) = y(x) - hy'(x) + \frac{h^2}{2}y''(x) - \frac{h^3}{6}y'''(x) + \frac{h^4}{24}y''''(\xi_2), \quad x-h \le \xi_2 \le x.$$

Addition dieser beiden Gleichungen und Subtraktion von $2y(x)$ liefert

$$\frac{y(x-h) - 2y(x) + y(x+h)}{h^2} = y''(x) + \frac{h^2}{24}\left(y''''(\xi_1) + y''''(\xi_2)\right) \tag{1.1}$$

Für große n (d.h. kleine h) ist also der Ausdruck auf der linken Seite von (1.1) eine gute Näherung für $y''(x)$. Es liegt also nahe, $y''(x) + y(x) = 0$ zu ersetzen durch

$$\frac{y(x-h) - 2y(x) + y(x+h)}{h^2} + y(x) = 0$$

$$\Leftrightarrow y(x-h) + (h^2-2)y(x) + y(x+h) = 0.$$

Setzen wir in diese Gleichung für x die Stellen ih ein und setzen $y_i := y(ih)$, $i = 1, ..., n-1$, so erhalten wir folgende Gleichungen für die y_i:

$$\begin{aligned} i = 1: &\quad 0 + (h^2-2)y_1 + y_2 = 0 \quad \text{wegen } y(0) = 0 \\ i = 2, \ldots, n-2: &\quad y_{i-1} + (h^2-2)y_i + y_{i+1} = 0 \\ i = n-1: &\quad y_{n-2} + (h^2-2)y_{n-1} + 1 = 0 \quad \text{wegen } y(1) = 1 \end{aligned}$$

Dies ist offensichtlich ein System von $n-1$ linearen Gleichungen für die $n-1$ unbekannten $y_1, \ldots, y_{n-1}$. Für $n = 4$ z.B. erhalten wir ($h = 0.25, h^2 - 2 = -1.9375$) das System $A\bar{y} = b$ mit

$$A = \begin{pmatrix} -1.9375 & 1 & 0 \\ 1 & -1.9375 & 1 \\ 0 & 1 & -1.9375 \end{pmatrix}, \quad b = \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix}, \quad \bar{y} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

Die Lösung (auf 4 Stellen genau) ist $\bar{y} = (0.2934, 0.5702, 0.8104)^T$. Die exakte Lösung auf 4 Stellen ist $(y(0.25), y(0.5), y(0.75))^T = (0.2940, 0.5697, 0.8101)^T$.

Bemerkung:

1. Wir haben einen Satz aus der reinen Mathematik (den Satz von Taylor) benutzt, um zu einer numerischen Lösung zu gelangen.

2. Die Lösung ist kein algebraischer Ausdruck wie $y(x) = \sin(x)/\sin(1)$, sondern ein Vektor, der zahlenmäßige Approximationen zu y an verschiedenen x-Werten (hier: $x = 0.25, 0.5, 0.75$) enthält.

1.2 Gleitpunktarithmetik

Meistens wird man numerische Berechnungen wie im obigen Beispiel nicht „per Hand“ durchführen, sondern auf einem Computer. Dazu benötigt man zunächst eine *Basis* $E \in \{2, 3, ...\}$ der Zahldarstellung. Jedes $x \in \mathbb{R}$ läßt sich dann darstellen als

$$x = \text{sign}(x) \cdot a \times E^b$$

mit sign(x) = Vorzeichen von x, $a =$ Mantisse $\in [E^{-1}, 1] \cup \{0\}$, $b =$ Exponent $\in \mathbb{Z}$.

Beispiel:

1. $x = \frac{10}{3}$, $E = 10 \Rightarrow x = +0.\bar{3} \times 10^1$, also $sign(x) = +$, $a = 0.\bar{3}$, $b = 1$

2. $x = 277.872$, $E = 10$, dann $x = +0.277872 \times 10^3$, also $sign(x) = +$, $a = 0.277872$, $b = 3$ ($x = +0.0277872 \times 10^4$ wäre keine zulässige Darstellung in diesem Sinne, da $0.0277872 < E^{-1} = 0.1$).

3. $x = -8$, $E = 2 \Rightarrow x = -1.0 \times 2^3$, also $sign(x) = -$, $a = 1.0$, $b = 3$. Ebenfalls eine gültige Darstellung gemäß obiger Definition wäre $x = -0.1 \times 2^4$ mit $a = 0.1$, $b = 4$ (beachte 0.1 (Dualsystem) entspricht 0.5 (Dezimalsystem)!)

Problem: Auf einen Computer mit endlich viel Speicher kann es keine unendlich langen Mantissen oder Exponenten geben! Viele Systeme beschränken sich auf Mantissenlänge $t = 8$ (sog. einfache Genauigkeit) und wenn genauer gerechnet werden muß, $t = 16$ (doppelte Genauigkeit). So ist z. B.

$$\pi \simeq 0.31415926 \times 10^1 \text{ (einf. Genauigkeit)}$$
$$\pi \simeq 0.31415926E + 01 \text{ (altern. Schreibweise im Dezimalsystem)}$$
$$\pi \simeq 0.3141592653589793 \times 10^1 \text{ (doppelte Genauigkeit)}$$

Rundung oder *Abschneiden* erfolgt bei Rechenoperationen, deren Ergebnis die erlaubte Mantissenlänge t überschreitet.

Beispiel: $\pi \simeq 0.314E + 01$ $(t = 3)$
$3.14^2 = 9.8596$, also $\pi^2 \simeq 0.985E+01$ (Abschneiden) oder $\pi^2 \simeq 0.986E+01$ (Aufrunden). In jedem Fall entsteht ein Rundungsfehler.

Bemerkung: Die obige Darstellung ist vereinfacht. Die interne Zahldarstellung in Computern erfolgt gewöhnlich nicht mit E=10 (dezimal), sondern $E = 2$ (dual), $E = 8$ (oktal) oder $E = 16$ (hexadezimal). Der Benutzer sieht davon aber meistens nichts, er kann „extern" mit E=10 arbeiten, wie man es gewohnt ist. Man beachte aber, daß mit $E = 2$ z.B. $x = \frac{1}{10}$ nicht exakt darstellbar ist (wohl aber mit $E = 10$), so daß auf den meisten System die Darstellung von 0.1 mit einem Rundungsfehler behaftet ist.
Viele Gleitpunkteigenschaften sind in einem sogenannten IEEE[1]-Standard festgelegt. Alle gängigen Systeme erfüllen diesen Standard. Man spricht deshalb anstatt von Gleitpunktarithmetik auch häufig von *IEEE-Arithmetik*.

Symbole für *arithmetische Ausnahmesituationen:* `NaN` (Not a Number) und `Infinity` zur Darstellung von Ergebnissen „unerlaubter"Operationen wie z.B. Division durch Null:

Beispiel:

$$1.0/0.0 \rightarrow \texttt{Infinity}$$
$$0.0/0.0 \rightarrow \texttt{NaN}$$

Das Auftreten dieser Symbole in numerischen Ergebnissen sollte stets zur Vorsicht gemahnen, meistens liegt in diesen Fällen ein Fehler im Programm vor.

[1] Institute of Electrical and Electronic Engineers

```
program matmult
do i=1,n                                ! Schleife über die Zeilen von C
  do j=1,n                              ! Schleife über die Spalten
    C(i,j) = 0.0                        ! Initialisierung der Summe
    do k=1,n                            ! Schleife zur Berechnung der Summe
      C(i,j) = C(i,j) + A(i,k)*B(k,j)
    end do                              ! Ende k-Schleife
  end do                                ! Ende j-Schleife
end do                                  ! Ende i-Schleife
end program matmult
```

Algorithmus 1.1: Multiplikation zweier $n \times n$-Matrizen

1.3 Begriff des Fehlers

- *Rundungsfehler* bei Operationen mit endlicher Gleitpunktarithmetik, siehe oben
- *Verfahrensfehler* (oder *Abbruchfehler*, *Diskretisierungsfehler*) bei Darstellung unendlicher oder infinitesimaler Prozesse im Rechner.
  Beispiel:
  - Approximation von $y''(x)$ durch $[y(x-h)-2y(x)+y(x-h)]/h^2$ wie oben, $h>0$
  - Approximation von $\pi^2/6$ durch $\sum\limits_{i=1}^{n} 1/i^2$, $n<\infty$
- *Absoluter Fehler:* Differenz zwischen der exakten Lösung x eines Problems und einer numerischen Approximation $\bar{x}$: $x-\bar{x}$
- *Relativer Fehler:* $\frac{x-\bar{x}}{x}$
  Beispiel: $x = 0.01$, $\bar{x} = 0.005 \to x - \bar{x} = 0.005$, $(x-\bar{x})/x = 0.5 \;\Rightarrow$ Der relative Fehler ist i.A. aussagekräftiger.

1.4 Algorithmische Schreibweise

Zweck: Formulierung eines Algorithmus zwecks Ausführung auf einem Computer.
Beispiel: Algorithmus 1.1 ist ein Fortran-Programm zur Berechnung des Matrixproduktes $C = AB$ mit der Formel $c_{ij} = \sum\limits_{k=1}^{n} a_{ik}b_{kj}$ $(A, B, C \in \mathbb{R}^{n,n})$.

Bemerkung:

1. „!“ leitet in Fortran einen Kommentar ein, d.h. Text, der danach in derselben Zeile steht, wird vom System ignoriert

2. „`C(i,j) = C(i,j) + ...`“ ist im Sinne einer Zuweisung zu verstehen, **nicht** im Sinne einer mathematischen Gleichung! Es bedeutet: „Addiere `A(i,k)*B(k,j)` zu `C(i,j)`“.

3. In anderen Programmiersprachen ist die Syntax verschieden, z. B. „`for i=1 to n`“ in Basic anstatt „`do i=1,n`“.

1.5 „Groß-Oh“–Notation

Zweck: Angabe von Größenordnungen, z.B. bei der Abschätzung des Rechenaufwandes oder bei der Fehlerabschätzung. Vernachlässigung „unwichtiger“ Terme, Zusammenfassung von Termen gleicher Größenordnung.

Definition 1.1 *Es seien $f, g : \mathbb{N} \to \mathbb{N}$ Abbildungen.*
$f(n) = O(g(n))$, falls es Konstanten $K \in \mathbb{R}$, $N \in \mathbb{N}$ gibt, so daß $f(n) \leq K \cdot g(n)$ für alle $n \geq N$. Man sagt „$f(n)$ gleich Oh von $g(n)$ für große n“.

Beispiel:

$$
\begin{aligned}
f(n) &= 3n^3 + 5n^2 + 7n, \text{ dann} \\
f(n) &= O(n^3), \text{ denn } f(n) \leq 4n^3 \text{ für große } n \\
&= 3n^3 + O(n^2), \text{ denn } 5n^2 + 7n \leq 6n^2 \text{ für große } n \\
&= 3n^3 + 5n^2 + O(n)
\end{aligned}
$$

Im diesem Sinne hat man auch eine O-„Arithmetik:“

$$
\begin{gathered}
f(n) + n^2 = 3n^3 + O(n^2) + n^2 = 3n^2 + O(n^2) \\
O(n) + O(n) = O(n) \\
O(n^3) + O(n^2) = O(n^3), \text{ usw.}
\end{gathered}
$$

Definition 1.2 *Es seien $f, g : \mathbb{R} \to \mathbb{R}$ Abbildungen.*
$f(x) = O(g(x))$, falls es Konstanten $K, x_0 \in \mathbb{R}_+$ gibt, so daß $f(x) \leq K \cdot g(x)$ für alle $x \in (-x_0, x_0)$. Man sagt „$f(x)$ gleich Oh von $g(x)$ für kleine x“.

Beispiel:

$$
\begin{aligned}
\sin(x) &= O(x) \text{ für kleine } x \\
f(x+h) &= \sum_{k=0}^{\infty} \frac{f^{(k)}(x)}{k!} h^k, \text{ dann } f(x+h) = \sum_{k=0}^{n} \frac{f^{(k)}(x)}{k!} h^k + O(h^{n+1}) \text{ für kleine } h \\
y''(x) &= [y(x-h) - 2y(x) + y(x+h)]/h^2 + O(h^2) \text{ für kleine } h
\end{aligned}
$$

1.6 Kondition, Stabilität und Effizienz

Kondition eines mathematischen Problems

Ein Problem nennt man *gut/schlecht konditioniert*, wenn kleine Änderungen in den „Eingangsdaten" kleine/große Änderungen in der Lösung hervorrufen. Schlecht konditionierte Probleme sind prinzipiell schwierig zu lösen.

Beispiel: (Wilkinson) Berechnung der größten Nullstelle des Polynoms

$$p(z) := (z-1)(z-2)...(z-20) = \sum_{k=0}^{20} a_k z^k$$

Eingangsdaten: Koeffizienten $a_0, ..., a_{20}$ (die kann man ausrechnen), speziell gilt $a_{19} = -210$. Die Lösung ist $x = 20$.
Betrachte nun eine kleine Störung der Eingangsdaten: $\bar{p}(z) := p(z) - 2^{-23} z^{19}$, also $\bar{a}_{19} = -210 - 2^{-23}$. Man kann ausrechnen, daß die größte Nullstelle von $\bar{p}$ der Wert $\bar{x} = 20.8469...$ ist.

Also: eine relative Änderung in den Eingangsdaten von

$$\left|\frac{a_{19} - \bar{a}_{19}}{a_{19}}\right| = \frac{2^{-23}}{210} < 10^{-9}$$

bewirkt eine relative Änderung in der Lösung von

$$\left|\frac{x - \bar{x}}{x}\right| = \frac{0.8469}{20} > 4 \times 10^{-2} \; !$$

$\rightarrow$ Die Änderung in den Eingangsdaten wird verstärkt mit einem Faktor $> 10^7$! Eine Veränderung der Koeffizienten in einer Größenordnung von Rundungsfehlern bei der einfachen Genauigkeit ($\simeq 10^{-8}$) bewirkt schon eine deutliche Veränderung in der Lösung. Das bedeutet, daß man mit einfach genauer Rechnung keine Chance hat, $\bar{x}$ genau zu ermitteln. Das Problem ist sehr schlecht konditioniert.

Stabilität eines numerischen Algorithmus

Man nennt einen numerischen Algorithmus *numerisch stabil* oder *gutartig*, wenn Rundungsfehler sich bei Ausführung des Algorithmus „nicht zu stark" fortpflanzen.

Beispiel: Berechnung von $x = a^2 - b^2 = (a-b)(a+b)$ für $a = 0.31416E+01$, $b = -0.31523E+01$, $t = 5$, auf zwei verschiedene Arten. Exakte Lösung ist -0.06734473.

```
Algorithmus 1:
r1 = a*a    !  0.98697E+01
r2 = b*b    !  0.99370E+01
x = r1-r2   ! -0.67300E-01
```

```
Algorithmus 2:
r1 = a-b    !  0.62939E+01
r2 = a+b    ! -0.10700E-01
x = r1*r2   ! -0.67345E-01
```

Wie wir sehen, ist das Ergebnis von Alg.2 auf alle zur Verfügung stehenden 5 Stellen genau (Rundung von -0.06734473 auf 5 Stellen ergibt -0.67345E-01), während das Ergebnis von Alg.1 nur auf 3 Stellen genau ist. Die Erklärung ist, daß bei der Subtraktion $x = r_1 - r_2$ zweier etwa gleichgroßer Werte r_1, r_2 in Alg.1 eine *Auslöschung* von Dezimalziffern eintritt, welche die zuvor bei der Berechnung von r_1 und r_2 begangenen Rundungsfehler verstärkt! Bei Alg.2 hat man zwar auch Auslöschung ($r_2 = a+b$), aber hier wurden vorher keine Rundungsfehler begangen.

Die folgende grobe **Fehleranalyse** bestätigt dies: Zur Vereinfachung nehmen wir an, daß Additionen/Subtraktionen im Rahmen der zur Verfügung stehenden Dezimalstellen exakt erfolgen (bei diesem Beispiel stimmt das), bei jeder Multiplikation aber ein Rundungsfehler der Form

$$s_1 * s_2 = s_1 s_2 (1+\varepsilon), \quad |\varepsilon| \text{ „klein“}$$

ensteht. Wir erhalten dann für die relativen Fehler

1. bei Alg.1: $\bar{x} = a^2(1+\varepsilon_1) - b^2(1+\varepsilon_2)$, daraus errechnet sich

$$\frac{x-\bar{x}}{x} = \frac{b^2\varepsilon_2 - a^2\varepsilon_1}{a^2-b^2}$$

2. bei Alg.2: $\bar{x} = (a-b)(a+b)(1+\varepsilon_3)$, daraus errechnet sich

$$\frac{x-\bar{x}}{x} = \varepsilon_3$$

Falls $a \simeq b$ und $\varepsilon_1 \neq \varepsilon_2$, ist der relative Fehler von Algorithmus 1 sehr groß. Der relative Fehler von Alg. 2 dagegen ist — unabhängig von a, b — klein. Alg.1 ist instabil, Alg.2 dagegen stabil.

Effizienz

Ein Algorithmus soll „möglichst schnell“ sein, d.h. mit möglichst wenigen Rechenoperationen auskommen.

Beispiel: Berechnung von $\det(A)$, $A \in \mathbb{R}^{n,n}$.
Sei $\mathcal{A}(n)$ der Aufwand zur Berechnung (Anzahl der Rechenoperationen). Offensichtlich gilt $\mathcal{A}(1) = O(1)$.

Idee: Berechne $\det(A)$ mit dem Entwicklungssatz von Laplace:

$$\det(A) = \sum_{i=1}^{n} a_{i1} \det(A_{i1}),$$

wobei $A_{i1} \in \mathbb{R}^{n-1,n-1}$ die Hauptuntermatrizen der ersten Spalte bezeichnen. Um $\det(A)$ nach dieser Formel zu berechnen, müssen also auf jeden Fall n Determinanten von $(n-1) \times (n-1)$-Matrizen berechnet werden, es gilt also

$$\mathcal{A}(n) \geq n\mathcal{A}(n-1).$$

Zusammen mit $\mathcal{A}(1) = 1$ erhält man, daß $\mathcal{A}(n) \geq n!$ Die Idee war nicht so gut, der Algorithmus ist extrem ineffizient. Eine effizientere Methode wäre, die Matrix wie bei der Gaußelimination auf obere Dreiecksgestalt zu bringen und die Determinante als das Produkt der Diagonaleinträge zu berechnen, siehe folgendes Kapitel.

Fazit: Die „erstbeste“ oder naheliegenste Wahl eines Algorithmus ist nicht immer die beste!