# Einleitung 1

In der numerischen Mathematik werden Verfahren zum „Lösen“ von mathematischen Problemen und Aufgaben entworfen und analysiert. Dabei ist die numerische Mathematik eng mit anderen Zweigen der Mathematik und der Informatik verbunden und kann oft nicht von dem Anwendungsgebiet wie der Chemie, Physik oder Medizin, den Ingenieurswissenschaften oder Ökonomie getrennt werden. Die Aufgaben, die wir zu lösen versuchen, stellen wir uns meist nicht selbst, sondern sie kommen aus den unterschiedlichsten Anwendungsbereichen. Auf diese Veranschaulichungen legen wir in diesem Buch besonders Wert und kennzeichnen die Abschnitte als *Exkurse*.

Der übliche Weg vom Problem zur Lösung ist lang und kann in folgende Schritte unterteilt werden:

1. *Mathematische Modellierung*. Das zugrunde liegende Problem wird mathematisch erfasst, es werden Gleichungen (oft basierend auf physikalischen Gesetzmäßigkeiten) entwickelt, die das Problem beschreiben. Ergebnis des mathematischen Modells kann ein lineares Gleichungssystem sein, aber auch eine Differentialgleichung.
2. *Analyse des Modells*. Das mathematische Modell muss auf seine Eigenschaften untersucht werden: Existiert eine Lösung, ist diese Lösung eindeutig? Hängt die Lösung stetig von den Daten ab? Hier kommen alle Teilgebiete der Mathematik zum Einsatz, von der Analysis, linearen Algebra über Statistik, Gruppentheorie, Funktionalanalysis zur Theorie von partiellen Differentialgleichungen.
3. *Numerische Verfahrensentwicklung*. Ein numerisches Lösungs- oder Approximationsverfahren wird für das Modell entwickelt. Viele mathematische Modelle (etwa Differentialgleichungen) können nicht exakt gelöst werden und oft kann eine Lösung, auch wenn sie existiert, nicht angegeben werden. Die Lösung einer Differentialgleichung ist eine Funktion $f\colon [a,b] \to \mathbb{R}$ und zur genauen Beschreibung müsste der Funktionswert an den unendlich vielen Punkten des Intervalls $[a,b]$ bestimmt werden. Jede durchführbare Verfahrensvorschrift kann allerdings nur aus endlich vielen Schritten bestehen. Das Problem muss zunächst *diskretisiert* werden, also auf eine endlich-dimensionale



T. Richter, T. Wick, *Einführung in die Numerische Mathematik*,
https://doi.org/10.1007/978-3-662-54178-4_1

Aufgabe reduziert werden. Die numerische Mathematik befasst sich mit der Analyse der numerischen Verfahren, also mit Untersuchung von Konvergenz und Approximationsfehlern, wenn sich die diskretisierte Lösung der unendlich-dimensionalen Lösung immer weiter annähert (durch Hinzunahme weiterer Punkte in $[a, b]$).

4. *Implementierung*. Das numerische Verfahren muss auf einem Computer implementiert werden. Eine *effiziente Implementierung* erfordert gute Kenntnisse in einer Programmiersprache (zum Beispiel C++, Fortran, MATLAB, Octave, Python). Ein *effizientes Programm* erfordert die Entwicklung spezieller Algorithmen. Um moderne Computerarchitekturen nutzen zu können, muss das Verfahren z. B. für die Verwendung von Parallelrechnern modifiziert werden. Gleichzeitig ist man an der Entwicklung *robuster Algorithmen* interessiert. Das heißt, für Änderungen in der Geometrie, der Intervalllänge oder verschiedener Parameter (z. B. Steifigkeit einer Feder) sollte der Algorithmus immer ähnlich zuverlässig funktionieren.
5. *Auswertung und Interpretation*. Die numerischen Ergebnisse müssen ausgewertet werden. Dies beinhaltet eine grafische Darstellung der Ergebnisse sowie in technischen Anwendungen z. B. die Auswertung von Kräften, die nur indirekt gegeben sind. Anhand von Plausibilitätsanalysen muss die Qualität der Ergebnisse beurteilt werden. Solche Analysen erfolgen entweder in Vergleichen zu bekannten (analytischen) Lösungen oder zu experimentellen Daten.

Die oben genannten Teilaspekte dürfen nicht getrennt voneinander gesehen werden. Ein effizienter Algorithmus ist nichts wert, wenn das zugrunde liegende Problem überhaupt keine wohldefinierte Lösung hat, und ein numerisches Verfahren für ein Anwendungsproblem ist wertlos, wenn der Computer in absehbarer Zeit zu keiner Lösung kommt.

Des Weiteren wird die obige Liste in der Regel nicht sequenziell abgearbeitet, die Auswertung der Ergebnisse kann Anlass zur Überprüfung der zugrunde liegenden mathematischen Modelle geben und zu modifizierten numerischen Verfahren führen. Wir haben es somit mit einem ständigen Zusammenspiel von Modellierung, Analyse, numerischer Approximation und Anwendung zu tun.

Letztendlich legen wir mit den oben diskutierten Schritten die Basis des *Wissenschaftlichen Rechnens*, welches sich als dritte Säule neben Experiment und Theorie etabliert hat. Die numerische Mathematik hat Züge einer experimentellen Wissenschaft, in der wir mit Algorithmen und Computerprogrammen, unter Ausnutzung klar definierter mathematischer Strukturen experimentieren.

## 1.1 Konzepte der numerischen Mathematik

Da sich die Ziele und das Vorgehen in der numerischen Mathematik zum Teil stark von anderen Disziplinen unterscheiden, führen wir in diesem Abschnitt die wesentlichen Konzepte ein, welche charakteristisch sind und uns in allen weiteren Kapiteln dieses Buches immer wieder begegnen.

Typischerweise geht es in der numerischen Mathematik um die Berechnung von mathematischen Problemen und die Angabe von konkreten Lösungen. Von vielen Problemen wissen wir, dass eine Lösung existiert, kennen aber kein Verfahren, um diese Lösung auszurechnen. Ein Beispiel ist die Suche nach Nullstellen eines allgemeinen Polynoms

$$p(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \ldots + a_n x^n.$$

Der Fundamentalsatz der Algebra [40] besagt, das jedes (nichtkonstante) Polynom mindestens eine (gegebenenfalls komplexe) Nullstelle besitzt. Für Polynome vom Grad $n$ (wobei $n$ beliebig) existiert jedoch keine analytische Lösungsformel. Wir wissen, dass es eine Lösung gibt, kennen jedoch keine Methode, um diese zu berechnen. Hier kommt die numerische Mathematik ins Spiel. Auch wir werden nicht in der Lage sein, dieses Problem *exakt* zu lösen, können aber Methoden entwickeln, um eine Lösung zu *approximieren*.

*Approximationen* sind natürlich auch zentral in anderen Bereichen der Mathematik. Der *Approximationssatz von Weierstraß* besagt, dass jede stetige Funktion auf einem Intervall $I = [a, b]$ beliebig gut durch ein Polynom approximiert werden kann. Dieser Begriff der Approximation ist eng mit dem Begriff der *Konvergenz* verbunden: Es gibt eine Folge von Polynomen $p_n \in P_n$, welche gegen eine gegebene Funktion $f \in C[a, b]$ konvergiert:

$$\max_{x \in [a,b]} \|f(x) - p_n(x)\| \to 0 \quad (n \to \infty)$$

In der numerischen Mathematik werden wir den Lösungsbegriff weitestgehend durch den Begriff einer hinreichend genauen Approximation ersetzen. Wir betrachten ein Problem – zum Beispiel die Approximation einer stetigen Funktion durch ein Polynom – als gelöst, wenn es uns gelingt, eine Polynom $p_n \in P_n$ zu bestimmen, sodass der Fehler zwischen $f(x)$ und $p_n(x)$ auf $I = [a, b]$ hinreichend klein ist. Was bedeutet hinreichend? Hierauf können wir keine Antwort geben. Der Schritt vom Weierstraßschen Approximationssatz zur konkreten Angabe eines approximierenden Polynoms ist jedoch noch sehr weit. Muss die Lösung konkret angegeben werden, so genügt es bei Weitem nicht zu wissen, dass eine solche Lösung existiert. Vielmehr müssen wir Wege finden, diese Lösung zu konstruieren.

Können mathematische Probleme nicht analytisch gelöst werden oder kann die analytische Lösung nicht in endlicher Zeit berechnet werden, so muss die Lösung **approximiert** werden. In vielen Fällen werden wir eine **numerische Approximation** als Lösung akzeptieren, wenn sie hinreichend genau ist.

Ein einfaches Beispiel ist die Berechnung der Zahl $\pi$. Es existieren zahlreiche Formeln zu deren Approximation. Auf Leibniz geht die sogenannte *Leibniz-Reihe* zurück:

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} \pm \cdots \tag{1.1}$$

Sie beruht auf einer Reihenentwicklung von $y = \arctan(x)$ sowie der Beziehung $\arctan(1) = \pi/4$. Wollen wir diese Reihe benutzen, um $\pi$ zu approximieren, so müssen wir die Berechnung an irgendeinem Punkt abbrechen. Wir definieren die endliche Summe $\Pi_n$ als Approximation

$$\pi \approx \Pi_n := 4\left(\sum_{k=1}^{n} \frac{(-1)^{k+1}}{2k-1}\right).$$

Die Analysis liefert uns Methoden zur Untersuchung der *Konvergenz* dieser Folge $\Pi_n$. Es gilt

$$\Pi_n \to \pi \quad (n \to \infty),$$

da es sich um eine alternierende Reihe handelt, deren Glieder eine Nullfolge bilden. Konvergenz ist einer der zentralen Begriffe der Analysis, der auch in der numerischen Mathematik eine wichtige Rolle einnimmt.

**Konvergenz** ist ein qualitativer Begriff. Die Konvergenz einer Folge $(a_n)_{n\in\mathbb{N}}$ gegen einen Grenzwert $a_n \to a$ für $n \to \infty$ besagt, dass wir für hinreichend große Indizes $n \in \mathbb{N}$ dem Grenzwert $a$ beliebig nahekommen. Der Grenzwert $a$ ist im Rahmen der numerischen Mathematik meist die gesuchte Lösung des Problems. Die Folge $(a_n)_{n\in\mathbb{N}}$ ist das numerische Approximationsverfahren.

Die Leibniz-Reihe konvergiert gegen die gesuchte Lösung, die Zahl $\pi$. So wichtig diese Aussage ist, so wenig hilft sie uns in der praktischen numerischen Mathematik. Wie können wir die Zahl $\pi$ mit einem Fehler von 1 % bestimmen? An welcher Stelle dürfen wir die Berechnung der Reihe abbrechen? Es gilt

$$\Pi_1 = 4,\ \Pi_2 \approx 2{,}67,\ \Pi_3 \approx 3{,}47,\ \Pi_4 \approx 2{,}90, \ldots, \Pi_{31} \approx 3{,}17,\ \Pi_{32} \approx 3{,}11, \ldots$$

und schließlich gilt die Abschätzung

$$\frac{|\Pi_{32} - \pi|}{\pi} \approx 0{,}99\%.$$

Dieses Ergebnis war so nicht vorherzusehen und die *Fehlerabschätzung* gelingt uns hier nur, weil wir die gesuchte Zahl $\pi$ bereits kennen.

Die numerische Mathematik kann als die Mathematik der **Fehler** betrachtet werden. Numerische Mathematik ist nicht falsch, inexakt oder unpräzise. In der Analyse numerischer Verfahren müssen wir jedoch **Fehler** berücksichtigen. Lösen wir Probleme aus der Anwendung, so treten Messfehler auf. Lösen wir Probleme mit dem Computer, so treten Rechenfehler z. B. durch eine endliche Darstellung von Zahlen auf. Approximieren wir Lösungen, so tritt ein Fehler bei endlichem Abbruch eines eigentlich unendlichen Prozesses auf.

Im Allgemeinen versuchen wir in der numerischen Mathematik aber natürlich Probleme zu lösen, deren Lösung wir vorher noch nicht kennen. Der Begriff der *Fehlerabschätzung* erklärt ein weiteres wichtiges Konzept der numerischen Mathematik. Wenn wir eine numerische Approximation $\Pi_n$ zu einem Problem mit (unbekannter) Lösung $\pi$ erstellt haben, müssen wir sicherstellen, dass der Fehler zwischen Näherung und exakter Lösung klein ist. So eine *Fehlerabschätzung* sollte berechenbar sein. Wir suchen also eine berechenbare Schranke für den Fehler, d. h. eine Formel, die garantiert, dass

$$|\Pi_n - \pi| \leq E_n.$$

Unter einer **Fehlerabschätzung** verstehen wir eine (berechenbare) Formel, die eine Abschätzung für den Fehler angibt, welcher zum Beispiel durch den Abbruch eines unendlichen Prozesses nach einer endlichen Anzahl von Schritten entsteht. Eine **Fehlerabschätzung** kann aber auch eine berechenbare Schranke für den Fehler sein, der durch Rundung auf dem Computer entsteht oder durch fehlerhafte Daten gegeben ist. Ein Ziel von **Fehlerabschätzungen** ist es stets, die Qualität von numerischen Approximationen zu beurteilen. **Fehlerabschätzungen** und **Fehlerkontrolle** sind wesentliche Merkmale der numerischen Mathematik.

Am Beispiel der Leibniz-Reihe können wir zur Herleitung einer entsprechenden Fehlerabschätzung wieder eine Idee in der Analysis finden: Die Leibniz-Reihe ist eine sogenannte *alternierende Reihe*, deren Summanden eine *monoton fallende Nullfolge* bilden. Es gilt:

$$\Pi_n = \sum_{k=1}^{n} (-1)^{n+1} a_k, \quad a_k := \frac{1}{2k-1}$$

Für solche Reihen gilt stets, dass der Grenzwert zwischen zwei aufeinanderfolgenden Partialsummen liegt. Angewendet auf dieses Beispiel bedeutet dies:

$$\Pi_{2n} \leq \pi \leq \Pi_{2n+1}$$

Aus dieser Abschätzung können wir eine Fehlerabschätzung herleiten. Es gilt:

$$0 \leq \pi - \Pi_{2n} \leq \Pi_{2n+1} - \Pi_{2n} = \frac{1}{2n+1} \tag{1.2}$$

Um einen Fehler von 1 % sicherzustellen, muss für $n \in \mathbb{N}$ gelten:

$$\frac{1}{2n+1} < \frac{1}{100} \quad \Leftrightarrow \quad n > 49{,}5$$

Für $\Pi_{2n} = \Pi_{100}$ können wir also garantieren, dass wir einen Fehler von 1 % nicht übersteigen. Es gilt:

$$\Pi_{100} \approx 3{,}132, \quad \frac{|\Pi_{100} - \pi|}{\pi} \approx 0{,}32\%$$

Die Fehlerabschätzung ist nicht scharf, d. h., wir können den Fehler von 1 % schon bei $n = 32$ erreichen, aber sie ist *robust*, d. h., sie garantiert uns eine entsprechende Approximationsgüte. Auch dann, wenn wir das gesuchte Ergebnis noch gar nicht kennen.

Eine abgewandelte Reihe zur Approximation von $\pi$ ist die *Madhava-Leibniz-Reihe:*

$$\pi \approx \Pi_n := \sqrt{12} \sum_{k=1}^{n} \frac{(-3)^{-k+1}}{2k-1} \tag{1.3}$$

Für die entsprechenden Partialsummen $\Pi_n$ gilt:

$$\Pi_1 \approx 3{,}46, \quad \Pi_2 \approx 3{,}08, \quad \Pi_3 \approx 3{,}16, \quad \Pi_4 \approx 3{,}14, \ldots$$

Nach nur vier Schritten ist bereits ein Fehler von weniger als 1 % erreicht. Diese Approximation scheint viel leistungsfähiger und schneller zu sein als die vorherige und zur numerischen Approximation besser geeignet.

Zur Beurteilung und zum Vergleich verschiedener Methoden wird die **Konvergenzgeschwindigkeit** eingeführt: Konvergiert eine Folge $(a_n)_{n\in\mathbb{N}}$ schneller oder langsamer als eine zweite Folge $(b_n)_{n\in\mathbb{N}}$ gegen denselben Grenzwert? Oft verwenden wir den Begriff der **Konvergenzordnung** zur Einordnung und Klassifikation verschiedener Methoden: Halbiert sich der Fehler in jedem Schritt der Approximation? Konvergiert eine Methode linear oder quadratisch?

Numerische Mathematik sollte (besser muss!) *effizient* sein. Dies ist eine Forderung, die sich in vielen Bereichen der Mathematik nicht stellt. Aber wir stellen uns nun die Aufgabe, die Zahl $\pi$ auf 10 Stellen Genauigkeit zu approximieren. Mithilfe der Fehlerabschätzung (1.2) können wir für die erste Leibniz-Reihe einen Zusammenhang zwischen dem gewünschten relativen Fehler $\epsilon$ und der Anzahl an Schritten $2n_\epsilon$ herleiten:

$$\frac{|\pi - \Pi_{2n_\epsilon}|}{\pi} < \frac{1}{(2n_\epsilon - 1)\pi} \overset{!}{<} \epsilon \quad \Leftrightarrow \quad n_\epsilon > \frac{1+\pi\epsilon}{2\epsilon\pi}$$

Grob gesprochen: Um eine Stelle zu gewinnen, müssen wir 10-mal mehr Schritte durchführen. Um die ersten 10 Stellen von $\pi$ zu bestimmen, müssen wir mit der Leibiz-Reihe etwa $10^{10} = 10\,000\,000\,000$ Schritte durchführen. Auch moderne Computer sind damit eine Weile beschäftigt. Dieser Zugang mag also *robust* sein, er ist aber sicher nicht effizient. Auch die modifizierte Leibniz-Reihe ist die Reihe einer alternierenden Nullfolge und wir leiten eine entsprechende Fehlerabschätzung her:

$$0 \leq \pi - \Pi_{2n} \leq \Pi_{2n+1} - \Pi_{2n} = \sqrt{12}\frac{(-3)^{-2n}}{4n+1}$$

Jetzt gilt in starker Vereinfachung

$$\frac{|\pi - \Pi_{2n_\epsilon}|}{\pi} < \sqrt{12}\frac{(-3)^{-2n}}{(4n+1)\pi} \overset{!}{<} \epsilon \quad \Leftrightarrow \quad n_\epsilon > \frac{\log\left(\frac{4\pi\epsilon}{\sqrt{12}}\right)}{2\log\left(\frac{1}{3}\right)}.$$

Für $\epsilon = 10^{-10}$ erhalten wir

$$n_\epsilon > 10.$$

Die modifizierte Folge scheint daher bei Weitem schneller zu konvergieren!

Wir nennen ein numerisches Verfahren **effizient**, wenn es ein mathematisches Problem „ressourcenschonend“ löst oder approximiert. Zu den wichtigen „Ressourcen“ zählt dabei neben der *Rechenzeit* auch der *Speicheraufwand*. Der Begriff der **Effizienz** ist meist ein relativer Begriff. Ein Verfahren ist **effizienter** oder **weniger effizient** als ein alternatives Verfahren. Bei den meisten mathematischen Problemen ist nicht bekannt, welches Verfahren das **effizienteste** ist.

Von Srinivasa Ramanujan existiert zur Approximation von $\pi$ die Reihe

$$\Pi_n := \frac{9\,801}{2\sqrt{2}}\left(\sum_{k=0}^{n}\frac{(4k)!(1\,103+26\,390k)}{(k!)^4 396^{4k}}\right)^{-1} \to \pi \quad (n \to \infty).$$

Für die ersten drei Approximationen gilt:

$$\Pi_0 \approx \underline{\mathbf{3{,}141592}}730013305660313996$$
$$\Pi_1 \approx \underline{\mathbf{3{,}14159265358979}}3877998905$$
$$\Pi_2 \approx \underline{\mathbf{3{,}14159265358979323846264}}9$$

Die richtigen Stellen sind unterstrichen. In jeder Iteration kommen etwa acht richtige Stellen hinzu. Statt 10 000 000 000 Schritten sind hier nur zwei Schritte notwendig. Jeder Schritt in diesem Verfahren ist natürlich viel aufwendiger. Statt einer Division und einer Addition bei der Leibniz-Reihe sind zwei Fakultäten, eine Potenz und einige Multiplikationen durchzuführen. Aber selbst wenn wir diesen größeren Aufwand berücksichtigen, so ist dieses Verfahren bei Weitem *effizienter*.

Wir wenden uns einem anderen wichtigen Problem der numerischen Mathematik zu, nämlich der Berechnung von Eigenwerten einer Matrix $A \in \mathbb{R}^{n\times n}$. Diese sind als Nullstellen des charakteristischen Polynoms gegeben:

$$\det(\lambda I - A) = 0$$

Beispielsweise habe die Matrix $A \in \mathbb{R}^{5\times 5}$ die Eigenwerte 1,2,3,4,5. Das charakteristische Polynom hat also die Form

$$p(\lambda) = \det(\lambda I - A) = \prod_{i=1}^{5}(\lambda - i) = \lambda^5 - 15\lambda^4 + 85\lambda^3 - 225\lambda^2 + 274\lambda - 120.$$

**Tab. 1.1** Die Nullstellen eines gestörten Polynoms zur Berechnung der Eigenwerte einer Matrix. Schon bei einem relativen Fehler von 0,01 % beträgt der Fehler in den Eigenwerten fast 5 %. Bei einer Störung des Polynomkoeffizienten von nur 0,08 % hat das Polynom bereits komplexe Nullstellen

| $\epsilon$ | $\lambda_1$ | $\lambda_2$ | $\lambda_3$ | $\lambda_4$ | $\lambda_5$ | rel. Fehler |
|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 0 |
| $1 \cdot 10^{-4}$ | 1,00 | 1,97 | 3,13 | 3,82 | 5,08 | 4,5 % |
| $2 \cdot 10^{-4}$ | 1,00 | 1,95 | 3,38 | 3,52 | 5,14 | 12 % |
| $3 \cdot 10^{-4}$ | 1,00 | 1,93 | 3,43 + 0,31i | 3,43 – 0,31i | 5,21 | 18 % |

Wir müssen davon ausgehen, dass es einem Computer nicht gelingen wird, das Polynom exakt zu bestimmen. Die Koeffizienten werden mit einem (kleinen) Fehler versehen sein. Wir betrachten eine kleine Störung $\epsilon$ und das Polynom

$$p_\epsilon(\lambda) = \lambda^5 - 15(1+\epsilon)\lambda^4 + 85(1-\epsilon)\lambda^3 - 225(1+\epsilon)\lambda^2 + 274(1-\epsilon)\lambda - 120(1+\epsilon).$$

Zu verschiedenen Störungen bestimmen wir in Tab. 1.1 die Nullstellen dieses Polynoms (wir geben nur die ersten drei Stellen an) und bestimmen jeweils auch den maximalen relativen Fehler.

Bei $\epsilon = 10^{-4}$, also einem relativen Fehler der Koeffizienten von nur 0,01 %, beträgt der Fehler in den Eigenwerten bereits fast 5 %. Der Fehler wird um den Faktor 500 verstärkt! Bei einem Fehler von nur 0,03 % treten bereits komplexe Eigenwerte auf. Eine wichtige Struktureigenschaft von z. B. symmetrischen Matrizen bleibt nicht mehr erhalten. Die Berechnung von Eigenwerten mit dem Umweg über das charakteristische Polynom scheint kein *numerisch stabiler* Algorithmus zu sein.

Die *Stabilität* ist vermutlich der wichtigste und auch schwierigste Begriff der numerischen Mathematik. Es gibt hierfür keine einheitliche Definition. Das Konzept der *Stabilität* kommt immer dann zur Anwendung, wenn in der numerischen Mathematik die allgemeinen Denkmuster nicht mehr gelten:

- Wir weisen in der Analysis Konvergenz von Folgen gegen ihren Grenzwert $a_n \to a$ nach. Aber wie gut ist die Approximation für *kleine* $n$? Wir werden numerische Verfahren *stabil* nennen, wenn sie auch für kleine $n$, also weit weg von der Konvergenz *sinnvolle Ergebnisse* liefern. Was *sinnvoll* ist, muss von Fall zu Fall unterschieden werden.
- Einfache mathematische Gesetze wie das Distributivgesetz gelten auf dem Computer nicht mehr. Unvermeidbare Rundungsfehler führen – je nach Rechenweg – zu unterschiedlichen Ergebnissen. Wir werden numerische Verfahren stabil nennen, wenn sie trotz fehlerhafter Daten und fehlerhafter Computerrealisierung robust sind.

Der Begriff der **Stabilität** bezeichnet das Verhalten von numerischen Verfahren bei ihrer praktischen Umsetzung. Ist die Methode **robust** gegenüber Fehlern? Wie verhalten sich konvergente Prozesse, wenn sie nach endlicher Anzahl von Iterationen abgebrochen werden?

Die hier als *Konzepte* eingeführten Begriffe haben alle gemein, dass diese sich, ganz anders als in den meisten Bereichen der Mathematik üblich, nicht klar definieren lassen. Wir sprechen von großen und kleinen Fehlern, können aber keine klaren Grenzen angeben, wann ein Fehler groß oder klein ist. Eine gewisse Ausnahme bildet die Konvergenz, die aus der Analysis wohlbekannt ist und dort bereits eines der wesentlichen Konzepte darstellt. Deren Definition und Idee erfüllt den identischen Zweck in der numerischen Mathematik. Die Wagheit vieler Aussagen in der numerischen Mathematik macht letztere aber nicht einfacher als andere Disziplinen, nur weil gegebenenfalls ungenauere Ergebnisse akzeptabel sind. Im Gegenteil: Da wir wissen, dass Fehler ständig auftreten, gilt es, diese in Definitionen, Sätzen und Beweisen klar zu beschreiben. Darüber hinaus ist es in der Praxis (bei Computersimulationen) sogar essenziell, die auftretenden Fehler zu kontrollieren, um einen unerwünschten Abbruch des Algorithmus oder falsche Ergebnisse zu vermeiden oder die Effizienz zu verbessern. Zahlreiche Beispiele in diesem Buch belegen die Notwendigkeit zur sorgfältigen Untersuchung der *Fehler* in der numerischen Mathematik.

## 1.2 Definition eines Algorithmus

Das Ergebnis einer numerischen Aufgabe ist im Allgemeinen eine Zahl oder eine endliche Menge von Zahlen. Beispiele für numerische Aufgaben sind das Berechnen der Nullstellen einer Funktion, die Berechnung von Integralen, die Berechnung der Ableitung einer Funktion in einem Punkt, aber auch komplexere Aufgaben wie das Lösen einer Differentialgleichung.

Zum Lösen von numerischen Aufgaben werden wir unterschiedliche Verfahren kennenlernen. Wir grenzen zunächst ein:

► **Definition 1.1 (Numerisches Verfahren, Algorithmus)** Ein numerisches Verfahren ist eine Vorschrift zum Lösen oder zur Approximation einer mathematischen Aufgabe. Ein numerisches Verfahren heißt *direkt*, falls die Lösung bis auf Rundungsfehler exakt berechnet werden kann. Ein Verfahren heißt *approximativ*, falls die Lösung nur angenähert werden kann. Ein Verfahren heißt *iterativ*, falls die Näherung durch mehrfache Ausführung einer Vorschrift schrittweise verbessert wird.

Beispiele für direkte Lösungsverfahren sind die $p/q$-Formel zur Berechnung von Nullstellen quadratischer Polynome (siehe Kap. 6) oder der Gaußsche Eliminationsalgorithmus (Kap. 3) zum Lösen von linearen Gleichungssystemen. Approximative Verfahren müssen z. B. zum Bestimmen von komplizierten Integralen oder auch zum Berechnen von Nullstellen allgemeiner Funktionen eingesetzt werden. Oft ist ein exaktes Lösen von Aufgaben dem Problem nicht angemessen. Wenn Eingabedaten zum Beispiel mit großen Messfehlern behaftet sind, so ist ein exaktes Lösen nicht notwendig.

Wir betrachten ein Beispiel und dessen Analyse eines direkten Verfahrens im Folgenden im Detail:

**Beispiel 1.2 (Polynomauswertung)**

Es sei durch

$$p(x) = a_0 + a_1 x + \cdots + a_n x^n$$

ein Polynom gegeben. Dabei sei $n \in \mathbb{N}$ sehr groß. Wir werten $p(x)$ in einem Punkt $\xi \in \mathbb{R}$ mit dem trivialen Verfahren aus:

**Algorithmus 1.1 Polynomauswertung**
Gegeben sei ein Polynom $p(x) = a_0 + a_1 x + \cdots + a_n x^n$ mit Auswertungsstelle $\xi \in \mathbb{R}$.

```
1  p = 0
2  Für i von 0 bis n
3     y_i := a_i ξ^i
4     p = p + y_i
```

Wir berechnen den *Aufwand* zur Polynomauswertung: In Schritt 3 des Algorithmus sind zur Berechnung der $y_i$ $i$ Multiplikationen notwendig, insgesamt

$$0 + 1 + 2 + \cdots + n = \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2}.$$

In Schritt 4 sind weitere $n$ Additionen notwendig. Der Gesamtaufwand des Algorithmus beträgt demnach $n^2/2 + n/2$ Multiplikationen sowie $n$ Additionen. Wir fassen eine Addition und eine Multiplikation zu einer *elementaren Operation* zusammen und erhalten zusammen als Aufwand der trivialen Polynomauswertung

$$A_1(n) = \frac{n^2}{2} + \frac{n}{2}$$

elementare Operationen.

Wir schreiben das Polynom nun um

$$p(x) = a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + x a_n)\ldots)) \tag{1.4}$$

und leiten hieraus einen zweiten Algorithmus her:

**Algorithmus 1.2 Horner-Schema**

Gegeben seien das Polynom $p(x) = a_0 + a_1 x + \cdots + a_n x^n$ und die Auswertungsstelle $\xi \in \mathbb{R}$.

```
1 p = a_n
2 Für i von n − 1 bis 0
3   p = a_i + ξ · p
```

Jeder der $n$ Schritte des Verfahrens benötigt eine Multiplikation sowie eine Addition, also ergibt sich ein Aufwand von

$$A_2(n) = n$$

elementaren Operationen. Das *Horner-Schema* benötigt für die gleiche Aufgabe wesentlich weniger Operationen, man denke an Polynome $n \sim 1\,000$ (und $n \gg 1\,000$) und vergleiche $A_1(1\,000) \sim 500\,000$ und $A_2(1\,000) \sim 1\,000$.

**Beispiel 1.3 (Horner-Schema)**

Wir berechnen den Wert des Polynoms $p(x) = 3x^5 - 2x^4 + 1$ an der Stelle $\xi = 2$. Die Koeffizienten lauten also $a_5 = 3, a_4 = -2, a_3 = a_2 = a_1 = 0, a_0 = 1$. Dann erhalten wir das Schema:

| $i$ | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|
| $a_i$ | 3 | -2 | 0 | 0 | 0 | 1 | |
| $p_i$ | 3 | 4 | 8 | 16 | 32 | 65 | $= p(\xi)$ |

Somit ist $p(2) = 65$.

Wir präsentieren im Folgenden einen ersten Mini-Exkurs, welcher in die beiden Beispiele 1.4 und 1.13 aufgespalten ist.

**Beispiel 1.4 (Die Steuerfunktion)**

Wir veranschaulichen das Horner-Schema anhand eines praktischen Beispiels aus der Steuergesetzgebung. Die Einkommensteuern können durch Polynome beschrieben werden, um diese anschließend in Steuertabellen anzugeben. In den bis 2009 gültigen Versionen des Einkommensteuergesetzes (EstG §32a) wird explizit vermerkt (siehe Absatz (3) in EstG §32a), dass die Polynome mit dem Horner-Schema auszuwerten sind.[1] Seit dem Jahr 2010 sind allerdings die Absätze (2) – (4) im Gesetz weggefallen. Nichtsdestotrotz schauen wir uns die aktuellen Werte (Jahr 2017) an. Der amtliche Einkommensteuertarif ist bereits in der Form (1.4) angegeben [13] (2017):

[1] EstG §32a, Absatz 3 [13] (Jahr 2002): „Die zur Berechnung der tariflichen Einkommensteuer erforderlichen Rechenschritte sind in der Reihenfolge auszuführen, die sich nach dem Horner-Schema ergibt. Dabei sind die sich aus den Multiplikationen ergebenden Zwischenergebnisse für jeden weiteren Rechenschritt mit drei Dezimalstellen anzusetzen; die nachfolgenden Dezimalstellen sind fortzulassen. Der sich ergebende Steuerbetrag ist auf den nächsten vollen Euro-Betrag abzurunden."

► **Definition 1.5** Es sei $x$ das (Jahres-)Einkommen in Euro. Die zu zahlende Einkommensteuer $s(x)$ wird nach folgenden Kriterien berechnet:

(1) bis 8 820 EUR (Grundfreibetrag): 0,
(2) von 8 821 EUR bis 13 769 EUR: $(1\,007{,}27 \cdot y + 1\,400) \cdot y$,
(3) von 13 770 EUR bis 54 057 EUR: $(223{,}76 \cdot z + 2\,397) \cdot z + 939{,}57$,
(4) von 54 058 EUR bis 256 303 EUR: $0{,}42 \cdot x - 8\,475{,}44$,
(5) von 256 304 EUR an: $0{,}45 \cdot x - 16\,164{,}53$,

mit $y = \frac{x-8\,821}{10\,000}$ und $z = \frac{x-13\,770}{10\,000}$. Die aus (1) – (5) stückweise definierte Funktion wird Steuerfunktion $s(x)$ genannt. Wir erkennen sofort, beispielsweise in (3), die Form (1.4):

$$p(z) = a_0 + z(a_1 + za_2)$$

mit $a_0 = 939{,}57,\ a_1 = 2\,397{,}00,\ a_2 = 223{,}76.$

Wir nehmen als Einstiegsgehalt eines Ingenieurs $x = 48\,000$ EUR Brutto-Jahreseinkommen an. Das heißt, wir werten die Steuerfunktion an der Stelle $\xi = 48\,000$ aus. Damit fällt dieser in Gruppe (3). Zunächst ist $z = \frac{48\,000-13\,770}{10\,000} = 3{,}423$ (in der amtlichen Rechnung werden drei Dezimalstellen angesetzt). Das Horner-Schema liest sich nun:

| $i$ | 2 | 1 | 0 | |
|---|---|---|---|---|
| $a_i$ | 223,76 | 2 397,00 | 939,57 | |
| $s_i$ | 223,76 | 3 162,90 | 11 766,00 | $= s(\xi)$ |

Im Jahr 2017 muss der Ingenieur damit $s(48\,000) = 11\,766$ EUR an Einkommensteuer bezahlen.

Im Folgenden führen wir eine allgemeine Notation zur Beschreibung des Aufwands eines Verfahrens an.

► **Definition 1.6 (Numerischer Aufwand)** Der *Aufwand* eines numerischen Verfahrens ist die Anzahl der notwendigen elementaren Operationen. Eine *elementare Operation* ist eine Addition oder eine Multiplikation.

Meist hängt der Aufwand eines Verfahrens von der Problemgröße ab. Die Problemgröße $N \in \mathbb{N}$ wird von Problem zu Problem definiert, beim Lösen eines linearen Gleichungssystems $Ax = b$ mit einer Matrix $A \in \mathbb{R}^{N\times N}$ ist die Größe der Matrix die Problemgröße. Beim Auswerten eines Polynoms $p(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_0$ in einem Punkt $x$ ist die Problemgröße die Anzahl der Koeffizienten $n$.

Zur einfachen Schreibweise definieren wir:

▶ **Definition 1.7 (Landau-Symbole)**

*(i)* Es sei $g(n)$ eine Funktion mit $g \to \infty$ für $n \to \infty$. Dann ist $f \in O(g)$ genau dann, wenn

$$\limsup_{n\to\infty} \left|\frac{f(n)}{g(n)}\right| < \infty,$$

sowie $f \in o(g)$ genau dann, wenn

$$\lim_{n\to\infty} \left|\frac{f(n)}{g(n)}\right| = 0.$$

*(ii)* Sei $g(h)$ eine Funktion mit $g(h) \to 0$ für $h \to 0$. Wir definieren wie oben $f \in O(g)$ sowie $f \in o(g)$.

Einfach gesprochen: $f \in O(g)$, falls $f$ höchstens so schnell gegen $\infty$ konvergiert wie $g$, und $f \in o(g)$, falls $g$ schneller als $f$ gegen $\infty$ geht. Entsprechend gilt für $g \to 0$, dass $f \in O(g)$, falls $f$ mindestens so schnell gegen null konvergiert wie $g$, und $f \in o(g)$, falls $f$ schneller als $g$ gegen null konvergiert. Mithilfe der Landau-Symbole lässt sich der Aufwand eines Verfahrens einfacher charakterisieren.

**Beispiel 1.8 (Landau-Symbole)**

*(i)* Es sei $\varepsilon \to 0$ und $h \to 0$. Wir schreiben

$$h = o(\varepsilon),$$

wenn

$$\frac{h}{\varepsilon} \to 0 \quad \text{für } h \to 0, \quad \varepsilon \to 0,$$

was bedeutet, dass $h$ schneller gegen 0 konvergiert als $\varepsilon$.

*(ii)* Ein typische Aussage bei Fehlerabschätzungen sind Terme der Form

$$\|\bar{y} - y_n\| = O(k),$$

wobei $\bar{y}$ der exakte Lösungswert ist und $y_n, n = 0,1,2,\ldots$ eine Folge von numerischen Approximationen in Abhängigkeit eines (Diskretisierungs-)Parameters $k$. Hier bedeutet die $O$-Notation nichts anderes als

$$\frac{\|\bar{y} - y_n\|}{k} \to C \quad \text{für } k \to 0.$$

Der Bruch auf der linken Seite konvergiert also gegen eine Konstante $C$, die nicht notwendigerweise 0 ist. Dies verdeutlicht auch, dass $O$-Konvergenz schwächer als $o$-Konvergenz ist.

*(iii)* Im Fall der trivialen Polynomauswertung in Algorithmus 1.1 gilt für den Aufwand $A_1(n)$ in Abhängigkeit der Polynomgröße $n$

$$A_1(n) \in O(n^2)$$

und im Fall des Horner-Schemas von Algorithmus 1.2

$$A_2(n) \in O(n).$$

Wir sagen: Der Aufwand der trivialen Polynomauswertung wächst quadratisch, der Aufwand des Horner-Schemas linear. Weiter können mit den Landau-Symbolen Konvergenzbegriffe quantifiziert und verglichen werden. In den entsprechenden Kapiteln kommen wir auf diesen Punkt zurück.

**Bemerkung 1.9 (Landau-Symbole)** *Das Symbol $O(g)$ beschreibt eine Menge von Funktionen. Es ist*

$$f \in O(g),$$

*wenn $f/g$ für $n \to \infty$ beschränkt bleibt. Oft werden wir jedoch bei der Verwendung der Landau-Symbole das Gleichheitszeichen nutzen, also*

$$f = O(g)$$

*schreiben. Dies bedeutet keine echte Gleichheit. Der Schluss*

$$f_1 = O(g), \quad f_2 = O(g) \quad \Rightarrow \quad f_1 = f_2$$

*ist im Allgemeinen falsch. $f = O(g)$ ist nur eine Schreibweise für $f \in O(g)$. In vielen Fällen ist diese jedoch praktisch. Wir können dann mit den Landau-Symbolen einfach rechnen und schreiben zum Beispiel*

$$f(n) = \frac{1}{1 - \frac{1}{n}} = 1 + \frac{1}{n} + O\left(\frac{1}{n^2}\right),$$

*um zu verdeutlichen, dass für das Restglied*

$$R := f(n) - \frac{1}{1 - \frac{1}{n}}$$

*gilt:*

$$R \in O\left(\frac{1}{n^2}\right)$$

*Das dies wirklich der Fall ist, ist einfach mit Taylor-Entwicklung von $1/(1-x)$ für $x = 1/n$ nachzurechnen.*

Zum Ende dieses Abschnitts kommen wir noch mal auf das Horner-Schema zurück. Wie wir bereits gesehen haben, hat diese Art der Polynomauswertung sehr praktische Anwendungen.

Das Horner-Schema kann verallgemeinert werden, um auch die Ableitungen des Polynoms zu berechnen. Dieser Algorithmus ist dann als *vollständiges Horner-Schema* bekannt. Wir rekapitulieren zuerst in allgemeinerer Form das einfache Horner-Schema. Das Polynom $p$ habe die Darstellung

$$p(x) = \sum_{k=0}^{n} a_k^{(0)} x^k, \quad \text{wobei } a_n^{(0)} \neq 0,$$

und wir setzen $a_k^{(0)} := a_k$. Der Index oben erlaubt eine konsistente Verallgemeinerung des vereinfachten Horner-Schemas. Die Berechnung von $p(\xi)$ resultiert in der Darstellung

$$p(\xi) = \sum_{k=0}^{n} a_k^{(0)} \xi^k. \tag{1.5}$$

Dann liefert Algorithmus 1.2 die Rekursion:

$$\begin{aligned} a_n^{(1)} &= a_n^{(0)}, \\ a_j^{(1)} &= a_j^{(0)} + a_{j+1}^{(1)} \xi, \quad \text{für } j = n-1, \ldots, 0, \end{aligned} \tag{1.6}$$

mit

$$p(\xi) = a_0^{(1)}. \tag{1.7}$$

Zur Herleitung des vollständigen Horner-Schemas benötigen wir zunächst das Zwischenresultat der linearen Polynomdivision der Form

$$p(x):(x-\xi)$$

mit $\xi \in \mathbb{R}$.

**Satz 1.10** *Es seien $p$ und $q$ Polynome vom Grad $n$ bzw. $n-1$:*

$$p(x) = \sum_{j=0}^{n} a_j^{(0)} x^j, \quad q(x) = \sum_{j=1}^{n} a_j^{(0)} x^{j-1}.$$

*Des Weiteren sei $\xi \in \mathbb{R}$ beliebig und $a_0^{(1)} = p(\xi)$ (siehe (1.7)) aufgrund der Rekursion (1.6). Dann gilt die folgende Darstellung:*

$$p(x) = q(x)(x-\xi) + p(\xi)$$

**Beweis:** Der Beweis besteht aus einer geschickten Abspaltung des Linearfaktors $(x-\xi)$ unter Ausnutzung der Rekursionsformel (1.6):

$$p(x) = \sum_{j=0}^{n} a_j^{(0)} x^j = \sum_{j=0}^{n} (a_j^{(1)} - a_{j+1}^{(1)} \xi) x^j = \sum_{j=0}^{n} a_j^{(1)} x^j - \xi \sum_{j=0}^{n} a_{j+1}^{(1)} x^j$$

$$
\begin{aligned}
&= a_0^{(1)} + \sum_{j=1}^{n} a_j^{(1)} x^j - \xi \sum_{j=1}^{n+1} a_j^{(1)} x^{j-1}, \quad \text{wobei } a_{n+1}^{(1)} := 0 \\
&= a_0^{(1)} + x \sum_{j=1}^{n} a_j^{(1)} x^{j-1} - \xi \sum_{j=1}^{n} a_j^{(1)} x^{j-1}, \\
&= a_0^{(1)} + x \cdot q(x) - \xi \cdot q(x) \\
&= q(x)(x-\xi) + a_0^{(1)} \\
&= q(x)(x-\xi) + p(\xi).
\end{aligned}
$$

□

Wir bemerken abschließend, dass im vorherigen Beweis $a_{n+1}^{(1)} := 0$ per Definition gelten muss, da sonst bereits in der ersten Zeile des Beweises ein Widerspruch zum maximalen Grad $n$ des Polynoms $p(x)$ entstehen würde.

Die Rekursion (1.6) kann nun wie folgt verallgemeinert werden:

$$
\begin{aligned}
&a_n^{(1)} = a_n^{(2)} = a_n^{(3)} = \ldots = a_n^{(n+1)} := a_n^{(0)}, \\
&a_j^{k+1} = a_j^k + a_{j+1}^{k+1} \xi, \quad k = 0, \ldots, n-1, \quad j = n-1, \ldots k.
\end{aligned} \tag{1.8}
$$

Im Folgenden sei $p_{n-k}(x)$ ein Polynom vom Grad $n-k$, sprich

$$
p_{n-k}(x) = \sum_{j=k}^{n} a_j^{(k)} x^{j-k}.
$$

Mithilfe der verallgemeinerten Rekursion gilt dann nach Abspaltung eines Linearfaktors $(x-\xi)$ die Zerlegung

$$
p_{n-k}(x) = p_{n-(k+1)}(x)(x-\xi) + a_k^{(k+1)},
$$

welche durch Satz 1.10 begründet ist. Um nun den Ableitungswert an der Stelle $\xi$ berechnen zu können, arbeiten wir mit der Taylor-Darstellung:

**Satz 1.11** *Es sei ein Polynom $p(x)$ vom Grad $n$ gegeben. Dann erhalten wir für das $n$-te Taylor-Polynom die Darstellung:*

$$
p(x) = \sum_{k=0}^{n} a_k^{(k+1)} (x-\xi)^k
$$

*Insbesondere gilt:*

$$
p^{(k)}(\xi) = k! \cdot a_k^{(k+1)}
$$

**Beweis:** Der erste Teil basiert auf vollständiger Induktion:

$$
\begin{aligned}
p(x) = p_n(x) &= p_{n-1}(x)(x-\xi) + a_0^{(1)} \\
&= (p_{n-2}(x)(x-\xi) + a_1^{(2)})(x-\xi) + a_0^{(1)} \\
&= p_{n-2}(x)(x-\xi)^2 + a_1^{(2)}(x-\xi) + a_0^{(1)}
\end{aligned}
$$

$$= \dots$$
$$= p_0(x)(x-\xi)^n + a_{n-1}^{(n)}(x-\xi)^{n-1} + \dots + a_1^{(2)}(x-\xi) + a_0^{(1)}$$

Schließlich identifizieren wir noch $p_0(x)$ als Koeffizient $a_n^{(n+1)}$. Für Teil zwei arbeiten wir mit Koeffizientenvergleich:

$$\sum_{k=0}^{n} \frac{p^{(k)}(\xi)}{k!}(x-\xi)^k = p(x) = \sum_{k=0}^{n} a_k^{(k+1)}(x-\xi)^k,$$

woraus unmittelbar

$$\frac{p^{(k)}(\xi)}{k!} = a_k^{(k+1)}$$

folgt. Also ist

$$p^{(k)}(\xi) = k! \cdot a_k^{(k+1)}.$$ □

Mithilfe der Rekursion (1.8) und dem vorangegangenen Satz erhalten wir zur Berechnung der Ableitungen $p^{(k)}(\xi)$ eines Polynoms den folgenden Algorithmus:

**Algorithmus 1.3 Vollständiges Horner-Schema**
Gegeben sei das Polynom $p(x) = a_0^{(0)} + a_1^{(0)}x + \dots + a_n^{(0)}x^n$ mit Auswertungsstelle $\xi \in \mathbb{R}$.

```
1  a_n^(1) = a_n^(0), a_n^(2) = a_n^(0), ..., a_n^(n+1) = a_n^(0)
2  Für k von 0 bis n−1
3    Für j von n−1 bis k
4      a_j^(k+1) = a_j^(k) + a_{j+1}^(k+1) ξ
5    p^(k)(ξ) = k! · a_k^(k+1)
```

Dieser Algorithmus kann anhand des folgenden Schemas veranschaulicht werden:

| | $a_n^{(0)}$ | $a_{n-1}^{(0)}$ | $a_{n-2}^{(0)}$ | $\dots$ | $a_2^{(0)}$ | $a_1^{(0)}$ | $a_0^{(0)}$ |
|---|---|---|---|---|---|---|---|
| $\xi$ | $a_n^{(1)}$ | $a_{n-1}^{(1)}$ | $a_{n-2}^{(1)}$ | $\dots$ | $a_2^{(1)}$ | $a_1^{(1)}$ | $a_0^{(1)}$ |
| $\xi$ | $a_n^{(2)}$ | $a_{n-1}^{(2)}$ | $a_{n-2}^{(2)}$ | $\dots$ | $a_2^{(2)}$ | $a_1^{(2)}$ | |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\ddots$ | | | |
| $\xi$ | $a_n^{(n-1)}$ | $a_{n-1}^{(n-1)}$ | $a_{n-2}^{(n-1)}$ | | | | |
| $\xi$ | $a_n^{(n)}$ | $a_{n-1}^{(n)}$ | | | | | |
| $\xi$ | $a_n^{(n+1)}$ | | | | | | |

Wir führen das obige Beispiel fort:

**Beispiel 1.12 (Vollständiges Horner-Schema)**

Wir berechnen die Werte der Ableitungen des Polynoms $p(x) = 3x^5 - 2x^4 + 1$ an der Stelle $\xi = 2$. Dazu erhalten wir das folgende Schema:

| $i$ | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|
| $a_i$ | 3 | -2 | 0 | 0 | 0 | 1 | |
| $p_i$ | 3 | 4 | 8 | 16 | 32 | 65 | $\Rightarrow \quad p(\xi) = 0! \cdot 65 = 65$ |
| $p_i'$ | 3 | 10 | 28 | 72 | 176 | | $\Rightarrow \quad p'(\xi) = 1! \cdot 176 = 176$ |
| $p_i''$ | 3 | 16 | 60 | 192 | | | $\Rightarrow \quad p''(\xi) = 2! \cdot 192 = 386$ |
| $p_i^{(3)}$ | 3 | 22 | 104 | | | | $\Rightarrow \quad p^{(3)}(\xi) = 3! \cdot 104 = 624$ |
| $p_i^{(4)}$ | 3 | 28 | | | | | $\Rightarrow \quad p^{(4)}(\xi) = 4! \cdot 28 = 672$ |
| $p_i^{(5)}$ | 3 | | | | | | $\Rightarrow \quad p^{(5)}(\xi) = 5! \cdot 3 = 360$ |

Wir beenden diesen Abschnitt mit dem zweiten Teil des Steuerbeispiels:

**Beispiel 1.13 (Berechnung des Spitzensteuersatzes mittels vollständigem Horner-Schema)**

Wir kommen zuletzt auf das Steuerbeispiel zurück und berechnen den Spitzensteuersatz an der Stelle $\xi = 48\,000$ EUR. Der Spitzensteuersatz zum Einkommen $x$ gibt an, welcher Anteil jedes zusätzlich verdienten Euros an Steuern abgeführt werden muss.

Die Spitzensteuersatzkurve ist ein Grenzsteuersatz und mathematisch durch die erste Ableitung $s'(x)$ der Steuerfunktion $s(x)$ gegeben. Wie wir leicht in Definition 1.5 sehen, ergibt sich für den 5. Einkommensbereich der Spitzensteuersatz von $s'(x) = 0{,}45$, d. h., 45 %, da $s(x) = 0{,}45x - 16\,164{,}53$. Zur Berechnung des Spitzensteuersatzes für unseren Ingenieur mit $\xi = 48\,000$ EUR Einkommen ist Vorsicht bei der Ableitung geboten, da die Funktion $s(z)$ in $z$ gegeben ist, wir aber an $x$ interessiert sind. Dementsprechend ist die Kettenregel anzuwenden, da $s(z) := s(z(x))$ und $s'(z(x)) \cdot z'(x)$ gilt.

Dann gilt für den mittleren Einkommensbereich (3), siehe Definition 1.5, das um die 1. Ableitung erweiterte Horner-Schema mit der Darstellung aus Tab. 1.2. Um nun den Spitzensteuersatz zu bekommen, müssen wir noch die Ableitung der inneren Funktion $z'(x)$ miteinbeziehen:

$$s'(\xi) = a_1^2 \cdot z'(x) = 3\,928{,}8 \cdot \frac{1}{10\,000},$$

da $z(x) = \frac{x - 13\,770}{10\,000}$. Der Spitzensteuersatz bei einem Einkommen von 48 000 EUR liegt also bei 0,39, sprich 39 %. Im Vergleich dazu liegt der Durchschnittssteuersatz, welcher durch $d(x) = s(x)/x$ definiert ist, bei $d(x) = \frac{11\,766}{48\,000} = 0{,}25$, sprich 25 %.

**Tab. 1.2** Vollständiges Horner-Schema zur Berechnung der Steuerbetragsfunktion und dessen Ableitung im Einkommensbereich (3) aus Definition 1.5

| $i$ | 2 | 1 | 0 | |
|---|---|---|---|---|
| $a_i$ | 223,76 | 2 397,00 | 939,57 | |
| $s_i$ | 223,76 | 3 162,90 | 11 766,00 | $= s(\xi)$ |
| $s_i'$ | 223,76 | 3 928,80 | | |

## 1.3 Fehler, Fehlerverstärkung und Konditionierung

Numerische Lösungen sind oft mit Fehlern behaftet. Fehlerquellen sind zahlreich: Die Eingabe kann mit einem Messfehler versehen sein, ein approximatives Verfahren wird die Lösung nicht exakt berechnen, sondern nur annähern, manche Aufgaben können nur mithilfe eines Computers oder Taschenrechners gelöst werden und so ist die Lösung mit *Rundungsfehlern* versehen. Wir definieren:

► **Definition 1.14 (Fehler)** Sei $\tilde{x} \in \mathbb{R}$ die Approximation einer Größe $x \in \mathbb{R}$. Mit $|\delta x| = |\tilde{x} - x|$ bezeichnen wir den *absoluten Fehler* und mit $|\delta x|/|x|$ den *relativen Fehler*.

Üblicherweise ist die Betrachtung des relativen Fehlers von größerer Bedeutung: Denn ein absoluter Messfehler von $100\,m$ ist klein, wenn man den Abstand zwischen Erde und Sonne zu bestimmen versucht, jedoch groß, wenn der Abstand zwischen Mensa und Mathematikgebäude gemessen werden soll.

Bei der Analyse von numerischen Verfahren spielen Fehler, insbesondere die Fortpflanzung von Fehlern, eine entscheidende Rolle. Wir betrachten ein Beispiel:

**Beispiel 1.15 (Größenbestimmung)**

Thomas will seine Größe $h$ bestimmen, hat allerdings kein Maßband zur Verfügung. Dafür hat er eine Uhr, einen Ball und im Physikunterricht gut aufgepasst. Zur Lösung der Aufgabe hat er zwei Ideen:

- Verfahren 1: Thomas lässt den Ball aus Kopfhöhe fallen und misst die Zeit $t_0$, bis der Ball auf dem Boden ankommt. Die Höhe berechnet er aus der Formel für den freien Fall,
$$y(t) = h - \frac{1}{2}gt^2 \quad \Rightarrow \quad h = \frac{1}{2}gt_0^2,$$
mit der Gravitationskonstanten $g = 9{,}81\,m/s^2$.
- Verfahren 2: Der Ball wird $2\,m$ über den Kopf geworfen und wir messen die Zeit bis der Ball wieder auf dem Boden angekommen ist. Die Strecke $s = 2\,m$ wird in $t' = \sqrt{2s/g}$ Sekunden zurückgelegt, hierfür benötigt der Ball eine Startgeschwindigkeit von $v_0 = gt' = \sqrt{2\,sg} \approx 6{,}26\,m/s$. Es gilt für die Flugbahn:
$$y(t) = h + v_0 t - \frac{1}{2}gt^2 \quad \Rightarrow \quad h = \frac{1}{2}gt_0^2 - v_0 t_0$$

Das zweite Verfahren wird gewählt, weil die Zeit $t_0$, die der Ball in Verfahren 1 zum Fallen benötigt, sehr klein ist. Große Messfehler werden vermutet. Wir führen zunächst Algorithmus 1 durch und messen 5-mal (exakte Lösung $h = 1{,}80\,m$ und $t_0 \approx 0{,}606\,s$). Die Ergebnisse sind in Tab. 1.3 zusammengefasst. In der letzten Spalte wurde als Zeit der Mittelwert aller Messergebnisse gewählt. Dies geschieht in der Hoffnung, den Messfehler zu optimieren.

**Tab. 1.3** Experiment 1: Größenbestimmung durch Fallenlassen eines Balles

| Messung | 0,58 s | 0,61 s | 0,62 s | 0,54 s | 0,64 s | 0,598 s |
|---|---|---|---|---|---|---|
| Messfehler (rel.) | 4% | 0,5% | 2% | 10% | 6% | 1% |
| Größe | 1,65 m | 1,82 m | 1,89 m | 1,43 m | 2,01 m | 1,76 m |
| Fehler (abs.) | 0,15 m | 0,02 m | 0,09 m | 0,37 m | 0,21 m | 0,04 m |
| Fehler (rel.) | 8% | 1 % | 5% | 21% | 12% | 2% |

**Tab. 1.4** Experiment 2: Größenbestimmung durch Hochwerfen eines Balles

| Messung | 1,60 s | 1,48 s | 1,35 s | 1,53 s | 1,45 s | 1,482 s |
|---|---|---|---|---|---|---|
| Messfehler (rel.) | 5% | 2,5% | 11% | <1% | 5% | 2% |
| Größe | 2,53 m | 1,47 m | 0,48 m | 1,90 m | 1,23 m | 1,49 m |
| Fehler (abs.) | 0,73 m | 0,33 m | 1,32 m | 0,10 m | 0,57 m | 0,31 m |
| Fehler (rel.) | 40% | 18% | 73% | 6% | 32% | 17% |

Wir führen nun Algorithmus 2 durch und messen 5-mal (exakte Lösung $h = 1{,}80\,m$ und $t_0 \approx 1{,}519\,s$). In Tab. 1.4 sind die Messergebnisse und ermittelten Größen zusammengefasst. In der letzten Spalte wird wieder der Mittelwert aller Messergebnisse betrachtet.

Trotz anfänglicher Zweifel erweist sich Algorithmus 1 als stabiler. Mögliche Fehlerquellen sind der Messfehler bei der Bestimmung der Zeit $t_0$ sowie bei Algorithmus 2 die Genauigkeit beim Erreichen der Höhe von $2\,m$. In Algorithmus 1 führt der Messfehler zu etwa dem doppelten relativen Fehler in der Größe. Bei Algorithmus 2 führen selbst kleine Fehler $\leq 1\,\%$ in den Messungen zu wesentlich größeren Fehlern im Ergebnis. Der immer noch kleine Fehler von 5 % bei der ersten Messung führt zu einem Ergebnisfehler von etwa 40 %. Auch bei Betrachtung des Mittelwerts über alle Messwerte ist die ermittelte Größe von $1{,}49\,m$ keine gute Näherung.

Wir wollen nun untersuchen, welche Auswirkung der Fehler in der Eingabe eines Algorithmus auf das Ergebnis hat. Hierzu sei die Aufgabe wieder allgemein durch die Vorschrift $A{:}x \mapsto y$ beschrieben. Die Eingabe $x$ sei mit einem Fehler $\delta x$ behaftet. Die gestörte Eingabe $\tilde{x} = x + \delta x$ liefert ein gestörtes Ergebnis $\tilde{y} = y + \delta y$:

$$\tilde{y} = A(\tilde{x}), \quad \delta y = \tilde{y} - y = A(\tilde{x}) - A(x) = A(x + \delta x) - A(x)$$

Wir dividieren durch $y = A(x)$ und erweitern rechts mit $\delta x$ sowie mit $x$

$$\frac{\delta y}{y} = \frac{A(x + \delta x) - A(x)}{\delta x} \frac{x}{A(x)} \frac{\delta x}{x}.$$

Auf der linken Seite verbleibt der relative Fehler im Ergebnis. Gehen wir nun davon aus, dass die Störung $\delta x$ infinitesimal klein ist, so bleibt rechts ein Differenzenquotient für die erste Ableitung der Aufgabe. Im Fall $|\delta x| \to 0$ gilt

$$\left|\frac{\delta y}{y}\right| \approx \left|\frac{\partial A(x)}{\partial x}\frac{x}{A(x)}\right| \cdot \left|\frac{\delta x}{x}\right|,$$

und wir nennen die Größe

$$\kappa_{A,x} := \frac{\partial A(x)}{\partial x}\frac{x}{A(x)},$$

die *Konditionszahl* der Aufgabe $A(\,\cdot\,)$ in Bezug auf die Eingabe $x$. Die Konditionszahl beschreibt die relative Fehlerverstärkung einer Aufgabe in Bezug auf eine Eingabegröße. Wir definieren:

► **Definition 1.16 (Konditionszahl)** Es sei $A{:}\mathbb{R}^n \to \mathbb{R}^m$. Wir nennen

$$\kappa_{i,j} := \frac{\partial A_i(x)}{\partial x_j}\frac{x_j}{A_i(x)}$$

die *relative Konditionszahl* der Aufgabe. Eine Aufgabe heißt *schlecht konditioniert*, falls $|\kappa_{i,j}| \gg 1$, ansonsten *gut konditioniert*. Im Fall $|\kappa_{i,j}| < 1$ spricht man von *Fehlerdämpfung*, ansonsten von *Fehlerverstärkung*.
Wir setzen das Beispiel zur experimentellen Größenbestimmung fort:

**Beispiel 1.17 (Größenbestimmung, Fortsetzung)**

- Verfahren 1: Zum Durchführen des Verfahrens $h(t_0)$ ist als Eingabe die gemessene Zeit $t_0$ erforderlich. Für die Konditionszahl gilt:

  $$\kappa_{h,t_0} := \frac{\partial h(t_0)}{\partial t_0}\frac{t_0}{h(t_0)} = gt_0\,\frac{t_0}{\frac{1}{2}gt_0^2} = 2$$

  Ein relativer Fehler in der Eingabe $\delta t_0/t_0$ kann demnach einen doppelt so großen relativen Fehler in der Ausgabe verursachen.
- Verfahren 2: Wir bestimmen die Konditionszahl in Bezug auf die Zeit $t_0$:

  $$\kappa_{h,t_0} = (gt_0 - v_0)\frac{t_0}{\frac{1}{2}gt_0^2 - v_0t_0} = 2\frac{gt_0 - v_0}{gt_0 - 2v_0}$$

  Wir wissen, dass der Ball bei exaktem Wurf und ohne Messfehler $t_0 \approx 1{,}519\,s$ unterwegs ist bei einer Startgeschwindigkeit von $v_0 \approx 6{,}26\,m/s$. Dies ergibt:

  $$\kappa_{h,t_0} \approx \kappa_{h,v_0} \approx 8$$

  Fehler in der Eingaben $\delta t_0/t_0$ sowie $\delta v_0/v_0$ werden um den Faktor 8 verstärkt. Die durch die Konditionszahlen vorhergesagten Fehlerverstärkungen lassen sich in den Tab. 1.3 und 1.4 zu Beispiel 1.15 gut wiederfinden.

## 1.4 Zahldarstellung

Die Analyse von Fehlern und Fehlerfortpflanzungen spielt eine zentrale Rolle in der numerischen Mathematik. Fehler treten vielfältig auf, auch ohne ungenaue Eingabewerte. Oft sind numerische Algorithmen sehr komplex, setzen sich aus vielen Operationen zusammen. Bei der Approximation mit dem Computer treten unweigerlich *Rundungsfehler* auf. Da der Speicherplatz im Computer bzw. die Anzahl der Ziffern auf dem Taschenrechner beschränkt ist, treten Rundungsfehler schon bei der bloßen Darstellung einer Zahl auf. Auch wenn es für die numerische Aufgabe

$$x^2 = 2, \quad \Leftrightarrow x = \pm\sqrt{2},$$

mit $x = \pm\sqrt{2}$ eine einfach anzugebende Lösung gibt, kann diese auf einem Computer nicht exakt dargestellt werden:

$$\sqrt{2} = 1{,}41421356237309504880\ldots$$

Der naheliegende Grund für einen zwingenden Darstellungsfehler ist der beschränkte Speicher eines Computers. Ein weiterer Grund liegt in der Effizienz. Ein Computer kann effizient nicht mit beliebig langen Zahlen rechnen. Grund ist die beschränkte Datenbandbreite (das sind die 8 Bit, 16 Bit, 32 Bit oder 64 Bit der Prozessoren). Operationen mit Zahlen in längerer Darstellung müssen zusammengesetzt werden, ähnlich dem schriftlichen Multiplizieren oder Addieren aus der Schule.

Computer speichern Zahlen gerundet in Binärdarstellung, also zur Basis 2:

$$\mathrm{rd}\,(x) = \pm \sum_{i=-n_1}^{n_2} a_i 2^i, \quad a_i \in \{0,1\}.$$

Die Genauigkeit der Darstellung somit der Rundungsfehler hängen von der Anzahl der Ziffern, also von $n_1$ und $n_2$ ab. Die *Fixkommadarstellung* der Zahl im Binärsystem lautet:

$$\mathrm{rd}\,(x) = [a_{n_2} a_{n_2-1} \ldots a_1 a_0 . a_{-1} a_{-2} \ldots a_{-n_1}]_2$$

Praktischer ist die *Gleitkommadarstellung* von Zahlen. Hierzu wird die Binärdarstellung normiert und ein gemeinsamer Exponent eingeführt:

$$\mathrm{rd}\,(x) = \pm \left( \sum_{i=-n_1-n_2}^{0} a_{i+n_2} 2^i \right) 2^{n_2}, \quad a_i \in \{0,1\}.$$

Der führende Term (die $a_i$) heißt die *Mantisse* und wird von uns mit $M$ bezeichnet, den *Exponenten* bezeichnen wir mit $E$. Der Exponent kann dabei eine positive oder negative Zahl sein. Zur Vereinfachung wird der Exponent $E$ als $E = e - b$ mit einer positiven Zahl $e \in \mathbb{N}$ und dem Bias $b \in \mathbb{N}$ geschrieben. Der Biaswert $b$ wird in einer konkreten Zahldarstellung fest gewählt und bestimmt somit den größtmöglichen negativen Exponenten. Der variable Exponentenanteil $e$ wird selbst im Binärformat gespeichert. Es bleibt, die

Anzahl der Binärstellen für Mantisse und Exponent zu wählen. Hinzu kommt ein Bit für das Vorzeichen $S \in \{+,-\}$. Die Gleitkommadarstellung im Binärsystem lautet:

$$\mathrm{rd}\,(x) = S \cdot [m_0.m_{-1}m_{-2}\ldots m_{-\#m}]_2 \cdot 2^{[e_{\#e}\ldots e_1 e_0]_2 - b}$$

Die Mantisse wird üblicherweise mit $m_0 = 1$ normiert zu $M \in [1,2)$. Das heißt, die führende Stelle muss nicht explizit gespeichert werden.

Auf modernen Computern hat sich das *IEEE-754*-Format[2] zum Speichern von Zahlen etabliert:

► **Definition 1.18 (Normalisierte Gleitkommazahlen im IEEE 754-Format)** Das IEEE 754-Format [36] beschreibt die *Gleitkommadarstellung* einer Zahl im *Binärformat*

$$x = s \cdot M \cdot 2^{e-b}$$

mit einem Vorzeichen $s \in \{+,-\}$, einer Mantisse $M \in [1,2)$ sowie einem Exponenten $e \in \mathbb{N}$ mit Bias $b \in \mathbb{N}$. Die Gleitkommazahl wird in Binärdarstellung gespeichert:

$$se_{\#e}\ldots e_2 e_1 m_{\#m}\ldots m_2 m_1,$$

mit $\#e$ Bit für den Exponenten und $\#m$ Bit für die Mantisse. Der Bias wird im Allgemeinen als $2^{\#e-1} - 1$ gewählt. Die Interpretation der Zahlen hängt vom Exponenten ab:

**Null** Im Fall $e = 0$ und $m_1 = \cdots = m_{\#m} = 0$ ist $x = \pm 0$. Die Unterscheidung zwischen +0 und –0 entsteht beim Runden kleiner Zahlen zur Null. Im direkten Vergleich interpretiert ein Computer +0 = –0.

**Unendlich** Im Fall $e = 2^{\#e} - 1$ (also alle Bit im Exponenten gleich 1) und $m_1 = \cdots = m_{\#m} = 0$ ist $x = \pm\infty$. Unendlich entsteht etwa beim Teilen durch 0 (mit Vorzeichen!) oder falls das Ergebnis zu groß (oder negativ zu klein) ist, um darstellbar zu sein.

**NaN** Im Fall $e = 2^{\#e} - 1$ und $M > 0$ steht der Wert für *not a number* und tritt zum Beispiel bei der Operation $0/0$ oder $\infty - \infty$ auf.

**Normalisierte Zahl** Im Fall $0 \leq e < 2^{\#e-1}$ steht die Zahl für

$$s \cdot 1.m_{\#m}\ldots m_2 m_1 \cdot 2^{e-b}.$$

Das Rechnen mit Gleitkommazahlen kann im Computer effizient realisiert werden. Im IEEE 754-Format wird die Gleitkommadarstellung durch *denormalisierte Zahlen* erweitert. Wir erwähnen dies zur Vollständigkeit:

**Bemerkung 1.19 (Denormalisierte Zahlen)** *Denormalisierte Zahlen dienen zum Schließen der Lücke zwischen Null und der kleinsten positiven darstellbaren Zahl* $1{,}0\ldots001\cdot 2^{-b}$.

[2] Das IEEE-754-Format wurde 1985 erstmalig als technischer Standard für binäre Gleitkommazahlen in Computern vom Institute of Electrical and Electronics Engineers (IEEE) festgelegt

**Tab. 1.5** IEEE 754-Format in einfacher und doppelter Genauigkeit sowie Gleitkommaformate in aktueller und historischer Hardware

| | Größe | Vorzeichen | Exponent | Mantisse | Bias |
|---|---|---|---|---|---|
| einfache Genauigkeit (single) | 32 Bit | 1 Bit | 8 Bit | 23+1 Bit | 127 |
| doppelte Genauigkeit (double) | 64 Bit | 1 Bit | 11 Bit | 52+1 Bit | 1023 |
| Zuse Z1 (1938) | 24 Bit | 1 Bit | 7 Bit | 15 Bit | — |
| IBM 704 (1954) | 36 Bit | 1 Bit | 8 Bit | 27 Bit | 128 |
| i8087 Coprozessor (1980) | Erste Verwendung von IEEE (single + double) | | | | |
| Intel 486 (1989) | Erste integrierte FPU in Standard-PC | | | | |
| NVIDIA G80 (2007) | GPU (single) | | | | |
| NVIDIA Fermi (2010) | GPU (double) | | | | |

*Im Fall* $e = 0$ *und* $M > 0$ *wird die Zahl interpretiert als:*

$$s \cdot 0.m_{\#m} \ldots m_2 m_1 \cdot 2^{1-b}$$

*Die Genauigkeit bei der Rechnung mit denormalisierten Zahlen ist reduziert.*

In Tab. 1.5 sind verschiedene Gleitkommadarstellungen zusammengefasst, die historisch und aktuell benutzt werden. Derzeit wird fast ausschließlich das IEEE-Format verwendet. Hier sind zwei Darstellungen üblich, *single-precision* (in C++ float) und *double-precision* (in C++ double). Durch die Normierung der Mantisse wird ein Bit, das sogenannte *hidden-bit*, gewonnen. Historisch wurden in Rechensystemen jeweils unterschiedliche Zahlendarstellungen gewählt. Während die ersten Computer noch im Prozessor integrierte Recheneinheiten für Gleitkomma-Arithmetik, die eine sogenannte *floating-point processing unit* (FPU) hatten, verschwand diese zunächst wieder aus den üblichen Computern und war nur in speziellen Rechnern vorhanden. In Form von *Coprozessoren* konnte eine FPU nachgerüstet werden (z. B. der Intel 8087 zum Intel 8086). Der „486er“ war der erste Prozessor für Heimcomputer mit integrierter FPU.

Heute können Gleitkommaberechnungen effizient auf Grafikkarten ausgelagert werden. Die Prozessoren der Grafikkarten, die *graphics processing units* (GPU) sind speziell für solche Berechnungen ausgelegt (z. B. schnelle Berechnung von Lichtbrechungen und Spiegelungen, Abbilden von Mustern auf 3D-Oberflächen). Spezielle Steckkarten (z. B. NVIDIA Tesla), die gleich mehrere GPUs enthalten, werden in Höchstleistungssystemen eingesetzt. Die Genauigkeit der Darstellung ist im Wesentlichen von den in der Mantisse zu Verfügung stehenden Stellen bestimmt. Größte und kleinste darstellbare Zahlen sind durch die Stellen im Exponenten bestimmt. In numerischen Verfahren ist die Verwendung von doppelt-genauer Zahlendarstellung (double) üblich. Die Recheneinheiten moderner Computer nutzen intern eine erhöhte Genauigkeit zum Durchführen von elementaren Operationen (80 Bit bei modernen Intel-CPUs). Gerundet wird erst nach Berechnung des Ergebnisses.

**Beispiel 1.20 (Gleitkommadarstellung)**

Wir gehen von vierstelliger Mantisse und vier Stellen im Exponenten aus mit Bias $2^{4-1} - 1 = 7$.

- Die Zahl $x = -96$ hat zunächst negatives Vorzeichen, also $S = 1$. Die Binärdarstellung von 96 ist

$$96_{10} = 1 \cdot 64 + 1 \cdot 32 + 0 \cdot 16 + 0 \cdot 8 + 0 \cdot 4 + 0 \cdot 2 + 0 \cdot 1 = 1100000_2,$$

  normalisiert

$$96_{10} = 1{,}1000_2 \cdot 2^{6_{10}} = 1{,}1_2 \cdot 2^{13_{10}-7_{10}} = 1{,}1_2 \cdot 2^{1101_2-b}.$$

  Als Gleitkommadarstellung ergibt sich $111011000_2$.
- Die Zahl $x = -384$ hat wieder negatives Vorzeichen und $S = 1$. Die Binärdarstellung von 384 ist:

$$\begin{aligned} 384_{10} &= 1 \cdot 256 + 1 \cdot 128 + 0 \cdot 64 + 0 \cdot 32 + 0 \cdot 16 + 0 \cdot 8 + 0 \cdot 4 + 0 \cdot 2 + 0 \cdot 1 \\ &= 110000000_2, \end{aligned}$$

  also normalisiert

$$384_{10} = 1{,}1000 \cdot 2^{8_{10}} = 1{,}1000 \cdot 2^{15_{10}-7_{10}} = 1{,}1000_2 \cdot 2^{1111_2-b}.$$

  Alle Bits im Exponenten sind 1. Der spezielle Wert $e = 1111_2$ ist jedoch zur Speicherung von NaN (not a number) vorgesehen. Die Zahl 384 ist zu groß, um in diesem Zahlenformat gespeichert zu werden. Stattdessen wird $-\infty$ oder binär $111110000_2$ gespeichert.
- Die Zahl $x = 1/3 = 0{,}33333\ldots$ ist positiv, also $S = 0$. Die Binärdarstellung von $1/3$ ist

$$\frac{1}{3} = 0{,}01010101\ldots_2.$$

  Normalisiert mit vierstelliger Mantisse erhalten wir

$$\frac{1}{3} \approx 1{,}0101_2 \cdot 2^{-2} = 1{,}0101_2 \cdot 2^{5_{10}-7_{10}} = 1{,}0101_2 \cdot 2^{0101_2-b},$$

  also die Binärdarstellung $001010101_2$. Wir mussten bei der Darstellung runden und erhalten rückwärts

$$1{,}0101_2 \cdot 2^{0101_2-7} = 1{,}3125 \cdot 2^{-2} = 0{,}328125$$

mit dem relativen Fehler

$$\left| \frac{\frac{1}{3} - 1{,}0101_2 \cdot 2^{0101_2 - b}}{\frac{1}{3}} \right| \approx 0{,}016.$$

- Die Zahl $x = \sqrt{2} \approx 1{,}4142135623\ldots$ ist positiv, also $S = 0$. Gerundet gilt:

$$\sqrt{2} \approx 1{,}0111_2$$

Diese Zahl liegt bereits normalisiert vor. Mit Bias $b = 7$ gilt für den Exponenten $e = 0 = 7 - 7$

$$\sqrt{2} \approx 1{,}0111_2 \cdot 2^{0111_2 - b},$$

also 001110111. Rückwärts in Dezimaldarstellung erhalten wir

$$1{,}0111_2 \cdot 2^{0111_2 - b} = 1{,}4375$$

mit dem relativen Darstellungsfehler

$$\left| \frac{\sqrt{2} - 1{,}4375}{\sqrt{2}} \right| \approx 0{,}016.$$

- Schließlich betrachten wir die Zahl $x = -0{,}003$. Mit $S = 1$ gilt:

$$0{,}003_{10} \approx 0{,}000000001100012$$

und normalisiert

$$0{,}003_{10} \approx 1{,}1001_2 \cdot 2^{-9} = 1{,}1001_2 \cdot 2^{-2-7}.$$

Der Exponent $-9 = -2 - b$ ist zu klein und kann in diesem Format (also vier Stellen für den Exponenten) nicht dargestellt werden. Die Zahl kann hingegen denormalisiert dargestellt werden als

$$0{,}003_{10} \approx \cdot 0{,}0011_2 \cdot 2^{1-7}.$$

Binär ergibt dies $100000011_2$. Umrechnen in Dezimaldarstellung liefert

$$0{,}0011_2 \cdot 2^{1-7} = 0{,}1875 \cdot 2^{-6} \approx 0{,}0029$$

mit dem relativen Darstellungsfehler

$$\left| \frac{0{,}003 - 0{,}0029}{0{,}003} \right| \approx 0{,}033.$$

Aus der begrenzten Anzahl von Ziffern ergibt sich zwangläufig ein Fehler bei der Durchführung von numerischen Algorithmen. Der relative Fehler, der bei der Computerdarstellung $\tilde{x}$ einer Zahl $x \in \mathbb{R}$ entstehen kann,

$$\left| \frac{\mathrm{rd}\,(x) - x}{x} \right|,$$

ist durch die sogenannte *Maschinengenauigkeit* beschränkt: .

► **Definition 1.21 (Maschinengenauigkeit)** Die *Maschinengenauigkeit eps* ist der maximale relative Rundungsfehler der Zahldarstellung und wird bestimmt als:

$$eps := \inf\{x > 0 \colon \mathrm{rd}\,(1 + x) > 1\}$$

Sind im Zahlenformat denormalisierte Zahlen vorgesehen, so verschlechtert sich die Genauigkeit für $x \to 0$.

Da Rundungsfehler zwangsläufig auftreten, gelten grundlegende mathematische Gesetze wie das Assoziativgesetz

$$(a + b) + c = a + (b + c), \quad (a \cdot b) \cdot c = a \cdot (b \cdot c)$$

oder das Distributivgesetz

$$a \cdot (b + c) = ab + ac, \quad (a + b) \cdot c = ac + bc$$

auf Computern nicht mehr. Aufgrund von Rundungsfehlern spielt die Reihenfolge, in der Operationen ausgeführt werden, eine wichtige Rolle und verändert das Ergebnis. Auch ein einfacher Vergleich von Zahlen ist oft nicht möglich, die Abfrage `if (3,8/10,0==0,38)` kann durch Rundung das falsche Ergebnis liefern und muss durch Abfragen der Art `if (fabs ↪ (3,8/10,0 – 0,38) < eps)` ersetzt werden (der Befehl `fabs( )` steht in C/C++ für den Betrag einer Zahl).

**Bemerkung 1.22** *Die Maschinengenauigkeit hat nichts mit der kleinsten Zahl zu tun, die auf einem Computer darstellbar ist. Diese ist wesentlich durch die Anzahl der Stellen im Exponenten bestimmt. Die Maschinengenauigkeit wird durch die Anzahl der Stellen in der Mantisse bestimmt. Bei der üblichen Gleitkommadarstellung mit doppelter Genauigkeit (double in C++) gilt* $eps \approx 10^{-16}$*, bei einfacher Genauigkeit, z. B. auf Grafikkarten, gilt* $eps \approx 10^{-8}$.

Rundungsfehler treten bei jeder elementaren Operation auf. Daher kommt der Konditionierung der Grundoperationen, aus denen alle Algorithmen aufgebaut sind, eine entscheidende Bedeutung zu:

**Beispiel 1.23 (Konditionierung der Grundoperationen)**

1. Addition, Subtraktion: $A(x, y) = x + y$:

$$\kappa_{A,x} = \left| \frac{x}{x+y} \right| = \left| \frac{1}{1+\frac{y}{x}} \right|$$

Im Fall $x \approx -y$ kann die Konditionszahl der Addition ($x \approx y$ bei der Subtraktion) beliebig groß werden. Ein Beispiel mit vierstelliger Rechnung:

$$x = 1{,}021, \quad y = -1{,}019 \quad \Rightarrow \quad x + y = 0{,}002$$

Jetzt sei $\tilde{y} = 1{,}020$ gestört. Der relative Fehler in $y$ ist sehr klein: $|1{,}019 - 1{,}020|/|1{,}019| \leq 0{,}1\,\%$. Wir erhalten das gestörte Ergebnis

$$x + \tilde{y} = 0{,}001$$

und einen Fehler von 100 %. Die enorme Fehlerverstärkung bei der Addition von Zahlen mit etwa gleichem Betrag wird *Auslöschung* genannt. Hier gehen wesentliche Stellen verloren. Auslöschung tritt üblicherweise dann auf, wenn das Ergebnis einer numerischen Operation verglichen mit den Eingabewerten einen sehr kleinen Betrag hat. Kleine relative Fehler in der Eingabe verstärken sich zu großen relativen Fehlern im Ergebnis.

2. Multiplikation: $A(x, y) = x \cdot y$. Die Multiplikation zweier Zahlen ist stets gut konditioniert:

$$\kappa_{A,x} = \left| y \frac{x}{xy} \right| = 1$$

3. Division: $A(x, y) = x/y$. Dasselbe gilt für die Division:

$$\kappa_{A,x} = \left| \frac{1}{y} \frac{x}{\frac{x}{y}} \right| = 1, \quad \kappa_{A,y} = \left| \frac{x}{y^2} \frac{y}{\frac{x}{y}} \right| = 1$$

4. Wurzelziehen: $A(x) = \sqrt{x}$:

$$\kappa_{A,x} = \left| \frac{1}{2\sqrt{x}} \frac{x}{\sqrt{x}} \right| = \frac{1}{2}$$

Ein Fehler in der Eingabe wird im Ergebnis sogar reduziert.

Die meisten numerischen Algorithmen bestehen im Kern aus der wiederholten Ausführung dieser Grundoperationen. Der Aufwand von Algorithmen wird in der Anzahl der notwendigen elementaren Operationen (in Abhängigkeit von der Problemgröße) gemessen. Die Laufzeit eines Algorithmus hängt wesentlich von der Leistungsfähigkeit des Rechners ab. Diese wird in *FLOPS*, also *floating point operations per second* gemessen.

**Tab. 1.6** Gleitkommageschwindigkeit sowie Anschaffungskosten einiger aktueller und historischer Computer

| CPU | Jahr | FLOPS | Preis | Energieverbrauch |
|---|---|---|---|---|
| Zuse Z3 | 1941 | 0,3 | Einzelstücke | 4 kW |
| IBM 704 | 1955 | $5 \cdot 10^3$ | >10 000 000 EUR | 75 kW |
| Intel 8086 + 8087 | 1980 | $50 \cdot 10^3$ | 2 000 EUR | 200 W |
| i486 | 1991 | $1{,}4 \cdot 10^6$ | 2 000 EUR | 200 W |
| IPhone 4s | 2011 | $100 \cdot 10^6$ | 500 EUR | 10 W |
| 2 x Pentium III | 2001 | $800 \cdot 10^6$ | 2 000 EUR | 200 W |
| Core i7 | 2011 | $100 \cdot 10^9$ | 1 000 EUR | 200 W |
| Helics I (Parallelrechner in Heidelberg) | 2002 | $1 \cdot 10^{12}$ | 1 000 000 EUR | 40 kW |
| NVIDIA GTX 580 (GPU) | 2010 | $1 \cdot 10^{12}$ | 500 EUR | 250 W |
| K Computer | 2011 | $10 \cdot 10^{15}$ | >500 000 000 EUR | 12 000 kW |
| Sunway TaihuLight | 2016 | $93 \cdot 10^{15}$ | 250 000 000 EUR | 15 500 kW |

In Tab. 1.6 fassen wir die erreichbaren FLOPS für verschiedene Computersysteme zusammen. Die Leistung ist im Laufe der Jahre rapide gestiegen, alle zehn Jahre wird etwa der Faktor 1 000 erreicht. Die Leistungssteigerung beruht zum einen auf effizienterer Hardware. Erste Computer hatten Register mit 8 Bit Breite, die in jedem Takt (das ist die MHz-Angabe) verarbeitet werden konnten. Auf aktueller Hardware stehen Register mit 64 Bit zur Verfügung. Hinzu kommt eine effizientere Abarbeitung von Befehlen durch sogenanntes *Pipelining*: Die übliche Abfolge im Prozessor beinhaltet „Speicher lesen, Daten bearbeiten, Ergebnis speichern“. Das Pipelining erlaubt es dem Prozessor, schon den Speicher für die nächste Operation auszulesen, während noch die aktuelle bearbeitet wird. So konnte die Anzahl der notwendigen Prozessortakte pro Rechenoperation erheblich reduziert werden. Die Kombination aus Intel 8086 mit FPU 8087 hat bei einem Takt von 8 Mhz und 50 000 FLOPS etwa 150 Takte pro Fließkommaoperation benötigt. Der 486er braucht bei 66 MHz und etwa 1 000 000 FLOPS nur 50 Takte pro Operation. Der Pentium III liegt bei etwa 5 Takten pro Fließkommaoperation.

In den letzten Jahren beruht die Effizienzsteigerung wesentlich auf einem sehr hohen Grad an Parallelisierung. Ein aktueller Core I7-Prozessor kann mehrere Operationen gleichzeitig durchführen. Eine NVIDIA 580-GPU erreicht ihre Leistung mit über 500 Rechenkernen. Um diese Leistung effizient nutzen zu können, müssen die Algorithmen entsprechend angepasst werden, sodass auch alle 500 Kerne ausgelastet werden. Kann ein Algorithmus diese spezielle Architektur nicht ausnutzen, so fällt die Leistung auf etwa ein GigaFLOP zurück, also auf das Niveau des Pentium III aus dem Jahr 2001. Werden die 500 Kerne hingegen optimal ausgenutzt, so erreicht eine Grafikkarte die gleiche Leistung wie der Heidelberger Linux-Cluster Helics aus dem Jahr 2002.

Modernste Supercomputer wie der *Sunway TaihuLight* (schnellster Computer der Welt in 2016) vernetzen über 10 000 000 Kerne. Einen aktuellen Überblick gibt die *Top 500*

*Liste der Supercomputer*.[3] Bei parallelen Höchstleistungsrechnern spielt auch der Stromverbrauch eine bedeutende Rolle. Der *Sunway TaihuLight* verbraucht im Jahr so viel Strom wie 70 000 Haushalte. Jede Nutzungsstunde schlägt mit einer Stromrechnung von einigen Tausend Euro zu Buche. Durch neue Bauweisen, insbesondere durch kleineres Layout der Platinen, kann der Stromverbrauch pro FLOP ständig gesenkt werden.

## 1.5 Rundungsfehleranalyse und Stabilität von numerischen Algorithmen

Beim Entwurf von numerischen Algorithmen für eine Aufgabe sind oft unterschiedliche Wege möglich, die sich z. B. in der Reihenfolge der Verfahrensschritte unterscheiden. Unterschiedliche Algorithmen zu ein und derselben Aufgabe können dabei rundungsfehlerbedingt zu unterschiedlichen Ergebnissen führen:

**Beispiel 1.24 (Distributivgesetz)**

Wir betrachten die Aufgabe $A(x, y, z) = x \cdot z - y \cdot z = (x - y)z$ und zur Berechnung zwei Vorschriften:

$$a_1 := x \cdot z, \qquad a_1 := x - y,$$
$$a_2 := y \cdot z, \qquad a := a_1 \cdot z,$$
$$a := a_1 - a_2.$$

Es sei $x = 0{,}519$, $y = 0{,}521$, $z = 0{,}941$. Bei vierstelliger Arithmetik erhalten wir:

$$a_1 := \mathrm{rd}\,(x \cdot z) = 0{,}4884, \qquad b_1 := \mathrm{rd}\,(x - y) = -0{,}002,$$
$$a_2 := \mathrm{rd}\,(y \cdot z) = 0{,}4903, \qquad b_2 := \mathrm{rd}\,(a_1 \cdot z) = -0{,}001882,$$
$$a_3 := \mathrm{rd}\,(a_1 - a_2) = -0{,}0019.$$

Mit $A(x, y, z) = -0{,}001882$ ergeben sich die relativen Fehler:

$$\left|\frac{-0{,}0001882 - a_3}{0{,}0001882}\right| \approx 0{,}01, \qquad \left|\frac{-0{,}0001882 - b_2}{0{,}0001882}\right| = 0.$$

Das Distributivgesetz gilt auf dem Computer nicht!

Die Stabilität hängt also entscheidend vom Design des Algorithmus ab. Eingabefehler oder auch Rundungsfehler, die in einzelnen Schritten entstehen, werden in darauffolgenden Schritten des Algorithmus verstärkt. Wir analysieren nun beide Verfahren im Detail und

[3] https://www.top500.org Liste der jeweils 500 schnellsten Computer der Welt.

gehen davon aus, dass in jedem der elementaren Schritte (wir haben hier nur Addition und Multiplikation) ein relativer Rundungsfehler $\epsilon$ mit $|\epsilon| \leq eps$ entsteht, also

$$\mathrm{rd}\,(x+y) = (x+y)(1+\epsilon), \quad \mathrm{rd}\,(x\cdot y) = (x\cdot y)(1+\epsilon).$$

Zur Analyse eines gegebenen Algorithmus verfolgen wir die Rundungsfehler, welche in jedem Schritt entstehen, und deren Akkumulation:

**Beispiel 1.25 (Stabilität des Distributivgesetzes)**

Wir berechnen zunächst die Konditionszahlen der Aufgabe:

$$\kappa_{A,x} = \left|\frac{1}{1-\frac{y}{x}}\right|, \quad \kappa_{A,y} = \left|\frac{1}{1-\frac{x}{y}}\right|, \quad \kappa_{A,z} = 1.$$

Für $x \approx y$ ist die Aufgabe schlecht konditioniert. Wir starten mit Algorithmus 1 und schätzen in jedem Schritt den Rundungsfehler ab. Zusätzlich betrachten wir Eingabefehler (oder Darstellungsfehler) von $x, y$ und $z$. Wir berücksichtigen stets nur Fehlerterme erster Ordnung und fassen alle weiteren Terme mit den Landau-Symbolen (für kleine $\epsilon$) zusammen:

$$\begin{aligned}
a_1 &= x(1+\epsilon_x)z(1+\epsilon_z)(1+\epsilon_1) = xz(1+\epsilon_x+\epsilon_z+\epsilon_1+O(eps^2)) \\
&= xz(1+3\epsilon_1+O(eps^2)) \\
a_2 &= y(1+\epsilon_y)z(1+\epsilon_z)(1+\epsilon_2) = yz(1+\epsilon_y+\epsilon_z+\epsilon_2+O(eps^2)) \\
&= yz(1+3\epsilon_2+O(eps^2)) \\
a_3 &= (xz(1+3\epsilon_1) - yz(1+3\epsilon_2)+O(eps^2))(1+\epsilon_3) \\
&= (xz-yz)(1+\epsilon_3) + 3xz\epsilon_1 - 3yz\epsilon_2 + O(eps^2)
\end{aligned}$$

Wir bestimmen den relativen Fehler:

$$\left|\frac{a_3-(xz-yz)}{xz-yz}\right| = \frac{|(xz-yz)\epsilon_3 + 3xz\epsilon_1 - 3yz\epsilon_2|}{|xz-yz|} \leq eps + 3\frac{|x|+|y|}{|x-y|}eps$$

Die Fehlerverstärkung dieses Algorithmus kann für $x \approx y$ groß werden und entspricht etwa (Faktor 3) der Konditionierung der Aufgabe. Wir nennen den Algorithmus daher stabil.

Wir betrachten nun Algorithmus 2:

$$\begin{aligned}
a_1 &= (x(1+\epsilon_x) - y(1+\epsilon_y))(1+\epsilon_1) \\
&= (x-y)(1+\epsilon_1) + x\epsilon_x - y\epsilon_y + O(eps^2)
\end{aligned}$$

$$\begin{aligned} a_2 &= z(1+\epsilon_z)\big((x-y)(1+\epsilon_1)+x\epsilon_x-y\epsilon_y+O(eps^2)\big)(1+\epsilon_2) \\ &= z(x-y)(1+\epsilon_1+\epsilon_2+\epsilon_z+O(eps)^2)+zx\epsilon_x-zy\epsilon_y+O(eps^2) \end{aligned}$$

Für den relativen Fehler gilt in erster Ordnung:

$$\begin{aligned} \left|\frac{a_2-(xz-yz)}{xz-yz}\right| &= \frac{|z(x-y)(\epsilon_1+\epsilon_2+\epsilon_z)+zx\epsilon_x-zy\epsilon_y|}{|xz-yz|} \\ &\leq 3eps+\frac{|x|+|y|}{|x-y|}eps \end{aligned}$$

Die Fehlerverstärkung kann für $x \approx y$ wieder groß werden. Der Verstärkungsfaktor ist jedoch geringer als bei Algorithmus 1. Insbesondere fällt auf, dass dieser zweite Algorithmus bei fehlerfreien Eingabedaten keine Fehlerverstärkung aufweist. (Das ist der Fall $\epsilon_x = \epsilon_y = \epsilon_z = 0$.)

Beide Algorithmen sind stabil, der zweite hat bessere Stabilitätseigenschaften als der erste.

Wir nennen die beiden Algorithmen stabil, obwohl der eine wesentlich bessere Stabilitätseigenschaften hat. Der Stabilitätsbegriff dient daher oft zum relativen Vergleich verschiedener Algorithmen. Wir definieren:

► **Definition 1.26 (Stabilität)** Ein numerischer Algorithmus zum Lösen einer Aufgabe heißt *stabil*, falls die bei der Durchführung akkumulierten Rundungsfehler den durch die Kondition der Aufgabe gegebenen unvermeidlichen Fehler nicht übersteigen.

Wir halten fest: Für ein schlecht konditioniertes Problem existiert kein stabiler Algorithmus mit einer geringeren Fehlerfortpflanzung als durch die Konditionierung bestimmt. Für gut konditionierte Probleme können jedoch beliebig instabile Verfahren existieren.

Der wesentliche Unterschied zwischen beiden Algorithmen aus dem Beispiel ist die Reihenfolge der Operationen. In Algorithmus 1 ist der letzte Schritt eine Subtraktion, deren schlechte Kondition wir unter dem Begriff *Auslöschung* kennengelernt haben. Bereits akkumulierte Rundungsfehler zu Beginn des Verfahrens werden hier noch einmal wesentlich verstärkt. Bei Algorithmus 2 werden durch die abschließende Multiplikation die Rundungsfehler, die zu Beginn auftreten, nicht weiter verstärkt. Aus dem analysierten Beispiel leiten wir eine Regel her:

Bei dem Entwurf von numerischen Algorithmen sollen schlecht konditionierte Operationen zu Beginn durchgeführt werden.

http://www.springer.com/978-3-662-54177-7

Einführung in die Numerische Mathematik
Begriffe, Konzepte und zahlreiche Anwendungsbeispiele
Richter, Th.; Wick, Th.
2017, X, 478 S. 61 Abb., 10 Abb. in Farbe., Softcover
ISBN: 978-3-662-54177-7