1

Vorbereitungen

Im Rahmen dieses Buches verstehen wir unter einem *Programm* eine zusammengefasste Folge von Anweisungen, die ein Computer zu einem bestimmten Zweck ausführen soll.

Über die vom Programm zu bewältigende Aufgabe und die dazu notwendigen Anweisungen sollte, ja muss man sich vor der eigentlichen *Implementierung* des Programms im Klaren sein. Dies führt dazu, dass spätestens bei der rechnergestützten Lösung umfangreicherer Probleme einiges an Vorbereitungen notwendig wird. Wie man zur Aufgabenstellung gelangt und was unter einer klar formulierten Vorgehensweise zu verstehen ist, bildet den Ausgangspunkt für unsere Betrachtungen in diesem Kapitel. Anschließend beschäftigen wir uns mit der Frage, nach welchen Kriterien man verschiedene Lösungsstrategien für eine bestimmte Aufgabe objektiv bewerten und einordnen kann. Dies führt auf die Begriffe Komplexität und Stabilität.

Insbesondere bei der Entwicklung von Programmen zur Behandlung von mathematischen Aufgabenstellungen spielt es eine große Rolle, wie stark die Lösung von den Parametern abhängt, die das Problem bestimmen und welchen Einfluss die Rechnerarithmetik auf die Lösung hat. Um nicht zu viel an mathematischen Vorkenntnissen zu benötigen, werden wir diese Aspekte meist in Form von einfachen Beispielen behandeln. Die Techniken zur systematischen Untersuchung werden in einführenden Vorlesungen zur numerischen Mathematik sowie der zugehörigen Literatur vermittelt (siehe etwa [2], [13]).

1.1 Modellierung und Algorithmen

Modellierung

Bei der Planung und Konstruktion von Flugzeugen und Schiffen hat man schon immer auf das Experimentieren mit maßstabsgetreuen *Modellen* zurückgegriffen, um bereits vor dem Bau von Prototypen möglichst viele Konstruktionsfehler auszumerzen und damit vermeidbare Entwicklungskosten einzuspa-

ren. Mit der zunehmenden Leistungsfähigkeit von Rechnern hat sich die Möglichkeit eröffnet, solche Tests zu einem erheblichen Teil in Form von *Computersimulationen* durchzuführen, was zu einer weiteren Kostenreduktion führt. Voraussetzung hierfür ist natürlich, dass man die wesentlichen physikalisch-technischen Aspekte wie z.B. Aerodynamik und Materialeigenschaften möglichst realistisch in den Computer überträgt. Es ist also naheliegend, den Modellbegriff entsprechend zu verallgemeinern:

> *Unter einem* Modell *versteht man eine abstrahierte, klar formulierte Darstellung eines Teils der Wirklichkeit.*

Um ein konkret gegebenes Problem überhaupt überblicken und lösen zu können, ist die *Modellierung* sehr oft mit einer Vereinfachung verbunden. Als Folge weicht das Modell in manchen Aspekten von den tatsächlichen Gegebenheiten ab und es treten *Modellierungsfehler* auf. Die Kunst bei der Bildung eines guten Modells besteht also darin,

- die für das interessierende Phänomen weniger relevanten Aspekte zu erkennen und im Modell nicht zu berücksichtigen,
- und die wesentlichen Aspekte des Phänomens im Modell möglichst einfach und korrekt herauszuarbeiten.

Aus der Klarheit der Formulierung des so gebildeten Modells folgt, dass die interessierende Frage, das *Problem*, ebenfalls klar formulierbar ist. In Abb. 1.1 ist der Weg von der „Realität“ über das Modell hin zur konkreten Aufgabenstellung illustriert.

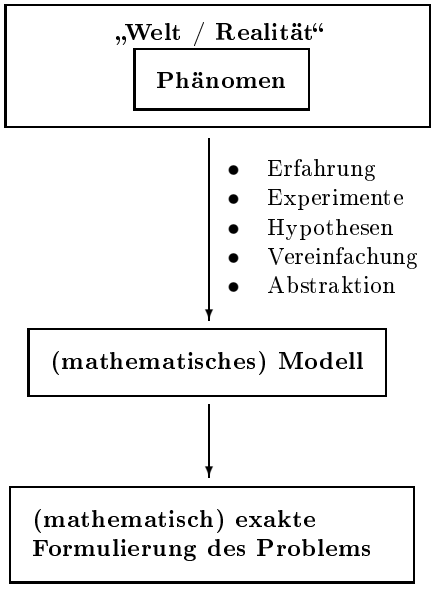

Abb. 1.1. Modellierung: Von der Frage zur Aufgabenstellung

Durch die Abstraktion erreicht man, dass Modelle, die sich in einem bestimmten Anwendungsbereich bereits gut bewährt haben, mit vergleichsweise geringen Modifikationen auf andere Anwendungen mit ähnlicher Struktur übertragen werden können. Um die Übertragbarkeit zu vereinfachen und die Klarheit in der Formulierung zu gewährleisten, gibt es so genannte Modellierungssprachen. Für die Darstellung von Geschäftsprozessen in einem Unternehmen ist z.B. die *Unified Modeling Language (UML)* weit verbreitet. UML ist noch sehr jung, verglichen mit einer anderen Modellierungssprache, die sich bereits seit Jahrtausenden in unzähligen Anwendungsbereichen bewährt hat und immer noch intensiv weiterentwickelt wird: Gemeint ist die Mathematik. Die beiden folgenden Beispiele illustrieren, wie man bei der Modellierung von bestimmten Phänomenen zu universell verwendbaren mathematischen Objekten gelangt, die uns im Verlauf dieses Buchs immer wieder begegnen werden:

Beispiel 1.1 (Bakterien und gewöhnliche Differentialgleichungen). Zum Zeitpunkt t_0 besteht eine Zellkultur im Labor aus N_0 Bakterien. In jedem Zeitintervall $[t, t+\Delta t]$ ($t \geq t_0$, $\Delta t > 0$) vermehrt sich ein Teil der Zellen, während andere Zellen absterben. Wir wollen die recht vernünftige Annahme treffen, dass für hinreichend kurze Zeitintervalle sowohl Zuwachs als auch Abnahme der Anzahl an Zellen in der Kultur proportional zu der aktuellen Anzahl $N(t)$ und der Länge Δt des Zeitintervalls sind. Als mathematische Formel liest sich das folgendermaßen:

$$\Delta N(t) = N(t+\Delta t) - N(t) = \Delta t \left(\lambda_+ - \lambda_-\right) N(t)\,. \tag{1.1}$$

Dabei ist λ_+ die Zuwachs- und λ_- die Sterberate in der Zellkultur. Zur weiteren Abkürzung setzen wir

$$\lambda = \lambda_+ - \lambda_-\,.$$

Wir kümmern uns bei der Gleichung (1.1) nicht weiter darum, dass die Größe N eigentlich ganzzahlig sein müsste, sondern nehmen sogar an, dass N eine differenzierbare reellwertige Funktion der Variablen t ist. Dividieren wir in (1.1) auf beiden Seiten durch Δt und betrachten den Grenzübergang $\Delta t \to 0$, so erhalten wir

$$N'(t) = \lambda N(t)\,, \tag{1.2}$$

wobei $N'(t)$ die Ableitung der Funktion N im Punkt t bezeichnet. In Form dieser *Differentialgleichung* steht uns nun ein mathematisches Modell für die zeitliche Entwicklung der Populationsgröße N zur Verfügung. Die so genannte *Anfangsbedingung*

$$N(t_0) = N_0 \tag{1.3}$$

ergänzt die Differentialgleichung (1.2) zum *Anfangswertproblem*. Die durch unser mathematisches Modell klar formulierte Aufgabe lautet, das Anfangswertproblem zu lösen, d.h. eine differenzierbare Funktion N zu finden, die sowohl die Differentialgleichung (1.2) als auch die Anfangsbedingung (1.3)

erfüllt. Durch Differenzieren überzeugt man sich sofort davon, dass die Exponentialfunktion

$$N(t) = N_0 \, \mathrm{e}^{\,\lambda(t-t_0)} \tag{1.4}$$

eine Lösung ist. Mit elementaren Methoden kann man sogar nachweisen, dass es sich dabei um die einzige Lösung handelt (Aufgabe 1.1).

Die gefundene Lösung (1.4) ist grob gesehen schon recht vernünftig, denn für $\lambda > 0$ (d.h. $\lambda_+ > \lambda_-$) wächst die Population und für $\lambda < 0$ (d.h. $\lambda_+ < \lambda_-$) schrumpft sie. Aber bereits die Tatsache, dass nach (1.4) die Zahl der Zellen für $\lambda > 0$ mit der Zeit jede beliebige Schranke übersteigt, deutet bereits auf einen Modellierungsfehler hin, denn es können ja auch nach noch so langer Zeit t nicht beliebig viele Zellen auf begrenztem Raum existieren. Der Grund für diesen Fehler ist, dass Wachstums- und Sterberate als konstant angenommen wurden. Ein realistischeres Modell müsste berücksichtigen, dass

- λ_+ und λ_- von äußeren, zeitabhängigen Einflüssen abhängen (z.B. Umgebungstemperatur, Lichtverhältnisse), wir haben es also eigentlich mit zeitabhängigen Raten $\lambda_+(t)$ und $\lambda_-(t)$ zu tun, die im Allgemeinen unterschiedlich auf diese Einflüsse reagieren;
- λ_+ und λ_- auch von $N(t)$ selbst abhängen, denn eine große Zellenanzahl bedeutet ja unter anderem, dass z.B. Raum- und Nährstoffangebot knapper werden. Daher ist anzunehmen, dass mit größer werdendem N die Wachstumsrate λ_+ abnimmt und λ_- anwächst.

Insgesamt muss man also davon ausgehen, dass man statt einer Konstanten λ eher eine Funktion

$$\lambda(t, N(t)) = \lambda_+(t, N(t)) - \lambda_-(t, N(t))$$

betrachten muss, die eine komplizierte Gestalt haben kann. Die Differentialgleichung (1.2) wird hiermit zu

$$N'(t) = \lambda\big(t, N(t)\big)\, N(t)\,. \tag{1.5}$$

An diesen Überlegungen sieht man, dass das Studium allgemeiner Anfangswertprobleme der Form

$$y'(t) = f(t, y(t)) \quad , \quad y(t_0) = y_0\,, \tag{1.6}$$

mit $t_0, y_0 \in \mathbb{R}$ lohnenswert ist. Im Beispiel der Bakterienkultur ist N die gesuchte Funktion, so dass f dort die spezielle Form

$$f(t, N(t)) = \lambda(t, N(t))\, N(t)$$

besitzt. Durch die Einführung einer Funktion f, die „irgendwie“ von t und y abhängt, gewinnt man wesentlich an Flexibilität bei der mathematischen Modellierung zeitabhängiger Phänomene (siehe etwa [5]). Die Beantwortung der Fragen nach Existenz und Eindeutigkeit einer Lösung des Anfangswertproblems (1.6) gestaltet sich vergleichsweise einfach (siehe [16]), die konkrete

Berechnung von Lösungen ist allerdings nur sehr selten wie in (1.4) durch scharfes Hinsehen möglich. Wir kommen im nächsten Unterabschnitt darauf zurück. □

Beispiel 1.2 (Funknetzwerke, Vektoren und Matrizen).
Bei einem lokalen Funknetzwerk (WLAN), wie man es z.B. in Flughäfen und Universitäten findet, werden mehrere so genannte Zugangspunkte (*access points*) zum Senden und Empfangen von Datenpaketen an n bestimmten Positionen installiert. Diese Positionen beschreiben wir jeweils als Punkte in der Ebene mit den kartesischen Koordinaten

$$x^{(j)} = (x_1^{(j)}, x_2^{(j)}) \quad \text{für } j = 1, \dots n\,.$$

Um festzustellen, ob damit ein flächendeckender Zugang zum Netzwerk gewährleistet ist, soll an m stichprobenartig ausgewählten Messpositionen mit Koordinaten

$$y^{(i)} = (y_1^{(i)}, y_2^{(i)}) \quad \text{für } i = 1, \dots m\,,$$

die von den *access points* jeweils empfangene Signalstärke r_i gemessen werden.

Wir wollen ein mathematisches Modell aufstellen, das diesen Vorgang theoretisch beschreibt. Dabei hilft uns die Physik weiter:

1. Die Signalstärke nimmt umgekehrt proportional zum Quadrat des Abstands zwischen Sender und Empfänger ab. Ist p_j die Signalstärke des Senders an der Position x_j, so kommt das Signal dieses *access points* an der Stelle y_i mit der Intensität

$$R_{ij} = \frac{C}{\|y_i - x_j\|^2}\, p_j \quad \text{für } i = 1, \dots, m\,,\ j = 1, \dots, n\,, \tag{1.7}$$

   an. In dieser Gleichung steht $\|\cdot\|$ für den *euklidischen Abstand* zweier Punkte in der Ebene:

$$\|y - x\| = \sqrt{(y_1 - x_1)^2 + (y_2 - x_2)^2}\,, \quad x = (x_1, x_2)\,,\ y = (y_1, y_2).$$

   Hinter der Proportionalitätskonstanten $C > 0$ verbergen sich alle Einflüsse auf die Signalübertragung, die mit den örtlichen Gegebenheiten zusammenhängen (Lage und Beschaffenheit von Trennwänden usw.). Man beachte, dass wir für alle möglichen Paarungen von Sendestationen und Messpunkten die gleiche Konstante zu Grunde legen. Wir nehmen also zur Vereinfachung bei der Modellierung an, dass die äußeren Einflüsse auf die Signalübertragung für alle Richtungen gleich sind.
2. Die Sender überlagern sich, ohne sich gegenseitig zu beeinträchtigen. D.h. die Gesamtstärke des Empfangs am Ort y_i ergibt sich als Summe der Beiträge aller Sender:

$$r_i = R_{i1} + R_{i2} + \dots + R_{in} = \sum_{j=1}^{n} R_{ij} \tag{1.8}$$

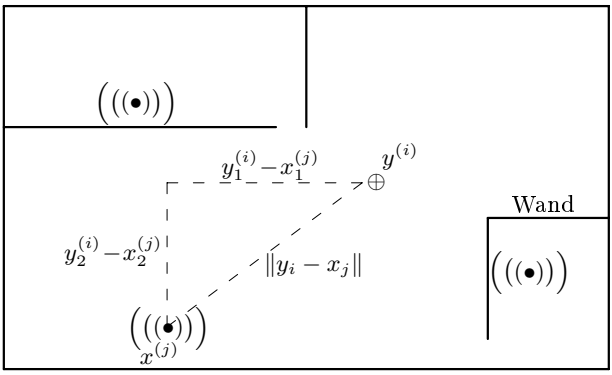

Abb. 1.2. Zur Modellierung des Funknetzwerks. Die Positionen der *access points* sind durch $\Big(\big((\bullet)\big)\Big)$ markiert, der Messpunkt durch $\oplus$.

Beachten wir, dass die Gleichung (1.8) für alle $i = 1, \ldots, m$ gilt und setzen wir in Anlehnung an (1.7)

$$a_{ij} = \frac{C}{\|y_i - x_j\|^2}\,, \qquad \text{für } i = 1, \ldots, m\,,\ j = 1, \ldots, n\,,$$

so ist wegen (1.7) $R_{ij} = a_{ij}\, p_j$ und wir erhalten:

$$r_i = a_{i1}\, p_1 + a_{i2}\, p_2 + \cdots + a_{in}\, p_n = \sum_{j=1}^{n} a_{ij}\, p_j\,, \quad i = 1, \ldots, m\,. \tag{1.9}$$

Das aber ist nichts anderes als das Produkt der *Matrix*

$$A = \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & & a_{2n} \\ \vdots & \cdots & \cdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} = \begin{pmatrix} \dfrac{C}{\|y_1 - x_1\|^2} & \cdots & \dfrac{C}{\|y_1 - x_n\|^2} \\ \vdots & \cdots & \vdots \\ \dfrac{C}{\|y_m - x_1\|^2} & \cdots & \dfrac{C}{\|y_m - x_n\|^2} \end{pmatrix}$$

mit dem (Spalten-) *Vektor*

$$p = \begin{pmatrix} p_1 \\ p_2 \\ \vdots \\ p_n \end{pmatrix}\,,$$

(siehe z.B. [7]). Fassen wir die an den Punkten y_i gemessenen Signalstärken r_i zum Spaltenvektor

$$r = \begin{pmatrix} r_1 \\ r_2 \\ \vdots \\ r_m \end{pmatrix},$$

zusammen, so lässt sich die Gleichung (1.9) kurz und elegant schreiben als

$$Ap = r\,, \quad p \in \mathbb{R}^n\,,\ r \in \mathbb{R}^m\,,\ A \in \mathbb{R}^{m\times n}\,, \tag{1.10}$$

wobei $\mathbb{R}^m$ (bzw. $\mathbb{R}^n$) die Menge aller Spaltenvektoren mit m (bzw. n) reellen *Komponenten* bezeichnet und $\mathbb{R}^{m\times n}$ für die Menge aller reellen Matrizen mit m Zeilen und n Spalten steht.

Durch das *Matrix-Vektor-Produkt* (1.10) kann man also die an den Positionen y_i ankommende Signalstärke r berechnen, wenn C und $p \in \mathbb{R}^n$ bekannt sind. Umgekehrt kann man auch einen gewünschten Empfang $r \in \mathbb{R}^m$ vorgeben und nach der dazu nötigen Sendeleistung p fragen. In diesem Fall fasst man (1.10) als *lineares Gleichungssystem* mit Systemmatrix A und rechter Seite r auf. Eine andere Fragestellung könnte lauten: Wie groß müssen die Werte der Komponenten p_i mindestens sein, damit die Komponenten r_i eine gewisse Schranke nicht unterschreiten? Das bedeutet, dass man ein funktionstüchtiges Netzwerk mit einer minimalen Funkwellenbelastung bereitstellen will und man muss eine *Optimierungsaufgabe* lösen.

Gleichungen der Art (1.10) treten immer dann auf, wenn ein linearer Zusammenhang zwischen bekannten und gesuchten Größen besteht und das ist in vielen Anwendungsproblemen der Fall. Wir werden uns daher in diesem Buch immer wieder mit der datentechnischen Handhabung von Matrizen und Vektoren beschäftigen. Für die Beantwortung der Frage, wie man die oben genannten Aufgaben löst, verweisen wir auf die entsprechenden Vorlesungen der linearen Algebra, der Numerik und der Optimierung. □

Algorithmen

Ist nach abgeschlossener Modellierung die Aufgabe klar umrissen, so kann man sich an die Entwicklung und Formulierung einer Lösungsstrategie machen.

> *Ein* Algorithmus *ist eine eindeutig formulierte Vorschrift zur Lösung einer Aufgabe bzw. eines Aufgabentyps.*

Die Eindeutigkeit der Formulierung soll nicht nur die korrekte Anwendung der Lösungsmethode sicher stellen, sondern ermöglicht auch die Untersuchung der *Korrektheit* des Algorithmus bezüglich der gestellten Aufgabe, d.h. ob die Vorgehensweise auch wirklich die Lösung des Problems liefert. Außerdem ist die exakte Darstellung der Verfahrensschritte unabdingbar, wenn man verschiedene korrekte Algorithmen zur Lösung ein und derselben Aufgabe vergleichen und beurteilen will.

Mit anderen Worten: Wie ein Kochrezept beschreibt ein Algorithmus, *was* man *wie womit* tun soll, um die Aufgabe zu lösen. Das beinhaltet speziell, dass die einzelnen Anweisungen auch durchführbar sein müssen. Das kann z.B. dadurch gewährleistet sein, dass die vorzunehmenden Arbeitsschritte ganz elementarer Natur sind und nicht weiter erläutert werden müssen. Dazu ein einfaches Beispiel:

Beispiel 1.3 (Vertauschen von Koordinaten).

1. Lies die Koordinaten (x, y) ein.
2. Setze
$$h := y \quad , \quad y := x \quad , \quad x := h\,.$$
3. Liefere die neuen Koordinaten (x, y) zurück.

Aus geometrischer Sicht beschreibt der Algorithmus, wie man einen Punkt (x, y) in der Ebene an der ersten Winkelhalbierenden spiegelt. □

Dabei verwenden wir in Algorithmen die Bezeichnung := für eine Wertzuweisung, um sie von der Gleichheit im mathematischen Sinn zu unterscheiden. Bereits an dem einfachen Beispiel 1.3 kann man erkennen, welche Informationen ein Algorithmus im Einzelnen beinhaltet:

- Die *Eingabedaten*, auch *Parameter* genannt, die zur Durchführung des Algorithmus benötigt werden (im Beispiel die Zahlen x und y).
- Die *Ausgabedaten*, d.h. das vom Algorithmus gelieferte Ergebnis (im Beispiel die getauschten Koordinaten (x, y)).
- Die Beschreibung der Schritte, die durchgeführt werden sollen, um das richtige Resultat zu erhalten (im Beispiel die Zuweisungen der x- und y-Koordinaten).
- Die für die korrekte Durchführung nötigen *Hilfsgrößen*, die weder zu den Eingabe- noch zu den Ausgabedaten zählen. In Beispiel 1.3 ist h eine solche Hilfsgröße und man macht sich leicht klar, dass auf diese Hilfsgröße nicht verzichtet werden kann.

Die Durchführbarkeit eines Algorithmus kann auch dadurch gewährleistet sein, dass neben elementaren Arbeitsschritten auch bereits existierende Algorithmen zum Einsatz kommen, deren Korrektheit bekannt ist. Diese *Unterprogramme* müssen natürlich mit den geeigneten Eingabedaten versorgt werden:

Beispiel 1.4 (Aufsteigendes Sortieren eines Zahlenpaares).

1. Lies die Zahlen x und y ein.
2. Falls $x \leq y$: liefere (x, y) zurück,
andernfalls: führe den Algorithmus aus Beispiel 1.3 für (x, y) durch.

□

Durch den Einsatz bereits bestehender Algorithmen bei der Entwicklung von neuen Verfahren bietet sich die Möglichkeit der *Partitionierung* eines Problems. Man unterteilt die Aufgabenstellung in Teilprobleme und analysiert, für welche der Teilaufgaben bereits Lösungsmethoden existieren. Dadurch kann man sich voll und ganz auf die Entwicklung neuer Verfahren für die noch nicht behandelten Problembestandteile konzentrieren und spart Zeit. Zum Schluss werden alle Unterprogramme zu einer Lösungsmethode der ursprünglichen Aufgabe zusammengefasst.

Algorithmen lassen sich zunächst dem Inhalt nach, d.h. nach ihrem Aufgabengebiet einteilen: So gibt es z.B. Sortieralgorithmen, Suchalgorithmen, oder Lösungsalgorithmen für Gleichungen der unterschiedlichsten Art. Eine andere Art der Unterscheidung wird hinsichtlich der Eigenschaften eines Algorithmus bei der Durchführung vorgenommen: Da wir uns in diesem Buch mit der Programmierung auseinander setzen, beschränken wir uns auf die Betrachtung von *statisch finiten Algorithmen*, d.h. solchen Handlungsvorschriften, die aus endlich vielen Schritten bestehen. Oft enthalten Algorithmen die Anweisung, bestimmte Schritte so oft zu wiederholen, bis bestimmte Bedingungen erfüllt sind. Dann stellt sich die Frage, ob der Algorithmus *terminierend* ist, d.h. ob seine Durchführung *stets* nach endlicher Zeit abgeschlossen ist. Auf diese Eigenschaft legen wir sicher großen Wert, wenn wir die Lösung einer Aufgabe berechnen wollen. Dass nicht alle Algorithmen terminierend sind, ist z.B. bei der regelmäßigen automatischen Abfrage von neuer E-Mail erwünscht. Hochgradig unerwünscht dagegen sind unbeabsichtigte *Endlosschleifen* in Berechnungsprogrammen!

Man könnte meinen, dass nach Definition jeder Algorithmus *determiniert* ist, also bei gleichen Eingabedaten stets das gleiche Ergebnis liefert. Die Definition verbietet allerdings nicht, dass ein Algorithmus Zufallselemente enthält, die den konkreten Ablauf der Handlungsanweisungen und damit das Ergebnis verändern können.

Bei den Beispielen 1.3 und 1.4 kann man sich unmittelbar von Korrektheit und Determiniertheit überzeugen. Weniger offensichtlich ist dies für das folgende klassische Verfahren zur Bestimmung des größten gemeinsamen Teilers zweier Zahlen:

Beispiel 1.5 (Euklidischer Algorithmus).

1. Lies $a_0, a_1 \in \mathbb{N}$ ein.
2. Dividiere mit Rest
$$a_0 = q\, a_1 + r$$
und setze
$$a_0 := a_1 \quad , \quad a_1 := r\,,$$
solange, bis $a_1 = 0$.
3. Liefere $\mathrm{ggT}(a_0, a_1) = a_0$ zurück.

Der Nachweis, dass für beliebige natürliche Zahlen $a_0, a_1 \in \mathbb{N}$ nach endlich vielen Divisionen mit Rest der Fall $a_1 = 0$ eintritt und a_0 dann auch tatsächlich der größte gemeinsame Teiler aus den beiden Eingabeparametern ist, findet sich z.B. in [1]. □

Wir wenden uns wieder der Lösung von Anfangswertproblemen der Form (1.6) zu und überlegen uns eine Methode, die weitgehend unabhängig von der konkreten Gestalt der Funktion f Ergebnisse liefert. Als Preis zahlen wir dafür, dass das Ergebnis nur eine Näherung an die exakte Lösung darstellt.

Beispiel 1.6 (Das Euler-Verfahren). Die Idee, die hinter dem *Euler-Verfahren* zur näherungsweisen Lösung des Anfangswertproblems

$$y'(t) = f(t, y(t)), \quad y(t_0) = y_0,$$

steckt, ist die folgende: Wir nähern in der Differentialgleichung die Ableitung $y'(t_0)$ durch den Differenzenquotienten

$$\frac{y(t_0 + h) - y(t_0)}{h} = \frac{y(t_0 + h) - y_0}{h}$$

an, wobei $h > 0$ eine von uns gewählte *Schrittweite* ist. Da wir nach einer differenzierbaren Lösung y suchen, wird diese Ersetzung für genügend klein gewähltes h keinen allzu großen Fehler verursachen. Wir erhalten formal

$$y(t_0 + h) = y_0 + h\, f(t_0, y_0),$$

wobei die rechte Seite dieser Gleichung nur bekannte Größen enthält und sich ohne Weiteres berechnen lässt. Weil wir aber die Ableitung durch den Differenzenquotienten ersetzt haben, steht auf der linken Seite eben nicht der exakte Wert $y(t_0 + h)$ der gesuchten Lösung, sondern nur ein Näherungswert, den wir y_1 nennen wollen. Setzen wir weiter

$$t_1 = t_0 + h,$$

dann können wir die Vorgehensweise mit (t_1, y_1) als neuem Startpunkt wiederholen und erhalten

$$y_2 = y_1 + h\, f(t_1, y_1)$$

als Näherungswert für $y(t_2)$ mit $t_2 = t_1 + h = t_0 + 2h$. Abbildung 1.3 illustriert diese Vorgehensweise und es wird ersichtlich, warum die Methode auch *Eulersches Polygonzugverfahren* heißt.

Die Schrittweite muss nicht für alle Näherungen gleich sein. Möchte man eine Näherung an der Stelle $t > t_0$ berechnen, so kann man durch Wahl einer Zerlegung

$$t_0 < t_1 < \cdots < t_{n-1} < t_n = t,$$

jeweils als Schrittweite

$$h_i = t_i - t_{i-1} \text{ für } i = 1, \ldots, n.$$

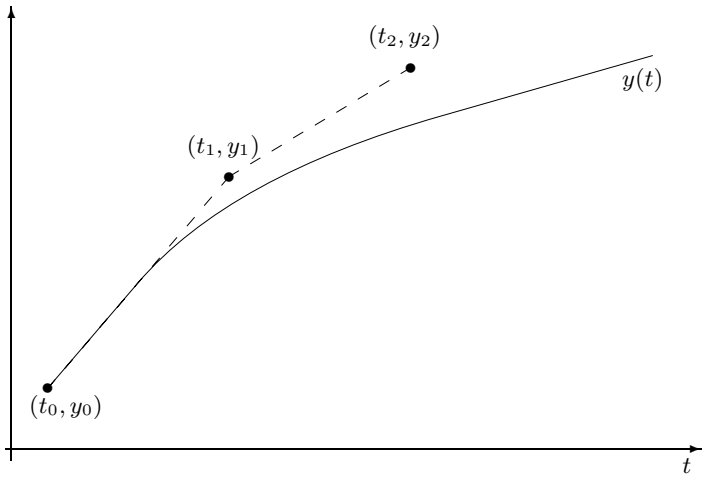

Abb. 1.3. Das Euler-Verfahren zur Approximation der exakten Lösung y.

vorgeben. Das Euler-Verfahren zur Berechnung der Näherungen $y_1, \ldots, y_n$ lautet dann:

$$y_{i+1} = y_i + h_{i+1}\, f(t_i, y_i) \text{ für } i = 0, \ldots, n-1\,. \tag{1.11}$$

□

1.2 Komplexität und $\mathcal{O}$-Notation

Bei der Beurteilung der Qualität eines Algorithmus sind folgende Aspekte zu berücksichtigen:

Laufzeitkomplexität: Sie gibt an, wieviele Arbeitsschritte der Algorithmus zur Bewältigung der Aufgabe benötigt. Da ein Computerprozessor nur eine bestimmte Anzahl von Operationen pro Zeiteinheit ausführen kann, wird man versuchen, die Anzahl der zur Lösung erforderlichen Arbeitsschritte zu minimieren.

Speicherkomplexität: Der Computerprozessor greift über einen schnellen Hilfsspeicher (*Cache*) auf die im Hauptspeicher gelegenen Daten zu. Wegen der begrenzten Speicherkapazität kommen für die Praxis nur solche Algorithmen in Frage, die zu jedem Zeitpunkt der Ausführung mit einer beschränkten Datenmenge arbeiten. Der ökonomische Umgang mit Speicherressourcen ist ein weiteres Ziel bei der Entwicklung von Algorithmen.

Qualität der Ergebnisse: Speziell bei Approximationsalgorithmen wie dem Euler-Verfahren in Beispiel 1.6 ist zu untersuchen, wie nahe die berechneten Näherungswerte dem exakten Ergebnis kommen.

Die jeweilige Gewichtung der einzelnen Kriterien richtet sich nach den Erfordernissen der konkreten Problemstellung. Rechenzeit und Speicher kosten letztlich Geld und so bezeichnet man Algorithmen mit langer Laufzeit bzw. großem Speicherbedarf auch als *teuer*. Natürlich will man über die Komplexität eines Algorithmus gern im Bilde sein, *bevor* man ihn implementiert und sich im schlimmsten Fall über unerträglich lange Laufzeiten ärgert. Zu diesem Zweck analysiert man die benötigten Operationen und den Speicherbedarf mit mathematischen Methoden.

Wenn man verschiedene Algorithmen zur Lösung ein und derselben Aufgabe objektiv miteinander vergleichen will, dann müssen Eigenschaften der Algorithmen wie Laufzeit oder Speicherbedarf in Beziehung gesetzt werden zu den Größen, die das Problem charakterisieren. Dazu ein klassisches Beispiel:

Beispiel 1.7 (Polynomauswertung).
Für die Auswertung eines Polynoms vom Grad $n \in \mathbb{N}$

$$P(x) = \sum_{k=0}^{n} a_k \, x^k \,,$$

an einer Stelle $x_0 \in \mathbb{R}$ kann man folgende „naive“ Methode verwenden:

1. Setze $p_0 := a_0$.
2. Für $k = 1, 2, \ldots, n$:
   - setze Hilfsgröße $M := a_k$,
   - Für $m = 1, \ldots, k$ berechne $M := M \cdot x_0$.
   - $p_0 := p_0 + M$.
3. Liefere p_0 zurück.

In jedem der n Schritte werden jeweils eine Addition und k Multiplikationen durchgeführt. Da wir den Zeitaufwand für die Zuweisungen gegenüber den arithmetischen Operationen vernachlässigen können, ist die Gesamtanzahl der Operationen

$$Op(n) = \sum_{k=1}^{n}(k+1) = n + \sum_{k=1}^{n} k = n + \frac{1}{2}n(n+1) = \frac{1}{2}n^2 + \frac{3}{2}n \,.$$

Mit ein wenig Vorarbeit können wir die Anzahl der Operationen reduzieren: Schreiben wir das Polynom in Form ineinander geschachtelter Linearfaktoren,

$$P(x) = \Big(\cdots \Big(\big((a_n \, x + a_{n-1})x + a_{n-2} \big) x + a_{n-3} \Big) \cdots \Big) x + a_0 \,,$$

so bietet sich folgende Methode an, die auch *Horner-Schema* genannt wird:

1. Setze Hilfsgröße $p_0 := a_n$.
2. Für $k = n-1, \ldots, 0$ berechne

$$p_0 := p_0 \cdot x_0 + a_k \,.$$

3. Liefere p_0 zurück.

Diese Vorgehensweise ist nicht nur kürzer in der Formulierung: In jedem der n Schritte werden lediglich eine Addition und nur eine Multiplikation vorgenommen, so dass für diese Methode $Op(n) = 2n$ folgt.

Sprachlich drückt man diesen Sachverhalt dadurch aus, dass der erste Algorithmus von *quadratischer* und der zweite von *linearer* Komplexität ist. Der Speicherbedarf beider Algorithmen ist vergleichbar: Beide benötigen die n Koeffizienten des Polynoms und verwenden p_0, zusätzlich benötigt die naive Methode die Hilfsgröße M. Daraus ergibt sich für beide Methoden eine lineare Speicherkomplexität. □

Beispiel 1.8 (Matrizen und Vektoren).

a) Um einen Vektor $x \in \mathbb{R}^n$ abzuspeichern, benötigt man für die n Komponenten natürlich n Speicherplätze.
b) Für eine Matrix $A \in \mathbb{R}^{m \times n}$ hat man entsprechend $m\,n$ Einträge abzuspeichern. Im speziellen Fall einer *quadratischen Matrix* $A \in \mathbb{R}^{n \times n}$ benötigt man n^2 Speicherplätze, es liegt also quadratische Speicherkomplexität vor.
c) Das *euklidische Skalarprodukt* zweier Vektoren $x, y \in \mathbb{R}^n$ ist definiert als

$$\langle x, y \rangle = \sum_{i=1}^{n} x_i \, y_i \,,$$

d.h. die Berechnung erfordert n Multiplikationen und $n-1$ Additionen.
d) Aus Beispiel 1.2 wissen wir, dass das Produkt einer Matrix $A \in \mathbb{R}^{m \times n}$ und einem Vektor $x \in \mathbb{R}^n$ wieder ein Vektor ist, den wir mit $y \in \mathbb{R}^m$ bezeichnen. Die Komponenten von y sind definiert durch

$$y_i = \sum_{j=1}^{n} a_{ij} \, x_j \,, \quad i = 1, \ldots, m \,,$$

wobei a_{ij} die Matrixeinträge sind und x_j die j-te Komponente des Vektors x ist. Wir haben also für jede der m Komponenten n Multiplikationen und $n-1$ Additionen zu berechnen, d.h. der Gesamtaufwand des Matrix-Vektor-Produkts $y = Ax$ beträgt

$$Op(m, n) = m(2n-1) \,.$$

□

Die $\mathcal{O}$-Notation

Wir konnten im Beispiel 1.7 die exakte Anzahl der benötigten arithmetischen Operationen leicht berechnen, da beide Algorithmen sehr einfacher Natur sind. Für kompliziertere und umfangreicher formulierte Algorithmen wird die exakte Berechnung der Laufzeitkomplexität sehr mühselig und ist auch aus einem weiteren Grund eine eher undankbare Aufgabe: Der Unterschied in der Laufzeit der beiden Verfahren zur Polynomauswertung wird mit wachsendem Polynomgrad n erst richtig spürbar, da dann der quadratische Summand in der Komplexität der naiven Methode dominiert und man den linearen Anteil vernachlässigen kann.

Für einen Vergleich der Komplexität von Algorithmen genügt also oft bereits die Kenntnis der *Größenordnungen* von Laufzeit- und Speicherkomplexität. Der folgende Begriff ist eine mathematische Präzisierung hiervon:

Definition 1.9 ($\mathcal{O}$-Notation, LANDAU-Symbole).
Es sei $I \subseteq \mathbb{R}$ ein Intervall und

$$f, g : I \longrightarrow \mathbb{R}$$

seien zwei Funktionen.

a) Sei $x_0 \in \mathbb{R}$. Die Funktion f heißt *von der Ordnung* $\mathcal{O}(g(x))$ *für* $x \to x_0$, wenn es eine Konstante $C > 0$ und ein $\delta > 0$ gibt, so dass die folgende Ungleichung gilt:

$$|f(x)| \leq C\,|g(x)| \quad \text{für alle } x \in I \text{ mit } |x - x_0| < \delta\,. \tag{1.12}$$

Die Funktion f heißt *von der Ordnung* $\mathcal{O}(g(x))$ *für* $x \to \infty$ (bzw. $x \to -\infty$), wenn es eine Konstante $C > 0$ und ein $M \in \mathbb{R}$ gibt, so dass die folgende Ungleichung gilt:

$$|f(x)| \leq C\,|g(x)| \quad \text{für alle } x \in I \text{ mit } x \geq M \text{ (bzw. } x \leq M)\,. \tag{1.13}$$

b) Sei $x_0 \in \mathbb{R}$. Die Funktion f heißt *von der Ordnung* $o(g(x))$ *für* $x \to x_0$, wenn gilt:

$$\lim_{x \to x_0} \frac{|f(x)|}{|g(x)|} = 0\,.$$

Entsprechend heißt die Funktion f von der Ordnung $o(g(x))$ für $x \to \infty$ bzw. $x \to -\infty$, wenn gilt:

$$\lim_{x \to \infty} \frac{|f(x)|}{|g(x)|} = 0 \quad \text{bzw.} \quad \lim_{x \to -\infty} \frac{|f(x)|}{|g(x)|} = 0\,.$$

Bemerkung 1.10. Man beachte, dass in der Definition nicht gefordert wird, dass $x_0 \in I$ gilt, so dass die Funktionen f und g an dieser Stelle gar nicht definiert sein müssen. Es genügt, dass man der Stelle $x_0 \in \mathbb{R}$ mit Punkten aus dem Intervall I beliebig nahe kommen kann. Möchte man daher eine Aussage über das Verhalten einer Funktion für $x \to \pm\infty$ treffen, so muss I natürlich unbeschränkt sein.

Beispiel 1.11 ($\mathcal{O}$-Notation).

a) Für die naive Polynomauswertung in Beispiel 1.7 ist

$$Op(n) = \mathcal{O}(n^2)\,,$$

und für das Horner-Schema gilt

$$Op(n) = \mathcal{O}(n)\,.$$

Der Speicherbedarf beider Algorithmen ist

$$Mem(n) = \mathcal{O}(n)\,.$$

Wir folgen hier der Konvention, dass man bei der $\mathcal{O}$-Notation für Größen wie die Anzahl der benötigten Operationen bzw. den Speicherbedarf stets zu Grunde legt, dass man das Verhalten für $n \to \infty$ beschreibt.

b) Ein Polynom vom Grad n,

$$p(x) = \sum_{k=0}^{n} a_k x^k$$

ist für $x \to \infty$ von der Ordnung $\mathcal{O}(x^n)$ bzw. für jedes $\epsilon > 0$ von der Ordnung $o(x^{n+\epsilon})$. Das Gleiche gilt offensichtlich auch für $x \to -\infty$.

c) Für beliebige $n \in \mathbb{N}$ gilt

$$x^n = o(\mathrm{e}^x) \quad \text{für } x \to \infty\,.$$

Man sagt hierzu auch: „Die Exponentialfunktion wächst echt schneller als jede Potenz von x.“ □

Natürlich ist stets $f(x) = \mathcal{O}(f(x))$ und aus der Definition liest man sofort die Gültigkeit von

$$f(x) = o(g(x)) \;\Rightarrow\; f(x) = \mathcal{O}(g(x))$$

ab. Wir fassen weitere Eigenschaften in dem folgenden Satz zusammen:

Satz 1.12. *Für das Landau-Symbol $\mathcal{O}$ gilt:*

a) Für alle $K \in \mathbb{R} \setminus \{0\}$ gilt

$$f(x) = \mathcal{O}(Kg(x)) \;\Leftrightarrow\; f(x) = \mathcal{O}(g(x))\,.$$

b) $f(x) = \mathcal{O}\big(g(x) + h(x)\big)$ und $h(x) = \mathcal{O}(g(x)) \Rightarrow f(x) = \mathcal{O}(g(x))$.

c) Wenn $f_1(x) = \mathcal{O}(g_1(x))$ und $f_2(x) = \mathcal{O}(g_2(x))$, dann gilt

$$f_1(x)\, f_2(x) = \mathcal{O}(g_1(x)\, g_2(x))\,.$$

Den Beweis überlassen wir als Übung (Aufgabe 1.3).

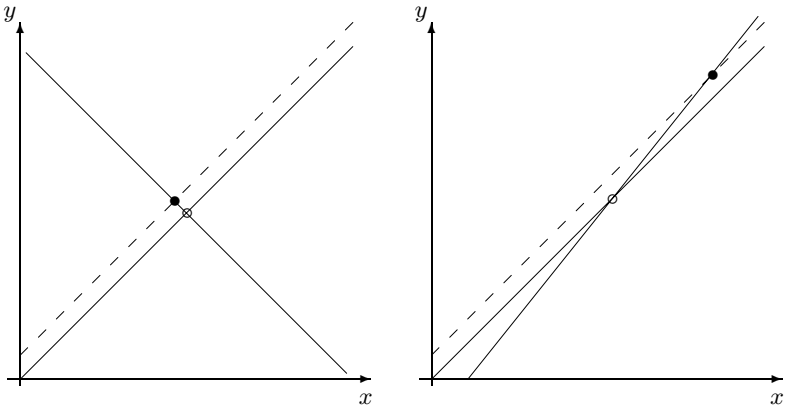

Abb. 1.4. Abweichung des Ergebnisses bei der Schnittpunktbestimmung in Abhängigkeit vom Schnittwinkel.

1.3 Kondition eines Problems

Bereits bei der Entwicklung einer Lösungsstrategie für ein Problem sollte man die gestellte Aufgabe auf ihre „Gutartigkeit" hin untersuchen. Damit ist gemeint, wie sich *Datenfehler* in den Eingabeparametern (z.B. Mess- und Übertragungsfehler) auf die Resultate auswirken.

> *Die* Kondition eines Problems *gibt an, wie stark sich Änderungen an den Eingabedaten auf die Lösung auswirken. Die Kondition ist eine Eigenschaft des Problems und unabhängig von einer konkreten Lösungsmethode.*

Ein einfaches Paradebeispiel hierfür ist die Bestimmung des Schnittpunkts zweier Geraden. Verschiebt man eine der Geraden etwas, so weicht die neue Position des Schnittpunkts umso stärker von der alten ab, je mehr die Steigungen der sich schneidenden Geraden übereinstimmen (siehe Abb. 1.4). Im Falle eines solchen „schleifenden Schnitts" wirken sich kleine Änderungen in den Eingabedaten (Steigung und Achsenabschnitte) erheblich auf das Resultat (Schnittpunktkoordinaten) aus.

Bezeichnen wir mit $x \in \mathbb{R}$ den korrekten Wert einer Größe und mit $\tilde{x}$ einen hiervon abweichenden, so wird

$$e_{abs}(x) = |x - \tilde{x}| \tag{1.14}$$

absoluter Fehler in x und für $x \neq 0$ der Quotient

$$e_{rel}(x) = \frac{|x - \tilde{x}|}{|x|} \tag{1.15}$$

relativer Fehler in x genannt.

Wir nennen ein Problem gut konditioniert, *wenn sich relative Fehler in den Eingangsdaten nur mäßig auf den relativen Fehler im Ergebnis auswirken, andernfalls nennen wir das Problem* schlecht konditioniert. *Können selbst kleinste relative Fehler in den Eingabedaten beliebig große relative Abweichungen im Ergebnis hervorrufen, so heißt das Problem* schlecht gestellt.

Selbst eine so einfache Aufgabe wie die Addition zweier Zahlen $a, b \in \mathbb{R}$ kann mitunter tückisch sein. Beschränkt man sich auf die Betrachtung des absoluten Fehlers, so liefert die Dreiecksungleichung

$$|(a+b) - (\tilde{a}+\tilde{b})| \leq |a-\tilde{a}| + |b-\tilde{b}| \leq 2\max\{|a-\tilde{a}|, |b-\tilde{b}|\}\,, \tag{1.16}$$

d.h. der absolute Fehler im Ergebnis ist höchstens doppelt so groß wie der maximale absolute Fehler in den Summanden. Bevor wir uns aber beruhigt zurücklehnen, werfen wir noch einen Blick auf den relativen Fehler bei der Addition:

Beispiel 1.13 (Konditionsanalyse der Addition). Bei der Analyse der Verstärkung des relativen Fehlers bei der Berechnung von $a+b$ nehmen wir zunächst an, dass $a, b \in \mathbb{R} \setminus \{0\}$. Ist nur der Summand b fehlerbehaftet, so lautet der absolute Fehler im Ergebnis

$$|(a+b) - (a+\tilde{b})| = |b-\tilde{b}|$$

und daraus folgt für die relativen Fehler die Beziehung

$$\frac{|(a+b)-(a+\tilde{b})|}{|a+b|} = \frac{|b|}{|a+b|}\, e_{rel}(b)\,.$$

Analog finden wir für den Fall, dass nur a fehlerbehaftet ist, die Gleichung

$$\frac{|(a+b)-(\tilde{a}+b)|}{|a+b|} = \frac{|a|}{|a+b|}\, e_{rel}(a)\,.$$

Wir können nun den allgemeinen Fall behandeln: Wenn die absoluten Fehler in a und b hinreichend klein sind, liefert die Dreiecksungleichung die folgende recht brauchbare Abschätzung nach oben:

$$\begin{aligned} |(a+b) - (\tilde{a}+\tilde{b})| &= |(a+b) - (a+\tilde{b}) + (a+b) - (\tilde{a}+b)| \\ &\leq |(a+b) - (a+\tilde{b})| + |(a+b) - (\tilde{a}+b)|\,. \end{aligned}$$

Daraus erhalten wir durch Einsetzen der obigen Gleichungen

$$e_{rel}(a+b) = \frac{|(a+b)-(\tilde{a}+\tilde{b})|}{|a+b|} \leq \frac{|b|\, e_{rel}(b) + |a|\, e_{rel}(a)}{|a+b|}\,. \tag{1.17}$$

Mit Blick auf die rechte Seite dieser Fehlerabschätzung stellen wir fest, dass die relativen Fehler in den Summanden extrem verstärkt werden, wenn $|a+b|$ sehr klein ist. Anders ausgedrückt:

Die Addition zweier Zahlen $a, b \in \mathbb{R}$ ist schlecht konditioniert, wenn gilt:

$$a \approx -b\,.$$

Andernfalls handelt es sich um eine gut konditionierte Aufgabe. Wenn die Summanden gleiches Vorzeichen haben, so gilt $|a+b| = |a| + |b|$ und die Datenfehler werden wegen

$$\frac{|a|}{|a+b|} \leq 1\,, \quad \frac{|b|}{|a+b|} \leq 1$$

im Ergebnis sogar gedämpft. □

Auf ähnliche Art und Weise überzeugt man sich davon, dass die Multiplikation zweier Zahlen sowie das Ziehen der Quadratwurzel gut konditionierte Aufgaben sind (Aufgabe 1.4).

1.4 Rechnerarithmetik

Die Datenfehler sind nicht die einzigen Störfaktoren auf dem Weg zum Ergebnis. Die endlichen Ressourcen eines Computers haben zur Folge, dass man nur mit endlich vielen Zahlen arbeiten kann, was sich natürlich auch auf die Art des Rechnens mit ihnen und somit auf die Ergebnisse auswirkt. Um uns der Fallstricke, die hinter dieser Tatsache lauern, bewusst zu werden, müssen wir uns mit der Darstellung von Zahlen und der Rechnerarithmetik etwas genauer befassen.

1.4.1 Zahldarstellung

Sei $B \in \mathbb{N}$ mit $B \geq 2$. Dann existiert zu jeder ganzen Zahl x eine Darstellung der Form

$$x = (-1)^s \sum_{j=0}^{N} x_j\, B^j\,, \tag{1.18}$$

wobei

$$N \in \mathbb{N}_0\,,\ x_j \in \{0, \ldots, B-1\} \text{ für } j = 1, \ldots, N\,,\ s \in \{0, 1\}\,.$$

Für $x \neq 0$ ist diese Darstellung eindeutig, wenn man

$$x_N \neq 0$$

verlangt. Ist die so genannte *Basis* B festgelegt, so genügt die Kenntnis der Ziffern x_j. Die *Zifferndarstellung* der Zahl x zur Basis B lautet

$$x = x_N x_{N-1} \ldots x_{0|B}\,,$$

wobei wir für den Fall der Dezimaldarstellung ($B = 10$) die Angabe der Basis weglassen.

Beispiel 1.14. Die dezimale Zifferndarstellung der Zahl x sei 30. Dann lautet die

- *Binärdarstellung* (Dualzahl, $B = 2$): $x = 11110_{|2}$,
- *Oktaldarstellung* ($B = 8$): $x = 36_{|8}$,
- *hexadezimale Darstellung* ($B = 16$): $x = 1E_{|16}$.
  Dabei stehen zur Darstellung die Ziffern $0, \ldots, 9, A, B, C, D, E, F$ zur Verfügung. □

Dieses Konzept lässt sich auf die rellen Zahlen übertragen, denn es gilt allgemein der folgende Satz (siehe etwa [15]):

Satz 1.15 (B-adische Zahldarstellung). *Sei $B \in \mathbb{N}$, $B \geq 2$. Dann kann jede Zahl $x \in \mathbb{R} \setminus \{0\}$ auf die folgende Art dargestellt werden:*

$$x = (-1)^s \, B^N \sum_{n=0}^{\infty} x_n \, B^{-n} \,. \tag{1.19}$$

Dabei ist $N \in \mathbb{Z}$, $x_n \in \{0, \ldots, B-1\}$ und $s \in \{0, 1\}$.
Die Darstellung ist eindeutig, wenn gilt $x_0 \neq 0$ und wenn zu jedem $m \in \mathbb{N}$ ein $n \geq m$ existiert mit $x_n \neq B - 1$.

Die zweite Bedingung für die Eindeutigkeit trägt der Tatsache Rechnung, dass z.B. $0.99999\cdots = 0.\overline{9}$ und 1 identisch sind.

Beispiel 1.16 (Umwandlung in die Binärdarstellung). Die Binärdarstellung der Dezimalzahl 12.75 lautet $1100.11_{|2}$. Die Umwandlung der „harmlosen" Dezimaldarstellung 0.2 in eine Dualzahl führt allerdings auf die unendliche Darstellung

$$0.0011001100110011_{|2} \cdots = 0.\overline{0011}_{|2} \,.$$

Gibt man nun die Maschinendarstellung von 0.2 aus, so erhält man je nach Genauigkeit und Maschine z.B. `0.20000000000000001`, was fälschlicherweise häufig als Fehler der Programmiersprache oder des Rechners interpretiert wird. □

Darstellungen mit unendlich vielen Stellen wie in (1.19) sind weder auf einem Computer mit seinen begrenzten Ressourcen realisierbar noch werden sie für die Praxis benötigt. Statt dessen arbeitet der Rechner mit einer endlichen Teilmenge, den *normalisierten Gleitpunktzahlen*

$$x = (-1)^s \, B^E \sum_{n=0}^{P-1} x_n \, B^{-n} \,, \tag{1.20}$$

wobei $P \in \mathbb{N}$ fest gewählt ist und der *Exponent* E nur die ganzen Zahlen zwischen zwei vorgegebenen Schranken $E_{\min}$ und $E_{\max}$ durchläuft. Die Normalisierung der Darstellung besteht darin, dass auch in (1.20)

$$x_0 \neq 0$$

gefordert wird. Die Zahl

$$m = \sum_{n=0}^{P-1} x_n B^{-n} \tag{1.21}$$

heißt *Mantisse von* x. Bei festgelegter Basis B schreibt man die Mantisse auch als

$$m = x_0 \,.\, x_1 \ldots x_{P-1} \tag{1.22}$$

und bezeichnet die festgelegte Anzahl P der *signifikanten Stellen* als *Mantissenlänge*. Eine normalisierte Gleitpunktzahl zur Basis B mit Vorzeichen $(-1)^s$, Mantisse m und Exponent E ist also vollständig durch das folgende Tripel beschrieben:

$$x = \big(s, m, E\big)_B \,. \tag{1.23}$$

1.4.2 Rundung und Gleitpunktrechnung

Durch die Mantissenlänge P und die Schranken $E_{\min}$ und $E_{\max}$ ist die Menge $\mathbb{G}$ aller zulässigen normalisierten Gleitpunktzahlen festgelegt. Speziell gilt für alle $x \in \mathbb{G}$:

$$B^{E_{\min}} \leq |x| < B^{E_{\max}+1} \,. \tag{1.24}$$

Bei der Ausführung von Programmen muss der Computer aber häufig Zahlenwerte handhaben, die nicht zu $\mathbb{G}$ gehören. Diese Werte können z.B. extern in Form von Messwerten entstanden sein. Solche Zahlenwerte werden aber auch durch arithmetische Operationen vom Computer selbst erzeugt: An einfachen Beispielen macht man sich leicht klar, dass für $x, y \in \mathbb{G}$ weder die Summe $x+y$ noch das Produkt $x\,y$ wieder in $\mathbb{G}$ liegen müssen. Es ist damit notwendig, reelle Zahlen in die Menge $\mathbb{G}$ der normalisierten Gleitpunktzahlen abzubilden. Diesem Zweck dient die so genannte Rundung.

Rundung und Rundungsfehler. Unter der *Rundung* kann man zunächst eine Abbildung

$$\mathrm{rd} : \mathbb{R} \longrightarrow \mathbb{G}$$

verstehen, die die Eigenschaft

$$\mathrm{rd}(x) = x \text{ für alle } x \in \mathbb{G}$$

besitzt. Diese Abbildung kann auf mehrere Arten realisiert sein. Die gebräuchlichste Rundungsvorschrift basiert auf der normalisierten Darstellung

$$x = (-1)^s \, B^E \, x_0.x_1x_2 \ldots$$

für $x \neq 0$, die ja nach Satz 1.15 existiert und lautet

$$\mathrm{rd}(x) = (-1)^s B^E \begin{cases} x_0.x_1 \ldots x_{P-1} & \text{, falls } x_P < \dfrac{B}{2} \\ (x_0.x_1 \ldots x_{P-1} + B^{-(P-1)}) & \text{, falls } x_P \geq \dfrac{B}{2} \end{cases} . \quad (1.25)$$

Diese Rundungsvorschrift besitzt offensichtlich die Eigenschaft,

$$|x - \mathrm{rd}(x)| \leq |x - y| \text{ für alle } y \in \mathbb{G} ,$$

d.h. die *Rundung von* x ist diejenige normalisierte Gleitpunktzahl, die x am nächsten liegt (*round to nearest*). Die Fallunterscheidung schafft Eindeutigkeit, wenn x genau in der Mitte zwischen zwei Gleitpunktzahlen liegt. Wir setzen dabei für den Moment noch voraus, dass E zwischen $E_{\min}$ und $E_{\max}$ liegt und gehen weiter unten auf die anderen Fälle ein.

Neben (1.25) gibt es noch die so genannten gerichteten Rundungsvorschriften wie Auf- bzw. Abrunden (*round to (minus) infinity*) sowie das Abschneiden, bei dem alle Ziffern ab der P-ten Stelle verworfen werden, d.h. man wählt als Rundung die nächst gelegene Gleitpunktzahl mit kleinerem Betrag (*round to zero*).

Der absolute Fehler bei der Rundung hängt offensichtlich vom Exponenten E ab und ist daher nicht besonders aussagekräftig, so dass man den *relativen Rundungsfehler* betrachtet. Bei Verwendung der Rundungsvorschrift (1.25) gilt für $x \in \mathbb{R} \setminus \{0\}$ mit $B^{E_{\min}} \leq |x| < B^{E_{\max}+1}$ die Ungleichung

$$\frac{|x - \mathrm{rd}(x)|}{|x|} \leq \frac{1}{2} B^{-(P-1)} . \quad (1.26)$$

Gleitpunktrechnung. Der Ausdruck auf der rechten Seite der Fehlerabschätzung (1.26) ist eine obere Schranke für die *relative Maschinengenauigkeit*, die wir mit `eps` bezeichnen wollen:

$$\texttt{eps} = \frac{1}{2} B^{-(P-1)} .$$

Diese Größe bestimmt auch die Genauigkeit, mit der arithmetische Operationen in $\mathbb{G}$ durchgeführt werden. Wie eingangs erwähnt, können exakt gebildete Summen, Produkte und auch Quotienten von normalisierten Gleitpunktzahlen außerhalb von $\mathbb{G}$ liegen. Deshalb müssen diese arithmetischen Operationen durch entsprechende Gleitpunktoperationen $\oplus$, $\odot$ und $\oslash$ ersetzt werden, die die folgenden Minimalbedingungen erfüllen: Für alle $x, y \in \mathbb{G}$ gilt

$$x \oplus y , \; x \odot y \text{ und } x \oslash y \in \mathbb{G} . \quad (1.27)$$

Es liegt nahe, die Umsetzung dieser Forderungen mit der Rundungsvorschrift zu verknüpfen und für $x, y \in \mathbb{R}$ zu fordern, dass nach Möglichkeit die folgenden Gleichheiten gelten:

$$\begin{aligned} \mathrm{rd}(x) \oplus \mathrm{rd}(y) &= \mathrm{rd}(x+y)\,, \\ \mathrm{rd}(x) \odot \mathrm{rd}(y) &= \mathrm{rd}(x\,y)\,, \\ \mathrm{rd}(x) \oslash \mathrm{rd}(y) &= \mathrm{rd}(x/y)\,. \end{aligned} \tag{1.28}$$

Mit anderen Worten: Die Anwendung einer Gleitpunktoperation auf die gerundeten Größen sollte äquivalent zu der Rundung des Ergebnisses der entsprechenden exakten Operation sein. Im Allgemeinen sind Gleitpunktoperationen weder assoziativ noch distributiv, wie einfache Beispiele zeigen.

Da die Operanden der Gleitpunktoperationen Eingabedaten sind, die durch die Rundung mit einem relativen Fehler behaftet sind, gibt die Kondition der jeweiligen arithmetischen Operation schon einen Hinweis darauf, ob und wann Probleme zu erwarten sind. Aus Beispiel 1.13 wissen wir, dass die Subtraktion zweier näherungsweise gleicher Zahlen eine schlecht konditionierte Aufgabe darstellt. Wie zu befürchten ist, sind die Auswirkungen auf das Ergebnis der entsprechenden Gleitpunktoperation bei dieser Konstellation erheblich:

Beispiel 1.17 (Auslöschung signifikanter Stellen). Wir wählen als Basis $B = 10$. Die Summe der reellen Zahlen

$$x = 1.004\,, \quad y = -0.9986$$

ist $5.4 \cdot 10^{-3}$. Die normalisierte Gleitpunktdarstellung sei durch die Mantissenlänge $P = 3$ und $E_{\min} = -4$ charakterisiert. Dann gilt nach (1.25)

$$\mathrm{rd}(x) = 1.00 \cdot 10^{0}\,, \quad \mathrm{rd}(y) = -9.99 \cdot 10^{-1}\,,$$

und es wird

$$\mathrm{rd}(x) + \mathrm{rd}(y) = 1.00 \cdot 10^{-3}$$

berechnet. Dieses Ergebnis weist gegenüber dem exakten Resultat einen relativen Fehler von ungefähr 0.8148 auf, was in etwa 81.5 % entspricht. Vergleicht man das mit dem Rundungsfehler der Summanden von jeweils etwa 0.4%, so stellt man eine Verstärkung um mehr als das 200fache fest. Diesen Effekt nennt man *Auslöschung signifikanter Stellen*.

Bei dieser Betrachtung ist noch zu beachten, dass die Addition der gerundeten Werte exakt ausgeführt wurde, was nicht unbedingt so sein muss. Bei der Gleitpunktaddition $\oplus$ werden zunächst die Exponenten der Summanden angeglichen, indem in der Mantisse des Summanden mit kleinerem Exponenten entsprechend viele Nullen von links eingefügt werden. Wenn für die Berechnung die Mantissenlänge um die entsprechende Anzahl von Stellen verlängert wird, gelangt man zu dem obigen Resultat. Behält man aber etwa die Mantissenlänge $P = 3$ bei, so gilt

$$\mathrm{rd}(x) \oplus \mathrm{rd}(y) = \big(1.00 + (-0.99)\big) \cdot 10^{0} = 1.00 \cdot 10^{-2}\,,$$

und es ergibt sich ein relativer Fehler von ungefähr 85.2%. Übrigens ist in keinem der beiden Fälle die erste Forderung in (1.28) erfüllt. □

1.4.3 Binäre Realisierung

Die kleinste Informationseinheit auf einem Digitalrechner ist das *Bit*[1], das nur die Werte *0* und *1* annehmen kann. Im Folgenden soll $\mathbb{G}$ stets für eine Menge normalisierter Gleitpunktzahlen zur Basis $B = 2$ stehen.

Ganze Zahlen (*integer*). Die Menge der durch (1.18) darstellbaren ganzen Zahlen ist durch die hierfür reservierte *Bitlänge* eingegrenzt, wobei ein Bit den Wert von s und damit das Vorzeichen speichert (*Vorzeichenbit*). Beträgt die Bitlänge z.B. 32 Bit, so können damit alle ganzen Zahlen z zwischen -2^{31} und $2^{31} - 1$ dargestellt werden, wobei die 0 durch $x_j = 0$ für alle $j = 0, \ldots, 31$ gegeben ist.

Sofern das Ergebnis innerhalb des durch die Bitlänge festgelegten Bereichs liegt, werden ganzzahlige Addition sowie Multiplikation exakt und unter Einhaltung von Assoziativität und Distributivität ausgeführt. Liegt das Ergebnis aber außerhalb, so spricht man von einem *ganzzahligen Überlauf* (*integer overflow*). Zum ganzzahligen Überlauf kommt es z.B. auch, wenn eine betragsmäßig zu große Gleitpunktzahl in das ganzzahlige Format umgewandelt wird.

Gleitpunktzahlen (*floating point number*). Bei der normalisierten Gleitpunktdarstellung im Binärsystem folgt aus $x_0 \neq 0$ sofort $x_0 = 1$, d.h. die Mantisse hat die Gestalt

$$m = 1.x_1 \ldots x_{P-1} = 1.f\,,$$

mit dem gebrochenen Anteil f (*fraction*). Das führende Bit mit Wert *1* wird daher meist gar nicht explizit abgespeichert und man spricht vom *impliziten Bit*. Dadurch wird eine weitere Stelle der Mantisse frei für den gebrochenen Anteil, so dass nun P Stellen für f zur Verfügung stehen. Das Tripel (1.23) zur Darstellung einer normalisierten Gleitpunktzahl wird also im Binärsystem durch

$$x = (s, f, E)_2 \quad \text{mit} \quad f = f_1 \ldots f_P \tag{1.29}$$

ersetzt. Die Rundung auf die nächst gelegene Gleitpunktzahl lässt sich für das Binärsystem recht einfach realisieren: Liegt x genau in der Mitte zwischen zwei benachbarten Gleitpunktzahlen, so liefert (1.25) diejenige der beiden Zahlen, für die $f_P = 0$ gilt. Wegen $B = 2$ und der Verlängerung des gebrochenen Anteils um eine Stelle leitet sich aus der Rundungsfehlerabschätzung (1.26) die Ungleichung

$$\frac{|x - \mathrm{rd}(x)|}{|x|} \leq 2^{-(P+1)} \tag{1.30}$$

ab. Wir betrachten zwei übliche Beispiele für dieses Zahlenformat:

[1] Abkürzung für *binary digit* (Binärziffer).

Beispiel 1.18 (Standard-Gleitpunktzahlen). Um Gleitpunktoperationen schnell ausführen zu können, verwenden die meisten Computer einen speziellen Hardwarebaustein, die so genannte *FPU (floating point unit)*. Um für die unterschiedlichen Computerarchitekturen eine weit gehende Transparenz in den berechneten Resultaten zu gewährleisten und eine einheitliche Schnittstelle zu den Programmiersprachen zu schaffen, definiert der Industriestandard *IEEE 754-1985* u.a. die beiden folgenden Typen von Gleitpunktzahlen:

Einfache Genauigkeit (single precision): Für diesen Typ von Gleitpunktzahlen beträgt die Bitlänge 32 Bit. Ein Bit enthält das Vorzeichen, 8 Bit sind für den Exponenten reserviert und die verbleibenden 23 Bit für die Mantisse. Dabei ist

$$E_{\min} = -126\,,\ E_{\max} = 127\,,$$

wobei der Exponent nicht mit Hilfe eines Vorzeichenbits dargestellt wird, sondern durch Verschiebung um einen konstanten Wert b (*Bias*):

$$E_b = E + b\,. \tag{1.31}$$

Bei diesem Gleitpunkttyp gilt $b = 127$. So wird z.B. $E_{min} = -126$ als $E_b = -126 + 127 = 1$ dargestellt und dem Exponenten $E = 73$ entspricht $E_b = 200$.

Doppelte Genauigkeit (double precision): Die Bitlänge beträgt 64 Bit. Neben dem Vorzeichenbit werden 11 Bit für den Exponenten und 52 für die Mantisse verwendet. Es ist

$$E_{\min} = -1022\,,\ E_{\max} = 1023\,,$$

und der Bias-Wert für den Exponenten beträgt hier $b = 1023$. □

An den beiden Beispielen fällt auf, dass die jeweilige Bitlänge des Exponenten durch die Grenzen $E_{\min}$ und $E_{\max}$ nicht ganz ausgeschöpft wird: Die beiden verbliebenen möglichen Exponenten $E_b = 0$ und $E_b = E_{\max} + b + 1$ werden nicht für die Darstellung normalisierter Gleitpunktzahlen verwendet. Diese *reservierten Exponenten* sorgen dafür, dass die Forderungen (1.27) nach Möglichkeit auch in arithmetischen „Grenzsituationen“ erfüllt werden (siehe Tabelle 1.1). Speziell beachte man, dass die Zahl 0 zwar als Ergebnis der Summe $x \oplus (-x)$ mit $x \in \mathbb{G}$ auftreten kann, jedoch nicht als normalisierte Gleitpunktzahl mit implizitem Bit darstellbar ist. Tabelle 1.1 gibt einen Überblick über die Verwendung der reservierten Exponenten. Dabei steht $f = 0$ für eine Mantisse, die ausschließlich binäre Nullen enthält. `inf` wird z.B. bei der Division einer Zahl $x \neq 0$ durch 0 als Ergebnis geliefert. `NaN` dient u.a. als Hinweis auf „dubiose Operationen“ wie $0 \oslash 0$ oder `inf`$\oslash$`inf`. Die in der Tabelle definierten *denormalisierten Gleitpunktzahlen* sind eigentlich Festkommazahlen, denn ihre Darstellung beinhaltet einen konstanten Exponenten:

$$x = (-1)^s\, 2^{E_{min}}\, 0.f\,.$$

Tabelle 1.1. Die Verwendung der reservierten Exponenten.

Fall		Name	Bedeutung
$E_b = 0$	$f = 0$		$x = 0$
	$f \neq 0$		x denormalisiert
$E_b = E_{\max} + b + 1$	$f = 0$	`inf`	$x = \pm\infty$ (je nach Vorzeichenbit)
	$f \neq 0$	`nan`	x ist „keine Zahl“ (*not a number*)

Während die positiven normalisierten Gleitpunktzahlen mit fallendem Exponenten E immer näher zusammenrücken, füllen die denormalisierten Zahlen den Bereich zwischen $2^{E_{min}-P}$ und $2^{E_{min}}$ gleichmäßig aus (siehe Abb. 1.5). Eine wichtige Konsequenz hieraus ist, dass die relativen Fehler sowohl der Rundung als auch der arithmetischen Operationen umso größer werden, je mehr man sich in diesem Bereich der 0 nähert.

Über- und Unterlauf. Ähnlich wie bei den ganzen Zahlen, kann es auch bei den Gleitpunktzahlen zu einem *Gleitpunktüberlauf* kommen. Damit ist gemeint, dass der Exponent einer Maschinenzahl E_{max} übersteigt. Dies kann z.B. während der Multiplikation zweier Gleitpunktzahlen geschehen, wenn die Summe ihrer Exponenten dem Betrag nach zu groß ist, oder der Überlauf wird erst durch die Rundung der Mantisse mit anschließender Anpassung des Exponenten verursacht. Das Ergebnis des Exponentenüberlaufs durch Rundung ist $\pm$ `inf` (siehe Tabelle 1.1), d.h. der Wert des Vorzeichenbits bleibt bestehen. Entsprechend kommt es zu einem *Unterlauf*, wenn der Betrag des Exponenten E_{min} unterschreitet. Auch bei Rundung zu 0 bleibt der Wert des Vorzeichenbits bestehen, so dass man durchaus -0 erhalten kann. Ist dass Ergebnis der Rundung nicht 0, sondern die nächstgelegene denormalisierte Zahl, so spricht man auch von einem *allmählichen Unterlauf* (*gradual underflow*). Neben der geringeren Rechengenauigkeit besteht eine weitere Tücke dieses Zahlenbereichs darin, dass die Bildung des Kehrwerts einer denormalisierten Zahl zu einem Überlauf führt. Für eine detailliertere Darstellung der Gleitpunktarithmetik verweisen wir auf [4].

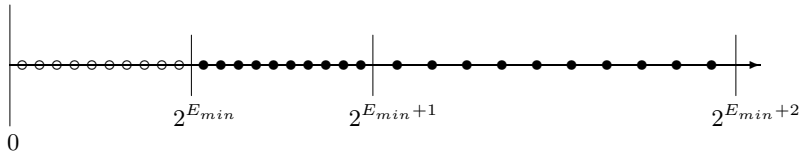

Abb. 1.5. Die Lage der denormalisierten (○) und normalisierten (●) Gleitpunktzahlen.

1.5 Stabilität

Die in der Gleitpunktarithmetik auftretenden Rundungsfehler führen bei der konkreten Realisierung eines Algorithmus dazu, dass man selbst bei fehlerfreien Eingabedaten x nicht die exakte Lösung $f(x)$ des Problems f erhält, sondern ein davon abweichendes Resultat $\tilde{f}(x)$. Zu den Rundungsfehlern gesellen sich in vielen Algorithmen noch die *Verfahrensfehler*, die dadurch entstehen, dass man das ursprüngliche Problem durch ein anderes ersetzt, das leichter zu lösen ist. Die Abb. 1.3 zum Euler-Verfahren aus Beispiel 1.6 illustriert diesen Effekt.

> *Ein Algorithmus heißt* stabil, *wenn sich Rundungs- und Verfahrensfehler nur mäßig auf das Resultat auswirken.*

Während die Kondition eine Eigenschaft des zu lösenden Problems ist, charakterisiert die Stabilität also das angewendete Lösungsverfahren im Hinblick auf die Qualität der gelieferten Ergebnisse.

An der Tatsache, dass die Gleitpunktoperationen weder assoziativ noch distributiv sind, kann man bereits erahnen, dass es nicht egal ist, welche arithmetischen Operationen in welcher Reihenfolge durchgeführt werden.

Beispiel 1.19 (Summenberechnung). Wir wollen die Summe

$$s = a + b + c$$

dreier Gleitpunktzahlen $a, b, c \in \mathbb{G}$ berechnen. Da die Gleitpunktaddition nicht assoziativ ist, bieten sich uns zwei Möglichkeiten:

Algorithmus 1: $x := a \oplus b\,, \quad s_1 := x \oplus c.$
Algorithmus 2: $y := b \oplus c\,, \quad s_2 := a \oplus y.$

Wir beginnen mit der Betrachtung von Algorithmus 1. Der Quotient

$$\epsilon_1 = \frac{(a \oplus b) - (a + b)}{a + b} = \frac{x - (a + b)}{a + b}$$

ist die relative Abweichung des berechneten Werts x vom exakten Ergebnis, wobei das Vorzeichen beachtet wird. Bezeichnen wir mit ϵ_2 die entsprechende relative Abweichung von s_1, so gilt offensichtlich

$$x = (a + b)(1 + \epsilon_1)\,, \quad s_1 = (x + c)(1 + \epsilon_2)\,.$$

Durch Zusammenfassen ergibt sich

$$s_1 = \big(s + (a + b)\epsilon_1\big)(1 + \epsilon_2) = s + s\epsilon_2 + (a + b)\epsilon_1(1 + \epsilon_2)\,,$$

und damit für den relativen Fehler im Endergebnis:

$$\frac{|s - s_1|}{|s|} = \left|\epsilon_2 + \frac{(a + b)}{|a + b + c|}\epsilon_1(1 + \epsilon_2)\right|.$$

Da alle relativen Rundungsfehler nach oben durch eps beschränkt sind, erhalten wir mit Hilfe der Dreiecksungleichung

$$\frac{|s - s_1|}{|s|} \leq \texttt{eps}\left(1 + \frac{|a+b|}{|a+b+c|}(1+\texttt{eps})\right).$$

Die Untersuchung von Algorithmus 2 führen wir auf die gleiche Art und Weise durch und erhalten

$$\frac{|s - s_2|}{|s|} \leq \texttt{eps}\left(1 + \frac{|b+c|}{|a+b+c|}(1+\texttt{eps})\right).$$

Die stabilere Methode beginnt die Summation also mit den Zahlen, deren Summe betragsmäßig die kleinere ist. □

In anderen Fällen kann man durch geeignetes Ersetzen von mathematischen Ausdrücken die Stabilität einer Methode verbessern. Dies können äquivalente Ausdrücke sein oder aber sogar solche, die nur näherungsweise gleich sind. Dazu jeweils ein Beispiel:

Beispiel 1.20 (Lösung der quadratischen Gleichung). Die Lösungen der quadratischen Gleichung

$$x^2 + px + q = 0$$

sind für $p^2 > 4q$ durch

$$x_+ = -\frac{p}{2} + \sqrt{D}\,,\; x_- = -\frac{p}{2} - \sqrt{D} \quad \text{mit } D = \frac{p^2}{4} - q > 0\,, \tag{1.32}$$

gegeben. Wir betrachten den folgenden Fall:

$$4\,|q| \ll p^2 \implies \sqrt{D} \approx \left|\frac{p}{2}\right|.$$

Dann liegt die betragsmäßig kleinere Lösung in der Nähe der 0 und bei der Verwendung der entsprechenden Formel in (1.32) ist Auslöschung zu befürchten: Die Berechnung dieser Lösung ist instabil.

Ratsamer ist es, zuerst die betragsmäßig größere Nullstelle mit Hilfe von

$$x_1 = -\left(\frac{p}{2} + \operatorname{sgn}(p)\sqrt{D}\right)$$

zu berechnen, wobei

$$\operatorname{sgn}(p) = \begin{cases} 1 & \text{, falls } p > 0 \\ 0 & \text{, falls } p = 0 \\ -1 & \text{, falls } p < 0 \end{cases}$$

das Vorzeichen (*Signum*) von p ist. Die beiden Summanden haben in diesem Fall das gleiche Vorzeichen, so dass Auslöschung vermieden wird. Der Satz

von Vieta besagt, dass q das Produkt der beiden Lösungen ist. Daher wird die instabile zweite Formel in (1.32) gar nicht benötigt, denn die Berechnung durch

$$x_2 = \frac{q}{x_1}$$

verursacht keine Probleme. □

Beispiel 1.21 (Auswertung von sinh**).** Die *hyperbolische Sinusfunktion* (Sinus hyperbolicus) ist für $x \in \mathbb{R}$ definiert durch

$$\sinh(x) = \frac{\mathrm{e}^x - \mathrm{e}^{-x}}{2}\,.$$

Offensichtlich gilt

$$\mathrm{e}^x \approx \mathrm{e}^{-x} \Leftrightarrow x \approx 0\,,$$

so dass bei Verwendung der Definition als Auswertungsmethode für betragsmäßig kleine x wegen der Auslöschung führender Stellen ein instabiles Verhalten zu erwarten ist. Dabei spielt es auch keine Rolle, wie exakt die Exponentialfunktion ausgewertet wird. Im Gegenteil: Für $x \approx 0$ kann man eine stabilere Methode erhalten, wenn man die Exponentialfunktion nur annähert. In der Analysis lernt man, dass für alle $x \in \mathbb{R}$ gilt:

$$\mathrm{e}^x = \lim_{n\to\infty} \sum_{k=0}^{n} \frac{x^k}{k!} = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} \ldots\,.$$

Wählen wir die Summe mit $n = 3$ als Näherung für e^x und e^{-x}, so lautet die entsprechende Approximation für die hyperbolische Sinusfunktion für $x \approx 0$:

$$\sinh(x) \approx x + \frac{x^3}{6}\,.$$

Die Summanden haben gleiches Vorzeichen und Auslöschung wird vermieden.

1.6 Vom Problem zum Programm – und zurück

Die binäre Realisierung von Zahlen auf Digitalrechnern zeigt, dass der Computer eine „eigene Sprache spricht". Das Formulieren von Algorithmen in dieser *Maschinensprache* erfordert eine Übersetzungstätigkeit durch den menschlichen Programmierer und stellt eine entsprechend zeitaufwendige Angelegenheit dar. Es ist bequemer und zeitsparender, die auszuführenden Anweisungen in einer dem Menschen eher zugänglichen *Programmiersprache* zu formulieren. Dieser *Quelltext* muss vor der Ausführung vom Computer in seine eigene Maschinensprache übersetzt werden. Zu diesem Zweck muss die zu übersetzende Sprache standardisiert sein, um einen automatisierten Übersetzungsvorgang zuverlässig durchführen zu können.

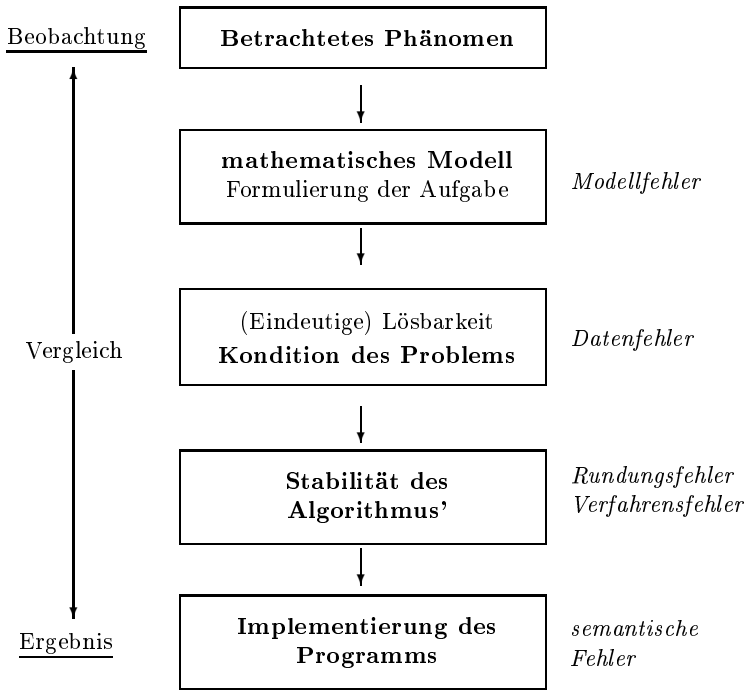

Abb. 1.6. „Gefahren“ auf dem Weg vom Problem zur rechnergestützten Lösung.

Bei den so genannten *Interpretersprachen* wird das Programm Anweisung für Anweisung auf seine *syntaktische Korrektheit* überprüft und ausgeführt. Programmierfehler jeglicher Art machen sich bei dieser Variante erst zur Laufzeit des Programms bemerkbar.

Im Unterschied dazu wird bei den *Compilersprachen* (wie z.B. C) der in einer oder mehreren Dateien enthaltene Quelltext zuerst als Ganzes von einem *Compiler*-Programm analysiert und anschließend in die Maschinensprache übersetzt. Dabei können nicht nur eventuelle Fehler im Quelltext bereits im Vorfeld aufgespürt werden, sondern durch die Betrachtung des gesamten Programms bietet sich auch die Möglichkeit, das Programm möglichst optimal der Computerarchitektur anzupassen. Hierbei ist aber zu beachten, dass bei der Analyse des Quelltextes lediglich syntaktische Fehler im Quelltext ausfindig gemacht werden können. Schließlich kann der Compiler nicht wissen, ob die Anweisungen auch exakt dem entsprechen, was man beim Verfassen des Codes im Sinn hatte. Solche *semantischen Fehler* bleiben also meist von den automatischen Helfern unentdeckt und der Entwickler muss sie selbst finden und beheben. Gelingt dies nicht, so wird der Algorithmus nicht korrekt aus-

geführt und es kommt zu Laufzeitfehlern (*run time error*) wie z.B. falschen Ergebnissen oder Programmabstürzen.

Ist die Implementierung als Programm jedoch korrekt, so muss man bei unbefriedigenden Ergebnissen den Algorithmus noch einmal auf seine Stabilität bzw. Korrektheit überprüfen und entsprechende Änderungen vornehmen. Hilft auch das nichts, so muss sogar das zugrundeliegende mathematische Modell wieder auf den Prüfstand.

Es ist also ein langer, von vielerlei Fehlerquellen gesäumter Weg von der gestellten Aufgabe bis zu ihrer rechnergestützten Lösung. In Abb. 1.6 ist dieser Prozess zusammenfassend dargestellt. Die Auflistung von möglichen Schwierigkeiten soll keinesfalls entmutigend wirken, trägt aber hoffentlich ein wenig dazu bei, dass man die Arbeit der Entwickler von gut funktionierenden Computersimulationen zu würdigen weiß.

1.7 Kontrollfragen zu Kapitel 1

Frage 1.1

Welche der folgenden Aussagen trifft *nicht* zu?

a) Die Kondition ist eine Eigenschaft des Problems und nicht der verwendeten Lösungsmethode. ☐
b) Auch zu gut konditionierten Problemen können instabile Lösungsverfahren existieren. ☐
c) Ein statisch finiter Algorithmus ist stets terminierend. ☐
d) Ein determinierter Algorithmus liefert bei gleichen Eingabedaten immer das gleiche Ergebnis. ☐
e) Ein Algorithmus darf auf bereits existierende Algorithmen zurückgreifen, wenn diese korrekt sind. ☐

Frage 1.2

Welche der folgenden Aussagen zur $\mathcal{O}$-Notation ist zutreffend?

a) Für hinreichend große $n \in \mathbb{N}$ gilt: $\mathrm{e}^x = \mathcal{O}(x^n)$ für $x \to \infty$. ☐
b) Wenn $f(x) = \mathcal{O}(g(x))$ für $x \to x_0$ gilt, so auch $f(x) = o(g(x))$ für $x \to x_0$. ☐
c) $x^2 = O(x)$ für $x \to \infty$. ☐
d) Für alle $n \in \mathbb{Z}$ gilt: $\mathrm{e}^x = \mathcal{O}(x^n)$ für $x \to -\infty$. ☐
e) $x = o(x^2)$ für $x \to 0$. ☐

Frage 1.3

Die Funktion $\sin(x)$ kann für kleine x durch die Funktion

$$f(x) = x - \frac{x^3}{6}$$

approximiert werden. Wie groß ist der relative Fehler bei Verwendung dieser Approximation für $x = 2.0$ (in Radiant), auf vier Nachkommastellen gerundet?

a) 0.2668 ☐
b) 0.3639 ☐
c) 0.2426 ☐
d) 0.2886 ☐
e) 0.2425 ☐

Frage 1.4

Wenn die Dezimaldarstellung einer Gleitpunktzahl durch 12.125 gegeben ist, so lautet ihre Binärdarstellung:

a) $1010.101_{|2}$ ☐
b) $1100.001_{|2}$ ☐
c) $1100.01_{|2}$ ☐
d) $1010.101_{|2}$ ☐
e) $1110.111_{|2}$ ☐

Frage 1.5

Bei welcher Operation in Gleitpunktarithmetik kann die „Auslöschung“ führender Stellen auftreten?

a) Bei der Addition $a + b$, wobei $a \approx b$ im Rahmen der Maschinengenauigkeit. ☐
b) Bei der Multiplikation zweier Zahlen unabhängig von deren Vorzeichen, wenn deren Beträge stark voneinander abweichen. ☐
c) Bei der Division zweier Zahlen, deren Betrag fast identisch ist. ☐
d) Bei der Subtraktion zweier Zahlen mit gleichem Vorzeichen, deren Beträge stark voneinander abweichen. ☐
e) Bei der Addition zweier Zahlen mit unterschiedlichem Vorzeichen, deren Beträge fast identisch sind. ☐

Frage 1.6

Welche der folgenden Aussagen zu den Gleitpunktoperationen trifft nicht zu?

a) Das Distributivgesetz gilt. ☐
b) Die Addition ist kommutativ. ☐
c) Die Multiplikation ist kommutativ. ☐
d) Die Division durch 0 liefert `inf` oder `NaN`. ☐
e) Die Addition ist nicht assoziativ. ☐

1.8 Übungsaufgaben zu Kapitel 1

1.1 (Eindeutigkeit der Lösung für die Bakterienkultur).
Zeigen Sie, dass

$$y(t) = y_0 \, \mathrm{e}^{\lambda(t-t_0)}$$

tatsächlich die einzige Lösung des Anfangswertproblems

$$y'(t) = \lambda y(t) \quad , \quad y(t_0) = y_0$$

ist. Nehmen Sie dazu an, dass z eine weitere Lösung ist und betrachten Sie die Ableitung des Quotienten $z(t)/y(t)$.

1.2 (Aufwand für die Berechnung des Matrizenprodukts).
Das Produkt einer Matrix $A \in \mathbb{R}^{m\times n}$ mit Einträgen a_{ij} und einer Matrix $B \in \mathbb{R}^{n\times p}$ mit Einträgen b_{ij} ist folgendermaßen definiert:
Das Produkt aus A und B ist eine Matrix $C \in \mathbb{R}^{m\times p}$, deren Einträge berechnet werden durch

$$c_{ik} = \sum_{j=1}^{n} a_{ij} \, b_{jk} \, , \quad i = 1, \ldots, m \, , \; k = 1 \ldots, p \, .$$

Berechnen Sie den Gesamtaufwand an Operationen für die Berechnung von $C = AB$.

1.3 (Eigenschaften der Landau-Symbole).
Beweisen Sie die Aussagen in Satz 1.12. Welche dieser Aussagen gelten auch für das Symbol $o(\cdot)$?

1.4 (Konditionsanalyse für Quadratwurzel und Produkt).

a) Zeigen Sie, dass das Ziehen der Quadratwurzel ein gut konditioniertes Problem ist: Für alle $x > 0$ gilt

$$e_{rel}(\sqrt{x}) \leq e_{rel}(x) \, ,$$

d.h. der relative Fehler wird sogar gedämpft.

b) Zeigen Sie, dass für den relativen Fehler bei der Multiplikation zweier reeller Zahlen $a, b \neq 0$ gilt:

$$e_{rel}(a \, b) \leq e_{rel}(a) + e_{rel}(b) + e_{rel}(a) \, e_{rel}(b) \, .$$

1.5 (Eigenschaften der Gleitpunktoperationen).
Zeigen Sie anhand einfacher Beispiele von Gleitpunktzahlenmengen $\mathbb{G}$ und entsprechenden Elementen $x, y, z \in \mathbb{G}$, dass im allgemeinen gilt:

a) $x + y \notin \mathbb{G}$ bzw. $x \, y \notin \mathbb{G}$;
b) Gleitpunktoperationen $\oplus$ und $\odot$ sind im Allgemeinen weder assoziativ noch distributiv. Verwenden Sie dabei die übliche Rundungsvorschrift (1.25).

1.6 („Reichweite“ der Gleitpunktzahlen).
Schätzen Sie jeweils die obere und untere Schranke der Zahlenbereiche, die durch die IEEE-Gleitpunktzahlen aus Beispiel 1.18 in normalisierter Form darstellbar sind.

1.7 (Relative Maschinengenauigkeit).

a) Leiten Sie aus der Rundungsvorschrift (1.25) die Fehlerabschätzung (1.26) ab und folgern Sie für die Binärdarstellung mit implizitem Bit die Gültigkeit von (1.30).
b) In der Literatur findet sich häufig eine alternative Definition der relativen Maschinengenäuigkeit: `eps` ist das kleinste positive $\epsilon \in \mathbb{G}$, für das $1 \oplus \epsilon \neq 1$ gilt. Ist diese Definition äquivalent?
c) In Prüfungen bekommt man hin und wieder zu hören, dass die relative Maschinengenauigkeit identisch mit der kleinsten darstellbaren Gleitpunktzahl ist. Machen Sie sich klar, warum das im Allgemeinen nicht so ist!

1.8 (Differenz zweier Quadratzahlen).
Zur Berechnung von $d = x^2 - y^2$ bieten sich die folgenden beiden Algorithmen an:

Algorithmus 1: $a_1 = x \odot x$, $b_1 = y \odot y$, $d_1 = a \oplus (-b)$.
Algorithmus 2: $a_2 = x \oplus y$, $b_2 = x \oplus (-y)$, $d_2 = a \odot b$.

Überlegen Sie sich, wann die Aufgabe schlecht konditioniert ist und welche der beiden Methoden dann stabiler ist.

http://www.springer.com/978-3-540-45383-3

Programmieren in C
Eine mathematikorientierte Einführung
Kirsch, R.; Schmitt, U.
2007, XII, 312 S., Softcover
ISBN: 978-3-540-45383-3