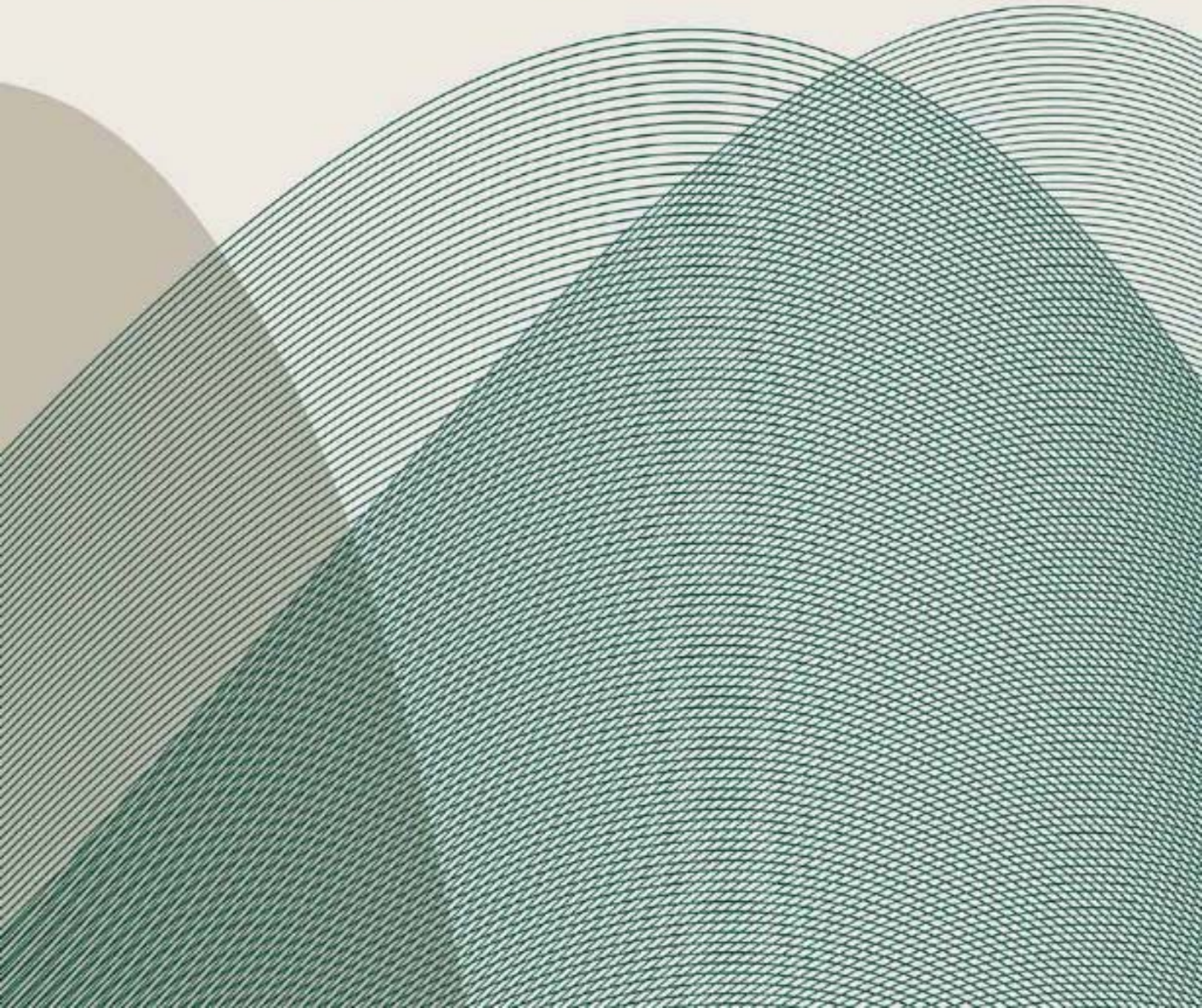
BIRKHÄUSER

Mathematik Kompakt

Algorithmische Methoden

Zahlen, Vektoren, Polynome

Philipp Kügler
Wolfgang Windsteiger



BIRKHÄUSER

Mathematik Kompakt

Herausgegeben von:

Martin Brokate
Heinz W. Engl
Karl-Heinz Hoffmann
Götz Kersting
Gernot Stroth
Emo Welzl

Die neu konzipierte Lehrbuchreihe *Mathematik Kompakt* ist eine Reaktion auf die Umstellung der Diplomstudiengänge in Mathematik zu Bachelor- und Masterabschlüssen. Ähnlich wie die neuen Studiengänge selbst ist die Reihe modular aufgebaut und als Unterstützung der Dozenten wie als Material zum Selbststudium für Studenten gedacht. Der Umfang eines Bandes orientiert sich an der möglichen Stofffülle einer Vorlesung von zwei Semesterwochenstunden. Der Inhalt greift neue Entwicklungen des Faches auf und bezieht auch die Möglichkeiten der neuen Medien mit ein. Viele anwendungsrelevante Beispiele geben dem Benutzer Übungsmöglichkeiten. Zusätzlich betont die Reihe Bezüge der Einzeldisziplinen untereinander.

Mit *Mathematik Kompakt* entsteht eine Reihe, die die neuen Studienstrukturen berücksichtigt und für Dozenten und Studenten ein breites Spektrum an Wahlmöglichkeiten bereitstellt.

Algorithmische Methoden

Zahlen, Vektoren, Polynome

Phillipp Kügler
Wolfgang Windsteiger

Birkhäuser
Basel · Boston · Berlin

Autoren:
Phillipp Kügler
Institut für Industriemathematik
Universität Linz
Altenberger Str. 69
4040 Linz
Austria
email: philipp.kuegler@jku.at

Wolfgang Windsteiger
Research Institute for Symbolic Computation (RISC)
Universität Linz
Altenberger Str. 69
4040 Linz
Austria
email: Wolfgang.Windsteiger@risc.uni-linz.ac.at

2000 Mathematical Subject Classification: 00-01, 68-01, 65-01, 65D15
1998 ACM Computing Classification: F.2.1 [Numerical Algorithms and Problems]: Computations on polynomials: Number-theoretic computations; G.1.0 [General]: Computer arithmetic: Conditioning (and ill-conditioning): Error analysis: Numerical algorithms: Stability (and instability); G.4 [Mathematical Software]: Algorithm design and analysis; I.1.2 [Algorithms]: Algebraic algorithms: Analysis of algorithms

Bibliografische Information der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.ddb.de> abrufbar.

ISBN 978-3-7643-8434-0 Birkhäuser Verlag AG, Basel – Boston – Berlin

Gedruckt auf säurefreiem Papier, hergestellt aus chlorfrei gebleichtem Zellstoff. TCF ∞
Satz und Layout: Protago-TEX-Production GmbH, Berlin, www.ptp-berlin.eu
Printed in Germany

ISBN 978-3-7643-8434-0

9 8 7 6 5 4 3 2 1 www.birkhauser.ch

Vorwort

Mathematische Algorithmen – darunter wollen wir uns zunächst einfach *Rechenvorschriften* vorstellen – werden oft als wertvolles Hilfsmittel zur Lösung von Problemen aus Industrie, Naturwissenschaft und Technik herangezogen. Eine Voraussetzung dafür ist, dass die zur Diskussion stehenden Prozesse und Phänomene samt der damit verbundenen Fragestellungen in die Sprache der Mathematik übersetzt werden. Dies ist Gegenstand der mathematischen Modellierung, an deren Ende im Erfolgsfall ein mathematisches Problem steht. Ein ideales Szenario ist dann, dass dieses Problem hinsichtlich seiner mathematischen Eigenschaften untersucht, durch Einsatz eines mathematischen Algorithmus gelöst, und schließlich diese Lösung in die Sprache des Ingenieurs, Naturwissenschafters oder Technikers zu dessen Zufriedenheit rückübersetzt wird. Ein realer Lösungsweg ist in der Regel natürlich ungleich steiniger als die hier skizzierte Einbahnstraße und kann vor Erreichen des Zieles viele Sackgassen, Irr- und Umwege beinhalten.

Der wichtige Teilschritt von der mathematischen Problemstellung zur mathematischen Lösung mittels eines Algorithmus sowie dessen Realisierung am Computer sind Gegenstand des vorliegenden Buches. Es basiert auf der Vorlesung *Algorithmische Methoden*, die von den Professoren Bruno Buchberger und Heinz W. Engl im Umfeld des SFB F013 „Numerical and Symbolic Scientific Computing“ an der Johannes Kepler Universität Linz ins Leben gerufen wurde, und die wir in den vergangenen Jahren im Rahmen des Bachelor-Studiums Technische Mathematik gehalten haben. In zwei Bänden behandeln wir gängige Problemstellungen aus der *Linearen Algebra* und der *Analysis*, deren Grundinhalte wir als bekannt voraussetzen. Der Schwerpunkt liegt vielmehr auf der Entwicklung und der Computerumsetzung der klassischen Algorithmen zur Lösung der in diesen Lehrveranstaltungen diskutierten Probleme. Die Gliederung erfolgt anhand der mathematischen Objekte, die in den vorgestellten Methoden die zentrale Rolle spielen. Im ersten Teil stehen Zahlen, Vektoren und univariate Polynome im Mittelpunkt. In Band *2*, damit ist die im Rahmen der Reihe „Mathematik Kompakt“ erscheinende Fortsetzung gemeint, behandeln wir Algorithmen für Funktionen, Matrizen und multivariate Polynome. Aus Rücksicht auf den Umfang verzichten wir auf Algorithmen, die komplexe Zahlen involvieren, und führen etwa die Skalarproduktsberechnung nur für Vektoren über $\mathbb{R}$ durch.

Das Buch richtet sich an Lehrende und Studierende im *dritten Semester*, kann aber auch begleitend in einer algorithmisch orientierten Präsentation der Linearen Algebra und/oder Analysis im *ersten Studienjahr* eingesetzt werden. Wir wiederholen jene Inhalte dieser Grundvorlesungen, die für das Verständnis der präsentierten Algorithmen erforderlich sind. Danach erläutern wir, wie die jeweiligen

mathematischen Objekte am Computer mit Hilfe von Datenstrukturen dargestellt werden können, und gehen auf damit verbundene elementare Rechenoperationen ein, etwa die Addition rationaler Zahlen oder die Multiplikation zweier Polynome. Im Zentrum jedes Abschnitts stehen dann Aufgaben, die hinsichtlich ihrer Lösbarkeit und – soweit sinnvoll – ihrer Sensitivität gegenüber Störungen der Eingangsdaten untersucht werden, sowie die Herleitung von Algorithmen zu deren Lösung. Bei der Diskussion der Algorithmen achten wir auf den Aufwand, mit dem die Berechnung einer Lösung am Computer verbunden ist, sowie auf Rechenfehler, die durch Diskretisierung, vorzeitigen Abbruch, Rundung und/oder fehlerhafte Eingangsdaten entstehen können. Wenn immer möglich und sinnvoll, stellen wir zudem *numerische* und *symbolische* Zugänge einander gegenüber.

Die Algorithmen erklären wir anhand von Pseudocode, so dass unsere Diskussion nicht an eine bestimmte Programmiersprache oder Mathematik-Software gebunden ist. Tatsächliche Implementierungen in *Mathematica* und/oder MATLAB der im Buch beschriebenen Algorithmen bieten wir frei verfügbar auf einer *E-content Plattform* unter

`http://www.risc.uni-linz.ac.at/publications/books/AlgorithmischeMethoden/`

an. Die Beispiele im Buch beziehen sich meist auf Aufrufe dieser *Mathematica*- oder MATLAB-Programme. Ein Download ist für das Verständnis jedoch nicht unbedingt erforderlich, vielmehr empfehlen wir eine selbstständige Umsetzung des Pseudocodes in ein lauffähiges Programm. Ein Ergebnis in blauer Schrift weist darauf hin, dass dieses Resultat mit einem unserer Algorithmen oder internen Befehlen in *Mathematica* oder MATLAB berechnet wurde.

Textpassagen, die sich auf die Repräsentation von mathematischen Objekten am Computer im weitesten Sinn beziehen, sind speziell als *Computerrepräsentation* gekennzeichnet. Solche, in denen bestimmte Aspekte der Programmierung in konkreten Programmiersprachen im Vordergrund stehen, sind als *Computerprogrammierung* hervorgehoben.

Kapitel I behandelt Grundbegriffe und Grundfragen einer algorithmischen Mathematik und legt damit das Fundament für alles später Folgende. Kapitel II erklärt, wie am Computer in verschiedenen Zahlbereichen gerechnet werden kann, und beleuchtet damit einige der Prinizipien, die hinter dem Rechnen mit Zahlen in Programmiersprachen und mathematischen Softwaresystemen stehen. Die Kapitel III und IV stellen Computerdarstellungen und ausgewählte Algorithmen bzw. Problemstellungen in Zusammenhang mit Vektoren und univariaten Polynomen vor. Sie hängen inhaltlich voneinander kaum ab und können damit im Grunde unabhängig voneinander studiert werden.

Mathematische Konventionen

Als mathematisches Rüstzeug setzen wir die üblichen Sprachkonstrukte der Mengenlehre wie $A \cup B$, $A \cap B$, $A \setminus B$, $\emptyset$ oder $\{x \in A \mid \ldots\}$ ohne weitere Erläuterungen voraus. Für die Teilmengenrelation verwenden wir $A \subset B$. Um hervorzuheben, dass A eine echte Teilmenge von B ist, schreiben wir $A \subsetneq B$. Die Menge der natürlichen Zahlen $\mathbb{N}$ beginnt mit 1, für $\mathbb{N} \cup \{0\}$ verwenden wir $\mathbb{N}_0$.

Dank

Prof. Heinz W. Engl hat es uns ermöglicht, an der Reihe „Mathematik Kompakt“ mitzuwirken, und ist uns während der Arbeit an diesem Buch mit Rat und Tat zur Seite gestanden. Dafür möchten wir uns herzlich bedanken. Unser Dank gilt auch *Prof. Peter Paule* für die spannenden Diskussionen über die Gestaltung des Buches sowie *Prof. Heinrich Rolletschek* und *Dr. Sven Beuchler* für das Korrekturlesen des Textes.

Prof. Karl-Heinz Hoffmann und *Dr. Thomas Hempfling*, stellvertretend für die Herausgeber der Serie und den Birkhäuser Verlag, sprechen wir unseren Dank aus für das angenehme Arbeitsklima und die Geduld, die dem Projekt entgegengebracht wurde. Bei *Prof. Gernot Stroth* und *Prof. Emo Welzl* bedanken wir uns für die inhaltliche Betreuung und die wertvollen Anregungen.

Dem Leser wünschen wir abschließend viel Spaß bei der Lektüre. Wir hoffen, mit dem Buch Studenten die algorithmische Mathematik schmackhaft zu machen, und Vortragenden bei der Gestaltung einer algorithmisch orientierten Einstiegslehrveranstaltung behilflich zu sein.

Inhaltsverzeichnis

Algorithmenverzeichnis

I Grundbegriffe und Grundfragen einer algorithmischen Mathematik

In diesem Kapitel schaffen wir die Voraussetzungen für einen algorithmisch orientierten Zugang zur Lösung mathematischer Fragestellungen. Dazu überlegen wir uns, wie mathematische Probleme spezifiziert und welche Aussagen über deren Lösung getroffen werden können. Anhand einführender Beispiele wenden wir uns Algorithmen zur Problemlösung zu und diskutieren Eigenschaften von Algorithmen, die auch zu deren Qualitätsbeurteilung herangezogen werden können.

1 Probleme, Lösungen und Algorithmen

Problemspezifikation

Eine *Problemspezifikation* legt fest, *was gegeben* und *was gesucht* ist, bestimmt *Eigenschaften* der gegebenen und gesuchten Größen und erklärt vor allem *den Zusammenhang* zwischen Gegebenem und Gesuchtem. Die Problemspezifikation bildet die Schnittstelle zwischen dem Problemsteller und dem Problemlöser. Anstelle von einzelnen konkreten Problemstellungen, wie beispielsweise „Löse die Gleichung $x^2 - 4x - 5 = 0$“ oder „Bestimme Quotient und Rest bei Division von 13 durch 6“, werden wir uns für ganze *Problemklassen* interessieren, wie etwa „Löse die Gleichung

$$x^2 - 2px + q = 0 \tag{1.1}$$

für x in $\mathbb{R}$ mit beliebigen $p,q \in \mathbb{R}$“ oder „Bestimme Quotient und Rest zweier natürlicher Zahlen m und n“. Allgemein werden wir Problemspezifikationen immer in der Form

Gegeben: $\mathtt{x}$
mit: $\mathcal{I}_{\mathtt{x}}$.
Gesucht: $\mathtt{y}$
mit: $\mathcal{O}_{\mathtt{x},\mathtt{y}}$.

anschreiben, wobei wir $\mathtt{x}$ und $\mathtt{y}$ die *Eingabe-* bzw. *Ausgabevariablen* sowie $\mathcal{I}_{\mathtt{x}}$ und $\mathcal{O}_{\mathtt{x},\mathtt{y}}$ die *Eingabe-* bzw. *Ausgabebedingung* nennen. Ein mathematisches Problem kann beliebig viele Eingabe- und Ausgabevariablen haben, in obiger Beschreibung stehen daher $\mathtt{x}$ und $\mathtt{y}$ jeweils für beliebig viele Variablen. Die Eingabebedingung

$\mathcal{I}_{\mathtt{x}}$ charakterisiert die in der Problemstellung als gegeben betrachteten Objekte und ist eine Formel mit frei vorkommenden Variablen[1] $\mathtt{x}$. Ein $\mathtt{x}$ mit der Eigenschaft $\mathcal{I}_{\mathtt{x}}$ nennen wir eine *zulässige Eingabe,* und die durch eine Belegung der Eingabevariablen mit zulässigen Eingaben enstehende konkrete Problemstellung nennen wir eine *Probleminstanz.* Die Ausgabebedingung $\mathcal{O}_{\mathtt{x},\mathtt{y}}$ ist das Herzstück der Spezifikation, sie legt die Beziehung zwischen Eingabe und Ausgabe fest und ist eine Aussage mit freien Variablen $\mathtt{x}$ und $\mathtt{y}$. Zu gegebenem zulässigen $\mathtt{x}$ nennen wir $\mathtt{y}$ mit der Eigenschaft $\mathcal{O}_{\mathtt{x},\mathtt{y}}$ eine *Lösung des Problems zu den Eingaben* $\mathtt{x}$. Außer den Eingabe- und Ausgabevariablen enthält eine Spezifikation keine weiteren freien Variablen. Es tauchen nur mathematische Begriffe auf, die entweder als bekannt vorausgesetzt werden oder separat definiert sind.

Problemstellungen dieser Art heißen auch *(explizite) Bestimmungsprobleme,* weil dabei die durch die Ausgabevariablen bezeichneten mathematischen Objekte zu bestimmen sind. Daneben gibt es auch noch sogenannte *Entscheidungsprobleme,* in denen eine Antwort „wahr“ oder „falsch“ in Abhängigkeit von bestimmten Eingabegrößen gesucht ist, siehe etwa [1]. Wir werden nicht weiter auf Entscheidungsprobleme eingehen und von nun an unter einer Problemstellung immer ein explizites Bestimmungsproblem verstehen.

Die Problemstellung des Lösens quadratischer Gleichungen aus (1.1) bzw. der Division mit Rest natürlicher Zahlen können wir in einem ersten Ansatz nach diesem Muster in folgender Weise spezifizieren.

Problemstellung (Reelle Lösung einer quadratischen Gleichung).

Gegeben:	p,q
mit:	$p,q \in \mathbb{R}$.
Gesucht:	x
mit:	$x \in \mathbb{R}$ und $x^2 - 2px + q = 0$.

Wir schreiben eine Typangabe für die Variablen häufig direkt in die Auflistung der Variablen, so dass in der Eingabe- und Ausgabebedingung die „interessanten“ Eigenschaften im Vordergrund stehen bzw. die Eingabebedingung in solchen Fällen zur Gänze entfallen kann.

Problemstellung (Division mit Rest natürlicher Zahlen).

Gegeben:	$m,n \in \mathbb{N}$.
Gesucht:	$q,r \in \mathbb{N}_0$
mit:	$m = nq + r$ und $r < n$.

Im Fall der quadratischen Gleichung sind die Eingabevariablen $\mathtt{x}$ durch p,q und die Ausgabevariablen $\mathtt{y}$ durch x gegeben. Die Werte $p = 2$ und $q = -5$ sind für dieses Problem zulässige Eingaben, daher ist das Lösen der Gleichung $x^2 - 4x - 5 = 0$ für $x \in \mathbb{R}$ eine Instanz dieser Problemstellung. Eine Lösung dazu lautet beispielsweise $x = -1$. Im Divisionsproblem sind m,n die Eingabe- und q,r die Ausgabevariablen. Eine Probleminstanz ist etwa die Division mit Rest von 13 durch 6, eine Lösung zu diesen Eingaben ist durch $q = 2$ und $r = 1$ gegeben.

[1] Eine *freie Variable* ist eine Variable, die nicht durch einen Quantor gebunden ist. Freie Variablen können, im Gegensatz zu gebundenen Variablen, mit beliebigen Werten belegt werden.

Existenz und Eindeutigkeit von Lösungen

Problemspezifikationen können Ausgabebedingungen enthalten, die nicht für alle Probleminstanzen erfüllbar sind. Es kann also sein, dass das Problem zu bestimmten zulässigen Eingaben keine Lösung besitzt. Nehmen wir etwa das oben spezifizierte Problem der Suche nach reellen Lösungen der quadratischen Gleichung $x^2 - 2px + q = 0$ für beliebige $p,q \in \mathbb{R}$. Bekanntlich besitzt die Gleichung nur für nichtnegative Diskriminante $p^2 - q$ reelle Lösungen.

Ist die Existenz von Lösungen gesichert – der Nachweis ist Gegenstand des *Existenzbeweises* – so stellt sich als nächstes die Frage nach der Eindeutigkeit von Lösungen, genauer: Ist für jede Probleminstanz die Lösung durch die Ausgabebedingung eindeutig festgelegt? Im Beispiel der quadratischen Gleichung mit nichtnegativer Diskriminante gibt es im Fall $p^2 - q = 0$ nur die Lösung $x = p$, für positive Diskriminante allerdings existieren zwei voneinander verschiedene reelle Lösungen

$$x = p + \sqrt{p^2 - q} \quad \text{und} \quad x = p - \sqrt{p^2 - q}.$$

Häufig werden wir die im Buch betrachteten Probleme so formulieren, dass Existenz und Eindeutigkeit der Lösung sichergestellt sind. Für unsere Problemspezifikation bedeutet dies dann, dass wir das gestellte Problem durch eine Abbildung φ charakterisieren können, die jeder zulässigen Eingabe die zugehörige Lösung zuordnet, d.h. wir definieren *für alle* x mit $\mathcal{I}_\mathrm{x}$

$$\varphi(\mathrm{x}) := \text{ dasjenige } \mathrm{y} \text{ mit } \mathcal{O}_{\mathrm{x},\mathrm{y}}. \tag{1.2}$$

Im Fall der eindeutigen Lösbarkeit werden wir oft das Problem mit φ und eine Instanz des Problems zu konkretem Input x mit (φ,x) bezeichnen, was nicht mit $\varphi(\mathrm{x})$ für die zu x gehörige Lösung zu verwechseln ist.

Im Beispiel der quadratischen Gleichung müssen wir zur Erfüllung dieser Kriterien die Problemspezifikation nochmals überdenken. Existenz können wir garantieren, indem wir eine nichtnegative Diskriminante als Bedingung an die Inputs p und q stellen. Eine Möglichkeit, Eindeutigkeit zu erreichen, besteht darin, dass wir im Falle zweier Lösungen nur nach der kleineren suchen.

Problemstellung (Kleinste reelle Lösung einer quadratischen Gleichung).

Gegeben:	$p,q \in \mathbb{R}$
mit:	$p^2 - q \geq 0$.
Gesucht:	$x \in \mathbb{R}$
mit:	$x^2 - 2px + q = 0$ und $x \leq p$.

In dieser Form ist für alle zulässigen p und q eine eindeutige Lösung spezifiziert, die zur Problemspezifikation passende Abbildung φ lässt sich *explizit* durch

$$\varphi(p,q) := p - \sqrt{p^2 - q} \tag{1.3}$$

beschreiben. Die Problemspezifikation zur Division mit Rest von Seite 2 erfüllt bereits die Kriterien hinsichtlich eindeutiger Existenz von Lösungen.

Satz

Division natürlicher Zahlen. Zu je zwei Zahlen $m,n \in \mathbb{N}$ existieren eindeutig bestimmte $q,r \in \mathbb{N}_0$ mit $m = nq + r$ und $r < n$.

Beweis. Existenz: Im Fall $m < n$ erfüllen $q = 0$ und $r = m$ genau die gewünschten Eigenschaften. Im Fall $m \geq n$ betrachten wir die Menge

$$M := \{m - nk \mid k \in \mathbb{N}_0\} \cap \mathbb{N}_0.$$

Wegen $m \in M$ ist $M \neq \emptyset$, und wegen der Wohlordnungseigenschaft[2] von $\mathbb{N}_0$ hat M daher ein kleinstes Element r. Insbesondere existiert ein $q \in \mathbb{N}_0$, so dass $r = m - nq$ bzw. $m = nq + r$ gilt. Wäre nun $r \geq n$, so wäre

$$\tilde{r} := m - n(q+1) = m - nq - n = r - n \geq 0 \tag{1.4}$$

und damit $\tilde{r} \in M$. Dies steht aber wegen $\tilde{r} = r - n < r$ im Widerspruch zur Minimalität von r. Daher muss gelten $r < n$.
Zum Nachweis der Eindeutigkeit nehmen wir an, dass sowohl q,r als auch $\bar{q},\bar{r}$ Quotient und Rest von m und n sind, also $nq + r = m = n\bar{q} + \bar{r}$. Dann gilt auch

$$n|q - \bar{q}| = |\bar{r} - r|,$$

was wegen $|\bar{r} - r| < n$ und $n \neq 0$ nur mit $|q - \bar{q}| = 0$ und $|\bar{r} - r| = 0$ erfüllt sein kann. Daher ist $q = \bar{q}$ und $r = \bar{r}$. □

Der *Quotient* q und der *Rest* r bei Division von m durch n sind also eindeutig bestimmt, wir schreiben dafür m div n bzw. m mod n. Auch im Beispiel der Division kann also das Problem durch eine Abbildung φ beschrieben werden, jedoch ist sie vorerst nur *implizit* durch

$$\varphi(m,n) := \text{ diejenigen } q,r \in \mathbb{N}_0 \text{ mit } m = nq + r \wedge r < n \tag{1.5}$$

gegeben.

Von der Problemspezifikation zum Algorithmus

Im Falle der eindeutigen Lösbarkeit des Problems ist die Lösung der Probleminstanz (φ,x) also durch $\varphi(x)$ beschrieben. Im Sinne einer *algorithmischen Betrachtung* ist aber das Problem durch die Angabe von φ noch nicht gelöst, da durch (1.2) im Allgemeinen nicht erklärt wird, wie $\varphi(x)$ tatsächlich berechnet werden kann, siehe etwa (1.5). Die algorithmische Lösung eines Problems beinhaltet zunächst die Angabe einer Rechenvorschrift zur Bestimmung von $\varphi(x)$ für jede Probleminstanz. Diese Rechenvorschrift bezeichnet man als *Algorithmus*, wenn sie eindeutig festlegt, wie aus den Eingangsdaten des Problems dessen Lösung in endlich vielen Schritten ermittelt werden kann. Im Gegensatz zu anderen mathematischen Begriffen wie

[2] Eine geordnete Menge heißt *wohlgeordnet* genau dann, wenn jede nicht-leere Teilmenge ein kleinstes Element besitzt.

Vektorraum oder Grenzwert existiert keine einheitliche Definition des Begriffs Algorithmus, so dass wir uns mit folgender Begriffsbildung begnügen.

Begriffsbildung (Algorithmus). Ein *Algorithmus* ist eine *präzise Vorschrift* zur Durchführung einer *endlichen Folge* von *Elementaroperationen,* um Probleme einer *bestimmten Klasse* zu lösen.

Dieser Algorithmenbegriff hebt die für dieses Buch wesentlichen Aspekte der Umsetzbarkeit von Algorithmen auf einem Computer hervor[3]. Dabei bedeutet der Begriff der präzisen Vorschrift, dass ein Algorithmus den Ablauf der einzelnen Schritte der Rechenvorschrift derart genau festlegt, dass diese in unmissverständlicher Weise in ein Computerprogramm umsetzbar ist. Die Elementaroperationen sind die Bausteine zur Entwicklung eines neuen Algorithmus. Von ihnen verlangen wir zunächst lediglich, dass sie in endlicher Zeit am Computer ausführbar sind. Da ein Algorithmus nur aus einer endlichen Folge von Elementaroperationen besteht, ist damit garantiert, dass ein Algorithmus *terminiert,* d.h. nach endlicher Zeit ein Resultat liefert. Algorithmen lösen typischerweise nicht nur einzelne Probleminstanzen, sondern sind durch Verwendung von Eingabevariablen auf eine ganze Klasse von Problemstellungen anwendbar. Diese Problemklasse legen wir in der zum Algorithmus gehörigen Spezifikation fest.

Eine explizite Darstellung von φ erlaubt es unmittelbar, einen Algorithmus zur Berechnung der Lösung anzugeben. Im Falle der quadratischen Gleichung gelangen wir mit (1.3) unter Berücksichtigung aller Zwischenschritte zum Algorithmus *QuadGlgI*.

Algorithmus *QuadGlgI*: Quadratische Gleichung, Variante I

$\xi_1 \leftarrow p \cdot p$	Aufruf: $QuadGlgI(p,q)$
$\xi_2 \leftarrow \xi_1 - q$	Eingabe: $p,q \in \mathbb{R}$
$\xi_3 \leftarrow \sqrt{\xi_2}$	mit: $p^2 - q \geq 0$.
$x \leftarrow p - \xi_3$	Ausgabe: $x \in \mathbb{R}$
return x	mit: $x^2 - 2px + q = 0$ und $x \leq p$.

Der Algorithmus kann also als Auswertung der Abbildung φ aus (1.3) an den Stellen p und q aufgefasst werden. Aufgebaut ist der Algorithmus auf den auf $\mathbb{R}$ bzw. $\mathbb{R}^+$ definierten Elementaroperationen $\cdot$, $-$ bzw. $\sqrt{\cdot}$, klarerweise terminiert er nach 4 Schritten.

Auch bei Eindeutigkeit der Lösung können unterschiedliche Algorithmen zu deren Berechnung existieren. So legt etwa die auf dem Wurzelsatz von Vieta[4] basierende Umformung

$$\varphi(p,q) = p - \sqrt{p^2 - q} = (p - \sqrt{p^2 - q}) \cdot \frac{p + \sqrt{p^2 - q}}{p + \sqrt{p^2 - q}} = \frac{q}{p + \sqrt{p^2 - q}} \tag{1.6}$$

[3] Für theoretische Untersuchungen, z.B. in der Berechenbarkeitstheorie, ist eine detailliertere Begriffsbildung notwendig.

[4] Vieta (Viète), Francois: 1540–1603, französischer (Hobby-)Mathematiker und Astronom. Auf ihn geht die Verwendung von Buchstaben für Variablen in der mathematischen Notation zurück.

den alternativen Algorithmus *QuadGlgII* nahe. Es führen dann sowohl der Aufruf[5] $QuadGlgI(p,q)$ als auch $QuadGlgII(p,q)$ auf den Wert $\varphi(p,q)$. Dennoch handelt es sich um verschiedene Algorithmen, da sie aus verschiedenen Folgen von Elementaroperationen zusammengesetzt sind.

Algorithmus *QuadGlgII*: Quadratische Gleichung, Variante II

$\xi_1 \leftarrow p \cdot p$
$\xi_2 \leftarrow \xi_1 - q$
$\xi_3 \leftarrow \sqrt{\xi_2}$
$\xi_4 \leftarrow p + \xi_3$
$x \leftarrow \frac{q}{\xi_4}$
return x

Aufruf: $QuadGlgII(p,q)$
Eingabe: $p,q \in \mathbb{R}$
mit: $p^2 - q \geq 0$.
Ausgabe: $x \in \mathbb{R}$
mit: $x^2 - 2px + q = 0$ und $x \leq p$.

Im Falle einer impliziten Darstellung von φ wie in (1.5) ist die Erstellung einer Vorschrift zur Berechnung der Lösung $\varphi(x)$ nicht offensichtlich. Liegt jedoch ein *konstruktiver Existenzbeweis* vor, so lässt sich daraus stets ein Algorithmus ableiten. Auch im Beispiel der Division mit Rest steckt die Idee für einen Divisionsalgorithmus im Existenzbeweis. Nach (1.4) lässt sich die Darstellung $m = nq + r$ mit $r \geq n$ stets auf $m = n\tilde{q} + \tilde{r}$ mit $0 \leq \tilde{r} < r$ überführen, nämlich durch $\tilde{q} = q + 1$ und $\tilde{r} = r - n$. Die Idee zur Berechnung der Lösung des Divisionproblems ist, mit der trivialen Darstellung $m = n \cdot 0 + m$, also $q = 0$ und $r = m$, zu beginnen, und solange noch $r \geq n$ gilt, q durch $q + 1$ und r durch $r - n$ zu ersetzen. Wir gelangen so zum Algorithmus *QuotRestN*.

Algorithmus *QuotRestN*: Division mit Rest in $\mathbb{N}$

$q \leftarrow 0, r \leftarrow m$
while $r \geq n$
  $q \leftarrow q + 1$
  $r \leftarrow r - n$
return (q,r)

Aufruf: $QuotRestN(m,n)$
Eingabe: $m,n \in \mathbb{N}$.
Ausgabe: $q,r \in \mathbb{N}_0$
mit: $m = nq + r$ und $r < n$.

Dieser Algorithmus basiert auf den auf $\mathbb{N}$ definierten Elementaroperationen $+$ und $-$ sowie der Vergleichsoperation $\geq$. Die Anzahl der auszuführenden Rechenschritte ist hier nicht unmittelbar ersichtlich, wir werden uns jedoch auf Seite 36 davon überzeugen, dass auch dieser Algorithmus stets terminiert. Der Algorithmus erlaubt nun eine explizite Angabe der in (1.5) nur implizit definierten Abbildung φ, nämlich

$$\varphi(m,n) := QuotRestN(m,n).$$

Algorithmen präsentieren wir wie in den bisherigen Beispielen gezeigt mit Hilfe einer Ein-/Ausgabespezifikation, die neben dem Namen des Algorithmus die erwarteten Eingabeparameter sowie das zu berechnende Resultat festlegt. Die Spezifikation beschreibt, *was* der Algorithmus berechnen soll und was er dafür an Input benötigt.

[5] Zum Aufruf eines Algorithmus *Algo* mit Eingabe x verwenden wir den Namen mit den Parametern in runden Klammern, also $Algo(x)$.

Die Ein-/Ausgabespezifikation des Algorithmus ist stets an jener des Problems angelehnt, muss aber nicht zwangsläufig mit dieser übereinstimmen, für ein Beispiel verweisen wir auf Seite 11. *Wie* der Algorithmus im Detail arbeitet, beschreiben wir in Form von *Pseudocode*, der sich an Sprachmittel gängiger Programmiersprachen anlehnt. Beispiele dafür sind Wertzuweisungen[6], Aufrufe von Unteralgorithmen, if-then-else Verzweigungen oder Schleifen für Teile, die wiederholt auszuführen sind. Den Gültigkeitsbereich einer Verzweigung oder Schleife markieren wir durch Einrückung des Textes, die Rückgabewerte eines Algorithmus sind immer durch „return" gekennzeichnet. Der von uns gewählte Pseudocode ist jedenfalls eine präzise Vorschrift, durch den Verzicht auf den Formalismus einer echten Programmiersprache ist er leichter verständlich.

Mit der Entwicklung und Beschreibung eines Algorithmus mittels Pseudocode ist das gestellte Problem aber noch immer nicht gelöst. Praktischen Nutzen erhält der Algorithmus nur, wenn er in ein lauffähiges Computerprogramm umgesetzt wird. Dies erfordert die Kenntnis und Verwendung einer konkreten Programmiersprache. In diesem Buch bilden die Programmpakete *Mathematica* und MATLAB die Grundlage für die *Realisierung* der vorgestellten *Algorithmen*.

2 Einführende Beispiele zur algorithmischen Lösung am Computer

In diesem Abschnitt wollen wir einfach zu formulierende Aufgabenstellungen mit Hilfe von Algorithmen tatsächlich *am Computer* lösen. Anhand von Beispielen stellen wir wichtige Zugänge und Phänomene einer algorithmischen Mathematik vor, dazu zählen iterative Näherungsverfahren, diskretisierte Ersatzprobleme und die Umsetzung einer Lösungsidee in einen Algorithmus mittels Rekursion und Schleife. Dabei werden wir auch beobachten, dass bei der Durchführung von Algorithmen am Computer *Fehler* entstehen können, und die tatsächliche Rückgabe dann nicht der gesuchten Lösung des Problems entspricht. Allgemein verwendet man über den reellen Zahlen $\mathbb{R}$ folgende *Fehlerbegriffe*.

Definition

Absoluter und relativer Fehler. Sei $\tilde{z} \in \mathbb{R}$ eine Näherung von $z \in \mathbb{R}$. Dann nennt man

$$\tilde{z} - z \tag{2.7}$$

den *absoluten* Fehler von $\tilde{z}$. Ist $z \neq 0$, so ist

$$\varepsilon_{\tilde{z},z} = \frac{\tilde{z} - z}{z} \tag{2.8}$$

der *relative* Fehler von $\tilde{z}$.

[6] In vielen Programmiersprachen wird = für die Wertzuweisung verwendet. Wir schreiben allerdings $\leftarrow$, um die Wertzuweisung klar von der mathematischen Gleichheit zu unterscheiden.

Der Zusammenhang (2.8) zwischen $\tilde{z}$ und z kann auch durch

$$\tilde{z} = z(1 + \varepsilon_{\tilde{z},z})$$

formuliert werden. Damit ausgestattet beginnen wir unsere Einführung mit dem bereits erwähnten Problem der Lösung einer quadratischen Gleichung.

Symbolisches und numerisches Rechnen

Beispiel

Wir suchen die kleinere Lösung der quadratischen Gleichung (1.1) mit den Eingabewerten $p = 3$ und $q = 7$. Der Aufruf `QuadGlgI[3,7]` in *Mathematica*[7] liefert das Ergebnis $3 - \sqrt{2}$, jener von `QuadGlgII[3,7]` ergibt $\frac{7}{3+\sqrt{2}}$. Rufen wir in *Mathematica* die Algorithmen mittels `QuadGlgI[3.,7.]` und `QuadGlgII[3.,7.]`, so erhalten wir beide Male 1.58579 als Resultat[8].

In *Mathematica* spielt es offenbar eine Rolle, ob wir den Algorithmus mit den ganzen Zahlen 3 und 7 oder den Kommazahlen 3. und 7. aufrufen. Durch `QuadGlgI[3,7]` wird ein *symbolischer* Zugang, durch `QuadGlgI[3.,7.]` hingegen ein *numerischer* Zugang gewählt. Beim *symbolischen Rechnen*, zu dem Computeralgebrasysteme wie *Mathematica* in der Lage sind, wird das übliche Rechnen mit Formeln und Unbekannten automatisiert. Das Rechnen mit Zahlen basiert hier auf rationalen Zahlen, die den gängigen arithmetischen Operationen $+, -, \cdot, /$ unterliegen. Die dabei verwendete *rationale Arithmetik* stellen wir in Abschnitt 7 näher vor. Da $\sqrt{2} \notin \mathbb{Q}$ ist, wird $3 - \sqrt{2}$ beim Aufruf `QuadGlgI[3,7]` nicht weiter verarbeitet und bleibt als symbolischer Ausdruck im Resultat erhalten.

Beim *numerischen Rechnen* wird dagegen die Lösung des Problems in Form von Kommazahlen gesucht. Da $3 - \sqrt{2}$ eine irrationale Zahl ist, kann es sich bei 1.58579, also einer Zahl mit endlich vielen Nachkommastellen, nicht um die exakte Lösung des Problems handeln. Als vorläufige Erklärung soll genügen, dass nicht alle reellen Zahlen auf einem Rechner exakt darstellbar sind. Jene endlich vielen Zahlen, für die es eine exakte Darstellung gibt, nennen wir *Maschinenzahlen*. Auch die auf $\mathbb{R}$ definierten arithmetischen Operationen $+, -, \cdot, /$ können am Rechner nicht immer exakt durchgeführt werden. Selbst wenn es sich bei den Operanden um Maschinenzahlen handelt, muss das Ergebnis zunächst nicht zwangsläufig in der Menge der Maschinenzahlen liegen. Der gängige Zugang zum Rechnen mit reellen Zahlen am Computer ist die *Gleitkommaarithmetik*, die wir für alle numerischen Berechnungen im Buch heranziehen und in Abschnitt 8 näher erläutern.

Beispiel

Obiges Beispiel können wir natürlich auch in MATLAB behandeln. Die Aufrufe `QuadGlgI(3,7)`, `QuadGlgI(3.,7.)`, `QuadGlgII(3,7)` und `QuadGlgII(3.,7.)` liefern[9] alle das numerische Ergebnis 1.5858.

Im Gegensatz zu *Mathematica* spielt es in MATLAB keine Rolle, ob wir als Input 3 und 7 oder 3. und 7. verwenden. In beiden Fällen wird numerisch gerechnet, sym-

[7] Die Verwendung eckiger Klammern weist auf eine Ausführung des Algorithmus in *Mathematica* hin.
[8] Wir erinnern an die Vereinbarung, Computerergebnisse stets in blauer Schrift anzuführen.
[9] Die Verwendung runder Klammern weist auf eine Ausführung des Algorithmus in MATLAB hin.

bolische Berechnungen erfordern in MATLAB die Zuhilfenahme der Symbolic Toolbox.

Ein Grund für den Unterschied in unseren bisherigen numerischen Resultaten 1.58579 und 1.5858 ist, dass numerische Ergebnisse in unterschiedlichen Formaten ausgegeben werden können.

Beispiel

Das durch `QuadGlgI(3,7)` in MATLAB berechnete Ergebnis kann mit Hilfe entsprechender Format-Befehle auf unterschiedliche Weise ausgegeben werden. Der Standard in MATLAB verwendet 5 Ziffern in der Ausgabe, daher das Resultat 1.5858. Auch in *Mathematica* gibt es natürlich unterschiedliche Ausgabeformate. Dort liegt der Standard bei 6 Ziffern, daher also 1.58579.

Für die Ausgabe von Maschinenzahlen im Buch wählen wir stets ein Format, welches im Zusammenhang mit der jeweiligen Diskussion die wichtige Information vermittelt und zugleich die Lesbarkeit erhält.

Der Umstand, dass bei der numerischen Realisierung eines Algorithmus stets ein Fehler aufgrund des Computermodells von $\mathbb{R}$ entstehen kann, mag den Eindruck erwecken, dass symbolischen Methoden immer der Vorzug zu geben ist. Dies ist jedoch nicht der Fall, beide Zugänge zum Rechnen am Computer haben Vorteile und Nachteile. Symbolisches und numerisches Rechnen stehen nicht in Konkurrenz zueinander, vielmehr kann bei anspruchsvollen Problemen oft ein kombinierter Einsatz erforderlich werden.

Iterative Algorithmen

Wir wenden uns nun der Idee zu, die Lösung eines Problems mit Hilfe eines *iterativen Algorithmus* Schritt für Schritt anzunähern. Dieser Zugang ist insofern von zentraler Bedeutung, als dass viele Aufgabenstellungen am Computer nur näherungsweise gelöst werden können. Zur Veranschaulichung betrachten wir die Berechnung der Quadratwurzel einer positiven reellen Zahl r.

Problemstellung (Wurzel aus r).

Gegeben: $r \in \mathbb{R}^+$
Gesucht: $x \in \mathbb{R}^+$
mit: $x^2 = r$.

Steht uns das Wurzelziehen als Elementaroperation zur Verfügung, in *Mathematica* und MATLAB lauten die entsprechenden Befehle `Sqrt` und `sqrt`, ist die Angabe eines Lösungsalgorithmus natürlich trivial. Im Beispiel $r = 2$ erhält man so durch Aufruf von `Sqrt[2]` eine symbolische, durch Aufruf von `Sqrt[2.]` oder `sqrt(2)` eine numerische Lösung. Hinter dem (numerischen) Wurzelbefehl steckt aber in der Tat ein iterativer Algorithmus, dessen Grundgedanken wir nun vorstellen.

Dazu betrachten wir zunächst die Beziehung

$$\sqrt{r} = \frac{1}{2}\left(\sqrt{r} + \frac{r}{\sqrt{r}}\right) \tag{2.9}$$

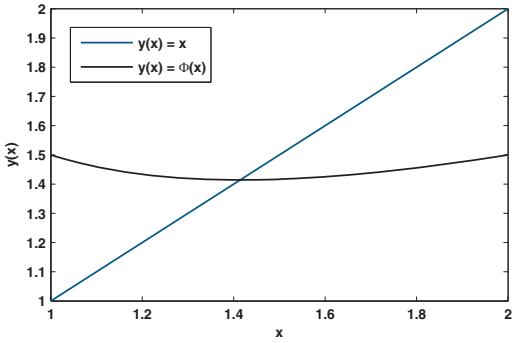


Abb. I.1. Graphische Lösung der Gleichung $x = \Phi(x)$ in $\mathbb{R}^+$ mit $r = 2$.

und stellen fest, dass $\sqrt{r}$ eine Lösung der Gleichung

$$x = \Phi(x) \tag{2.10}$$

mit

$$\Phi(x) = \frac{1}{2}\left(x + \frac{r}{x}\right) \tag{2.11}$$

ist. Da man bei Auswertung von Φ an der Stelle $\sqrt{r}$ wieder $\sqrt{r}$ erhält, ist $\sqrt{r}$ ein *Fixpunkt* der Abbildung (2.11). Die Gleichung (2.10) wird *Fixpunktgleichung* genannt, graphisch kann ihre Lösung durch Schnitt der Geraden $y(x) = x$ mit der Kurve $y(x) = \Phi(x)$ ermittelt werden, siehe Abbildung I.1 für den Fall $r = 2$. Zur numerischen Berechnung wählen wir einen beliebigen Startpunkt $x_0 > 0$ und definieren die Iterationsfolge $(x_n)_{n\in\mathbb{N}_0}$ durch

$$x_{n+1} = \Phi(x_n) \quad \text{für } n = 0,1,2,\ldots \tag{2.12}$$

Die Idee dieser *Fixpunktiteration* ist, solange Folgenglieder zu bestimmen, bis sich bei der Anwendung von Φ nichts mehr ändert, und der Fixpunkt erreicht ist. Um zu sehen, ob dieser Zugang auch tatsächlich zum Ziel führt, überzeugt man sich zunächst mit Hilfe eines Induktionsarguments davon, dass alle x_n positiv sind und die Folge wohldefiniert ist. Zudem gilt nach der Ungleichung zwischen dem geometrischen und arithmetischen Mittel

$$\sqrt{r} = \sqrt{x_n \cdot \frac{r}{x_n}} \leq \frac{x_n + \frac{r}{x_n}}{2} = \Phi(x_n) = x_{n+1},$$

also folgt $r \leq x_n^2$ für $n \geq 1$. Mit $\frac{r}{x_n} \leq x_n$ ergibt sich aus (2.12) und (2.11) dann $x_{n+1} \leq x_n$ für $n \geq 1$. Die Folge ist also monoton fallend und durch $\sqrt{r}$ nach unten beschränkt. Daher muss sie einen Grenzwert $\bar{x}$ besitzen, dem die Glieder x_n mit wachsendem n immer näher kommen, wofür man auch

$$\lim_{n\to\infty} x_n = \bar{x} \tag{2.13}$$

schreibt. Da Φ stetig ist, folgt mit $n \to \infty$ in (2.12), dass $\bar{x} = \Phi(\bar{x})$ und damit $\bar{x}^2 = r$ gilt. Da alle Folgenglieder größer gleich $\sqrt{r}$ sind, erhält man schließlich $\bar{x} = \sqrt{r}$. Die

Vorschrift (2.12) zur schrittweisen Annäherung des Grenzwertes $\sqrt{r}$ kann nun im iterativen Algorithmus *ItWurzelI* umgesetzt werden.

Algorithmus *ItWurzelI*: Iterative Näherung von $\sqrt{r}$, Variante I

```
x ← x0
while A nicht erfüllt
   x ← ½(x + r/x)
return x
```

Aufruf: $ItWurzelI(r, x_0)$
Eingabe: $r, x_0 \in \mathbb{R}^+$
Ausgabe: $x \in \mathbb{R}^+$
mit: x als Näherung von $\sqrt{r}$, die vom Startwert x_0 und dem Abbruchskriterium $\mathcal{A}$ abhängt.

Der Algorithmus basiert auf den auf $\mathbb{R}$ definierten Elementaroperationen + und /, erst durch ihn wird das Wurzelziehen auf $\mathbb{R}^+$ zur Elementaroperation. Der Begriff der Elementaroperation ist also dehnbar, in Kapitel II werden wir sehen, dass sich selbst hinter den Grundrechenarten Algorithmen verbergen, die ihrerseits auf noch einfacheren Operationen basieren.

Das Konvergenzresultat (2.13) garantiert, dass $\sqrt{r}$ – unter Vernachlässigung etwaiger Fehler aufgrund des Computermodells von $\mathbb{R}$ – mit Hilfe des Algorithmus *ItWurzelI* beliebig angenähert werden kann. Der exakte Wert von $\sqrt{r}$ kann damit aber in der Regel nicht berechnet werden, da der Algorithmus ja nicht unendlich lange laufen kann. Der Algorithmus liefert also nicht exakt die gesuchte Lösung der Problemspezifikation, was sich auch darin widerspiegelt, dass die Ausgabebedingung des Algorithmus nicht mit jener der Problemstellung übereinstimmt. Zusätzlich weicht auch die Eingabe des Algorithmus von jener der Problemspezifikation durch Hinzunahme des Startwertes x_0 ab.

Der Abbruch der Schleife nach endlich vielen Schritten wird durch ein *Abbruchskriterium* $\mathcal{A}$ bewerkstelligt. Dieses terminiert den Algorithmus beispielsweise, wenn eine gegebene Anzahl von Schleifendurchläufen erreicht wird, und/oder der Betrag der Differenz zweier aufeinander folgender Näherungswerte unter eine gegebene Schranke fällt, siehe auch Seite 13.

Beispiel

Wir betrachten die quadratische Gleichung $x^2 = 2$ in $\mathbb{R}^+$, für die die Lösung nach Abbildung I.1 etwa bei 1.4 liegt. Um die Stärke des Verfahrens (2.12) zu demonstrieren, wählen wir dennoch den vergleichsweise schlechten Startwert $x_0 = 1$ im Algorithmenaufruf `ItWurzelI(2,1)`. Brechen wir den Algorithmus nach 5 Schleifendurchläufen ab, so erhalten wir damit bei Ausgabe aller Zwischenergebnisse[10]

$$\begin{aligned} x^{(1)} &= 1.5 \\ x^{(2)} &= 1.416666666666667 \\ x^{(3)} &= 1.414215686274510 \\ x^{(4)} &= 1.414213562374690 \\ x^{(5)} &= 1.414213562373095. \end{aligned}$$

[10] Hier steht $x^{(i)}$ für den Wert der Variable x am Ende des i-ten Schleifendurchlaufs. Diese Notation ist erforderlich, weil im Algorithmus der Wert von x in jedem Schleifendurchlauf überschrieben wird.

Wie oben gezeigt, ist ab $x^{(1)}$ die Folge monoton fallend, zudem unterscheiden sich die Iterierten mit zunehmendem Iterationsindex in immer weniger Dezimalstellen voneinander, $x^{(5)}$ stimmt dann bereits mit jenem Wert überein, den man auch bei Aufruf `sqrt(2)` erhält.

Iterative Algorithmen, die auf einem theoretischen Konvergenzresultat beruhen, werden in vielen Bereichen der Mathematik zur Lösung von Problemen eingesetzt. Oft handelt es sich dabei um Fixpunktiterationen zur Lösung von Fixpunktgleichungen. Iterative Verfahren zur Näherung von Grenzwerten müssen stets nach endlich vielen Schritten abgebrochen werden, so dass mit einem *Abbruchsfehler* zu rechnen ist. Bei Durchführung eines iterativen Verfahrens mit Maschinenzahlen ist zusätzlich die Möglichkeit eines Rechenfehlers aufgrund des Computermodells von $\mathbb{R}$ zu beachten.

Abbruchkriterien sollen neben ihrer eigentlichen Aufgabe natürlich auch sicherstellen, dass der Abbruchsfehler hinreichend klein ist, und damit die Ausgabe des Iterationsalgorithmus eine akzeptable Näherung an den gesuchten Grenzwert darstellt. Das Auffinden geeigneter Kriterien erfordert meist weitere theoretische Untersuchungen des Konvergenzverhaltens. Nicht nur in diesem Zusammenhang spielt der *Banachsche* Fixpunktsatz[11] eine zentrale Rolle, in $\mathbb{R}$ lautet er wie folgt.

Satz

Banachscher Fixpunktsatz. Seien $a,b \in \mathbb{R}$ mit $a < b$ und $\Phi : [a,b] \to [a,b]$. Falls Φ für alle $x,y \in \mathbb{R}$ die Lipschitzbedingung

$$|\Phi(x) - \Phi(y)| \leq L|x - y| \tag{2.14}$$

mit einer Lipschitzkonstanten $L < 1$ erfüllt, existiert genau ein Fixpunkt $\bar{x}$ in $[a,b]$ mit $\Phi(\bar{x}) = \bar{x}$. Weiters konvergiert für alle $x_0 \in [a,b]$ die durch $x_{n+1} = \Phi(x_n)$ definierte Folge gegen $\bar{x}$, und es gilt die Abschätzung

$$|x_n - \bar{x}| \leq \frac{L^n}{1-L}|x_0 - x_1|. \tag{2.15}$$

Beweis. Zu Beginn zeigen wir, dass die Folge $(x_n)_{n\in\mathbb{N}_0}$ eine Cauchy[12]-Folge ist, woraus dann die Konvergenz aufgrund der Abgeschlossenheit des Intervalls $[a,b]$ in $\mathbb{R}$ folgt. Es gilt

$$|x_n - x_{n+1}| = |\Phi(x_{n-1}) - \Phi(x_n)| \leq L|x_{n-1} - x_n|,$$

und die wiederholte Anwendung des Arguments führt auf

$$|x_n - x_{n+1}| \leq L^n|x_0 - x_1|.$$

[11] Banach, Stefan: 1892–1945, polnischer Mathematiker. In seiner Dissertation definierte er axiomatisch den Begriff „Banachraum“, die Namensgebung stammt allerdings von Fréchet. Andere von ihm bewiesene Sätze sind etwa der Satz von Hahn-Banach oder der Satz von Banach-Steinhaus.

[12] Cauchy, Augustin Louis: 1789–1857, französischer Mathematiker. Einer der Pioniere der Analysis, von ihm stammt beispielsweise die ϵ-Definition des Grenzwerts. Mit ca. 700 publizierten Arbeiten, darunter auch fundamentale Arbeiten zur Algebra und mathematischen Physik, hat er ein besonders umfangreiches Werk.

Daraus folgt für $n < l$ die Abschätzung

$$\begin{aligned} |x_n - x_l| &\leq |x_n - x_{n+1}| + |x_{n+1} - x_{n+2}| + \cdots + |x_{l-1} - x_l| \\ &\leq L^n|x_0 - x_1| + L^{n+1}|x_0 - x_1| + \cdots + L^{l-1}|x_0 - x_1| \\ &= \left(L^n + L^{n+1} + \cdots + L^{l-1}\right)|x_0 - x_1| \\ &\leq \left(L^n + L^{n+1} + \cdots + L^{l-1} + L^l + \ldots\right)|x_0 - x_1| \\ &= \frac{L^n}{1-L}|x_0 - x_1|. \end{aligned} \tag{2.16}$$

Wegen $0 \leq L < 1$ ist daher das Cauchy-Kriterium $|x_n - x_l| \to 0$ für $n, l \to \infty$ erfüllt, und die Folge $(x_n)_{n\in\mathbb{N}_0}$ konvergiert gegen einen Grenzwert $\bar{x} \in [a,b]$. Aufgrund der Lipschitzbedingung (2.14) ist die Abbildung Φ stetig, also gilt

$$\bar{x} = \lim_{n\to\infty} x_n = \lim_{n\to\infty} \Phi(x_{n-1}) = \Phi(\lim_{n\to\infty} x_{n-1}) = \Phi(\bar{x}).$$

Der Grenzwert $\bar{x}$ ist somit ein Fixpunkt von Φ. Nehmen wir nun an, es gäbe einen weiteren Fixpunkt $\hat{x} \in [a,b]$. Dann gilt aber

$$|\hat{x} - \bar{x}| = |\Phi(\hat{x}) - \Phi(\bar{x})| \leq L|\hat{x} - \bar{x}|,$$

was wegen $0 \leq L < 1$ nur mit $\hat{x} = \bar{x}$ erfüllbar ist. Der Fixpunkt $\bar{x}$ ist daher eindeutig in $[a,b]$. Die Abschätzung (2.15) folgt schließlich aus (2.16) mit $l \to \infty$. □

Der Satz liefert also nicht nur Abschätzung (2.15) für den *absoluten* Fehler $x_n - \bar{x}$, sondern garantiert auch die eindeutige Lösbarkeit der Fixpunktgleichung sowie die Konvergenz der Fixpunktiterierten. Die Abschätzung (2.15) informiert auch über die *Konvergenzgeschwindigkeit* der Folge und besagt, dass x_n mindestens so schnell gegen $\bar{x}$ konvergiert wie die geometrische Folge $\left(\frac{L^n}{1-L}|x_0 - x_1|\right)_{n\in\mathbb{N}_0}$ gegen null konvergiert. In unserem Beispiel (2.11) mit $r = 2$ sind für das Interval $[1,2]$ die Voraussetzungen mit $L = 1/2$ erfüllt, siehe Übungsaufgabe I.4, zum Nachweis der Konvergenz (2.13) hätte also einfach ein Verweis auf Banach genügt.

Algorithmus *ItWurzelII* : Iterative Näherung von $\sqrt{r}$, Variante II

```
x1 ← ½(x0 + r/x0)
n ← 1, x ← x1
while (L^n/(1−L))|x1 − x0| > η
   x ← ½(x + r/x)
   n ← n + 1
return x
```

Aufruf: *ItWurzelII*(r, x_0, η)
Eingabe: $r, x_0, \eta \in \mathbb{R}^+$.
Ausgabe: $x \in \mathbb{R}^+$
mit: $|x - \sqrt{r}| \leq \eta$.

Im Algorithmus *ItWurzelII* verwenden wir die Abschätzung (2.15) für eine Umsetzung des oben allgemein gehaltenen Abbruchskriteriums $\mathcal{A}$. Dabei ist neben r und x_0 auch die Eingabe der gewünschten Genauigkeit η erforderlich. Diese wird wegen der Fehlerabschätzung (2.15) nach endlich vielen Schritten erreicht. Bei L handelt es sich um die Lipschitzkonstante aus (2.14) für Φ wie in (2.11) definiert. Der

Algorithmus *ItWurzelII* berechnet eine *Näherung* an $\sqrt{r}$ mit vorgegebener Genauigkeit, genau genommen löst er die folgende, modifizierte Aufgabenstellung[13].

Problemstellung (Näherung an Wurzel aus r).

Gegeben: $r, x_0, \eta \in \mathbb{R}^+$.
Gesucht: $x \in \mathbb{R}^+$
mit: $|x - \sqrt{r}| \leq \eta$.

Nicht allen iterativen Verfahren in der Mathematik liegt eine Grenzwertüberlegung samt damit verbundener Abbruchsfehlerproblematik zugrunde. Auch der Divisionsalgorithmus von Seite 6 berechnet Quotient und Rest Schritt für Schritt in einer Schleife, ist also iterativer Natur[14]. Auf Seite 36 werden wir aber zeigen, dass dieser Iterationsalgorithmus nach endlich vielen Schritten ohne Abbruchsfehler zur Lösung führt.

Rundungsfehler und deren Fortpflanzung

Gegenstand dieses Abschnitts ist der bereits erwähnte Rechenfehler bei Verwendung des auf Maschinenzahlen basierenden Computermodells von $\mathbb{R}$.

Beispiel

Wir suchen die kleinere Lösung der quadratischen Gleichung (1.1) mit den Eingabewerten $p = 5 \cdot 10^{11} - \frac{1}{20}$ und $q = -10^{11}$. Der Aufruf `QuadGlgII(`$5 \cdot 10^{11} - \frac{1}{20}, -10^{11}$`)` liefert in MATLAB als Ergebnis -0.100000000000000. Wegen

$$x^2 - 2 \cdot (5 \cdot 10^{11} - \frac{1}{20})x - 10^{11} = (x + \frac{1}{10})(x - 10^{12}) \qquad (2.17)$$

sehen wir, dass $-0.1 = -\frac{1}{10}$ auch tatsächlich die Lösung des gestellten Problems ist. Auch das Rechnen mit Maschinenzahlen kann also zur exakten Lösung des Problems führen.

Beispiel

Die Verwendung des Algorithmus `QuadGlgI` führt mit `QuadGlgI(`$5 \cdot 10^{11} - \frac{1}{20}, -10^{11}$`)` in MATLAB jedoch auf das falsche Resultat -0.099975585937500. In *Mathematica* erhalten wir durch `QuadGlgI[`$5 \cdot 10^{11} - \frac{1}{20}, -10^{11}$`]` die rationale Zahl $-\frac{1}{10}$ als Ergebnis.

Bei der numerischen Lösung eines Problems ist also in der Regel mit einem fehlerhaften Ergebnis $\tilde{x}$ anstelle der gesuchten Größe x zu rechnen. Obige Beispiele zeigen auch, dass mathematisch äquivalente Ausdrücke wie zum Beispiel $p - \sqrt{p^2 - q}$ und $\frac{q}{p + \sqrt{p^2 - q}}$ beim Rechnen am Computer mit Maschinenzahlen nicht zwangsläufig zum selben Ergebnis führen müssen. Wie bereits erwähnt, ist die Ursache für all dies, dass die Menge $\mathbb{R}$ und die darauf definierten Operationen auf einem Computer mit Hilfe von Maschinenzahlen nur unvollständig modelliert werden können.

[13] Die Spezifikation des modifizierten Problems stimmt mit jener des Algorithmus *ItWurzelII* überein.

[14] In der Informatik nennt man jedes auf einer Schleife beruhende Verfahren einen iterativen Algorithmus. In der numerischen Mathematik ist die Bezeichnung iteratives Verfahren in erster Linie für solche reserviert, denen ein Grenzwertargument zugrunde liegt.

Beispiel

Das gängige Modell für das Rechnen in $\mathbb{R}$ mit Gleitkommaarithmetik ist das *IEEE double precision* Modell, das als Standard in MATLAB in all unseren bisherigen numerischen Beispielen insgeheim verwendet wurde. Wir stellen es in Abschnitt 8 näher vor. Die größte darin verfügbare reelle Zahl kann in MATLAB durch den Befehl `realmax` aufgerufen werden und liegt bei $1.797693134862316 \cdot 10^{308}$. Während der Aufruf `1.1 * 5.0` auf (das exakte Ergebnis) 5.5 führt, erhält man bei `1.1 * realmax` jedoch Inf, das IEEE Symbol für ∞. Ein anderes Computermodell für $\mathbb{R}$ ist das *IEEE single precision* Modell. Darin ist die größte darstellbare Zahl in MATLAB durch $3.4028235 \cdot 10^{38}$ gegeben. Die kleinste verfügbare reelle Zahl erhält man jeweils mit `-realmax`.

Neben oberen und unteren Schranken gibt es beim Rechnen mit reellen Zahlen eine weitere wesentliche Einschränkung. Die am Computer darstellbaren Maschinenzahlen bilden nur eine *diskrete* Teilmenge von $\mathbb{R}$, zwischen den am Rechner darstellbaren reellen Zahlen gibt es also stets Lücken. Der Abstand[15] zwischen 1.0 und der nächst größeren im Modell darstellbaren Zahl wird *relative Maschinengenauigkeit* (machine epsilon) genannt und mit **eps** bezeichnet.

Beispiel

In MATLAB gilt bei IEEE double precision **eps** $= 2.220446049250313 \cdot 10^{-16}$, bei IEEE single precision **eps** $= 1.1920929 \cdot 10^{-7}$. In *Mathematica* erhält man durch `$MachineEpsilon` ebenfalls den Wert $2.220446049250313 \cdot 10^{-16}$ für die relative Maschinengenauigkeit.

Eine an sich mathematisch exakte Zahl $x \in \mathbb{R}$ wird infolge dieser Lücken bei der Eingabe in den Rechner mit Hilfe einer (rechnerabhängigen) Abbildung

$$\mathrm{rd} : x \mapsto \mathrm{rd}(x)$$

namens *Runden* durch die ihr nächstgelegene Maschinenzahl $\mathrm{rd}(x)$ ersetzt.

Beispiel

In IEEE single precision ist $x = 1.2345654321$ nicht darstellbar und wird bei der Eingabe in den Rechner auf $rd(x) = 1.2345654$ gerundet. Auch die von x verschiedene Zahl 1.2345654333 wird auf 1.2345654 gerundet.

Nach dem Runden kann die Zahl $x \in \mathbb{R}$ also nicht mehr von allen anderen reellen Zahlen, deren Rundung ebenfalls auf $\mathrm{rd}(x)$ führt, unterschieden werden. Eine Maschinenzahl $\mathrm{rd}(x)$ repräsentiert also immer eine Menge von reellen Zahlen. Es kann somit bereits bei der Eingabe einer Zahl $x \in \mathbb{R}$ in den Computer ein *Fehler durch Runden* entstehen. Bezeichnen wir diesen mit $\varepsilon_x \in \mathbb{R}$, so gilt

$$\mathrm{rd}(x) = x(1 + \varepsilon_x), \tag{2.18}$$

[15] Die Abstände zwischen den Zahlen im *IEEE* Modell von $\mathbb{R}$ sind im absoluten Sinn nicht einheitlich groß, siehe Abschnitt 8.

wobei der relative *Rundungsfehler* ε_x von x abhängt. Liegt x innerhalb der durch das Computermodell von $\mathbb{R}$ festgelegten Schranken, so lässt sich ε_x betragsmäßig durch eine vom Computermodell abhängige, aber von x unabhängige Größe $\mathbf{u}$ namens *Rundungseinheit* (unit roundoff) abschätzen, also

$$|\varepsilon_x| \leq \mathbf{u}. \tag{2.19}$$

Der Zusammenhang mit der relativen Maschinengenauigkeit ist durch

$$\mathbf{u} = \frac{1}{2}\mathbf{eps} \tag{2.20}$$

gegeben, nähere Ausführungen folgen in Abschnitt 8.

Beispiel

In Fortsetzung des vorangegangenen Beispiels ergibt sich mit $x = 1.2345654321$ und $rd(x) = 1.2345654$ der Rundungsfehler

$$\varepsilon_x = \frac{1.2345654 - 1.2345654321}{1.2345654321} = -2.6001 \cdot 10^{-8}.$$

Der Betrag dieses Rundungsfehlers ist gemäß (2.19) nach oben durch die Rundungseinheit $\mathbf{u} = \frac{1}{2}\mathbf{eps} = 5.9604645 \cdot 10^{-8}$ beschränkt.

Wegen (2.18) kann $\mathbf{u}$ für $x \neq 0$ auch als eine Schranke für den Betrag des *relativen Fehlers* $(\mathrm{rd}(x) - x)/x$ von $\mathrm{rd}(x)$ aufgefasst werden.

Rundungsfehler können aber nicht nur bei der Eingabe von reellen Zahlen, sondern auch beim Rechnen mit den Maschinenzahlen entstehen. Anstelle der auf $\mathbb{R}$ definierten Elementaroperationen $\diamond \in \{+, -, \cdot, /\}$ stehen bei Verwendung von Gleitkommaarithmetik am Computer nur sogenannte *Gleitkommaoperationen* $\tilde{\diamond} \in \left\{\tilde{+}, \tilde{-}, \tilde{\cdot}, \tilde{/}\right\}$ zur Verfügung. Der relative Fehler, der bei deren Durchführung entsteht, ist jedoch wieder durch die Rundungseinheit $\mathbf{u}$ abschätzbar. Für zwei Maschinenzahlen x_1 und x_2 gilt[16] also

$$x_1 \,\tilde{\diamond}\, x_2 = (x_1 \diamond x_2)(1 + \varepsilon_{x_1,x_2,\diamond}) \quad \text{mit } |\varepsilon_{x_1,x_2,\diamond}| \leq \mathbf{u}, \tag{2.21}$$

wobei der *Rundungsfehler* $\varepsilon_{x_1,x_2,\diamond}$ hier von x_1, x_2 und $\diamond$ abhängt[17]. Die Operationen $\tilde{\diamond}$ erfüllen weder das Assoziativ- noch das Distributivgesetz, so dass beim Rechnen am Computer die Reihenfolge der auszuführenden Operationen eine Rolle spielt.

Wir sind nun in der Lage zu verstehen, weshalb wir im Beispiel von Seite 14 ein fehlerhaftes Ergebnis erhalten haben. Bei numerischer Durchführung des Algorithmus *QuadGlgI* werden die Eingabedaten p und q zunächst auf $\tilde{p} = \mathrm{rd}(p)$ und $\tilde{q} = \mathrm{rd}(q)$ gerundet und anschließend gemäß

[16] Auf Ausnahmen von dieser Regel, wie etwa im `realmax`-Beispiel von Seite 15, gehen wir in Abschnitt 8 ein.

[17] Auch bei Verwendung der Wurzeloperation $\sqrt{\cdot}$ kann ein durch $\mathbf{u}$ beschränkter Rundungsfehler entstehen.

$\tilde{\xi}_1 \leftarrow \tilde{p} \,\tilde{\cdot}\, \tilde{p}$
$\tilde{\xi}_2 \leftarrow \tilde{\xi}_1 \,\tilde{-}\, \tilde{q}$
$\tilde{\xi}_3 \leftarrow \tilde{\sqrt{\xi_2}}$
$\tilde{x} \leftarrow \tilde{p} \,\tilde{-}\, \tilde{\xi}_3$
return $\tilde{x}$

weiterverarbeitet. Da das tatsächlich berechnete Resultat $\tilde{x}$ durch Rundungsfehler beeinflusst ist, stimmt es in der Regel nicht mit der gesuchten Lösung x aus der Algorithmenspezifikation überein. Die naheliegende Frage ist daher, ob $\tilde{x}$ als *Näherungslösung anstelle der exakten Lösung* x akzeptiert werden kann.

Beispiel

Im Beispiel auf Seite 14 hat der Algorithmus *QuadGlgI* (bei Durchführung in IEEE double precision) zur Näherungslösung $\tilde{x} = -0.099975585937500$ anstelle von $x = -0.1$ geführt. Im relativen Sinn ist der Betrag des entstandenen Fehlers von $\tilde{x}$ durch

$$\left|\frac{\tilde{x} - x}{x}\right| = 2.4414 \cdot 10^{-4}$$

gegeben. Der maximale relative Fehler $\mathbf{u}$ in den Daten p und q aufgrund der Rundung bei der Eingabe wurde also um den Faktor $2.1990 \cdot 10^{12}$ verstärkt, die Näherungslösung $\tilde{x}$ kann daher nicht als akzeptabel betrachtet werden.

Auch wenn die Rundungsfehler bei der Eingabe der Daten bzw. bei der Durchführung der einzelnen Operationen in der Regel vernachlässigbar klein sind, sie können sich im Laufe eines Algorithmus *fortpflanzen* und so zu großen Abweichungen zwischen berechneter und exakter Lösung führen. Der durch die Rundungsfehlerfortpflanzung verursachte Fehler im Ergebnis kann im Rahmen der *Stabilitätsanalyse eines (numerischen) Algorithmus*, siehe Abschnitt 4, abgeschätzt werden. Verschiedene Algorithmen können sich dabei in ihrem Fehlerverhalten bei ein und derselben Probleminstanz stark unterscheiden. Dies haben wir schon auf Seite 14 beim Vergleich der Algorithmen *QuadGlgI* und *QuadGlgII* beobachtet. Dort hat sich für die Probleminstanz mit $p = 5 \cdot 10^{11} - \frac{1}{20}$ und $q = -10^{11}$ der Algorithmus *QuadGlgII* als der bessere herausgestellt.

Neben Rundungsfehlern bei der Eingabe und beim Rechnen selbst muss man auch noch den *Rundungsfehler bei der Ausgabe* berücksichtigen.

Beispiel

Der Aufruf `QuadGlgI(`$5 \cdot 10^{11} - \frac{1}{20}, -10^{11}$`)` in MATLAB liefert bei Ausgabe im Standardformat das Resultat -0.1000. Wählen wir jedoch `format long` für die Ausgabe, so erhalten wir -0.099975585937500. Dies zeigt, dass `QuadGlgI` nicht die exakte Lösung berechnet, und durch die Resultatrundung auf -0.1000 beim Standardformat ein falscher Eindruck geweckt wird.

Rundungsfehler bei der Ausgabe können durch die Wahl eines geeigneten Ausgabeformats vermieden werden. Aus Gründen der Lesbarkeit, werden diese jedoch häufig in Kauf genommen.

Datenfehler

Neben Rundungs- und Abbruchsfehler bilden *Datenfehler* eine weitere Fehlerart, zu der streng genommen auch die bereits erwähnten Rundungsfehler bei der Eingabe von Zahlen zählen. Datenfehler können aber auch dadurch bedingt sein, dass die Eingangsdaten des betrachteten Problems das fehlerhafte Ergebnis einer zuvor gelösten Aufgabe sind. Eine weitere Möglichkeit ist, dass die Eingangsdaten aus experimentellen Messungen stammen und daher durch Messfehler bzw. Messungenauigkeiten verfälscht wurden. Liegen also anstelle der exakten Daten x gestörte Daten $\tilde{x}$ vor, so wird die (eindeutige) Lösung anstelle von $\varphi(x)$ durch $\varphi(\tilde{x})$ beschrieben.

Beispiel

Wir betrachten das Problem der Addition einer reellen Zahl x zu 0.1234567, also

$$\varphi(x) = x + 0.1234567. \tag{2.22}$$

Mit der Eingabe $x = -0.1234576$ lautet das Ergebnis -0.0000009. Nehmen wir nun an, dass anstelle von x ein fehlerhafter Wert $\tilde{x} = -0.1235576$ vorliegt. Der absolute Datenfehler $\tilde{x} - x$ ist 0.0001, die entsprechende Lösung des Problems lautet $\varphi(\tilde{x}) = -0.0001009$. Wegen $\varphi(\tilde{x}) - \varphi(x) = \tilde{x} - x$ stimmt der absolute Fehler im Ergebnis mit dem absoluten Datenfehler überein. Hingegen gilt für den Betrag des relativen Datenfehlers

$$\left|\frac{\tilde{x} - x}{x}\right| = 8.0999 \cdot 10^{-4},$$

für den relativen Fehler im Ergebnis folgt

$$\left|\frac{\varphi(\tilde{x}) - \varphi(x)}{\varphi(x)}\right| = 111.11.$$

Der relative Datenfehler wird also um einen Faktor $1.3718 \cdot 10^5$ verstärkt.

Wie im Beispiel gezeigt wird bei der Addition zweier betragsmäßig ungefähr gleich großer Zahlen mit entgegengesetztem Vorzeichen der relative Datenfehler stark verstärkt. Diesen Effekt nennt man *Auslöschung*, da seine Ursache in der Annullierung führender Ziffern bei der Bildung von Differenzen liegt.

Da sich Datenfehler zwangsläufig und unabhängig vom Algorithmus im Ergebnis niederschlagen, wird der durch sie verursachte Fehler auch als *unvermeidbar* bezeichnet. Die Frage, wie sehr sich der Datenfehler auf den Fehler in der Lösung auswirkt, ist Gegenstand der *Konditionsanalyse des Problems*, siehe Abschnitt 3. Sie ist unabhängig vom Algorithmus und folgt bei der theoretischen Untersuchung eines mathematischen Problems meist der Klärung von Existenz und Eindeutigkeit einer Lösung.

Diskretisierte Ersatzprobleme

Wir wenden uns nun einer weiteren Fehlerart beim Rechnen am Computer zu, die entsteht, wenn ein am Rechner nicht greifbares mathematisches Problem durch ein

Näherungsproblem ersetzt werden muss. Dies tritt meist dann ein, wenn das ursprüngliche Problem eine mit einer „unendlichen Dimension behaftete Größe", etwa eine stetige Funktion $f : \mathbb{R} \to \mathbb{R}$, involviert. Deren *Diskretisierung*, etwa das Ersetzen von f durch endlich vielen Funktionswerte $f_i = f(x_i)$ an gegegebenen Stützstellen x_i, führt dann zu einem diskretisierten Ersatzproblem. Anstelle des Integrals $I = \int_0^1 f(x)\,dx$ berechnet man so dann zum Beispiel die Summe $I_n = \sum_{i=0}^{n} w_i f_i$ mit geeigneten Gewichten w_i, erhält dabei jedoch einen *Diskretisierungsfehler* $I - I_n$. Als ausführlicheres Beispiel zum Thema Diskretisierungsfehler[18] wollen wir nun jedoch das Problem der Berechnung des Umfangs des Kreises mit Radius $r = 1/2$ betrachten.

Problemstellung (Umfang des Kreises mit Radius $1/2$).

Gegeben:	—.[19]
Gesucht:	$U \in \mathbb{R}^+$
mit:	U ist Umfang des Kreises mit Radius $1/2$.

Aus der Schule ist bekannt, dass die Lösung des Problems durch $U = \pi$ gegeben ist. Ohne Kenntnis des Zahlenwerts von π (oder zumindest einer Näherung davon) können wir auch den Zahlenwert des Umfangs nicht angeben. Natürlich steht auf jedem Rechner ein Näherungswert für π zur Verfügung, der jedoch selbst wieder Resultat eines Algorithmus ist. Es gibt mehrere Verfahren zur π-Approximation, eines davon beruht auf der Kreisumfangsberechnung mit Hilfe eines einfacher zu handhabenden Ersatzproblems.

Die Idee, den Zahlenwert von π mit Hilfe von in einen Kreis eingeschriebenen Vielecken anzunähern, stammt von Archimedes[20]. Abbildung I.2 lässt vermuten, dass durch Einschreiben eines 12-, 24-, 48-Ecks etc. immer bessere Näherungen erzielt werden können. Zum Problem der Berechnung des Kreisumfangs U lautet ein diskretisiertes Ersatzproblem also, den Umfang U_n eines in den Kreis mit Radius $1/2$ eingeschriebenen regelmäßigen $6 \cdot 2^n$-Ecks zu berechnen.

Problemstellung (Umfang des regelmäßigen $6 \cdot 2^n$-Ecks).

Gegeben:	$n \in \mathbb{N}_0$
Gesucht:	$u \in \mathbb{R}^+$
mit:	$u = U_n$.

Während das Ausgangsproblem keine Eingabegröße besitzt, taucht im Ersatzproblem nun der *Diskretisierungsparameter* n als Input auf[21]. Wir bezeichnen die Seitenlänge des in den Kreis mit Radius $1/2$ eingeschriebenen regelmäßigen $6 \cdot 2^n$-Ecks durch l_n, für sie gilt jedenfalls $l_n \leq r = 1/2$. Die Lösung der diskretisierten

[18] Auch der Rundungsfehler bei Gleitkommaarithmetik hat seine Ursache in einer Diskretisierung, nämlich jener beim Übergang von $\mathbb{R}$ auf die Maschinenzahlen, die ja eine diskrete Teilmenge von $\mathbb{R}$ sind. Weiters könnte man bei iterativen Verfahren mit Abbruchsfehlern die Suche nach dem n-ten Folgenglied als Ersatzaufgabe formulieren. Wir wollen jedoch den Begriff des Diskretisierungsfehlers klar von jenem des Rundungsfehlers bzw. Abbruchsfehlers trennen.

[19] Eine Problemspezifikation kann auch keine Eingabevariable besitzen. Die Problemklasse besteht dann nur aus einer Probleminstanz.

[20] Archimedes von Syrakus: 287–212 v. Chr., griechischer Mathematiker. Die hier verwendete Ausschöpfungsmethode gilt als Vorläufer der modernen Integralrechnung. Er bewies auch, dass in einem Kreis das Verhältnis von Umfang zu Durchmesser gleich jenem von Fläche zum Quadrat des Radius ist, heutzutage spricht man dabei von π. Nach ihm benannt ist auch das Axiom von Archimedes, wonach es zu zwei Größen $x, y > 0$ immer ein $n \in \mathbb{N}$ gibt mit $nx > y$.

[21] Allgemein, also auch bei Ausgangsproblemen mit Eingabe x, bringt die Diskretisierung für das Ersatzproblem eine zusätzliche Inputgröße in Form des Diskretisierungsparameters.

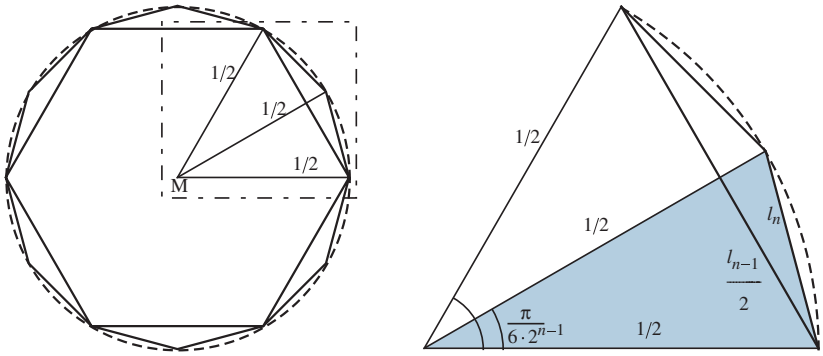


Abb. I.2. In einen Kreis eingeschriebenes $6 \cdot 2^n$-Eck und $6 \cdot 2^{n-1}$-Eck.

Aufgabe lautet damit

$$U_n = 6 \cdot 2^n \cdot l_n. \tag{2.23}$$

Mit Hilfe des Sinussatzes im rechtwinkeligen Dreieck zeigt man, dass

$$l_n = \sin(\frac{\pi}{6 \cdot 2^n}) \tag{2.24}$$

gilt, siehe Abbildung I.2. Diese Formel ist aber für die direkte numerische Berechnung von l_n nicht verwendbar, da sie ja den Zahlenwert der Lösung $U = \pi$ des Ausgangsproblems erfordert[22]. Aus (2.23) und (2.24) lässt sich jedoch eine obere Schranke für den Diskretisierungsfehler $U_n - U$ ableiten. Dazu verwenden wir den Satz von Taylor[23], dessen Beweis Gegenstand jeder Analysis-Vorlesung ist.

Satz

Satz von Taylor. Seien $k \in \mathbb{N}$ und $x_0, x \in \mathbb{R}$ mit $x \neq x_0$. Weiters sei die Funktion $f : [x_0, x]$ (bzw. $[x, x_0]$) $\to \mathbb{R}$ k-mal stetig differenzierbar und in (x_0, x) (bzw. (x, x_0)) existiere die $k+1$-te Ableitung $f^{(k+1)}$. Dann gilt

$$\exists \theta \in (0,1) : f(x) = \sum_{i=0}^{k} \frac{f^{(i)}(x_0)}{i!}(x - x_0)^i + \frac{f^{(k+1)}(x_0 + \theta(x - x_0))}{(k+1)!}(x - x_0)^{k+1}.$$

Zudem gilt für den Fehler zwischen $f(x)$ und[24]

$$T_k(x) := \sum_{i=0}^{k} \frac{f^{(i)}(x_0)}{i!}(x - x_0)^i \tag{2.25}$$

[22] Ein auf (2.24) beruhender Algorithmus benötigt auch die Sinusfunktion als Elementaroperation.

[23] Taylor, Brook: 1685–1731, englischer Mathematiker. Neben dem hier erwähnten Satz stammt von ihm auch die Methode der partiellen Integration.

[24] Man nennt $\sum_{i=0}^{k} \frac{f^{(i)}(x_0)}{i!}(x - x_0)^i$ das k-te Taylorpolynom von f zum Entwicklungspunkt x_0.

die Abschätzung

$$|f(x) - T_k(x)| \leq \frac{|x - x_0|^{k+1}}{(k+1)!} \sup_{\theta \in (0,1)} |f^{(k+1)}(x_0 + \theta(x - x_0))|. \tag{2.26}$$

Mit $f(x) = \sin(x), x_0 = 0$ und $k = 2$ folgt aus (2.26) zunächst

$$|\sin(x) - x| \leq \frac{x^3}{6} \quad \text{für } x > 0.$$

Für $x = \frac{\pi}{6 \cdot 2^n}$ folgt daraus mit (2.23)

$$|U_n - U| \leq \frac{U^3}{6(6 \cdot 2^n)^2}, \tag{2.27}$$

und daher

$$\lim_{n \to \infty} U_n = U. \tag{2.28}$$

Dies zeigt, dass U_n mit wachsendem n eine immer bessere Näherung des gesuchten Umfangs U des Kreises ist.

Wir wenden uns nun dem Unterproblem der Berechnung der Seitenlänge l_n ohne Verwendung von (2.24) zu, dessen Lösung wir dann nach (2.23) zur Lösung des Hauptproblems der Berechnung von U_n heranziehen können.

Problemstellung (Seitenlänge des regelmäßigen $6 \cdot 2^n$-Ecks).

Gegeben: $n \in \mathbb{N}_0$
Gesucht: $l \in \mathbb{R}^+$
mit: $l = l_n$.

Eine erste Idee für ein Verfahren zur Berechnung von l_n ohne Verwendung von (2.24) beruht auf der Beziehung

$$l_n = \sqrt{\frac{1}{2} - \frac{1}{2}\sqrt{1 - l_{n-1}^2}} \tag{2.29}$$

zwischen den Seitenlängen l_n und l_{n-1} des $6 \cdot 2^n$-Ecks bzw. des $6 \cdot 2^{n-1}$-Ecks, die aus Abbildung I.2 mit Hilfe des Satzes von Pythagoras[25] abgeleitet werden kann. Die Berechnung von l_n kann also auf jene von l_{n-1} zurückgeführt werden. Nach n derartigen Reduktionen landen wir bei $n = 0$, also der bekannten Seitenlänge des Sechsecks $l_0 = 1/2$.

Für die tatsächliche numerische Realisierung eignet sich (2.29) jedoch nur bedingt, da bei großen Werten von n die Seitenlänge l_{n-1} sehr klein ist, und dies die Subtraktion der nahe beieinander liegenden Zahlen $\frac{1}{2}$ und $\frac{1}{2}\sqrt{1 - l_{n-1}^2}$ nach sich

[25] Pythagoras von Samos: ca. 570–475 v. Chr., griechischer Mathematiker. Sein Leben ist wenig erforscht, und authentische Berichte über sein Leben sind nicht überliefert. Der berühmte Satz über die Seitenlängen im rechtwinkeligen Dreieck war schon lange vor ihm bekannt, doch Euklid benannte ihn wahrscheinlich nach ihm, da Pythagoras den Satz wahrscheinlich als Erster bewies. Vermutlich war Pythagoras auch Führer einer mystischen Sekte.

zieht. Rundungsfehler im Zwischenresultat $\sqrt{1-l_{n-1}^2}$, die ja dann als Datenfehler bei der Berechnung von l_n aufgefasst werden können, pflanzen sich dann infolge der auf Seite 18 erwähnten Auslöschung stark fort.

Einen Ausweg liefert die Identität

$$l_n = \sqrt{\frac{1}{2} - \frac{1}{2}\sqrt{1-l_{n-1}^2}} = \frac{l_{n-1}}{\sqrt{2(1+\sqrt{1-l_{n-1}^2})}}, \tag{2.30}$$

da damit die Subtraktion etwa gleich großer Zahlen vermieden wird[26]. Wie (2.29) reduziert auch (2.30) die Berechnung von l_n auf die Berechnung von l_{n-1}. Dies führt auf den Algorithmus *SeitenlängeR* zur Berechnung von l_n.

Algorithmus *SeitenlängeR*: Rekursive Berechnung der Seitenlänge des $6 \cdot 2^n$-Ecks

```
if n = 0
    l ← 0.5
else
    l ← SeitenlängeR(n − 1)
    l ← l / √(2(1+√(1−l²)))
return l
```

Aufruf: *SeitenlängeR*(n)
Eingabe: $n \in \mathbb{N}_0$
Ausgabe: $l \in \mathbb{R}^+$
mit: $l = l_n$.

Das Bemerkenswerte an diesem Algorithmus ist nun, dass *SeitenlängeR sich selbst* als Unteralgorithmus aufruft. Selbstaufrufe werden meist als *rekursive Aufrufe* bezeichnet, sich selbst aufrufende Algorithmen werden daher *rekursive Algorithmen* genannt.

Die Funktionsweise derartiger Algorithmen beruht zum Einen darauf, dass es (mindestens) einen *Basisfall* gibt, in dem das Ergebnis ohne rekursiven Aufruf ermittelt werden kann. Zum Anderen muss sichergestellt sein, dass der Algorithmus bei jedem rekursiven Aufruf einem Basisfall näher kommt und diesen tatsächlich nach endlich vielen Aufrufen erreicht. Im Fall des Algorithmus *SeitenlängeR* lautet der Basisfall $n = 0$. Zudem nimmt in jedem rekursiven Aufruf der Parameter $n \in \mathbb{N}$ um eins ab[27], so dass der Algorithmus nach endlich vielen Schritten im Basisfall $n = 0$ landet und somit terminiert.

Mit Hilfe von *SeitenlängeR* lässt sich nach (2.23) leicht der Algorithmus *VieleckUmfang* zur Berechnung von U_n aufstellen.

Algorithmus *VieleckUmfang*: Näherung des Kreisumfangs bei Radius 1/2

```
l ← SeitenlängeR(n)
u ← 6 · 2^n · l
return u
```

Aufruf: *VieleckUmfang*(n)
Eingabe: $n \in \mathbb{N}_0$
Ausgabe: $u \in \mathbb{R}^+$
mit: $u = U_n$.

Dieser Algorithmus verwendet also das zuvor entwickelte Verfahren *SeitenlängeR* als *Unteralgorithmus*. Im Sinne des Algorithmenbegriffs von Seite 5 kann dann

[26] Einen ähnlichen Umformungstrick haben wir auch schon in (1.6) verwendet.
[27] Für $n = 0$ findet kein rekursiver Aufruf statt.

SeitenlängeR neben dem Potenzieren und dem Multiplizieren auch als Elementaroperation in *VieleckUmfang* aufgefasst werden.

Approximation von π. Eine Näherung für π steht in allen modernen Programmiersprachen zur Verfügung, in MATLAB etwa führt der Aufruf `pi` auf den Näherungswert

Beispiel

$$3.141592653589793. \tag{2.31}$$

Diesen Wert verwenden wir nun, um die durch `VieleckUmfang(n)` erhaltenen Näherungswerte zu beurteilen. Tabelle I.1 zeigt, dass mit wachsendem n die durch (2.27) motivierte Schranke $\frac{\mathtt{pi}^3}{6(6\cdot 2^n)^2}$ für den absoluten Fehler abnimmt, und immer mehr Stellen der Näherung mit dem Referenzwert (2.31) übereinstimmen. `VieleckUmfang(24)` unterscheidet sich von (2.31) noch in der letzten Stelle, für $n \geq 25$ liefert `VieleckUmfang(n)` dann genau den Wert aus (2.31).

n	`VieleckUmfang(n)`	$\frac{\mathtt{pi}^3}{6(6\cdot 2^n)^2}$
5	3.141452472285462	$4.4621 \cdot 10^{-5}$
10	3.141592516692156	$4.3576 \cdot 10^{-8}$
15	3.141592653456103	$4.2555 \cdot 10^{-11}$
20	3.141592653589662	$4.1842 \cdot 10^{-14}$
24	3.141592653589792	$2.8272 \cdot 10^{-16}$
25	3.141592653589793	0.

Tabelle I.1. Approximation von π

Von Rekursionen zu Schleifen

Nachdem wir nun Iterationsverfahren, diskretisierte Ersatzprobleme sowie verschiedene Fehlerquellen beim Rechnen am Computer vorgestellt haben, wenden wir uns zum Abschluss unserer einführenden Diskussion noch der Überführung rekursiver Verfahren in Schleifenalgorithmen zu. Dies ist aufgrund des zusätzlichen Speicherplatzbedarfs bei der Abarbeitung rekursiver Aufrufe am Computer von Vorteil.

Zur Berechnung der Seitenlänge eines regelmäßigen $6 \cdot 2^n$-Ecks haben wir im vorangegangenen Abschnitt den rekursiven Algorithmus *SeitenlängeR* vorgestellt. Zum leichteren Verständnis des damit verbundenen Organisationsaufwandes ist es hilfreich, sich zuerst zu überlegen, wie der Algorithmus mit Papier und Bleistift ausgeführt werden kann.

Wir berechnen die Seitenlänge eines regelmäßigen $6 \cdot 2^2$-Ecks durch Ausführung von *SeitenlängeR*(2). Solange der Basisfall noch nicht erreicht ist, finden rekursive Aufrufe für immer kleiner werdenden Input statt. Wir rufen demnach zuerst *SeitenlängeR*(1) auf, merken uns aber, dass wir den berechneten Wert anschließend noch weiterverarbeiten müssen. Zur Erinnerung schreiben wir die noch ausstehende Anweisung $l \leftarrow \frac{l}{\sqrt{2(1+\sqrt{1-l^2})}}$ auf ein Kärtchen und legen es zur Seite. In der Abarbeitung von *SeitenlängeR*(1) rufen wir nochmals rekursiv

Beispiel

SeitenlängeR(0) auf, beschreiben wieder ein Kärtchen mit der Anweisung $l \leftarrow \frac{l}{\sqrt{2(1+\sqrt{1-l^2})}}$ und legen es auf das zuvor abgelegte Kärtchen. In *SeitenlängeR*(0) sind wir mit $n = 0$ im Basisfall angelangt und setzen $l \leftarrow 0.5$. Während der rekursiven Aufrufe hat sich ein Stapel von Kärtchen mit noch auszuführenden Anweisungen gebildet, den wir nun noch in umgekehrter Reihenfolge abarbeiten müssen. Dies führt der Reihe nach zu

$$l \leftarrow \frac{0.5}{\sqrt{2(1+\sqrt{1-0.5^2})}} = 0.258819 \text{ und}$$
$$l \leftarrow \frac{0.258819}{\sqrt{2(1+\sqrt{1-0.258819^2})}} = 0.130526.$$

Da nun der gesamte Stapel von Kärtchen verarbeitet ist, sind wir fertig, und *SeitenlängeR*(2) liefert 0.130526 als Resultat.

Was wir in der händischen Abarbeitung des Algorithmus durch einen Stapel von Kärtchen simulieren, läuft auch in einem echten Programm ab, sofern die verwendete Programmiersprache rekursive Funktionsaufrufe erlaubt. Ein Computer verwendet dabei zusätzlich einen Speicherbereich, der gerade so verwaltet wird wie ein Stapel mit Kärtchen – den sogenannten Stack (engl. für Stapel). In Berechnungen, die viele rekursive Aufrufe benötigen, führt das zu merkbarer Verlangsamung des Programmablaufs und unter Umständen sogar zu Speicherüberlauf am Stack.

Ein rekursiver Aufruf lässt sich aber oft vermeiden, indem man auf das eng damit verwandte Konstrukt einer Schleife, siehe Seite 7, zurückgreift. In der Rekursion wird das Ausgangsproblem zuerst *top-down* auf ein (oder mehrere) einfachere(s) Teilproblem(e) zurückgeführt, aus dessen/deren Lösung dann die Lösung des Ausgangsproblems rekonstruiert wird. In der Schleife hingegen baut man das Resultat *bottom-up* beginnend beim Basisfall auf. Wir wollen die Idee anhand unseres einfachen Verfahrens zur Seitenlängenberechnung demonstrieren.

Beispiel

Übersetzung des rekursiven Algorithmus *SeitenlängeR* in eine Schleife. Wir identifizieren zuerst den Basisfall der Rekursion. Dieser ist bei $n = 0$ mit $l \leftarrow 0.5$ erreicht. Weiters überzeugt man sich schnell davon, dass sich in der rekursiven Abarbeitung zum Zeitpunkt des Erreichens des Basisfalls genau n Anweisungen der Form

$$l \leftarrow \frac{l}{\sqrt{2(1+\sqrt{1-l^2})}} \tag{2.32}$$

am Stack befinden. Ein Algorithmus *SeitenlängeS* erreicht somit das gleiche Resultat, wenn mit $l \leftarrow 0.5$ begonnen wird und dann *in einer Schleife* n-mal (2.32) wiederholt wird.

Algorithmus *SeitenlängeS*: Berechnung der Seitenlänge des $6 \cdot 2^n$-Ecks mittels Schleife

$l \leftarrow 0.5$
for j **from** 1 **to** n
  $l \leftarrow \frac{l}{\sqrt{2(1+\sqrt{1-l^2})}}$
return l

Aufruf: *SeitenlängeS*(n)
Eingabe: $n \in \mathbb{N}_0$
Ausgabe: $l \in \mathbb{R}^+$
mit: $l = l_n$.

Mit *SeitenlängeR* und *SeitenlängeS* stehen nun zwei Algorithmen zur Berechnung der Seitenlänge des regelmäßigen $6 \cdot 2^n$-Ecks zur Verfügung. Wir können somit im Algorithmus *VieleckUmfang* von Seite 22 auch auf den Unteralgorithmus *SeitenlängeS* anstelle von *SeitenlängeR* zurückgreifen. Wir wollen abschließend untersuchen, wie sich der zusätzliche Organisations- bzw. Speicherplatzaufwand der rekursiven Variante im Vergleich zur Schleife auf die benötigte Rechenzeit des Algorithmus auswirkt.

Beispiel

Näherung an π. Wir berechnen mit `VieleckUmfang` in MATLAB Näherungen an π, einmal unter Verwendung des rekursiven Unteralgorithmus `SeitenlängeR` und einmal mit der Schleifenvariante `SeitenlängeS`. Die dazu benötigte Rechenzeit ist in Abbildung I.3 dargestellt.

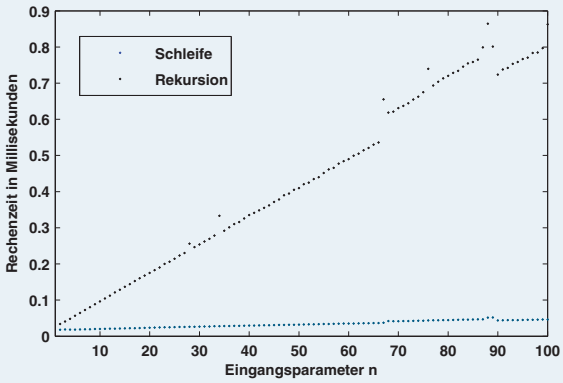


Abb. I.3. Vergleich des zeitlichen Rechenaufwands.

Bei beiden Algorithmen zeigt sich in Abhängigkeit vom Input n ein annähernd linearer Anstieg[28]. Bei fixem n liegt die Rechenzeit für den rekursiven Algorithmus deutlich höher als beim Schleifenalgorithmus. Grund dafür ist die unterschiedliche Abarbeitung von Rekursion und Schleife. Die numerische Qualität des Resultats ist unabhängig davon, ob wir `SeitenlängeR` oder `SeitenlängeS` verwenden, auch der Schleifenalgorithmus berechnet genau die in Tabelle I.1 gezeigten Werte.

[28] Wiederholte Ausführung des Algorithmus mit identischen Eingaben können leichte Schwankungen in den Rechenzeiten zeigen. Eine Ursache dafür ist, dass der Prozessor neben der Abarbeitung des Algorithmus auch mit unterschiedlichen Tätigkeiten zur Systemerhaltung beschäftigt ist.

3 Kondition eines Problems

Neben Untersuchungen hinsichtlich Existenz und Eindeutigkeit einer Lösung stellt sich bei der Diskussion eines mathematischen Problems auch die Frage, wie sich Änderungen der Eingangsdaten x auf die Lösung y auswirken. Mögliche Ursachen für Störungen der Daten bzw. Datenfehler haben wir auf Seite 18 bereits kennen gelernt. Nach Hadamard[29] bezeichnet man ein mathematisches Problem als *korrekt oder gut gestellt*, wenn zu allen zulässigen Daten x eine Lösung y existiert, diese Lösung eindeutig ist, und die Lösung stetig von den Daten abhängt. Ist eines dieser drei Kriterien nicht erfüllt, so wird das Problem *inkorrekt oder schlecht gestellt*[30] genannt.

Im Falle der eindeutigen Lösbarkeit haben wir auf Seite 3 die Bezeichnung $\varphi(x)$ für die eindeutige Lösung des Problems (φ,x) zu den Daten x eingeführt. Liegen anstelle von x gestörte Daten $\tilde{x}$ vor, so erhält man anstelle von $\varphi(x)$ jedoch $\varphi(\tilde{x})$. Das dritte Kriterium von Hadamard für Korrekt-Gestelltheit verlangt, dass eine Annäherung von $\tilde{x}$ an x einen stetigen Übergang von $\varphi(\tilde{x})$ nach $\varphi(x)$ mit sich bringt. Entscheidend dabei ist natürlich, bezüglich welches Abstandsbegriffs die Stetigkeit von φ untersucht wird. Eng damit verbunden ist die *Kondition des Problems* (φ,x), welche ein Maß für das Verhältnis zwischen dem Fehler im Ergebnis $\varphi(\tilde{x}) - \varphi(x)$ und dem ihn verursachenden Datenfehler $\tilde{x} - x$ ist.

Der Einfachheit halber betrachten wir in der weiteren Diskussion ein eindeutig lösbares Problem (φ,x) zunächst über $\mathbb{R}$, also

$$\varphi : I \subset \mathbb{R} \to \mathbb{R} \tag{3.33}$$

mit geeignetem Definitionsintervall I. Stellen wir uns zum Beispiel $z \in \mathbb{R}$ als nicht fehleranfällige, feste[31] Größe vor, so können wir das Problem der Addition einer Zahl $x \in \mathbb{R}$ zu z durch

$$\varphi : \mathbb{R} \to \mathbb{R},\ x \mapsto x + z \tag{3.34}$$

im Sinne von (3.33) beschreiben. Bei der Konditionsanalyse des Problems (φ,x) untersucht man dann für gestörte Daten $\tilde{x} \in I$, wie sich der Fehler $\varphi(\tilde{x}) - \varphi(x)$ im Ergebnis zum Datenfehler $\tilde{x} - x$ verhält, wobei man sich letzteren als immer kleiner werdend vorstellt. Für diese *asymptotische* Betrachtung ist die Verwendung des *Landau*[32] Symbols o hilfreich.

---

[29] Hadamard, Jacques Salomon: 1865–1963, französischer Mathematiker. Lieferte den ersten Beweis (unabhängig von ihm auch der Belgier Charles de la Vallée Poussin) des Primzahlsatzes, wonach die Anzahl der Primzahlen $\leq n$ gleich schnell wächst wie $n/\ln n$.

[30] Schlecht gestellte Probleme sind entgegen der ursprünglichen Auffassung nicht Folge einer fehlerhaften Modellierung. Viele *inverse Probleme* aus Naturwissenschaft, Industrie und Technik, bei denen nach Ursachen für gewünschte oder beobachtete Effekte gesucht wird, sind schlecht gestellt. Zur numerischen Lösung inverser und schlecht gestellter Probleme werden sogenannte Regularisierungsmethoden eingesetzt. Dabei wird das schlecht gestellte Problem durch ein benachbartes und gut gestelltes ersetzt, wodurch ein zusätzlicher Approximationsfehler entsteht, siehe [4].

[31] Die Zahl z kann in unserer Diskussion also nicht durch Rundungs- oder Datenfehler verfälscht sein.

[32] Landau, Edmund: 1877–1938, deutscher Mathematiker. Gab einen wesentlich einfacheren Beweis des Primzahlsatzes, den auch schon Hadamard bewiesen hatte, siehe Seite 26.

Landau-Symbol o für reelle Funktionen. Sei $f : \mathbb{R} \to \mathbb{R}$. Zu gegebenem $x \in \mathbb{R}$ beschreibt der Ausdruck $o(f(\tilde{x}))$ für $\tilde{x} \to x$ eine Funktion $g : \mathbb{R} \to \mathbb{R}$, für die

Bezeichnung

$$\lim_{\tilde{x} \to x} \frac{g(\tilde{x})}{f(\tilde{x})} = 0$$

gilt. Dafür verwendet man die Schreibweise

$$g(\tilde{x}) = o(f(\tilde{x})) \quad \text{für } \tilde{x} \to x.$$

Intuitiv beschreibt $o(f(\tilde{x}))$ für $\tilde{x} \to x$ eine nicht näher spezifizierte Funktion, die in der Nähe von x viel kleiner als f ist.

Wegen $\lim_{\tilde{x} \to 0} 3\tilde{x} + \tilde{x}^2 = 0$ gilt

Beispiel

$$3\tilde{x}^2 + \tilde{x}^3 = o(\tilde{x}) \quad \text{für } \tilde{x} \to 0.$$

Bei Vorliegen von Ausdrücken für den Resultatsfehler $\varphi(\tilde{x}) - \varphi(x)$ oder auch von Abschätzungen für dessen Betrag $|\varphi(\tilde{x}) - \varphi(x)|$ erlaubt diese Landau-Notation eine kompakte Darstellung.

Sei φ auf $[\tilde{x},x]$ (bzw. $[x,\tilde{x}]$) stetig und wenigstens auf $(\tilde{x},x)$ (bzw. $(x,\tilde{x})$) differenzierbar. Nach dem Mittelwertsatz der Differentialrechnung gilt dann

Beispiel

$$\begin{aligned} \varphi(\tilde{x}) - \varphi(x) &= \varphi'(\xi)(\tilde{x} - x) \\ &= \varphi'(x)(\tilde{x} - x) + (\varphi'(\xi) - \varphi'(x)) \cdot (\tilde{x} - x) \end{aligned} \tag{3.35}$$

für ein $\xi \in (\tilde{x},x)$ (bzw. $(x,\tilde{x})$). Ist φ' stetig, folgt

$$\varphi(\tilde{x}) - \varphi(x) = \varphi'(x)(\tilde{x} - x) + o(|\tilde{x} - x|) \quad \text{für } \tilde{x} \to x. \tag{3.36}$$

Durch Betragsbildung in (3.36) erhält man die Abschätzung

$$|\varphi(\tilde{x}) - \varphi(x)| \leq |\varphi'(x)||\tilde{x} - x| + o(|\tilde{x} - x|) \quad \text{für } \tilde{x} \to x.$$

Symbole $\approx$ und $\lesssim$. Seien g und h Funktionen von $\mathbb{R}$ nach $\mathbb{R}$. Anstelle von

Bezeichnung

$$\begin{aligned} g(\tilde{x}) &= h(\tilde{x}) + o(|\tilde{x} - x|) \quad \text{für } \tilde{x} \to x, \text{ bzw.} \\ g(\tilde{x}) &\leq h(\tilde{x}) + o(|\tilde{x} - x|) \quad \text{für } \tilde{x} \to x \end{aligned}$$

schreiben wir kurz

$$g(\tilde{x}) \asymp h(\tilde{x}) \quad \text{für } \tilde{x} \to x, \text{ bzw.}$$
$$g(\tilde{x}) \lesssim h(\tilde{x}) \quad \text{für } \tilde{x} \to x.$$

Die Betrachtung des Fehlers bis zur Ordnung $o(|\tilde{x}-x|)$ liegt nun auch der Definition der Kondition des Problems (φ,x) zu Grunde.

Definition

Absolute Kondition. Sie φ wie in (3.33). Die *absolute Kondition* des Problems (φ,x) ist die kleinste Zahl κ_{abs}, für die

$$|\varphi(\tilde{x}) - \varphi(x)| \lesssim \kappa_{\text{abs}} |\tilde{x} - x| \quad \text{für } \tilde{x} \to x \tag{3.37}$$

erfüllt ist.

Die Konditionszahl κ_{abs} beschreibt also die im ungünstigsten Fall auftretende Verstärkung des Betrags des *absoluten* Datenfehlers $\tilde{x}-x$. Sie hängt sowohl von φ als auch von x ab. Existiert so eine Zahl nicht (formal $\kappa_{\text{abs}} = \infty$), so ist das Problem schlecht oder inkorrekt gestellt. Bei $\kappa_{\text{abs}} \leq 1$ nennt man das Problem *gut konditioniert* oder auch *datenstabil*, bei $\kappa_{\text{abs}} > 1$ ist es *schlecht konditioniert* oder *dateninstabil*.

Die Bedingung (3.37) kann auch als asymptotische Lipschitzstetigkeit der Abbildung φ in x aufgefasst werden. Man beachte, dass die Definition der Konditionszahl nicht die Differenzierbarkeit von φ voraussetzt. Ist jedoch φ stetig differenzierbar, so folgt aus (3.36)

$$\kappa_{\text{abs}} = |\varphi'(x)|. \tag{3.38}$$

Beispiel

Im Beispiel der Abbildung (3.34) ist nach (3.38) die Konditionszahl κ_{abs} durch 1 gegeben. Es gilt auch die gröbere Abschätzung

$$|\varphi(\tilde{x}) - \varphi(x)| \lesssim 2|\tilde{x} - x| \quad \text{für } \tilde{x} \to x, \tag{3.39}$$

der Schluss $\kappa_{\text{abs}} = 2$ ist jedoch falsch, da die Konditionszahl ja die kleinstmögliche Zahl ist, mit der (3.37) erfüllt werden kann.

Bislang haben wir in diesem Abschnitt nur das Konzept des absoluten Fehlers betrachtet. Die Kondition eines Problems kann jedoch auch bzgl. *relativer* Fehler untersucht werden.

Definition

Relative Kondition. Sei φ wie in (3.33). Die *relative Kondition* des Problems (φ,x) mit $\varphi(x) \neq 0$ und $x \neq 0$ ist die kleinste Zahl κ_{rel}, für die

$$\frac{|\varphi(\tilde{x}) - \varphi(x)|}{|\varphi(x)|} \lesssim \kappa_{\text{rel}} \frac{|\tilde{x} - x|}{|x|} \quad \text{für } \tilde{x} \to x \tag{3.40}$$

erfüllt ist.

Im Falle der stetigen Differenzierbarkeit von φ folgt aus (3.36)

$$\kappa_{\text{rel}} = \frac{|\varphi'(x)|}{|\varphi(x)|}|x|. \tag{3.41}$$

Bei Verwendung von κ_{rel} ist die Trennung zwischen gut und schlecht konditioniert nicht mehr klar festgelegt. Richtwerte für Datenstabilität können neben $\kappa_{\text{rel}} = 1$ dann auch $\kappa_{\text{rel}} = 10, 100, 1000$ sein. Welches der beiden Fehlerkonzepte man für die Konditionsanalyse heranzieht, hängt auch davon ab, in welcher Form Abschätzungen für den Datenfehler vorliegen. Steht $\tilde{x}$ lediglich für den gerundeten Wert $\text{rd}(x)$ von x, wird man jedenfalls die Abschätzung (2.19) des relativen Fehlers mit Hilfe der Rundungseinheit $\mathbf{u}$ heranziehen.

Beispiel

Bei relativer Betrachtung ist die Kondition der Addition aus Beispiel (3.34) durch

$$\kappa_{\text{rel}} = \frac{|x|}{|x+z|}$$

gegeben. Falls x entgegengesetztes Vorzeichen zu z, aber ungefähr denselben Betrag wie z hat, ist das Problem (φ, x) im relativen Sinn schlecht konditioniert, da dann κ_{rel} auch deutlich größer als 1000 sein kann. Wir verweisen dazu auch auf das Beispiel von Seite 18.

Die Auslöschung kann in Kombination mit Datenfehlern infolge von Rundung die zentrale Ursache für numerische Instabilitäten und stark fehlerhafte Rechenergebnisse sein. Dies haben wir im Beispiel auf Seite 17 infolge der Differenz zwischen p und ξ_3 in Algorithmus *QuadGlgI* schon beobachtet, näheres dazu folgt auf Seite 41. Die Kondition eines Problems ist aber von etwaigen Lösungsalgorithmen völlig unabhängig. In diesem Zusammenhang bezeichnet man häufig auch das Produkt von Konditionszahl und Betrag des Datenfehlers, also

$$\kappa_{\text{abs}}|\tilde{x} - x| \quad \text{bzw.} \quad \kappa_{\text{rel}}\frac{|\tilde{x} - x|}{|x|}, \tag{3.42}$$

als *unvermeidbaren Fehler*, obwohl es sich dabei streng genommen um *obere Schranken* für den Betrag des jeweiligen Resultatfehlers handelt. Somit ist (3.42) jener Fehlerbetrag, mit dem bei der Lösung des Problems (φ, x) infolge von Datenfehlern – unabhängig vom gewählten Algorithmus – gerechnet werden *muss*. Zu beachten ist, dass diese Abschätzungen stets im Sinne von $\lesssim$ zu verstehen sind, nur bei linearen Abbildungen φ kann ja $\lesssim$ durch $\leq$ in (3.37) bzw. (3.40) ersetzt werden. Der Betrag des Resultatsfehlers liegt also *größenordnungsmäßig* nicht über (3.42), kann aber tatsächlich diese Schranken noch durchbrechen.

Kondition in höheren Dimensionen

Die Frage nach der Kondition eines Problems stellt sich natürlich nicht nur über $\mathbb{R}$ sondern insbesondere auch für Probleme, die durch Abbildungen

$$\varphi : D \subset \mathbb{R}^n \to \mathbb{R}^m,\ x \mapsto \varphi(x) \tag{3.43}$$

charakterisiert sind. Weichen wir etwa in (3.34) von unserer Vorstellung ab, dass z eine absolut fehlerfreie und feste Größe ist, fällt auch die Addition bereits mit

$$\varphi : \mathbb{R}^2 \to \mathbb{R},\ \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \mapsto x_1 + x_2 \tag{3.44}$$

in das mehrdimensionale Schema (3.43). Die Definition der Kondition von (φ, x) hängt in diesem Fall dann auch wesentlich von der Art der Fehlermessung ab. Dazu bemerken wir, dass allgemein der Fehler zwischen zwei Vektoren $\tilde{y}, y \in \mathbb{R}^d$ *komponentenweise* betrachtet werden kann, d.h. die Fehlerkonzepte (2.7) und (2.8) für reelle Zahlen einfach auf die einzelnen Komponenten $\tilde{y}_i - y_i$ des Vektors $\tilde{y} - y$ angewendet werden. Um den Fehler mit Hilfe einer einzigen Zahl anstelle von d Zahlen abschätzen zu können, führen wir in $\mathbb{R}^d$ den Begriff der Norm ein.

Definition

Norm. Eine *Norm* auf $\mathbb{R}^d$ ist eine Abbildung $\|\cdot\| : \mathbb{R}^d \to \mathbb{R}_0^+$, die für alle $y, \tilde{y} \in \mathbb{R}^d$ und $\lambda \in \mathbb{R}$ die Normkriterien

$$\begin{aligned} \|y\| = 0 &\iff y = 0, \\ \|\lambda \cdot y\| &= |\lambda| \cdot \|y\|, \\ \|y + \tilde{y}\| &\le \|y\| + \|\tilde{y}\| \end{aligned}$$

erfüllt.

Beispiel

Wichtige Normen auf $\mathbb{R}^d$ sind die Euklidsche[33] Norm

$$\|y\|_2 := \sqrt{\sum_{i=1}^{d} y_i^2} \tag{3.45}$$

und die Maximumsnorm

$$\|y\|_\infty := \max_{i=1,\ldots,d} |y_i|. \tag{3.46}$$

Beispiel

Sei $\tilde{y} \in \mathbb{R}^2$ eine Näherung des Vektors $y \in \mathbb{R}^2$. Dann kann der Fehler von $\tilde{y}$ unter Verwendung der Euklidschen Norm im *absoluten* Sinn durch $\|\tilde{y} - y\|_2$ und im

[33] Euklid von Alexandria: 365–300 v.Chr, griechischer Mathematiker. Versuchte die Mathematik (Geometrie) axiomatisch aufzubauen. Sein Hauptwerk „Die Elemente“ fasst das damals bekannte mathematische Wissen zusammen, in Band 7 findet sich der nach ihm benannte Euklidsche Algorithmus zum Berechnen des ggT zweier Zahlen, siehe auch Seite 66.

relativen Sinn, falls $y \neq 0$, durch $\frac{\|\tilde{y}-y\|_2}{\|y\|_2}$ angegeben werden. Bei komponentenweiser Betrachtung kann etwa der *relative* Fehler, falls $y_1, y_2 \neq 0$, durch Angabe der beiden Zahlen $\frac{|\tilde{y}_1-y_1|}{|y_1|}$ und $\frac{|\tilde{y}_2-y_2|}{|y_2|}$ beschrieben werden.

Auch im vektoriellen Fall kann die Rundungseinheit $\mathbf{u}$ als obere Schranke für den Rundungsfehler aufgefasst werden. Sei dazu $\tilde{y} = \mathrm{rd}(y)$ jener Vektor, der durch Rundung von $y \in \mathbb{R}^d$ entsteht, d.h. $\tilde{y}_i = \mathrm{rd}(y_i)$ für $i = 1, \ldots, d$. Aus (3.45), (3.46), (2.18) und (2.19) folgt dann

$$\begin{aligned} \|\tilde{y}-y\|_2 &= \sqrt{\sum_{i=1}^{d}(\tilde{y}_i-y_i)^2} = \sqrt{\sum_{i=1}^{d}\varepsilon_{y_i}^2 y_i^2} \leq \mathbf{u}\|y\|_2, \\ \|\tilde{y}-y\|_\infty &= \max_{i=1,\ldots,d}|\varepsilon_{y_i} y_i| \leq \mathbf{u}\|y\|_\infty. \end{aligned}$$

Mit $y \neq 0$ bzw. $y_i \neq 0$ gelten somit die Abschätzungen

$$\frac{|\tilde{y}_i-y_i|}{|y_i|} \leq \mathbf{u}, \quad \frac{\|\tilde{y}-y\|_2}{\|y\|_2} \leq \mathbf{u} \quad \text{und} \quad \frac{\|\tilde{y}-y\|_\infty}{\|y\|_\infty} \leq \mathbf{u}. \tag{3.47}$$

Ersetzen wir nun in (3.37) und (3.40) den Absolutbetrag durch Normen auf $\mathbb{R}^n$ und $\mathbb{R}^m$, gelangen wir zu folgenden Definitionen der Kondition.

Definition

Normweise Kondition. Es seien $\|\cdot\|_{\mathbb{R}^n}$ und $\|\cdot\|_{\mathbb{R}^m}$ Normen auf $\mathbb{R}^n$ bzw. $\mathbb{R}^m$ und φ wie in (3.43). Die *absolute Kondition* des Problems (φ, x) ist die kleinste Zahl κ_{abs}, für die

$$\|\varphi(\tilde{x})-\varphi(x)\|_{\mathbb{R}^m} \lesssim \kappa_{\mathrm{abs}}\|\tilde{x}-x\|_{\mathbb{R}^n} \quad \text{für } \tilde{x} \to x \tag{3.48}$$

erfüllt ist[34]. Zu $x \neq 0$ und $\varphi(x) \neq 0$ ist die *relative Kondition* des Problems (φ, x) die kleinste Zahl κ_{rel}, mit der

$$\frac{\|\varphi(\tilde{x})-\varphi(x)\|_{\mathbb{R}^m}}{\|\varphi(x)\|_{\mathbb{R}^m}} \lesssim \kappa_{\mathrm{rel}}\frac{\|\tilde{x}-x\|_{\mathbb{R}^n}}{\|x\|_{\mathbb{R}^n}} \quad \text{für } \tilde{x} \to x \tag{3.49}$$

gilt.

Die normweisen Konditionszahlen hängen also nicht nur von φ und x sondern auch von den gewählten Normen auf $\mathbb{R}^n$ und $\mathbb{R}^m$ ab.

[34] Die Notation $g(\tilde{x}) \lesssim h(\tilde{x})$ für $\tilde{x} \to x$ mit Funktionen $g, h : \mathbb{R}^n \to \mathbb{R}$ ist die Kurzschreibweise von $g(\tilde{x}) \leq h(\tilde{x}) + o(\|\tilde{x}-x\|_{\mathbb{R}^n})$ für $\tilde{x} \to x$, und $g(\tilde{x}) = o(h(\tilde{x}))$ für $\tilde{x} \to x$ ist die Kurzschreibweise von $\lim_{\tilde{x}\to x}\frac{|g(\tilde{x})|}{|h(\tilde{x})|} = 0$.

Beispiel

Wir betrachten erneut das Problem (3.44) der Addition zweier reeller Zahlen x_1 und x_2. Zunächst gilt

$$|\varphi(\tilde{x}) - \varphi(x)| \leq |\tilde{x}_1 - x_1| + |\tilde{x}_2 - x_2| \leq \sqrt{2}\sqrt{|\tilde{x}_1 - x_1|^2 + |\tilde{x}_2 - x_2|^2},$$

wobei die zweite Ungleichung aus $0 \leq (|\tilde{x}_1 - x_1| - |\tilde{x}_2 - x_2|)^2$ folgt. Für die absolute normweise Kondition (3.48) folgt bei Verwendung der Euklidschen Norm auf $\mathbb{R}^2$ dann jedenfalls $\kappa_{\text{abs}} \leq \sqrt{2}$, bei Verwendung der Maximumsnorm folgt $\kappa_{\text{abs}} \leq 2$. Tatsächlich gilt sogar die Gleichheit. Zum Nachweis betrachten wir $\delta \in \mathbb{R}^+$ und $\tilde{x}_1 = x_1 + \delta, \tilde{x}_2 = x_2 + \delta$, womit

$$|\varphi(\tilde{x}) - \varphi(x)| = 2\delta$$

gilt. Wegen $\|\tilde{x} - x\|_2 = \sqrt{2}\delta$ bzw. $\|\tilde{x} - x\|_\infty = \delta$ ergibt sich dann aber, dass $\kappa_{\text{abs}} = \sqrt{2}$ bzw. $\kappa_{\text{abs}} = 2$ gilt.

Für die relative normweise Konditionszahl (3.49) folgt dann unmittelbar

$$\kappa_{\text{rel}} = \sqrt{2}\frac{\sqrt{x_1^2 + x_2^2}}{|x_1 + x_2|} \quad \text{bzw.} \quad \kappa_{\text{rel}} = 2\frac{\max\limits_{i=1,2} |x_i|}{|x_1 + x_2|} \tag{3.50}$$

bei Verwendung von $\|\cdot\|_2$ bzw. $\|\cdot\|_\infty$ auf $\mathbb{R}^2$.

Das Problem der Addition zweier Zahlen mit annähernd gleichem Betrag aber unterschiedlichem Vorzeichen ist also im relativen Sinn schlecht konditioniert, da dann

$$|x_1 + x_2| \ll \sqrt{x_1^2 + x_2^2} \quad \text{bzw.} \quad |x_1 + x_2| \ll \max_{i=1,2} |x_i|$$

gilt[35], und κ_{rel} deutlich größer als 1 ist. Wie schon auf Seite 18 erwähnt, bezeichnet man diesen Effekt als Auslöschung.

Bei der Diskussion der Kondition in mehreren Dimensionen könnte man auch das komponentenweise Fehlerkonzept heranziehen und untersuchen, wie sich der Datenfehler von $\tilde{x}_i$ auf den Resultatsfehler von $\varphi_j(\tilde{x})$ auswirkt, wobei $\varphi_j(\tilde{x})$ die j-te Komponente des Vektors $\varphi(\tilde{x})$ bezeichnet. Dazu müsste man aber dann mit insgesamt $m \cdot n$ Konditionszahlen arbeiten, was in höheren Dimensionen schnell unübersichtlich und unpraktisch wird. In [3] findet man daher folgende Definition der *relativen komponentenweisen* Kondition, die mit einer einziger Konditionszahl ihr Auslangen findet.

Definition

Relative komponentenweise Kondition. Es sei φ wie in (3.43). Für $x_i \neq 0$ mit $i = 1, \ldots, n$ und $\varphi_j(x) \neq 0$ mit $j = 1, \ldots, m$ ist die *relative komponentenweise Kondition* des Problems (φ, x) die kleinste Zahl $\kappa_{\text{rel,komp}}$, für die

$$\max_{j=1,\ldots,m} \frac{|\varphi_j(\tilde{x}) - \varphi_j(x)|}{|\varphi_j(x)|} \lesssim \kappa_{\text{rel,komp}} \max_{i=1,\ldots,n} \frac{|\tilde{x}_i - x_i|}{|x_i|} \quad \text{für } \tilde{x} \to x \tag{3.51}$$

erfüllt ist.

[35] Wir verwenden $a \ll b$ als Kurzschreibweise für a ist viel kleiner als b.

Der Fall der absoluten komponentenweisen Kondition ist schon durch die normweise Kondition bei Verwendung der Maximumsnorm abgedeckt.

Welches Konzept man auch heranzieht, für stetig differenzierbare Abbildungen φ spielt der Mittelwertsatz der Differentialrechnung in seiner mehrdimensionalen Fassung eine wichtige Rolle, da er eine Darstellung des Fehlers $\varphi(\tilde{x}) - \varphi(x)$ mit Hilfe von Ableitungen von φ an der Stelle x ermöglicht. Bei Betrachtung der j-ten Komponente von φ existiert demnach ein $t_j \in (0,1)$, so dass

$$\varphi_j(\tilde{x}) - \varphi_j(x) = \sum_{i=1}^{n} \frac{\partial \varphi_j}{\partial x_i}(x + t_j(\tilde{x} - x)) \cdot (\tilde{x}_i - x_i) \tag{3.52}$$

gilt. Darin bezeichnet $\frac{\partial \varphi_j}{\partial x_i}(x)$ die partielle Ableitung der j-ten Komponentenfunktion $\varphi_j : \mathbb{R}^n \to \mathbb{R}$ nach x_i, die man durch gewöhnliche Differentiation nach x_i erhält, wenn man dabei alle Variablen x_k mit $k \neq i$ als konstant betrachtet.

Beispiel

Die Abbildung $\varphi : D \subset \mathbb{R}^2 \to \mathbb{R}$, $\binom{p}{q} \mapsto p - \sqrt{p^2 - q}$ aus (1.3) besitzt die partiellen Ableitungen

$$\frac{\partial \varphi}{\partial p}(p,q) = 1 - \frac{p}{\sqrt{p^2 - q}} \qquad \frac{\partial \varphi}{\partial q}(p,q) = \frac{1}{2\sqrt{p^2 - q}}. \tag{3.53}$$

Mit Hilfe eines ähnlichen Arguments wie in (3.35) lässt sich aus (3.52) die relative komponentenweise Kondition

$$\kappa_{\text{rel,komp}} = \frac{\max\limits_{j=1,\ldots,m} \sum\limits_{i=1}^{n} |\frac{\partial \varphi_j}{\partial x_i}(x)| \cdot |x_i|}{\|\varphi(x)\|_\infty} \tag{3.54}$$

folgern, wir verzichten jedoch auf die Details.

Beispiel

Wir untersuchen die relative komponentenweise Kondition des Problems der quadratischen Gleichung (1.3). Mit (3.53) und (3.54) gilt

$$\kappa_{\text{rel,komp}} = \frac{|\frac{\partial \varphi}{\partial p}(p,q)| \cdot |p| + |\frac{\partial \varphi}{\partial q}(p,q)| \cdot |q|}{|p - \sqrt{p^2 - q}|} = \left| \frac{p}{\sqrt{p^2 - q}} \right| + \left| \frac{p + \sqrt{p^2 - q}}{2\sqrt{p^2 - q}} \right|,$$

was deutlich zeigt, dass die Kondition von p und q abhängt. Da für $q < 0$ die Abschätzung

$$\kappa_{\text{rel,komp}} \leq 2 \tag{3.55}$$

folgt, ist das Problem in diesem Fall im relativen Sinn gut konditioniert. Schlechte relative Kondition liegt etwa vor, falls $q \approx p^2$ gilt – das bekannte Problem der Auslöschung.

Der Fehlervektor $\varphi(\tilde{x})-\varphi(x)$ kann mit Hilfe des Mittelwertsatzes auch normweise abgeschätzt werden, so dass auch die normweisen Konditionszahlen in (3.48) und (3.49) mit Hilfe der Ableitungen von φ beschrieben werden können. Dazu benötigen wir den Begriff der Jacobimatrix[36] von φ aus der Analysis und das Konzept einer Matrixnorm, welches uns erst in Band *2* zur Verfügung stehen wird.

4 Eigenschaften von Algorithmen

In den einführenden Beispielen haben wir bereits verschiedene Verfahren zur Lösung mathematischer Probleme kennengelernt. Wir stellen nun Eigenschaften von Algorithmen vor, die dann auch zu deren Qualitätsbeurteilung herangezogen werden können.

Korrektheit eines Algorithmus

In der *Korrektheitsanalyse* eines Algorithmus untersucht man die Fragestellung, ob er tatsächlich die gesuchte Lösung aus der zugehörigen Problemspezifikation berechnet.

Definition

Korrektheit eines Algorithmus[37]**.** Seien $\mathcal{I}_x$ und $\mathcal{O}_{x,y}$ die Ein- und Ausgabebedingung eines Problems. Ein Algorithmus *Algo* wird *korrekt bzgl. des Problems* genannt genau dann, wenn

$$\mathcal{O}_{x,Algo(x)} \quad \text{für alle } x \text{ mit } \mathcal{I}_x. \tag{4.56}$$

Die Aussage (4.56) heißt die *Korrektheitsaussage* für *Algo* (bzgl. des Problems), einen Beweis von (4.56) nennt man den *Korrektheitsbeweis* für *Algo* (bzgl. des Problems).

Beispiel

Der durch $QuadGlgI(p,q)$ berechnete Wert $p-\sqrt{p\cdot p-q}$ erfüllt *für alle* zulässigen Eingaben die Ausgabebedingung der zugehörigen Problemspezifikation, da für alle p,q mit $p^2-q\geq 0$

$$QuadGlgI(p,q)^2 - 2pQuadGlgI(p,q) + q = \\ (p-\sqrt{p\cdot p-q})^2 - 2p(p-\sqrt{p\cdot p-q}) + q = 0 \tag{4.57}$$

gilt. *QuadGlgI* ist somit ein korrekter Algorithmus für das auf Seite 3 spezifizierte Problem des Lösens einer quadratischen Gleichung. Wegen (1.6) ist auch *QuadGlgII* korrekt.

[36] Jacobi, Carl Gustav Jacob: 1804–1851, deutscher Mathematiker. Lieferte wichtige Beiträge zur Theorie der elliptischen Funktionen und partiellen Differenzialgleichungen.

[37] Der Begriff Korrektheit eines Algorithmus hat nichts mit jenem der Korrekt-Gestelltheit eines Problems zu tun.

Auf Seite 14 haben wir aber gesehen, dass der eben als korrekt bewiesene Algorithmus *QuadGlgI* bei der Gleitkommaausführung am Computer ein Resultat liefern kann, das die Ausgabebedingung der Spezifikation nicht erfüllt. In einer Korrektheitsanalyse werden jedoch Rundungsfehler, die bei Gleitkommarechnung entstehen können, ebenso wenig berücksichtigt wie etwaige Datenfehler. So bezieht sich der Korrektheitsbeweis (4.57) auf die in $\mathbb{R}$ bzw. $\mathbb{R}_0^+$ definierten Operationen $-$, $\cdot$ und $\sqrt{\cdot}$ und nicht auf deren Gleitkommarealisierungen.

Doch selbst bei Vernachlässigung von Rundungs- und Datenfehlern kann die Korrektheit eines an sich wohlüberlegten Algorithmus verletzt sein. Diesem Umstand sind wir etwa auf Seite 13 beim Algorithmus *ItWurzelII* zur Lösung der Gleichung $x^2 = r$ in $\mathbb{R}^+$ begegnet. Dort haben wir für die dem Algorithmus zugrunde liegende Iterationsfolge $(x_n)_{n\in\mathbb{N}}$ das Konvergenzresultat

$$\lim_{n\to\infty} x_n = \bar{x} \tag{4.58}$$

mit $\bar{x} = \sqrt{r}$ bewiesen. Zwar erfüllt der Grenzwert die Bedingung $x^2 = r$, der Algorithmus jedoch liefert nur einen Näherungswert x mit $|x - \sqrt{r}| \leq \eta$, der in der Regel $x^2 = r$ nicht erfüllt. Bezüglich der Problemstellung von Seite 9 ist *ItWurzelII* damit nicht korrekt. Bezüglich der modifizierten Problemstellung von Seite 14 ist *ItWurzelII* aber aufgrund des Konvergenzresultats (4.58) samt der Fehlerabschätzung (2.15) sehr wohl ein korrekter Algorithmus.

Bemerkung. Die Verletzung der Korrektheit durch vorzeitigen Abbruch ist typisch für iterative Verfahren, die auf einem Grenzwertresultat der Gestalt (4.58) beruhen. Genau aus jenem aber schöpft man das nötige Vertrauen in den Algorithmus.

Grenzwertüberlegungen spielen auch im Zusammenhang mit Algorithmen zur Lösung diskretisierter Ersatzprobleme eine Rolle. So haben wir uns etwa auf Seite 21 davon überzeugt, dass die Lösung U_n des diskretisierten Problems mit wachsendem Diskretisierungsparameter $n \in \mathbb{N}_0$ gegen die Lösung U des Ausgangsproblems konvergiert, also

$$\lim_{n\to\infty} U_n = U. \tag{4.59}$$

Ziel ist es in einer solchen Situation dann, zunächst einen bezüglich des diskretisierten Ersatzproblems korrekten Algorithmus zu entwickeln, in unserem Beispiel also ein Verfahren zu finden, das bei Eingabe von n die Lösung U_n ermittelt. Zwar ist für so ein Verfahren die Korrektheit bezüglich des Ausgangsproblems in der Regel nicht gegeben, die Konvergenzaussage (4.59) garantiert aber, dass es dann (bei geeignetem n) zumindest eine verlässliche Näherung an die eigentlich gesuchte Lösung des Ausgangsproblems liefert. Bei der Kreisumfangsberechnung kann man sich von der Korrektheit des Algorithmus *VieleckUmfang* bezüglich des Ersatzproblems der Umfangsberechnung des Vielecks mit Hilfe eines Induktionsarguments (bzgl. n) für den rekursiven Unteralgorithmus *SeitenlängeR* überzeugen, wir verzichten jedoch auf die Details.

Schließlich betrachten wir noch iterative Algorithmen, die nicht aus einem Grenzwertresultat wie (4.58) hervorgehen. Wir erinnern dazu an den Algorithmus *QuotRestN* zur Division mit Rest von Seite 6. Um die Korrektheit solcher Algorithmen zu überprüfen, identifiziert man eine sogenannte *Schleifeninvariante*. Dabei handelt es sich um eine über alle Schleifendurchläufe hinweg geltende Aussage, die allerdings

aus dem Algorithmus alleine nicht direkt ablesbar ist. Sie enthält zumeist die wesentliche Idee des Algorithmus und trägt damit zum Verständnis des Verfahrens bei. Die von einem Algorithmus grundsätzlich geforderte Termination nach endlich vielen Schritten wird ebenfalls im Zuge der Korrektheitsanalyse von Schleifenalgorithmen untersucht.

Beispiel

Korrektheit des Divisionsalgorithmus. Für den Algorithmus zur Division mit Rest *QuotRestN* von Seite 6 lautet die zu prüfende Korrektheitsaussage (4.56), dass für alle $m,n \in \mathbb{N}$

$$m = nq + r \quad \text{und} \quad r < n \quad \text{und} \quad q,r \in \mathbb{N}_0 \tag{4.60}$$

gilt, wobei $(q,r) = \mathit{QuotRestN}(m,n)$. Ein Kandidat für eine Schleifeninvariante zum Nachweis von (4.60) ist[38]

$$m = nq^{(i)} + r^{(i)} \quad \text{und} \quad q^{(i)},r^{(i)} \in \mathbb{N}_0. \tag{4.61}$$

Um zu zeigen, dass (4.61) nach jedem Durchlauf gilt, beginnen wir ähnlich wie in einem Induktionsbeweis bei $i = 0$ und schließen dann von i auf $i + 1$. Die Aussagen $m = nq^{(0)} + r^{(0)}$ und $q^{(0)},r^{(0)} \in \mathbb{N}_0$ sind klarerweise wahr, da nach dem 0-ten Schleifendurchlauf – also vor dem ersten Betreten der Schleife – ja $q^{(0)} = 0$ und $r^{(0)} = m$ ist. Es gelte nun (4.61) für fixes i, zusätzlich sei die Bedingung zum Abbruch der Schleife noch nicht erfüllt, d.h. $r^{(i)} \geq n$. Laut Algorithmus gilt dann am Ende des nächsten Schleifendurchlaufs

$$q^{(i+1)} = q^{(i)} + 1 \in \mathbb{N}_0 \quad \text{und} \quad r^{(i+1)} = r^{(i)} - n \in \mathbb{N}_0,$$

also gilt (4.61) auch für $i + 1$, weil

$$nq^{(i+1)} + r^{(i+1)} = n(q^{(i)} + 1) + (r^{(i)} - n) = nq^{(i)} + r^{(i)} = m.$$

Die Schleife bricht garantiert nach endlich vielen, etwa $k \in \mathbb{N}_0$, Schritten ab, da sonst die Folge $r^{(i)}$ eine unendliche, streng monoton fallende Folge in $\mathbb{N}_0$ wäre, was im Widerspruch dazu steht, dass $\mathbb{N}_0$ mit 0 ein kleinstes Element besitzt. Als Resultat liefert der Algorithmus dann $\mathit{QuotRestN}(m,n) = (q,r) = (q^{(k)},r^{(k)})$. Die Ursache des Abbruchs muss in der Verletztheit der Schleifenbedingung $r \geq n$ liegen, d.h. $r^{(k)} < n$, was zusammen mit (4.61) letztlich (4.60) sicherstellt.

Abschließend sei noch darauf hingewiesen, dass für die Korrektheit eines Algorithmus nur zulässige Eingaben in Betracht gezogen werden. Sie lässt offen, was passieren soll, wenn die Eingabebedingung nicht erfüllt ist. Sinnvollerweise wird man bei einer Realisierung am Computer zu Beginn des Programms die Eingabebedingung überprüfen, um zumindest Programmabstürze und Programmfehler zu verhindern, etwa Wurzelziehen aus einer negativen Zahl in $\mathbb{R}$ oder Division durch 0.

[38] Dabei stehen $q^{(i)}$ bzw. $r^{(i)}$ wieder für die Werte von q und r nach dem i-ten Schleifendurchlauf.

Stabilität

Auch wenn ein korrekter Algorithmus zur Lösung eines Problems (φ, x) vorliegt, seine Computerrealisierung kann infolge von Daten- und Rundungsfehlern dennoch zu einem Resultat $\tilde{\varphi}(\tilde{x})$ führen, das nicht mit $\varphi(x)$ übereinstimmt[39]. Dabei bezeichnet $\tilde{x}$ die fehlerhaften Daten, und die Abbildung $\tilde{\varphi}$ steht für die Computerrealisierung samt aller Fehler des gewählten Algorithmus.

Die Frage, ob $\tilde{\varphi}(\tilde{x})$ als *Näherungslösung*, also als eine hinreichend gute Näherung für die eigentlich gesuchte Lösung $\varphi(x)$ akzeptiert werden kann, ist Gegenstand der *Stabilitätsanalyse* eines Algorithmus. Von der Konditionsanalyse des Problems (φ, x) aus Abschnitt 3 wissen wir bereits, dass ein Datenfehler $\tilde{x} - x$ unabhängig vom Algorithmus zu einem unvermeidbaren Fehler $\varphi(\tilde{x}) - \varphi(x)$ im Resultat führt. Ausgehend von der Fehleraufschlüsselung

$$\tilde{\varphi}(\tilde{x}) - \varphi(x) = \tilde{\varphi}(\tilde{x}) - \varphi(\tilde{x}) + \varphi(\tilde{x}) - \varphi(x) \tag{4.62}$$

fordern wir daher nicht mehr, als dass der durch $\tilde{\varphi}$ verursachte Fehler $\tilde{\varphi}(\tilde{x}) - \varphi(\tilde{x})$ im Bereich des unvermeidbaren Fehlers $\varphi(\tilde{x}) - \varphi(x)$ bleibt.

Da Stabilitätsuntersuchungen meist im Zusammenhang mit numerischen Verfahren durchgeführt werden, konzentrieren wir uns auf durch

$$\varphi : D \subset \mathbb{R}^n \to \mathbb{R}^m,\ x \mapsto \varphi(x) \tag{4.63}$$

charakterisierte Probleme. Weiters betrachten wir in unserer Diskussion lediglich den Einfluss von Rundungsfehlern bei der Eingabe der Daten, also $\tilde{x} = \mathrm{rd}(x)$ wie im Beispiel auf Seite 31, und bei der Ausführung der einzelnen Rechenschritte in Gleitkommaarithmetik[40]. Dann ist nach (2.21) und (3.47) die Rundungseinheit $\mathbf{u}$ nicht nur eine obere Schranke für den relativen Fehler, der bei der Gleitkommarealisierung $\tilde{\diamond}$ einer Elementaroperation $\diamond \in \{+, -, \cdot, /\}$ am Computer entstehen kann, sondern auch für den relativen Fehler von $\tilde{x}$. Darüber hinaus ist bei so einer *Rundungsfehleranalyse* die obere Schranke für den unvermeidbaren Fehler durch $\kappa_{\mathrm{rel}}\mathbf{u}$ bzw. $\kappa_{\mathrm{rel,komp}}\mathbf{u}$ gegeben, siehe (3.42), (3.40) und (3.51).

Die Gleitkommaoperationen $\tilde{\diamond}$, und damit auch jede darauf aufbauende Algorithmenrealisierung $\tilde{\varphi}$, sind streng genommen nur auf den verfügbaren Maschinenzahlen, also einer diskreten Teilmenge der reellen Zahlen, definiert. Dennoch wird in der Regel nicht nur φ sondern auch $\tilde{\varphi}$ als eine Abbildung von $\mathbb{R}^n$ nach $\mathbb{R}^m$ aufgefasst. Damit stehen dann die Hilfsmittel der Analysis wie Grenzwertbildung und Differentiation zur Verfügung.

Bei der Rundungsfehleranalyse eines Algorithmus werden in der Regel untere und obere Schranken des am Computer darstellbaren Zahlbereichs ignoriert, und ein immer kleiner werdender relativer Abstand zwischen den Maschinenzahlen betrachtet. Zur Untersuchung des asymptotischen Verhaltens des Fehlers (4.62) bei immer kleiner werdender Rundungseinheit $\mathbf{u}$ gibt es dann mit Vorwärts- und Rückwärtsanalyse zwei prinzipielle Zugänge. In beiden Fällen kann sowohl mit dem normweisen als auch dem komponentenweisen Fehlerkonzept gearbeitet werden.

[39] Fehler im Resultat können natürlich auch durch Diskretisierung und/oder Abbruch von Iterationen entstehen.

[40] Der Rundungsfehler bei der Ergebnisausgabe ist im Zusammenhang mit Stabilität vernachlässigbar. Auf Diskretisierungs- und Abbruchsfehler gehen wir jeweils nur bei konkreten Algorithmen ein.

Aufgrund unserer Einschränkung auf Rundungsfehler betrachten wir dabei im folgenden stets den relativen Fehler.

Vorwärtsstabilität

Im Rahmen einer *Vorwärtsanalyse* untersucht man, wie sehr die Gleitkommarealisierung eines Algorithmus den unvermeidbaren Fehler[41] $\kappa_{\text{rel}}\mathbf{u}$ oder $\kappa_{\text{rel,komp}}\mathbf{u}$ verstärken kann.

Bezeichnung

Normweise Vorwärtsstabilität. Sei (φ,x) ein Problem mit $x \neq 0$ und $\varphi(\tilde{x}) \neq 0$ für alle $\tilde{x}$ mit $\frac{\|\tilde{x}-x\|_{\mathbb{R}^n}}{\|x\|_{\mathbb{R}^n}} \leq \mathbf{u}$. Eine Gleitkommarealisierung $\tilde{\varphi}$ eines Algorithmus zur Lösung des Problems (φ,x) genüge der Abschätzung

$$\frac{\|\tilde{\varphi}(\tilde{x}) - \varphi(\tilde{x})\|_{\mathbb{R}^m}}{\|\varphi(\tilde{x})\|_{\mathbb{R}^m}} \lesssim C_V \cdot \kappa_{\text{rel}}\mathbf{u} \quad \text{für } \mathbf{u} \to 0, \tag{4.64}$$

wobei wir die Konstante $C_V > 0$ als *Stabilitätszahl* bezeichnen. Ist C_V nicht allzu groß, so nennen wir $\tilde{\varphi}$ *stabil im Sinne der normweisen Vorwärtsanalyse*.

Die Stabilitätszahl C_V betrachten wir dabei als nicht zu groß, wenn sie die Anzahl der vom Algorithmus benötigten Elementaroperationen[42] nicht (wesentlich) übersteigt.

Die Vorwärtsstabilität einer Gleitkommarealisierung des Algorithmus bedeutet also, dass sie die ungefähre Lösung $\tilde{\varphi}(\tilde{x})$ zu einer zu (φ,x) benachbarten Fragestellung $(\varphi,\tilde{x})$ liefert. Das Produkt $C_V\kappa_{\text{rel}}$ beschreibt dabei, wie stark der maximale Rundungsfehler $\mathbf{u}$ verstärkt werden kann. Die Beurteilung der Vorwärtsstabilität von $\tilde{\varphi}$ erfordert also stets auch eine Konditionsanalyse zur Ermittlung von κ_{rel}. Die Normen zur Beschreibung des von $\tilde{\varphi}$ verursachten Fehlers müssen dann natürlich mit jenen zur Bestimmung von κ_{rel} übereinstimmen. Eine Algorithmenrealisierung, die zur Lösung von (φ,x) stabil ist, ist nicht zwangsläufig auch zur Lösung (φ,z) mit geänderten Daten z stabil.

Entsprechend kann im Fall $\varphi_j(\tilde{x}) \neq 0$ für $j = 1,\ldots,m$ der Begriff der *komponentenweisen Vorwärtsstabilität* eingeführt werden, wenn man (4.64) durch die Abschätzung

$$\max_{1\leq j\leq m} \frac{|\tilde{\varphi}_j(\tilde{x}) - \varphi_j(\tilde{x})|}{|\varphi_j(\tilde{x})|} \lesssim C_{V,\text{komp}} \cdot \kappa_{\text{rel,komp}}\mathbf{u} \quad \text{für } \mathbf{u} \to 0. \tag{4.65}$$

ersetzt.

[41] Streng genommen handelt es sich um eine obere Schranke des relativen, unvermeidbaren Fehlers, siehe auch Seite 29.

[42] Darunter verstehen wir hier in erster Linie alle am Computer unmittelbar zur Verfügung stehenden Operationen wie $+,-,\cdot,/$ aber auch $\sqrt{\cdot}$, $\sin(\cdot)$, ... Wir verweisen aber auch hier auf die Dehnbarkeit des Begriffes der Elementaroperation.

Beispiel

Vorwärtsstabilität der Subtraktion. Wir betrachten zunächst das *Problem* der Subtraktion zweier reeller Zahlen, also $\varphi(x_1,x_2) = x_1 - x_2$. Aus (3.50) folgt, dass für die relative Konditionszahl sowohl bzgl. der Euklidschen als auch der Maximumsnorm jedenfalls $\kappa_{\text{rel}} \geq 1$ gilt. Damit gilt für die Realisierung der Subtraktion, also für $\tilde{\varphi}(x_1,x_2) = x_1 \tilde{-} x_2$, nach (2.21) die Abschätzung

$$\frac{|\tilde{\varphi}(\tilde{x}_1,\tilde{x}_2) - \varphi(\tilde{x}_1,\tilde{x}_2)|}{|\varphi(\tilde{x}_1,\tilde{x}_2)|} \leq \mathbf{u} \leq \kappa_{\text{rel}}\mathbf{u}.$$

Da es sich bei der Subtraktion nur um *eine* Elementaroperation handelt, ist sie nach (4.64) stabil im Sinne der normweisen Vorwärtsanalyse mit Stabilitätszahl $C_V = 1$.

Dies gilt auch bei komponentenweiser Betrachtung, da ja aus (3.54) die Abschätzung

$$\kappa_{\text{rel,komp}} = \frac{|x_1| + |x_2|}{|x_1 - x_2|} \geq 1 \tag{4.66}$$

folgt. Insbesondere erfüllt die Subtraktion also die Abschätzung (4.65) mit

$$C_{V,\text{komp}} \cdot \kappa_{\text{rel,komp}} \leq 1. \tag{4.67}$$

Die *Gleitkommarealisierung* der Subtraktion ist somit stets vorwärtsstabil, auch in Situationen, in denen das *Problem* der Subtraktion infolge der Auslöschung schlecht konditioniert ist.

Bemerkung. Bereits auf Seite 18 haben wir erwähnt, dass die Auslöschung bei der Subtraktion häufig die Ursache für stark fehlerhafte Rechenergebnisse ist. Im Falle der Auslöschung werden nämlich bei der Subtraktion von $\tilde{x}_1$ und $\tilde{x}_2$ *bereits* in $\tilde{x}_1$ und $\tilde{x}_2$ *enthaltene Fehler* (infolge von Rundung bei der Eingabe oder zuvor gelöster Aufgaben mit $\tilde{x}_1$ und/oder $\tilde{x}_2$ als Näherungslösung) verstärkt. Sind die Daten für die Subtraktion jedoch exakt und auch ohne Rundungsfehler am Computer darstellbar, so entsteht bei der Subtraktion lediglich ein durch die Rundungseinheit $\mathbf{u}$ beschränkter Fehler.

Bei der Analyse der Vorwärtsstabilität ist es oft günstig, zunächst Unteralgorithmen zu untersuchen und daraus Rückschlüsse für die Realisierung des Gesamtalgorithmus zu ziehen. Wir veranschaulichen die Idee anhand einer Zerlegung in zwei Teilschritte, folgen dabei [3] und verwenden das komponentenweise Fehlerkonzept[43]. Dazu nehmen wir an, dass die Abbildung φ als Hintereinanderausführung zweier Abbildungen $\psi : \mathbb{R}^n \to \mathbb{R}^p$ und $\chi : \mathbb{R}^p \to \mathbb{R}^m$ darstellbar ist, also dass

$$\varphi = \chi \circ \psi : \mathbb{R}^n \to \mathbb{R}^m \tag{4.68}$$

gilt. Das Problem (φ,x) zerfällt damit in die zwei Teilprobleme (ψ,x) und $(\chi,\psi(x))$.

[43] Die Diskussion bei normweiser Fehlerbetrachtung ist analog.

Beispiel

Mit

$$\psi : \mathbb{R}^2 \to \mathbb{R}^2,\ \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \mapsto \begin{pmatrix} x_1 \\ \sqrt{x_1^2 - x_2} \end{pmatrix} \text{ und } \chi : \mathbb{R}^2 \to \mathbb{R},\ \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} \mapsto y_1 - y_2 \tag{4.69}$$

zerfällt das Problem (φ, p, q) aus (1.3) in die Teilprobleme (ψ, p, q) und $(\chi, \psi(p,q))$.

Unter der Annahme, dass für die Gleitkommarealisierungen $\tilde{\psi}$ und $\tilde{\chi}$ der entsprechenden Unteralgorithmen Abschätzungen (4.65) vorliegen, gibt das folgende Lemma Auskunft über eine obere Schranke für den Fehlerverstärkungsfaktor der Gleitkommarealisierung des Gesamtalgorithmus.

Lemma

Zur Aufspaltung (4.68) seien die relativen komponentenweisen Konditionszahlen (3.54) der beiden Teilprobleme mit κ_ψ und κ_χ bezeichnet. Weiters mögen für die Gleitkommarealisierungen $\tilde{\psi}$ und $\tilde{\chi}$ der entsprechenden Unteralgorithmen Abschätzungen[44] (4.65) mit den Stabilitätszahlen C_ψ bzw. C_χ vorliegen. Für die Hintereinanderausführung

$$\tilde{\varphi} = \tilde{\chi} \circ \tilde{\psi} \tag{4.70}$$

gilt dann die Abschätzung

$$\max_{1 \le j \le m} \frac{|\tilde{\varphi}_j(\tilde{x}) - \varphi_j(\tilde{x})|}{|\varphi_j(\tilde{x})|} \lesssim C_\varphi \cdot \kappa_\varphi \mathbf{u} \quad \text{für } \mathbf{u} \to 0$$

mit

$$C_\varphi \cdot \kappa_\varphi \le C_\chi \cdot \kappa_\chi + C_\psi \cdot \kappa_\psi \kappa_\chi. \tag{4.71}$$

Beweis. Es gilt

$$\begin{aligned}
|\tilde{\varphi}_j(\tilde{x}) - \varphi_j(\tilde{x})| &= |\tilde{\chi}_j(\tilde{\psi}(\tilde{x})) - \chi_j(\psi(\tilde{x}))| \\
&\le |\tilde{\chi}_j(\tilde{\psi}(\tilde{x})) - \chi_j(\tilde{\psi}(\tilde{x}))| + |\chi_j(\tilde{\psi}(\tilde{x})) - \chi_j(\psi(\tilde{x}))| \\
&= \frac{|\tilde{\chi}_j(\tilde{\psi}(\tilde{x})) - \chi_j(\tilde{\psi}(\tilde{x}))|}{|\chi_j(\tilde{\psi}(\tilde{x}))|} |\chi_j(\tilde{\psi}(\tilde{x}))| + \frac{|\chi_j(\tilde{\psi}(\tilde{x})) - \chi_j(\psi(\tilde{x}))|}{|\chi_j(\psi(\tilde{x}))|} |\chi_j(\psi(\tilde{x}))| \\
&\lesssim C_\chi \kappa_\chi \mathbf{u} |\chi_j(\tilde{\psi}(\tilde{x}))| + \kappa_\chi \max_{1 \le i \le p} \frac{|\tilde{\psi}_i(\tilde{x}) - \psi_i(\tilde{x})|}{|\psi_i(\tilde{x})|} |\chi_j(\psi(\tilde{x}))| \\
&\lesssim C_\chi \kappa_\chi \mathbf{u} |\chi_j(\tilde{\psi}(\tilde{x}))| + \kappa_\chi C_\psi \kappa_\psi \mathbf{u} |\chi_j(\psi(\tilde{x}))| \\
&\lesssim (C_\chi \kappa_\chi + \kappa_\chi C_\psi \kappa_\psi) \mathbf{u} |\varphi_j(\tilde{x})|
\end{aligned}$$

für $\mathbf{u} \to 0$, womit die Behauptung folgt. □

Der Faktor $C_\varphi \kappa_\varphi$, mit dem $\tilde{\varphi}$ den maximalen Rundungsfehler $\mathbf{u}$ verstärken kann, ist also durch $C_\chi \kappa_\chi + C_\psi \kappa_\psi \kappa_\chi$ beschränkt. Gelingt es, diesen Ausdruck selbst wieder in vernünftige Schranken zu weisen, kann über die Zerlegung in Teilalgorithmen die Gleitkommarealisierung $\tilde{\varphi}$ als vorwärtsstabil eingestuft werden. Andernfalls besteht

[44] Wir nehmen zusätzlich an, dass $\tilde{\chi}_j(\tilde{y}) \neq 0$ für alle $j = 1, \ldots, m$ und $\tilde{y}$ in einer hinreichend großen Umgebung um $\psi(x)$ gilt.

die Gefahr, dass es bei der Ausführung des Algorithmus in Gleitkommaarithmetik zu Verstärkungen des Rundungsfehlers jenseits jeglichen Stabilitätsempfindens kommt.

Beispiel

Fortsetzung. Wir kehren zu obigem Beispiel zurück und untersuchen die Stabilität der aus der Zerlegung (4.69) resultierenden Gleitkommarealisierung $\tilde{\varphi} = \tilde{\chi} \circ \tilde{\psi}$. Da es sich bei $\tilde{\chi}$ um die Gleitkommasubtraktion zweier reeller Zahlen handelt, wissen wir aus (4.67), dass stets $C_\chi \kappa_\chi \leq 1$ gilt. Ist aber das Problem der Subtraktion schlecht konditioniert, nimmt also κ_χ einen sehr großen Wert an, so kann selbst bei $C_\psi \kappa_\psi \leq 1$ die obere Schranke der Ungleichung (4.71) sehr hoch liegen. Unsere Diskussion der Kondition des Problems $\varphi(p,q)$ von Seite 33 hat gezeigt, dass für $q < 0$ gute Kondition, nämlich $\kappa_\varphi \leq 2$, vorliegt. Dies schließt jedoch die Situation $|q| \ll p^2$, für die nach (4.66) die Kondition κ_χ des Problems $(\chi, \psi(p,q))$ sehr hoch ist, nicht aus. Bei kleinem Wert von κ_φ aber großem Wert von κ_χ ist durch die Abschätzung (4.71) ein akzeptabler Wert von $C_\varphi \kappa_\varphi$ nicht garantiert, so dass die Gefahr starker Rundungsfehlerverstärkung durch $\tilde{\varphi}$ gegeben ist.

Diesem Szenario sind wir bereits auf den Seiten 14 und 17 bei Verwendung des Algorithmus *QuadGlgI* zur Lösung der Aufgabe $\varphi(5 \cdot 10^{11} - \frac{1}{20}, -10^{11})$ begegnet. Da *QuadGlgI* gerade auf der Aufschlüsselung (4.69) basiert, kann seine Gleitkommarealisierung nach obiger Diskussion nicht als stabil beurteilt werden, so dass trotz der Korrektheit von *QuadGlgI* die beobachtete Fehlerverstärkung um den Faktor $2.1990 \cdot 10^{12}$ nicht verwunderlich ist.

Handelt es sich bei $\tilde{\chi}$ in der Aufspaltung (4.70) um eine Gleitkommasubtraktion so besteht nach (4.71) infolge von Auslöschung, d.h. hoher Konditionszahl κ_χ, die Gefahr, dass die Fehlerverstärkung von $\tilde{\varphi}$ sehr hoch wird. Hingegen wirkt sich eine an erster Stelle liegende Subtraktion wegen $C_\psi \kappa_\psi \leq 1$ nicht negativ auf die Stabilität der Gleitkommarealisierung des Gesamtalgorithmus aus. Nach Möglichkeit sollen unvermeidbare Subtraktionen daher an den Anfang eines Algorithmus gestellt werden. Noch besser ist es, wenn auf die Subtraktion zweier ungefähr gleich großer Zahlen zur Gänze verzichtet werden kann, wie es etwa in den Beispielen rund um (2.30) und (1.6) der Fall war.

Sind über die Fehlerverstärkungsfaktoren $C_\chi \kappa_\chi$ und $C_\psi \kappa_\psi$ keine unmittelbaren Aussagen möglich, so kann nach der Zerlegung eines Algorithmus in zwei Teilschritte noch kein Urteil über die Stabilität seiner Realisierung gefällt werden. Dann ist es erforderlich, die Probleme (ψ, x) und $(\chi, \psi(x))$ so lange weiter zu unterteilen, bis die Fehlerverstärkungsfaktoren der entsprechenden Realisierungen der Teilalgorithmen bestimmt werden können, und damit auf das Verhalten der Realisierung des Gesamtalgorithmus zurück geschlossen werden kann.

Beispiel

Zerlegt man den Algorithmus *QuadlgII* zur Lösung der quadratischen Gleichung in seine 5 Teilschritte, so kann man bei wiederholter Anwendung des Arguments (4.71) zeigen, dass für $q < 0$ und $p > 0$ die Abschätzung (4.65) mit $C_{V,\text{komp}} \kappa_{\text{rel,komp}} = 4$ erfüllt ist. Der fortgepflanzte Rundungsfehler ist dann also durch $4\mathbf{u}$ beschränkt und damit nicht wesentlich größer als der unvermeidbare Fehler, der nach (3.55) durch $2\mathbf{u}$ abgeschätzt werden kann. Die Gleitkommarealisierung des Algorithmus *QuadGlgII* ist also für $q < 0$ und $p > 0$

stabil im Sinne der Vorwärtsanalyse, weshalb *QuadGlgII*$(5 \cdot 10^{11} - \frac{1}{20}, -10^{11})$ auf Seite 14 ein verlässliches Resultat geliefert hat.

Bemerkung. Wir betonen, dass die Stabilität eines Algorithmus stets auch von den vorliegenden Eingabegrößen abhängt. So tritt im Fall von $p < 0, q < 0$ und $|q| \ll p^2$ eine kritische Subtraktion nicht im Algorithmus *QuadGlgI* sondern bei *QuadGlgII* auf.

Rückwärtsstabilität

Eine Alternative zur Vorwärtsanalyse stellt die *Rückwärtsanalyse* eines Algorithmus dar, bei der man versucht, das vom Algorithmus bei Gleitkommaarithmetik gelieferte Resultat $\tilde{\varphi}(\tilde{x})$ als Lösung des Problems φ mit anderen Eingangsdaten als x darzustellen. Eine Konditionsanalyse des Problems ist dazu nicht zwingend erforderlich.

Bezeichnung

Normweise Rückwärtsstabilität. Gegeben seien ein Algorithmus zur Lösung des Problems (φ, x) mit $x \neq 0$ und $\tilde{x} \neq 0$ mit $\frac{\|\tilde{x}-x\|_{\mathbb{R}^n}}{\|x\|_{\mathbb{R}^n}} \leq \mathbf{u}$. Weiters seien $\hat{x}$ Daten mit

$$\tilde{\varphi}(\tilde{x}) = \varphi(\hat{x}), \tag{4.72}$$

worin $\tilde{\varphi}$ wieder für die Gleitkommarealisierung des Algorithmus steht. Gibt es mehrere $\hat{x}$, die (4.72) erfüllen, wählt man eines mit geringster Abweichung $\|\hat{x} - \tilde{x}\|_{\mathbb{R}^n}$ von den Daten $\tilde{x}$. Gilt dann

$$\frac{\|\hat{x} - \tilde{x}\|_{\mathbb{R}^n}}{\|\tilde{x}\|_{\mathbb{R}^n}} \lesssim C_R \cdot \mathbf{u} \quad \text{für } \mathbf{u} \to 0, \tag{4.73}$$

mit mäßig großer Stabilitätszahl $C_R > 0$, so heißt $\tilde{\varphi}$ *stabil im Sinne der Rückwärtsanalyse*.

Eine rückwärtsstabile Gleitkommarealisierung berechnet also wegen (4.72) die *exakte* Lösung zu einem benachbarten Problem. Rückwärtsstabilität ist somit ein strengeres Konzept als Vorwärtsstabilität, was für $\varphi(\tilde{x}) \neq 0$ auch anhand der Abschätzung

$$\frac{\|\tilde{\varphi}(\tilde{x}) - \varphi(\tilde{x})\|_{\mathbb{R}^m}}{\|\varphi(\tilde{x})\|_{\mathbb{R}^m}} = \frac{\|\varphi(\hat{x}) - \varphi(\tilde{x})\|_{\mathbb{R}^m}}{\|\varphi(\tilde{x})\|_{\mathbb{R}^m}} \lesssim \kappa_{\text{rel}} \frac{\|\hat{x} - \tilde{x}\|_{\mathbb{R}^n}}{\|\tilde{x}\|_{\mathbb{R}^n}} \lesssim C_R \cdot \kappa_{\text{rel}} \mathbf{u} \quad \text{für } \mathbf{u} \to 0$$

ersichtlich ist. Jede rückwärtsstabile Gleitkommarealisierung ist also mit der Wahl $C_V = C_R$ auch vorwärtsstabil, die Umkehrung dieser Aussage gilt im Allgemeinen aber nicht. Ist (4.73) nicht erfüllbar oder bereits eine Darstellung (4.72) nicht möglich, so heißt $\tilde{\varphi}$ instabil im Sinne der Rückwärtsanalyse.

Beispiel

Für die Gleitkommarealisierung der Subtraktion gilt nach Seite 16

$$\tilde{\varphi}(\tilde{x}) = \tilde{x}_1 \tilde{-} \tilde{x}_2 = (\tilde{x}_1 - \tilde{x}_2)(1 + \varepsilon_{\tilde{x}_1, \tilde{x}_2, -}) \text{ mit } |\varepsilon_{\tilde{x}_1, \tilde{x}_2, -}| \leq \mathbf{u}.$$

Das Resultat $\tilde{x}_1 \tilde{-} \tilde{x}_2$ lässt sich nun leicht mit Hilfe der Daten $\hat{x}_1 = (1 + \varepsilon_{\tilde{x}_1,\tilde{x}_2,-})\tilde{x}_1$ und $\hat{x}_2 = (1 + \varepsilon_{\tilde{x}_1,\tilde{x}_2,-})\tilde{x}_2$ als Lösung $\hat{x}_1 - \hat{x}_2$ des Problems $(\varphi,\hat{x})$ darstellen, womit (4.72) erfüllt ist. Weiters gilt $\|\tilde{x} - \hat{x}\|/\|\tilde{x}\| \leq \mathbf{u}$ (sowohl bei Verwendung der Euklidschen als auch der Maximumsnorm), somit ist die Subtraktion mit $C_R = 1$ rückwärtsstabil.

Zur Rückwärtsstabilitätsanalyse kann natürlich in analoger Weise auch das komponentenweise Fehlerkonzept herangezogen werden, wobei die Ungleichung (4.73) dann durch

$$\max_{i=1,\ldots,n} \frac{|\hat{x}_i - \tilde{x}_i|}{|\tilde{x}_i|} \lesssim C_{R,komp} \cdot \mathbf{u} \quad \text{für } \mathbf{u} \to 0 \tag{4.74}$$

zu ersetzen ist.

Komplexität

Bei der Entwicklung und Untersuchung eines Algorithmus interessiert man sich neben etwaig entstehenden Rechenfehlern auch dafür, mit welchem Speicherplatzbedarf und mit welchem Rechenaufwand die Ausführung des Algorithmus in Abhängigkeit von der *Größe der Eingaben* verbunden ist. Ein Algorithmus, der nach 100 Jahren Rechenzeit tatsächlich zum exakten Resultat führt, ist dabei genauso wenig nützlich, wie ein Algorithmus, der mit dem zur Verfügung stehenden Speicherplatz nicht auskommt oder in 10 Sekunden ein völlig unbrauchbares Ergebnis berechnet. Damit in Zusammenhang stehende Fragestellungen werden im Rahmen einer *Komplexitätsanalyse* untersucht. Da die Größe der Eingaben stets durch eine natürliche Zahl beschrieben wird, erweist sich die O-Notation für auf $\mathbb{N}$ definierte, reell-wertige Funktionen als nützlich.

Bezeichnung

Landau-Notation für reell-wertige Funktionen auf $\mathbb{N}$. Sei $f : \mathbb{N} \to \mathbb{R}$. Der Ausdruck $O(f(n))$ beschreibt eine Funktion $g : \mathbb{N} \to \mathbb{R}$, für die Konstanten $C \in \mathbb{R}^+$ und $N \in \mathbb{N}$ existieren, so dass

$$|g(n)| \leq C \cdot |f(n)| \quad \text{für } n \geq N. \tag{4.75}$$

Für (4.75) ist auch die Schreibweise $g(n) = O(f(n))$ üblich, und man sagt, $g(n)$ hat die *Ordnung* $f(n)$.

Intuitiv beschreibt $O(f(n))$ eine Funktion, die für große Inputwerte nicht wesentlich größere Funktionswerte als $f(n)$ annimmt. Manchmal interpretiert man (4.75) auch dahingehend, dass $O(f(n))$ eine Größe ist, die nicht schneller wächst als $f(n)$.

Beispiel

Jede konstante Funktion hat Ordnung $O(1)$. Ausserdem gilt $3n^2 + n^3 = O(n^3)$, weil $3n^2 + n^3 \leq 2n^3$ für alle $n \geq 3$ ist.

Wir betrachten im Folgenden die Analyse der *asymptotischen Zeitkomplexität*[45] eines Algorithmus, die beschreibt, wie sich die Rechenzeit mit größer werdenden Eingaben verhält. Dazu trifft man in der Regel Annahmen über den Zeitaufwand der Elementaroperationen des Algorithmus, und schließt dann aus der Anzahl der benötigten Elementaroperationen auf die Gesamtrechenzeit.

Beispiel

Näherung an π. Wir untersuchen den zeitlichen Rechenaufwand des Algorithmus *VieleckUmfang*, wobei wir die Eingabe $n \in \mathbb{N}_0$ selbst als Maß für deren Größe auffassen. Dazu treffen wir die Annahme, dass die arithmetischen Grundoperationen unabhängig von der Größe der jeweiligen Operanden in konstanter Zeit ausgeführt werden können[46]. Zur Ermittlung der Anzahl von Operationen erinnern wir daran, dass im Unteralgorithmus *SeitenlängeR*(n) genau n rekursive Aufrufe getätigt werden. Pro Aufruf sind 8 Elementaroperationen notwendig, zusammen mit den 3 Operationen nach dem Aufruf des Unteralgorithmus benötigt *VieleckUmfang*(n) demnach $8n + 3$ bzw. $O(n)$ arithmetische Operationen. Aufgrund unserer Annahme folgt daraus eine Zeitkomplexität $O(n)$. Verwenden wir anstelle des rekursiven Unteralgorithmus dessen Schleifenversion *SeitenlängeS*, so benötigt diese ebenfalls in jedem ihrer n Schleifendurchläufe 8 Operationen. Insgesamt ergibt das wieder $8n + 3$ arithmetische Grundoperationen und eine asymptotische Zeitkomplexität $O(n)$.

Die Zeitkomplexität $O(n)$ eines Algorithmus bedeutet, dass für große Werte von n die Rechenzeit höchstens linear mit n ansteigt. In Abbildung I.3 auf Seite 25 ist dieser lineare Zusammenhang sogar ab $n = 0$ gegeben. Angesichts des Aufwands von $8n + 3$ Operationen in beiden Varianten des Algorithmus stechen die laut Abbildung I.3 dennoch vorhandenen Rechenzeitunterschiede ins Auge. Eine Ursache dafür ist, dass wir den im Vergleich zur Schleife größeren Aufwand zur Abarbeitung der Rekursion in unserer Untersuchung vernachlässigt haben.

Die tatsächliche Zeit, die die Abarbeitung eines Algorithmus benötigt, ist natürlich auch von der Rechenleistung des verwendeten Computers abhängig. Am Supercomputer Eugene des Forschungszentrums Jülich können bis zu 223 Billionen Gleitkommaoperationen pro Sekunde[47] durchgeführt werden. Im Vergleich dazu schafft ein heute handelsüblicher PC einige Milliarden Gleitkommaoperationen pro Sekunde.

Im Beispiel der Annäherung von π konnten wir aus der Größe der Eingabe direkt und eindeutig auf die Anzahl der benötigten Rechenoperationen schließen. Die Anzahl der benötigten Rechenschritte eines Algorithmus kann aber selbst bei fixer *Größe* der Eingaben noch in Abhängigkeit von der konkreten Eingabe variieren. In solchen Fällen untersucht man dann typischerweise den *best möglichen Fall*, den *schlechtest möglichen Fall* oder den *durchschnittlich zu erwartenden Fall*.

[45] Bei der Analyse der Raumkomplexität untersucht man den bei der Abarbeitung des Algorithmus benötigten Speicherplatz.

[46] Wir werden in Abschnitt 5 sehen, dass abhängig vom Rechenmodell der Aufwand für elementare Rechenschritte von der Stellenanzahl der Operanden abhängen kann.

[47] Die englische Bezeichnung lautet floating point operations per second und wird mit FLOPS abgekürzt.

Beispiel

Für eine Komplexitätsanalyse des Algorithmus *QuotRestN*(m,n) von Seite 6 zur Division mit Rest betrachten wir $M = \max(m,n)$ als Größe der Eingabe. Der Algorithmus benötigt im günstigsten Fall $n = M$ und $m < n$ nur eine Vergleichsoperation $r \geq n$. Der ungünstigste Fall tritt andererseits für $n = 1$ und $m = M$ ein, wobei hier die Schleife M-mal durchlaufen werden muss, und in jedem Durchlauf zwei arithmetische Operationen und ein Vergleich durchzuführen sind. Zum Abbruch der Schleife wird noch ein Vergleich benötigt, so dass der Algorithmus insgesamt $3M + 1$ Operationen braucht. Für den durchschnittlich zu erwartenden Aufwand betrachten wir die $2M-1$ möglichen Eingabepaare (m,n), nämlich

- $M - 1$ Paare mit $n = M$ und $m < M$, in denen der Algorithmus nur eine Vergleichsoperation benötigt, und
- M Paare mit $n \leq M$ und $m = M$, in denen der Algorithmus $\lfloor \frac{m}{n} \rfloor$ Schleifendurchläufe benötigt[48]. Pro Durchlauf sind wie oben drei Operationen durchzuführen, und ein weiterer Vergleich wird zum Verlassen der Schleife gebraucht.

Die durchschnittliche Anzahl von Operationen ist somit

$$\frac{1}{2M-1}\left(M - 1 + \sum_{n=1}^{M}(3\left\lfloor \frac{M}{n} \right\rfloor + 1)\right) = 1 + \sum_{n=1}^{M} \frac{3}{2M-1}\left\lfloor \frac{M}{n} \right\rfloor \leq 1 + 3\sum_{n=1}^{M} \frac{1}{n}.$$

Mit $H_M := \sum_{n=1}^{M} \frac{1}{n}$ für die M-te *harmonische Zahl* benötigt der Algorithmus somit im Schnitt $O(H_M)$ Operationen. Da auch für die harmonischen Zahlen eine Näherung bekannt ist, nämlich $H_M = O(\ln(M))$, benötigt *QuotRestN*(m,n) also im Schnitt $O(\ln(\max(m,n)))$ Operationen.

Bei asymptotischen Komplexitätsaussagen ist stets zu beachten, dass die Ordnung eines Algorithmus dessen tatsächlichen Aufwand zwar beschränkt, jedoch nichts über die Größe der Konstanten C und N in (4.75) aussagt. Ist etwa der tatsächliche Aufwand eines $O(n^2)$-Algorithmus $1000n^2$, so ist dieser erst für Eingaben $n \geq 30$ besser als ein $O(2^n)$-Algorithmus mit Aufwand $\frac{1}{1000}2^n$. Obwohl man einen Algorithmus mit Komplexität $O(n^2)$ im Allgemeinen für besser einschätzt als einen mit Ordnung $O(2^n)$, kann für kleine Eingaben der asymptotisch schlechtere Algorithmus sehr wohl effizienter sein als der asymptotisch bessere.

Bemerkung. Es sei an dieser Stelle auch auf die spezielle Bedeutung von = im Zusammenhang mit Landau-Notationen hingewiesen. Der Ausdruck $g(n) = O(f(n))$ ist reine Notation und hat nichts mit einer mathematischen Gleichheit zu tun. Würden wir die üblichen Regeln für Gleichheit wie Symmetrie und Transitivität im Beisein von O benutzen, dann kämen wir mit $3n^2 + n^3 = O(n^3)$ und $n^3 = O(n^3)$ auf $3n^2 + n^3 = n^3$, was natürlich nicht gilt[49].

[48] Für $x \in \mathbb{R}$ steht $\lfloor x \rfloor$ (floor von x) für die größte ganze Zahl z mit $z \leq x$. Mit $\lceil x \rceil$ (ceiling von x) wird die kleinste ganze Zahl z mit $z \geq x$ bezeichnet.

[49] Dies gilt sinngemäß auch für die auf Seite 27 eingeführte o-Notation.

Übungsaufgaben

I.1 Spezifizieren Sie das aus der Analysis bekannte Problem der Berechnung des Grenzwerts einer reellen Zahlenfolge. Untersuchen Sie die Problemstellung hinsichtlich Existenz und Eindeutigkeit von Lösungen.

I.2 Spezifizieren Sie das aus der Linearen Algebra bekannte Problem der Berechnung der inversen Matrix. Untersuchen Sie die Problemstellung hinsichtlich Existenz und Eindeutigkeit von Lösungen.

I.3 Sei $\Phi : [0,1] \to [0,1]$ eine stetige Selbstabbildung. Zeigen Sie mit Hilfe des Zwischenwertsatzes die Existenz eines Fixpunktes in $[0,1]$.

I.4 Zeigen Sie, dass Φ aus (2.11) die Voraussetzungen des Banachschen Fixpunktsatzes auf $[1,2]$ mit $L = 1/2$ erfüllt.

I.5 Bestimmen Sie zu $\varphi : \mathbb{R}^2 \to \mathbb{R}$, $x \mapsto x_1 \cdot x_2$ die Konditionszahl $\kappa_{\text{rel,komp}}$ des Problems (φ, x) sowie κ_{abs} und κ_{rel} unter Verwendung von Maximums- und Euklidscher Norm.

I.6 Seien $x, \tilde{x} \in \mathbb{R}$ mit $\tilde{x} = \text{rd}(x)$. Zeigen Sie, dass

$$\mathbf{u}\tilde{x} \precsim \mathbf{u}x \text{ für } \mathbf{u} \to 0$$

gilt.

I.7 Zeigen Sie, dass bei fixem $z \in \mathbb{R}$ die Gleitkommarealisierung der Multiplikation $\varphi(x) = z \cdot x$ rückwärtsstabil ist. Unter welchen Voraussetzungen ist die Gleitkommarealisierung der Addition $\varphi(x) = z + x$ rückwärtsstabil?

I.8 Die Abbildung $\varphi : \mathbb{R} \to \mathbb{R}$ sei als Hintereinanderausführung der differenzierbaren Funktionen $\psi, \chi : \mathbb{R} \to \mathbb{R}$ darstellbar, also $\varphi = \chi \circ \psi$. Zeigen Sie, dass sich dann die relative Kondition des Gesamtproblems als Produkt der Konditionen der Teilprobleme darstellen lässt. Wie lautet die Abschätzung für die Stabilitätszahl C_φ, die sich damit aus (4.71) ergibt?

I.9 Wie lautet die relative Kondition der Auswertung von $\varphi(x) = \sin(x)$ für $x \to \pi/2$? Ist die Gleitkommaberechnung von $\sin(x)^2$ in der Nähe von $\pi/2$ vorwärtsstabil?

I.10 Beweisen Sie die Korrektheit des Algorithmus *SummeS* von Seite 94. Ermitteln Sie eine geeignete Schleifeninvariante und gehen Sie dann vor wie beim Korrektheitsbeweis von *QuotRestN* auf Seite 36.

I.11 Zeigen Sie, dass für jedes $m \in \mathbb{N}_0$ und beliebige $a_0, \ldots, a_m$ die Beziehung

$$a_0 + a_1 n + \cdots + a_m n^m = O(n^m)$$

gilt.

II Zahlbereiche

Zahlen sind die Grundbausteine der Mathematik. Wir besprechen daher in diesem Kapitel verschiedene Zahlenmengen und ihre Darstellung am Computer. Für eine Menge A und $n \in \mathbb{N}$ ist

$$A^n := \underbrace{A \times \cdots \times A}_{n \text{ mal}} := \{(a_1,\ldots,a_n) \mid a_i \in A \text{ für } 1 \leq i \leq n\},$$

wobei $(a_1,\ldots,a_n)$ für ein n-Tupel steht. Als Grundoperationen auf einem Tupel $t \in A^n$ betrachten wir die Bestimmung der Anzahl der Komponenten, oft auch die *Länge* von t genannt und mit $|t|$ bezeichnet, und den Zugriff t_i (für $1 \leq i \leq |t|$) auf die einzelnen Komponenten des Tupels. Tupel sind nicht in ihrer Länge fixiert, wir wollen auch Operationen zum Einfügen und Löschen von Elementen an bestimmten Positionen und zum Zusammenhängen zweier Tupel zu einem längeren Tupel verwenden. Für jedes Tupel t bezeichnen wir mit $t_{i:j}$ das Tupel $(t_i,\ldots,t_j)$, wobei dies für $i > j$ im leeren Tupel () resultiert. Zwei Tupel sind genau dann gleich, wenn sie gleiche Länge haben und alle Komponenten übereinstimmen.

Beispiel

Sei $t = (3,7,5) \in \mathbb{N}^3$. Dann ist $t_1 = 3$, $t_2 = 7$ und $t_3 = 5$. Klarerweise ist $|t| = 3$, weiters ist etwa $t_{2:3} = (7,5)$. Speziell ist $t_{3:3} = (5)$ ein Tupel der Länge 1 und $t_{3:2} = ()$ das leere Tupel. Zur Bildung von Tupel verwenden wir manchmal auch Schreibweisen wie

$$(k^2 \mid k = 1,\ldots,3) \quad \text{oder} \quad (k^2)_{k=1,\ldots,3},$$

die beide das Tupel $(1,4,9)$ beschreiben.

Tupel unterscheiden sich von endlichen Mengen im Wesentlichen darin, dass in einem Tupel die Reihenfolge der Elemente festgelegt ist, und dass in einem Tupel Elemente mehrfach auftreten können. Die Mengen $\{1,2\}$ und $\{2,1,1\}$ sind gleich, die Tupel $(1,2)$ und $(2,1,1)$ jedoch nicht.

Computerprogrammierung. *Mathematische Objekte können oft mit Hilfe von Tupel beschrieben werden. Zur Realisierung von Algorithmen, die mit solchen Objekten arbeiten, setzen wir daher eine Programmiersprache voraus, die Tupel mitsamt den eben eingeführten Grundoperationen zur Verfügung stellt. Insbesondere setzen wir Elementaroperationen EinfügenBeginn(t,e) und EinfügenEnde(t,e) zum*

Einfügen eines *Elements $e \in A$ am Beginn bzw. Ende eines Tupels $t \in A^n$ voraus. So liefert EinfügenBeginn(t,e) das Tupel $(e,t_1,\ldots,t_n)$ und EinfügenEnde(t,e) das Tupel $(t_1,\ldots,t_n,e)$. Durch diese Operationen nimmt also die Länge des Tupels um 1 zu. In Mathematica stehen für Tupel* Listen *mit einer Vielzahl von Listenoperationen bereit, in* MATLAB *können Tupel basierend auf Arrays bzw. Vektoren realisiert werden.*

5 Natürliche und ganze Zahlen

Mathematische Grundlagen

Die natürlichen Zahlen und darauf die Grundrechenoperationen Addition und Multiplikation können durch die Peano[1]-Axiome charakterisiert werden. Zum Rechnen erweist sich aber die Darstellung durch Ziffern bzgl. einer Basis als vorteilhaft. Wir alle sind mit dem Dezimalsystem mit Basis 10 und den Ziffern $0,\ldots,9$ vertraut, in dem wir etwa die Zahl 234 als

$$234 = 2\cdot 100 + 3\cdot 10 + 4 = 2\cdot 10^2 + 3\cdot 10^1 + 4\cdot 10^0 = \sum_{i=1}^{3} z_i 10^{3-i}$$

mit $z_1 = 2, z_2 = 3$ und $z_3 = 4$ schreiben. Es zeigt sich aber, dass auch jede andere natürliche Zahl außer 1 als Basis dienen kann.

Problemstellung (Basisdarstellung).

Gegeben:	$a,B \in \mathbb{N}$
mit:	$B \geq 2$.
Gesucht:	$L \in \mathbb{N}, z \in \{0,\ldots,B-1\}^L$
mit:	$a = \sum_{i=1}^{L} z_i B^{L-i}$ und $z_1 \neq 0$.

Die Existenz und Eindeutigkeit einer Lösung dieser Problemstellung garantiert der folgende Satz.

Satz

Basisdarstellung natürlicher Zahlen. Sei $B \in \mathbb{N}$ mit $B \geq 2$ und $a \in \mathbb{N}$. Dann existieren eindeutig bestimmte $L \in \mathbb{N}$ und $z_i \in \{0,\ldots,B-1\}$ für alle $1 \leq i \leq L$, so dass

$$a = \sum_{i=1}^{L} z_i B^{L-i} \tag{5.1}$$

mit $z_1 \neq 0$. Man spricht dabei von der B-adischen Darstellung von a.

[1] PEANO, GIUSEPPE: 1858–1932, italienischer Mathematiker. Gilt als einer der Begründer der symbolischen Logik und beschäftigte sich großteils mit den formalen Grundlagen der Mathematik, wie eben beispielsweise der Beschreibung der natürlichen Zahlen durch Mengen in den von ihm eingeführten Axiomen.

Beweis. Zum Beweis der eindeutigen Existenz der Darstellung verwenden wir Induktion[2] nach a. Sei daher nun $a \in \mathbb{N}$, als Induktionsannahme habe jede natürliche Zahl kleiner a eine eindeutige B-adische Darstellung. Durch Division mit Rest von a durch B, siehe den Satz auf Seite 4, erhalten wir eindeutig bestimmte $q, r \in \mathbb{N}_0$ mit $a = Bq + r$ und $r < B$. Ist $q = 0$, so lautet mit $L = 1$ und $z = (r)$ die eindeutige Basisdarstellung von a einfach $a = r = z_1 B^0 = \sum_{i=1}^{1} z_i B^{1-i}$. Falls $q > 0$, dann ist wegen $B \geq 2$ auch $q < Bq \leq a$, und laut Induktionsvoraussetzung existiert eine eindeutige B-adische Darstellung für q

$$q = \sum_{i=1}^{\eta} \zeta_i B^{\eta - i}$$

mit $\zeta_i \in \{0, \ldots, B-1\}$ für alle $1 \leq i \leq \eta$ und $\zeta_1 \neq 0$. Nun gilt

$$a = Bq + r = B \sum_{i=1}^{\eta} \zeta_i B^{\eta - i} + r = \sum_{i=1}^{\eta} \zeta_i B^{\eta + 1 - i} + rB^0$$

und mit $L = \eta + 1$, $z_L = r$ und $z_i = \zeta_i$ für alle $1 \leq i \leq \eta$ folgt daraus die B-adische Darstellung (5.1) für a.

Zum Nachweis der Eindeutigkeit nehmen wir an, $a = \sum_{i=1}^{l} z'_i B^{l-i}$ wäre eine weitere B-adische Darstellung für a, dann ist $a = B \sum_{i=1}^{l-1} z'_i B^{l-i-1} + z'_l$. Wegen der Eindeutigkeit von Quotient und Rest bei Division durch B muss also $z'_l = r = z_L$ und $q = \sum_{i=1}^{l-1} z'_i B^{l-i-1}$ die (eindeutige) Basisdarstellung von q sein. Damit ist auch $l = L$ gesichert und die Darstellung (5.1) ist eindeutig. □

Definition

Ziffern. In (5.1) heißen $z_1, \ldots, z_L$ die *Ziffern* von a (bzgl. der Basis B), z_1 ist dabei die *führende Ziffer* von a, und die Zahl L nennt man die *Stellenanzahl* bzw. die *Länge* von a (bzgl. B). Um deren Abhängigkeit von a und B zu betonen, schreibt man anstelle von L oft $L_B(a)$.

Ist die Basis aus dem Zusammenhang klar, so verzichtet man oft auf deren Angabe und schreibt einfach $L(a)$. Eine Zahl a in einem Ziffernsystem mit Basis B wird üblicherweise als $(z_1 \ldots z_L)_B$ geschrieben, im Spezialfall $B = 10$ oft auch einfach $z_1 \ldots z_L$. Ist eine natürliche Zahl a durch das Zifferntupel z bzgl. der Basis B dargestellt, so ist natürlich $L_B(a) = |z|$.

Beispiel

Die Zahl $a = 2 \cdot 10^2 + 3 \cdot 10 + 4$ wird im *Dezimalsystem*, d.h. $B = 10$, als $(234)_{10}$ oder kurz 234 geschrieben. Dieselbe Zahl kann wegen $234 = 3 \cdot 8^2 + 5 \cdot 8 + 2$ im *Oktalsystem* als $(352)_8$ oder wegen $234 = 2^7 + 2^6 + 2^5 + 2^3 + 2$ im Dualsystem als

[2] Wir verwenden hier eine Verallgemeinerung der vollständigen Induktion (dem Schluss von a auf $a+1$). Um eine Aussage $P(a)$ für alle $a \in \mathbb{N}$ zu beweisen, reicht es nämlich zu zeigen, dass *für fixes* $a \in \mathbb{N}$ die Aussage $P(a)$ gilt, unter der Voraussetzung, dass $P(b)$ für alle natürlichen Zahlen $b < a$ gilt, siehe etwa [1, Seite 216]. Es sei darauf hingewiesen, dass bei dieser Art des Induktionsbeweises der Induktionsanfang entfallen kann, da er im Induktionsschritt enthalten ist.

$(11101010)_2$ ausgedrückt werden. Somit ist $L_{10}(a) = L_8(a) = 3$ und $L_2(a) = 8$. Mit $B = 2^{31}$ steht $(54321\ 12345\ 67890)_{2^{31}}$ für die Zahl

$$54321 \cdot 2^{62} + 12345 \cdot 2^{31} + 67890 = 2505113962335048240356 34.$$

Einer Basisdarstellung mit Basis B liegt immer eine Menge von B unterscheidbaren Ziffern zugrunde, etwa $0, \ldots, 9$ im Dezimalsystem. Im *Hexadezimalsystem* mit $B = 16$ verwendet man üblicherweise die Ziffern $0, \ldots, 9, A, \ldots, F$. Bei $B = 2^{31}$ bräuchte man dann 2^{31} verschiedene Symbole. Wie im Beispiel kann man aber als „Ziffern" zur Basis 2^{31} die Zahlen $0, \ldots, 2^{31} - 1$ verwenden und diese ihrerseits im Dezimalsystem anschreiben.

Der obige Induktionsbeweis zur Existenz einer (eindeutigen) Basisdarstellung zeigt konstruktiv, wie zu gegebenen $a \in \mathbb{N}$ und Basis B die Länge L und das Zifferntupel z aus (5.1) ermittelt werden kann. Der rekursive Algorithmus *BasisdarstellungN* kann daraus sofort abgeleitet werden.

Algorithmus *BasisdarstellungN*: Basisdarstellung in $\mathbb{N}$

```
q ← a div B, r ← a mod B
if q = 0
   L ← 1, z ← (r)
else
   (η,ζ) ← BasisdarstellungN(q,B)
   z ← EinfügenEnde(ζ,r)
   L ← η + 1
return (L,z)
```

Aufruf: *BasisdarstellungN*(a,B)
Eingabe: $a,B \in \mathbb{N}$
mit: $B \geq 2$
Ausgabe: $L \in \mathbb{N}, z \in \{0, \ldots, B-1\}^L$
mit: $a = \sum_{i=1}^{L} z_i B^{L-i}$ und $z_1 \neq 0$.

Die Algorithmen zum Addieren und Multiplizieren von Zahlen in Dezimaldarstellung sind uns aus der Schule bekannt. Wir wollen nun einige algorithmisch interessante Aspekte dieser Verfahren etwas genauer unter die Lupe nehmen und beginnen mit der Addition. Zur Veranschaulichung soll ein einfaches Beispiel mit $B = 10$ dienen.

```
      9  7  3  1
+     0  8  2  9
   1  1  0  1  0    Übertrag
---------------------------
1  0  5  6  0
   ←  ←  ←  ←
```

In der bekannten Methode wird der kürzere der beiden Summanden bei Bedarf am Beginn mit Nullen gefüllt, so dass die Summanden gleiche Länge aufweisen. Die Ziffern der Summe erhält man dann, indem man von hinten nach vorne die Ziffern der Summanden addiert und den Übertrag aus der vorher berechneten Stelle dazu addiert. Dieses Verfahren wird oft als der *klassische Additionsalgorithmus* bezeichnet, es funktioniert auch für Basen $B \neq 10$ völlig analog. Durch den Übertrag, der bei der letzten Ziffernaddition entstehen kann, kann das Resultat um eine Ziffer mehr enthalten als der längere Summand.

Dieser Algorithmus beruht auf lediglich einer Elementaroperation, dem Addieren zweier Ziffern und eines Übertrags resultierend in einer Ziffer und einem Übertrag. In jeder Ziffernaddition ist unabhängig von der Basis B der Übertrag entweder 0 oder 1.

Beispiel

Im obigen Beispiel ergibt das Addieren der Ziffern 1 und 9 und dem Übertrag 0 das Resultat $1 + 9 + 0 = 10$. Den Übertrag 1 und die Resultatsziffer 0 erhält man daraus als Quotient und Rest bei Division von 10 durch $B = 10$. Die Ziffern 3 und 2 mit dem Übertrag 1 addieren zu 6, Division von 6 durch $B = 10$ ergeben Übertrag 0 und Ziffer 6.

Ähnlich zu dem eben gezeigten Verfahren gibt es eine Methode für die Subtraktion basierend auf der Zifferndarstellung der Operanden, auf die wir jedoch nicht näher eingehen. Stattdessen konzentrieren wir uns auf den *klassischen Multiplikationsalgorithmus* für Zifferntupel a und b, den wir zunächst im bekannten Schema zur Basis $B = 10$ veranschaulichen.

				a_1	a_2	a_3	a_4		b_1	b_2	b_3	
				↓	↓	↓	↓		↓	↓	↓	
				9	7	3	1	·	8	2	9	
			8	7	5	7	9					$a \cdot b_3$
+		1	9	4	6	2	0					$a \cdot b_2 \cdot 10^1$
+	7	7	8	4	8	0	0					$a \cdot b_1 \cdot 10^2$
	8	0	6	6	9	9	9					
	↑	↑	↑	↑	↑	↑	↑					
	z_1	z_2	z_3	z_4	z_5	z_6	z_7					

Allgemein wird für $B \geq 2$ das Produkt $\sum_{j=1}^{|a|} a_j B^{|a|-j} \cdot \sum_{i=1}^{|b|} b_i B^{|b|-i}$ unter Ausnutzung des Distributivgesetzes berechnet, indem man a einzeln mit $b_i B^{|b|-i}$ multipliziert und die so entstehenden Teilprodukte addiert. Dabei wiederum multipliziert man aufgrund des Distributivgesetzes jede Ziffer a_j mit b_i und addiert den Übertrag aus der vorhergehenden Stelle. Die nachfolgende Multiplikation mit $B^{|b|-i}$ kann dabei wie im Schema gezeigt durch Verschieben der Ziffern um $|b| - i$ Positionen nach links erreicht werden, was einem Auffüllen am Ende des Zifferntupels mit $|b| - i$ Nullen gleichkommt. Aus $a \cdot b < B^{L(a)} \cdot B^{L(b)} = B^{L(a)+L(b)}$ sieht man sofort, dass $L(a \cdot b) \leq L(a) + L(b)$ ist.

Die in diesem Verfahren benötigten Elementaroperationen sind das Multiplizieren zweier Ziffern resultierend in einer Ziffer und einem Übertrag, das Verschieben und das Addieren von Zahlen. Den Übertrag und die Resultatsziffer erhält man wieder als Quotient und Rest bei durch Division mit Rest durch die Basis B. Anders als bei der Addition kann es bei $B > 2$ auch zu einem Übertrag größer als 1 kommen. Bei $B = 2$ hingegen kann kein Übertrag entstehen, da das Produkt zweier Ziffern in diesem Fall immer einstellig ist.

Beispiel

Wir betrachten $a \cdot b_3$ im obigen Schema. Das Multiplizieren der Ziffern 1 und 9 mit anschließender Addition des Übertrags 0 ergibt $1 \cdot 9 + 0 = 9$. Daraus erhält

man als Quotient und Rest bei Division durch $B = 10$ den Übertrag 0 und die Resultatsziffer 9. Die nächste Ziffer ergibt sich aus $9 \cdot 3 + 0 = 27$ als 7 mit Übertrag 2, die folgende Ziffer wegen $9 \cdot 7 + 2 = 65$ als 5 mit Übertrag 6, und $9 \cdot 9 + 6 = 87$ liefert abschließend die Ziffern 7 und 8.

Als Grundoperation auf natürlichen und ganzen Zahlen gilt auch noch die Division mit Rest. Auf Seite 6 haben wir schon einen Algorithmus *QuotRestN* kennengelernt, der simultan den Quotient und Rest zweier natürlicher Zahlen basierend auf Addition und Subtraktion berechnet. Ähnlich der Spezifikation für natürliche Eingabewerte von Seite 2 spezifiziert man auch für $a, b \in \mathbb{Z}$ mit $b \neq 0$ den Quotient $q \in \mathbb{Z}$ und den Rest $r \in \mathbb{N}_0$ bei Division von a durch b so, dass gilt

$$a = bq + r \quad \text{und} \quad r \in \{0, \ldots, |b| - 1\}.$$

Auch für diese Problemstellung[3] sind q und r eindeutig bestimmt, das Schema zu deren Bestimmung setzen wir als aus der Schule bekannt voraus. In $\mathbb{Z}$ schreiben wir wieder a div b und a mod b für Quotient und Rest bei Division von a durch b.

Beispiel

Seien $a = -13$ und $b = 6$. Dann ist $-13 = 6 \cdot (-3) + 5$, somit ist a div $b = -3$ und a mod $b = 5$, vergleiche auch das Beispiel zur Division in $\mathbb{N}$ auf Seite 2.

Natürliche und ganze Zahlen am Computer

Wir beschäftigen uns nun damit, wie natürliche bzw. ganze Zahlen – etwa die Zahl 234, die im obigen Beispiel schon als Eingabe für den Algorithmus *BasisdarstellungN* verwendet wurde – auf einem Computer dargestellt werden, und wie mit ihnen gerechnet wird.

Natürliche und ganze Zahlen fixer Länge

Kern jeder effektiven Umsetzung der arithmetischen Grundoperationen am Computer ist eine fixe Stellenanzahl der Operanden, daher wollen wir die Darstellung solcher Zahlen an den Anfang stellen.

Computerrepräsentation (Natürliche und ganze Zahlen fixer Länge)**.** Zur Darstellung von natürlichen Zahlen am Computer wird $B = 2$ verwendet, die Stellenanzahl L in (5.1) mit einer Konstanten ω *fixiert* und die Forderung $z_1 \neq 0$ aufgehoben. Jede auf diese Weise darstellbare Zahl hat somit die Form

$$a = \sum_{i=1}^{\omega} z_i 2^{\omega - i} \quad \text{mit } z_i \in \{0, 1\},$$

[3] Es sei darauf hingewiesen, dass Quotient und Rest ganzer Zahlen manchmal auch so definiert wird, dass der Rest negativ sein kann.

wobei nun ω *nicht* von a abhängt, sondern durch physikalische Parameter des Prozessors vorgegeben ist. Man nennt die einzelnen Ziffern $z_i \in \{0,1\}$ auch *Bits* und ω die *Wortlänge*. Mit $N_{\min}$ und $N_{\max}$ bezeichnen wir die kleinste bzw. die größte auf diese Weise darstellbare Zahl. Klarerweise ist $N_{\min} = (00\ldots0)_2 = 0$ und $N_{\max} = (11\ldots1)_2 = 2^\omega - 1$. Obwohl auf die Forderung $z_1 \neq 0$ im Gegensatz zu (5.1) verzichtet wird, ist auch diese Darstellung eindeutig, weil das Tupel z im Fall $L(a) < \omega$ am Beginn genau $\omega - L(a)$ führende Nullen aufweist.

Zur Darstellung ganzer Zahlen verwendet man in der *Vorzeichen-Betrag-Darstellung*[4] z_1 für das Vorzeichen und $\omega - 1$ Bits für den Betrag der Zahl. Den darstellbaren Bereich beschreiben $Z_{\min} = -2^{\omega-1} + 1$ bzw. $Z_{\max} = 2^{\omega-1} - 1$.

Beispiel

Die Kenngrößen für die gebräuchlichen Wortlängen $\omega = 16, \omega = 32$ bzw. $\omega = 64$ sind in Tabelle II.1 zusammengefasst. Die mit Wortlänge $\omega = 16$ darstellbaren Zahlen sind in Tabelle II.2 gezeigt.

	Vorzeichenlose Darstellung		Vorzeichen-Betrag-Darstellung	
ω	$N_{\min}$	$N_{\max}$	$Z_{\min}$	$Z_{\max}$
16	0	$2^{16} - 1 = 65535$	$-2^{15} + 1 = -32767$	$2^{15} - 1 = 32767$
32	0	$2^{32} - 1 \approx 4.3 \cdot 10^9$	$-2^{31} + 1 \approx -2.1 \cdot 10^9$	$2^{31} - 1 \approx 2.1 \cdot 10^9$
64	0	$2^{64} - 1 \approx 1.8 \cdot 10^{19}$	$-2^{63} + 1 \approx -9 \cdot 10^{18}$	$2^{63} - 1 \approx 9 \cdot 10^{18}$

Tabelle II.1. Kenngrößen für $\omega = 16, \omega = 32$ bzw. $\omega = 64$.

Vorzeichenlose Darstellung		Vorzeichen-Betrag-Darstellung	
Ziffernfolge $z_1 \ldots z_{16}$	$a \in \mathbb{N}_0$	Ziffernfolge $z_1 \ldots z_{16}$	$a \in \mathbb{Z}$
0000000000000000	0	0000000000000000	0
0000000000000001	1	0000000000000001	1
0000000000000010	2	0000000000000010	2
0000000000000011	3	0000000000000011	3
⋮	⋮	⋮	⋮
0111111111111111	32767	0111111111111111	32767
1000000000000000	32768	1000000000000000	−0
1000000000000001	32769	1000000000000001	−1
⋮	⋮	⋮	⋮
1111111111111111	65535	1111111111111111	−32767

Tabelle II.2. Vorzeichenlose und Vorzeichen-Betrag-Darstellung mit 16 Bits.

Computerprogrammierung ($\mathbb{N}$ und $\mathbb{Z}$ in Programmiersprachen). *In vielen Programmiersprachen stehen bei maximaler Wortlänge ω des Prozessors auch spezielle Datentypen mit Wortlänge $\omega/2$ und/oder 2ω zur Verfügung. In manchen Sprachen hat man*

[4] In der Vorzeichen-Betrag-Darstellung gibt es zwei Möglichkeiten, die Zahl 0 darzustellen, nämlich −0 und +0. Dies ist in sogenannten *B-Komplement-Darstellungen* nicht der Fall, wodurch (im negativen Bereich) um eine Zahl mehr zur Verfügung steht. In der 2-Komplement-Darstellung sind mit Wortlänge $\omega = 16$ Dezimalzahlen zwischen −32768 und 32767 darstellbar, wir gehen allerdings auf Details nicht weiter ein.

auch die Wahl zwischen vorzeichenlosen Zahlen und solchen mit Vorzeichen. So bietet MATLAB *etwa* `int8`, `int16`, `int32` *oder* `int64` *für ganze Zahlen bzw.* `uint8`, *etc. für natürliche Zahlen verschiedener Wortlänge an. In C/C++ heißen die entsprechenden Datentypen* `int`, `short int`, `long`, `unsigned int`, *etc., wobei hier verschiedene Konventionen betreffend deren Länge gelten, auf die wir nicht näher eingehen. In Mathematica stehen Zahlen fixer Länge nicht zur Verfügung, zur Zahlendarstellung in Computeralgebrasystemen wie Mathematica verweisen wir auf den nächsten Abschnitt.*

Computerprogrammierung (Arithmetische Grundoperationen in Programmiersprachen)**.** *In einem Computerprozessor kommt zum Addieren zweier Zahlen a und b mit fixer Wortlänge ω der oben beschriebene klassische Algorithmus zum Einsatz. Die darin benötigte Elementaroperation zum Addieren zweier Ziffern und des Übertrags kann wegen der Wahl $B = 2$ durch einen einfachen elektronischen Bauteil realisiert werden, den sogenannten* Volladdierer. *Ingesamt werden ω Stück solcher Bauteile hintereinander geschaltet.*

Der klassische Additionsalgorithmus liefert nun ein Resultat z der Länge ω und einen Übertrag $c \in \{0,1\}$ (engl.: carry) aus der Addition der Ziffern a_1 und b_1 und dem Übertrag aus der zweiten Stelle. Zahlen fixer Länge ω sind daher bezüglich der Addition nicht abgeschlossen. Trotzdem stellen übliche Programmiersprachen eine Additionsoperation zur Verfügung, die die Summe $a + b$ als eine Zahl $\tilde{z}$ der Länge ω berechnet. Im Fall $c = 0$ ist $\tilde{z} = z$ das korrekte Ergebnis. Im Fall $c = 1$ kann aber $a + b$ nicht als Zahl der Länge ω dargestellt werden, somit muss $\tilde{z}$ fehlerhaft sein und man spricht von Überlauf *(engl.: overflow).*

Auch das Multiplizieren von Zahlen fixer Länge basiert im Prozessor auf dem klassischen Multiplikationsalgorithmus, auch bezüglich dieser Operation sind die Zahlen fixer Länge nicht abgeschlossen. Die Multiplikation zweier Ziffern kann wegen $B = 2$ wieder durch einen elektronischen Bauteil realisiert werden.

In manchen Programmiersprachen, etwa in C/C++, werden dabei zwei Zahlen der Länge ω zunächst intern zu einem Produkt der Länge 2ω multipliziert. Soll nun $a \cdot b$ als ω-stellige Zahl dargestellt werden, so spricht man im Fall $a \cdot b \geq 2^\omega$ auch hier von Überlauf, und das Resultat hängt von der gewählten Sprache ab. MATLAB *etwa liefert bei Überlauf die größte darstellbare Zahl $2^\omega - 1$.*

Auch für die Subtraktion und zur Berechnung von Quotient und Rest[5] *zweier Zahlen der Länge ω stellen übliche Programmiersprachen entsprechende Operationen zur Verfügung. Die dahinter liegenden Algorithmen orientieren sich an den aus der Schule bekannten Vefahren, für Details verweisen wir auf [7].*

Bei der Division mit Rest ist zu beachten, dass bei Berechnung des Quotienten der Rest als Nebenprodukt abfällt, so dass manche Sprachen es erlauben, Quotient und Rest simultan zu berechnen, wie wir es auch in QuotRestN getan haben. Wenn man sowohl a div *b als auch a* mod *b benötigt, ist dies natürlich effizienter als den Quotient und den Rest durch zwei separate Aufrufe zu berechnen.*

Beispiel

MATLAB liefert bei einem Überlauf als Resultat die größte darstellbare natürliche Zahl $\tilde{z} = 2^\omega - 1$, so ergibt etwa `uint8(250)+uint8(17)` die größte mit 8 Bits darstellbare Zahl $2^8 - 1 = 255$. Die Aufrufe `idivide(uint8(13),uint8(6))` und `mod(uint8(13),uint8(6))` liefern Quotient und Rest bei der Division von 13 durch 6.

[5] In der Praxis wird selbst für $a,b \in \mathbb{N}$ nicht der Algorithmus von Seite 6 eingesetzt, sondern ein auf der Zifferndarstellung beruhendes Verfahren.

Die Addition und die Subtraktion benötigen $O(\omega)$ Ziffernoperationen. Bei der Multiplikation muss im Wesentlichen jede Ziffer von a mit jeder Ziffer von b multipliziert werden, das sind ω^2 Ziffernmultiplikationen. Zusätzlich braucht man noch einige Additionen, deren Aufwand allerdings jenen der Multiplikationen nicht übersteigt, somit ergibt sich insgesamt ein Aufwand von $O(\omega^2)$ Ziffernoperationen. Details dazu werden auf Seite 60 klarer, wo wir den Algorithmus näher kennenlernen. Auch die Division mit Rest kann mit einem Aufwand von $O(\omega^2)$ Ziffernoperationen ausgeführt werden. Die Komplexität der Algorithmen hängt demnach rein von der Wortlänge ω und nicht von den eigentlichen Inputs a und b ab. Da die Wortlänge ω durch den Prozessor fixiert ist, kann man von konstantem Rechenaufwand für die Grundoperationen ausgehen.

Natürliche und ganze Zahlen beliebiger Länge

Das *Fixieren* einer Wortlänge ω zur Darstellung natürlicher und ganzer Zahlen bildet also die Grundlage für die effektive Umsetzung von Addition und Multiplikation am Computer mit Hilfe von elektronischen Bauteilen. Die Einschränkung des darstellbaren Bereichs durch $N_{\min}$ und $N_{\max}$ bzw. $Z_{\min}$ und $Z_{\max}$ und die damit verbundenen Rechenfehler bei Überlauf sind der Preis, der dafür zu zahlen ist. Bei *Zahlen beliebiger Länge* wird durch Wahl einer geeigneten *Datenstruktur* die Stellenanzahl einer Zahl am Computer *nicht* a-priori fixiert. Eine natürliche Zahl wird dabei wie in (5.1) durch ihr (beliebig langes) Zifferntupel bzgl. einer geeigneten Basis B dargestellt, für ganze Zahlen wird zusätzlich dazu das Vorzeichen abgespeichert. Die klassischen Algorithmen können dann aber nicht mehr in elektronischen Bauteilen realisiert werden, sondern müssen in *Programmen* umgesetzt werden, die die variable Länge der Eingaben verarbeiten können. Zahlen fixer Länge ω und 2ω und die arithmetischen Grundoperationen auf solchen Zahlen setzt man dabei als vorhanden voraus.

Computerrepräsentation (Datenstruktur $\mathcal{Z}_B$ für ganze Zahlen beliebiger Länge)**.** Für die Darstellung ganzer Zahlen am Computer definieren wir eine Datenstruktur $\mathcal{Z}_B$, in der für positive Zahlen das Zifferntupel zur Basis B aus (5.1) abgespeichert wird. Die Zahl $0 \in \mathbb{Z}$ wird durch $() \in \mathcal{Z}_B$ dargestellt, für negative Zahlen wird das Vorzeichen mit der führenden Ziffer abgespeichert. Wir identifizieren $z \in \mathcal{Z}_B$ mit dem darin enthaltenen Zifferntupel[6], was die Anwendung aller auf Tupel zur Verfügung stehenden Operationen auch auf z erlaubt. Die von $z = (z_1, \ldots, z_{L(z)}) \in \mathcal{Z}_B$ beschriebene ganze Zahl $a \in \mathbb{Z}\{0\}$ ist dann durch

$$a = \begin{cases} \sum_{i=1}^{L(z)} z_i B^{L(z)-i} & \text{falls } z_1 > 0 \\ -(|z_1| + \sum_{i=2}^{L(z)} z_i B^{L(z)-i}) & \text{falls } z_1 < 0 \end{cases} \tag{5.2}$$

gegeben. Man nennt $\mathcal{Z}_B$ *ganze Zahlen beliebiger Länge*, im Englischen wird meist von *(arbitrarily) long integers* gesprochen. Wir werden also auch Zifferntupel in $\mathcal{Z}_B$ der Einfachheit halber meist Zahlen nennen.

Manchmal wird man auch Zifferntupel mit führenden Nullen betrachten, in denen dann die Information über das Vorzeichen in der ersten von Null verschiedenen

[6] Das Symbol $\mathcal{Z}_B$ deutet an, dass auch die Basis B Teil der Datenstruktur ist. Im Zusammenhang mit Zahlen beliebiger Länge hat ein Wechsel der Basis keine Relevanz. Daher wird man in einer Realisierung von $\mathcal{Z}_B$ die Basis B als globalen Parameter halten anstatt sie mit jedem Zifferntupel abzuspeichern.

Ziffer steckt. Um die Eindeutigkeit der Darstellung zu gewährleisten, speichert man in $\mathcal{Z}_B$ aber immer die *kanonische Form*[7] eines Zifferntupels z, in der entweder $z = ()$ oder $z_1 \neq 0$ gilt. Für ein Tupel $z = ()$ oder $z \in \{-B+1, \ldots, 0, \ldots, B-1\}^n$ mit $z_i \geq 0$ für $i \geq 2$ und beliebigem $n \in \mathbb{N}$ ermittelt

$$\text{kanonisch}_{\mathcal{Z}_B}(z) := \begin{cases} \text{kanonisch}_{\mathcal{Z}_B}(z_{2:n}) & \text{falls } z \neq () \text{ und } z_1 = 0 \\ z & \text{sonst} \end{cases}$$

rekursiv die kanonische Form in $\mathcal{Z}_B$.

Als Basis kann im Prinzip jede natürliche Zahl $B \geq 2$ dienen, in der Praxis wählt man bei Wortlänge ω des verwendeten Computers $B = 2^{\omega-1}$. In diesem Fall sind die einzelnen Ziffern z_i nicht länger als $\omega - 1$ und damit jedenfalls als natürliche Zahlen fixer Länge ω darstellbar. Man hat somit in jeder Ziffer 1 Bit als Reserve, das im klassischen Additionsalgorithmus auf $\mathcal{Z}_B$ gebraucht wird, um Ziffernoperationen auf Zahlen fixer Länge *ohne Überlauf* ausführen zu können, siehe Seite 58 für die Details. In der führenden Ziffer kann das Reserve-Bit zum Abspeichern des Vorzeichens verwendet werden.

Beispiel

$(54321, 12345, 67890) \in \mathcal{Z}_{2^{31}}$ stellt die Zahl

$$a = 54321 \cdot 2^{62} + 12345 \cdot 2^{31} + 67890 = 250511396233504824035634 \in \mathbb{Z}$$

dar. $(-54321, 12345, 67890)$ steht dann für $-a$.

Computerrepräsentation (Datenstrukturen)**.** Datenstrukturen stellen eine Möglichkeit dar, Daten[8] strukturiert zusammenzufassen und von anderen Daten unterscheidbar zu machen. Für jede Datenstruktur definiert man Operationen, um aus den Einzelteilen ein neues Objekt zusammenbauen bzw. auf die Einzelteile eines Objektes zugreifen zu können. Wenn wir neue mathematische Objekte einführen, werden wir parallel dazu immer entsprechende Datenstrukturen beschreiben, die zur Darstellung dieser Objekte (am Computer) geeignet sind. Die konkrete Umsetzung von Datenstrukturen, deren Aufbau und der Zugriff auf deren Bestandteile hängen von der verwendeten Programmiersprache ab und werden hier nicht besprochen. Die Wahl einer Datenstruktur zur Darstellung eines mathematischen Objekts orientiert sich nicht nur an der Speicherung der Daten, sie hat auch Einfluss darauf, wie einfach/kompliziert die auf den mathematischen Objekten auszuführenden Operationen – im Fall von $\mathcal{Z}_B$ etwa die arithmetischen Grundoperationen – zu realisieren sind.

Beispiel

Sei $z = (54321, 12345, 67890) \in \mathcal{Z}_{2^{31}}$. Die einzelnen Ziffern $54321, 12345, 67890$ von z können wir mit z_1, z_2, z_3 ansprechen, die Basis 2^{31} erkennen wir aus dem Zusammenhang, da $z \in \mathcal{Z}_{2^{31}}$.

[7] Anstelle von kanonischer Form könnte man auch von einer *Standardform* sprechen.

[8] Das Wort „Daten" ist hier in seiner allgemeinen Bedeutung für „Informationen" verwendet, es handelt sich dabei nicht notwendigerweise um die Eingangs*daten* eines Problems, über die wir in Kapitel I ausführlich gesprochen haben.

Wir wenden uns nun den arithmetischen Grundoperationen auf ganzen Zahlen beliebiger Länge zu und beginnen mit der Addition.

Problemstellung (Addition).

Gegeben: $a,b \in \mathcal{Z}_B$.
Gesucht: $z \in \mathcal{Z}_B$
mit: $\sum_{i=1}^{L(z)} z_i B^{L(z)-i} = \sum_{i=1}^{L(a)} a_i B^{L(a)-i} + \sum_{i=1}^{L(b)} b_i B^{L(b)-i}$.

Der klassische Additionsalgorithmus kann auch für $a,b \in \mathcal{Z}_B$ herangezogen werden. Wir beschränken uns der Einfachheit halber auf die Addition zweier Zahlen $a,b \in \mathcal{Z}_B$ mit $L(a) \geq L(b)$ und betrachten im Algorithmus *AddZ* lediglich den Fall strikt positiver Zahlen, also $a_1,b_1 > 0$, die Verallgemeinerung auf 0 und negative Zahlen als mögliche Eingaben stellt keine wesentliche Schwierigkeit dar. Man füllt bei Verwendung von $\mathcal{Z}_B$ anders als auf Seite 50 den kürzeren Summanden nicht mit Nullen auf die Länge des längeren auf, sondern berücksichtigt die Länge der beiden Summanden beim Abbruch der Schleifen.

Algorithmus *AddZ*: Klassischer Additionsalgorithmus in $\mathbb{Z}$

```
c ← 0, z ← ()
for i from 0 to L(b) − 1
   s ← a_{L(a)−i} + b_{L(b)−i} + c
   c ← s div B, d ← s mod B
   z ← EinfügenBeginn(z,d)
for i from L(b) to L(a) − 1
   s ← a_{L(a)−i} + c
   c ← s div B, d ← s mod B
   z ← EinfügenBeginn(z,d)
if c = 1
   z ← EinfügenBeginn(z,c)
return z
```

Aufruf: $AddZ(a,b)$
Eingabe: $a,b \in \mathcal{Z}_B$
mit: $a_1,b_1 > 0$ und $L(a) \geq L(b)$.
Ausgabe: $z \in \mathcal{Z}_B$
mit: $\sum_{i=1}^{L(z)} z_i B^{L(z)-i} = \sum_{i=1}^{L(a)} a_i B^{L(a)-i} + \sum_{i=1}^{L(b)} b_i B^{L(b)-i}$.

Beispiel

Mit $B = 2$ stehen $a = (1,1,1,1,1,0,1,0) \in \mathcal{Z}_2$ und $b = (1,0,0,0,1) \in \mathcal{Z}_2$ für 250 bzw. 17. $AddZ(a,b)$ liefert $(1,0,0,0,0,1,0,1,1) \in \mathcal{Z}_2$, was dem korrekten Resultat 267 entspricht, vergleiche dazu das Beispiel mit `uint8` von Seite 54.

Beispiel

Auf einem Computer mit Wortlänge $\omega = 32$ wählt man $B = 2^{31}$. Mit den Eingaben $a = (54321,12345,67890) \in \mathcal{Z}_{2^{31}}$ und $b = \left(2^{31}-1,2,2^{31}-1\right) \in \mathcal{Z}_{2^{31}}$ berechnet $AddZ(a,b)$ die Summe

$$(1,54320,12348,67889) \in \mathcal{Z}_{2^{31}}.$$

Die letzte Resultatsziffer 67889 ergibt sich dabei als Rest bei Division von

$$67890 + 2^{31} - 1 = 2^{31} \cdot 1 + 67889$$

durch 2^{31}, der dabei entstehende Quotient 1 ist der Übertrag für die nächste Stelle.

Interessant in *AddZ* ist vor allem die Addition $a_{L(a)-i} + b_{L(b)-i} + c$ der Ziffern und des Übertrags c. Im Standardfall $B = 2^{\omega-1}$ ist wegen $c \leq 1$, siehe Seite 51,

$$s = a_{L(a)-i} + b_{L(b)-i} + c \leq 2(2^{\omega-1} - 1) + 1 = 2^{\omega} - 1,$$

und damit $L(s) \leq \omega$, so dass diese Elementaroperation auf den zugrunde liegenden Zahlen fixer Länge ω ohne Überlauf ausgeführt werden kann. Dies erklärt letztlich die spezielle Wahl von B. Da der Übertrag bei der Addition stets nur 0 oder 1 ist, gilt für die Aufspaltung von s in einen Übertrag c und ein $(\omega - 1)$-stelliges Resultat d

$$(c,d) = \begin{cases} (1, s - B) & \text{falls } s \geq B \\ (0, s) & \text{sonst.} \end{cases} \tag{5.3}$$

Daher empfiehlt sich in *AddZ* zur Division mit Rest die Verwendung des Algorithmus *QuotRestN* von Seite 6 auf Zahlen fixer Länge ω , da dieser genau auf (5.3) beruht.

Als Gesamtergebnis liefert *AddZ* anstelle eines Resultats mit Übertrag wie auf Seite 54 dann lediglich *eine* Zahl, weil im Falle eines Übertrags $c = 1$ nach Verlassen der Schleife dieser als führende Ziffer im Resultat eingefügt wird.

Für die Addition von Zahlen a und b beliebiger Länge benötigt *AddZ* im Durchschnitt $O(\min(L(a), L(b)))$ Elementaroperationen, im schlechtesten Fall sind $O(\max(L(a), L(b)))$ Operationen notwendig.

Vereinbarung (Mathematische Objekte und Datenstrukturen)**.** Prinzipiell wollen wir ein mathematisch abstrakt definiertes Objekt – wie $\mathbb{Z}$ – und Datenstrukturen zur Darstellung dieses Objekts – wie $\mathcal{Z}_B$ oder `int16` – auseinanderhalten. Stehen aber für einen abstrakten Bereich M mehrere Datenstrukturen zur Realisierung bereit, so wollen wir im algorithmischen Kontext die abstrakte Bezeichnung M auch als Synonym für „eine (beliebige) Datenstruktur für M“ verwenden. Schreiben wir also in einem Algorithmus „gegeben $z \in \mathbb{Z}$“, so lassen wir offen, ob z in $\mathcal{Z}_B$ oder `int16` repräsentiert sein soll. Schreiben wir hingegen „gegeben $z \in \mathcal{Z}_B$“, so deutet dies darauf hin, dass der entsprechende Algorithmus nur unter Verwendung der Datenstruktur $\mathcal{Z}_B$ funktioniert, oder es zumindest empfehlenswert ist, die Datenstruktur $\mathcal{Z}_B$ zu verwenden.

Computerprogrammierung (Polymorphismus)**.** *Wir haben bisher Algorithmen zum Addieren von Zahlen fixer und beliebiger Länge kennengelernt. In mathematischen Texten schreibt man jedoch meist einfach „+“ für die Addition, auch wenn dahinter verschiedene Operationen stecken können. In der Programmierung versteht man unter* Polymorphismus *eine Möglichkeit,* mehrere Definitionen *für* einen Funktionsnamen *in Abhängigkeit von den Operanden zuzulassen. In der Umsetzung mathematischer Algorithmen kann man sich dies zu Nutze machen, wenn arithmetische Operationen, etwa „+“, auf verschiedenen Bereichen, etwa $\mathbb{N}$ und $\mathbb{Z}$, definiert sind. Anstelle von verschiedenen Funktionsnamen für die dahinterliegenden Algorithmen kann man in Sprachen mit Polymorphismus einfach* einen Funktionsnamen *verwenden, etwa Add oder sogar „+“, und diesen Namen* überladen.

Wenn wir also in Zukunft in einem Algorithmus eine Operation wie $a + b$ verwenden, dann setzen wir voraus, dass ein entsprechender Algorithmus verfügbar ist, der diese Operation auf den Daten a und b realisiert, ohne den Algorithmus beim Namen nennen zu müssen. Wir sind übrigens schon in AddZ bei der zugrunde liegenden Ziffernaddition so vorgegangen, wo + für die Addition auf Zahlen fixer Länge ω steht. Falls die verwendete Programmiersprache Polymorphismus nicht unterstützt, so ist es Aufgabe des Programmierers, aus dem Zusammenhang zu erkennen, welcher Algorithmus hinter $a + b$ steckt und diesen dann etwa bei $a,b \in \mathcal{Z}_B$ als $AddZ(a,b)$ explizit aufzurufen.

Auch die Multiplikation zweier Zahlen beliebiger Länge orientiert sich am bekannten Verfahren aus der Schule.

Problemstellung (Multiplikation).

Gegeben: $a,b \in \mathcal{Z}_B$.
Gesucht: $z \in \mathcal{Z}_B$
mit: $\sum_{i=1}^{L(z)} z_i B^{L(z)-i} = \sum_{i=1}^{L(a)} a_i B^{L(a)-i} \cdot \sum_{i=1}^{L(b)} b_i B^{L(b)-i}$.

Wegen $a \cdot b = \operatorname{sgn}(a)\operatorname{sgn}(b) \cdot |a| \cdot |b|$ reduziert[9] sich die Multiplikation $a \cdot b$ im Wesentlichen auf das Produkt der Beträge $|a| \cdot |b|$, das Vorzeichen ergibt sich einfach als Produkt der Vorzeichen der Operanden. Auch die Multiplikation mit 0 ist trivial, wir konzentrieren uns für einen Multiplikationsalgorithmus daher auf den Fall strikt positiver Operanden, siehe dazu auch Übung II.3.

Im Multiplikationsschema von Seite 51 werden alle Teilprodukte $ab_i B^{L(b)-i}$ angeschrieben, bevor sie aufaddiert werden, was die Speicherung von $O(L(a) \cdot L(b))$ Ziffern[10] erfordert. Im Multiplikationsalgorithmus *MultZ* kann diese Speicherung vermieden werden, wenn man jedes Teilprodukt sofort zum vorläufigen Gesamtergebnis hinzuaddiert. Dabei wird ausgenutzt, dass ein Teilprodukt aufgrund der Verschiebung durch die Multiplikation mit $B^{L(b)-i}$ die letzten $L(b) - i$ Ziffern des Gesamtergebnisses unberührt lässt. Da das Ergebnistupel z mit einem Tupel von Nullen der maximal benötigten Länge $L(a) + L(b)$ initialisiert wird, muss z am Ende in die kanonische Form in $\mathcal{Z}_B$ gebracht werden.

Beispiel

Wir veranschaulichen den Algorithmus *MultZ* mit $B = 10$ und dem Beispiel von Seite 51 mit $a = (9,7,3,1)$ und $b = (8,2,9)$. $MultZ(a,b)$ startet mit $z = (0,0,0,0,0,0,0)$ und berechnet für $i = L(b) = 3$ das Zwischenresultat $z = (0,0,8,7,5,7,9)$. Da die nachfolgenden Teilprodukte die letzte Ziffer nicht mehr verändern, ist die Ziffer 9 an der letzten Position schon endgültig. Nach dem Schritt für $i = 2$ ist $z = (0,2,8,2,1,9,9)$ und die letzten beiden Ziffern werden sich im weiteren Verlauf nicht mehr ändern. Nach dem letzten Durchlauf ist $z = (8,0,6,6,9,9,9)$, es sind keine führenden Nullen zu entfernen, z ist somit das Endergebnis.

[9] Dabei ist $\operatorname{sgn}(x) := \begin{cases} -1 & \text{falls } x < 0 \\ 0 & \text{falls } x = 0 \\ 1 & \text{falls } x > 0 \end{cases}$ die *Signum-Funktion*, die sich gut zur Beschreibung des Vorzeichens einer Zahl x eignet.

[10] Durch den Übertrag, der in jedem Teilprodukt entstehen kann, können maximal $(L(a) + 1) \cdot L(b)$ Ziffern anfallen, vgl. mit dem Schema von Seite 51. Die aufgefüllten Nullen müssen nicht abgespeichert werden.

Algorithmus *MultZ*: Klassischer Multiplikationsalgorithmus in $\mathbb{Z}$

```
z ← (0 | k = 1, ..., L(a) + L(b))
for i from L(b) to 1 by −1
    c ← 0
    for j from L(a) to 1 by −1
        p ← a_j · b_i
        q ← p + z_{i+j} + c
        (c, z_{i+j}) ← (q div B, q mod B)
    z_i ← c
z ← kanonisch_{Z_B}(z)
return z
```

Aufruf: $MultZ(a,b)$
Eingabe: $a,b \in \mathcal{Z}_B$
mit: $a_1, b_1 > 0$.
Ausgabe: $z \in \mathcal{Z}_B$
mit: $\sum_{k=1}^{L(z)} z_k B^{L(z)-k} = \sum_{j=1}^{L(a)} a_j B^{L(a)-j} \cdot \sum_{i=1}^{L(b)} b_i B^{L(b)-i}$.

Um in *MultZ* Überlauf zu vermeiden, muss man das Resultat p der Ziffernmultiplikation als Zahl der Länge 2ω darstellen und die anschließenden Additionen auf Zahlen der Länge 2ω durchführen. Die Division mit Rest durch $B = 2^{\omega-1}$ liefert dann den Übertrag für die nachfolgende Stelle und die gesuchte Ziffer des Resultats. Dieser Übertrag ist im Gegensatz zur Addition nicht auf 0 und 1 beschränkt. Eine Rückführung der Division mit Rest auf maximal eine Subtraktion wie in (5.3) ist nicht möglich, weshalb hier die auf Zahlen der Länge 2ω verfügbaren Grundoperationen zur Berechnung von Quotient und Rest verwendet werden müssen.

Der Algorithmus $MultZ(a,b)$ benötigt für die Multiplikation in jedem Fall $L(a) \cdot L(b)$ Ziffernmultiplikationen und $O(L(a) \cdot L(b))$ Ziffernadditionen, insgesamt $O(L(a) \cdot L(b))$ elementare Ziffernoperationen.

Bei Addition und Multiplikation in $\mathcal{Z}_B$ können beliebig große Zahlen entstehen, und diese können *ohne eine konzeptionelle Einschränkung* betreffend ihrer Größe am Rechner auch dargestellt werden. Natürlich ist man auch in diesem Modell durch den auf einem Computer gerade verfügbaren Speicherplatz beschränkt, diese Grenze hat aber in der Praxis keine Relevanz.

Computerprogrammierung ($\mathcal{Z}_B$ in Programmiersprachen)**.** *In Computeralgebrasystemen wie Mathematica werden ganze Zahlen standardmäßig mit beliebiger Länge repräsentiert. In* MATLAB *und anderen Programmiersprachen wie C/C++ oder Java hingegen gehören Zahlen beliebiger Länge nicht zum Standardumfang, sind aber meist in Zusatzpaketen erhältlich. In* MATLAB *stehen sie über die Symbolic Toolbox zur Verfügung.*

Ein schnellerer Multiplikationsalgorithmus

Wir wollen abschließend ein Verfahren zur Multiplikation von a und b in $\mathcal{Z}_B$ vorstellen, das mit weniger als den oben erwähnten $O(L(a) \cdot L(b))$ Elementaroperationen auskommt. Dazu definieren wir zuerst Hilfsoperationen auf Zifferntupel, die sich leicht mit den elementaren Tupeloperationen *EinfügenBeginn* und *EinfügenEnde* von Seite 47 realisieren lassen.

Verschiebe$_{\mathcal{Z}_B}(a,n)$ liefert $b \in \mathcal{Z}_B$, indem die Ziffern in $a \in \mathcal{Z}_B$ um $n \in \mathbb{N}_0$ Stellen nach links verschoben und am Ende mit Nullen aufgefüllt wird, d.h.

$$L(b) = L(a) + n \quad \text{und} \quad b_i = \begin{cases} a_i & \text{für } i \le L(a) \\ 0 & \text{sonst.} \end{cases}$$

Fülle$_{\mathcal{Z}_B}(a,n)$ liefert für $a \in \mathcal{Z}_B$ und $n \in \mathbb{N}_0$ ein Tupel b, indem a am Beginn mit Nullen auf Gesamtlänge $\max(n, L(a))$ aufgefüllt wird, d.h. mit $d = \max\{0, n - L(a)\}$

$$L(b) = L(a) + d \quad \text{und} \quad b_i = \begin{cases} 0 & \text{für } i \le d \\ a_{i-d} & \text{sonst.} \end{cases}$$

Beispiel

Verschiebe$_{\mathcal{Z}_B}((9,7),2)$ ergibt das Tupel $(9,7,0,0)$, als Tupel von Ziffern zur Basis $B = 10$ interpretiert kommt dies einer Multiplikation $97 \cdot B^2 = 9700$ gleich, siehe Übung II.4. *Fülle*$_{\mathcal{Z}_B}((8,2,9),4)$ resultiert in $(0,8,2,9)$. Wir wollen auch zu Tupel mit führenden Nullen eine ganze Zahl analog zu (5.2) assoziieren[11], im Falle von $(0,8,2,9)$ und $B = 10$ ergibt sich daraus mit $0 \cdot 10^3 + 8 \cdot 10^2 + 2 \cdot 10 + 9$ wieder 829, führende Nullen lassen die dargestellte Zahl unverändert.

Zur Entwicklung des Multiplikationsverfahrens betrachten wir wieder strikt positive Eingaben[12] a und b. Zusätzlich verlangen wir gleiche und darüberhinaus gerade Stellenanzahl $2n$, was man gegebenenfalls durch Auffüllen mit Nullen am Beginn der jeweiligen Zifferntupel erreichen kann. Die durch a beschriebene Zahl α kann dann gemäß

$$\alpha = \sum_{i=1}^{2n} a_i B^{2n-i} = B^n \cdot \sum_{i=1}^{n} a_i B^{n-i} + \sum_{i=n+1}^{2n} a_i B^{2n-i} = B^n \bar{\alpha} + \tilde{\alpha}$$

geschrieben werden, wobei die Zahlen $\bar{\alpha}$ und $\tilde{\alpha}$ durch die Tupel

$$\bar{a} := a_{1:n} \quad \text{bzw.} \quad \tilde{a} := a_{n+1:2n}$$

dargestellt sind. Analog dazu kann die durch b beschriebene Zahl β durch die Tupel $\bar{b} := b_{1:n}$ bzw. $\tilde{b} := b_{n+1:2n}$ ausgedrückt werden.

Beispiel

Für $a = (9,7,3,1)$ und $b = (8,2,9)$ ist $n = 2$. Mit $B = 10$ beschreibt a die Zahl

$$\alpha = 9731 = 10^2 \cdot 97 + 31 = 10^2 \cdot \bar{\alpha} + \tilde{\alpha}.$$

Die Zifferndarstellung für $\bar{\alpha} = 97$ und $\tilde{\alpha} = 31$ ist durch $\bar{a} = a_{1:2} = (9,7)$ und $\tilde{a} = a_{3:4} = (3,1)$ gegeben.

[11] Im Fall einer negativen Zahl ist die erste von Null verschiedene Ziffer wie z_1 in (5.2) gesondert zu behandeln.

[12] Das Vorzeichen kann man wie auf Seite 59 gesondert behandeln.

Die Berechnung des Produkts $\alpha \cdot \beta$ kann wegen

$$\alpha \cdot \beta = (B^n \bar{\alpha} + \tilde{\alpha}) \cdot (B^n \bar{\beta} + \tilde{\beta}) = B^{2n} \bar{\alpha}\bar{\beta} + B^n((\bar{\alpha} + \tilde{\alpha})(\bar{\beta} + \tilde{\beta}) - \bar{\alpha}\bar{\beta} - \tilde{\alpha}\tilde{\beta}) + \tilde{\alpha}\tilde{\beta},$$

auf die Berechnung *dreier Produkte* kürzerer Zahlen zurückgeführt werden, nämlich $\bar{\alpha} \cdot \bar{\beta}, \tilde{\alpha} \cdot \tilde{\beta}$ und $(\bar{\alpha} + \tilde{\alpha})(\bar{\beta} + \tilde{\beta})$. Diese Rekursion führt auf den *Karatsuba*[13] *Algorithmus MultZKaratsuba*.

Algorithmus *MultZKaratsuba*: Karatsuba Multiplikation in $\mathbb{Z}$

```
n ← (max(L(a),L(b)) + 1) div 2
if n = 1
   z ← a · b
else
   a ← Fülle_{Z_B}(a,2n), b ← Fülle_{Z_B}(b,2n)
   ā ← kanonisch_{Z_B}(a_{1:n})
   ã ← kanonisch_{Z_B}(a_{n+1:2n})
   b̄ ← kanonisch_{Z_B}(b_{1:n})
   b̃ ← kanonisch_{Z_B}(b_{n+1:2n})
   p ← MultZKaratsuba(ā,b̄)
   q ← MultZKaratsuba(ã,b̃)
   r ← MultZKaratsuba(ā + ã, b̄ + b̃)
   s ← Verschiebe_{Z_B}(r − p − q,n)
   t ← Verschiebe_{Z_B}(p,2n)
   z ← t + s + q
return z
```

Aufruf: $MultZKaratsuba(a,b)$
Eingabe: $a,b \in \mathcal{Z}_B$
mit: $a_1, b_1 > 0$
Ausgabe: $z \in \mathcal{Z}_B$
mit: $\sum_{i=1}^{L(z)} z_i B^{L(z)-i} = \sum_{i=1}^{L(a)} a_i B^{L(a)-i} \cdot \sum_{i=0}^{L(b)} b_i B^{L(b)-i}$.

Die Berechnung von n am Beginn stellt sicher, dass $2n$ die kleinste gerade Zahl größer gleich $\max(L(a),L(b))$ ist. Im Basisfall von Operanden der Länge 1 kann man in diesem Verfahren auf den klassischen Algorithmus *MultZ* zurückgreifen. Zum Auffüllen der Tupel am Beginn und zur Multiplikation mit B^n bzw. B^{2n} kommen die auf Seite 61 vorgestellten Algorithmen $Fülle_{\mathcal{Z}_B}$ und $Verschiebe_{\mathcal{Z}_B}$ zum Einsatz. Durch das Auffüllen der Operanden auf Länge $2n$ und Aufspalten der Tupel können führende Nullen entstehen, etwa wenn $a = (2,0,3)$ zuerst auf Länge 4 gebracht wird und dann in zwei Hälften $(0,2)$ und $(0,3)$ geteilt wird. Daher müssen die nach dem Aufspalten erhaltenen Tupel in kanonische Form in $\mathcal{Z}_B$ gebracht werden, bevor mit ihnen weitergerechnet wird. Den im Karatsuba Algorithmus ebenfalls benötigten Subtraktionsalgorithmus auf $\mathcal{Z}_B$ haben wir nicht vorgestellt, er funktioniert aber im Wesentlichen ähnlich zu *AddZ*. Auch die Division mit Rest lässt sich von Zahlen fixer Länge auf $\mathcal{Z}_B$ übertragen. Eine solche findet sich am Beginn von *MultZKaratsuba* zur Bestimmung von n für die Aufspaltung der Eingabetupel in zwei Hälften. Diese kann aber auch ohne explizite Berechnung der Längen durchgeführt werden kann, siehe Übung II.5.

[13] Karatsuba, Anatolii Alexeevich: 1937–heute, russischer Mathematiker. Ein schneller Multiplikationsalgorithmus wurde von ihm entwickelt, ansonsten arbeitet der Vorstand des Steklov Instituts für Mathematik in Moskau in der Zahlentheorie an trigonometrischen Summen bzw. Integralen oder auch an der Theorie der endlichen Automaten. Er ist begeisterter Bergsteiger und hat schon die höchsten Gipfel im Kaukasus bezwungen.

Beispiel

Wir demonstrieren den Algorithmus wieder mit $B = 10$ und dem Beispiel von Seite 51, also $a = (9,7,3,1)$ und $b = (8,2,9)$. Hier ist $n = 2$, die Eingaben sind jeweils auf Länge 4 aufzufüllen und dann in zwei Hälften der Länge jeweils 2 aufzuspalten. Somit ist

$$\bar{a} = a_{1:2} = (9,7) \quad \tilde{a} = a_{3:4} = (3,1) \quad \bar{b} = b_{1:2} = (0,8) \quad \tilde{b} = b_{3:4} = (2,9)$$

In den rekursiven Aufrufen werden dann

$$p = 97 \cdot 8 = (7,7,6) \quad q = 31 \cdot 29 = (8,9,9) \quad r = (97 + 31)(8 + 29) = (4,7,3,6)\,,$$

daraus

$$s = (r - p - q)10^2 = (3,0,6,1,0,0) \qquad t = p \cdot 10^4 = (7,7,6,0,0,0,0)$$

und schließlich das Endergebnis

$$z = t + s + q = (8,0,6,6,9,9,9)$$

berechnet.

Wir zeigen nun den Grundgedanken der Aufwandsanalyse für den Karatsuba Algorithmus, für Details verweisen wir auf [11] oder [7]. Es bezeichne $M(n)$ den Aufwand zur Multiplikation zweier Zahlen mit maximaler Länge n. Seien nun α und β wie oben zwei Zahlen der Länge $2n$, dann benötigt *MultZKaratsuba* die drei Multiplikationen $\bar{\alpha} \cdot \bar{\beta}, \tilde{\alpha} \cdot \tilde{\beta}$ und $(\bar{\alpha} + \tilde{\alpha})(\bar{\beta} + \tilde{\beta})$. Die ersten beiden sind dabei Produkte von n-stelligen Zahlen, bedeuten daher einen Aufwand $M(n)$. Auch die dritte Multiplikation kann mit Aufwand $M(n) + O(n)$ bewältigt werden. Darüber hinaus benötigt er vier Additionen maximal $2n$-stelliger Zahlen und Verschiebungen der Ziffern um $2n$ bzw. n Stellen, wobei der Aufwand für all diese Operationen jeweils $O(n)$ ist. Für den Gesamtaufwand $M(2n)$ gilt demnach

$$M(2n) \leq 3M(n) + C \cdot n \tag{5.4}$$

mit einer geeigneten Konstante C. Aus (5.4) kann dann die Zeitkomplexität des Karatsuba Algorithmus von

$$O(n^{\log_2(3)}) \approx O(n^{1.585})$$

ermittelt werden. Der klassische Algorithmus benötigt im Vergleich $O(n^2)$ Elementaroperationen, jedoch macht sich der zusätzliche Aufwand von *MultZKaratsuba* für Additionen, Verschiebungen und Rekursion erst für sehr lange Zahlen bezahlt. Doch auch der Karatsuba Algorithmus ist noch nicht der schnellste Weg zur Multiplikation, es gibt andere Methoden basierend etwa auf der schnellen Fouriertransformation, deren asymptotische Komplexität noch geringer ist, siehe etwa [5] oder [10].

Vereinbarung. Wir haben nun mit *MultZ* und *MultZKaratsuba* zwei Algorithmen zum Multiplizieren in $\mathcal{Z}_B$ kennengelernt. In zukünftigen Algorithmen werden wir einen Unteralgorithmus zum Multiplizieren beliebig langer Zahlen a und

b einfach als $a \cdot b$ schreiben, ohne festzulegen, ob wir damit $MultZ(a,b)$ oder $MultZKaratsuba(a,b)$ meinen. Ein solcher Algorithmus ist dann im Sinne des Algorithmenbegriffs von Seite 5 *unvollständig spezifiziert*, da er ja einen Teilschritt nicht mehr *präzise* festlegt, wichtig ist uns aber in solchen Fällen nur, dass *mindestens ein* Algorithmus existiert, der $a \cdot b$ realisiert.

Der größte gemeinsame Teiler

Das Bestimmen des größten gemeinsamen Teilers (ggT) zweier Zahlen ist eine der zentralen Problemstellungen in $\mathbb{N}$ bzw. $\mathbb{Z}$. Auch in anderen Gebieten – z.B. beim Rechnen mit Brüchen auf Seite 78 – werden wir dieser Fragestellung wieder begegnen. Wir rufen zuerst einige grundlegende Begriffe in Erinnerung, bevor wir uns einem Verfahren zur ggT-Berechnung zuwenden.

Definition

Teilbarkeit in $\mathbb{Z}$. Seien $a,b,t \in \mathbb{Z}$. Die Zahl t *teilt* a genau dann, wenn ein $q \in \mathbb{Z}$ existiert, so dass $a = tq$. Wir schreiben dafür $t|a$ und nennen t einen *Teiler* von a. Gilt $t|a$ und $t|b$, so ist t ein *gemeinsamer Teiler* von a und b. Wir vereinbaren $T(a) := \{t \in \mathbb{Z} \mid t|a\}$ und $gT(a,b) := \{t \in \mathbb{Z} \mid t|a \wedge t|b\}$.

Beispiel

Es ist $T(25) = \{\pm 1, \pm 5, \pm 25\}$ und $T(10) = \{\pm 1, \pm 2, \pm 5, \pm 10\}$ und somit $gT(25,10) = T(25) \cap T(10) = \{\pm 1, \pm 5\}$. Laut Definition teilt jede Zahl 0, somit ist $T(0)$ unendlich.

Definition

ggT in $\mathbb{Z}$. Seien $a,b \in \mathbb{Z}$ mit $a \neq 0$ oder $b \neq 0$. Dann heißt

$$ggT(a,b) := \max gT(a,b)$$

der *größte gemeinsame Teiler* von a und b. Weiters legt man $ggT(0,0) := 0$ fest. Im Fall $ggT(a,b) = 1$ nennt man a und b *relativ prim*.

Problemstellung (ggT in $\mathbb{Z}$).

Gegeben:	$a,b \in \mathbb{Z}$.
Gesucht:	$t \in \mathbb{N}_0$
mit:	$t = ggT(a,b)$.

Im Fall $a \neq 0$ oder $b \neq 0$ ist die Existenz einer Lösung dieses Problems gesichert, da die Menge $gT(a,b)$ der gemeinsamen Teiler von a und b endlich ist und daher ein größtes Element $t \in \mathbb{N}_0$ bzgl. der natürlichen Ordnung $\leq$ hat. Die Eindeutigkeit des ggT folgt aus der Eindeutigkeit des Maximums in der Definition. Für $a = b = 0$ existiert per Definition eine eindeutige Lösung, nämlich $t = 0$. Der folgende Satz bildet die Grundlage für einen Algorithmus zur Berechnung des ggT.

Satz

Seien $a,b \in \mathbb{Z}$.

1. Wenn $t \in \mathrm{gT}(a,b)$, dann $t \in \mathrm{T}(a \pm b)$.
2. Wenn $t \in \mathrm{T}(b)$, dann ist für jedes $q \in \mathbb{Z}$ auch $t \in \mathrm{T}(b \cdot q)$.
3. $\mathrm{ggT}(a,b) = \mathrm{ggT}(|a|,|b|)$.
4. Ist $a \in \mathbb{N}_0$, so ist $\mathrm{ggT}(a,0) = a$.

Beweis.

1. Sei $t \in \mathrm{gT}(a,b)$, d.h. $a = tq_a$ und $b = tq_b$ für geeignete $q_a,q_b \in \mathbb{Z}$. Somit ist $a \pm b = t(q_a \pm q_b)$ und damit $t \in \mathrm{T}(a \pm b)$.
2. Sei $t \in \mathrm{T}(b)$, d.h. $b = tq_b$ für ein $q_b \in \mathbb{Z}$. Wegen $b \cdot q = t(q_b \cdot q)$ ist $t \in \mathrm{T}(b \cdot q)$.
3. Wenn $t \in \mathrm{T}(-a)$ dann auch $t \in \mathrm{T}(a)$, und daher $\mathrm{T}(-a) = \mathrm{T}(a) = \mathrm{T}(|a|)$.
4. Ist $a = 0$, so ist die Aussage per Definition wahr. Für $a \in \mathbb{N}$ ist jeder Teiler von a kleiner gleich a. Da $a \in \mathrm{gT}(a,0)$, muss $a = \mathrm{ggT}(a,0)$ gelten. □

Wegen Teil 3 und 4 des Satzes reicht es aus, ein Verfahren zur Berechnung des ggT für $a,b \in \mathbb{N}_0$ zu entwickeln. Seien nun q und r Quotient und Rest bei Division von a durch b. Ein gemeinsamer Teiler von a und b teilt dann auch $r = a - b \cdot q = a \bmod b$. Umgekehrt teilt ein gemeinsamer Teiler von b und $r = a - b \cdot q$ auch $r + b \cdot q = a$. Damit ist für $b \neq 0$ jedenfalls $\mathrm{gT}(a,b) = \mathrm{gT}(b,a \bmod b)$ und daher auch

$$\mathrm{ggT}(a,b) = \mathrm{ggT}(b,a \bmod b). \tag{5.5}$$

Beispiel

$$\mathrm{ggT}(25,10) = \mathrm{ggT}(10,\underbrace{25 \bmod 10}_{=5}) = \mathrm{ggT}(5,\underbrace{10 \bmod 5}_{=0}) = 5.$$

Mit Hilfe von (5.5) kann $\mathrm{ggT}(a,b)$ rekursiv berechnet werden. Die der Reihe nach auftretenden Reste bilden eine streng monoton fallende Folge in $\mathbb{N}_0$, daher *muss* die Rekursion *nach endlich vielen Schritten* terminieren. Der einzig mögliche Abbruchgrund ist $b = 0$. Der letzte in dieser Reihe auftretende echt positive Rest $r^{(k)}$ ist dann wegen Teil 4 des obigen Satzes der gesuchte ggT. Das auf diese Weise entstehende Verfahren *GGTZEuklid* heißt *Euklidscher Algorithmus*. Die im Laufe des Algorithmus erzeugten

$$r^{(0)} := |a| \quad r^{(1)} := |b| \quad \ldots \quad r^{(i)} := r^{(i-2)} \bmod r^{(i-1)} \quad \ldots \quad r^{(k)} \quad 0 \tag{5.6}$$

nennt man die *Euklidsche Restfolge* (zu a und b).

Computerprogrammierung. *Die bei der* ggT*-Berechnung auftretende spezielle Form der Rekursion, in der das Resultat des rekursiven Aufrufs nicht mehr weiterverwendet wird, heißt* Endrekursion *(engl.* tail recursion*). Diese sind besonders leicht als Schleifen realisierbar, weil im Zuge der rekursiven Aufrufe kein „Stack" aufgebaut werden muss, siehe dazu die Erläuterungen zum rekursiven Funktionsaufruf auf Seite 24.*

Algorithmus *GGTZEuklid*: Euklidscher Algorithmus in $\mathbb{Z}$

```
a ← |a|, b ← |b|
if b = 0
   t ← a
else
   r ← a mod b
   t ← GGTZEuklid(b,r)
return t
```

Aufruf: $GGTZEuklid(a,b)$
Eingabe: $a,b \in \mathbb{Z}$.
Ausgabe: $t \in \mathbb{N}_0$
mit: $t = \mathrm{ggT}(a,b)$.

Eine genaue Komplexitätsuntersuchung des Euklidschen Algorithmus gestaltet sich sehr aufwändig und auch mathematisch schwierig. Wir begnügen uns daher ohne Beweis mit einem Komplexitätsresultat, wonach in *GGTZEuklid* höchstens $\lceil \log_\beta(\sqrt{5}\max(a,b))\rceil - 2$ mit $\beta = \frac{1}{2}(1+\sqrt{5})$ Divisionen mit Rest benötigt werden, wobei $\log_\beta(\sqrt{5}\max(a,b)) \approx 4.785\log_{10}(\max(a,b)) + 0.6723$ ist, siehe [7]. Da $\log_{10}(a)$ im wesentlichen die Länge von a im Dezimalsystem angibt, ist der Rechenaufwand für den Euklidschen Algorithmus $O(L_{10}(\max(a,b)))$.

Beispiel

Rufen wir $GGTZEuklid(a,b)$ mit $a = 55$ und $b = 35$ auf, so lautet die Euklidsche Restfolge

$$r^{(0)} = 55 \quad r^{(1)} = 35 \quad r^{(2)} = 20 \quad r^{(3)} = 15 \quad r^{(4)} = 5 \quad r^{(5)} = 0.$$

An dieser Stelle ist der Basisfall der Rekursion erreicht, daher ist $\mathrm{ggT}(55,35) = 5$. Mit $a = 35$ und $b = 55$ erhält man

$$r^{(0)} = 35 \quad r^{(1)} = 55 \quad r^{(2)} = 35 \quad r^{(3)} = 20 \quad r^{(4)} = 15 \quad r^{(5)} = 5 \quad r^{(6)} = 0.$$

Im Fall $a < b$ ist also ein zusätzlicher Divisionsschritt am Beginn nötig, der einem Vertauschen von a und b gleichkommt.

In einer Erweiterung des Euklidschen Algorithmus kann neben dem ggT auch noch eine Darstellung des ggT als Linearkombination der Ausgangszahlen berechnet werden.

Problemstellung (Größter gemeinsamer Teiler als Linearkombination).

Gegeben: $a,b \in \mathbb{Z}$.
Gesucht: $t \in \mathbb{N}_0, x,y \in \mathbb{Z}$
mit: $t = \mathrm{ggT}(a,b)$ und $ax + by = t$.

Falls eine Lösung t,x,y dieser Problemstellung existiert, so ist diese nicht eindeutig, da dann z.B. auch durch $t, x+b$ und $y-a$ eine Lösung gegeben ist. Die Existenz einer Lösung garantiert der folgende Satz.

Satz

Für alle $a,b \in \mathbb{Z}$ existieren $t \in \mathbb{N}_0$ und $x,y \in \mathbb{Z}$ mit $t = \mathrm{ggT}(a,b)$ und $ax+by = t$.

Beweis. Die Existenz eines t mit $t = \mathrm{ggT}(a,b)$ ist trivial. Zum Nachweis der Existenz von x und y mit den geforderten Eigenschaften betrachten wir die Euklidsche Restfolge aus (5.6) und zeigen, dass jedes $r^{(i)}$ und damit auch $r^{(k)} = \mathrm{ggT}(a,b) = t$ als Linearkombination von a und b darstellbar ist. Die Darstellungen für $r^{(0)}$ und $r^{(1)}$ sind wegen $|a| = a \cdot \mathrm{sgn}(a) + b \cdot 0$ und $|b| = a \cdot 0 + b \cdot \mathrm{sgn}(b)$ einfach. Kann sowohl $r^{(i-2)}$ als auch $r^{(i-1)}$ als Linearkombination von a und b dargestellt werden, also

$$r^{(i-2)} = ax^{(i-2)} + by^{(i-2)} \qquad r^{(i-1)} = ax^{(i-1)} + by^{(i-1)},$$

dann gilt wegen $r^{(i)} = r^{(i-2)} \bmod r^{(i-1)}$, d.h. $r^{(i-2)} = r^{(i-1)}q^{(i)} + r^{(i)}$, auch

$$\begin{aligned} r^{(i)} = r^{(i-2)} - r^{(i-1)}q^{(i)} &= ax^{(i-2)} + by^{(i-2)} - (ax^{(i-1)} + by^{(i-1)})q^{(i)} \\ &= a\underbrace{(x^{(i-2)} - x^{(i-1)}q^{(i)})}_{=:x^{(i)}} + b\underbrace{(y^{(i-2)} - y^{(i-1)}q^{(i)})}_{=:y^{(i)}}. \end{aligned} \tag{5.7}$$

Damit ist $r^{(i)}$ als Linearkombination $ax^{(i)} + by^{(i)}$ von a und b dargestellt. □

Wir können also im Euklidschen Algorithmus zusätzlich Koeffizienten $x^{(i)}$ und $y^{(i)}$ wie in (5.7) beschrieben mitrechnen, mit denen beim Abbruch des Algorithmus eine Darstellung von $r^{(k)} = \mathrm{ggT}(a,b)$ als Linearkombination $ax^{(k)} + by^{(k)}$ gegeben ist. Diese Variante des Verfahrens wird *erweiterter Euklidscher Algorithmus* genannt. Wir realisieren das Verfahren mittels einer Schleife, da die rekursive Variante hier komplizierter zu formulieren ist, siehe Algorithmus *ErwGGTZEuklid*.

Algorithmus *ErwGGTZEuklid*: Erweiterter Euklidscher Algorithmus in $\mathbb{Z}$

```
r'' ← |a|, x'' ← sgn(a), y'' ← 0
r' ← |b|, x' ← 0, y' ← sgn(b)
while r' ≠ 0
   q ← r'' div r',  r ← r'' mod r'
   x ← x'' − x' · q,  y ← y'' − y' · q
   r'' ← r',  r' ← r
   x'' ← x',  x' ← x,  y'' ← y',  y' ← y
t ← r'',  x ← x'',  y ← y''
return (t,x,y)
```

Aufruf: *ErwGGTZEuklid*(a,b)
Eingabe: $a,b \in \mathbb{Z}$.
Ausgabe: $t \in \mathbb{N}_0, x,y \in \mathbb{Z}$
mit: $t = \mathrm{ggT}(a,b)$ und $ax + by = t$.

Einfache diophantische Gleichungen

Gleichungen mit ganzzahligen Koeffizienten, in denen ganzzahlige Lösungen gesucht sind, heißen allgemein *diophantische*[14] *Gleichungen*, etwa

$$x^2 + y^2 = z^2 \tag{5.8}$$

[14] Diophant von Alexandria: ca. 200–280 n. Chr., griechischer Mathematiker. Gilt als einer der Väter der Algebra, da er in seiner „Arithmetica“ zu 130 Problemstellung mit algebraischen Gleichungen zahlenmäßige Lösungen zusammengestellt hat.

für gesuchte $x,y,z \in \mathbb{Z}$. Geometrisch interpretiert sind hier ganze Zahlen gesucht, die als Seitenlänge eines rechtwinkeligen Dreiecks auftreten können, wobei man naturgemäß an positiven Lösungen interessiert ist. Die Gleichung (5.8) besitzt unendlich viele Lösungen, genannt *pythagoräische Tripel*. Wir wollen uns nun *linearen diophantischen Gleichungen* zuwenden, also solchen, die keine Potenzen von Unbekannten enthalten, und hier wiederum den Fall von Gleichungen mit *zwei Unbekannten* studieren.

Problemstellung (Lineare diophantische Gleichung in zwei Unbekannten).

Gegeben: $a,b,c \in \mathbb{Z}$.
Gesucht: $x,y \in \mathbb{Z}$
mit: $ax + by = c$.

Eindeutigkeit der Lösung ist wie in der Problemstellung des ggT als Linearkombination der Eingangswerte nicht gegeben. Für eine Untersuchung der Existenz von Lösungen einer diophantischen Gleichung bemerken wir zuerst, dass natürlich immer $\text{ggT}(a,b)|(ax+by)$ gilt, siehe den Satz auf Seite 65. Im Falle der Lösbarkeit der diophantischen Gleichung muss damit auch $\text{ggT}(a,b)|c$ gelten. Sei nun umgekehrt $\text{ggT}(a,b)|c$ erfüllt, also $c = q \cdot \text{ggT}(a,b)$ mit $q \in \mathbb{Z}$. Da nach dem Satz von Seite 66 der ggT von a und b als Linearkombination $ax' + by' = \text{ggT}(a,b)$ dargestellt werden kann, ist mit

$$x = q \cdot x' \quad \text{und} \quad y = q \cdot y' \tag{5.9}$$

eine Lösung der ursprünglichen diophantischen Gleichung gefunden. Damit haben wir ein Kriterium für die Lösbarkeit einer linearen diophantischen Gleichung bewiesen.

Satz

Lösbarkeit einer linearen diophantischen Gleichung. Seien $a,b,c \in \mathbb{Z}$. Dann gilt:

$$\text{Die Gleichung } ax + by = c \text{ ist lösbar für } x,y \in \mathbb{Z} \iff \text{ggT}(a,b)|c.$$

Der Algorithmus *LöseLinDiophant* zum Lösen diophantischer Gleichungen basiert auf dem konstruktiven Argument (5.9) aus obigem Beweis und dem erweiterten Euklidschen Algorithmus.

Algorithmus *LöseLinDiophant*: Lösen einer linearen diophantischen Gleichung in zwei Unbekannten

```
if a = b = c = 0
   x ← 0, y ← 0
else
   (t,x',y') ← ErwGGTZEuklid(a,b)
   q ← c div t
   x ← qx', y ← qy'
return (x,y)
```

Aufruf: *LöseLinDiophant*(a,b,c)
Eingabe: $a,b,c \in \mathbb{Z}$
mit: $\text{ggT}(a,b)|c$.
Ausgabe: $x,y \in \mathbb{Z}$
mit: $ax + by = c$.

Beispiel

Für eine Lösung der diophantischen Gleichung $55x + 35y = -10$ berechnet *LöseLinDiophant*(55,35, − 10) mit dem erweiterten Euklidschen Algorithmus zuerst ggT(55,35) = 5 und die Darstellung des ggT als

$$55 \cdot 2 + 35 \cdot (-3) = 5. \tag{5.10}$$

Eine Lösung, die wegen $5 | -10$ existiert, erhalten wir mit $q = -2$ aus (5.10) als $x = -4$ und $y = 6$.

6 Kongruenzklassen modulo m

Mathematische Grundlagen

Aufbauend auf der bekannten Teilbarkeitsrelation auf $\mathbb{Z}$ können wir eine weitere Beziehung zwischen zwei ganzen Zahlen untersuchen.

Definition

Kongruenz modulo m. Sei $m \in \mathbb{N}$. Die Zahlen $a, b \in \mathbb{Z}$ heißen *kongruent modulo m* genau dann, wenn $m | (a - b)$ gilt. Für die Kongruenz von a und b modulo m schreiben wir $a \equiv_m b$.

Es lässt sich zeigen, siehe Übung II.8, dass $\equiv_m$ eine *Kongruenzrelation* auf $\mathbb{Z}$ ist, d.h. eine Äquivalenzrelation auf $\mathbb{Z}$, die mit Addition und Multiplikation auf $\mathbb{Z}$ *verträglich* ist im Sinne von

$$a \equiv_m b \text{ und } a' \equiv_m b' \Rightarrow (a + a' \equiv_m b + b' \quad \text{und} \quad a \cdot a' \equiv_m b \cdot b'). \tag{6.11}$$

Definition

Kongruenzklassen, Rechnen modulo m. Für $a \in \mathbb{Z}$ nennt man die Äquivalenzklasse von a bzgl. $\equiv_m$ auch *Kongruenzklasse* (oder *Restklasse*) von a *modulo m* und bezeichnet sie mit $[a]_m$. Weiters ist

$$\mathbb{Z}_m := \{[a]_m \mid a \in \mathbb{Z}\},$$

und in $\mathbb{Z}_m$ heißt m der *Modul*. Auf $\mathbb{Z}_m$ können für $\diamond \in \{+, -, \cdot\}$ durch

$$[a]_m \diamond [b]_m := [a \diamond b]_m \tag{6.12}$$

selbst wieder Addition, Subtraktion und Multiplikation definiert werden. Diese sind wegen (6.11) *wohldefiniert*, d.h. die Resultate hängen nicht von den konkret gewählten Repräsentanten der Klassen ab.

Seien q_a bzw. r_a Quotient und Rest bei Division von a durch m. Für jedes $b \in [a]_m$ existiert per Definition ein q_b mit $a - b = q_b \cdot m$. Wegen

$$b = a - q_b \cdot m = q_a \cdot m + r_a - q_b \cdot m = (q_a - q_b) \cdot m + r_a$$

ist jedenfalls $b \bmod m = r_a = a \bmod m$. Damit haben alle Elemente in $[a]_m$ bei Division durch m denselben Rest, nämlich r_a, daher auch der Name Restklasse. Da auch $r_a + q \cdot m \in [a]_m$ für jedes $q \in \mathbb{Z}$ gilt, ist $[a]_m = \{r_a + q \cdot m \mid q \in \mathbb{Z}\}$. Daraus sieht man auch leicht, dass es in $\mathbb{Z}_m$ genau m verschiedene Klassen gibt, also $\mathbb{Z}_m = \{[0]_m, \ldots, [m-1]_m\}$.

Beispiel

In $\mathbb{Z}_3$ lauten die Kongruenzklassen

$$[0]_3 = \{\ldots, -6, -3, 0, 3, 6, \ldots\}$$
$$[1]_3 = \{\ldots, -5, -2, 1, 4, 7, \ldots\}$$
$$[2]_3 = \{\ldots, -4, -1, 2, 5, 8, \ldots\}.$$

und jede Klasse hat die Gestalt $[a]_3 = \{a + q \cdot 3 \mid q \in \mathbb{Z}\}$. Addition und Multiplikation auf $\mathbb{Z}_3$ verhalten sich laut Definition wie in den folgenden Verknüpfungstafeln gezeigt.

$+$	$[0]_3$	$[1]_3$	$[2]_3$
$[0]_3$	$[0]_3$	$[1]_3$	$[2]_3$
$[1]_3$	$[1]_3$	$[2]_3$	$[0]_3$
$[2]_3$	$[2]_3$	$[0]_3$	$[1]_3$

$\cdot$	$[0]_3$	$[1]_3$	$[2]_3$
$[0]_3$	$[0]_3$	$[0]_3$	$[0]_3$
$[1]_3$	$[0]_3$	$[1]_3$	$[2]_3$
$[2]_3$	$[0]_3$	$[2]_3$	$[1]_3$

Aus den Verknüpfungstafeln von $\mathbb{Z}_3$ sieht man sofort, dass jedes Element außer $[0]_3$ bzgl. der Multiplikation invertierbar ist. In $\mathbb{Z}_4$ hingegen ist $[2]_4$ wegen

$$[2]_4 \cdot [1]_4 = [2]_4 \qquad [2]_4 \cdot [2]_4 = [0]_4 \qquad [2]_4 \cdot [3]_4 = [2]_4$$

nicht invertierbar bzgl. der Multiplikation. Die Erklärung dafür liegt im folgenden Satz.

Satz

Seien $b \in \mathbb{Z}$ und $m \in \mathbb{N}$. Dann ist $[b]_m \in \mathbb{Z}_m$ invertierbar bzgl. der Multiplikation genau dann, wenn $\mathrm{ggT}(m,b) = 1$.

Beweis. Es gilt jedenfalls

$$[b]_m \cdot [x]_m = [1]_m \iff bx \equiv_m 1 \iff bx - km = 1 \text{ für ein } k \in \mathbb{Z}. \tag{6.13}$$

Aufgrund der Lösbarkeitsbedingung für diophantische Gleichungen von Seite 68 existieren $x, k \in \mathbb{Z}$ mit $bx - km = 1$ aber genau dann, wenn $\mathrm{ggT}(m,b) | 1$, also $\mathrm{ggT}(m,b) = 1$. □

Repräsentantensystem. Ein *Repräsentantensystem* einer Äquivalenzrelation $\sim$ auf einer Menge M ist eine Menge $R \subseteq M$, in der aus jeder Äquivalenzklasse genau ein Element enthalten ist. Definition

$\mathbb{Z}_m$ besteht aus genau m Restklassen, somit enthält ein Repräsentantensystem von $\equiv_m$ immer m Elemente. Im Beispiel von $\mathbb{Z}_3$ sind typische Repräsentantensysteme $\{0,1,2\}$ oder wegen $-1 \in [2]_3$ auch $\{-1,0,1\}$.

Für die in (6.12) auf $\mathbb{Z}_m$ definierten Grundoperationen + und $\cdot$ gelten die gewohnten Rechengesetze, wie Assoziativ-, Kommutativ- und Distributivgesetz, $\mathbb{Z}_m$ ist also bzgl. der Addition und Multiplikation ein Ring. Falls m eine Primzahl ist, so sind darüberhinaus alle $[b]_m \in \mathbb{Z}_m \setminus \{[0]_m\}$ invertierbar, da $\text{ggT}(m,b) = 1$ in diesem Fall für jedes solche b garantiert ist. Daher ist $\mathbb{Z}_m$ dann sogar ein Körper.

Kongruenzklassen modulo m am Computer

Jede Kongruenzklasse $[a]_m \in \mathbb{Z}_m$ ist eine unendliche Menge. Zur Darstellung am Computer ersetzen wir $\mathbb{Z}_m$ durch ein Repräsentantensystem R von $\equiv_m$ und rechnen mit Restklassen, indem wir mit Repräsentanten der Klassen „wie in $\mathbb{Z}$“ rechnen. Jedes $a \in \mathbb{Z}$ liegt in genau einer Kongruenzklasse bzgl. $\equiv_m$, nämlich $[a]_m$, und aus jeder solchen Klasse ist genau ein Element in R enthalten. Damit gibt es zu jedem $a \in \mathbb{Z}$ genau ein $a' \in R$ mit $a \equiv_m a'$. Die Operation

$$\text{kanonisch}_{\mathbb{Z}_m} : \mathbb{Z} \to R,\ a \mapsto a'$$

nennt man in der Regel einen *kanonischen Simplifikator* für $\equiv_m$ und a' nennt man die *kanonische Form* von a bzgl. $\equiv_m$.

Computerrepräsentation (Datenstrukturen $\mathcal{Z}_m^+$ und $\mathcal{Z}_m^\pm$ für Kongruenzklassen)**.** Der kanonische Repräsentant hängt natürlich vom gewählten Repräsentantensystem ab, wobei für $\mathbb{Z}_m$ in der Regel zwei Varianten in Verwendung sind.

1. Wählt man das Repräsentantensystem $R = \{0, \ldots, m-1\}$, so entspricht dies genau der Menge der möglichen Reste bei der Division durch m, siehe Seite 52. Die auf diesem System beruhende Datenstruktur nennen wir $\mathcal{Z}_m^+$, der zugehörige kanonische Simplifikator lautet

   $$\text{kanonisch}_{\mathcal{Z}_m^+}(a) := a \bmod m.$$

   In $\mathcal{Z}_m^+$ wird also die kanonische Form von a als Rest bei Division von a durch m berechnet.

2. Das Repräsentantensystem $R = \{-\lfloor \frac{m-1}{2} \rfloor, \ldots, 0, \ldots, \lfloor \frac{m}{2} \rfloor\}$ stellt eine Alternative dar, bei der die Elemente möglichst symmetrisch um 0 angeordnet sind. Die darauf aufbauende Datenstruktur nennen wir $\mathcal{Z}_m^\pm$, der kanonische Simplifikator hierfür

lautet[15]

$$\text{kanonisch}_{\mathcal{Z}_m^{\pm}}(a) := \begin{cases} a \bmod m & \text{falls } a \bmod m \leq \lfloor \frac{m}{2} \rfloor \\ (a \bmod m) - m & \text{sonst.} \end{cases}$$

Sowohl $r \in \mathcal{Z}_m^+$ als auch $r \in \mathcal{Z}_m^{\pm}$ beinhalten für eine Restklasse $[a]_m$ deren kanonischen Repräsentanten und den Modul. Den Repräsentanten in r sprechen wir mit $rep(r)$ an, den Modul erkennt man aus dem Zusammenhang. Für $r \in \mathcal{Z}_m^+$ bzw. $r \in \mathcal{Z}_m^{\pm}$ mit $rep(r) = a$ schreiben wir einfach $a \in \mathcal{Z}_m^+$ und $a \in \mathcal{Z}_m^{\pm}$.

Beispiel

Um $[5]_3$ in der Datenstruktur $\mathcal{Z}_3^+$ darzustellen, speichern wir den kanonischen Repräsentanten $\text{kanonisch}_{\mathcal{Z}_3^+}(5)$ in $rep(r)$ ab und erhalten ein $r \in \mathcal{Z}_3^+$ mit $rep(r) = 2$, wofür wir einfach $2 \in \mathcal{Z}_3^+$ schreiben. Stellen wir $[5]_3$ jedoch in $\mathcal{Z}_3^{\pm}$ dar, so liefert $\text{kanonisch}_{\mathcal{Z}_3^{\pm}}(5)$ die kanonische Form -1, wir erhalten daher $r = -1 \in \mathcal{Z}_3^{\pm}$, und $rep(r)$ liefert hier -1.

Die in (6.12) definierten Operationen auf $\mathbb{Z}_m$ können auf $\mathcal{Z}_m^+$ bzw. $\mathcal{Z}_m^{\pm}$ übertragen werden, indem man mit den Repräsentanten normal in $\mathbb{Z}$ rechnet und das Resultat am Ende in kanonische Form bringt. Die Operation $\diamond \in \{+, -, \cdot\}$ kann also auf $\mathcal{Z}_m^+$ bzw. $\mathcal{Z}_m^{\pm}$ als

$$r \diamond s := \text{kanonisch}(rep(r) \diamond rep(s))$$

definiert werden[16]. Dies ist zugleich ein *allgemeines Rezept*, wie man das Rechnen mit Kongruenzklassen durch das Rechnen auf einem Repräsentantensystem mit einem kanonischen Simplifikator realisieren kann. In dieser Form können die arithmetischen Grundoperationen auf Restklassen leicht am Computer realisiert werden, wir führen exemplarisch im Algorithmus *MultZm* die Multiplikation in $\mathcal{Z}_m^+$ an. Für einen entsprechenden Algorithmus für $\mathcal{Z}_m^{\pm}$ braucht man nur den kanonischen Simplifikator für $\mathcal{Z}_m^{\pm}$ anstelle von jenem für $\mathcal{Z}_m^+$ verwenden.

Algorithmus *MultZm*: Multiplikation in $\mathbb{Z}_m$

```
a ← rep(r)
b ← rep(s)
rep(p) ← kanonisch_{Z_m^+}(a · b)
return p
```

Aufruf: $MultZm(r,s)$
Eingabe: $r,s \in \mathcal{Z}_m^+$
Ausgabe: $p \in \mathcal{Z}_m^+$
mit: $p = r \cdot s$.

Hinsichtlich des Invertierens von $[b]_m \in \mathbb{Z}_m$ kehren wir noch einmal zu (6.13) zurück. Im Fall $\text{ggT}(m,b) = 1$ ist $[b]_m$ invertierbar, und das gesuchte Inverse $[x]_m \in \mathbb{Z}_m$ kann durch Lösen der diophantischen Gleichung $bx + (-k)m = 1$ (für $x,k \in \mathbb{Z}$) gefunden werden. Das Auffinden eines solchen x kann aber direkt durch den erweiterten Euklidschen Algorithmus bewerkstelligt werden, da $ErwGGTZEuklid(m,b)$, siehe Seite 67, neben $\text{ggT}(m,b) = 1$ genau die gesuchten Koeffizienten $-k$ und x be-

[15] Verwendet man wie auf Seite 52 angedeutet eine alternative Definition für Quotient und Rest in $\mathbb{Z}$, so sind die hier vorgestellten kanonischen Simplifikatoren entsprechend anzupassen.

[16] Die Operatoren $+, -, \cdot$ werden überladen, wir erinnern an die Diskussion über Polymorphismus von Seite 58.

rechnet. Auch in *InversZmEuklid* zeigen wir die Variante für $\mathcal{Z}_m^+$, für $\mathcal{Z}_m^\pm$ funktioniert alles wieder analog.

Algorithmus *InversZmEuklid*: Invertieren in $\mathbb{Z}_m$

```
b ← rep(r)
(t, −k, x) ← ErwGGTZEuklid(m, b)
rep(s) ← kanonisch_{Z_m^+}(x)
return s
```

Aufruf: $InversZmEuklid(r)$
Eingabe: $r \in \mathcal{Z}_m^+$
mit: $\mathrm{ggT}(m, rep(r)) = 1$
Ausgabe: $s \in \mathcal{Z}_m^+$
mit: $r \cdot s = 1 \in \mathcal{Z}_m^+$.

Beispiel

Für $[7]_{19} \in \mathbb{Z}_{19}$ soll das multiplikative Inverse berechnet werden. Dazu stellen wir $[7]_{19}$ zuerst in einem Computermodell von $\mathbb{Z}_{19}$ dar, etwa als $7 \in \mathcal{Z}_{19}^+$. Mit $InversZmEuklid(r)$ erhalten wir $s = 11 \in \mathcal{Z}_{19}^+$, also $[7]_{19}^{-1} = [11]_{19}$. Zur Kontrolle ergibt $r \cdot s$ mit $MultZm(r,s)$ tatsächlich $1 \in \mathcal{Z}_{19}^+$. Verwenden wir hingegen das Modell $\mathcal{Z}_{19}^\pm$, so ergibt $InversZmEuklid(r)$ jetzt $s = -8 \in \mathcal{Z}_{19}^\pm$, d.h. $[7]_{19}^{-1} = [-8]_{19}$, und $MultZm(r,s)$ liefert wieder $1 \in \mathcal{Z}_{19}^\pm$.

Systeme von Kongruenzen in $\mathbb{Z}$

Wir wollen nun spezielle Systeme von Kongruenzen betrachten, in denen bei gegebenen $r_1, \ldots, r_n \in \mathbb{Z}$ und $m_1, \ldots, m_n \in \mathbb{Z}$ ein $z \in \mathbb{Z}$ gesucht ist mit

$$z \equiv_{m_1} r_1 \quad \ldots \quad z \equiv_{m_n} r_n. \tag{6.14}$$

Nicht jedes System der Form (6.14) besitzt eine Lösung, ein $z \in \mathbb{Z}$ mit $z \equiv_2 1$ und $z \equiv_4 0$ kann es nicht geben, da z dann sowohl ungerade als auch ein Vielfaches von 4 und damit gerade sein müsste. Die Lösbarkeit von (6.14) kann also bestenfalls nur unter Zusatzbedingungen an die gegebenen Module m_i gegeben sein.

Problemstellung (Chinesisches Restproblem der Dimension n).

Gegeben: $r, m \in \mathbb{Z}^n$
mit: $\mathrm{ggT}(m_i, m_j) = 1$ für $i \neq j$.
Gesucht: $z \in \mathbb{Z}$
mit: $z \equiv_{m_i} r_i \quad$ für $i = 1, \ldots, n$.

Beispiel

Bei gegebenen $r = (5,12)$ und $m = (17,31)$ erfüllt etwa $z = 260$ die geforderten Eigenschaften, da $260 \equiv_{17} 5$ und $260 \equiv_{31} 12$. Weitere Lösungen sind etwa durch 787 oder -267 gegeben. Es gilt aber $z \equiv_{17 \cdot 31} 787$ und $z \equiv_{17 \cdot 31} -267$.

Die Lösung $z \in \mathbb{Z}$ des chinesischen Restproblems ist also nicht eindeutig, der folgende Satz, der schon den alten Griechen und Chinesen bekannt war, garantiert uns zumindest die Existenz einer Lösung.

Satz

Chinesischer Restsatz. Seien $r,m \in \mathbb{Z}^n$ und $\mathrm{ggT}(m_i,m_j) = 1$ für $i \neq j$. Dann existiert ein $z \in \mathbb{Z}$ mit

$$z \equiv_{m_i} r_i \quad \text{für } i = 1,\ldots,n. \tag{6.15}$$

Die Menge aller Lösungen von (6.15) in $\mathbb{Z}$ ist dann durch $[z]_{m_1\cdot\ldots\cdot m_n}$ gegeben.

Im Beweis des chinesischen Restsatzes erweist sich folgendes Lemma als hilfreich.

Lemma

Chinesischer Restsatz für 2 Kongruenzen. Für $r_1,m_1,r_2,m_2 \in \mathbb{Z}$ mit $\mathrm{ggT}(m_1,m_2) = 1$ existiert ein $z' \in \mathbb{Z}$ mit $z' \equiv_{m_1} r_1$ und $z' \equiv_{m_2} r_2$. Weiters gilt für alle $z \in \mathbb{Z}$

$$z \equiv_{m_1} r_1 \text{ und } z \equiv_{m_2} r_2 \iff z \equiv_{m_1 m_2} z'. \tag{6.16}$$

Beweis. Jede Kongruenz $z \equiv_m r$ ist gleichbedeutend mit der Existenz eines $x \in \mathbb{Z}$ mit $z = mx + r$. Die Existenz von z' ist daher äquivalent zur Existenz von $x',y' \in \mathbb{Z}$ mit $m_1x' + r_1 = z' = m_2y' + r_2$. Das wiederum entspricht der Lösbarkeit der diophantischen Gleichung $m_1x' - m_2y' = r_2 - r_1$, die wegen $\mathrm{ggT}(m_1,m_2) = 1$ und $1|(r_2 - r_1)$ gegeben ist, vgl. mit dem Satz von Seite 68. Zum Nachweis von (6.16) sei $z \in \mathbb{Z}$ beliebig aber fix. Falls $z \equiv_{m_1} r_1$ und $z \equiv_{m_2} r_2$ gilt, so existieren $x,y \in \mathbb{Z}$ mit $m_1x + r_1 = z = m_2y + r_2$, womit

$$z - z' = m_1(x - x') = m_2(y - y')$$

gilt. Wegen $\mathrm{ggT}(m_1,m_2) = 1$ teilt m_2 aber $x - x'$, also $x - x' = m_2q$ für ein geeignetes $q \in \mathbb{Z}$, so dass schlussendlich $z - z' = m_1m_2 \cdot q$ und damit $z \equiv_{m_1m_2} z'$ ist. Gilt umgekehrt $z \equiv_{m_1m_2} z'$, so ist für ein passendes $q \in \mathbb{Z}$

$$z = z' + m_1m_2 \cdot q = m_1(x' + m_2q) + r_1, \text{ d.h. } z \equiv_{m_1} r_1.$$

Analog dazu zeigt man $z \equiv_{m_2} r_2$. □

Wir können damit an den Beweis des chinesischen Restsatzes herangehen.

Beweis. (Chinesischer Restsatz) Wir verwenden Induktion nach n. Für $n = 1$ ist die Aussage klarerweise wahr. Für den Induktionsschritt von $n-1$ auf n seien $n > 1$ und $r,m \in \mathbb{Z}^n$ mit $\mathrm{ggT}(m_i,m_j) = 1$ für $i \neq j$ beliebig aber fix. Dann ist (6.15) aufgrund des Lemmas äquivalent zu

$$z \equiv_{m_1m_2} z' \text{ und } z \equiv_{m_i} r_i \text{ für } i = 3,\ldots,n \tag{6.17}$$

mit $z' \in \mathbb{Z}$ wie im Lemma konstruiert. Da (6.17) nur mehr aus $n-1$ Kongruenzen besteht und die Module m_1m_2 und m_i (für $i = 3,\ldots,n$) wieder paarweise relativ prim sind, existiert laut Induktionsvoraussetzung so ein $z \in \mathbb{Z}$, und die Menge aller Lösungen von (6.17) – und damit auch von (6.15) – ist durch $[z]_{(m_1m_2)\cdot m_3\cdot\ldots\cdot m_n} = [z]_{m_1\cdot\ldots\cdot m_n}$ gegeben. □

Der Induktionsbeweis legt einen rekursiven Algorithmus zum Lösen von (6.15) nahe. Man löst die ersten beiden Kongruenzen in (6.15) und ersetzt sie durch *eine* neue Kongruenz $z \equiv_{m_1 m_2} z'$, siehe (6.17). Die Berechnung des z' beruht auf obigem Lemma, dessen Beweis ebenfalls konstruktiver Natur ist und auf das Lösen einer diophantischen Gleichung führt. Mit dieser Reduktion fährt man rekursiv fort, solange in (6.15) $n > 1$ ist. Im Fall $n = 1$ ist der Basisfall der Rekursion erreicht, eine Lösung ist hier durch $r_1 \bmod m_1$ leicht zu berechnen. Da für zwei Lösungen $z, z' \in \mathbb{Z}$ des chinesischen Restproblems stets $z \equiv_{m_1 \cdot \ldots \cdot m_n} z'$ gilt, gibt es genau eine Lösung $z \in \mathbb{Z}$ mit $0 \leq z < m_1 \cdot \ldots \cdot m_n$, der Algorithmus *CRAZ* berechnet genau diese.

Algorithmus *CRAZ*: Chinesischer Restalgorithmus in $\mathbb{Z}$

```
n ← |r|
if n = 1
   z ← r_1 mod m_1
else
   (x',y') ←
      LöseLinDiophant(m_1, − m_2, r_2 − r_1)
   z' ← m_1 · x' + r_1 mod m_1 · m_2
   r_2 ← z', m_2 ← m_1 · m_2
   z ← CRAZ(r_{2:n}, m_{2:n})
return z
```

Aufruf: $CRAZ(r,m)$
Eingabe: $r,m \in \mathbb{Z}^n$
mit: $\mathrm{ggT}(m_i,m_j) = 1$ für $i \neq j$.
Ausgabe: $z \in \mathbb{Z}$
mit: $z \equiv_{m_i} r_i$ für $i = 1,\ldots,n$ und $0 \leq z < m_1 \cdot \ldots \cdot m_n$.

Beispiel

Gesucht sei eine ganze Zahl z mit

$$z \equiv_{17} 5 \quad z \equiv_{31} 12 \quad z \equiv_{23} 11.$$

Zur Lösung dieser Aufgabe rufen wir mit $r = (5,12,11)$ und $m = (17,31,23)$ den Algorithmus $CRAZ(r,m)$ auf. Im ersten Rekursionsschritt wird mit $x' = 77$ und $y' = 42$ eine Lösung der diophantischen Gleichung $17x' - 31y' = 7$ berechnet. Daraus ergibt sich mit $m_1 m_2 = 527$ das Zwischenresultat $z' = 17 \cdot 77 + 5 \bmod 527 = 260$. Im rekursiven Aufruf lauten die neuen Eingaben dann $r = (260,11)$ und $m = (527,23)$. Die diophantische Gleichung $527x' - 23y' = -249$ besitzt eine Lösung $x' = -2739$ (und $y' = -62748$), aus der sich mit $m_1 m_2 = 12121$ dann $z' = 527 \cdot (-2739) + 260 \bmod 12121 = 11327$ ergibt. Mit $r = (11327)$ und $m = (12121)$ ist der Basisfall der Rekursion erreicht, und $CRAZ(r,m)$ liefert letztlich $z = 11327$.

Bemerkung. Im Algorithmus kann für z' eine beliebige zu $m_1 \cdot x' + r_1$ modulo $m_1 m_2$ kongruente ganze Zahl genommen werden, die Restbildung modulo $m_1 m_2$ ist nicht nötig, sie wird im Basisfall ohnedies durchgeführt. Eine Division durch $m_1 m_2$ in jedem Reduktionsschritt ist jedoch einer einzigen Division durch $m_1 \cdot \ldots \cdot m_n$ am Ende der Rekursion vorzuziehen, da dadurch die Berechnungen mit kleineren Zahlen zu bewältigen sind.

7 Rationale Zahlen

Mathematische Grundlagen

Rationale Zahlen werden als Erweiterung der ganzen Zahlen eingeführt, da ganze Zahlen bzgl. der Multiplikation keine Inversen in $\mathbb{Z}$ besitzen und daher keine Division im Sinne einer Umkehrung der Multiplikation ermöglichen. Man bildet dazu Paare von ganzen Zahlen und betrachtet die Relation

$$(z,n) \sim (z',n') :\Leftrightarrow z \cdot n' = n \cdot z'.$$

Satz

Die Relation $\sim$ ist eine Äquivalenzrelation auf $\mathbb{Z} \times (\mathbb{Z} \setminus \{0\})$.

Der Beweis dieses Satzes ist dem Leser überlassen, siehe Übung II.10.

Definition

Brüche und Brucharithmetik. Die Äquivalenzklasse von (z,n) bezüglich $\sim$ wird als $\frac{z}{n}$ geschrieben. Jedes $\frac{z}{n}$ nennt man einen *Bruch* (über $\mathbb{Z}$), wobei z der *Zähler* und n der *Nenner* des Bruchs genannt wird. Die rationalen Zahlen sind dann definiert als

$$\mathbb{Q} := \left\{ \frac{z}{n} \;\middle|\; z,n \in \mathbb{Z} \wedge n \neq 0 \right\}.$$

Auf $\mathbb{Q}$ definiert man nun Addition und Multiplikation wie folgt:

$$\frac{z}{n} + \frac{z'}{n'} := \frac{z \cdot n' + z' \cdot n}{n \cdot n'} \qquad \frac{z}{n} \cdot \frac{z'}{n'} := \frac{z \cdot z'}{n \cdot n'}. \tag{7.18}$$

Wie bei $\mathbb{Z}_m$ sind auch hier die Operationen wohldefiniert, hängen also nicht von der Wahl der Repräsentanten ab, siehe Übung II.11. Für $z \neq 0$ ist das multiplikative Inverse zu $\frac{z}{n}$ durch $\frac{n}{z}$ gegeben, und die Division von Brüchen ist für $z' \neq 0$ durch

$$\frac{z}{n} \Big/ \frac{z'}{n'} := \frac{z}{n} \cdot \frac{n'}{z'} \tag{7.19}$$

festgelegt. Weiters kann jedes $z \in \mathbb{Z}$ mit dem Bruch $\frac{z}{1}$ identifiziert werden, so dass $\mathbb{Q}$ tatsächlich als eine *Erweiterung* von $\mathbb{Z}$ betrachtet werden kann. In dieser Erweiterung kann man nun auch $z \in \mathbb{Z}$ durch $n \in \mathbb{Z}$ mit $n \neq 0$ „dividieren“, indem z und n als $\frac{z}{1}$ bzw. $\frac{n}{1} \in \mathbb{Q}$ interpretiert werden, und die Division z/n als Multiplikation von z mit dem Inversen von n in $\mathbb{Q}$ gesehen wird[17], d.h.

$$z/n = \frac{z}{1} \cdot \left(\frac{n}{1}\right)^{-1} = \frac{z}{1} \cdot \frac{1}{n} = \frac{z}{n}.$$

[17] Diese Division z/n hat nun *nichts* mit der Division mit Rest z div n zu tun.

Beispiel

Es gilt $(1,2) \sim (2,4)$, da $1 \cdot 4 = 2 \cdot 2$. Damit sind die Äquivalenzklassen von $(1,2)$ und $(2,4)$ identisch, d.h. $\frac{1}{2} = \frac{2}{4}$. Nach (7.18) gilt

$$\frac{1}{2} + \frac{2}{4} = \frac{1 \cdot 4 + 2 \cdot 2}{2 \cdot 4} = \frac{8}{8} \qquad \frac{1}{2} \cdot \frac{2}{4} = \frac{1 \cdot 2}{2 \cdot 4} = \frac{2}{8}.$$

Die Division der beiden Brüche führt nach (7.19) auf

$$\frac{1}{2} \Big/ \frac{2}{4} = \frac{1 \cdot 4}{2 \cdot 2} = \frac{4}{4}$$

und die „Division“ der ganzen Zahlen 2 durch 3 (in $\mathbb{Q}$) resultiert in $\frac{2}{3} \in \mathbb{Q}$.

Aus der Definition von $\sim$ folgt für jedes $t \in \mathbb{Z} \setminus \{0\}$ sofort

$$\frac{z}{n} = \frac{t \cdot z}{t \cdot n}.$$

Je nach dem, ob man diese Gleichheit von links nach rechts oder von rechts nach links liest, heißt das, dass man Brüche mit ganzen Zahlen im Zähler und Nenner *erweitern* oder gemeinsame Teiler im Zähler und Nenner *kürzen* kann. Kürzt man den ggT von Zähler und Nenner, so sind Zähler und Nenner des resultierenden Bruches relativ prim.

Satz

Für $z, n \in \mathbb{Z}$ mit $z \neq 0$ oder $z \neq 0$ und $t = \mathrm{ggT}(z,n)$ gilt immer

$$\mathrm{ggT}(z \operatorname{div} t, n \operatorname{div} t) = 1.$$

Beweis. Laut Voraussetzung existieren $q_z, q_n \in \mathbb{Z}$ mit $z = t \cdot q_z$ und $n = t \cdot q_n$, somit ist $q_z = z \operatorname{div} t$ und $q_n = n \operatorname{div} t$. Angenommen $g := \mathrm{ggT}(q_z, q_n) \neq 1$. Da der ggT nie negativ ist, ist sogar $g > 1$. Damit ist $t \cdot g > t$ ebenfalls ein gemeinsamer Teiler von z und n im Widerspruch zur Maximalität von t. □

Dieser Satz spielt bei der Darstellung rationaler Zahlen am Computer eine wesentliche Rolle.

Rationale Zahlen am Computer

Analog zu $\mathbb{Z}_m$ werden auch die Äquivalenzklassen in $\mathbb{Q}$ am Computer am besten durch *kanonische Repräsentanten* dargestellt.

Computerrepräsentation (Datenstruktur $\mathcal{Q}$ für rationale Zahlen)**.** Zur Darstellung rationaler Zahlen verwenden wir eine Datenstruktur $\mathcal{Q}$, in der wir für eine rationale Zahl ein kanonisches Paar von Zähler und Nenner abspeichern. Zum Ermitteln der kanonischen Form dient wieder ein kanonischer Simplifikator für die Relation $\sim$. Als kanonische Form eines Paares (z,n) wählen wir ein Paar $(z',n') \sim (z,n)$ so,

dass z' und n' relativ prim sind mit $n' > 0$. Der letzte Satz aus dem vorangegangenen Abschnitt zeigt, dass die kanonische Form von (z,n) im Wesentlichen durch Kürzen durch $t = \text{ggT}(z,n)$ ermittelt werden kann. Die Forderung nach positiver zweiter Komponente ist durch Erweitern mit -1 (falls nötig) leicht zu erfüllen. Der kanonische Simplifikator ist also

$$\text{kanonisch}_{\mathcal{Q}}((z,n)) := \big((z \operatorname{div} t) \cdot \text{sgn}(n), |n| \operatorname{div} t\big) \quad \text{wobei } t = \text{ggT}(z,n).$$

Zähler und Nenner von $a \in \mathcal{Q}$ sprechen wir durch *zähler*(a) und *nenner*(a). Für $a \in \mathcal{Q}$ mit *zähler*$(a) = z$ und *nenner*$(a) = n$ schreiben wir der Einfachheit halber wieder $(z,n) \in \mathcal{Q}$.

Beispiel

Die kanonische Form von $(10,-15)$ erhalten wir mit $\text{ggT}(10,15) = 5$ als $((10 \operatorname{div} 5)(-1), |-15| \operatorname{div} 5) = (-2,3)$.

Dem *Rechnen mit Kongruenzklassen* sind wir auf Seite 72 schon begegnet, so dass sich auch in $\mathbb{Q}$ die arithmetischen Grundoperationen auf den kanonischen Repräsentanten in $\mathcal{Q}$ realisieren lassen. Dies führt für (z,n), $(z',n') \in \mathcal{Q}$ auf

$$\begin{aligned}
(z,n) \pm (z',n') &:= \text{kanonisch}_{\mathcal{Q}}((z \cdot n' \pm z' \cdot n, n \cdot n')) \\
(z,n) \cdot (z',n') &:= \text{kanonisch}_{\mathcal{Q}}((z \cdot z', n \cdot n')) \\
(z,n)^{-1} &:= \text{kanonisch}_{\mathcal{Q}}((n,z)) \quad \text{für } z \neq 0 \\
(z,n) \,/\, (z',n') &:= (z,n) \cdot (n',z') \quad \text{für } z' \neq 0.
\end{aligned}$$

Diese Definitionen lassen sich direkt in Algorithmen zur Arithmetik mit Brüchen umsetzen, wir zeigen exemplarisch einen Additionsalgorithmus *AddQ*.

Algorithmus *AddQ*: Addition in $\mathbb{Q}$

```
z ← zähler(a), n ← nenner(a)
z' ← zähler(b), n' ← nenner(b)
(z,n) ← kanonisch_Q((z · n' + z' · n, n · n'))
zähler(s) ← z, nenner(s) ← n
return s
```

Aufruf: $AddQ(a,b)$
Eingabe: $a,b \in \mathcal{Q}$
Ausgabe: $s \in \mathcal{Q}$
mit: $s = a + b$.

Beispiel

Um $\frac{1089}{140}, \frac{633}{350} \in \mathbb{Q}$ zu addieren, stellt man die beiden Brüche zuerst als $a = (1089,140) \in \mathcal{Q}$ und $b = (633,350) \in \mathcal{Q}$ dar. Nach Aufruf von $AddQ(a,b)$ wird zuerst $(1089 \cdot 350 + 633 \cdot 140, 140 \cdot 350) = (469770,49000)$ berechnet, dessen kanonische Form sich nach Kürzen durch $\text{ggT}(469770,49000) = 70$ als $(6711,700) \in \mathcal{Q}$ ergibt. Das entspricht natürlich dem Bruch $\frac{6711}{700} \in \mathbb{Q}$.

Beim Rechnen mit Brüchen können Zähler und Nenner beliebig groß werden. Um fehlerhafte Resultate aufgrund von Überlauf zu vermeiden, empfiehlt es sich daher, diese als beliebig lange ganze Zahlen darzustellen. Der Hauptaufwand in

den arithmetischen Grundalgorithmen ist das Berechnen des größten gemeinsamen Teilers beim Ermitteln der kanonischen Form, der wie auf Seite 66 festgestellt von der Länge der Operanden abhängt. Eine Reduktion des Aufwands basiert darauf, die ggT-Berechnung auf kleinere Zahlen zu verlagern. Im Fall der Addition bzw. Subtraktion ist dabei $\text{ggT}(u',v')$ mit $u' = z \cdot n' \pm z' \cdot n$ und $v' = n \cdot n'$ zu berechnen. Mit $t = \text{ggT}(n,n')$ existieren $N,N' \in \mathbb{N}$ mit $n = t \cdot N$ und $n' = t \cdot N'$. Damit ist

$$u' = z \cdot tN' \pm z' \cdot tN = t(\underbrace{z \cdot N' \pm z' \cdot N}_{=u}) \quad \text{und} \quad v' = t \cdot \underbrace{n \cdot N'}_{=v},$$

so dass t jedenfalls ein gemeinsamer Teiler von u' und v' ist. Nach dem Dividieren von u' und v' durch t verbleibt also die Berechnung von $\text{ggT}(z \cdot N' \pm z' \cdot N, n \cdot N')$. Man kann also zur Berechnung von $\text{ggT}(u',v',)$ auf die Berechnung von $\text{ggT}(z \cdot N' \pm z' \cdot N, n \cdot N')$ zurückgreifen. Der folgende Satz ermöglicht es, letztere ggT-Berechnung durch eine einfachere ggT-Berechnung, d.h. eine mit kürzeren Operanden, zu ersetzen.

Satz

Henrici[18]Addition. Seien $z,n,z',n' \in \mathbb{Z}$ mit $n,n' \neq 0$ und $\text{ggT}(z,n) = \text{ggT}(z',n') = 1$, und seien weiters $t = \text{ggT}(n,n')$, $N = n \operatorname{div} t$ und $N' = n' \operatorname{div} t$. Dann ist

$$\text{ggT}(z \cdot N' \pm z' \cdot N, n \cdot N') = \text{ggT}(z \cdot N' \pm z' \cdot N, t).$$

Beweis. Es ist $n \cdot N' = t \cdot N \cdot N'$. Wäre nun $T > 1$ ein gemeinsamer Primteiler von N und $z \cdot N' \pm z' \cdot N$, dann müsste T auch $z \cdot N'$ teilen, und damit müsste $T|z$ oder $T|N'$ gelten. Im Fall $T|z$ wäre T ein gemeinsamer Teiler von z und n im Widerspruch zu $\text{ggT}(z,n) = 1$. Im Fall von $T|N'$ wäre aber T auch ein gemeinsamer Teiler von N und N', die wegen des Satzes von Seite 77 aber relativ prim sind. Somit kann $T > 1$ kein gemeinsamer Primteiler von N und $z \cdot N' \pm z' \cdot N$ sein, und damit können N und $z \cdot N' \pm z' \cdot N$ keinen gemeinsamen Teiler größer 1 haben. Mit dem selben Argument kann N' keinen gemeinsamen Teiler mit $z \cdot N' \pm z' \cdot N$ haben. Damit ist

$$\text{gT}(z \cdot N' \pm z' \cdot N, t \cdot N \cdot N') = \text{gT}(z \cdot N' \pm z' \cdot N, t)$$

und schlussendlich auch $\text{ggT}(z \cdot N' \pm z' \cdot N, n \cdot N') = \text{ggT}(z \cdot N' \pm z' \cdot N, t)$. □

Das auf dieser Erkenntnis aufbauende Verfahren *AddQHenrici* nennt man auch den *Henrici Algorithmus* zur Addition von Brüchen. Da die Operanden a und b in kanonischer Form sind, ist das in *AddQHenrici* berechnete Resultat s wieder in kanonischer Form.

Beispiel

Mit $a = (1089,140) \in \mathcal{Q}$ und $b = (633,350) \in \mathcal{Q}$ berechnet $\mathit{AddQHenrici}(a,b)$ zuerst $t = \text{ggT}(140,350) = 70$ bzw. $N = 140/70 = 2$ und $N' = 350/70 = 5$. Die vorläufigen Zähler und Nenner ergeben sich nun als $u = 1089 \cdot 5 + 633 \cdot 2 = 6711$ und $v = 140 \cdot 5 = 700$. Mit $\text{ggT}(6711,70) = 1$ führt das Verfahren wieder auf $\frac{6711}{700}$.

[18] Henrici, Peter: 1923–1987, Schweizer Mathematiker. Die ihm zugeschriebenen Algorithmen zum Rechnen mit rationalen Zahlen gelten als die ersten Arbeiten, in denen explizit rationale Zahlen am Computer behandelt werden.

Algorithmus *AddQHenrici*: Henrici Addition in $\mathbb{Q}$

$z \leftarrow zähler(a), n \leftarrow nenner(a)$
$z' \leftarrow zähler(b), n' \leftarrow nenner(b)$
$t \leftarrow \text{ggT}(n,n'), N \leftarrow n \text{ div } t, N' \leftarrow n' \text{ div } t$
$u \leftarrow z \cdot N' + z'N, v \leftarrow n \cdot N', t' \leftarrow \text{ggT}(u,t)$
$zähler(s) \leftarrow u \text{ div } t'$
$nenner(s) \leftarrow v \text{ div } t'$
return s

Aufruf: $AddQHenrici(a,b)$
Eingabe: $a,b \in \mathcal{Q}$
Ausgabe: $s \in \mathcal{Q}$
mit: $s = a + b$.

Auch ein Multiplikationsalgorithmus basiert direkt auf der Definition der Multiplikation. Hier besagt ein weiterer Satz von Henrici, dass im Fall von kanonischen Brüchen als Input nach „kreuzweisem Kürzen", d.h. jeweils gemeinsame Teiler eines Zählers und des anderen Nenner werden gekürzt, keine gemeinsamen Faktoren im Zähler und Nenner des Produkts enthalten sein können.

Satz

Henrici Multiplikation. Seien $z,n,z',n' \in \mathbb{Z}$ mit $n,n' \neq 0$ und $\text{ggT}(z,n) = \text{ggT}(z',n') = 1$, und seien $t = \text{ggT}(z,n')$, $t' = \text{ggT}(z',n)$ und $Z = z \text{ div } t, N = n \text{ div } t', Z' = z' \text{ div } t', N' = n' \text{ div } t$. Dann ist

$$\text{ggT}(Z \cdot Z', N \cdot N') = 1.$$

Der Beweis dieses Satzes verwendet die gleiche Argumentation wie jener zur Addition, seine Anwendung findet der Satz im Algorithmus *MultQHenrici*. Auch hier ist bei Operanden a und b in kanonischer Form das Resultat p wieder automatisch in kanonischer Form.

Algorithmus *MultQHenrici*: Henrici Multiplikation in $\mathbb{Q}$

$z \leftarrow zähler(a), n \leftarrow nenner(a)$
$z' \leftarrow zähler(b), n' \leftarrow nenner(b)$
$t \leftarrow \text{ggT}(z,n'), t' \leftarrow \text{ggT}(z',n)$
$Z \leftarrow z \text{ div } t, N \leftarrow n \text{ div } t'$
$Z' \leftarrow z' \text{ div } t', N' \leftarrow n' \text{ div } t$
$zähler(p) \leftarrow Z \cdot Z'$
$nenner(p) \leftarrow N \cdot N'$
return p

Aufruf: $MultQHenrici(a,b)$
Eingabe: $a,b \in \mathcal{Q}$
Ausgabe: $p \in \mathcal{Q}$
mit: $p = a \cdot b$.

Beispiel

Zum Multiplizieren von $\frac{1089}{140}$ und $\frac{633}{350}$ ist im Standardverfahren die Berechnung von $\text{ggT}(689337,49000)$ notwendig, der Henrici Algorithmus hingegen führt die Berechnungen $\text{ggT}(1089,350)$ und $\text{ggT}(140,633)$ mit kürzeren Eingabezahlen durch. Sowohl die Standardmethode als auch die Henrici Multiplikation liefern das Endergebnis $\frac{689337}{49000}$.

Die Henrici Algorithmen bringen gegenüber den klassischen Algorithmen keine Verbesserung der asymptotischen Komplexität (gemessen in der Länge der Zahlen,

die in die ggT-Berechnung eingehen), jedoch besitzen sie einen deutlich niedrigeren Proportionalitätsfaktor, was in der Praxis schon bei relativ kleinen Inputs niedrigere Rechenzeiten zur Folge hat.

8 Reelle Zahlen

Mathematische Grundlagen

In der Analysis wird die Menge $\mathbb{R}$ der reellen Zahlen als angeordneter ordnungsvollständiger Körper eingeführt. Dieses Vorgehen ist auch mit der intuitiven Vorstellung verbunden, dass die reellen Zahlen – im Gegensatz zu den rationalen Zahlen – lückenlos sein sollen. Im Alltag begegnen uns reelle Zahlen meist in Dezimalpunktschreibweise, etwa -123.45 oder 0.00765. Ähnlich wie bei den natürlichen Zahlen auf Seite 48 lassen sich aber auch die reellen Zahlen bezüglich einer beliebigen Basis $B \geq 2$ darstellen, wobei durch Verschiebung die erste von Null verschiedene Ziffer dann einheitlich nach dem Punkt steht. Diese Verschiebung erreicht man durch Multiplikation mit einer geeigneten Potenz der Basis. Wir stützen uns auf folgenden Satz aus der Analysis.

Satz

Basisdarstellung reeller Zahlen. Seien $x \in \mathbb{R}$ mit $x \neq 0$ und $B \in \mathbb{N}$ mit $B \geq 2$. Dann existiert ein Exponent $e \in \mathbb{Z}$, so dass x sich in der Form

$$x = v \cdot B^e \cdot \sum_{i=1}^{\infty} z_i B^{-i} \tag{8.20}$$

mit Vorzeichen $v \in \{-1, +1\}$ und Ziffern $z_i \in \{0, \ldots, B-1\}$ darstellen lässt, wobei für die führende Ziffer $z_1 \neq 0$ gilt.

Als alternative Schreibweise[19] für (8.20) verwenden wir $x = \pm\,(0.z_1 z_2 \ldots)_B \cdot B^e$ bzw. auch $x = \pm 0.z_1 z_2 \ldots \cdot 10^e$ im Fall $B = 10$. Ein wesentlicher Unterschied zur Darstellung natürlicher Zahlen in (5.1) ist also, dass in (8.20) zur Darstellung einer reellen Zahl eine in der Regel unendliche Ziffernfolge $(z_i)_{i \in \mathbb{N}}$ benötigt wird. Weiters ist (8.20) nur eindeutig, falls x nicht von der Form

$$x = \pm k B^{-l} \quad \text{mit } k, l \in \mathbb{N} \tag{8.21}$$

ist. Im Fall (8.21) gibt es dann genau zwei Darstellungen zur Basis B der Form (8.20). Bei (8.21) handelt es sich stets um eine rationale Zahl mit der zusätzlichen Eigenschaft, dass in einer der beiden Darstellungen ab einem bestimmten Index alle Ziffern gleich Null sind. Wir veranschaulichen dies im folgenden Beispiel.

[19] Es ist auch $\pm 0, z_1 z_2 \ldots \cdot 10^e$ an Stelle von $\pm 0.z_1 z_2 \ldots \cdot 10^e$ gebräuchlich.

Beispiel

Im Dezimalsystem mit $B = 10$ gilt für $-100\sqrt{2} \in \mathbb{R}$ die Darstellung $-0.1414213\ldots \cdot 10^3$ mit führender Ziffer $z_1 = 1$. Diese Darstellung ist eindeutig, da $-100\sqrt{2}$ irrational ist, aber (8.21) stets in $\mathbb{Q}$ liegt. Insbesondere ist die Ziffernfolge nicht periodisch. Für $\frac{1}{25} \in \mathbb{R}$ gilt die Darstellung $0.40000\ldots \cdot 10^{-1}$ mit führender Ziffer $z_1 = 4$, weiters gilt $z_i = 0$ für alle $i \geq 2$. Wegen

$$\frac{1}{25} = 4 \cdot 10^{-2}$$

gibt es noch genau eine weitere Darstellung zur Basis 10. Diese lautet $0.3999\ldots \cdot 10^{-1}$ mit führender Ziffer $z_1 = 3$ und $z_i = 9$ für alle $i \geq 2$.

Die reellen Zahlen am Computer

Der gängige Zugang zum Rechnen mit reellen Zahlen am Computer ist die *Gleitkommaarithmetik*, die Gegenstand dieses Abschnitts ist. Dabei werden reelle Zahlen, ähnlich wie die Zahlen fixer Länge auf Seite 52, mit Hilfe eines endlichen Zifferntupels dargestellt. Mit Blick auf (8.20) folgt unmittelbar, dass auf diese Weise nicht alle reelle Zahlen darstellbar sind, und es sowohl Einschränkungen an die Länge der Ziffernfolge als auch den Bereich des Exponenten geben muss. Wir beginnen die Diskussion mit der Einführung der *normalisierten Gleitkommazahlen*, bei denen es sich zusammen mit 0 um jene Zahlen handelt, für die Ziffernfolge in (8.20) endlich ist.

Definition

Gleitkommazahlen. Seien $B, t \in \mathbb{N}$ mit $B \geq 2$. Die Zahl $x \in \mathbb{R}$ ist eine *t-stellige normalisierte Gleitkommazahl*[20] *zur Basis B* genau dann, wenn

$$x = 0 \quad \text{oder} \quad x = v \cdot m \cdot B^e \quad \text{mit } m = \sum_{i=1}^{t} z_i B^{-i} \tag{8.22}$$

und den Ziffern $z_i \in \{0, 1\ldots, B-1\}$, $z_1 \neq 0$, dem Exponenten $e \in \mathbb{Z}$ und dem Vorzeichen $v \in \{1, -1\}$. Die Zahl m bezeichnet man als normalisierte Mantisse der Länge t, wir verwenden dafür auch die Schreibweise $m = (0.z_1 \ldots z_t)_B$.

Die Normalisierung im Fall $x \neq 0$, d.h. die Forderung $z_1 \neq 0$, hat zur Folge, dass die Mantisse m der Abschätzung

$$B^{-1} \leq m \leq 1 - B^{-t} < 1 \tag{8.23}$$

genügt.

[20] Die Definition einer normalisierten Gleitkommazahl ist in der Literatur nicht einheitlich. So findet man etwa in [6] auch die Darstellung $x = v\bar{m}B^{e-t}$ mit $B^{t-1} \leq \bar{m} \leq B^t - 1$, in [3] wiederum wird das Vorzeichen als Teil der Mantisse aufgefasst, zudem wird für 0 die Mantisse mit $m = 0$ definiert. Auf die Kernaussagen zur Gleitkommaarithmetik haben diese Unterschiede jedoch keinerlei Auswirkung.

Beispiel

Eine 5-stellige normalisierte Gleitkommazahl zur Basis 10 mit Exponent 3 ist $0.21425 \cdot 10^3$. Wegen

$$\begin{aligned} 214.25 &= 2^7 + 2^6 + 2^4 + 2^2 + 2^1 + 2^{-2} \\ &= \left(2^{-1} + 2^{-2} + 2^{-4} + 2^{-6} + 2^{-7} + 2^{-10}\right) \cdot 2^8 \end{aligned}$$

gilt

$$0.21425 \cdot 10^3 = (0.1101011001)_2 \cdot 2^8,$$

wobei die rechte Seite eine 10-stellige Gleitkommazahl zur Basis 2 mit Exponenten 8 ist.

Aus (8.22) ist ersichtlich, dass es sich bei einer t-stelligen Gleitkommazahl x stets um eine rationale Zahl handelt. Insbesondere kann damit keine irrationale Zahl wie etwa $\sqrt{2}$ als Gleitkommazahl dargestellt werden. Doch auch bei rationalen Zahlen kann es sein, dass eine Darstellung mit endlicher Ziffernfolge nicht möglich ist.

Beispiel

Wäre $\frac{1}{10} \in \mathbb{Q}$ eine Gleitkommazahl zur Basis $B = 2$, so müsste wegen (8.22) der Nenner 10 als Potenz von 2 darstellbar sein. Dies ist jedoch nicht möglich, also besitzt bei $B = 2$ die Zahl $\frac{1}{10}$ eine unendliche Ziffernfolge. Ähnlich sieht man, dass $\frac{1}{3} \in \mathbb{Q}$ nicht als Gleitkommazahl zur Basis $B = 10$ darstellbar ist.

Bemerkung. Anders als in den vorangegangenen Abschnitten führen wir für Gleitkommazahlen keine eigene Datenstruktur ein. Unser Schwerpunkt liegt in diesem Abschnitt nicht auf der algorithmischen Umsetzung der elementaren Rechenoperationen sondern auf dem Rechenfehler, der durch die Verwendung von Gleitkommazahlen entsteht.

Computerrepräsentation (Gleitkommasystem)**.** Für die Darstellung von Gleitkommazahlen am Computer fixiert man in (8.22) neben der Basis B und der Mantissenlänge t auch mit $e_{\min}$ und $e_{\max} \in \mathbb{Z}$ einen Wertebereich für den Exponenten. Zu gegebenen $B, t \in \mathbb{N}$ mit $B \geq 2$ und $e_{\min}, e_{\max} \in \mathbb{Z}$ mit $e_{\min} \leq e_{\max}$ und $e_{\max} \geq 1$ versteht man unter einem *Gleitkommasystem* die Menge

$$\begin{aligned} \mathbb{G}_{B,t,e_{\min},e_{\max}} := \{0\} \cup \big\{ & x \in \mathbb{R} \mid x \text{ ist eine normalisierte } t\text{-stellige Gleit-} \\ & \text{kommazahl zur Basis } B \text{ mit } e_{\min} \leq e \leq e_{\max} \big\}. \end{aligned} \tag{8.24}$$

Damit lassen sich endlich viele t-stellige normalisierte Gleitkommazahlen zur Basis B darstellen.

Beispiel

Für $B = 2$ und $t = 3$ sind $(0.100)_2 = 0.5$, $(0.101)_2 = 0.625$, $(0.110)_2 = 0.75$, und $(0.111)_2 = 0.875$ die möglichen (normalisierten) Mantissen, die wie in (8.23) behauptet allesamt zwischen $2^{-1} = 0.5$ und $1 - 2^{-3} = 0.875$ liegen. Mit $e_{\min} = -1$ und $e_{\max} = 2$ lautet das zugehörige Gleitkommasystem dann

$$\begin{aligned} \mathbb{G}_{2,3,-1,2} = \{ & 0, \pm 0.25, \pm 0.3125, \pm 0.375, \pm 0.4375, \pm 0.5, \pm 0.625, \pm 0.75, \\ & \pm 0.875, \pm 1, \pm 1.25, \pm 1.5, \pm 1.75, \pm 2, \pm 2.5, \pm 3, \pm 3.5 \}. \end{aligned} \tag{8.25}$$

Darin hat etwa -0.375 die Gleitkommadarstellung $-(0.110)_2 \cdot 2^{-1}$, 1.25 ist durch $(0.101)_2 \cdot 2^1$ dargestellt. Wählt man $e_{\max} = 3$, so kommen noch die Zahlen ± 4, ± 5, ± 6 und ± 7 hinzu.

Gleitkommasysteme bilden stets echte und diskrete Teilmengen von $\mathbb{R}$. Die Elemente von $\mathbb{G}_{B,t,e_{\min},e_{\max}}$ werden auch als Maschinenzahlen bezeichnet, da (bei Gleitkommaarithmetik) nur mit diesen am Computer gerechnet werden kann. Durch die Forderung $e_{\max} \geq 1$ wird sichergestellt, dass $1 = (0.100\ldots00)_B \cdot B^1$ stets im Gleitkommasystem enthalten ist. Aus den das Gleitkommasystem festlegenden Werten B, t, $e_{\min}$ und $e_{\max}$ lassen sich nun weitere charakterisierende Kenngrößen ableiten. Dazu zählen die in $\mathbb{G}_{B,t,e_{\min},e_{\max}}$ *größte normalisierte Zahl*

$$R_{\max} = (0.aa\ldots aa)_B \cdot B^{e_{\max}} = B^{e_{\max}}(1 - B^{-t})$$

mit $a = B - 1$ sowie die in $\mathbb{G}_{B,t,e_{\min},e_{\max}}$ *kleinste normalisierte positive Zahl*

$$R_{\min} = (0.100\ldots00)_B \cdot B^{e_{\min}} = B^{e_{\min}-1}.$$

Unter dem *Wertebereich* von $\mathbb{G}_{B,t,e_{\min},e_{\max}}$ versteht man dann die Menge

$$\{x \in \mathbb{R} \mid R_{\min} \leq |x| \leq R_{\max}\}. \tag{8.26}$$

Bereits auf Seite 15 haben wir die *relative Maschinengenauigkeit* **eps** als den Abstand zwischen 1 und der nächstgrößeren Zahl des Gleitkommasystems kennengelernt. Unter Verwendung der entsprechenden Gleitkommadarstellungen (8.22) folgt

$$\mathbf{eps} = (0.100\ldots01)_B \cdot B^1 - (0.100\ldots00)_B \cdot B^1 = B^{1-t}. \tag{8.27}$$

Da also die Mantissenlänge t (bei fixer Basis) unmittelbar Aufschluss über **eps** gibt, wird t auch als *Präzision* oder *Genauigkeit* von $\mathbb{G}_{B,t,e_{\min},e_{\max}}$ bezeichnet.

Abbildung II.1 veranschaulicht, dass die Lücken in einem Gleitkommasystem, also die Abstände zwischen den Zahlen, nicht einheitlich sind. Für $x \in \mathbb{G}_{B,t,e_{\min},e_{\max}}$ mit $x \neq 0$ und $x = v \cdot m \cdot B^e$ ist der Abstand zur betragsmäßig nächstgrößeren Gleitkommazahl in $\mathbb{G}_{B,t,e_{\min},e_{\max}}$ durch

$$(0.00\ldots01)_B \cdot B^e = \mathbf{eps} \cdot B^{e-1}$$

gegeben. Da aus (8.23) die Abschätzung $B^{e-1} \leq |x| < B^e$ folgt, ist der Abstand einer Zahl $x \in \mathbb{G}_{B,t,e_{\min},e_{\max}}$ zu ihren Nachbarn höchstens $\mathbf{eps}|x|$ und mindestens $B^{-1}\mathbf{eps}|x|$, es sei denn, bei x oder der Nachbarzahl handelt es sich um 0. Die Abstände

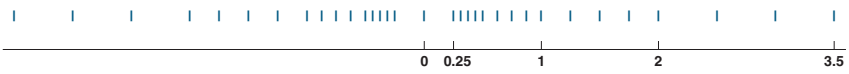


Abb. II.1. Das Gleitkommasystem (8.25).

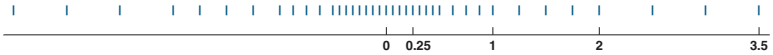


Abb. II.2. Das um subnormale Zahlen erweiterte Gleitkommasystem (8.25).

werden also mit $|x|$ größer, der Abstand von 0 zu den unmittelbaren Nachbarn ist genau $R_{\min}$. Die Lücke zwischen 0 und $R_{\min}$ bzw. $-R_{\min}$ kann gefüllt werden, wenn man bei Zahlen mit Exponenten $e_{\min}$ auch Mantissen mit führender Ziffer $z_1 = 0$ zulässt. Die so entstehenden Gleitkommazahlen nennt man *subnormal* oder auch *denormalisiert*. Der Abstand zwischen ihnen ist einheitlich gleich $B^{e_{\min}-t}$ und stimmt damit mit jenem der Zahlen zwischen $R_{\min}$ und $B \cdot R_{\min}$ überein. Insbesondere gilt für die in $\mathbb{G}_{B,t,e_{\min},e_{\max}}$ *kleinste positive subnormale Gleitkommazahl*

$$S_{\min} = (0.00\ldots01)_B \cdot B^{e_{\min}} = B^{e_{\min}-t} = R_{\min}\mathbf{eps}.$$

Beispiel

Durch Zulassung denormalisierter Mantissen bei $e = e_{\min} = -1$ kommen zum Gleitkommasystem (8.25) die subnormalen Zahlen $\pm 0.0625 = \pm(0.001)_2 \cdot 2^{-1}$, $\pm 0.125 = \pm(0.010)_2 \cdot 2^{-1}$ und $\pm 0.1875 = (0.011)_2 \cdot B^{-1}$ hinzu, siehe Abbildung II.2. Es gilt $S_{\min} = 0.0625$ und $R_{\min} = 2^{-2} = 0.25$.

Der IEEE Standard

Um die Gleitkommadarstellung (sowie die Arithmetik, siehe weiter unten) auf Computern zu vereinheitlichen, wurde 1985 der sogenannte IEEE-754 Standard[21] festgelegt, der inzwischen von nahezu allen Computerherstellern befolgt wird. Dieser definiert mit *Single* und *Double* zwei Hauptformate[22] für Gleitkommazahlen in Binärdarstellung, d.h. mit Basis $B = 2$, letzteres haben wir bereits in allen numerischen Beispielen aus Kapitel I verwendet. Beim Single-Format werden Gleitkommazahlen als 32-Bit Wörter abgespeichert. Dabei beschreibt das erste Bit das Vorzeichen ($0 = +$, $1 = -$). Die nachfolgenden 8 Bits dienen der Darstellung des verschobenen Exponenten $E \in \mathbb{N}_0$ mit einem Wertebereich von 0 bis 255. Der Wert $e \in \mathbb{Z}$ des Exponenten der darzustellenden Zahl ergibt sich dann für $1 \le E \le 254$ gemäß

$$e = E - 126, \text{ d.h. } e_{\min} = -125 \text{ und } e_{\max} = 128.$$

Die Werte $E = 0$ bzw. $E = 255$ werden für 0, -0 und die subnormalen Zahlen bzw. $\pm\infty$ und NaN (Not a Number) verwendet[23]. Die verbleibenden 23 Bits repräsentieren die Mantisse. Bei Basis $B = 2$ ist die führende Ziffer z_1 von normalisierten Zahlen stets gleich 1, bei subnormalen Zahlen stets gleich 0. Da zwischen diesen Zahlen aber bereits anhand des Exponenten E unterschieden werden kann, wird z_1 nicht mitabgespeichert. Man spricht vom *hidden bit* und erhält effektiv die Mantissenlänge

[21] IEEE = Institute of Electrical and Electronics Engineers.
[22] Daneben sind noch die Formate Single-Extended und Double-Extended standardisiert.
[23] Die Bedeutung von $\pm\infty$ und NaN wird auf Seite 90 kurz erläutert.

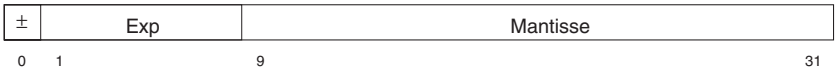


Abb. II.3. IEEE Single Zahl als 32-Bit Wort.

$t = 23 + 1$. Abbildung II.3 gibt den schematischen Aufbau einer Gleitkommazahl im Single-Format als 32-Bit Wort wider, Tabelle II.3 gibt einen Überblick über die IEEE-Single Zahlen.

Bitmuster im Exponenten E	dargestellte Gleitkommazahl
$(00000000)_2 = 0$	$\pm(0.0z_2 \dots z_{24})_2 \cdot 2^{-125}$
$(00000001)_2 = 1$	$\pm(0.1z_2 \dots z_{24})_2 \cdot 2^{-125}$
$\vdots$	$\vdots$
$(01111111)_2 = 127$	$\pm(0.1z_2 \dots z_{24})_2 \cdot 2^{1}$
$(10000000)_2 = 128$	$\pm(0.1z_2 \dots z_{24})_2 \cdot 2^{2}$
$\vdots$	$\vdots$
$(11111110)_2 = 254$	$\pm(0.1z_2 \dots z_{24})_2 \cdot 2^{128}$
$(11111111)_2 = 255$	$\pm\infty$ falls $z_2 = \dots = z_{24} = 0$ sonst NaN

Tabelle II.3. IEEE-Single Zahlen.

Beim IEEE-Double Format werden Gleitkommazahlen mit Hilfe zweier 32-Bit Wörter abgespeichert. Von den zur Verfügung stehenden 64 Bits werden 1 Bit für das Vorzeichen, 11 Bits für den Exponenten und 52 für die Mantisse verwendet, vergleiche auch mit Seite 15. Die charakteristischen Kenngrößen sind in Tabelle II.4 zusammengefasst. Aus **eps** $\approx 2.2 \cdot 10^{-16}$ ist ersichtlich, weshalb man im Zusammenhang mit IEEE double precision auch vom *Rechnen mit* 16 *Dezimalstellen* spricht.

Computerprogrammierung. *In C entspricht der Datentyp* `float` *dem IEEE Single Format, der Datentyp* `double` *entspricht dem IEEE Double Format. In* MATLAB *wird standardmäßig mit IEEE Double gearbeitet*[24]*, mit Hilfe des single Befehls lassen sich dort IEEE Single Zahlen erzeugen. Mathematica verwendet ein Modell erweiterter Genauigkeit, auf das wir erst auf Seite 91 kurz eingehen werden.*

Bemerkung. Die Formate Single und Double sind für Rechner entwickelt worden, deren Prozessoren auf eine Wortlänge $\omega = 32$ ausgerichtet[25] sind. Man vergleiche dazu auch die Integer Datentypen mit Wortlänge ω und 2ω.

Rundung

Liegt eine Eingangsgröße (oder ein Zwischenergebnis) $x \in \mathbb{R} \backslash \mathbb{G}_{B,t,e_{\min},e_{\max}}$ vor, so muss dafür ein Ersatz $\tilde{x} \in \mathbb{G}_{B,t,e_{\min},e_{\max}}$ bestimmt werden. Diesen Vorgang nennt man

[24] Voraussetzung dafür ist natürlich, dass der verwendete Computer dem IEEE Standard genügt.

[25] Ob es infolge des verstärkten Aufkommens von Rechnern mit 64-Bit Architektur zu einer Ablösung von Single und Double kommt, bleibt abzuwarten.

	IEEE Single	IEEE Double
$e_{\min}, e_{\max}$	$-125, 128$	$-1021, 1024$
eps	$2^{-23} \approx 1.2 \cdot 10^{-7}$	$2^{-52} \approx 2.2 \cdot 10^{-16}$
$R_{\max}$	$2^{128} \cdot (1 - 2^{-24}) \approx 3.4 \cdot 10^{38}$	$2^{1024} \cdot (1 - 2^{-53}) \approx 1.8 \cdot 10^{308}$
$R_{\min}$	$2^{-126} \approx 1.2 \cdot 10^{-38}$	$2^{-1022} \approx 2.2 \cdot 10^{-308}$
$S_{\min}$	$2^{-149} \approx 1.4 \cdot 10^{-45}$	$2^{-1074} \approx 4.9 \cdot 10^{-324}$

Tabelle II.4. Kenngrößen von IEEE Single und Double.

Rundung. Zu deren Erklärung führen wir die Menge

$$\bar{\mathbb{G}}_{B,t} := \{0\} \cup \left\{ x \in \mathbb{R} \;\middle|\; x \text{ ist normalisierte } t\text{-stellige Gleitkommazahl zur Basis } B \right\}$$

ein, bei der wir im Gegensatz zu (8.24) keine Einschränkungen an den Exponenten e treffen. Weiters seien zu $x \in \mathbb{R}$ mit x_-, x_+ die beiden nächstgelegenen Zahlen in $\bar{\mathbb{G}}_{B,t}$ kleiner bzw. größer als x bezeichnet. Die *Rundung zur nächstgelegenen Zahl in* $\bar{\mathbb{G}}_{B,t}$ ist dann definiert[26] durch

$$\begin{aligned} \mathrm{rd} : \mathbb{R} &\to \bar{\mathbb{G}}_{B,t} \\ x &\mapsto \text{jenes } \bar{x} \in \{x_-, x_+\}, \text{ das näher an } x \text{ liegt.} \end{aligned} \tag{8.28}$$

Für $x \in \bar{\mathbb{G}}_{B,t}$ gilt $\mathrm{rd}(x) = x$. Liegt x genau zwischen x_- und x_+, so gibt es mehrere Varianten in der Definition von $\mathrm{rd}(x)$. Bei der auch im IEEE Standard verwendeten *geraden Rundung* (engl.: round to even) wählt man jene Zahl in $\{x_-, x_+\}$, deren letzte Ziffer z_t gerade ist. Liegt der gerundete Wert $\mathrm{rd}(x)$ nicht im Gleitkommasystem $\mathbb{G}_{B,t,e_{\min},e_{\max}}$, so spricht man bei $|\mathrm{rd}(x)| > R_{\max}$ von einem *Exponentenüberlauf* (engl.: overflow), bei $|\mathrm{rd}(x)| < R_{\min}$ von einem *Exponentenunterlauf* (engl.: underflow).

Beispiel

Wir betrachten das Gleitkommasystem (8.25) und die Rundung rd nach (8.28). Zu $x = 0.6875 \notin \mathbb{G}_{2,3,-1,2}$ sind die beiden nächstgelegenen Zahlen durch $x_- = 0.625 = (0.101)_2 \cdot 2^0$ und $x_+ = 0.75 = (0.110)_2 \cdot 2^0$ gegeben. Die Abstände von x zu x_- und x_+ sind identisch, da die letzte Ziffer von x_+ gerade ist, folgt $\mathrm{rd}(0.6875) = 0.75$. Während $\mathrm{rd}(3.6) = 3.5 \in \mathbb{G}_{2,3,-1,2}$ gilt, erhält man bei Rundung von 3.75 einen Exponentenüberlauf, da $\mathrm{rd}(3.75) = 4.0 > R_{\max} = 3.5$ ist. Die Rundung von 0.20 führt zu einen Unterlauf, da $\mathrm{rd}(0.20) = 0.1875 < R_{\min} = 0.25$.

Bei der Eingabe von $x \in \mathbb{R}$ am Computer entsteht also ein Rundungsfehler, sobald x kein Element des verwendeten Gleitkommasystems $\mathbb{G}_{B,t,e_{\min},e_{\max}}$ ist. Ist x im Wertebereich $\mathbb{G}_{B,t,e_{\min},e_{\max}}$, siehe (8.26), so ist der Betrag des relativen Fehlers durch die auf Seite 16 eingeführte Rundungseinheit $\mathbf{u} = \frac{1}{2}\mathbf{eps}$ beschränkt.

Satz

Es sei $\mathbb{G}_{B,t,e_{\min},e_{\max}}$ ein Gleitkommasystem und $x \in \mathbb{R}$ mit $R_{\min} \le |x| \le R_{\max}$. Dann gilt

$$\mathrm{rd}(x) = x(1 + \varepsilon_x) \quad \text{mit } |\varepsilon_x| \le \mathbf{u}. \tag{8.29}$$

[26] Andere Rundungsmodi sind das Ab-, Aufrunden sowie das Runden zur oder von der Null.

Beweis. Es sei zunächst $x > 0$. Nach (8.20) existiert dann ein $e \in \mathbb{Z}$, so dass die Darstellung

$$x = \eta \cdot B^e \quad \text{mit } \eta \in \mathbb{R}^+ \text{ und } B^{-1} \leq \eta < 1$$

gilt. Weiters liegt x nach Annahme zwischen den benachbarten Maschinenzahlen

$$x_- = \lfloor \eta \cdot B^t \rfloor \cdot B^{e-t} \quad \text{und} \quad x_+ = \begin{cases} \lceil \eta \cdot B^t \rceil \cdot B^{e-t} & \text{für} \quad \eta \leq 1 - B^{-t}, \\ \lceil \eta \cdot B^t \rceil / B \cdot B^{e-t+1} & \text{für} \quad \eta > 1 - B^{-t}. \end{cases}$$

Daraus folgt

$$|\mathrm{rd}(x) - x| \leq \frac{|x_+ - x_-|}{2} \leq \frac{B^{e-t}}{2}$$

bzw.

$$\left| \frac{\mathrm{rd}(x) - x}{x} \right| \leq \frac{B^{e-t}}{2\eta B^e} \leq \frac{1}{2}\mathbf{eps} = \mathbf{u}.$$

Der Fall $x < 0$ wird analog behandelt. □

Gleitkommaarithmetik

Die elementaren arithmetischen Operationen $+, -, \cdot, /$ von t-stelligen Gleitkommazahlen x und y zur Basis B können als eine Folge von Operationen auf deren Mantissen und Exponenten realisiert werden. Entscheidend dabei ist, dass ein Gleitkommasystem $\mathbb{G}_{B,t,e_{\min},e_{\max}}$ bezüglich der Grundoperationen nicht abgeschlossen (selbst wenn die Einschränkung an den Exponenten aufgegeben wird) ist.

Beispiel

$x = 1.75 = (0.111)_2 \cdot 2^1$ und $\bar{x} = 0.625 = (0.101)_2 \cdot 2^0$ sind Elemente des Gleitkommasystems $\mathbb{G}_{2,3,-1,2}$ aus (8.25), die Summe $2.375 = (0.10011)_2 \cdot 2^2$ und das Produkt $1.09375 = (0.100011)_2 \cdot 2^1$ liegen jedoch nicht in $\mathbb{G}_{2,3,-1,2}$. Im IEEE-Single Format sind 1 und 2^{-24} beides Maschinenzahlen, die Summe $1 + 2^{-24}$ hingegen nicht.

Die grundlegende Idee zur Realisierung der Grundoperationen am Computer ist es daher, für zwei Maschinenzahlen x und y in $\mathbb{G}_{B,t,e_{\min},e_{\max}}$ zunächst das Ergebnis mit höherer Genauigkeit[27] zu berechnen und es anschließend auf eine Maschinenzahl in $\mathbb{G}_{B,t,e_{\min},e_{\max}}$ zu runden. Dabei verlangt man von den Grundoperationen üblicherweise, dass sie dem Modell der *korrekt gerundeten Gleitkommaarithmetik* genügen.

Definition

Korrekt gerundete Gleitkommaoperation. Seien $x, y \in \mathbb{G}_{B,t,e_{\min},e_{\max}}$. Für jede der Operationen $\diamond \in \{+, -, \cdot, /\}$ wird die entsprechende *korrekt gerundete Gleitkommaoperation* $\tilde{\diamond}$ durch

$$x \,\tilde{\diamond}\, y := \mathrm{rd}(x \diamond y) \tag{8.30}$$

definiert.

[27] Dabei können sogenannte Hilfsziffern zum Einsatz kommen. Wir gehen nicht näher darauf ein und verweisen auf [6].

Für alle $x, y \in \mathbb{G}_{B,t,e_{\min},e_{\max}}$ gilt daher, falls weder Exponentenunterlauf noch -überlauf eintritt,

$$x \tilde{\diamond} y = (x \diamond y)(1 + \varepsilon_{x,y,\diamond}) \text{ mit } |\varepsilon_{x,y,\diamond}| \leq \mathbf{u}, \tag{8.31}$$

siehe auch (8.29). Zusätzlich nimmt man an, dass auch der Rundungsfehler bei der Wurzeloperation durch $\mathbf{u}$ beschränkt ist. Die Beziehung (8.31) sagt aus, dass der am Computer tatsächlich berechnete Wert von $x \diamond y$ so gut ist, wie die Rundung des korrekten Resultats.

Die Forderung (8.31) lässt sich in entsprechenden Algorithmen für die Grundoperationen auch tatsächlich erfüllen. Exemplarisch betrachten wir das Grundprinzip der Addition zweier Zahlen x und $y \in \mathbb{G}_{B,t,e_{\min},e_{\max}}$ mit $x \geq y > 0$. Zusätzlich nehmen wir an, dass kein Überlauf auftritt, also $x + y \leq R_{\max}$. Wir verwenden $MantExp(x)$, um das Paar (m_x, e_x) aus Mantisse und Exponent einer Zahl $x = m_x \cdot B^{e_x}$ anzusprechen. Die Addition zweier Gleitkommazahlen unterteilt sich dann in eine Exponentenanpassung bei y, eine Mantissenaddition sowie Normalisierung und Rundung des Ergebnisses, siehe Algorithmus *AddG*.

Algorithmus *AddG*: Addition von Gleitkommazahlen

$(m_x, e_x) \leftarrow MantExp(x)$
$(m_y, e_y) \leftarrow MantExp(y)$
$d \leftarrow e_x - e_y$
$m'_y \leftarrow m_y \cdot B^{-d}$
$m'_z \leftarrow m_x + m'_y$
$e'_z \leftarrow e_x$
if $m'_z \geq 1$
  $m'_z \leftarrow m'_z \cdot B^{-1}$
  $e'_z \leftarrow e'_z + 1$
$(m_z, e_z) \leftarrow Runden(m'_z, e'_z, B, t)$
$z \leftarrow m_z \cdot B^{e_z}$
return z

Aufruf: $AddG(x,y)$
Eingabe: $x, y \in \mathbb{G}_{B,t,e_{\min},e_{\max}}$
mit: $x \geq y > 0$ und $x + y \leq R_{\max}$
Ausgabe: $z \in \mathbb{G}_{B,t,e_{\min},e_{\max}}$
mit: $\frac{z-(x+y)}{x+y} \leq \mathbf{u}$.

Darin steht *Runden* für eine algorithmische Umsetzung der Rundungsabbildung (8.28), die sicher stellt, dass $\mathrm{rd}(m'_z \cdot B^{e'_z}) = m_z \cdot B^{e_z}$ gilt. Dabei ist m'_z die normalisierte Mantisse mit (endlicher) Länge $t' \geq t$ des Zwischenergebnisses erhöhter Genauigkeit. Tatsächlich werden die Teilschritte ähnlich den klassischen Algorithmen aus Abschnitt 5 als Operationen auf den Zifferntupeln der Mantissen und Exponenten ausgeführt, im folgenden Beispiel veranschaulichen wir dies der Einfachheit halber nur für die Mantisse.

Beispiel

Zu $\mathbb{G}_{2,3,-1,2}$ betrachten wir die Addition nach *AddG* von $x = 1.75 = (0.111)_2 \cdot 2^1$ und $y = 0.625 = (0.101)_2 \cdot 2^0$. Wegen $e_x = 1, e_y = 0$ ist $d = 1$, daher muss $m_y \cdot B$ berechnet werden, was durch Verschieben der Mantisse $m_y = (0.101)_2$ um eine Stelle nach rechts leicht erreicht werden kann, also $m'_y = (0.0101)_2$. Die anschließende Addition von $m_x = (0.111)_2$ mit m'_y führt auf $m'_z = (1.0011)_2$, wobei es sich um keine normalisierte Mantisse der Gestalt (8.22) handelt. Zur Normalisierung muss daher m_z noch um eine Stelle nach rechts auf $m'_z = (0.10011)_2$ verschoben werden, und der Exponent $e'_z = e_x = 1$ um eins auf $e'_z = 2$ erhöht

werden. Das normalisierte Zwischenergebnis lautet $z = (0.10011)_2 \cdot 2^2 = 2.375$ und muss abschließend auf $z = (0.101)_2 \cdot 2^2 = 2.5 \in \mathbb{G}_{2,3,-1,2}$ gerundet werden. Für den durch Aufruf von $AddG(x,y)$ entstehenden Rundungsfehler gilt

$$\left|\frac{2.5 - 2.375}{2.375}\right| = 0.0526 \leq \mathbf{u} = 0.125.$$

Für nähere algorithmische Ausführungen der Gleitkommaoperationen verweisen wir auf [6], erwähnen aber, dass der Aufwand der Gleitkommaaddition $O(t + r)$ Ziffernoperationen beträgt, wenn t Bits für die Mantisse und r Bits für den Exponenten verwendet werden. Der Aufwand der Gleitkommamultiplikation ist dann durch $O(t^2 + r)$ Ziffernoperationen gegeben. Ähnlich wie bei den Zahlen fixer Länge hängt der Aufwand nicht von den eigentlichen Operanden ab, sondern von den physikalischen Parametern des Prozessors, die t und r bestimmen.

Zu beachten ist, dass die klassischen Gesetze der Arithmetik für die Gleitkommaoperationen $\tilde{\diamond}$ nicht mehr unbedingt gelten. Inbesondere können das Assoziativ- und das Distributivgesetz verletzt sein.

Beispiel

Wir betrachten erneut das Gleitkommasystem (8.25). Für 0.25, 0.4375 und $2.0 \in \mathbb{G}_{2,3,-1,2}$ gilt

$$\begin{aligned}(0.4375 \tilde{+} 0.25) \tilde{+} 2.0 &= \mathrm{rd}(0.6875) \tilde{+} 2.0 \\ &= \mathrm{rd}(2.75) = 3.0,\end{aligned}$$

aber

$$\begin{aligned}0.4375 \tilde{+} (0.25 \tilde{+} 2.0) &= 0.4375 \tilde{+} 2.0 \\ &= \mathrm{rd}(2.4375) = 2.5.\end{aligned}$$

Das skizzierte Modell der *korrekt gerundeten Gleitkommarithmetik* ist insbesondere auch im IEEE Standard umgesetzt. Dort ist für einen Exponentenüberlauf, also $|x \tilde{\diamond} y| > R_{\max}$, als Standardresultat $\pm\infty$ festgelegt, was man auch bei Division von $x \neq 0$ durch 0 erhält. Bei Ausführung ungültiger Operationen wie etwa der Multiplikation von 0 mit ∞ liefert IEEE als Ergebnis NaN.

Auch Exponentenunterlauf und -überlauf stellen also eine mögliche Fehlerquelle beim Rechnen mit Gleitkommazahlen dar. Diese kann durch geeignete *Skalierung* vermieden werden. Die Ideen des folgenden Beispiels können zu einer allgemeinen Skalierungsstrategie hochgezogen werden, die Details dazu sind allerdings recht anspruchsvoll.

Beispiel

Die kleinere reelle Lösung von

$$x^2 - 2 \cdot 2 \cdot x + 3 = 0 \tag{8.32}$$

ist 1, liegt also insbesondere im Wertebereich des Gleitkommasystems (8.25). Rufen wir jedoch unter dessen Verwendung den Algorithmus *QuadGlgI* von

Seite 5 mit $p = 2$ und $q = 3$ auf, so erhalten wir bereits im ersten Schritt wegen $p^2 > R_{\max}$ einen Exponentenüberlauf. Dieser kann durch die Skalierung $\bar{x} = 0.5x$ vermieden werden. Anstelle von (8.32) erhalten wir damit die Gleichung

$$4\bar{x}^2 - 8\bar{x} + 3 = 0 \quad \text{bzw.} \quad \bar{x}^2 - 2 \cdot 1 \cdot \bar{x} + 0.75 = 0.$$

Die Ausführung von *QuadGlgI* unter Verwendung von $\mathbb{G}_{B,t,e_{\min},e_{\max}}$ mit $p = 1$ und $q = 0.75$ verläuft nun ohne Überlauf und liefert das exakte[28] Resultat $\bar{x} = 0.5$. Die Rückskalierung $x = 2 \cdot \bar{x}$ führt dann auf die Lösung $x = 1$ von (8.32). Ähnliche Beispiele lassen sich natürlich auch bei Verwendung von IEEE single oder double leicht anführen, jedes p eines Gleitkommasystems führt ja bei $|p| > \sqrt{R_{\max}}$ in *QuadGlgI* zu einem Überlauf.

Exponentenunterlauf und -überlauf werden bei der Stabilitätsanalyse eines Algorithmus in der Regel nicht mitberücksichtigt. Wie dann der bei dessen Gleitkommarealisierung entstehende Rundungsfehler mit Hilfe der Rundungseinheit **u**, also der Fehlerschranke in (8.29) und (8.31), abgeschätzt werden kann, haben wir bereits in Kapitel I ausführlich diskutiert.

Rechnen mit erhöhter und unendlicher Genauigkeit

Der Einfluss von Rundungsfehlern bei der Gleitkommarealisierung eines Algorithmus kann reduziert werden, indem man die Rechengenauigkeit durch Verlängerung der Mantisse erhöht, siehe (8.27). Liegt für den Algorithmus eine Fehlerschranke vor, die proportional zur Rundungseinheit **u** ist, können vorgegebene Genauigkeiten durch eine hinreichende Verlängerung der Mantisse erzwungen werden. In Abhängigkeit der zur Verfügung stehenden Rechnerarchitektur kann dies etwa durch einen Wechsel von IEEE single mit $t = 23$ auf IEEE double precision mit $t = 53$ oder auf IEEE extended precision mit $t = 64$ geschehen. Findet man damit nicht das Auslangen, kann man – ähnlich wie bei den ganzen Zahlen beliebiger Länge – die durch die Hardware bedingte Genauigkeitsschranke mit Hilfe von Software durchbrechen, siehe dazu auch [7] für Details. In der MATLAB Symbolic Toolbox kann man mit Hilfe des Befehls `digits(d)` die Anzahl `d` an Dezimalstellen festlegen, mit der dann unter Verwendung von *variable precision arithmetic* gerechnet wird. Erhöht man `d`, so erhöht man auch die Rechengenauigkeit, allerdings zu Lasten von Geschwindigkeit und Speicherplatzbedarf. Dagegen kann *Mathematica* mit Hilfe des Befehls `N[expr,d]` dazu aufgefordert werden, den numerischen Wert des Ausdrucks `expr` mit `d` korrekten Dezimalziffern zu liefern[29]. Dabei muss *Mathematica* aber unter Umständen Zwischenresultate mit deutlich höherer Genauigkeit berechnen, über den Systemparameter `$MaxExtraPrecision` wird festgelegt, wieviele zusätzliche Stellen bei der Berechnung von Zwischenergebnissen zum Erreichen der geforderten Genauigkeit herangezogen werden dürfen. Der Standardwert für `$MaxExtraPrecision` ist 50, er kann aber auch auf ∞ gesetzt werden. Auch in diesem Zusammenhang spricht man von *variable precision arithmetic (vpa)*.

[28] Das Beispiel wurde so gewählt, dass es zu keinen Rundungsfehlern kommt.
[29] Eine Garantie dafür gibt es allerdings nicht.

Zur Veranschaulichung ziehen wir ein Beispiel heran, bei dem die Fehler infolge endlicher Genauigkeit deutlich erkennbar sind. Zur Berechnung einer Näherungslösung für das Problem der Kreisumfangsbestimmung von Seite 19 haben wir auf Seite 22 den Algorithmus *VieleckUmfang* vorgeschlagen. Dessen Unteralgorithmus *SeitenlängeR* zur Bestimmung der Seitenlänge l_n des $6 \cdot 2^n$-Ecks basiert dabei auf der rekursiven Beziehung (2.30) zwischen l_n und l_{n-1}. Die dazu äquivalente Beziehung (2.29) legt somit den alternativen Algorithmus *SeitenlängeRII* nahe.

Algorithmus *SeitenlängeRII*: Rekursive Berechnung der Seitenlänge des $6 \cdot 2^n$-Ecks, Variante II

if $n = 0$
  $l \leftarrow 1/2$
else
  $l \leftarrow$ *SeitenlängeRII*$(n-1)$
  $l \leftarrow \sqrt{\frac{1}{2} - \frac{1}{2}\sqrt{1-l^2}}$
return l

Aufruf: *SeitenlängeRII*(n)
Eingabe: $n \in \mathbb{N}_0$
Ausgabe: $l \in \mathbb{R}^+$
mit: $l = l_n$.

Beispiel

Approximation von π. Wir wiederholen in MATLAB, also IEEE double precision, das Beispiel von Seite 23, verwenden in *VieleckUmfang* nun jedoch den Unteralgorithmus *SeitenlängeRII*. Der relative Fehler

$$\frac{|\texttt{VieleckUmfang}(n) - \texttt{pi}|}{|\texttt{pi}|}, \tag{8.33}$$

wobei *pi* wieder den MATLAB-Wert (2.31) für π bezeichnet, ist in Abbildung II.4 in Abhängigkeit von n dargestellt. Sie zeigt, dass mit wachsendem n der Fehler (8.33) nur bis $n = 12$ fällt und danach ansteigt, bis er ab $n = 28$ konstant 1 bleibt. Insbesondere bedeutet dies, dass man für alle $n \geq 28$ den Wert 0 als Ergebnis, also als Näherung von π, erhält. Ursache für dieses völlig fehlerhafte Resultat ist die durch die Auslöschung in (2.29) bedingte Fortpflanzung der Rundungsfehler.

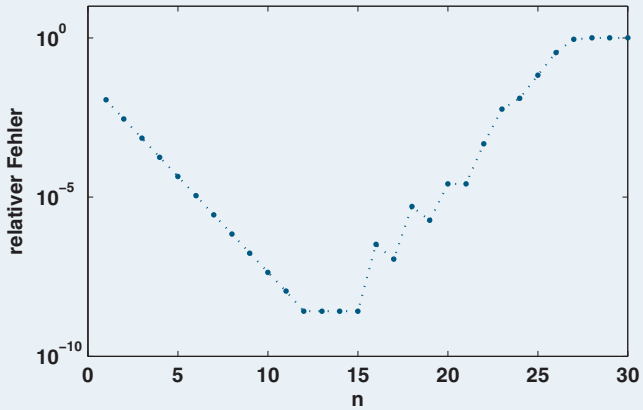


Abb. II.4. Der relative Fehler (8.33) in Abhängigkeit von n.

In *Mathematica* führt der Aufruf `VieleckUmfang[n]` (hier erneut unter Verwendung von `SeitenlängeRII`) zunächst auf einen exakten, aber beinahe unlesbaren verschachtelten Wurzelausdruck für den Vielecksumfang U_n. Mit `N[VieleckUmfang[n],d]` wird *Mathematica* aufgefordert, den numerischen Wert des Wurzelausdrucks auf d korrekte Dezimalstellen zu berechnen. Möchten wir also U_{120} mit 30 korrekten Dezimalstellen bestimmen, so rufen wir `N[VieleckUmfang[120],30]` auf. Allerdings erhalten wir damit zunächst nur das Resultat 3.141593 zusammen mit dem Hinweis, dass die für die Zwischenresultate zur Verfügung stehende Rechengenauigkeit nicht ausreicht, um 30 korrekte Resultatsstellen zu sichern. Setzen wir `$MaxExtraPrecision` $= \infty$, so führt der erneute Aufruf von `N[VieleckUmfang[120],30]` dann auf 3.14159265358979323846264338328. Dies ist auch jenes Ergebnis, dass man bei Aufruf von `N[Pi,30]` erhält. Der Diskretisierungsfehler $|U_n - \pi|$ bleibt aber natürlich beim Rechnen mit erhöhter Genauigkeit erhalten. So führt `N[VieleckUmfang[10],30]` auf 3.14159251669215744759287408477.

Summation

Der Summation reeller Zahlen $x_1, \ldots, x_n$ begegnet man in der numerischen Mathematik[30] an den unterschiedlichsten Stellen, man denke etwa an die Berechnung von Mittelwerten, der Auswertung nichtlinearen Funktionen wie sin, cos oder $\sqrt{}$, vgl. mit Seite 11, und der numerischen Berechnung bestimmter Integrale. Obwohl man hinter der Summation zunächst wenig algorithmisch Interessantes vermuten mag, gibt es doch eine Reihe unterschiedlicher Summationstechniken, für einen Überblick verweisen wir auf [6].

Problemstellung (Summation).

Gegeben: $x \in \mathbb{R}^n$
Gesucht: $s \in \mathbb{R}$
mit: $s = \sum_{i=1}^{n} x_i$

Die dazu passende Abbildung von Seite 3, die den Eingangsdaten[31] x die eindeutige Lösung zuordnet, ist natürlich durch

$$\varphi^n : \mathbb{R}^n \to \mathbb{R},\ x \mapsto \sum_{i=1}^{n} x_i \tag{8.34}$$

gegeben[32]. Der folgende Satz gibt Auskunft über die relative komponentenweise Kondition des Summationsproblems, vergleiche auch mit Seite 32.

[30] Summationsaufgaben gibt es auch im Bereich der symbolischen Mathematik. Dort sucht man beispielsweise für den Summenausdruck $\sum_{i=1}^{n} i^2$ nach einer geschlossenen Formel für den Summenwert in Abhängigkeit von n, allerdings ohne Verwendung des $\sum$-Zeichens. Durch den Einsatz symbolischer Summationsalgorithmen erhält man im Beispiel die Lösung $\frac{1}{6}n(1+n)(1+2n)$.

[31] n ist in dieser Spezifikation keine eigene Eingabevariable sondern eine Größe, die aus der Eingangsvariable x ermittelt werden kann.

[32] Die Schreibweise φ^n ist reine Notation zur Betonung der Abhängigkeit von n.

Satz

Kondition der Summation. Seien φ^n wie in (8.34) und $x \in \mathbb{R}^n$ mit $x_i \neq 0$, $i = 1, \ldots, n$, und $\varphi^n(x) \neq 0$ gegeben. Dann ist

$$\kappa_{\text{rel,komp}} = \frac{\sum_{i=1}^{n} |x_i|}{|\sum_{i=1}^{n} x_i|} \tag{8.35}$$

die relative komponentenweise Kondition des Problems (φ^n, x), wir schreiben dafür κ_{φ^n}.

Beweis. Die Abbildung (8.34) ist partiell differenzierbar, und es gilt $\frac{\partial \varphi^n}{\partial x_j}(x) = 1$ für $j = 1, \ldots, n$. Somit folgt nach (3.54), siehe Seite 33,

$$\kappa_{\varphi^n} = \frac{\sum_{j=1}^{n} |\frac{\partial \varphi^n}{\partial x_j}(x)||x_j|}{|\sum_{i=1}^{n} x_i|} = \frac{\sum_{j=1}^{n} |x_j|}{|\sum_{i=1}^{n} x_i|}.$$

□

Die Kondition der Summation ist groß, wenn in (8.35) der Nenner klein im Verhältnis zum Zähler ist. Dies ist etwa dann der Fall, wenn die Komponenten x_i von ungefähr gleicher Größenordnung sind, aber alternierendes Vorzeichen[33] haben. Für (8.35) gilt jedenfalls

$$\kappa_{\varphi^n} \geq 1. \tag{8.36}$$

Ein naheliegendes Verfahren zur Lösung des Summationsproblems besteht natürlich darin, die gegebenen Zahlen in einer Schleife aufzusummieren[34], siehe Algorithmus *SummeS*. Dabei handelt es sich um eine Schleifenumsetzung der rekursiven Beziehung

$$\varphi^2 = a^2 \quad \text{und}$$
$$\varphi^n = \varphi^{n-1} \circ a^n \quad \text{für } n > 2,$$

Algorithmus *SummeS*: Summation in $\mathbb{R}$

```
s ← 0
for i from 1 to n
    s ← s + x_i
return s
```

Aufruf: *SummeS*(x)
Eingabe: $x \in \mathbb{R}^n$.
Ausgabe: $s \in \mathbb{R}$
mit: $s = \sum_{i=1}^{n} x_i$.

[33] Siehe dazu auch die Diskussion des Spezialfalls $n = 2$ auf Seite 32.

[34] In einer Realisierung des Algorithmus muss n aus der Eingabe x ermittelt werden, im Pseudocode betrachten wir es aufgrund der Eingabebedingung $x \in \mathbb{R}^n$ als gegeben.

wobei

$$a^n : \mathbb{R}^n \to \mathbb{R}^{n-1},\ (x_1,x_2,\ldots,x_n) \mapsto (x_1 + x_2,\ldots,x_n) \tag{8.37}$$

der Addition der ersten beiden Komponenten entspricht. Insbesondere gilt damit die Zerlegung

$$\begin{aligned} \varphi^n &= \varphi^{n-1} \circ a^n \\ &= \varphi^{n-2} \circ a^{n-1} \circ a^n \\ &= a^2 \circ a^3 \circ \cdots \circ a^{n-1} \circ a^n. \end{aligned} \tag{8.38}$$

für die Abbildung (8.34). Im ersten Reduktionsschritt betrachtet man also anstelle von (φ,x) die Teilprobleme (a^n,x) und $(\varphi^{n-1},a^n(x))$. Die komponentenweise Kondition des Teilproblems $(\varphi^{n-1},a^n(x))$ ist wegen (8.35) durch

$$\kappa_{\varphi^{n-1}} = \frac{|x_1 + x_2| + \sum_{j=3}^{n} |x_j|}{|\sum_{i=1}^{n} x_i|}$$

gegeben. Zudem gilt – bei fixem $x \in \mathbb{R}^n$ – die Ungleichung

$$\kappa_{\varphi^{n-1}} \leq \kappa_{\varphi^n}. \tag{8.39}$$

Damit sind wir nun in der Lage, nach [3] die komponentenweise Vorwärtsstabilität (4.65) der Gleitkommarealisierung von *SummeS* zu beweisen.

Satz

Seien $x \in \mathbb{R}^n, x_j \neq 0$ für $j = 1,\ldots,n$ und $\varphi^n(\tilde{x}) \neq 0$ für alle $\tilde{x}$ mit $\frac{|\tilde{x}_j - x_j|}{|x_j|} \leq \mathbf{u}$ für $j = 1,\ldots,n$. Die Gleitkommarealisierung des Algorithmus *SummeS* ist komponentenweise vorwärtsstabil mit einer Stabilitätszahl

$$C_{\varphi^n} \leq n - 1. \tag{8.40}$$

Beweis. Wegen (8.38) entspricht die Gleitkommarealisierung $\tilde{\varphi}$ des Algorithmus *SummeS* der Hintereinanderausführung

$$\tilde{\varphi}^n = \tilde{\varphi}^{n-1} \circ \tilde{a}^n = \tilde{a}^2 \circ \tilde{a}^3 \circ \cdots \circ \tilde{a}^{n-1} \circ \tilde{a}^n. \tag{8.41}$$

Darin sind $\tilde{\varphi}^n$ und $\tilde{a}^n$ die Gleitkommarealisierungen der Abbildungen φ^n und a^n, deren Stabilitätszahlen, siehe Seite 43, bezeichnen wir mit C_{φ^n} und C_{a^n}.

Nach dem Lemma von Seite 40 gilt mit obigen Bezeichnungen für den ersten Reduktionsschritt die Abschätzung

$$C_{\varphi^n}\kappa_{\varphi^n} \leq C_{\varphi^{n-1}}\kappa_{\varphi^{n-1}} + C_{a^n}\kappa_{a^n}\kappa_{\varphi^{n-1}}. \tag{8.42}$$

Darin ist

$$C_{a^n}\kappa_{a^n} \leq 1, \tag{8.43}$$

da $\tilde{a}^n$ wegen (8.31) die Abschätzung[35]

$$\max_{i=1,\dots,n-1} \frac{|\tilde{a}_i^n(\tilde{x}) - a_i^n(\tilde{x})|}{|a_i^n(\tilde{x})|} \leq \mathbf{u}$$

erfüllt, siehe auch (4.65). Zusammen mit (8.39) folgt aus (8.42) dann $C_{\varphi^n}\kappa_{\varphi^n} \leq (C_{\varphi^{n-1}} + 1)\kappa_{\varphi^n}$ bzw.

$$C_{\varphi^n} \leq 1 + C_{\varphi^{n-1}}.$$

Durch wiederholte Anwendung dieses Arguments sieht man basierend auf (8.41) leicht, dass $C_{\varphi^n} \leq n - 2 + C_{a^2}$ gilt. Aus (8.43) folgt $C_{a^2} \leq 1/\kappa_{a^2}$. Wegen (8.36) ist dann aber $C_{a^2} \leq 1$ und damit schließlich (8.40). Die Stabilitätszahl C_{φ^n} ist also kleiner gleich der Anzahl $n - 1$ der Elementaroperationen von *SummeS*, somit ist dessen Gleitkommarealisierung stabil im Sinne der Vorwärtsanalyse. □

Der Faktor $C_{\varphi^n}\kappa_{\varphi^n}$, mit dem der maximale Rundungsfehler $\mathbf{u}$ der Eingabe bei der Gleitkommarealisierung von *SummeS* verstärkt werden kann, ist also durch das Produkt von $(n - 1)$ mit κ_{φ^n} aus (8.35) nach oben beschränkt. Weitere Informationen erhalten wir bei einer genaueren Betrachtung des Fehlers

$$\tilde{\varphi}^n(\tilde{x}) - \varphi^n(\tilde{x}), \tag{8.44}$$

der durch die Gleitkommarealisierung von *SummeS* verursacht wird, vergleiche mit (4.62). Darin bezeichnet $\tilde{x}$ wieder die gerundeten Eingangsdaten mit (3.47).

Satz

Seien $x \in \mathbb{R}^n, x_j \neq 0$ für $j = 1, \dots, n$ und $\varphi^n(\tilde{x}) \neq 0$ für alle $\tilde{x}$ mit $\frac{|\tilde{x}_j - x_j|}{|x_j|} \leq \mathbf{u}$ für $j = 1, \dots, n$. Der relative Gesamtrundungsfehler bei der Gleitkommarealisierung des Algorithmus *SummeS* genügt der Abschätzung

$$\frac{|\tilde{\varphi}^n(\tilde{x}) - \varphi(\tilde{x})|}{|\varphi^n(\tilde{x})|} \lesssim \frac{\mathbf{u}}{|\varphi^n(\tilde{x})|} \sum_{i=2}^{n} \sum_{j=1}^{i} |\tilde{x}_j| \quad \text{für } \mathbf{u} \to 0. \tag{8.45}$$

Beweis. Sei

$$s^{(i)} = \sum_{j=1}^{i} \tilde{x}_j \tag{8.46}$$

der Wert der Variable s, den man bei exakter Addition in Algorithmus *SummeS* am Ende des Schleifendurchlaufs mit dem Index i erhalten würde. Den tatsächlichen Wert, der infolge der Rundungsfehler bei der Gleitkommaaddition von $s^{(i)}$ abweichen kann, bezeichnen wir im Folgenden mit $\tilde{s}^{(i)}$. Insdesondere gilt damit $\tilde{\varphi}^n(\tilde{x}) = \tilde{s}^{(n)}$ und $\varphi^n(\tilde{x}) = s^{(n)}$, weiters setzen wir $\tilde{s}^{(1)} = \tilde{x}_1$. Für $i = 2, \dots, n$ sei nun ε_i mit $|\varepsilon_i| \leq \mathbf{u}$ der bei der Gleitkommaaddition von $\tilde{s}^{(i-1)}$ und $\tilde{x}_i$ entstehende Rundungsfehler, so dass nach (8.31)

$$\tilde{s}^{(i-1)} \mathbin{\tilde{+}} \tilde{x}_i = (\tilde{s}^{(i-1)} + \tilde{x}_i)(1 + \varepsilon_i) \quad \text{für } i = 2, \dots, n$$

[35] Wir nehmen an, dass $a_i^n(\tilde{x}) \neq 0$ für $i = 1, \dots n - 1$ gilt.

gilt. Damit ist

$$\tilde{s}^{(i)} = \tilde{s}^{(i-1)} + \tilde{x}_i + (\tilde{s}^{(i-1)} + \tilde{x}_i)\varepsilon_i,$$

woraus mit $\tilde{x}_i = s^{(i)} - s^{(i-1)}$ nach (8.46) die rekursive Darstellung

$$\tilde{s}^{(i)} - s^{(i)} = \tilde{s}^{(i-1)} - s^{(i-1)} + \varepsilon_i \tilde{s}^{(i)} + \varepsilon_i \left(\tilde{s}^{(i-1)} - s^{(i-1)}\right) \tag{8.47}$$

des Fehlers im i-ten Zwischenresultat folgt. Wegen $\varepsilon_i \tilde{s}^{(i-1)} \approx \varepsilon_i s^{(i-1)}$ für $\mathbf{u} \to 0$, vergleiche mit Übungsaufgabe I.6, erhält man durch Aufsummieren für i von 2 bis n auf der rechten und linken Seite von (8.47) für den absoluten Fehler (8.44) dann die Abschätzung

$$|\tilde{\varphi}^n(\tilde{x}) - \varphi^n(\tilde{x})| \lesssim \mathbf{u} \sum_{i=2}^{n} |s^{(i)}| \quad \text{für } \mathbf{u} \to 0.$$

Zusammen mit $\sum_{i=2}^{n} |s^{(i)}| \le \sum_{i=2}^{n} \sum_{j=1}^{i} |\tilde{x}_j|$ folgt daraus (8.45). □

Mit

$$\sum_{i=2}^{n} \sum_{j=1}^{i} |\tilde{x}_j| = (n-1) \cdot |\tilde{x}_1| + \sum_{j=2}^{n} (n+1-j)|\tilde{x}_j| \tag{8.48}$$

wird deutlich, dass die obere Schranke in (8.45) für den Rundungsfehler am kleinsten ist, wenn man die Eingangsdaten $\tilde{x}_j$ betragsmäßig der Grösse nach aufsteigend vorsortiert.

Beispiel

Zur Approximation der Reihe

$$\sum_{i=1}^{\infty} \frac{1}{i^2} = \frac{\pi^2}{6} = 1.64493406684822\ldots$$

rufen wir für verschiedene Werte von n den Algorithmus `SummeS` in MATLAB mit dem absteigend sortierten Vektor $x \in \mathbb{R}^n$ mit

$$x_i = \frac{1}{i^2} \quad \text{für } i = 1, \ldots n \tag{8.49}$$

als Input auf. Mit IEEE single precision erhalten wir für $n = 4096$ den Wert 1.6447253, den man auch für alle $n > 4096$ als Resultat bekommt. Grund dafür ist, dass die immer kleiner werdenden Komponenten $x_i = \frac{1}{i^2}$ ab $i = 4096$ infolge der Gleitkommaarithmetik nichts mehr zur Summe beitragen können. Wählt man stattdessen den aufsteigend sortierten Vektor $y \in \mathbb{R}^n$ mit

$$y_i = \frac{1}{(n+1-i)^2} \quad \text{für } i = 1, \ldots n$$

als Eingangsgröße, so liefert mit $n \ge 31663069$ der Aufruf `SummeS(y)` das Ergebnis 1.6449341. Dieses stimmt mit jenem überein, das man durch Rundung bei der Eingabe von $\pi^2/6$ erhält.

Eine Vorsortierung geht dabei natürlich zu Lasten der Komplexität des Summationsalgorithmus. Außerdem ist zu beachten, dass eine Vorsortierung lediglich die obere Schranke für den Rundungsfehler in (8.45) reduziert. Es kann also durchaus auch Situationen geben, in denen der tatsächliche Rundungsfehler ohne Vorsortierung kleiner ist als mit, siehe etwa das Beispiel von Seite 90. Anhaltspunkte für den Entwurf numerisch stabiler Algorithmen kann man aber nur aus Aussagen gewinnen, die den unbekannten Rundungsfehler einer Gleitkommaoperation betragsmäßig durch die bekannte Rundungseinheit **u** abschätzen.

Schätzt man (8.48) weiter nach oben durch $(n-1)\sum_{j=1}^{n}|\tilde{x}_j|$, so führt dies letztlich wieder auf die Vorwärtstabilitätsaussage (8.40), siehe Übungsaufgabe II.12.

Bemerkung. In obigem Beispiel waren alle Summanden Elemente in $\mathbb{Q}$. In diesem Fall kann man die Summationsaufgabe auch lösen, indem anstelle von Gleitkommaarithmetik rationale Arithmetik, etwa unter Nutzung der Additionsalgorithmen über $\mathbb{Q}$ aus Abschnitt 7, verwendet wird. Der Preis, der dabei für die Vermeidung von Rundungsfehlern zu zahlen ist, ist der vergleichsweise hohe Rechenaufwand.

Übungsaufgaben

II.1 Entwickeln Sie analog zu *AddZ* einen Subtraktionsalgorithmus auf $\mathcal{Z}_B$.

II.2 Zeigen Sie, dass die im Algorithmus *MultZ* benötigten Ziffernoperationen auf Zahlen der Länge 2ω ohne Überlauf ausgeführt werden können.

II.3 Erweitern Sie den Algorithmus *MultZ* so, dass er für beliebige Eingaben $a,b \in \mathcal{Z}_B$ anwendbar ist, und implementieren Sie ihn in einer Programmiersprache Ihrer Wahl.

II.4 Zeigen Sie, dass für beliebige Basis $B \geq 2$ und $b = \mathit{Verschiebe}_{\mathcal{Z}_B}(a,n)$ wie auf Seite 61 $b = a \cdot B^n$ gilt.

II.5 Implementieren Sie eine Methode, die die Aufspaltung der Eingabetupel im Karatsuba Algorithmus *ohne explizite Berechnung der Länge der Tupel* realisiert.

II.6 Vergleichen Sie das Laufzeitverhalten der Algorithmen *MultZ* und *MultZKaratsuba*.

II.7 Wandeln Sie den rekursiven Algorithmus *GGTZEuklid* in einen Schleifenalgorithmus um.

II.8 Zeigen Sie, dass $\equiv_m$ eine Kongruenzrelation auf $\mathbb{Z}$ ist.

II.9 Analysieren Sie den Ablauf des rekursiven Algorithmus *CRAZ* und wandeln Sie diesen dann in einen Schleifenalgorithmus um.

II.10 Beweisen Sie, dass die Relation $\sim$ von Seite 76 eine Äquivalenzrelation auf $\mathbb{Z} \times (\mathbb{Z} \setminus \{0\})$ ist.

II.11 Zeigen Sie, dass die Addition und Multiplikation in $\mathbb{Q}$ von Seite 76 von der Wahl der konkreten Repräsentanten unabhängig sind.

II.12 Zeigen Sie, dass die Stabilitätsaussage des Satzes von Seite 96 die komponentenweise Vorwärtsstabilität der Gleitkommarealisierung des Algorithmus *SummeS* impliziert.

II.13 Zeigen Sie für $n = 4$, dass die Gleitkommarealisierung des Algorithmus *SummeS* komponentenweise rückwärtsstabil ist.

II.14 Bestimmen Sie die komponentenweise Kondition des Problems (a^n, x) mit a^n aus (8.37).

II.15 Machen Sie sich durch Betrachtung der Konditionszahlen des Unterschieds der beiden Probleme $(\varphi^{n-1} \circ a^n, x)$ und $(\varphi^{n-1}, a^n(x))$ bewusst.

III Vektoren

Ein Vektorraum (über einem Körper K) ist eine mathematische Struktur, deren Objekte – die Vektoren – addiert und mit Elementen aus K multipliziert werden können. Die aus der Schule bekannten Vektoren mit zwei oder drei Koordinaten liefern geometrisch anschauliche Beispiele für Vektorräume (über $\mathbb{R}$), die Elemente eines Vektorraumes können aber auch viel abstrakterer Natur sein.

Vereinbarung. In allen Betrachtungen über Vektoren bezeichne die Variable K einen Körper bzgl. + und ·, in dem die neutralen Elemente bzgl. + und · mit 0 bzw. 1 bezeichnet sind.

9 Mathematische Grundlagen

Aus der Diskussion von Vektorräumen in der Linearen Algebra greifen wir jene Themen heraus, die für die Entwicklung und das Verständnis der vorgestellten Algorithmen relevant sind.

Definition

Vektorraum, Vektor, Skalar, Nullvektor. Wir nennen $(V, \oplus, \odot)$ einen *Vektorraum über K* genau dann, wenn $(V, \oplus)$ eine Abelsche[1] Gruppe ist, und die Abbildung $\odot : K \times V \to V$ für alle $\lambda, \mu \in K$ und alle $u, v \in V$ die Rechengesetze

Assoziativgesetz: $\lambda \odot (\mu \odot v) = (\lambda \cdot \mu) \odot v$

Identität: $1 \odot v = v$

Distributivgesetz: $(\lambda + \mu) \odot v = (\lambda \odot v) \oplus (\mu \odot v)$

Distributivgesetz: $\lambda \odot (u \oplus v) = (\lambda \odot u) \oplus (\lambda \odot v)$

erfüllt. Die Elemente von V heißen *Vektoren*, jene von K sind *Skalare*. Das neutrale Element von $(V, \oplus)$ wird *Nullvektor* genannt und mit 0 bezeichnet. Für einen Vektorraum über K schreiben wir V_K.

[1] Abel, Niels Henrik: 1802–1829, norwegischer Mathematiker. Bewies 1824, dass eine allgemeine algebraische Gleichung 5. Grades algebraisch nicht lösbar ist. Er arbeitete zur gleichen Zeit wie Jacobi, siehe Seite 34, an elliptischen Funktionen, was beiden große Hochachtung von Legendre, dem Vater dieses Gebiets, einbrachte. Ihm zu Ehren wird auch der Abelpreis für außergewöhnliche mathematische Arbeiten verliehen.

Bezeichnung

Anstelle von $u \oplus v$ schreibt man oft der Einfachheit halber wieder $u + v$, und für $\lambda \odot v$ ist $\lambda \cdot v$ oder sogar nur λv in Gebrauch.

Beispiel

Für jedes $m \in \mathbb{N}$ ist die Menge der Spaltenvektoren der Länge m

$$K^m := \left\{ \begin{pmatrix} x_1 \\ \vdots \\ x_m \end{pmatrix} \mid x_1, \ldots, x_m \in K \right\}$$

mit

$$\begin{pmatrix} x_1 \\ \vdots \\ x_m \end{pmatrix} + \begin{pmatrix} y_1 \\ \vdots \\ y_m \end{pmatrix} := \begin{pmatrix} x_1+y_1 \\ \vdots \\ x_m+y_m \end{pmatrix} \qquad \lambda \cdot \begin{pmatrix} x_1 \\ \vdots \\ x_m \end{pmatrix} := \begin{pmatrix} \lambda x_1 \\ \vdots \\ \lambda x_m \end{pmatrix} \qquad 0 := \begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix} \tag{9.1}$$

ein Vektorraum über K. Speziell mit $\mathbb{R}^m$ setzen wir uns im nächsten Abschnitt auseinander.

Bemerkung. Einen Spaltenvektor $x \in K^m$ betrachtet man meist als Matrix in $K^{m\times 1}$ bestehend aus lediglich einer Spalte[2]. Die transponierte Matrix $x^T \in K^{1\times m}$ wird dann als Zeilenvektor $(x_1\ x_2\ \ldots\ x_m)$ aufgefasst[3]. In weiterer Folge beschreibt $(x_1\ \ldots\ x_m)^T$ ein Element im Vektorraum K^m. Wir verwenden K^m sowohl für die Menge der m-Tupel über K als auch für die Menge der Spaltenvektoren der Länge m über K.

Wir haben den Begriff Linearkombination schon früher verwendet, etwa bei der Darstellung des größten gemeinsamen Teilers als Linearkombination der Operanden auf Seite 66. Eine *Linearkombination* (über K) von Elementen einer Familie[4] $(v_j)_{j\in J}$ ist eine *endliche Summe* der Form $\sum_{i\in I} \lambda_i \cdot v_i$, wobei I eine endliche Teilmenge der Indexmenge J ist, und $\lambda_i \in K$ für alle $i \in I$ gilt.

Definition

Lineare Hülle. Sei J eine Indexmenge und $v = (v_j)_{j\in J}$ eine Familie in V_K. Dann heißt

$$L(v) := \left\{ \sum_{i\in I} \lambda_i \cdot v_i \mid I \subseteq J, I \text{ endlich}, \lambda_i \in K \text{ für } i \in I \right\}$$

die *lineare Hülle* von v. Im Fall einer endlichen Familie $v = (v_1, \ldots, v_n)$ wird anstelle von $L(v)$ des Öfteren auch $\operatorname{span}(v_1, \ldots, v_n)$ geschrieben[5]. Die lineare Hülle einer Familie von Vektoren besteht also (auch für unendliche Familien) aus allen Linearkombinationen von Elementen der Familie.

[2] Aus der linearen Algebra setzen wir $K^{m\times n}$, die Menge der $(m \times n)$-Matrizen über K, und das Rechnen mit Matrizen als bekannt voraus, auch wenn wir die algorithmischen Aspekte erst in Band *2* behandeln werden.

[3] Im Unterschied zu Tupel werden in einem Zeilenvektor die Elemente ohne Beistriche nebeneinander geschrieben.

[4] Eine Familie $(v_j)_{j\in J}$ in V ist nichts anderes als eine Funktion $f : J \to V,\ j \mapsto v_j$, wobei man durch die Schreibweise aber auf den Namen f der Funktion verzichtet. Ein Tupel $(v_1, \ldots, v_n)$ kann als Familie mit Indexmenge $J = \{1, \ldots, n\}$ angesehen werden, und eine unendliche Folge $(v_1, v_2, \ldots)$ als eine Familie mit Indexmenge $J = \mathbb{N}$. Die Unterscheidung zwischen Tupel und Mengen von Seite 47 überträgt sich analog auf allgemeine Familien.

[5] span: engl. für Bereich, Spanne.

Definition

Lineare Unabhängigkeit. Sei J eine Indexmenge. Eine Familie $(v_j)_{j\in J}$ in V_K heißt *linear unabhängig* genau dann, wenn für jede endliche Menge $I \subseteq J$ und alle $\lambda_i \in K$ mit $i \in I$

$$\sum_{i\in I} \lambda_i \cdot v_i = 0 \Rightarrow \lambda_i = 0 \text{ für alle } i \in I$$

gilt. Andernfalls heißt $(v_j)_{j\in J}$ *linear abhängig*[6].

Definition

Basis. Eine Familie b in V_K heißt eine *Basis* von V_K genau dann, wenn $L(b) = V_K$ und b linear unabhängig ist.

Beispiel

Für $v' = (v'_j)_{j\in\{1,2\}} = ((1\ 0)^T, (0\ 1)^T) \in (\mathbb{R}^2)^2$ ist $L(v') = \text{span}(v'_1, v'_2) = \mathbb{R}^2$. Da $\lambda_1 v'_1 + \lambda_2 v'_2 = 0$ nur mit $\lambda_1 = \lambda_2 = 0$ erfüllbar ist, ist v' linear unabhängig, und somit ist

$$b' := v' = \left(\begin{pmatrix}1\\0\end{pmatrix}, \begin{pmatrix}0\\1\end{pmatrix}\right)$$

eine Basis von $\mathbb{R}^2$. Auch für $v'' = ((1\ 1)^T, (0\ 2)^T, (1\ -2)^T)$ ist die lineare Hülle $L(v'') = \mathbb{R}^2$, weil jeder Vektor $(x_1\ x_2)^T \in \mathbb{R}^2$ sich mit $\lambda_1 = x_1$, $\lambda_2 = (x_2 - x_1)/2$ und $\lambda_3 = 0$ als $\lambda_1 v''_1 + \lambda_2 v''_2 + \lambda_3 v''_3$ darstellen lässt. v'' ist nicht linear unabhängig, da $\lambda_1 v''_1 + \lambda_2 v''_2 + \lambda_3 v''_3 = 0$ auch mit $\lambda_1 = 1, \lambda_2 = -3/2$ und $\lambda_3 = -1$ erfüllbar ist. Es handelt sich also um keine Basis von $\mathbb{R}^2$. Lassen wir in v'' den dritten Vektor weg, so ist die Familie

$$b'' := v''' = \left(\begin{pmatrix}1\\1\end{pmatrix}, \begin{pmatrix}0\\2\end{pmatrix}\right)$$

wieder linear unabhängig mit $L(v''') = \mathbb{R}^2$, sie ist damit eine Basis von $\mathbb{R}^2$. Schlussendlich ist für $v'''' = ((1\ 0)^T, (2\ 0)^T)$ in $\mathbb{R}^2$ die lineare Hülle durch

$$L(v'''') = \left\{(\lambda_1\ 0)^T + (\lambda_2\ 0)^T \;\middle|\; \lambda_1, \lambda_2 \in \mathbb{R}\right\} = \left\{(\lambda\ 0)^T \;\middle|\; \lambda \in \mathbb{R}\right\} = \mathbb{R} \times \{0\} \subsetneq \mathbb{R}^2$$

gegeben, und v'''' ist linear abhängig, da $\lambda_1 v''''_1 + \lambda_2 v''''_2 = 0$ etwa auch $\lambda_1 = 2$ und $\lambda_2 = -1$ erlaubt.

Eines der zentralen Resultate der Linearen Algebra besagt, dass jeder Vektorraum eine Basis besitzt. Der Beweis dieser Existenzaussage ist nicht konstruktiver Natur[7], er basiert auf dem Auswahlaxiom der Mengenlehre. Wir haben im Beispiel oben gesehen, dass ein Vektorraum auch mehrere Basen haben kann, die Anzahl der Basiselemente ist aber für jeden Vektorraum V_K eindeutig bestimmt, wir nennen sie die *Dimension* $\dim(V_K)$ von V_K. Ist $\dim(V_K) \in \mathbb{N}_0$, dann[8] spricht man von einem *endlichdimensionalen*, andernfalls von einem *unendlichdimensionalen* Vektorraum V_K.

[6] Man nennt oft nicht nur die Familie als Ganzes sondern auch die einzelnen Elemente der Familie linear (un)abhängig.

[7] Aus dem Beweis kann also kein Algorithmus zur Konstruktion einer Basis abgeleitet werden.

[8] Die Dimension eines Vektorraums kann auch gleich 0 sein. Ein Beispiel dafür ist der Vektorraum $\{0\}$, der nur aus dem Nullvektor besteht, und dessen Basis durch $\varnothing$ gegeben ist.

Beispiel

In K^m bildet die Familie $((1\ 0\ \ldots\ 0)^T, \ldots, (0\ \ldots\ 0\ 1)^T)$ eine Basis, sie wird die *kanonische Basis* des K^m genannt. Es gilt $\dim(K^m) = m$, daher ist K^m ein endlichdimensionaler Vektorraum.

Für Beispiele von unendlichdimensionalen Räumen verweisen wir auf die lineare Algebra bzw. auf Seite 119. Eine linear unabhängige Familie mit $\dim(V_K)$ Vektoren bildet stets eine Basis von V_K, eine Familie mit mehr als $\dim(V_K)$ Vektoren ist immer linear abhängig, siehe Übung III.1.

Jedes Element eines Vektorraums lässt sich als eindeutige Linearkombination der Elemente einer Basis darstellen, vergleiche dazu auch die Basisdarstellungen für natürliche und reelle Zahlen auf den Seiten 48 bzw. 81.

Satz

Basisdarstellung von Vektoren. Sei $b = (b_j)_{j\in J}$ eine Basis von V_K. Dann lässt sich jedes $v \in V_K$ eindeutig als

$$v = \sum_{i\in I} \lambda_i \cdot b_i \quad \text{mit } \lambda_i \in K, I \subseteq J \text{ und } I \text{ endlich} \tag{9.2}$$

schreiben. In (9.2) heißt $(\lambda_i)_{i\in I}$ die *Familie der Koordinaten* von v bzgl. b, und λ_i nennt man die i-te Koordinate.

Beweis. Die Existenz einer Darstellung folgt unmittelbar aus $L(b) = V_K$. Zum Nachweis der Eindeutigkeit seien $\sum_{i\in I'} \lambda_i \cdot b_i$ und $\sum_{i\in I''} \mu_i \cdot b_i$ zwei Darstellungen des Vektors v zur Basis b. Dann gilt aber mit $S = I' \cap I''$

$$0 = v - v = \sum_{i\in I'\setminus S} \lambda_i \cdot b_i + \sum_{i\in I''\setminus S} -\mu_i \cdot b_i + \sum_{i\in S} (\lambda_i - \mu_i) \cdot b_i,$$

woraus aufgrund der linearen Unabhängigkeit von b alle Koeffizienten dieser Linearkombination 0 sein müssen, d.h. $\lambda_i - \mu_i = 0$ und damit $\lambda_i = \mu_i$ für $i \in S$ und $\lambda_i = 0$ für $i \in I' \setminus S$ und $\mu_i = 0$ für $i \in I'' \setminus S$. Damit läuft in beiden Darstellungen von v der Index effektiv nur über S, die Darstellungen sind somit identisch. □

Beispiel

Sei $x = (3\ 5)^T \in \mathbb{R}^2$. Die Koordinaten bzgl. der Basis b' aus obigem Beispiel lauten $\lambda_1 = 3$ und $\lambda_2 = 5$, bzgl. der Basis b'' lauten die Koordinaten aber $\lambda_1 = 3$ und $\lambda_2 = 1$.

Man beachte, dass die Darstellung (9.2) immer eine endliche Summe ist, selbst wenn die Basis unendlich viele Elemente enthält. Die Koordinaten eines Vektors sind natürlich von der gewählten Basis abhängig, weshalb in der Koordinatenfamilie die Basis oft explizit angeschrieben wird. In obigem Beispiel würde man etwa $x = (3\ 5)^T_{b'}$ bzw. $x = (3\ 1)^T_{b''}$ für die Koordinatendarstellung bzgl. der jeweiligen Basis schreiben. Die Addition und die Multiplikation mit Skalaren überträgt sich direkt auf die Koordinaten, d.h. zwei Vektoren können addiert werden, indem ihre Koordinaten (bzgl. einer beliebigen Basis b) komponentenweise addiert werden,

und die Multiplikation eines Vektors mit einem Skalar erfolgt durch komponentenweise Multiplikation der Koordinaten (bzgl. einer beliebigen, aber fixen Basis b) mit dem Skalar. Man identifiziert daher üblicherweise K^m mit dem Vektorraum $\{(\lambda_1 \ldots \lambda_m)_b^T \mid \lambda_i \in K \text{ für } i = 1, \ldots, m\}$ für eine beliebige Basis b von K^m.

Der Vektorraum $\mathbb{R}^m$

In weiterer Folge betrachten wir den Vektorraum $V_K = \mathbb{R}^m$ und führen dafür zunächst den Begriff des Skalarprodukts ein[9].

Definition

Skalarprodukt in $\mathbb{R}^m$. Eine Abbildung $\langle .,. \rangle : \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}$, die

bilinear, d.h. für alle $x, x', y, y' \in \mathbb{R}^m$ und $\lambda, \lambda' \in \mathbb{R}$

$$\langle \lambda \cdot x + \lambda' \cdot x', y \rangle = \lambda \cdot \langle x, y \rangle + \lambda' \cdot \langle x', y \rangle$$
$$\langle x, \lambda \cdot y + \lambda' \cdot y' \rangle = \lambda \cdot \langle x, y \rangle + \lambda' \cdot \langle x, y' \rangle,$$

symmetrisch, d.h. $\langle x, y \rangle = \langle y, x \rangle$ für alle $x, y \in \mathbb{R}^m$, und

positiv definit, d.h. $\langle x, x \rangle > 0$ für alle $x \in \mathbb{R}^m$ mit $x \neq 0$

ist, nennt man ein *Skalarprodukt* oder *inneres Produkt* auf $\mathbb{R}^m$.

Ein inneres Produkt genügt der Cauchy-Schwarz[10] Ungleichung

$$|\langle x, y \rangle| \leq \sqrt{\langle x, x \rangle \cdot \langle y, y \rangle}, \; x, y \in \mathbb{R}^m, \tag{9.3}$$

woraus sich abeiten lässt, dass

$$\|x\| := \sqrt{\langle x, x \rangle} \tag{9.4}$$

eine Norm auf $\mathbb{R}^m$ ist.

Beispiel

Für $x, y \in \mathbb{R}^m$ ist

$$\langle x, y \rangle = \sum_{i=1}^{m} x_i \cdot y_i \tag{9.5}$$

das *Euklidsche Skalarprodukt*. Die Euklidsche Norm ist auf Seite 30 gerade durch $\|x\|_2 = \sqrt{\sum_{i=1}^m x_i^2}$ definiert worden.

In der Matrix-Interpretation entspricht das Euklidsche Skalarprodukt genau der Matrixmultiplikation $x^T \cdot y$, zumindest wenn man eine Matrix $(m_{1,1}) \in \mathbb{R}^{1\times 1}$ mit

[9] Skalarprodukte und darauf aufbauende Begriffe werden in der Linearen Algebra auch auf anderen Vektorräumen über $\mathbb{R}$ bzw. über $\mathbb{C}$ definiert.

[10] Schwarz, Hermann Amandus: 1843–1921, deutscher Mathematiker. Sein wahrscheinlich wichtigstes Werk ist eine Festschrift zum 70. Geburtstag von Weierstraß, siehe Seite 148. Darin enthalten ist die Schwarzsche Ungleichung für Integrale, die die hier verwendete Ungleichung als einen Spezialfall enthält.

dem Skalar $m_{1,1} \in \mathbb{R}$ identifiziert. Genausogut kann man aber auch einen Spaltenmit einem Zeilenvektor im Sinne von Matrizen multiplizieren.

Definition

Äußeres Produkt von Vektoren. Seien $m,n \in \mathbb{N}$, $x \in \mathbb{R}^m$ und $y \in \mathbb{R}^n$. Dann heißt

$$x \cdot y^T := \begin{pmatrix} x_1 \cdot y_1 & \dots & x_1 \cdot y_n \\ \vdots & \ddots & \vdots \\ x_m \cdot y_1 & \dots & x_m \cdot y_n \end{pmatrix} \in \mathbb{R}^{m \times n}$$

das *äußere Produkt* von x und y.

Das äußere Produkt ist somit eine Abbildung von $\mathbb{R}^m \times \mathbb{R}^n$ nach $\mathbb{R}^{m \times n}$. Im äußeren Produkt werden alle Produkte $x_i \cdot y_j$ der Koordinaten von x und y gebildet und in der Matrix angeordnet, in (9.5) hingegen werden nur korrespondierende Koordinaten x_i und y_i multipliziert und dann zu einem Skalar addiert. Aufgrund der Assoziativität der Matrixmultiplikation gilt für $x \in \mathbb{R}^m$ und $y,z \in \mathbb{R}^n$ der einfache Zusammenhang

$$(x \cdot y^T) \cdot z = x \cdot (y^T \cdot z) \tag{9.6}$$

zwischen dem äußeren Produkt und dem Euklidschen Skalarprodukt.

Weiters ist für $x,y \in \mathbb{R}^m$ mit $x,y \neq 0$ der eindeutig bestimmte Winkel $\alpha \in [0,\pi)$ mit

$$\cos(\alpha) = \frac{|\langle x,y \rangle|}{\|x\| \cdot \|y\|} \tag{9.7}$$

als der Winkel zwischen x und y definiert. Er wird mit $\angle(x,y)$ bezeichnet und ist von der Wahl des Skalarprodukts abhängig. Für $\angle(x,y) = \pi/2$ stehen die Vektoren senkrecht aufeinander, was gleichbedeutend ist mit $\langle x,y \rangle = 0$.

Definition

Orthogonalität, Orthonormalität. Seien $x,y \in \mathbb{R}^m$. Die Vektoren x und y heißen zueinander *orthogonal* bzgl. des Skalarprodukts $\langle \cdot,\cdot \rangle$ genau dann, wenn $\langle x,y \rangle = 0$ gilt. Eine Familie von Vektoren $(v_j)_{j \in J}$ mit $v_j \neq 0$ für alle $j \in J$ heißt *orthogonal* genau dann, wenn ihre Elemente paarweise zueinander orthogonal sind, d.h. es gilt für alle $i \neq j \in J$ dann $\langle v_i, v_j \rangle = 0$. Die Familie heißt *orthonormal* genau dann, wenn sie orthogonal ist, und zusätzlich jedes Element *normiert* ist, also $\|v_j\| = 1$ für alle $j \in J$ gilt.

Satz

Die Vektoren einer orthogonalen Familie in $\mathbb{R}^m$ sind linear unabhängig.

Beweis. Sei $(v_j)_{j \in J}$ orthogonal, $I \subseteq J$, I endlich und $\sum_{i \in I} \lambda_i \cdot v_i = 0$. Dann ist für alle $i \in I$ zu zeigen, dass $\lambda_i = 0$ ist. Für ein beliebig aber fix gewähltes $i' \in I$ gilt nach Annahme

$$0 = \langle 0, v_{i'} \rangle = \langle \sum_{i \in I} \lambda_i \cdot v_i, v_{i'} \rangle = \sum_{i \in I} \lambda_i \cdot \langle v_i, v_{i'} \rangle.$$

Daraus folgt mit $\langle v_i, v_{i'} \rangle = 0$ für $i \neq i'$ wegen der Orthogonalität

$$\sum_{i \in I} \lambda_i \cdot \langle v_i, v_{i'} \rangle = \lambda_{i'} \cdot \langle v_{i'}, v_{i'} \rangle = 0.$$

Mit $v_{i'} \neq 0$ ist wegen der positiven Definitheit des Skalarprodukts $\langle v_{i'}, v_{i'} \rangle > 0$, somit muss $\lambda_{i'} = 0$ sein. □

Aus dem Satz folgt, dass eine orthogonale Familie in $\mathbb{R}^m$, die m Vektoren enthält, eine Basis des $\mathbb{R}^m$ ist. Mit Hilfe einer orthonormalen Familie lässt sich jeder beliebige Vektor in eine Summe von paarweise orthogonalen Vektoren zerlegen.

Satz

Sei $q \in (\mathbb{R}^m)^n$ eine Familie orthonormaler Vektoren im $\mathbb{R}^m$ und $v \in \mathbb{R}^m$ mit $n \leq m$. Dann ist für jedes $i = 1, \ldots, n$

$$r = v - \sum_{j=1}^{n} \langle q_j, v \rangle \cdot q_j \tag{9.8}$$

orthogonal zu q_i.

Beweis. Für beliebiges aber fixes $i \in \{1, \ldots, n\}$ ist

$$\langle q_i, r \rangle = \langle q_i, v \rangle - \sum_{j=1}^{n} \langle q_j, v \rangle \cdot \langle q_i, q_j \rangle.$$

Wegen der Orthonormalität gilt – unter Verwendung des Kronecker[11] Delta[12] – $\langle q_i, q_j \rangle = \delta_{ij}$, woraus

$$\langle q_i, r \rangle = \langle q_i, v \rangle - \langle q_i, v \rangle \langle q_i, q_i \rangle = 0$$

folgt. Damit ist r orthogonal zu q_i. □

Für den Vektor v gilt also die Zerlegung

$$v = r + \sum_{i=1}^{n} \langle q_i, v \rangle \cdot q_i \tag{9.9}$$

in die paarweise orthogonalen Vektoren r und $\langle q_i, v \rangle \cdot q_i$. Ist die orthonormale Familie q eine Basis, man spricht in diesem Fall von einer *Orthonormalbasis*, so muss $n = m$ und damit $r = 0$ in (9.9) gelten. Der Vorteil einer Orthonormalbasis ist, dass sich die Koordinaten λ_i eines Vektors v einfach durch $\langle v, q_i \rangle$ ermitteln lassen.

[11] Kronecker, Leopold: 1823–1891, deutscher Mathematiker. Er lieferte wesentliche Beiträge zur Algebra und zur Theorie der elliptischen Funktionen und algebraischen Zahlen sowie zu den Zusammenhängen zwischen diesen Gebieten. Er zweifelte die Relevanz von inkonstruktiven Beweisen an und weigerte sich daher, irrationale und transzendente Zahlen zu verwenden.

[12] $\delta_{ij} := \begin{cases} 1 & \text{für } i = j \\ 0 & \text{sonst.} \end{cases}$

Unter Zuhilfenahme des äußeren Produkts von Vektoren kann man (9.9) bei Betrachtung des Euklidschen Skalarprodukts auch als

$$v = r + \sum_{i=1}^{n} (q_i \cdot q_i^T) \cdot v \tag{9.10}$$

schreiben. Dabei *projiziert* die durch den Vektor q_i definierte Matrix $q_i \cdot q_i^T \in \mathbb{R}^{m\times m}$ den Vektor v auf jene Komponente $(q_i^T \cdot v) \cdot q_i$, die parallel zu q_i ist, vergleiche auch mit Übungsaufgabe III.3. Abschließend heben wir noch eine wichtige Eigenschaft dieser speziellen *Projektionsmatrizen* hervor.

Satz

Seien $p,q \in \mathbb{R}^m$ zueinander bzgl. (9.5) orthogonal. Dann gilt für das Matrixprodukt von $p \cdot p^T$ mit $q \cdot q^T$

$$(p \cdot p^T) \cdot (q \cdot q^T) = \mathbf{0}, \tag{9.11}$$

wobei $\mathbf{0}$ die Nullmatrix in $\mathbb{R}^{m\times m}$ bezeichnet.

Beweis. Die Aussage gilt wegen

$$(p \cdot p^T) \cdot (q \cdot q^T) = p \cdot \underbrace{(p^T \cdot q)}_{=0} \cdot q^T = 0 \cdot p \cdot q^T = \mathbf{0}.$$

□

Infolge gilt auch

$$(E - p \cdot p^T) \cdot (E - q \cdot q^T) = E - (p \cdot p^T) - (q \cdot q^T), \tag{9.12}$$

wobei E die Einheitsmatrix in $\mathbb{R}^{m\times m}$ bezeichnet.

10 Vektoren am Computer

Vektorräume können sehr abstrakte mathematische Objekte sein. Zur Darstellung am Computer konzentrieren wir uns daher auf die endlichdimensionalen Vektorräume K^m.

Computerrepräsentation (Datenstruktur $\mathcal{V}_K^m$ für Vektoren)**.** Zur Darstellung von Vektoren am Computer wählen wir eine beliebige Basis b von K^m und speichern die Koordinaten $\lambda_1, \ldots, \lambda_m$ bzgl. b in einer geeigneten Datenstruktur $\mathcal{V}_K^m$, z.B. als Tupel $(\lambda_1, \ldots, \lambda_m)$, ab. Die zugrunde liegende Basis b muss dabei nicht extra gespeichert werden, da ja ein Vektor in K^m wie oben ausgeführt ohnehin mit seinen Koordinaten bzgl. einer beliebigen Basis identifiziert wird[13]. Anstelle einer Zugriffsoperation auf

[13] Ist man an Darstellungen bzgl. verschiedener Basen interessiert, so können verschiedene Datenstrukturen für die jeweiligen Basen verwendet werden.

das in $x \in \mathcal{V}_K^m$ abgespeicherte Koordinatentupel stellen wir direkte Zugriffsoperationen auf die einzelnen Koordinaten zur Verfügung und sprechen die i-te Koordinate von $x \in \mathcal{V}_K^m$ mit x_i für $i = 1, \ldots, m$ an.

Die Grundoperationen zum Addieren von Vektoren x und y und zum Multiplizieren von x mit einem Skalar λ können in dieser Datenstruktur basierend auf (9.1) in einfacher Weise mit Schleifen realisiert werden, siehe etwa *AddV* zur Vektoraddition. Es ist leicht einzusehen, dass der Aufwand für *AddV* sowie für die Berechnung von $\lambda \cdot x$ jeweils $O(m)$ Grundoperationen in K beträgt.

Algorithmus *AddV*: Addition in K^m

```
for i from 1 to m
    s_i ← x_i + y_i
return s
```

Aufruf: $AddV(x,y)$
Eingabe: $x,y \in \mathcal{V}_K^m$.
Ausgabe: $s \in \mathcal{V}_K^m$
mit: $s = x + y$.

Computerprogrammierung (Parametrisierte Datenstrukturen)**.** *Im Algorithmus AddV steht $x_i + y_i$ für die Addition in K. Analog dazu baut die Berechnung von $\lambda \cdot x$ auf der Multiplikation $\lambda \cdot x_i$ in K auf. Für eine elegante Realisierung der Rechenoperationen auf Vektoren ist es empfehlenswert, den zugrunde liegenden Körper K in der Datenstruktur mit abzuspeichern, weil man dann mit einer einzigen universellen Realisierung der Algorithmen* für beliebiges K *das Auslangen finden kann. Wie man K in der Datenstruktur abspeichert, und wie die Auswahl der dazugehörigen Addition und Multiplikation in K basierend auf der in $\mathcal{V}_K^m$ gespeicherten Information über K getroffen wird, hängt von der verwendeten Programmiersprache ab. Ist ein Abspeichern von K in der Datenstruktur nicht möglich, so kann diese Information als zusätzlicher Parameter in den Aufruf der Vektoralgorithmen einfließen. Alternativ können die Algorithmen für jeden benötigten Koeffizientenbereich unter eigenem Namen realisiert werden. Dann würden etwa AddVG oder AddVQ für Algorithmen zur Addition von Vektoren mit Gleitkomma- oder rationalen Koordinaten stehen.*

11 Euklidsches Skalarprodukt in $\mathbb{R}^m$

Ausführlicher beschäftigen wir uns nun mit der Berechnung des Euklidschen Skalarprodukts zweier Vektoren in $\mathbb{R}^m$. Im Folgenden steht $\langle \cdot, \cdot \rangle$ stets für (9.5).

Problemstellung (Euklidsches Skalarprodukt in $\mathbb{R}^m$).

Gegeben: $x,y \in \mathbb{R}^m$
Gesucht: $s \in \mathbb{R}$
mit: $s = \langle x,y \rangle$

Für die Konditionsanalyse definieren wir den Vektor

$$z := \binom{x}{y} = (x_1 \ldots x_m \, y_1 \ldots y_m)^T \in \mathbb{R}^{2m}.$$

Damit lässt sich die Abbildung φ von Seite 3, die den Daten die eindeutige Lösung des Problems zuordnet, durch

$$\varphi : \mathbb{R}^{2m} \to \mathbb{R},\ z \mapsto \sum_{i=1}^{m} z_i z_{i+m} \tag{11.13}$$

angeben. Der folgende Satz gibt Auskunft über die relative komponentenweise Kondition des Problems, siehe Seite 31.

Satz

Kondition des Skalarprodukts. Seien φ wie in (11.13) und $x, y \in \mathbb{R}^m$ mit $x_i \neq 0$, $y_i \neq 0$ für $i = 1, \ldots, m$, mit $\langle x, y\rangle \neq 0$ gegeben. Dann ist

$$\kappa_{\text{rel,komp}} = \frac{2\sum_{i=1}^{m} |x_i| \cdot |y_i|}{|\langle x, y\rangle|}$$

die relative komponentenweise Kondition des Problems (φ, x, y).

Beweis. Die Abbildung (11.13) ist partiell differenzierbar, und es gilt $\frac{\partial \varphi}{\partial z_i}(z) = z_{i+m}$ für $i = 1, \ldots, m$ und $\frac{\partial \varphi}{\partial z_i}(z) = z_{i-m}$ für $i = m+1, \ldots, 2m$. Aus (3.54) auf Seite 33 folgt damit

$$\begin{aligned}
\kappa_{\text{rel,komp}} &= \frac{\sum_{i=1}^{2m} |\frac{\partial \varphi}{\partial z_i}(z)| \cdot |z_i|}{\left|\sum_{i=1}^{m} z_i z_{i+m}\right|} = \frac{\sum_{i=1}^{m} |z_{i+m}| \cdot |z_i| + \sum_{i=m+1}^{2m} |z_{i-m}| \cdot |z_i|}{\left|\sum_{i=1}^{m} z_i z_{i+m}\right|} \\
&= \frac{2\sum_{i=1}^{m} |z_{i+m}| \cdot |z_i|}{\left|\sum_{i=1}^{m} z_i z_{i+m}\right|} = \frac{2\sum_{i=1}^{m} |x_i| \cdot |y_i|}{|\langle x, y\rangle|}.
\end{aligned}$$

□

Die Kondition des Problems (φ, x, y) ist also insbesondere dann hoch, wenn die Vektoren x, y bei betragsmäßig großen Komponenten nahezu orthogonal zueinander sind, d.h. wenn $\langle x, y\rangle$ nahezu gleich null ist. Für $x = \lambda \cdot y$ mit $\lambda \in \mathbb{R}$, $\lambda \neq 0$, ist das Problem wegen $\kappa_{\text{rel,komp}} = 2$ gut konditioniert.

Durch *SkalarproduktEuklid* ist ein einfacher Algorithmus zur Berechnung des Skalarprodukts gegeben, worin die Summe über die komponentenweisen Produkte gebildet wird. Der Aufwand dafür beträgt m Multiplikationen sowie $m - 1$ Additionen, also $2m - 1$ Elementaroperationen.

Die Vorwärtsstabilität der Gleitkommarealisierung von *SkalarproduktEuklid* lässt sich einfach aus der auf Seite 95 für *SummeS* gezeigten ableiten. Wir untersuchen daher an dieser Stelle die Rückwärtsstabilität, siehe Seite 42, zu deren Nachweis der folgende Satz die Grundlage liefert. Darin stehen $\tilde{x}$ und $\tilde{y}$ wieder für die gerundeten Eingangsdaten.

Algorithmus *SkalarproduktEuklid*: Euklidsches Skalarprodukt in $\mathbb{R}^m$

```
s ← 0
for i from 1 to m
    s ← s + x_i · y_i
return s
```

Aufruf: *SkalarproduktEuklid*(x,y)
Eingabe: $x,y \in \mathbb{R}^m$.
Ausgabe: $s \in \mathbb{R}$
mit: $s = \langle x,y\rangle$.

Satz

Seien $x,y,\tilde{x},\tilde{y} \in \mathbb{R}^m$ mit $x_i,y_i,\tilde{x}_i,\tilde{y}_i \neq 0$ und $|x_i - \tilde{x}_i|/|x_i| \leq \mathbf{u}$ bzw. $|y_i - \tilde{y}_i|/|y_i| \leq \mathbf{u}$ für $i = 1,\ldots,m$. Die Gleitkommarealisierung des Algorithmus *SkalarproduktEuklid* zur Berechnung von $\langle x,y\rangle$ liefert eine Näherung $\tilde{s} \in \mathbb{R}$ an $\langle x,y\rangle$ mit

$$\tilde{s} = \langle \hat{x},\hat{y}\rangle, \tag{11.14}$$

wobei $\hat{x},\hat{y} \in \mathbb{R}^m$ mit $\hat{x} = \tilde{x}$ und

$$\frac{|\hat{y}_i - \tilde{y}_i|}{|\tilde{y}_i|} \lesssim (m - i + 2)\mathbf{u} \quad \text{für } \mathbf{u} \to 0. \tag{11.15}$$

Beweis. Bezeichnen wir mit $\tilde{p}_i$ jenes Produkt, das bei der Gleitkommamultiplikation von $\tilde{x}_i$ und $\tilde{y}_i$ entsteht, so gilt nach (2.21) auf Seite 16

$$\tilde{p}_i = \tilde{x}_i\tilde{y}_i(1 + \varepsilon_i) \tag{11.16}$$

mit der Abschätzung $|\varepsilon_i| \leq \mathbf{u}$ für den dabei entstehenden Rundungsfehler ε_i.

Sei nun $\tilde{s}^{(i)}$ das Zwischenergebnis am Ende des i-ten Schleifendurchlaufs, insbesondere also $\tilde{s}^{(1)} = \tilde{p}_1$ und $\tilde{s}^{(m)} = \tilde{s}$. Dann gilt

$$\begin{aligned}
\tilde{s} &= \tilde{s}^{(m-1)} \tilde{+} \tilde{p}_m = (\tilde{s}^{(m-1)} + \tilde{p}_m)(1 + \varepsilon'_m) \\
&= \left(\tilde{p}_m + (\tilde{s}^{(m-2)} + \tilde{p}_{m-1})(1 + \varepsilon'_{m-1})\right)(1 + \varepsilon'_m) \\
&= \sum_{i=2}^{m} \tilde{p}_i \prod_{j=i}^{m}(1 + \varepsilon'_j) + \tilde{p}_1 \prod_{j=2}^{m}(1 + \varepsilon'_j),
\end{aligned}$$

wobei ε'_j mit $|\varepsilon'_j| \leq \mathbf{u}$ für $j = 2,\ldots,m$ die bei den Gleitkommaadditionen entstehenden Rundungsfehler bezeichnen. Setzen wir zusätzlich $\varepsilon'_1 = 0$, so gilt

$$\tilde{s} = \sum_{i=1}^{m} \tilde{p}_i \prod_{j=i}^{m}(1 + \varepsilon'_j), \tag{11.17}$$

woraus mit (11.16)

$$\tilde{s} = \sum_{i=1}^{m} \tilde{x}_i \cdot \left(\tilde{y}_i(1 + \varepsilon_i) \prod_{j=i}^{m}(1 + \varepsilon'_j)\right)$$

folgt. Dabei handelt es sich mit $\hat{x}_i := \tilde{x}_i$ und $\hat{y}_i := \tilde{y}_i(1 + \varepsilon_i)\prod_{j=i}^{m}(1 + \varepsilon'_j)$ um die Darstellung (11.14). Die Abschätzung (11.15) ergibt sich aus den Beziehungen

$$(1+\varepsilon_i)\prod_{j=i}^{m}(1+\varepsilon'_j) \approx \left(1+\varepsilon_i+\sum_{j=i}^{m}\varepsilon'_j\right) \quad \text{für } \mathbf{u}\to 0 \quad \text{und}$$

$$\left|\varepsilon_i+\sum_{j=i}^{m}\varepsilon'_j\right| \le \mathbf{u}+\sum_{j=i}^{m}\mathbf{u} = (m-i+2)\mathbf{u}.$$

□

Das Gleitkommaresultat $\tilde{s}$ lässt sich also als exaktes inneres Produkt der Vektoren $\hat{x}$ und $\hat{y}$ darstellen. Aus (11.15) folgt mit $i=1$ die Stabilitätszahl $C_{R,komp} = m+1$, siehe auch (4.74). Da diese (für $m>1$) die Anzahl $2m-1$ der Elementaroperationen von *SkalarproduktEuklid* nicht übersteigt, ist dessen Gleitkommarealisierung stabil im Sinne der Rückwärtsanalyse. Nach Seite 42 ist diese damit auch vorwärtsstabil.

Bemerkung. Der obige Beweis liefert über (11.17) natürlich auch den Nachweis der Rückwärtsstabilität der Gleitkommarealisierung des Summationsalgorithmus *SummeS*. Wie auf Seite 96 erweist es sich natürlich auch bei *SkalarproduktEuklid* im Hinblick auf eine obere Rundungsfehlerschranke als günstig, die einzelnen Produkte p_i vor ihrer Summation dem Betrag nach aufsteigend zu sortieren.

12 Orthonormalisierung in $\mathbb{R}^m$

Eine der zentralen Fragestellung in der Linearen Algebra ist die Orthonormalisierung einer endlichen Familie von Vektoren. Dabei sucht man zu linear unabhängigen Vektoren $a_1,\ldots,a_n$ orthonormale Vektoren $q_1,\ldots q_n$, so dass für alle $j=1,\ldots,n$ die j-dimensionalen Räume $\text{span}(a_1,a_2,\ldots,a_j)$ und $\text{span}(q_1,q_2,\ldots,q_j)$ übereinstimmen. Auch hier betrachten wir wieder speziell den Fall $V_K = \mathbb{R}^m$ mit dem Euklidschen Skalarprodukt.

Problemstellung (Orthonormalisierung).

Gegeben:	$a \in (\mathbb{R}^m)^n$
mit:	a linear unabhängig.
Gesucht:	$q \in (\mathbb{R}^m)^n$
mit:	q orthonormal und
	$\text{span}(q_1,\ldots,q_j) = \text{span}(a_1,\ldots,a_j)$ für $j=1,\ldots,n$.

Die Lösbarkeit dieser Problemstellung beruht auf folgendem Satz.

Satz

Sei $a \in (\mathbb{R}^m)^n$ eine linear unabhängige Familie. Dann existiert eine orthonormale Familie $q \in (\mathbb{R}^m)^n$ mit

$$\text{span}(q_1,q_2,\ldots,q_j) = \text{span}(a_1,a_2,\ldots,a_j) \quad \text{für } j=1,\ldots,n. \tag{12.18}$$

Beweis. Wir zeigen die Aussage mit Hilfe von Induktion bzgl. j. Mit $q_1 = a_1/\|a_1\|_2$ ist (12.18) für $j=1$ trivialerweise erfüllt. Für ein fixes $j \in \{2,\ldots,n\}$ nehmen wir

nun an, dass (12.18) für $j-1$ gilt. Unser Ziel ist es dann, einen normalisierten Vektor $q_j \in \operatorname{span}(a_1,\ldots,a_j)$ zu finden, der zu $q_1,\ldots,q_{j-1}$ orthogonal ist. Der Vektor

$$v_j = a_j - \sum_{i=1}^{j-1} \langle q_i, a_j\rangle \cdot q_i \tag{12.19}$$

liegt im von $a_1,\ldots,a_j$ aufgespannten Raum und ist nach (9.9) orthogonal zu $q_1,\ldots,q_{j-1}$. Weiters gilt $v_j \neq 0$, da andernfalls ein Widerspruch zur linearen Unabhängigkeit der Familie a besteht. Somit erfüllt $q_j = v_j/\|v_j\|_2$ die geforderten Bedingungen, und es gilt (12.18) auch für j. □

Die Lösung der Problemstellung ist allerdings nicht eindeutig, da zur Normalisierung des Vektors v_j ja auch die Division durch $-\|v_j\|_2$ herangezogen werden kann. Vereinbaren wir aber die Normalisierung gemäß $q_j = v_j/\|v_j\|_2$, so folgt unter Verwendung der Notation $r_{ij} = \langle q_i, a_j\rangle$ für $i \neq j$ und $r_{jj} = \|a_j - \sum_{i=1}^{j-1} r_{ij} q_i\|_2$ aus obigem Beweis die Darstellung

$$q_j = \frac{a_j - \sum_{i=1}^{j-1} r_{ij} q_i}{r_{jj}} \quad \text{für } j = 1,\ldots,n. \tag{12.20}$$

Im Hinblick auf die Kondition dieses Problems, d.h. der Sensitivität der Vektoren q_j aus (12.20) bezüglich Störungen der Eingangsdaten a_j, erwähnen wir an dieser Stelle lediglich, dass die Verstärkung des Datenfehlers invers propertional zu einem Maß für die lineare Unabhängigkeit der Vektoren a_j ist. Bilden die Eingangsdaten a_j eine nahezu linear abhängige Familie, etwa wenn sich zumindest ein Vektor beinahe als Linearkombination der übrigen darstellen lässt, so ist das Problem also sehr schlecht konditioniert. Für eine exakte Konditionsformulierung benötigt man jedoch das Rüstzeug von Matrixnormen, welches wir erst in Band *2* einführen werden. Dort begegnen wir jedoch obiger Fragestellung in Form der *QR*-Zerlegung einer Matrix *A* ein zweites Mal, dann reichen wir auch die Details nach.

Unter dem klassischen Gram[14]-Schmidt[15] Verfahren versteht man die Umsetzung des obigen konstruktiven Existenzbeweises, es ist in Algorithmus *GramSchmidt* zusammengefasst.

Zur Berechnung der inneren Produkte und Normen müssen natürlich geeignete Unteralgorithmen aufgerufen werden. Unter Verwendung von *SkalarproduktEuklid* von Seite 108 und der Beziehung (9.4) beträgt der Aufwand zur Durchführung $O(mn^2)$ elementarer Rechenoperationen $+, -, \cdot, /$, siehe Übungsaufgabe III.5. Um Speicherplatz zu sparen, kann man a_j mit v_j und anschließend v_j mit q_j überschreiben. Auch die Verwendung verschiedener r_{ij} ist nicht unbedingt erforderlich,

[14] Gram, Jorgen Pedersen: 1850–1916, dänischer Mathematiker. Er arbeitete in einer Versicherung und gründete später eine eigene Versicherungsgesellschaft. Er beschäftigte sich mit Wahrscheinlichkeitsrechnung und numerischer Analysis. Das nach ihm benannte Orthonormalisierungsverfahren war aber schon Legendre und auch Cauchy bekannt.

[15] Schmidt, Erhard: 1876–1959, deutscher Mathematiker. Seine Hauptinteressen galten Hilberträumen und Integralgleichungen, er gilt als einer der Väter der modernen abstrakten Funktionalanalysis. Sein Beweis des Jordanschen Kurvensatzes gilt als Klassiker, der Originalbeweis von Jordan war übrigens nicht korrekt.

Algorithmus *GramSchmidt*: Klassisches Orthonormalisierungsverfahren in $\mathbb{R}^m$

```
for j from 1 to n
    v_j ← a_j
    for i from 1 to j − 1
        r_ij ← ⟨q_i, a_j⟩
        v_j ← v_j − r_ij · q_i
    r_jj ← ‖v_j‖_2
    q_j ← v_j / r_jj
return q
```

Aufruf: *GramSchmidt*(a)
Eingabe: $a \in (\mathbb{R}^m)^n$
mit: a linear unabhängig.
Ausgabe: $q \in (\mathbb{R}^m)^n$
mit: q orthonormal und
$\operatorname{span}(q_1,\ldots,q_j) = \operatorname{span}(a_1,\ldots,a_j)$ für $j = 1,\ldots,n$.

sie ist hier durch den Zusammenhang mit der QR-Zerlegung aus Band *2* motiviert.

Beispiel

Die Vektoren $a_1 = (1\ 1\ 1\ -1)^T, a_2 = (2\ -1\ -1\ 1)^T, a_3 = (-1\ 2\ 2\ 1)^T \in \mathbb{R}^4$ sind linear unabhängig, siehe Übung III.2. Mit $q_1 = \frac{1}{2}(1\ 1\ 1\ -1)^T$ ergibt sich nach *GramSchmidt* $r_{12} = -\frac{1}{2}$ und

$$v_2 = (2\ -1\ -1\ 1)^T + \frac{1}{2}\cdot\frac{1}{2}(1\ 1\ 1\ -1)^T = \frac{1}{4}(9\ -3\ -3\ 3)^T$$

also $q_2 = \frac{1}{2\sqrt{3}}(3\ -1\ -1\ 1)^T$, sowie $r_{13} = 1, r_{23} = -\sqrt{3}$ und

$$v_3 = (-1\ 2\ 2\ 1)^T - \frac{1}{2}(1\ 1\ 1\ -1)^T + \frac{1}{2}(3\ -1\ -1\ 1)^T = (0\ 1\ 1\ 2)^T,$$

also $q_3 = \frac{1}{\sqrt{6}}(0\ 1\ 1\ 2)^T$. Leicht überzeugt man sich von der Gültigkeit von

$$\langle q_1, q_2\rangle = \langle q_2, q_3\rangle = \langle q_1, q_3\rangle = 0.$$

Bei Durchführung des Gram-Schmidt Verfahrens in Gleitkommaarithmetik erhält man infolge der Rundungsfehlerfortpflanzung jedoch Vektoren $\tilde{q}_1, \ldots \tilde{q}_n$, für die dann $\langle \tilde{q}_i, \tilde{q}_j\rangle = 0$ mit $i \neq j$ in der Regel nicht erfüllt ist.

Beispiel

Die Vektoren $a_j = (0^{j-1}\ (\frac{1}{24})^{j-1}\ (\frac{2}{24})^{j-1}\ \ldots\ (\frac{23}{24})^{j-1}\ 1)^T \in \mathbb{R}^{25}$ für $j = 1,\ldots,15$ bilden eine lineare unabhängige Familie $a \in (\mathbb{R}^{25})^{15}$, siehe Übung III.7. Deren Orthonormalisierung mit Hilfe des Algorithmus *GramSchmidt* führt bei Gleitkommarealisierung jedoch auf eine Familie $(\tilde{q}_1,\ldots,\tilde{q}_{15})$, die die Orthogonalitätsforderung in keiner Weise erfüllt. So ergibt etwa dann das innere Produkt von $\tilde{q}_{13}$ mit $\tilde{q}_{15}$ anstelle von 0 den Wert 0.9998.

Ein Nachteil des klassischen Algorithmus *GramSchmidt* ist, dass sich dieser *numerische Verlust der Orthogonalität* nicht im Sinne einer numerischen Stabilitätsaussage durch das Produkt aus Rundungseinheit **u** und Konditionszahl abschätzen lässt.

Dies ist nur nach einer Modifikation des Algorithmus möglich. Grundlage dafür ist die Beobachtung, dass sich der Vektor v_j aus (12.19) wegen (9.6) auch als

$$v_j = \left(E - \sum_{i=1}^{j-1} q_i \cdot q_i^T \right) \cdot a_j$$

darstellen lässt, wobei $E \in \mathbb{R}^{m \times m}$ wiederum die Einheitsmatrix bezeichnet. Mit Hilfe von (9.12) lässt sich dann aber auch die Darstellung

$$v_j = \left(E - q_{j-1} q_{j-1}^T \right) \cdot \ldots \cdot \left(E - q_2 q_2^T \right) \cdot \left(E - q_1 q_1^T \right) \cdot a_j \tag{12.21}$$

ableiten. Basierend auf (12.21) berechnet das modifizierte Gram-Schmidt Verfahren den Vektor v_j nun Schritt für Schritt nach folgendem Schema

$$\begin{aligned} v_j^{(1)} &= a_j, \\ v_j^{(2)} &= \left(E - q_1 q_1^T \right) \cdot v_j^{(1)} = v_j^{(1)} - \langle q_1, v_j^{(1)} \rangle q_1 \\ &\vdots \quad\quad \vdots \\ v_j = v_j^{(j)} &= \left(E - q_{j-1} q_{j-1}^T \right) \cdot v_j^{(j-1)} = v_j^{(j-1)} - \langle q_{j-1}, v_j^{(j-1)} \rangle q_{j-1}. \end{aligned}$$

Berücksichtigt man noch, dass $\left(E - q_i \cdot q_i^T \right)$ auf alle $v_j^{(i)}$ mit $j > i$ angewendet werden kann, sobald q_i ermittelt[16] wurde, so führt dies auf den Algorithmus *GramSchmidtMod*.

Algorithmus *GramSchmidtMod*: Modifiziertes Orthonormalisierungsverfahren in $\mathbb{R}^m$

```
for i from 1 to n
    v_i ← a_i
for i from 1 to n
    r_ii ← ‖v_i‖_2
    q_i ← v_i / r_ii
    for j from i + 1 to n
        r_ij ← ⟨q_i, v_j⟩
        v_j ← v_j − r_ij · q_i
return q
```

Aufruf: *GramSchmidtMod*(a)
Eingabe: $a \in (\mathbb{R}^m)^n$
mit: a linear unabhängig.
Ausgabe: $q \in (\mathbb{R}^m)^n$
mit: q orthonormal und
$\text{span}(q_1, \ldots, q_j) = \text{span}(a_1, \ldots, a_j)$ für
$j = 1, \ldots, n$.

Auch bei *GramSchmidtMod* ist der Aufwand durch $O(mn^2)$ Elementaroperationen $+, -, \cdot, /$ gegeben, siehe Übungsaufgabe III.5. In der Praxis wird man auch hier a_j mit v_j und anschließend v_j mit q_j überschreiben. Der infolge von Gleitkommaarithmetik entstehende Rundungsfehler ist bei *GramSchmidtMod* jedoch deutlich kleiner als bei *GramSchmidt*.

[16] Sobald q_i zur Verfügung steht, werden alle also verbleibenden Vektoren gegen q_j orthonormalisiert.

Beispiel

Wir betrachten erneut das Beispiel von Seite 112, verwenden nun jedoch den Algorithmus *GramSchmidtMod*. Für $\tilde{q}_{13}$ und $\tilde{q}_{15}$ aus der Resultatsmenge $(\tilde{q}_1,\ldots,\tilde{q}_{15})$ gilt dann $\langle\tilde{q}_{13},\tilde{q}_{15}\rangle = -1.9379\cdot 10^{-13}$, der betragsmäßig größte Wert eines Skalarprodukts verschiedener Vektoren ist durch $\langle\tilde{q}_1,\tilde{q}_{15}\rangle = 4.0097\cdot 10^{-7}$ gegeben.

Um einen Einblick in das Stabilitätsverhalten der Gram-Schmidt Verfahren zu erhalten, folgen wir [6] und betrachten den Fall $n = 2$, für den *GramSchmidt* und *GramSchmidMod* identisch sind. Zu den Vektoren $a_1, a_2 \in \mathbb{R}^m$ ist zunächst der Vektor $q_1 = a_1/\|a_1\|_2$ und daraus der Vektor

$$v_2 = a_2 - \langle q_1, a_2\rangle \cdot q_1$$

zu bilden. Der Einfachheit halber nehmen wir an, dass bei der Eingabe der Daten keine Rundungsfehler entstehen und q_1 exakt berechnet wird. Nach dem Satz von Seite 109 führt der Aufruf *SkalarproduktEuklid*(q_1, a_2) in Gleitkommaarithmetik zu einem Ergebnis $\langle q_1, \hat{a}_2\rangle$ mit

$$\frac{|\hat{a}_{2,i} - a_{2,i}|}{|a_{2,i}|} \lesssim (m-i+2)\mathbf{u} \quad \text{für } \mathbf{u} \to 0, \tag{12.22}$$

wobei $a_{2,i}$ die i-te Komponente von a_2 bezeichnet. Anstelle von $v_{2,i}$ erhalten wir somit

$$\tilde{v}_{2,i} = (a_{2,i} - \langle q_1, \hat{a}_2\rangle q_{1,i}(1+\varepsilon_{*,i}))(1+\varepsilon_{-,i}),$$

worin $\varepsilon_{*,i}$, $\varepsilon_{-,i}$ die Rundungsfehler der Multiplikation bzw. Subtraktion sind. Aus $\langle q_1, \hat{a}_2\rangle = \langle q_1, a_2\rangle + \langle q_1, \hat{a}_2 - a_2\rangle$ folgt die Darstellung

$$\begin{aligned}\tilde{v}_{2,i} \quad = \quad & v_{2,i} - \langle q_1, \hat{a}_2 - a_2\rangle q_{1,i} - \varepsilon_{*,i}\langle q_1, \hat{a}_2\rangle q_{1,i} + \varepsilon_{-,i} a_{2,i} \\ & -\varepsilon_{-,i}(1+\varepsilon_{*,i})\langle q_1, \hat{a}_2\rangle q_{1,i}.\end{aligned}$$

Betrachtet man das innere Produkt von $\tilde{v}_2$ mit q_1, so folgt wegen $\langle v_2, q_1\rangle = 0$, $\langle q_1, q_1\rangle = 1$ und der Cauchy-Schwarz Ungleichung (9.3)

$$\begin{aligned}|\langle\tilde{v}_2, q_1\rangle| \quad &\leq \quad |\langle q_1, \hat{a}_2 - a_2\rangle| + \mathbf{u}|\langle q_1, \hat{a}_2\rangle| + \mathbf{u}|\langle a_2, q_1\rangle| + \mathbf{u}(1+\mathbf{u})|\langle q_1, \hat{a}_2\rangle| \\ &\leq \quad \|\hat{a}_2 - a_2\|_2 + 2\mathbf{u}\|\hat{a}_2\|_2 + \mathbf{u}\|a_2\|_2 + \mathbf{u}^2\|\hat{a}_2\|_2.\end{aligned}$$

Aus (12.22) mit $i = 1$, $\mathbf{u}\|\hat{a}_2\|_2 \approx \|a_2\|_2$, $\|\tilde{v}_2\|_2 \approx \|v_2\|_2$ für $\mathbf{u} \to 0$ und Übungsaufgabe III.6 folgt dann

$$|\langle q_1, \frac{\tilde{v}_2}{\|\tilde{v}_2\|_2}\rangle| \lesssim \mathbf{u}(m+4)\frac{\|a_2\|_2}{\|v_2\|_2} = \mathbf{u}\frac{m+4}{\sin(\angle_{a_1,a_2})} \quad \text{für } \mathbf{u} \to 0.$$

Die obere Schranke für die Verletzung der Orthogonalität infolge der Rundungsfehler ist also proportional zur Rundungseinheit $\mathbf{u}$ und dem Faktor $1/\sin(\angle_{a_1,a_2})$, der umso kleiner ist, je stärker die lineare Unabhängigkeit von a_1 und a_2 ist. Für *GramSchmidtMod* lassen sich diese Überlegungen auf den Fall $n > 2$ übertragen

und die numerische Rückwärtsstabilität zeigen. Bei Gleitkommarealisierung liefert *GramSchmidtMod* dann also eine Familie $\tilde{q} \in (\mathbb{R}^m)^n$, die sich als exakte Lösung des Problems zu benachbarten Eingangsdaten $\hat{a} \in (\mathbb{R}^m)^n$ darstellen lässt. Ausserdem ist der Verlust der Orthonormalität wieder durch **u** mal einem Faktor, der invers proportional zu einem Maß für die lineare Unabhängigkeit von $a \in (\mathbb{R}^m)^n$ ist, beschränkt, siehe [6]. Für *GramSchmidt* ist für $n > 2$ eine Beschränkung des Orthogonalitätsverlustes in diesem Sinne nicht möglich, daher auch der große Fehler im Beispiel von Seite 112. Ist die Familie $a \in (\mathbb{R}^m)^n$ nahezu linear unabhängig, das Problem also schlecht konditioniert, so kann es natürlich auch bei *GramSchmidtMod* zu stark fehlerhalten Resultaten kommen.

Übungsaufgaben

III.1 Zeigen Sie, dass in jedem Vektorraum V_K eine linear unabhängige Familie mit $\dim(V_K)$ Vektoren stets eine Basis von V_K bildet, eine Familie von mehr als $\dim(V_K)$ Vektoren immer linear abhängig ist.

III.2 Zeigen Sie, dass die Vektoren $a_1 = (1\ 1\ 1\ -1)^T$, $a_2 = (2\ -1\ -1\ 1)^T$, $a_3 = (-1\ 2\ 2\ 1)^T \in \mathbb{R}^4$ linear unabhängig sind.

III.3 Seien q ein normierter Vektor in $\mathbb{R}^m$. Zeigen Sie, dass für alle $v \in \mathbb{R}^m$ der Vektor $(E - q \cdot q^T) \cdot v$ orthogonal zu q ist.

III.4 Entwickeln Sie einen Algorithmus zur Berechnung der Maximumsnorm auf $\mathbb{R}^m$ und beweisen Sie seine Korrektheit.

III.5 Zeigen Sie, dass der Aufwand von *GramSchmidt* und *GramSchmidtMod* jeweils $O(mn^2)$ Elementaroperationen $+, -, \cdot, /$ beträgt.

III.6 Seien $a, q \in \mathbb{R}^m$ mit $\|q\|_2 = 1$. Zeigen Sie, dass mit $v = a - \langle q, a\rangle q$ die Gleichung $\frac{\|a\|_2}{\|v\|_2} = \frac{1}{\sin(\angle_{a,q})}$ gilt.

III.7 Überzeugen Sie sich unter Verwendung von *Mathematica* davon, dass die Vektoren $a_j = (0^{j-1}\ (\frac{1}{24})^{j-1}\ (\frac{2}{24})^{j-1}\ \ldots\ (\frac{23}{24})^{j-1}\ 1)^T \in \mathbb{R}^{25}$ für $j = 1, \ldots, 15$ eine lineare unabhängige Familie $a \in (\mathbb{R}^{25})^{15}$ bilden.

IV Univariate Polynome

In diesem Kapitel beschäftigen wir uns mit den Grundlagen zu univariaten Polynomen. Diese stellt man sich häufig als Summe von Potenzen *einer* Variable vor, auf der dann auch die Termdarstellung der Polynomfunktionen beruht. So besteht etwa ein Naheverhältnis zwischen dem Polynom $7-x+2x^3$ und der Funktion $x \mapsto 7-x+2x^3$. Wir erklären, wie man mit Polynomen rechnet und mit diesen am Computer umgeht. Insbesondere gehen wir auf die Polynomdivision mit Rest und die Bestimmung des größten gemeinsamen Teilers näher ein. Weiters stellen wir Algorithmen zur Polynomauswertung und zur klassischen Polynominterpolation vor.

Vereinbarung. Wie in Kapitel III bezeichne auch hier die Variable K einen Körper.

13 Mathematische Grundlagen

Polynome sollen Ausdrücke der Form $p_0 + p_1x + p_2x^2 + \cdots + p_nx^n$ oder – etwas formaler – Ausdrücke der Form $\sum_{i=0}^{n} p_ix^i$ darstellen. Was sind dabei die p_i's, was ist n, aber vor allem: Was ist das x in einem Polynom? Ist das Polynom $1 - x^2$ identisch mit dem Polynom $1 - y^2$? In Kürze werden wir sehen, dass ein Polynom allein durch die *Koeffizienten* $p_0, \ldots, p_n$ ausreichend bestimmt ist, und dass wir alle Operationen auf Polynomen auch durch entsprechende Operationen auf den Koeffizententupeln beschreiben können. Mit jedem solchen Polynom p assoziieren wir eine zugehörige *Polynomfunktion,* indem wir deren Funktionswert an jeder Stelle x durch eine spezielle Polynomoperation, das *Evaluieren* von p an der Stelle x, definieren.

Univariate Polynome und Polynomarithmetik

Wir beginnen unsere Diskussion mit der Definition des Begriffs *Polynom*.

Definition

Polynom, Koeffizientenbereich, Koeffizient. Wir nennen p ein *univariates Polynom über K* genau dann, wenn p eine Folge in K ist, die ab einem bestimmten Index nur 0 enthält. Wir nennen K den *Koeffizientenbereich* des Polynoms und die Folgenelemente von p die *Koeffizienten* von p.

Die in der Mathematik übliche Bezeichnung der Menge der univariaten Polynome über K lautet $K[\mathrm{x}]$, und wir sprechen von „Polynomen in x über K“. Das Symbol x

hat an dieser Stelle noch keine tiefere Bedeutung, wir werden seine Rolle etwas später klären.

Beispiel

Die Folge $\left(2,\frac{1}{5},0,\ldots\right)$ ist ein Polynom (über $\mathbb{R}$ oder $\mathbb{Q}$) mit den Koeffizienten $2,\frac{1}{5},0,\ldots$ und $(3,5,-1,0,2,0,\ldots)$ ist ein Polynom (über $\mathbb{R}$ oder $\mathbb{Q}$) mit Koeffizienten $3,5,-1,0,2,0,\ldots$.

Ist p ein Polynom über K, also eine spezielle Folge in K, so entsprechen die Folgenelemente p_i den Polynomkoeffizienten. Bei Polynomen ist es üblich, die Indizierung bei 0 zu beginnen, d.h. der erste Koeffizient von p ist p_0, der zweite ist p_1 etc. Eine besondere Rolle kommt jenem Polynom zu, in dem alle Koeffizienten gleich 0 sind. Wir nennen es das *Nullpolynom* und schreiben dafür 0. In jedem Polynom gibt es einen minimalen Index, ab dem alle Koeffizienten 0 sind.

Definition

Grad, Führender Koeffizient, Normiertheit. Sei p ein Polynom über K und sei $d \in \mathbb{N}_0$ der minimale Index, mit dem $p_i = 0$ für alle $i \geq d$ gilt. Dann definieren wir

$$\deg(p) := d - 1,$$

und wir nennen $\deg(p)$ den *Grad* des Polynoms p. Weiters sei

$$K[x]^n := \left\{p \in K[x] \;\middle|\; \deg(p) \leq n\right\}.$$

Für $p \neq 0$ nennen wir $p_{\deg(p)} \neq 0$ den *führenden Koeffizienten* von p und schreiben $\mathrm{fk}(p)$. Ist $\mathrm{fk}(p) = 1$, so nennen wir p *normiert*. Wir vereinbaren $\mathrm{fk}(0) := 0$. Polynome von Grad 0, 1, 2, 3 heißen auch konstante, lineare, quadratische bzw. kubische Polynome.

Mit deg und fk haben wir die ersten *Grundoperationen* auf Polynomen eingeführt, die wir nun anhand eines Beispiels illustrieren wollen.

Beispiel

p	$\deg(p)$	$\mathrm{fk}(p)$
$\left(2,\frac{1}{5},0,\ldots\right)$	1	$\frac{1}{5}$
$(3,5,-1,0,2,0,\ldots)$	4	2

Zur Definition von $\deg(0)$. Mit obiger Definition gilt $\deg(0) = -1$, dennoch werden wir auch 0 als konstantes Polynom bezeichnen. Der Grad des Nullpolynoms wird manchmal auch als $-\infty$ oder überhaupt nicht definiert. Je nach Definition kann man sich in der Formulierung von Sätzen oder Algorithmen Spezialfälle ersparen oder muss 0 als Spezialfall gesondert behandeln. Daher verdient bei Aussagen über den Grad von Polynomen das Nullpolynom *immer* spezielle Aufmerksamkeit.

Zwei univariate Polynome über K sind genau dann gleich, wenn alle Koeffizienten übereinstimmen, d.h.

$$p = q \iff \big(\deg(p) = \deg(q) \quad \text{und} \quad p_i = q_i \text{ für } i = 0,\ldots,\deg(p)\big).$$

Diese Eigenschaft nutzen wir beim sogenannten *Koeffizientenvergleich,* wenn wir die Gleichheit zweier Polynome p und q auf ein System von Gleichheiten aller Koeffizienten reduzieren.

Auf Basis der zugrunde gelegten Rechenoperationen in K (Addition, Multiplikation, etc.) können wir nun *Rechenoperationen auf Polynomen* definieren. Die Addition definieren wir dabei wie für Folgen üblich *komponentenweise,* die Multiplikation weist eine auf den ersten Blick etwas ungewohnte Form auf.

Definition

Polynomarithmetik in $K[\mathrm{x}]$. Seien $p,q \in K[\mathrm{x}]$.

$$p+q := (s_0,s_1,\ldots) \quad \text{mit } s_i := p_i + q_i \text{ für } i \in \mathbb{N}_0$$

$$p \cdot q := (r_0,r_1,\ldots) \quad \text{mit } r_i := \sum_{j=0}^{i} p_j q_{i-j} \text{ für } i \in \mathbb{N}_0.$$

Mit der so definierten Addition und Multiplikation wird $K[\mathrm{x}]$ selbst zu einem *kommutativen Ring mit Einselement,* insbesondere gilt $p+q \in K[\mathrm{x}]$ und $p \cdot q \in K[\mathrm{x}]$. Das Nullpolynom 0 ist das neutrale Element bzgl. der Addition, und das Inverse zu $p = (p_0,\ldots,p_n,0,\ldots)$ bzgl. der Addition ist $-p = (-p_0,\ldots,-p_n,0,\ldots)$. Damit ist die Subtraktion $p-q$ einfach als $p+(-q)$ zu definieren, sie erfolgt analog zur Addition durch komponentenweises Subtrahieren. Das Polynom $1 := (1,0,\ldots)$ ist das neutrale Element bzgl. der Multiplikation. Für $p,q \neq 0$ gilt immer

$$\deg(p \cdot q) = \deg(p) + \deg(q). \tag{13.1}$$

$K[\mathrm{x}]$ und $K[\mathrm{x}]^n$ weisen auch die Struktur eines *Vektorraums über K* auf. Insbesondere ist $K[\mathrm{x}]$ ein Beispiel eines unendlichdimensionalen Vektorraums über K, siehe Seite 102, die Dimension von $K[\mathrm{x}]^n$ ist $n+1$. Bei gegebener Basis b kann jedes $p \in K[\mathrm{x}]^n$ als Linearkombination

$$p = \sum_{i=0}^{n} \lambda_i \cdot b_i \quad \text{mit } \lambda_i \in K, \tag{13.2}$$

dargestellt werden[1]. Die kanonischen Basen von $K[\mathrm{x}]$ bzw. $K[\mathrm{x}]^n$ lauten

$$\begin{aligned}(b_0,b_1,\ldots,b_n,\ldots) &= ((1\,0\,\ldots),(0\,1\,0\,\ldots),\ldots,(0\ldots0\,1\,0\ldots),\ldots)\\ (b_0,b_1,\ldots,b_n) &= ((1\,0\,\ldots),(0\,1\,0\,\ldots),\ldots,(0\ldots0\,1\,0\ldots))\,.\end{aligned}$$

Die skalare Multiplikation im Vektorraum kann als Spezialfall der Polynommultiplikation gesehen werden, indem wir einen Skalar $\lambda \in K$ mit dem konstanten Polynom $(\lambda,0,\ldots)$ identifizieren. Da K ein Körper ist, können wir mit Hilfe der skalaren Multiplikation auch ein Polynom durch einen Skalar $\mu \neq 0$ dividieren. Eine Division zweier Polynome in $K[\mathrm{x}]$ ist aber nicht möglich, da ein Polynom im Allgemeinen *nicht* bzgl. der Multiplikation *invertierbar* ist. $K[\mathrm{x}]$ ist also *kein Körper,* und damit ist die in Körpern übliche Division durch Multiplikation mit dem Inversen nicht

[1] Auch im Fall des unendlichdimensionalen Raums $K[\mathrm{x}]$ basiert die Basisdarstellung auf einer endlichen Summe, siehe den Satz zur Basisdarstellung von Vektoren von Seite 102.

verfügbar. Eine Division mit Rest ähnlich jener in $\mathbb{Z}$ ist dagegen definiert, wir werden diese in Abschnitt 15 besprechen.

Beispiel

Für $p = \left(2,\frac{1}{5},0,\ldots\right)$ und $q = (3,5,-1,0,2,0,\ldots)$ aus $\mathbb{R}[x]$ erhalten wir nach obigen Definitionen

$p+q$	$p \cdot q$	$3p$	$p/2 \quad (=\frac{1}{2}p)$
$\left(5,\frac{26}{5},-1,0,2,0,\ldots\right)$	$\left(6,\frac{53}{5},-1,-\frac{1}{5},4,\frac{2}{5},0,\ldots\right)$	$\left(6,\frac{3}{5},0,\ldots\right)$	$\left(1,\frac{1}{10},0,\ldots\right).$

Bezeichnung

Bezeichnet $K[x]$ die Menge der univariaten Polynome über K, so ist

$$p_0 + p_1 x + \cdots + p_n x^n = \sum_{i=0}^{n} p_i x^i \tag{13.3}$$

die meist verwendete Schreibweise für ein Polynom p von Grad n. Bei x handelt es sich dabei lediglich um eine Bezeichnung für ein *ganz bestimmtes Polynom*, nämlich $x = (0,1,0,\ldots)$. Die Definitionen von Polynomaddition und -multiplikation rechtfertigen die $\sum$-Schreibweise (13.3) für Polynome, da die Polynomarithmetik genau den gleichen Regeln gehorcht wie das Addieren und Ausmultiplizieren von Summen. In diesem Licht erinnert die Polynommultiplikation auch an das *Cauchy-Produkt* für Reihen.

Das Polynom x spielt auch eine ganz spezielle Rolle bei der Multiplikation in $K[x]$. So, wie die Multiplikation einer zur Basis B dargestellten ganzen Zahl mit B^k nichts als eine Verschiebung der Ziffern um k Stellen ist, siehe Abschnitt 5, bewirkt die Multiplikation eines Polynoms mit x^k eine Verschiebung der Koeffizienten um k Stellen mit entsprechender Auffüllung durch Nullen[2]. Insbesondere ist

$$x^2 = x \cdot x = (0,0,1,0,\ldots) \qquad x^3 = x^2 \cdot x = (0,0,0,1,0,\ldots) \qquad \text{etc.}$$

Eine Multiplikation mit x^k in $K[x]$ kann daher sehr effizient durch die rein syntaktische Operation des „vorne Einfügens von k Nullen“ in der Koeffizientenfolge realisiert werden. Die kanonischen Basen von $K[x]$ bzw. $K[x]^n$ lauten in dieser Notation

$$\left(1,x,x^2,\ldots\right) \quad \text{bzw.} \quad \left(1,x,x^2,\ldots,x^n\right), \tag{13.4}$$

und die bisher nur als Schreibweise eingeführte $\sum$-Notation (13.3) entspricht genau der in (9.2) gezeigten Basisdarstellung von p im Vektorraum $K[x]$ oder $K[x]^n$ bzgl. der kanonischen Basis (13.4). Setzen wir darüberhinaus $x^0 := 1 = (1,0,\ldots)$, so gilt

[2] Man beachte an dieser Stelle, dass in Abschnitt 5 die Ziffern einer Zahl zur Basis B nach absteigenden Potenzen von B angeordnet sind, die Koeffizienten eines Polynoms jedoch aufsteigenden Potenzen von x zugeordnet sind. Im Vergleich zur Operation *Verschiebe*$_{Z_B}$ von Seite 61 müssen daher bei Polynomen die Koeffizienten *nach rechts* verschoben und die Folge *am Beginn* mit Nullen aufgefüllt werden.

mit den in $K[\mathtt{x}]$ definierten Rechenregeln

$$\sum_{i=0}^{n} p_i \mathtt{x}^i = p_0\mathtt{x}^0 + p_1\mathtt{x}^1 + \cdots + p_n\mathtt{x}^n = (p_0,\ldots,p_n,0,\ldots)\,.$$

Dies rechtfertigt es nun endgültig, das Polynom $(3,5,-1,0,2,0,\ldots)$ als $3+5\mathtt{x}-\mathtt{x}^2+2\mathtt{x}^4$ zu schreiben.

In $K[\mathtt{x}]$ ist $1-\mathtt{x}^2$ also eine Schreibweise für das Polynom $(1,0,-1,0,\ldots)$. Anstelle des Symbols $\mathtt{x}$ können wir auch jedes andere Symbol verwenden, so bezeichnet $1-\mathtt{y}^2$ wieder das Polynom $(1,0,-1,0,\ldots)$, dann allerdings in $K[\mathtt{y}]$. Als univariate Polynome betrachten wir $1-\mathtt{x}^2$ und $1-\mathtt{y}^2$ daher als gleich. Die Mengen $K[\mathtt{x}]$ und $K[\mathtt{y}]$ sind ebenfalls gleich, die Symbole $\mathtt{x}$ bzw. $\mathtt{y}$ weisen lediglich auf eine unterschiedliche Schreibweise für Polynome hin. Wir stellen in Band *2* auch den Bereich $K[\mathtt{x},\mathtt{y}]$ der multivariaten Polynome über K vor, also solche, in denen sowohl $\mathtt{x}$ als auch $\mathtt{y}$ vorkommen kann. In $K[\mathtt{x},\mathtt{y}]$ bezeichnen $1-\mathtt{x}^2$ und $1-\mathtt{y}^2$ dann unterschiedliche Polynome.

Bemerkung. Es sei an dieser Stelle erwähnt, dass Polynome auch über einem Koeffizientenbereich R definiert werden können, wenn R kein Körper ist. $R[\mathtt{x}]$ ist dann aber kein Vektorraum mehr. Oft wird nur verlangt, dass R ein (kommutativer) Ring (mit Einselement) ist, $R[\mathtt{x}]$ ist dann ebenfalls ein (kommutativer) Ring (mit Einselement). Als prominentestes Beispiel dafür führen wir $\mathbb{Z}[\mathtt{x}]$ an.

Polynomauswertung und Polynomfunktionen

In der Summenschreibweise $\sum_{i=0}^{n} p_i\mathtt{x}^i$ eines Polynoms wird $\mathtt{x}$ oft die *Polynomvariable* genannt, weil wir durch Ersetzen von $\mathtt{x}$ mit einem Wert $w \in K$ dem Polynom einen Wert an jeder Stelle w zuordnen können.

Definition

Polynomauswertung. Sei $p \in K[\mathtt{x}]$ und $w \in K$. Dann ist die Operation der *Polynomauswertung* (oder *Polynomevaluation*) definiert durch

$$\mathrm{eval}(p,w) := \sum_{i=0}^{\deg(p)} p_i w^i,$$

und wir nennen $\mathrm{eval}(p,w) \in K$ den *Wert* von p *an der Stelle* w.

Beispiel

Die Auswertung von $(3,5,-1,0,2,0,\ldots) \in \mathbb{R}[\mathtt{x}]$ an der Stelle $2 \in \mathbb{R}$ liefert

$$\mathrm{eval}((3,5,-1,0,2,0,\ldots),2) = 3\cdot 2^0 + 5\cdot 2^1 - 1\cdot 2^2 + 0\cdot 2^3 + 2\cdot 2^4 = 41.$$

Durch einfaches Nachrechnen lässt sich der folgende Satz zeigen.

Satz

Evaluations-Homomorphismus. Für jedes $w \in K$ ist die Evaluation an der Stelle w ein Homomorphismus von $K[\mathrm{x}]$ nach K, d.h.

$$\begin{aligned} \mathrm{eval}(p+q,w) &= \mathrm{eval}(p,w) + \mathrm{eval}(q,w) && (13.5)\\ \mathrm{eval}(p\cdot q,w) &= \mathrm{eval}(p,w)\cdot \mathrm{eval}(q,w). && (13.6) \end{aligned}$$

Zu jedem Polynom $p \in K[\mathrm{x}]$ kann man eine *Funktion* von K nach K definieren, deren Funktionswerte sich *durch Auswerten* von p ergeben.

Definition

Polynomfunktion. Sei $p \in K[\mathrm{x}]$. Wir nennen

$$\begin{aligned} \mathrm{pf}_p : K &\to K \\ x &\mapsto \mathrm{eval}(p,x) \end{aligned}$$

die *zu p gehörige Polynomfunktion (über K)*. Wir vereinbaren

$$\Pi_K := \{\mathrm{pf}_p \mid p \in K[\mathrm{x}]\} \quad \text{und} \quad \Pi_K^n := \{\mathrm{pf}_p \mid p \in K[\mathrm{x}], \deg(p) \le n\}$$

für Mengen von Polynomfunktionen über K.

Die Termdarstellung einer Polynomfunktion in Π_K^n lautet somit

$$\mathrm{pf}_p(x) = p_0 + p_1 x + \cdots + p_n x^n = \sum_{i=0}^{n} p_i x^i.$$

Beispiel

Für $p = (3,5,-1,0,2,0,\ldots) \in \mathbb{R}[\mathrm{x}]$ ist $\mathrm{pf}_p : \mathbb{R} \to \mathbb{R},\ x \mapsto 3 + 5x - x^2 + 2x^4$ die zu p gehörige Polynomfunktion pf_p.

Polynomfunktionen sind als Abbildungen von K nach K etwas grundsätzlich anderes als Polynome, die ja (spezielle) Folgen über K und damit Abbildungen von $\mathbb{N}_0$ nach K sind. Per Definition gibt es zu jedem Polynom eine eindeutig bestimmte Polynomfunktion, umgekehrt kann eine Funktion zu mehreren Polynomen gehören.

Beispiel

Für $K = \mathbb{Z}_2$ sind $p = \mathrm{x} + \mathrm{x}^2$ und $q = \mathrm{x} + \mathrm{x}^3$ *verschiedene Polynome* über $\mathbb{Z}_2$. Die dazugehörigen Polynomfunktionen sind jedoch wegen

x	$\mathrm{pf}_p(x)$
0	0
1	0

x	$\mathrm{pf}_q(x)$
0	0
1	0

identisch, sie entsprechen beide der Nullfunktion, sind also auch z.B. zu pf_0 identisch.

Über endlichem K kann es eine umkehrbar eindeutige Zuordnung nicht geben, da es dann zwar unendlich viele Polynome über K aber nur endlich viele Funktionen von K nach K, insbesondere also nur endlich viele Polynomfunktionen, gibt. Ist K jedoch ein unendlicher Körper (z.B. $\mathbb{R}$ oder $\mathbb{Q}$), so ist $K[x]$ zu Π_K isomorph, d.h. Polynome und Polynomfunktionen entsprechen einander umkehrbar eindeutig, und in beiden Bereichen wird „gleich gerechnet“. So gilt dann etwa für $p,q \in K[x]$ unter Verwendung von (13.5) für die Funktionsaddition

$$\begin{aligned}(\mathrm{pf}_p + \mathrm{pf}_q)(x) &= \mathrm{pf}_p(x) + \mathrm{pf}_q(x) = \mathrm{eval}(p,x) + \mathrm{eval}(q,x)\\ &= \mathrm{eval}(p+q,x) = \mathrm{pf}_{p+q}(x),\end{aligned} \tag{13.7}$$

d.h. die Summe der zu p bzw. q gehörigen Polynomfunktionen ist genau die zu $p+q$ gehörige Polynomfunktion. Begriffe wie Grad oder Koeffizient eines Polynoms verwenden wir dann auch für die zugehörige Polynomfunktion.

In den univariaten Polynomen $K[x]$ spiegeln sich viele Eigenschaften der ganzen Zahlen $\mathbb{Z}$ wider. Wir werden diesem Phänomen auch in späteren Abschnitten, etwa bei der Polynommultiplikation am Computer, bei der Division mit Rest und dem größtem gemeinsamen Teiler noch begegnen. Auch die Erweiterung von $\mathbb{Z}$ auf die rationalen Zahlen $\mathbb{Q}$ hat in $K[x]$ eine Analogie. In gleicher Weise, wie wir in Abschnitt 7 Bruchzahlen als Äquivalenzklassen von Paaren ganzer Zahlen eingeführt haben, können auch *Polynombrüche* als Äquivalenzklassen von Paaren univariater Polynome definiert werden. Die Rechenregeln für $\mathbb{Q}$ gelten in gleicher Weise auf Polynombrüchen, und auch die Henrici Algorithmen für Addition und Multiplikation lassen sich einfach übertragen. So, wie durch Auswerten eines Polynoms eine Polynomfunktion definiert ist, sind *rationale Funktionen* durch Auswerten der beiden Polynome in einem Polynombruch festgelegt, wobei diese Funktionen nur an Stellen definiert sind, an denen der Nenner nicht zu 0 evaluiert.

14 Polynome am Computer

Für die Darstellung von Polynomen am Computer benötigen wir eine Datenstruktur, in der wir Polynome aus $K[x]$, also spezielle unendliche Folgen, in „endlicher Form“ beschreiben können. Auf diese Datenstruktur übertragen wir dann alle bisher in $K[x]$ definierten Polynomoperationen.

Computerrepräsentation (Datenstruktur $\mathcal{P}_K$ für Polynome über K)**.** Ein univariates Polynom $p = (p_0,\ldots,p_n,0,\ldots) \in K[x]$ von Grad n ist durch das Tupel $t \in K^{n+1}$ mit

$$t_i = p_{i-1} \quad \text{für } i = 1,\ldots,n+1 \tag{14.8}$$

charakterisiert[3]. Wir führen daher für Polynome über K eine Datenstruktur $\mathcal{P}_K$ ein, in der wir für p das in (14.8) definierte Tupel t abspeichern. Für $p \in \mathcal{P}_K$ sprechen wir das Koeffiziententupel t mit $\mathrm{koef}(p)$ an. Der führende Koeffizient steht an der

[3]Wir erinnern daran, dass das erste Element in einem wie auf Seite 47 eingeführten Tupel t mit t_1 angesprochen wird.

letzten Position im Koeffiziententupel, das Nullpolynom 0 enthält ein leeres Koeffiziententupel, und $|p|$ bezeichnet die Länge des in p gespeicherten Tupels t. Zum Anschreiben eines Polynoms $p \in \mathcal{P}_K$ verwenden wir einfach das Tupel t.

Für $p \in \mathcal{P}_K$ und $t = \text{koef}(p)$ können die Grundoperationen deg und fk durch

$$\deg(p) := |t| - 1$$
$$\text{fk}(p) := \begin{cases} 0 & \text{falls } p = 0 \\ t_{|t|} & \text{falls } p \neq 0 \end{cases}$$

realisiert werden. Die einzelnen Koeffizienten wollen wir auch für $p \in \mathcal{P}_K$ einfach mit p_i für $i \in \mathbb{N}_0$ ansprechen. Dies erreichen wir durch

$$p_i := \begin{cases} t_{i+1} & \text{falls } i \in \{0, \ldots, \deg(p)\} \\ 0 & \text{falls } i > \deg(p). \end{cases}$$

Weiters vereinbaren wir in Anlehnung an die Schreibweise $K[x]^n$

$$\mathcal{P}_K^n := \{p \in \mathcal{P}_K \mid \deg(p) \leq n\}.$$

Neben der hier vorgestellten *dichten Darstellung* gibt es auch noch andere Darstellungsweisen für Polynome, etwa die *dünne Darstellung* für *dünn besetzte Polynome,* d.h. solche, in denen viele Koeffizienten gleich 0 sind, z.B. $x^2 - x^{100}$. In einer dünnen Darstellung finden nur die Koeffizienten ungleich 0 Eingang in die Datenstruktur, und obiges Polynom wäre durch $((1,2),(-1,100))$ repräsentiert. Das Nullpolynom ist auch hier wieder durch das leere Tupel dargestellt, auch die Bestimmung des Grads (deg) und der Zugriff auf einzelne und den führenden Koeffizienten (koef, fk) ist einfach realisierbar. Weiters kann es sinnvoll sein, für spezielle Problemstellungen maßgeschneiderte Datenstrukturen für Polynome zu entwickeln, siehe Seite 143 im Abschnitt 17 über Polynominterpolation für ein Beispiel dazu.

Wir wollen nun die oben für $K[x]$ eingeführten arithmetischen Grundoperationen derart auf $\mathcal{P}_K$ übertragen, dass Addition, Multiplikation etc. für in der Datenstruktur $\mathcal{P}_K$ dargestellte Polynome genau den oben definierten Operationen entsprechen, und alle Eigenschaften der Verknüpfungen erhalten bleiben.

Computerrepräsentation (Polynomarithmetik in $\mathcal{P}_K$)**.** Wir müssen bei der Definition der Rechenoperationen für Polynome sicherstellen, dass das resultierende Polynom wieder die in der Datenstruktur verlangte kanonische Form hat. Dazu verwenden wir eine Funktion namens $\textit{kanonisch}_{\mathcal{P}_K}$, die allenfalls am Ende eines Tupels auftretende Nullen eliminiert. Für $a = ()$ oder ein Tupel a über K der Länge n mit $n \in \mathbb{N}$ definieren wir rekursiv

$$\text{kanonisch}_{\mathcal{P}_K}(a) := \begin{cases} \text{kanonisch}_{\mathcal{P}_K}(a_{1:n-1}) & \text{falls } a \neq () \text{ und } a_n = 0 \\ a & \text{sonst.} \end{cases}$$

Darauf aufbauend definieren wir für $p,q \in \mathcal{P}_K$ und $\lambda,\mu \in K$ mit $\mu \neq 0$

$$p \pm q := s \in \mathcal{P}_K \text{ mit } \mathrm{koef}(s) = \mathrm{kanonisch}_{\mathcal{P}_K}\big((p_i \pm q_i)_{i=0,\ldots,\max(\deg(p),\deg(q))}\big),$$

$$p * q := \begin{cases} 0 & \text{falls } p = 0 \vee q = 0 \\ s \in \mathcal{P}_K \text{ mit } \mathrm{koef}(s) = (\sum_{j=0}^{i} p_j q_{i-j})_{i=0,\ldots,\deg(p)+\deg(q)} & \text{sonst,} \end{cases}$$

$$\lambda \cdot p := \begin{cases} 0 & \text{falls } \lambda = 0 \\ s \in \mathcal{P}_K \text{ mit } \mathrm{koef}(s) = (\lambda p_i)_{i=0,\ldots,\deg(p)} & \text{falls } \lambda \neq 0, \end{cases}$$

$$p/\mu := s \in \mathcal{P}_K \text{ mit } \mathrm{koef}(s) = (p_i/\mu)_{i=0,\ldots,\deg(p)}.$$

Obwohl die skalare Multiplikation als Spezialfall der Polynommultiplikation betrachtet werden kann, definieren wir einzelne Operationen $p * q, \lambda \cdot p$, da sich auch am Computer eine separate Realisierung empfiehlt. Wegen (13.1) ist die Verwendung von $\mathrm{kanonisch}_{\mathcal{P}_K}$ bei $p * q$ nicht notwendig.

Beispiel

Für $p = \left(2, \frac{1}{5}\right)$ und $q = (3,5,-1,0,2)$ aus $\mathcal{P}_{\mathbb{R}}$ erhalten wir

$p+q$	$p-q$	$p*q$	$3 \cdot p$	$p/2$
$\left(5, \frac{26}{5}, -1, 0, 2\right)$	$\left(-1, -\frac{24}{5}, 1, 0, -2\right)$	$\left(6, \frac{53}{5}, -1, -\frac{1}{5}, 4, \frac{2}{5}\right)$	$\left(6, \frac{3}{5}\right)$	$\left(1, \frac{1}{10}\right)$.

Bei Verwendung von $\mathbb{Z}_7$ als Koeffizientenkörper liefern obige Definitionen für $p = (2,3)$ und $q = (3,5,6,0,2)$ in $\mathcal{P}_{\mathbb{Z}_7}$ dann

$p+q$	$p-q$	$p*q$	$3 \cdot p$	$p/2$
$(5,1,6,0,2)$	$(6,5,1,0,5)$	$(6,5,6,4,4,6)$	$(6,2)$	$(1,5)$.

Die Definitionen der arithmetischen Grundoperationen in $\mathcal{P}_K$ können wir direkt in Algorithmen übersetzen. Exemplarisch führen wir einen Algorithmus *MultPoly* zur Polynommultiplikation an, Algorithmen für Summe, Differenz etc. können in analoger Weise erstellt werden.

Algorithmus *MultPoly*: Multiplikation in $K[x]$

```
if p = 0 ∨ q = 0
   return 0
for i from 0 to deg(p) + deg(q)
   r_i ← 0
   for j from 0 to i
      r_i ← r_i + p_j · q_{i−j}
return r
```

Aufruf: *MultPoly*(p,q)
Eingabe: $p,q \in \mathcal{P}_K$
Ausgabe: $r \in \mathcal{P}_K$
mit: $r = p * q$.

Computerprogrammierung (Parametrisierte Datenstrukturen). *Wie bei Vektoren empfiehlt sich auch hier das Abspeichern des Koeffizientenkörpers K in der Datenstruktur $\mathcal{P}_K$, wodurch eine universelle Umsetzung der Algorithmen für die Polynomarithmetik für beliebige Körper K ermöglicht wird, siehe dazu die Ausführungen zu*

Vektoren auf Seite 107. Sofern die Programmiersprache es erlaubt, kann man auch hier natürlich die Operatoren für die Grundoperationen wieder überladen.

Der Aufwand zur Berechnung von $MultPoly(p,q)$ beträgt $O(\deg(p)\cdot\deg(q))$. Eine Verbesserung lässt sich durch den Karatsuba Algorithmus erzielen, der für Polynome ähnlich wie für ganze Zahlen funktioniert. Analog zur Diskussion auf Seite 61 besteht die Hauptidee darin, die Multiplikation zweier Polynome p und q von Grad $2n+1$ auf 3 Multiplikationen von Polynomen von Grad $\leq n$ zurückzuführen. Mittels der Zerlegung

$$\begin{aligned} \tilde{p} &:= (p_0,\ldots,p_n,0,\ldots) & \bar{p} &:= (p_{n+1},\ldots,p_{2n+1},0,\ldots) \\ \tilde{q} &:= (q_0,\ldots,q_n,0,\ldots) & \bar{q} &:= (q_{n+1},\ldots,q_{2n+1},0,\ldots) \end{aligned} \tag{14.9}$$

gilt dann

$$p\cdot q = (\tilde{p}+\mathrm{x}^n\bar{p})(\tilde{q}+\mathrm{x}^n\bar{q}) = \tilde{p}\tilde{q}+\mathrm{x}^n((\tilde{p}+\bar{p})(\tilde{q}+\bar{q})-\tilde{p}\tilde{q}-\bar{p}\bar{q})+\mathrm{x}^{2n}\bar{p}\bar{q},$$

und wir benötigen lediglich die 3 Produkte $\tilde{p}\tilde{q}$, $\bar{p}\bar{q}$ und $(\tilde{p}+\bar{p})(\tilde{q}+\bar{q})$, in denen die beteiligten Polynome aus $K[\mathrm{x}]^n$ sind. Der darauf basierende rekursive Karatsuba Alorithmus benötigt zur Multiplikation zweier Polynome von Grad n nur $O(n^{1.585})$ arithmetische Operationen in K, siehe [11]. Analog zur Situation in $\mathcal{Z}_B$ macht sich der Karatsuba Algorithmus erst für Polynome größeren Grades bezahlt.

Da sowohl die in (14.9) gezeigte Zerlegung als auch die nötigen Multiplikationen mit Potenzen von x auf Koeffiziententupeln sehr effizient in $\mathcal{P}_K$ durchführbar sind, ist eine Realisierung des Karatsuba Algorithmus in dieser Datenstruktur besonders vorteilhaft. Noch bessere Verfahren zur Polynommultiplikation basieren wieder auf der schnellen Fouriertransformation, siehe dazu [5] oder [10].

Bemerkung. Die Repräsentation von Funktionen im Computer ist keine einfache Angelegenheit und hängt sehr stark von der verwendeten Programmiersprache ab. Da wir vorerst keine Algorithmen betrachten, die Polynomfunktionen als Ein- oder Ausgabe verwenden, verschieben wir die Diskussion über Datenstrukturen zur Computerdarstellung von Funktionen, die Polynomfunktionen als Spezialfall enthalten, auf Band *2*.

15 Polynomdivision und größter gemeinsamer Teiler

Polynomdivision mit Rest

Wir wenden uns nun einer weiteren wichtigen Operation auf Polynomen zu, der *Division mit Rest* zweier Polynome. Für gegebene Polynome $a,b\in K[\mathrm{x}]$ ist eine Gleichung $b\cdot q = a$ für $q\in K[\mathrm{x}]$ meist nicht lösbar, da b im Allgemeinen in $K[\mathrm{x}]$ bzgl. der Multiplikation nicht invertierbar ist. In $K[\mathrm{x}]$ können wir aber stets eine Gleichung der Form

$$a = b\cdot q + r \tag{15.10}$$

für $q,r \in K[x]$ mit gegebenen Polynomen a und b lösen, etwa $q = 0$ und $r = a$. Spannender wird es, wenn wir zusätzlich zu (15.10) noch $\deg(r) < \deg(b)$ fordern.

Problemstellung (Polynomdivision mit Rest).

Gegeben:	$a,b \in K[x]$
mit:	$b \neq 0$.
Gesucht:	$q,r \in K[x]$
mit:	$a = b \cdot q + r$ und $\deg(r) < \deg(b)$.

Beispiel

Zu $a = (3,5,-1,0,2,0,\ldots)$ und $b = \left(2,\frac{1}{5},0,\ldots\right)$ in $\mathbb{R}[x]$ erfüllen

$$q = (-9925,995,-100,10,0,\ldots) \quad \text{und} \quad r = (19853,0,\ldots)$$

die in der Problemstellung geforderten Bedingungen, wovon man sich durch einfaches Nachrechnen überzeugt.

Der nun folgende Satz gibt Aufschluss über die eindeutige Lösbarkeit der Polynomdivision mit Rest.

Satz

Polynomdivision mit Rest. Seien $a,b \in K[x]$ und $b \neq 0$. Dann existieren eindeutig bestimmte $q,r \in K[x]$ mit $a = b \cdot q + r$ und $\deg(r) < \deg(b)$.

Beweis. Wir weisen die Existenz von $q,r \in K[x]$ mit Induktion über den Grad von a nach[4]. Wir zeigen, dass jedes $a \in K[x]$ als

$$a = b \cdot q + r \quad \text{mit } \deg(r) < \deg(b) \tag{15.11}$$

geschrieben werden kann unter der Induktionsannahme, dass für jedes $a' \in K[x]$ mit $\deg(a') < \deg(a)$ Polynome q' und r' mit

$$a' = b \cdot q' + r' \quad \text{mit } \deg(r') < \deg(b) \tag{15.12}$$

existieren. Im Fall $\deg(a) < \deg(b)$ ist (15.11) mit $q = 0$ und $r = a$ einfach zu erfüllen. Für $\deg(a) \geq \deg(b)$ sei

$$m = \frac{\mathrm{fk}(a)}{\mathrm{fk}(b)} \cdot x^{\deg(a)-\deg(b)}, \tag{15.13}$$

womit $\deg(b \cdot m) = \deg(a)$ und $\mathrm{fk}(b \cdot m) = \mathrm{fk}(a)$ gilt. Subtrahiert man $b \cdot m$ von a, so wird damit der führende Koeffizient von a eliminiert, so dass jedenfalls $\deg(a-b\cdot m) < \deg(a)$ ist. Laut Induktionsannahme (15.12) existieren zu $a' = a-b\cdot m$ dann $q',r' \in K[x]$ mit

$$a - b \cdot m = b \cdot q' + r' \quad \text{mit } \deg(r') < \deg(b).$$

[4] Wir verwenden wieder die verallgemeinerte Induktion wie beim Beweis der Basisdarstellung in $\mathbb{N}$ auf Seite 48.

Daraus ergibt sich aber sofort $a = b \cdot (q' + m) + r'$, und mit

$$q = q' + m \quad \text{und} \quad r = r'$$

ist (15.11) erfüllt. Zum Nachweis der Eindeutigkeit nehmen wir an, dass q, r und $\bar{q}, \bar{r}$ zwei Lösungen sind, d.h.

$$b \cdot q + r = a = b \cdot \bar{q} + \bar{r},$$

und daher auch

$$b \cdot (q - \bar{q}) = \bar{r} - r$$

erfüllen. Damit müssen die Grade der Polynome auf beiden Seiten übereinstimmen. Nun gilt für die rechte Seite $\deg(\bar{r} - r) < \deg(b)$. Aus der Gradeigenschaft (13.1) der Multiplikation folgt, dass für die linke Seite $\deg(b(q - \bar{q})) < \deg(b)$ nur dann gelten kann, wenn $q - \bar{q} = 0$ und damit $q = \bar{q}$ ist. Damit muss auch $\bar{r} - r = 0$, also $r = \bar{r}$ gelten. □

Wie bei ganzen Zahlen nennen wir auch hier die eindeutig bestimmten q und r den *Quotient* und den *Rest* bei der Divsion von a durch b und schreiben dafür a div b bzw. a mod b. Für eine Analyse der Kondition der Polynomdivision mit Rest im Fall $K = \mathbb{R}$ verweisen wir auf [8]. Wir erwähnen lediglich, dass sich Datenfehler in den Koeffizienten von a und b besonders dann verstärken, wenn der führende Koeffizient $\mathrm{fk}(b)$ *klein* ist im Vergleich zu den anderen Koeffizienten von b.

Der Existenzbeweis für Quotient und Rest in $K[x]$ ist wieder konstruktiver Natur, daraus lässt sich der rekursive Algorithmus *QuotRestPolyRek* einfach ableiten. Für $\deg(a) < \deg(b)$ ist die Lösung durch $q = 0$ und $r = a$ gegeben, andernfalls berechnet man $a' = a - b \cdot m$ mit m wie in (15.13) und fährt mit a' anstelle von a rekursiv fort. Der Quotient q ergibt sich dann als $q' + m$, wobei q' den rekursiv berechneten Quotienten von a' und b bezeichnet. Der Rest wird unverändert aus dem Resultat des rekursiven Aufrufs übernommen.

Algorithmus *QuotRestPolyRek*: Division mit Rest in $K[x]$ rekursiv

```
if deg(a) < deg(b)
   q ← 0, r ← a
else
   m ← fk(a)/fk(b) · x^(deg(a)−deg(b))
   a' ← a − b ∗ m
   (q',r') ← QuotRestPolyRek(a',b)
   q ← q' + m, r ← r'
return (q,r)
```

Aufruf: *QuotRestPolyRek*(a,b)
Eingabe: $a, b \in K[x]$
mit: $b \neq 0$.
Ausgabe: $q, r \in K[x]$
mit: $a = b \cdot q + r$ und $\deg(r) < \deg(b)$.

Der Algorithmus *QuotRestPolyRek* beruht ausschließlich auf arithmetischen Operationen in $K[x]$ und kann somit in jeder Datenstruktur für univariate Polynome direkt umgesetzt werden. Der Aufwand dafür beträgt $O((\deg(b) + 1)(\deg(a) - \deg(b) + 1))$ Operationen in K, siehe dazu [11].

Der Rest r wird in *QuotRestPolyRek* ermittelt, indem vom Ausgangswert a startend in jedem rekursiven Aufruf $b \cdot m$ vom bisher ermittelten „Rest“ subtrahiert wird. Dies geschieht solange, bis der Grad des Rests kleiner als $\deg(b)$ ist, und damit

der Basisfall der Rekursion erreicht ist. Ist man an einer Umsetzung der Polynomdivision in einem Schleifenalgorithmus interessiert, so beginnt man mit $r = a$. Ist $\deg(r) < \deg(b)$, so ist $q = 0$ und r die Lösung. Andernfalls berechnet man in einer Schleife solange $\deg(r) \geq \deg(b)$

$$\begin{aligned} &m \leftarrow \tfrac{\mathrm{fk}(r)}{\mathrm{fk}(b)} \cdot \mathrm{x}^{\deg(r)-\deg(b)}, \\ &r \leftarrow r - b * m \quad \text{und} \\ &q \leftarrow q + m. \end{aligned} \tag{15.14}$$

Im Schleifenalgorithmus *QuotRestPoly* in $\mathcal{P}_K$ können wir im Fall $\deg(r) \geq \deg(b)$ für q mit einem Tupel von Nullen der Länge $\deg(a) - \deg(b) + 1$ starten, da der letztlich gesuchte Quotient genau den Grad $\deg(a) - \deg(b)$ hat. Mit $d = \deg(r) - \deg(b)$ kann die in (15.14) benötigte Addition von $m = \mathrm{fk}(r)/\mathrm{fk}(b) \cdot \mathrm{x}^d$ zu q bei Verwendung von $\mathcal{P}_K$ einfach erreicht werden, indem man $q_d = \mathrm{fk}(r)/\mathrm{fk}(b)$ setzt. Dies ist möglich, weil in jedem Schritt der Exponent d abnimmt und somit keine Position d mehrmals bearbeitet wird. Zur Berechnung von $b * m = b * q_d \cdot \mathrm{x}^d$ multipliziert man $q_d \cdot b$ zuerst *skalar* gefolgt von einer Multiplikation mit x^d, die man aber durch Verschiebung der Koeffizienten um d Stellen nach rechts mit Hilfe des Unteralgorithmus $\mathit{Verschiebe}_{\mathcal{P}_K}$ realisieren kann[5].

$\mathit{Verschiebe}_{\mathcal{P}_K}(p,d)$ liefert für $p \in \mathcal{P}_K$ und $d \in \mathbb{N}_0$ ein $\bar{p} \in \mathcal{P}_K$, in dem die Koeffizienten von p um d Stellen nach rechts verschoben sind, und in dem am Beginn mit Nullen aufgefüllt ist, d.h.

$$\deg(\bar{p}) = \deg(p) + d \quad \text{und} \quad \bar{p}_i = \begin{cases} 0 & \text{für } i < d \\ p_{i-d} & \text{sonst.} \end{cases}$$

Es gilt $\mathit{Verschiebe}_{\mathcal{P}_K}(p,d) = p * \mathrm{x}^d$.

Algorithmus *QuotRestPoly*: Division mit Rest in $K[\mathrm{x}]$

```
r ← a
if deg(r) < deg(b)
  q ← 0
else
  koef(q) ← (0 | i = 0, ..., deg(a) − deg(b))
  while deg(r) ≥ deg(b)
    d ← deg(r) − deg(b)
    q_d ← fk(r)/fk(b)
    r ← r − Verschiebe_{P_K}(q_d · b, d)
return (q, r)
```

Aufruf: $\mathit{QuotRestPoly}(a,b)$
Eingabe: $a,b \in \mathcal{P}_K$
mit: $b \neq 0$.
Ausgabe: $q,r \in \mathcal{P}_K$
mit: $a = b \cdot q + r$ und $\deg(r) < \deg(b)$.

Dieses Beispiel zeigt, dass ein Algorithmus durch Bezugnahme auf eine bestimmte Datenstruktur effizienter realisiert werden kann als in einer allgemeinen Formulierung, die von Datenstrukturen unabhängig ist. Auch im Karatsuba Algorithmus

[5] Es sei nochmals daran erinnert, dass wir in Zifferntupeln die Ziffern zu absteigenden Potenzen der Basis abgespeichert haben. Der Algorithmus $\mathit{Verschiebe}_{\mathcal{Z}_B}$ für Zifferntupel von Seite 61 verschiebt daher nach links, die hier für $\mathcal{P}_K$ gezeigte Version $\mathit{Verschiebe}_{\mathcal{P}_K}$ verschiebt nach rechts.

wird man bei einer Umsetzung in $\mathcal{P}_K$ anstelle der allgemeinen Multiplikation mit Potenzen von x natürlich auf den Algorithmus *Verschiebe*$_{\mathcal{P}_K}$ zurückgreifen.

Beispiel

Für unser Beispiel von Seite 127 mit $a = (3,5,-1,0,2)$ und $b = \left(2,\frac{1}{5}\right)$ zeigen wir die Wertebelegung der Variablen in den jeweiligen Schleifendurchläufen während der Abarbeitung von *QuotRestPoly*(a,b).

		$q^{(0)} = (0,0,0,0)$	$r^{(0)} = (3,5,-1,0,2)$
$d^{(1)} = 3$	$\frac{\mathrm{fk}(r^{(1)})}{\mathrm{fk}(b)} = 10$	$q^{(1)} = (0,0,0,10)$	$r^{(1)} = (3,5,-1,-20)$
$d^{(2)} = 2$	$\frac{\mathrm{fk}(r^{(2)})}{\mathrm{fk}(b)} = -100$	$q^{(2)} = (0,0,-100,10)$	$r^{(2)} = (3,5,199)$
$d^{(3)} = 1$	$\frac{\mathrm{fk}(r^{(3)})}{\mathrm{fk}(b)} = 995$	$q^{(3)} = (0,995,-100,10)$	$r^{(3)} = (3,-1985)$
$d^{(4)} = 0$	$\frac{\mathrm{fk}(r^{(4)})}{\mathrm{fk}(b)} = -9925$	$q^{(4)} = (-9925,995,-100,10)$	$r^{(4)} = (19853)$

Die Division mit Rest hat uns bei ganzen Zahlen zu den Kongruenzklassen $\mathbb{Z}_m$ geführt, siehe Abschnitt 6. Ähnlich dazu sprechen wir auch im Polynomring wieder von Restklassen modulo $m \in K[\mathrm{x}]$. Den kanonischen Repräsentanten der Restklasse von a modulo m erhalten wir als Rest bei der Polynomdivision von a durch m. Polynome a und b mit gleichem Rest bei Division durch m nennen wir *kongruent modulo* m und schreiben dafür analog zu den ganzen Zahlen $a \equiv_m b$. Damit können wir einen Zusammenhang zwischen Polynomauswertung und Polynomdivision herstellen.

Satz

Polynomauswertung und Polynomdivision. Seien $a \in K[\mathrm{x}]$ und $w,\xi \in K$. Dann gilt

$$\mathrm{eval}(a,w) = \xi \iff a \equiv_{\mathrm{x}-w} (\xi,0,\ldots).$$

Beweis. Die Polynomdivision mit Rest von a durch $\mathrm{x}-w$ garantiert eine Darstellung

$$a = (\mathrm{x}-w)\cdot q + (r_0,0,\ldots) \quad \text{mit eindeutig bestimmten } q \in K[\mathrm{x}] \text{ und } r_0 \in K,$$

und damit $a \equiv_{\mathrm{x}-w} (r_0,0,\ldots)$. Die Aussage des Satzes folgt dann aus

$$\mathrm{eval}(a,w) = \underbrace{\mathrm{eval}((\mathrm{x}-w)\cdot q,w)}_{=0} + \underbrace{\mathrm{eval}((r_0,0,\ldots),w)}_{=r_0} = r_0.$$

□

Durch diesen Satz kann die Polynomauswertung durch eine Kongruenz ausgedrückt werden und umgekehrt. Beispielsweise kann damit der Rest bei der Polynomdivision von a durch $\mathrm{x}-w$ einfach durch Auswertung von a an der Stelle w bestimmt werden.

Größter gemeinsamer Teiler von Polynomen

Den größten gemeinsamen Teiler zweier ganzer Zahlen haben wir in Abschnitt 5 schon kennengelernt. Auch im Polynomring $K[x]$ kann man für zwei Polynome von größten gemeinsamen Teilern sprechen.

Definition

Größte gemeinsame Teiler in $K[x]$. Seien $a,b,t \in K[x]$. Man nennt t einen *Teiler* von a genau dann, wenn ein $q \in K[x]$ existiert mit $a = t \cdot q$, man schreibt dafür $t|a$. Man spricht von einem *gemeinsamen Teiler* t von a und b genau dann, wenn t sowohl ein Teiler von a als auch von b ist. t heißt *ein größter gemeinsamer Teiler* von a und b genau dann, wenn t ein gemeinsamer Teiler von a und b ist, und $s|t$ für jeden (anderen) gemeinsamen Teiler s von a und b gilt.

Beispiel

In $\mathbb{R}[x]$ ist für $a = 1 + 2x + x^2$ und $b = -1 + x^2$ durch $t = 1 + x$ ein größter gemeinsamer Teiler gegeben. Auch $t = 2 + 2x$ ist ein größter gemeinsamer Teiler.

In $\mathbb{Z}$ war der ggT definiert als das größte Element (bzgl. der natürlichen Ordnung $\leq$) der Menge der gemeinsamen Teiler und damit eindeutig bestimmt. Für $a,b \in K[x]$ existiert stets ein größter gemeinsamer Teiler, der aber nur bis auf Multiplikation mit Skalaren eindeutig bestimmt ist, siehe etwa [11]. Unter allen größten gemeinsamen Teilern zeichnen wir jenen mit führendem Koeffizienten gleich 1 als *den* größten gemeinsamen Teiler aus und schreiben dafür $\mathrm{ggT}(a,b)$. Weiters sei $\mathrm{ggT}(0,0) = 0$, in diesem Fall ist der ggT nicht normiert.

Problemstellung (Größter gemeinsamer Teiler in $K[x]$).

Gegeben:	$a,b \in K[x]$
mit:	$a \neq 0$ oder $b \neq 0$.
Gesucht:	$t \in K[x]$
mit:	$t = \mathrm{ggT}(a,b)$.

Der Euklidsche Algorithmus zur Berechnung des größten gemeinsamen Teilers zweier ganzer Zahlen aus Abschnitt 5 beruht ausschließlich auf Division mit Rest in $\mathbb{Z}$. Da wir auch für Polynome über einem Körper eine Division mit Rest zur Verfügung haben, kann der rekursive Algorithmus von Seite 66 beinahe unverändert auf $K[x]$ übertragen werden. Als Alternative dazu präsentieren wir hier eine Schleifenvariante, deren Notation sich an den erweiterten Euklidschen Algorithmus von Seite 67 anlehnt. Im Vergleich zu $\mathbb{Z}$ ist lediglich am Ende des Algorithmus dafür zu

Algorithmus *GGTPolyEuklid*: Euklidscher Algorithmus in $K[x]$

```
r'' ← a, r' ← b
while r' ≠ 0
   r ← r'' mod r', r'' ← r', r' ← r
t ← r''/fk(r'')
return t
```

Aufruf:	$GGTPolyEuklid(a,b)$
Eingabe:	$a,b \in \mathcal{P}_K$
mit:	$a \neq 0$ oder $b \neq 0$.
Ausgabe:	$t \in \mathcal{P}_K$
mit:	$t = \mathrm{ggT}(a,b)$.

sorgen, dass das Resultat tatsächlich normiert ist. Wir verwenden die Datenstruktur $\mathcal{P}_K$, weil damit die Polynomdivision effizient realisiert werden kann, siehe dazu die Ausführungen im vorangegangenen Abschnitt.

Die Korrektheit des Algorithmus folgt mit dem gleichen Argument wie bei den ganzen Zahlen, da die dem Verfahren zugrunde liegende Eigenschaft

$$\mathrm{ggT}(a,b) = \mathrm{ggT}(b, a \bmod b),$$

siehe (5.5), auch in $K[x]$ gilt. Der Abbruch nach endlich vielen Schritten folgt daraus, dass in jedem Schleifendurchlauf der Grad von r' abnimmt. Für $\deg(a) \geq \deg(b) \geq 0$ beträgt der Rechenaufwand für den Euklidschen Algorithmus in $K[x]$ im schlechtesten Fall $2 \cdot \deg(a) \cdot \deg(b) + O(\deg(a))$ Additionen und Multiplikationen in K und maximal $\deg(b) + 1$ Berechnungen von Inversen in K, siehe [10].

Beispiel

Seien nun $a, b \in \mathcal{P}_{\mathbb{Q}}$ mit

$$a = \left(-12, -10, -\frac{40}{3}, 8, \frac{40}{3}, -\frac{26}{3}, \frac{4}{3}\right) \quad \text{und} \quad b = \left(14, 49, \frac{77}{2}, -\frac{77}{4}, -\frac{49}{2}, \frac{7}{4}, \frac{7}{2}\right).$$

Der Aufruf von `GGTPolyEuklid[a,b]` in *Mathematica* berechnet mit rationaler Arithmetik die Euklidsche Restfolge $r^{(0)} = a$, $r^{(1)} = b$,

$$\begin{aligned}
r^{(2)} &= \left(-\frac{52}{3}, -\frac{86}{3}, -28, \frac{46}{3}, \frac{68}{3}, -\frac{28}{3}\right) \\
r^{(3)} &= \left(-\frac{141}{28}, \frac{617}{56}, -3, -\frac{723}{56}, \frac{43}{7}\right) \\
r^{(4)} &= \left(-\frac{658217}{44376}, -\frac{3708859}{88752}, -\frac{54194}{5547}, \frac{1525321}{88752}\right) \\
r^{(5)} &= \left(-\frac{260339200}{19775809}, -\frac{130169600}{19775809}, \frac{130169600}{19775809}\right) \\
r^{(6)} &= 0.
\end{aligned}$$

Damit ist $r^{(5)}$ ein größter gemeinsamer Teiler von a und b, durch Normierung erhalten wir $\mathrm{ggT}(a,b) = (-2, -1, 1)$.

Dieses Beispiel zeigt sehr schön das für das Rechnen in $\mathbb{Q}$ typische Anwachsen der Zähler und Nenner in den Zwischenergebnissen. Zum Berechnen des ggT in $\mathbb{Q}[x]$ gibt es eine Variante des Euklidschen Algorithmus, nämlich den *modularen* ggT-*Algorithmus*. In diesem werden Berechnungen in $\mathbb{Q}[x]$ durch Berechnungen in $\mathbb{Z}_m[x]$ (für geeignete Primzahlen m) ersetzt, und durch Anwendung des Chinesischen Restalgorithmus aus Abschnitt 6 der ggT in $\mathbb{Q}[x]$ rekonstruiert. Da hier die Koeffizienten jeweils durch m beschränkt bleiben, kann ein Anwachsen wie in obigem Beispiel verhindert werden. Auch die Darstellung von $\mathrm{ggT}(a,b)$ als Linearkombination der Polynome a und b ist durch Übertragung des *erweiterten Euklidschen Algorithmus* aus Abschnitt 5 auf Polynome möglich. Für Details zu diesen Varianten des Euklidschen Algorithmus sei auf [10] oder [5] verwiesen.

16 Polynomauswertung in $\mathbb{R}$

Der Auswertung eines Polynoms $p \in K[x]$ an einer Stelle $w \in K$ sind wir bereits auf Seite 121 begegnet. In diesem Abschnitt führen wir die Diskussion aus numerischer Sicht, weshalb wir uns auf den Fall $K = \mathbb{R}$ einschränken.

Problemstellung (Polynomauswertung).

Gegeben: $p \in \mathbb{R}[x], w \in \mathbb{R}$.
Gesucht: $\xi \in \mathbb{R}$
mit: $\xi = \text{eval}(p, w)$.

Für die Konditionsanalyse des Problems fassen wir mit $n = \deg(p)$ die Polynomkoeffizienten $p_0, \ldots, p_n$ als einen Vektor in $\mathbb{R}^{n+1}$ auf. Die Abbildung von Seite 3, die den Eingangsdaten $p \in \mathbb{R}^{n+1}$ und $w \in \mathbb{R}$ die eindeutige Lösung zuordnet, ist dann durch

$$\varphi : \mathbb{R}^{n+2} \to \mathbb{R}, \quad \begin{pmatrix} p \\ w \end{pmatrix} \mapsto \sum_{i=0}^{n} p_i w^i \tag{16.15}$$

gegeben. Der folgende Satz gibt Auskunft über die relative komponentenweise Kondition des Problems (φ, p, w), vergleiche auch mit Seite 32.

Satz **Kondition der Polynomauswertung.** Seien φ wie in (16.15), $p \in \mathbb{R}^{n+1}$ mit $p_i \neq 0$, $i = 0, \ldots, n$, $w \in \mathbb{R}$ mit $w \neq 0$ und $\varphi(p, w) \neq 0$ gegeben. Dann ist

$$\kappa_{\text{rel,komp}} = \frac{\sum_{i=0}^{n} |p_i||w|^i + |\sum_{i=1}^{n} i p_i w^i|}{|\sum_{i=0}^{n} p_i w^i|} \tag{16.16}$$

die relative komponentenweise Kondition des Problems (φ, p, w).

Beweis. Die Abbildung φ ist partiell differenzierbar, die Aussage ist somit unmittelbare Konsequenz von Formel (3.54). □

Die Verstärkung relativer Datenfehler ist also besonders dann groß, wenn in (16.16) der Nenner sehr klein im Vergleich zum Zähler ist. Dies ist etwa dann der Fall, wenn die Terme $p_i w^i$ von ungefähr gleicher Größenordnung sind, aber alternierendes Vorzeichen haben.

Für eine algorithmische Lösung der Polynomauswertung könnte man einfach die Summe mit Hilfe einer Schleife, etwa unter Verwendung von Algorithmus *SummeS* von Seite 94, über die reellen Ausdrücke $p_i w^i$ bilden. Der Aufwand dafür beträgt $3n$ Elementaroperationen $+, \cdot$, siehe Übungsaufgabe IV.1. Effizienter hingegen ist die Ausnutzung der Beziehung

$$\text{eval}(p, w) = \sum_{i=0}^{\deg(p)} p_i w^i = p_0 + w \sum_{i=1}^{\deg(p)} p_i w^{i-1} = p_0 + w \cdot \text{eval}(p_{1:\deg(p)}, w) \tag{16.17}$$

für $p \neq 0$, wonach die Auswertung des Polynoms p die Auswertung jenes Polynoms mit Grad $\deg(p) - 1$ involviert, das durch Streichen des ersten Koeffizienten p_0 aus dem Tupel p entsteht. Nach $\deg(p)$-maliger Anwendung des Arguments (16.17) kommt letztlich die Auswertung des konstanten Polynoms $(p_{\deg(p)})$, also $\mathrm{eval}((p_{\deg(p)}), w) = p_{\deg(p)}$, ins Spiel.

Beispiel

Im Fall $\deg(p) = 4$ gilt

$$\mathrm{eval}(p,w) = p_0 + w \cdot \big(p_1 + w \cdot \big(p_2 + w \cdot \big(p_3 + w \cdot p_4\big)\big)\big) .$$

Die Abarbeitung dieser Rekursion in umgekehrter Reihenfolge mit Hilfe einer Schleife entspricht dem sogenannten Horner[6] Algorithmus *EvalPolyHorner* zur Polynomauswertung.

Algorithmus *EvalPolyHorner*: Polynomauswertung in $\mathbb{R}$, Horner Algorithmus

```
if p = 0
   return 0
else
   n ← deg(p), ξ ← p_n
   for i from n − 1 to 0 by -1
      ξ ← p_i + w · ξ
   return ξ
```

Aufruf: *EvalPolyHorner*(p,w)
Eingabe: $p \in \mathcal{P}_{\mathbb{R}}, w \in \mathbb{R}$.
Ausgabe: $\xi \in \mathbb{R}$
mit: $\xi = \mathrm{eval}(p,w)$.

Der Horner Algorithmus[7] benötigt in jedem Durchlauf je eine Multiplikation und Addition, insgesamt also $2n$ und damit $O(n)$ Elementaroperationen.

Die komponentenweise Rückwärtsstabilität der Gleitkommarealisierung, siehe Seite 43, des Algorithmus *EvalPolyHorner* lässt sich aus folgendem Satz ableiten. Darin stehen $\tilde{p}$ und $\tilde{w}$ wieder für gerundete Eingangsgrößen.

Satz

Seien $p \in \mathbb{R}^{n+1}, \tilde{p} \in \mathbb{R}^{n+1}, w, \tilde{w} \in \mathbb{R}$ mit $p_i, \tilde{p}_i \neq 0, i = 0, \ldots, n, w, \tilde{w} \neq 0$ und $|\tilde{p}_i - p_i|/|p_i| \leq \mathbf{u}$, $i = 0, \ldots, n$, $|\tilde{w} - w|/|w| \leq \mathbf{u}$ sowie φ wie in (16.15). Die Gleitkommarealisierung des Algorithmus *EvalPolyHorner* zur Berechnung von $\varphi(p,w)$ liefert eine Näherung $\tilde{\xi} \in \mathbb{R}$ mit

$$\tilde{\xi} = \varphi(\hat{p}, \hat{w}), \tag{16.18}$$

wobei $\hat{w} \in \mathbb{R}, \hat{p} \in \mathbb{R}^{n+1}$ mit $\hat{w} = \tilde{w}$ und

$$\frac{|\hat{p}_i - \tilde{p}_i|}{|\tilde{p}_i|} \lesssim (2i+1)\mathbf{u} \quad \text{für } \mathbf{u} \to 0. \tag{16.19}$$

[6] Horner, William George: 1786–1837, britischer Mathematiker. Neben dem hier diskutierten Schema zur Polynomauswertung gibt es auch noch eine nach Horner benannte Methode zur Approximation von Lösungen einer polynomialen Gleichung.

[7] Da der Algorithmus nur auf Addition und Multiplikation im Koeffizientenbereich beruht, kann er auch auf andere Koeffizientenbereiche als $\mathbb{R}$ übertragen werden.

Beweis. Sei $\xi^{(i)}$ der Wert der Variable ξ in *EvalPolyHorner*, den man bei exakter Arithmetik, aber mit gerundeten Eingangsdaten $\tilde{p}$, $\tilde{w}$, am Ende des Schleifendurchlaufs mit Index i erhalten würde. Infolge der bei jeder Gleitkommaoperation entstehenden Rundungsfehler, siehe (2.21), ist jedoch für $i = n-1, \ldots, 0$ der *tatsächlich berechnete Wert* durch

$$\tilde{\xi}^{(i)} = (\tilde{p}_i + \tilde{w} \cdot \tilde{\xi}^{(i+1)}(1+\varepsilon_i))(1+\varepsilon_i'),$$

mit $\tilde{\xi}^{(n)} = \tilde{p}_n$ und $|\varepsilon_i|, |\varepsilon_i'| \leq \mathbf{u}$ gegeben.

Für das Gleitkommaresultat $\tilde{\xi}^{(0)} = \tilde{\xi}$ des Algorithmus erhält man mit der Setzung $\varepsilon_n' = 0$ nach Auflösung der Rekursion die Darstellung

$$\tilde{\xi} = \sum_{i=0}^{n} \left((1+\varepsilon_i') \prod_{j=0}^{i-1} (1+\varepsilon_j)(1+\varepsilon_j') \right) \cdot \tilde{p}_i \tilde{w}^i,$$

siehe [2]. Mit $\hat{w} := \tilde{w}$ und $\hat{p}_i := (1+\varepsilon_i') \prod_{j=0}^{i-1}(1+\varepsilon_j)(1+\varepsilon_j') \cdot \tilde{p}_i$ ist dies aber äquivalent zu (16.18). Die Abschätzung (16.19) folgt aus den Beziehungen

$$(1+\varepsilon_i') \prod_{j=0}^{i-1} (1+\varepsilon_j)(1+\varepsilon_j') \approx 1 + \sum_{j=0}^{i-1} (\varepsilon_j + \varepsilon_j') + \varepsilon_i' \quad \text{für } \mathbf{u} \to 0$$

und

$$\left| \sum_{j=0}^{i-1} (\varepsilon_j + \varepsilon_j') + \varepsilon_i' \right| \leq (2i+1)\mathbf{u}.$$

□

Das Gleitkommaresultat $\tilde{\xi}$ lässt sich also als exakte Auswertung des Polynoms mit den Koeffizienten $\hat{p}_i$ an der Stelle $\hat{w}$ darstellen. Aus (16.19) folgt mit $i = n$ die Stabilitätszahl $C_{R,komp} = 2n+1$, siehe auch (4.74). Da diese die Anzahl $2n$ der Elementaroperationen von *EvalPolyHorner* nicht wesentlich übersteigt, ist dessen Gleitkommarealisierung stabil im Sinne der Rückwärtsanalyse. Nach Seite 42 ist diese damit auch vorwärtsstabil.

Wie auf Seite 42 erläutert, kann der maximale Rundungsfehler $\mathbf{u}$ bei der Gleitkommarealisierung eines Algorithmus (in erster Näherung) durch das Produkt aus Stabilitäts- und Konditionszahl multiplikativ verstärkt werden. Bei hoher Kondition des Problems ist somit auch das Resultat einer stabilen Gleitkommarealisierung, d.h. bei Vorliegen einer kleinen Stabilitätszahl, mit Vorsicht zu genießen. Dies soll in folgendem Beispiel veranschaulicht werden.

Beispiel

Wir betrachten die Auswertung des Polynoms

$$p = (1, -8, 28, -56, 70, -56, 28, -8, 1) \tag{16.20}$$

an der Stelle $w = 0.996$. Der Aufruf *EvalPolyHorner*(p, w) liefert bei Verwendung von IEEE double precision Gleitkommaarithmetik mit $-3.5527 \cdot 10^{-15}$ eine negative Zahl. Es gilt jedoch $\text{eval}(p, x) \geq 0$ für alle $x \in \mathbb{R}$, da (16.20) auch als $(x-1)^8$ dargestellt werden kann. Da eigentlich Daten $w \in \mathbb{Q}$ und $p \in \mathcal{P}_{\mathbb{Q}}$ vorliegen, kann

EvalPolyHorner auch mit rationaler Arithmetik, dann aber vergleichsweise hohem Rechenaufwand, durchgeführt werden. Dies führt auf das exakte Ergebnis $\text{eval}(p,0.996) = 6.5536 \cdot 10^{-20}$. Für den durch die Gleitkommarechnung verursachten relativen Fehler folgt somit

$$\frac{|-3.5527 \cdot 10^{-15} - 6.5536 \cdot 10^{-20}|}{|6.5536 \cdot 10^{-20}|} = 5.4211 \cdot 10^{4}.$$

Grund für das stark fehlerhafte Ergebnis trotz vorliegender Rückwärtsstabilität ist die hohe Konditionszahl des betrachteten Problems, für die nach (16.16) $\kappa_{\text{rel,komp}} = 3.8442 \cdot 10^{21}$ gilt. Der Fehler liegt aber dennoch unter der theoretischen Fehlerschranke $C_{R,komp}\kappa_{\text{rel,komp}}\mathbf{u} = 7.2554 \cdot 10^{6}$ mit $C_{R,komp} = 17$. Eine Fortsetzung des Beispiels findet sich in Übungsaufgabe IV.2.

17 Polynominterpolation in $\mathbb{R}$

Bei der Polynominterpolation in $\mathbb{R}$ sucht man zu gegebenen paarweise verschiedenen Stützstellen oder Knoten $x_0, \ldots, x_n \in \mathbb{R}$ und Stützwerten $y_0, \ldots, y_n \in \mathbb{R}$ ein Polynom $p \in \mathbb{R}[x]^n$, das die Stützwerte y_i an den Stützstellen x_i *interpoliert*, d.h. die Bedingung

$$\text{eval}(p,x_i) = y_i \quad \text{für } i = 0, \ldots, n$$

erfüllt.

Problemstellung (Polynominterpolation).

Gegeben:	$x,y \in \mathbb{R}^{n+1}$
mit:	x_i paarweise verschieden, d.h. $x_i \neq x_j$ falls $i \neq j$.
Gesucht:	$p \in \mathbb{R}[x]^n$
mit:	$\text{eval}(p,x_i) = y_i \quad \text{für } i = 0, \ldots, n.$

Satz

Existenz und Eindeutigkeit des Interpolationspolynoms. Zu $x,y \in \mathbb{R}^{n+1}$ mit $n \in \mathbb{N}$ und paarweise verschiedenen Stützstellen x_i existiert genau ein Polynom $p \in \mathbb{R}[x]^n$ mit $\text{eval}(p,x_i) = y_i$ für $i = 0, \ldots, n$.

Beweis. Existiert ein Polynom mit der gewünschten Eigenschaft in $\mathbb{R}[x]^n$, so ist es als

$$p = \sum_{i=0}^{n} p_i \cdot x^i \tag{17.21}$$

bzgl. der monomialen Basis (13.4) darstellbar. Die Interpolationsbedingung für p wird damit zu

$$\text{eval}(p,x_j) = \text{eval}(\sum_{i=0}^{n} p_i \cdot x^i, x_j) = \sum_{i=0}^{n} p_i \cdot \text{eval}(x^i,x_j) = \sum_{i=0}^{n} p_i \cdot x_j^i = y_j,$$

was auch als lineares Gleichungssystem

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{pmatrix} \cdot \begin{pmatrix} p_0 \\ p_1 \\ \vdots \\ p_n \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{pmatrix} \tag{17.22}$$

für die zu bestimmmenden Koeffizienten p_i formuliert werden kann. Wie aus der linearen Algebra bekannt ist, ist die in (17.22) auftretende *Vandermonde*[8]*-Matrix* von x invertierbar genau dann, wenn $x_i \neq x_j$ für $i \neq j$. Dies ist nach Voraussetzung der Fall, also folgen Existenz und Eindeutigkeit des Interpolationspolynoms. □

Beispiel

Zu $x = (-1\ 6\ 2)^T$ und $y = (3\ -5\ 4)^T$ suchen wir das eindeutige Interpolationspolynom in $\mathbb{R}[x]^2$. Durch Lösen des Gleichungssystems

$$\begin{pmatrix} 1 & -1 & 1 \\ 1 & 6 & 36 \\ 1 & 2 & 4 \end{pmatrix} \cdot \begin{pmatrix} p_0 \\ p_1 \\ p_2 \end{pmatrix} = \begin{pmatrix} 3 \\ -5 \\ 4 \end{pmatrix}$$

erhalten wir

$$p = \left(\frac{57}{14}, \frac{59}{84}, -\frac{31}{84} \right). \tag{17.23}$$

Der obige Existenzbeweis ist also konstruktiver Natur, aus Stabilitäts- und Aufwandsgründen ist aber ein Algorithmus basierend auf (17.22) eher nicht geeignet. Vorteilhaftere Methoden beruhen auf der Darstellung des Interpolationspolynoms bzgl. anderer Basen.

Polynominterpolation nach Lagrange

Eine Alternative zu (17.21) ist die Darstellung mit Hilfe der sogenannten Langrange[9]-Polynome.

Definition

Lagrange-Polynome. Sei $x \in \mathbb{R}^{n+1}$ mit $x_i \neq x_j$ falls $i \neq j$ gegeben. Dann sind die Lagrange-Polynome $L_0^{(x)}, \ldots, L_n^{(x)} \in \mathbb{R}[x]^n$ definiert als

$$L_i^{(x)} = \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j}. \tag{17.24}$$

[8] Vandermonde, Alexandre-Théophile: 1735–1796, französischer Mathematiker. Er wollte eigentlich Musiker werden und wandte sich erst im Alter von 35 Jahren der Mathematik zu. Er verfasste nur vier mathematische Publikationen, und in einer von diesen begründete er die Theorie der Determinanten. Die Vandermonde-Determinante kommt in seinen Arbeiten allerdings nicht vor. Sie ist wahrscheinlich deswegen nach ihm benannt, da die von Vandermonde verwendete Notation missinterpretiert wurde, und so der Eindruck entstand, er hätte diese Determinante eingeführt.

[9] Lagrange, Joseph Louis: 1736–1813, italienischer Mathematiker. Der sich auch für die Astronomie interessierende Mathematiker machte mit seinem Werk „Mécanique analytique“ die Mechanik zu einer

Für die Lagrange-Polynome gilt

$$\text{eval}(L_i^{(x)}, x_j) = \delta_{ij},$$

zudem bilden sie eine Basis des $\mathbb{R}[x]^n$. Für das Interpolationspolynom zu x,y gilt daher auch die Darstellung $p = \sum_{i=0}^{n} \lambda_i \cdot L_i^{(x)}$ mit geeigneten Koeffizienten $\lambda \in \mathbb{R}^{n+1}$. Diese stimmen aber mit den Stützwerten $y \in \mathbb{R}^{n+1}$ überein, was aus der Interpolationsbedingung

$$y_j = \text{eval}(p, x_j) = \text{eval}(\sum_{i=0}^{n} \lambda_i \cdot L_i^{(x)}, x_j) = \sum_{i=0}^{n} \lambda_i \cdot \text{eval}(L_i^{(x)}, x_j) = \lambda_j$$

ersichtlich ist. Das Interpolationspolynom ist in der Lagrange Darstellung also durch

$$p = \sum_{i=0}^{n} y_i \cdot L_i^{(x)} \tag{17.25}$$

gegeben.

Beispiel

Wir betrachten erneut die Daten $x = (-1\ 6\ 2)^T$ und $y = (3\ -5\ 4)^T$. Die Stützstellen x legen die Lagrange Basispolynome

$$\begin{aligned} L_0^{(x)} &= \frac{(x-6)}{(-1-6)}\frac{(x-2)}{(-1-2)} = \left(\frac{4}{7}, -\frac{8}{21}, \frac{1}{21}\right) \\ L_1^{(x)} &= \frac{(x-(-1))}{(6-(-1))}\frac{(x-2)}{(6-2)} = \left(-\frac{1}{14}, -\frac{1}{28}, \frac{1}{28}\right) \\ L_2^{(x)} &= \frac{(x-(-1))}{(2-(-1))}\frac{(x-6)}{(2-6)} = \left(\frac{1}{2}, \frac{5}{12}, -\frac{1}{12}\right) \end{aligned}$$

eindeutig fest. Mit Hilfe der Koeffizienten y landen wir laut (17.25) über

$$p = 3 \cdot L_0^{(x)} - 5 \cdot L_1^{(x)} + 4 \cdot L_2^{(x)}$$

wieder beim Interpolationspolynom $\left(\frac{57}{14}, \frac{59}{84}, -\frac{31}{84}\right)$ aus (17.23).

Auch die Darstellung (17.24) bezüglich der Lagrange Basis ist für algorithmische Zwecke aufgrund des damit verbundenen Aufwands eher unpraktisch, sie ist aber vor allem für theoretische Überlegungen von Vorteil, so auch für die Konditionsanalyse. Diese führen wir unter der Annahme durch, dass die Stützstellen x fixiert sind und Datenstörungen nur bzgl. der Stützwerte y auftreten können. Infolge beschreiben

Teildisziplin der mathematischen Analysis. Am Anfang seiner Karriere leidete er darunter, dass er keinen mathematisch wissenschaftlichen Betreuer hatte und sich vieles selbst beibringen musste. Er gewann mehrfach den Preis der „Académie des Sciences“ in Paris, unter anderem 1772 für eine Arbeit zum Dreikörperproblem der Himmelsmechanik. Den Preis musste er in diesem Jahr jedoch mit Euler teilen. Die nach ihm benannte Interpolationsmethode wurde erstmals von Waring 1779 publiziert, Lagrange selbst publizierte sie erst 1795. Er lieferte auch wichtige Beiträge zur Zahlentheorie.

wir das Interpolationsproblem, ähnlich wie auf Seite 3, durch die Abbildung

$$\varphi^{(x)} : \mathbb{R}^{n+1} \to \mathbb{R}[\mathrm{x}]^n,\ y \mapsto p, \tag{17.26}$$

wobei p das eindeutige Interpolationspolynom zu x und y ist. Mit $[a,b]$ bezeichnen wir ein Intervall, in dem alle Stützstellen liegen, und definieren mit dessen Hilfe auf $\mathbb{R}[\mathrm{x}]^n$ die Norm

$$\|p\| := \max_{w\in[a,b]} |\mathrm{eval}(p,w)|. \tag{17.27}$$

Der folgende Satz gibt dann Auskunft über die normweise absolute Kondition des Problems, wobei wir auf $\mathbb{R}^{n+1}$ die Maximumsnorm $\|\cdot\|_\infty$ verwenden.

Satz

Kondition der Polynominterpolation. Zu paarweise verschiedenen Stützstellen $x \in \mathbb{R}^{n+1}$ mit $x_i \in [a,b]$ sei $\varphi^{(x)}$ wie in (17.26) gegeben. Bezüglich der Maximumsnormen ist die absolute Konditionszahl des Problems $(\varphi^{(x)}, y)$ mit Stützwerten $y \in \mathbb{R}^{n+1}$ dann durch

$$\kappa_{\mathrm{abs}} = \Lambda^{(x)} := \max_{w\in[a,b]} \sum_{i=0}^{n} |\mathrm{eval}(L_i^{(x)}, w)| \tag{17.28}$$

gegeben. $\Lambda^{(x)}$ wird dabei als Lebesgue[10]-Konstante zu den Stützstellen $x \in \mathbb{R}^{n+1}$ bezeichnet.

Beweis. Mit $\tilde{y} \in \mathbb{R}^{n+1}$ bezeichnen wir zunächst eine beliebige Störung der Daten $y \in \mathbb{R}^{n+1}$. Aus (17.25) folgt dann

$$\|p - \tilde{p}\| = \max_{w\in[a,b]} |\sum_{i=0}^{n} (y_i - \tilde{y}_i)\mathrm{eval}(L_i^{(x)}, w)| \le \|y - \tilde{y}\|_\infty \max_{w\in[a,b]} \sum_{i=0}^{n} |\mathrm{eval}(L_i^{(x)}, w)|.$$

Somit gilt jedenfalls $\kappa_{\mathrm{abs}} \le \Lambda^{(x)}$. Zu beliebigem $\eta > 0$ betrachten wir nun die spezielle Datenstörung $\tilde{y}_i = y_i + \eta \cdot \mathrm{sign}(\mathrm{eval}(L_i^{(x)}, \bar{w}))$, wobei $\bar{w} \in [a,b]$ gemäß

$$\sum_{i=0}^{n} |\mathrm{eval}(L_i^{(x)}, \bar{w})| = \max_{w\in[a,b]} \sum_{i=0}^{n} |\mathrm{eval}(L_i^{(x)}, w)|$$

festgelegt wird. Damit folgt $\eta = \|y - \tilde{y}\|_\infty$ und

$$\begin{aligned}
\|p - \tilde{p}\| &\ge |\mathrm{eval}(p, \bar{w}) - \mathrm{eval}(\tilde{p}, \bar{w})| \\
&= \eta |\sum_{i=0}^{n} \mathrm{sign}(\mathrm{eval}(L_i^{(x)}, \bar{w})) \cdot \mathrm{eval}(L_i^{(x)}, \bar{w})| \\
&= \eta \sum_{i=0}^{n} |\mathrm{eval}(L_i^{(x)}, \bar{w})| = \|y_i - \tilde{y}\| \max_{w\in[a,b]} \sum_{i=0}^{n} |\mathrm{eval}(L_i^{(x)}, w)|.
\end{aligned}$$

Also gilt auch $\kappa_{\mathrm{abs}} \ge \Lambda^{(x)}$, woraus die Behauptung des Satzes folgt. □

[10] Lebesgue, Henri Leon: 1875–1941, französischer Mathematiker. Er formulierte die Maßtheorie und begründete den Begriff des Lebesgue Integrals, einer Verallgemeinerung des Riemann Integrals.

n	$\Lambda^{(x)}$ für äquidistante Knoten	$\Lambda^{(x)}$ für Tschebyscheff-Knoten
4	2.2	1.8
9	17.8	2.4
14	283.2	2.6
19	5889.6	2.8

Tabelle IV.1. Lebesgue Konstante für äquidistante Knoten bzw. Tschebyscheff Knoten $x \in \mathbb{R}^{n+1}$ im Intervall $[-1,1]$

Auch wenn κ_{abs} in (17.28) mit Hilfe der Lagrange Basispolynome beschrieben wird, für die Kondition ist es natürlich gleichgültig, bzgl. welcher Basis des $\mathbb{R}[x]^n$ das Interpolationspolynom betrachtet wird. Die Kondition ist jedoch abhängig von der Lage und auch der Anzahl der Stützstellen x. So wächst bei äquidistant verteilten Stützstellen

$$x_i = a + \frac{b-a}{n} i \quad \text{für } i = 0, \ldots, n \tag{17.29}$$

die Konditionszahl $\Lambda^{(x)}$ rasch mit n an, während sie bei den sogenannten *Tschebyscheff*[11] *Knoten*

$$x_i = \frac{a+b}{2} + \frac{b-a}{2} \cos\left(\frac{i}{n}\pi\right) \quad \text{für } i = 0, \ldots, n \tag{17.30}$$

nur sehr langsam steigt, siehe Tabelle IV.1 für den Fall $[a,b] = [-1,1]$.

Polynominterpolation nach Newton

Neben (17.21) und (17.25) kann das Interpolationspolynom zu $x, y \in \mathbb{R}^{n+1}$ auch mit Hilfe der der sogenannten Newton[12]-Polynome dargestellt werden.

Definition

Newton-Polynome. Sei $x \in \mathbb{R}^{n+1}$ mit $x_i \neq x_j$ falls $i \neq j$ gegeben. Dann sind die Newton-Polynome $N_0^{(x)}, \ldots, N_n^{(x)} \in \mathbb{R}[x]^n$ durch

$$N_i^{(x)} := \prod_{j=0}^{i-1} (x - x_j) \tag{17.31}$$

definiert.

[11] Tschebyscheff, Pafnuti Lwowitsch: 1821–1894, russischer Mathematiker. Neben Interpolations- und Approximationstheorie arbeitete er in der Zahlentheorie sowie in Wahrscheinlichkeitstheorie und Statistik, wurde aber auch durch die von ihm eingeführte und nach ihm benannte Familie von orthogonalen Polynomen bekannt. Er trug auch bei zum Beweis des Primzahlsatzes, den dann aber Hadamard, siehe Seite 26, zwei Jahre nach Tschebyscheffs Tod bewies.

[12] Newton, Sir Isaac: 1643–1727, englischer Mathematiker und Physiker. Begründete gemeinsam mit aber unabhängig von Leibniz die Infinitesimalrechnung. Es entbrannte ein Streit zwischen den beiden, wer nun der Erfinder sei. Newton wurde 1705 als erster Wissenschafter zum Ritter geschlagen.

Man beachte, dass die Stützstelle x_n in (17.31) nicht vorkommt, weiters gilt

$$\text{eval}(N_i^{(x)}, x_j) = 0 \quad \text{für alle } j < i. \tag{17.32}$$

Beispiel

Zu $x = (-1\ 6\ 2)^T$ lauten die Newton Basispolynome

$$\begin{aligned} N_0^{(x)} &= 1 = (1)\,, \\ N_1^{(x)} &= \mathrm{x} - (-1) = (1,1)\,, \\ N_2^{(x)} &= (\mathrm{x} - (-1)) \cdot (\mathrm{x} - 6) = (-6,-5,1)\,. \end{aligned} \tag{17.33}$$

Da auch die Newton-Polynome eine Basis des $\in \mathbb{R}[\mathrm{x}]^n$ bilden, existiert eine *Newton Darstellung*

$$p = \sum_{i=0}^{n} \lambda_i \cdot N_i^{(x)} \tag{17.34}$$

des Interpolationspolynoms mit eindeutig bestimmen Koeffizienten λ_i. Diese lassen sich unter Verwendung von (17.32) aus der Interpolationsbedingung

$$y_j = \text{eval}(p, x_j) = \text{eval}(\sum_{i=0}^{n} \lambda_i \cdot N_i^{(x)}, x_j) = \sum_{i=0}^{j} \lambda_i \cdot \text{eval}(N_i^{(x)}, x_j) \tag{17.35}$$

für p ableiten. Mit $a_{ji} := \text{eval}(N_i^{(x)}, x_j)$ ergibt sich $\lambda \in \mathbb{R}^{n+1}$ als Lösung des linearen Gleichungssystems

$$\begin{pmatrix} 1 & 0 & \cdots & 0 \\ 1 & a_{11} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & a_{n1} & \cdots & a_{nn} \end{pmatrix} \cdot \begin{pmatrix} \lambda_0 \\ \lambda_1 \\ \vdots \\ \lambda_n \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{pmatrix}. \tag{17.36}$$

Aus algorithmischer Sicht bedeutender ist jedoch die Beschreibung der Koeffizienten λ mit Hilfe sogenannter *dividierter Differenzen*. Dazu bezeichne im Folgenden $p_x^y(i,j)$ das eindeutige Interpolationspolynom aus $\mathbb{R}[\mathrm{x}]^{j-i}$ zu den Stützstellen $x_{i:j}$ und Stützwerten $y_{i:j}$ für $0 \le i \le j \le n$. Speziell ergibt sich in dieser Notation

$$p_x^y(i,i) = (y_i) \quad \text{für } i = 0, \ldots, n \tag{17.37}$$

und $p = p_x^y(0,n)$. Weiters gilt nach dem Lemma von Aitken[13] folgender rekursiver Zusammenhang.

[13] Aitken, Alexander Craig: 1895–1967, neuseeländischer Mathematiker. In der numerischen Analysis begründete er die Idee, die Konvergenz von numerischen Verfahren zu beschleunigen. Er war auch bekannt für sein außergewöhnliches Gedächtnis, beispielsweise wusste er π auswendig auf an die 2000 Stellen.

Satz Für $i = 0, \ldots, n$ und $k = 1, \ldots, n - i$ gilt

$$p_x^y(i, i+k) = \frac{(\mathrm{x} - x_i)p_x^y(i+1, i+k) - (\mathrm{x} - x_{i+k})p_x^y(i, i+k-1)}{x_{i+k} - x_i}. \tag{17.38}$$

Beweis. Wegen $p_x^y(i+1, i+k), p_x^y(i, i+k-1) \in \mathbb{R}[\mathrm{x}]^{k-1}$ stellt die rechte Seite in (17.38) ein Polynom in $\mathbb{R}[\mathrm{x}]^k$ dar. Aufgrund der Eindeutigkeit des Interpolationspolynoms $p_x^y(i, i+k) \in \mathbb{R}[\mathrm{x}]^k$ bleibt zu zeigen, dass auch die rechte Seite die Interpolationsbedingung mit $x_{i:k}$ und $y_{i:k}$ erfüllt. Dies ist jedoch unmittelbare Folge der Interpolationseigenschaften von $p_x^y(i+1, i+k)$ und $p_x^y(i, i+k-1)$ und des Evaluations-Homomorphismus von Seite 122. □

Bezeichnen wir nun mit $\Delta_x^y(i, i+k)$ den Koeffizienten von $p_x^y(i, i+k)$ bei x^k, so folgt aus (17.37) sofort

$$\Delta_x^y(i, i) = y_i \quad \text{für } i = 0, \ldots, n. \tag{17.39}$$

Weiters führt ein Koeffizientenvergleich in (17.38) bei x^k auf

$$\Delta_x^y(i, i+k) = \frac{\Delta_x^y(i+1, i+k) - \Delta_x^y(i, i+k-1)}{x_{i+k} - x_i} \tag{17.40}$$

für $i = 0, \ldots, n$ und $k = 1, \ldots, n - i$. Aufgrund dieser Berechnungsvorschrift werden die $\Delta_x^y(i, i+k)$ auch *dividierte Differenzen* von x und y genannt. Ausgehend von (17.34) folgt wegen (17.32) für $k = 0, \ldots, n$ die Darstellung

$$p_x^y(0, k) = \sum_{i=0}^{k} \lambda_i \cdot N_i^{(x)}$$

und damit schließlich der Zusammenhang

$$\lambda_k = \Delta_x^y(0, k) \quad \text{für } k = 0, \ldots, n \tag{17.41}$$

zwischen den dividierten Differenzen und den gesuchten Koeffizienten λ. Eine tabellarische Veranschaulichung der Rekursion (17.40) zu den Startwerten (17.39) stellt das *Differenzenschema* dar.

i	$k=0$		$k=1$	$\cdots$	$k=n-1$	$k=n$
0	$\Delta_x^y(0,0)$	—	$\Delta_x^y(0,1)$	$\cdots$	$\Delta_x^y(0,n-1)$ —	$\Delta_x^y(0,n)$
1	$\Delta_x^y(1,1)$	—	$\Delta_x^y(1,2)$	$\cdots$	$\Delta_x^y(1,n)$	
$\vdots$	$\vdots$		$\vdots$	$\iddots$		
$n-1$	$\Delta_x^y(n-1,n-1)$	—	$\Delta_x^y(n-1,n)$			
n	$\Delta_x^y(n,n)$					

Aus diesem können die Koeffizienten λ_k also einfach aus der obersten Zeile abgelesen werden.

Beispiel

Für unsere Beispielvektoren $x = (-1\ \ 6\ \ 2)^T$ und $y = (3\ \ -5\ \ 4)^T$ können die dividierten Differenzen

$$\begin{aligned}
\Delta_x^y(0,0) &= y_0 = 3\\
\Delta_x^y(1,1) &= -5\\
\Delta_x^y(2,2) &= 4\\
\Delta_x^y(0,1) &= \frac{\Delta_x^y(1,1)-\Delta_x^y(0,0)}{x_1-x_0} = \frac{-5-3}{6-(-1)} = -\frac{8}{7}\\
\Delta_x^y(1,2) &= \frac{\Delta_x^y(2,2)-\Delta_x^y(1,1)}{x_2-x_1} = \frac{4-(-5)}{2-6} = -\frac{9}{4}\\
\Delta_x^y(0,2) &= \frac{\Delta_x^y(1,2)-\Delta_x^y(0,1)}{x_2-x_0} = \frac{-\frac{9}{4}-\left(-\frac{8}{7}\right)}{2-(-1)} = -\frac{31}{84}
\end{aligned}$$

anhand des Differenzenschemas

i	$k=0$		$k=1$		$k=2$
0	**3**	—	$-\frac{8}{7}$	—	$-\frac{31}{84}$
		⟋		⟋	
1	-5	—	$-\frac{9}{4}$		
		⟋			
2	4				

zusammengefasst werden. Die Darstellung

$$p = 3 \cdot N_0^{(x)} - \frac{8}{7} \cdot N_1^{(x)} - \frac{31}{84} \cdot N_2^{(x)} \tag{17.42}$$

mit (17.33) entspricht wieder dem Interpolationspolynom $\left(\frac{57}{14}, \frac{59}{84}, -\frac{31}{84}\right)$ aus (17.23). Die Einträge im Differenzenschema haben natürlich nichts mit jenen der Matrix des Gleichungssystems (17.36) gemein, das in unserem Beispiel

$$\begin{pmatrix} 1 & 0 & 0 \\ 1 & 7 & 0 \\ 1 & 3 & -12 \end{pmatrix} \cdot \begin{pmatrix} \lambda_0 \\ \lambda_1 \\ \lambda_2 \end{pmatrix} = \begin{pmatrix} 3 \\ -5 \\ 4 \end{pmatrix}$$

lautet.

Für einen Algorithmus zur Berechnung der Koeffizienten λ_k nach (17.41) könnten wir den rekursiven Zusammenhang (17.40) der dividierten Differenzen verwenden. Eine direkte Umsetzung würde aber viele Werte des Differenzenschemas mehrfach berechnen. Zur Reduktion des Aufwands können wir alternativ die λ berechnen, indem wir das Differenzenschema „von links nach rechts“ und jeweils „von unten nach oben“ durchlaufen. Zu λ_i gelangen wir am oberen Ende der Spalte $k = i$, nicht mehr benötigte Werte des Schemas überschreiben wir einfach.

Computerrepräsentation (Polynome bzgl. der Newton Basis)**.** Die auf Seite 123 eingeführte Datenstruktur basiert auf der Darstellung von Polynomen bzgl. der mono-

mialen Basis. Je nach Anwendung können aber auch andere Basen des Vektorraums $\mathbb{R}[x]^n$ der Polynome als Grundlage für die Computerdarstellung dienen. Im Fall der Polynominterpolation nach Newton führen wir eine Datenstruktur $\mathcal{N}_{\mathbb{R}}^n$ ein, in der wir für $p \in \mathcal{N}_{\mathbb{R}}^n$ die Newton Koeffizienten λ mit $nk(p)$ und die Stützstellen x mit $st(p)$ ansprechen. Auch in dieser Datenstruktur sind die Vektorraumgrundoperationen auf Polynomen realisierbar, wir gehen aber nicht näher darauf ein. Stattdessen stellen wir in weiterer Folge einen Algorithmus zur Auswertung von Polynomen in $\mathcal{N}_{\mathbb{R}}^n$ vor.

Algorithmus *InterPolyNewton*: Polynominterpolation in $\mathbb{R}$ nach Newton

$\lambda \leftarrow y$
for k **from** 1 **to** n
  for i **from** n **to** k **by** -1
    $\lambda_i \leftarrow \frac{\lambda_i - \lambda_{i-1}}{x_i - x_{i-k}}$
$nk(p) \leftarrow \lambda, st(p) \leftarrow x$
return p

Aufruf: $InterPolyNewton(x,y)$
Eingabe: $x,y \in \mathbb{R}^{n+1}$
mit: $x_i \neq x_j$ falls $i \neq j$.
Ausgabe: $p \in \mathcal{N}_{\mathbb{R}}^n$
mit: $\text{eval}(p,x_i) = y_i \quad$ für $i = 0,\ldots,n$.

Der Aufwand zur Berechnung der dividierten Differenzen beträgt somit $O(n^2)$ elementarer Rechenoperationen. In [6] wird gezeigt, dass bei einer Anordnung der Stützstellen in auf- oder absteigender Reihenfolge der durch die Gleitkommaarithmetik verursachte Rundungsfehler den unvermeidbaren Fehler nicht übersteigt, die Gleitkommarealisierung von *InterPolyNewton* dann also numerisch stabil im Sinne der Vorwärtsanalyse ist.

Der Vorteil der Berechnung des Interpolationspolynoms bzgl. der Newtonschen Basis ist, dass es in dieser Form effizient ausgewertet werden kann. Dazu ist eine leichte Modifikation des Horner Algorithmus *EvalPolyHorner* erforderlich. Ähnlich zu (16.17) gilt für das Auswerten von p aus (17.34) an der Stelle w

$$\text{eval}(p,w) = \text{eval}(\sum_{i=0}^{n} \lambda_i \cdot N_i^{(x)}, w) = \sum_{i=0}^{n} \lambda_i \cdot \prod_{j=0}^{i-1}(w - x_j) =$$

$$= \lambda_0 + (w - x_0) \sum_{i=1}^{n} \lambda_i \cdot \prod_{j=1}^{i-1}(w - x_j) = \lambda_0 + (w - x_0) \cdot \text{eval}(\bar{p}, w).$$

Mit $\bar{p}$ bezeichnen wir dabei das Polynom in Newton Darstellung mit Koeffizienten $\lambda_1,\ldots,\lambda_n$ und Stützstellen $x_1,\ldots,x_n$. Das Evaluieren eines Interpolationspolynoms mit $n+1$ Koeffizienten und Stützstellen führen wir also auf das Evaluieren eines Interpolationspolynoms mit n Koeffizienten und Stützstellen zurück. Nach n solchen Reduktionsschritten gelangen wir zum Newton Polynom mit dem Koeffizienten λ_n und der Stützstelle x_n, das zu λ_n evaluiert. Die Abarbeitung dieser Rekursion in umgekehrter Reihenfolge mittels Schleife, vgl. mit Seite 133, entspricht dem Algorithmus *EvalPolyNewtonHorner*.

Beispiel

Das Newton Interpolationspolynom (17.42) zu den Daten $x = (-1\ 6\ 2)^T$ und $y = (3\ -5\ 4)^T$ ist durch $p \in \mathcal{N}_{\mathbb{R}}^2$ mit $nk(p) = \left(3, -\frac{8}{7}, -\frac{31}{84}\right)$ und $st(p) = (-1,6,2)$

Algorithmus *EvalPolyNewtonHorner*: Polynomauswertung in $\mathbb{R}$, Newton Horner Algorithmus

```
λ ← nk(p), x ← st(p)
n ← |λ| − 1, ξ ← λ_n
for i from n − 1 to 0 by -1
   ξ ← λ_i + (w − x_i)ξ
return ξ
```

Aufruf: *EvalPolyNewtonHorner*(p,w)
Eingabe: $p \in \mathcal{N}_{\mathbb{R}}^n, w \in \mathbb{R}$.
Ausgabe: ξ
mit: $\xi = \text{eval}(p,w)$.

beschrieben. Die Auswertung an der Stelle 5 nach dem Newton Horner Algorithmus ergibt

$$3 + (5 - (-1)) \cdot \left(-\frac{8}{7} + (5-6) \cdot \left(-\frac{31}{84}\right)\right) = -\frac{23}{14}.$$

Zum Vergleich liefert die Auswertung des Interpolationspolynoms (17.23) in Standarddarstellung nach dem Horner Algorithmus

$$\frac{57}{14} + 5 \cdot \left(\frac{59}{84} + 5 \cdot \left(-\frac{31}{84}\right)\right) = -\frac{23}{14}.$$

Approximationsfehler

Die Daten x und y für die Polynominterpolationsaufgabe könnten von experimentellen Beobachtungen chemischer oder physikalischer Prozesse stammen, etwa wenn x Zeitpunkte beschreibt, an denen die Höhe y eines fallenden Körpers gemessen wird. Ohne Zusatzinformationen kann die Frage, ob das Interpolationspolynom p den Prozess richtig beschreibt, also etwa ob $\tilde{y} = \text{eval}(p,\tilde{x})$ eine physikalisch sinnvolle Näherung für die tatsächliche Höhe zum Zeitpunkt $\tilde{x} \in \mathbb{R}$ mit $\tilde{x} \notin \{x_0, \ldots, x_n\}$ ist, nicht beantwortet werden.

Sind die Stützwerte y_i jedoch Werte einer gegebenen, hinreichend glatten Funktion f, die an den Stützstellen x_i interpoliert werden soll, also

$$y_i = f(x_i) \quad \text{für } i = 0, \ldots, n, \tag{17.43}$$

so können wir den Approximationsfehler, der bei der Polynominterpolation der Daten auftritt, abschätzen.

Satz

Approximationsfehler. Sei $f \in C^{n+1}[a,b]$ und $p \in \mathbb{R}[x]^n$ das Interpolationspolynom zu (17.43) mit paarweise verschiedenen $x_i \in [a,b]$. Dann gilt für jedes $w \in [a,b]$

$$|f(w) - \text{eval}(p,w)| \leq \frac{|E^{(x)}(w)|}{(n+1)!} \max_{t \in [a,b]} |f^{(n+1)}(t)|, \tag{17.44}$$

wobei die Fehlerfunktion $E^{(x)}$ durch

$$E^{(x)}(w) := \prod_{i=0}^{n}(w - x_i) \tag{17.45}$$

definiert ist.

Beweis. Für $w = x_i$ gilt der Satz trivialerweise. Für den Fall $w \neq x_i$ betrachten wir die $(n+1)$-mal stetig differenzierbare Funktion

$$\varphi(t) = f(t) - \mathrm{eval}(p,t) - \frac{f(w) - \mathrm{eval}(p,w)}{E^{(x)}(w)} E^{(x)}(t) \tag{17.46}$$

mit den $n+2$ verschiedenen Nullstellen $x_0, \ldots, x_n$ und w. Die wiederholte Anwendung des Satzes von Rolle[14] garantiert die Existenz einer Nullstelle $\xi \in [a,b]$ von $\varphi^{(n+1)}$ und liefert mit (17.46)

$$0 = \varphi^{(n+1)}(\xi) = f^{(n+1)}(\xi) - 0 - \frac{f(w) - \mathrm{eval}(p,w)}{E^{(x)}(w)}(n+1)!.$$

Auflösen nach $f(w) - \mathrm{eval}(p,w)$ ergibt dann (17.44). □

Der Approximationsfehler hängt somit wesentlich vom Verhalten der Funktion $E^{(x)}$ ab, diese wiederum von der Lage der Stützstellen im Intervall $[a,b]$. Bei äquidistanter Verteilung (17.29) weicht $E^{(x)}$ in der Nähe der Intervallränder deutlich stärker von Null ab als in der Intervallmitte. Entsprechen die Stützstellen den Tschebyscheff Knoten (17.30), so ist der Verlauf von $E^{(x)}$ gleichmäßiger. Abbildung IV.5 zeigt einen Vergleich der entsprechenden Fehlerfunktionen.

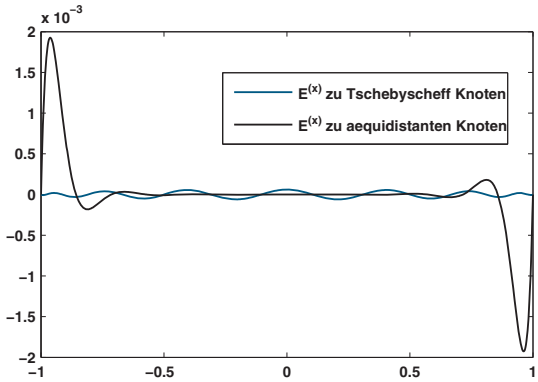


Abb. IV.5. $E^{(x)}$ für äquidistante und Tschebyscheff Knoten auf $[-1,1]$ mit $n = 14$

[14] Rolle, Michel: 1652–1719, französischer Mathematiker. Neben dem berühmten Satz stammt von ihm auch die Notation $\sqrt[n]{x}$ für die n-te Wurzel.

Beispiel

Gegeben sei die Funktion $f : x \mapsto \frac{1}{1+25x^2}$, die an 15 im Intervall $[-1,1]$ äquidistant verteilten Stützstellen, also (17.29) mit $n = 14$ und $[a,b] = [-1,1]$, interpoliert werden soll. Mit den entsprechenden Stützwerten aus (17.43) liefert der Aufruf *InterPolyNewton*(x,y) das Interpolationspolynom $p \in \mathcal{N}_{\mathbb{R}}^{14}$. Die linke Seite der Abbildung IV.6 zeigt die Daten x und y, die Funktion f sowie die dem Interpolationspolynom entsprechende Polynomfunktion. Aufgrund der Konstruktion erfüllt das Interpolationspolynom p die Interpolationsbedingung, zwischen den Stützstellen weist das Polynom im Gegensatz zur Funktion f jedoch Oszillationen auf, die von der Intervallmitte zu den Rändern hin immer stärker werden.

Im Vergleich dazu zeigt die rechte Seite der Abbildung IV.6 das Ergebnis der Polynominterpolation von f bei Verwendung der Tschebyscheff Knoten (17.30) mit $[a,b] = [-1,1]$ und $n = 14$. Die Oszillationen zwischen den Stützstellen sind gleichmäßiger und zu den Rändern hin deutlich geringer, das Interpolationspolynom von Grad 14 zu Tschebyscheff Knoten stellt also eine weit bessere Approximation der Funktion f dar. Die Erklärung dafür liefert der obige Satz zum Approximationsfehler der Polynominterpolation. Demnach wird der Fehler vom Verlauf der Fehlerfunktion (17.45) beeinflusst, $E^{(x)}$ selbst ist ja von der Lage der Stützstellen x_i abhängig. Die Betrachtung der Abbildung IV.5 macht alles klar. Im Beispiel wurde mit IEEE double precision Gleitkommaarithmetik gerechnet, mit Rundungsfehlereffekten haben die beobachteten Oszillationen aber nichts zu tun. Davon kann man sich leicht durch eine Wiederholung des Beispiels mit rationaler Arithmetik überzeugen.

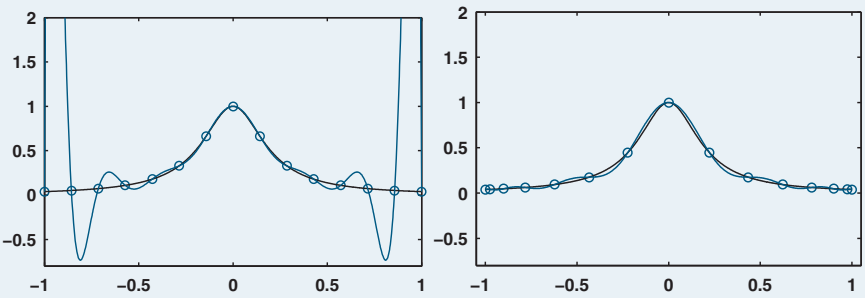


Abb. IV.6. Polynominterpolation der Funktion $f : x \mapsto \frac{1}{1+25x^2}$

Der Vergleich der beiden soeben berechneten Interpolationspolynome ist übrigens nur deshalb sinnvoll, weil neben den jeweiligen Daten x und y auch die Zusatzinformation vorhanden ist, dass die y-Werte von der Auswertung der Funktion f stammen. Liegen nur diskrete Daten x und y vor, kann man vom Interpolationspolynom lediglich verlangen, dass es die Interpolationsbedinung erfüllt.

Die in der linken Seite der Abbildung IV.6 beobachteten Oszillationen des Interpolationpolynoms werden durch eine Erhöhung der Stützstellenanzahl n in (17.29) nicht verringert sondern sogar noch erheblich verstärkt. Dies zeigt also, dass zu gegebener Funktion f und äquidistanter Stützstellenverteilung die Interpolationspolynome zu $(x_0 \ldots x_n)^T$ und $(f(x_0) \ldots f(x_n))^T$ für $n \to \infty$ im Allgemeinen nicht punktweise gegen f konvergieren. Zwar gibt es nach dem Approximationssatz von

Weierstraß[15] zu jedem $f \in C[a,b]$ mit $a,b \in \mathbb{R}, a < b$, und jedem $\varepsilon > 0$ ein Polynom $p \in K[x]$ mit

$$\max_{w\in[a,b]} |f(w) - \mathrm{eval}(p,w)| < \varepsilon,$$

– jede stetige Funktion kann also gleichmäßig durch ein Polynom approximiert werden – Interpolationspolynome zu äquidistanten Knoten kommen dafür in der Regel jedoch nicht in Frage.

Abschließend gehen wir anhand des obigen Beispiels auf die Rolle der Lage der Stützstellen im Hinblick auf die Kondition (17.28) der Polynominterpolation ein.

Beispiel

Fortsetzung. Wir kehren zur Polynominterpolation von Seite 147 der Funktion f mit äquidistant verteilten Knoten x zurück. Wir betrachten aber nun auch einen Datenvektor $\tilde{y} \in \mathbb{R}^{15}$, der aus $y \in \mathbb{R}^{15}$ durch ledigliche Ersetzung von $y_8 = f(x_8) = 1$ durch $\tilde{y}_8 = 1.1$ hervorgeht. Der absolute Datenfehler in der Maximumsnorm ist somit

$$\|\tilde{y} - y\|_\infty = 0.1. \tag{17.47}$$

Mit $\tilde{p} \in \mathcal{N}_{\mathbb{R}}^{14}$ bezeichnen wir jenes Interpolationspolynom vom Grad 14, das wir durch den Aufruf *InterPolyNewton*$(x,\tilde{y})$ in Gleitkommaarithmetik erhalten. Die linke Seite in Abbildung IV.7 zeigt einen Vergleich der Polynomfunktionen, die zu den Interpolationspolynomen p bzw. $\tilde{p}$ gehören. Ohne die Abweichungen in Zahlen beschreiben zu wollen, stellen wir fest, dass sich die Interpolationspolynome zu den Daten y bzw. $\tilde{y}$ bei äquidistant verteilten Stützstellen x im Sinne der Norm (17.27) deutlich unterscheiden.

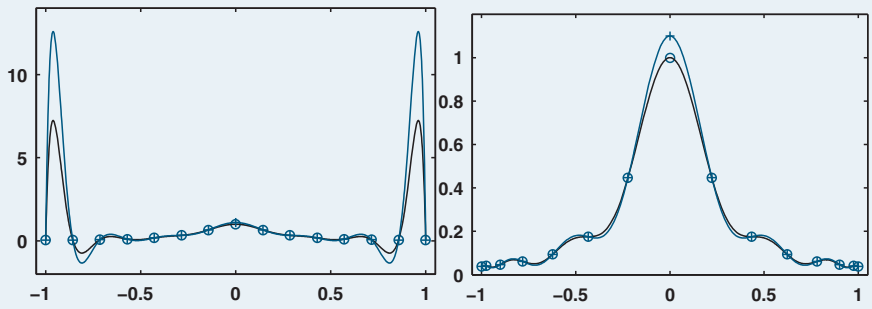


Abb. IV.7. Auswirkung einer Datenstörung auf die Polynominterpolation.

Bezeichne nun y die Daten (17.43) aus dem Beispiel von Seite 147 zu den Tschebyscheff Knoten x mit $n = 14$. Wieder führen wir eine Datenstörung durch, indem wir y_8 durch $\tilde{y}_8 = 1.1$ ersetzen. Auch in diesem Fall ist der Datenfehler durch (17.47) gegeben, der Aufruf *InterPolyNewton*$(x,\tilde{y})$ in Gleitkommaarithmetik liefert erneut ein Interpolationspolynom $\tilde{p} \in \mathcal{N}_{\mathbb{R}}^{14}$. Die rechte Seite in

[15] Weierstrass, Karl Theodor Wilhelm: 1815–1897, deutscher Mathematiker. Begründete die Theorie von komplexen Funktionen auf Basis von Potenzreihen und leistete wichtige Beiträge zur Theorie der elliptischen Funktionen. Von ihm stammt auch die ϵ-δ-Definition von Stetigkeit.

Abbildung IV.7 gibt einen graphischen Vergleich der Interpolationspolynome p zu den Daten y und $\tilde{p}$ zu den Daten $\tilde{y}$. Dieser zeigt, dass die Abweichungen zwischen p und $\tilde{p}$ bei Tschebyscheff Knoten mit $n = 14$ deutlich geringer sind als im Fall der äquidistanten Stützstellen mit $n = 14$.

Die Ursache für die unterschiedlich starken Auswirkungen der im Sinne von (17.47) identischen Datenstörung liegt in der unterschiedlich großen absoluten Konditionszahl $\Lambda^{(x)}$ der betrachteten Interpolationsprobleme. Laut Tabelle IV.1 gilt bei äquidistanten Knoten x für die Lebesque Konstante $\Lambda^{(x)} = 283.2$, bei Tschebyscheff Knoten x ist $\Lambda^{(x)} = 2.6$. Im ersten Fall ist der (maximale) Verstärkungsfaktor des absoluten Datenfehlers 0.1 also um den Faktor 107.3 größer als im zweiten.

Auch die Beobachtungen dieses Beispiels haben nichts mit Rundungsfehlereffekten zu tun, wovon man sich wieder durch Wiederholung mit rationaler Arithmetik überzeugen kann.

Varianten und Alternativen

Der Aufgabe, eine Datentabelle (x_i, y_i) oder eine Funktion $f : \mathbb{R} \to \mathbb{R}$, von der nur einige diskrete Funktionswerte $y_i = f(x_i)$ (und evtl. Ableitungswerte) an endlich vielen Stellen x_i vorliegen, durch eine einfache Funktion in geschlossener Form anzunähern, begegnet man in der Mathematik an vielen Stellen. Fordert man, dass die Näherungsfunktion an den gegebenen Stellen x_i mit den gegebenen Daten y_i bzw. $f(x_i)$ exakt übereinstimmt, so spricht man allgemein von einer *Interpolationsaufgabe*. Dazu kann man neben Polynomfunktionen etwa auch rationale, trigonometrische oder Exponentialfunktionen heranziehen, siehe [9]. Sind viele Daten gegeben, die zusätzlich Mess- und Beobachtungsfehlern unterliegen, wird man von der Interpolationsbedingung abweichen und durch Lösen einer *Approximationsaufgabe* versuchen, eine glatte Kurve, etwa mit Hilfe eines Polynom niedrigen Grades, durch die „Datenwolke“ zu legen. Wir kommen darauf in Band *2* zurück.

Auch rund um die zuvor diskutierte Polynominterpolation gibt es Varianten. Ist man zum Beispiel nur an einem (oder wenigen) Wert(en) des Interpolationspolynoms interessiert, so kann man diese(n) nach dem *Verfahren von Neville*[16] *und Aitken* auch berechnen, indem man aus dem rekursiven Zusammenhang (17.38) einen dazu ähnlichen Zusammenhang zum Auswerten des Interpolationspolynoms ableitet. Dabei muss das Interpolationspolynom selbst nicht bestimmt werden, siehe [9]. Auch kann (17.38) genutzt werden, um das Interpolationspolynom in Standarddarstellung $\mathcal{P}_{\mathbb{R}}$ basierend auf Polynomarithmetik zu berechnen.

[16] Neville, Eric Harold: 1889–1961, englischer Mathematiker. Beeinflusst von den Arbeiten Russels an den logischen Fundamenten der Mathematik verfasste er 1922 die „Prolegomena to analytical geometry“, in der er die logischen Grundlagen und einen axiomatischen Aufbau der analytischen Geometrie schafft. Sein Hauptwerk „Jacobian elliptic functions“ verfasste er 1944. er lehrte an der Universität Cambridge und spielte im Zusammenhang mit dem Aufenthalt des berühmten indischen Mathemtkers S. Ramanujan am Trinity College in Cambridge eine wichtige Rolle.

Varianten ergeben sich aber auch hinsichtlich der betrachteten Eingangsdaten. So wird bei der *Hermitschen*[17] *Polynominterpolationsaufgabe*, siehe [3], gefordert, dass an den Stützstellen neben den Stützwerten auch Ableitungswerte interpoliert werden. Dabei kann es sich um Ableitungen unterschiedlicher Ordnung handeln, die an verschiedenen Stützstellen gegeben sind. Sollen an den Stützstellen $x \in \mathbb{R}^{n+1}$ die Stützwerte $y \in \mathbb{R}^{n+1}$ sowie die Ableitungswerte (erster Ordnung) $y' \in \mathbb{R}^{n+1}$ interpoliert werden, so lautet die Aufgabe, ein Polynom $p \in \mathbb{R}[\mathrm{x}]^{2n+1}$ zu finden, das

$$\mathrm{eval}(p,x_i) = y_i \text{ und } \mathrm{eval}(p',x_i) = y'_i, \quad \text{für } i = 0,\ldots,n,$$

erfüllt, wobei $p' \in \mathbb{R}[\mathrm{x}]^{2n}$ das Ableitungspolynom zu p bezeichnet. Stimmen hingegen alle Knoten überein, also $x_0 = \cdots = x_n$, und sind neben $f(x_0)$ die Ableitungswerte $f'(x_0), f''(x_0), \ldots, f^{(n)}(x_0)$ gegeben, so führt dies auf das *Taylorpolynom*

$$T = \sum_{i=0}^{n} \frac{f^{(i)}(x_0)}{i!}(\mathrm{x} - x_0)^i, \tag{17.48}$$

vergleiche etwa mit (2.25).

Wie im Beispiel von Seite 147 angedeutet, weisen Interpolationspolynome in der Regel mit steigender Stützstellenanzahl $n+1$ immer stärkere Oszillationen auf. Eine qualitative Übereinstimmung mit der zu approximierenden Funktion ist bei der klassischen Polynominterpolation nur für kleine n oder bei spezieller Wahl der Stützstellen möglich. Solche Oszillation vermeidet man bei der *Spline Interpolation*, indem man die Funktion lediglich durch stückweise zusammengesetzte Polynomfunktionen niedrigen Grades approximiert. Ein *Spline* vom Grad $m \in \mathbb{N}$ ist dabei eine Funktion $s \in C^{m-1}[a,b]$, die auf jedem Intervall $[x_{i-1},x_i)$, $i = 1,\ldots,n$, zu den $n+1$ paarweise verschiedenen Knoten x_i mit einer Polynomfunktion $p_i \in \Pi^m_{\mathbb{R}}$ übereinstimmt. Zu den wichtigsten Vertretern zählen die linearen mit $m = 1$ und die kubischen Splines mit $m = 3$, siehe etwa [9].

Übungsaufgaben

IV.1 Weisen Sie nach, dass der naheliegende Summationsalgorithmus zur Auswertung eines Polynoms $p \in \mathcal{P}_{\mathbb{R}}$ mit $n = \deg(p)$ an der Stelle $w \in \mathbb{R}$ mit einem Aufwand von $3n$ Elementaroperationen $+,\cdot$ realisierbar ist.

IV.2 Implementieren Sie den Summationsalgorithmus zur Polynomauswertung aus Aufgabe IV.1 und werten Sie mit dessen Hilfe unter Verwendung von Gleitkommaarithmetik das Polynom

$$p = (1,-8,28,-56,70,-56,28,-8,1)$$

an den Stellen $w_i = 0.987 + 0.001i$ für $i = 0,\ldots,26$ aus. Vergleichen Sie das Ergebnis mit jenem, das Sie bei Verwendung von *PolyEvalHorner* erhalten.

IV.3 Implementieren Sie den Karatsuba Algorithmus zur Polynommultiplikation. Erzeugen Sie Polynome mit zufälligen Koeffizienten und vergleichen Sie die Laufzeit des Karatsuba Algorithmus mit der Standard-Multiplikation für wachsenden Grad der Polynome.

[17] Hermite, Charles: 1822–1901, französischer Mathematiker. Von ihm stammte 1873 der erste Beweis, dass e eine transzendente Zahl ist, und er löste als erster die Gleichung fünften Grades mit Hilfe elliptischer Funktionen. Nach ihm benannt sind die (orthogonalen) Hermiteschen Polynome.

IV.4 Experimentieren Sie mit dem Divisionsalgorithmus mit Gleitkomma-Koeffizienten in den Polynomen, und untersuchen Sie, wie sich Störungen in den Eingabepolynomen auf die Resultate auswirken. Verwenden Sie nach Möglichkeit eine Programmiersprache, in der Sie die Genauigkeit der Gleitkommarechnung steuern können.

IV.5 Berechnen Sie mit Hilfe des Algorithmus *GGTPolyEuklid* größte gemeinsame Teiler von Polynomen mit zufälligen rationalen Koeffizienten und untersuchen Sie die Koeffizienten der während des Algorithmus berechneten Polynome.

IV.6 Zeigen Sie, dass zu $x \in \mathbb{R}^{n+1}$ mit $x_i \neq x_j$ durch

$$\langle p,q\rangle = \sum_{i=0}^{n} \mathrm{eval}(p,x_i) \cdot \mathrm{eval}(q,x_i)$$

ein Skalarprodukt auf $\mathbb{R}[x]^n$ definiert ist, und dass die Lagrangschen Basispolynome diesbezüglich orthonormal sind.

IV.7 Entwickeln Sie einen rekursiven Algorithmus zur Berechnung der dividierten Differenzen $\Delta(x,y)$ nach (17.40) und darauf aufbauend ein Verfahren zur Berechnung der Koeffizienten des Interpolationspolynoms in Newton Darstellung. Vergleichen Sie das Laufzeitverhalten dieses Algorithmus mit jenem von Algorithmus *InterPolyNewton* hinsichtlich wachsender Anzahl von Stützstellen.

Literaturverzeichnis

[1] B. Buchberger and F. Lichtenberger. *Mathematik für Informatiker I*. Springer-Verlag, 2nd edition, 1981.

[2] J. W. Demmel. *Applied Numerical Linear Algebra*. Society for Industrial and Applied Mathematics (SIAM), 1997.

[3] P. Deuflhard and A. Hohmann. *Numerische Mathematik I*. de Gruyter, 2002.

[4] H. W. Engl, M. Hanke, and A. Neubauer. *Regularization of Inverse Problems*. Kluwer, Dordrecht Boston London, 1996.

[5] K. Geddes, S. Czapor, and G. Labahn. *Algorithms for Computer Algebra*. Kluwer Academic Press, 2nd edition, 1993.

[6] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics (SIAM), 2nd edition, 2002.

[7] D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison Wesley, 1969. Deutsche Übersetzung "Arithmetik" von R. Loos erschienen bei Springer-Verlag, 2001.

[8] H. J. Stetter. *Numerical Polynomial Algebra*. Society for Industrial and Applied Mathematics (SIAM), 2004.

[9] J. Stoer. *Einführung in die Numerische Mathematik I*. Springer-Verlag, 1972.

[10] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, 2nd edition, 2003.

[11] F. Winkler. *Polynomial Algorithms in Computer Algebra*. Springer-Verlag, 1996.

Symbolverzeichnis

$\mathtt{0}$	Nullpolynom, Seite 118
$\mathtt{0}$	Nullvektor, Seite 99
$\mathtt{1}$	Einspolynom, Seite 119
$\deg(p)$	Grad des Polynoms p, Seite 118
$a \operatorname{div} b$	Quotient bei Division mit Rest, Seite 4
$a \bmod b$	Rest bei Division, Seite 4
eps	relative Maschinengenauigkeit, Seite 15
ε_x	relativer Rundungsfehler von x, Seite 15
$\varepsilon_{x_1,x_2,\diamond}$	relativer Rundungsfehler bei $x_1 \,\tilde{\diamond}\, x_2$, Seite 16
$\operatorname{eval}(p,w)$	Wert von p an der Stelle w, Seite 121
$\operatorname{fk}(p)$	führender Koeffizient von p, Seite 118
$\mathbb{G}_{B,t,e_{\min},e_{\max}}$	Gleitkommasystem, Seite 83
$\operatorname{ggT}(a,b)$	größter gemeinsamer Teiler von a und b, Seite 64
$\operatorname{gT}(a,b)$	Menge der gemeinsamen Teiler von a und b, Seite 64
K^m	Raum der Spaltenvektoren, Seite 100
$K^{m\times n}$	Raum der$(m \times n)$-Matrizen über K, Seite 100
κ_{abs}	absolute Konditionszahl, Seite 28
κ_{abs}	absolute normweise Konditionszahl, Seite 31
κ_{rel}	relative Konditionszahl, Seite 28
κ_{rel}	relative normweise Konditionszahl, Seite 31
$\kappa_{\text{rel,komp}}$	relative komponentenweise Konditionszahl, Seite 32
$K[\mathtt{x}]$	Menge der univariaten Polynome über K, Seite 117
$K[\mathtt{x}]^n$	Menge der Polynome von Grad $\leq n$ über K, Seite 118
$L_i^{(x)}$	i-tes Lagrange-Polynom zu den Stützstellen x, Seite 137
$\Lambda^{(x)}$	Lebesgue-Konstante zu den Stützstellen x, Seite 139
$\mathcal{N}_{\mathbb{R}}^n$	Datenstruktur für Newtonsches Interpolationspolynom, Seite 144
$N_i^{(x)}$	i-tes Newton-Polynom zu den Stützstellen x, Seite 140
$N_{\max}$	größte darstellbare natürliche Zahl, Seite 53
$N_{\min}$	kleinste darstellbare natürliche Zahl, Seite 53
$\lesssim$	$\leq$ in erster Ordnung, Seite 28
$\ll$	viel kleiner, Seite 32
$\eqsim$	$=$ in erster Ordnung, Seite 28
$t \mid a$	t teilt a, Seite 64
$a \equiv_m b$	a kongruent b modulo m, Seite 69

$[a]_m$	Kongruenzklasse von a modulo m, Seite 69
A^n	Menge der n-Tupel über A, Seite 47
C_R	Stabilitätszahl bei Rückwärtsstabilität, Seite 42
C_V	Stabilitätszahl bei Vorwärtsstabilität, Seite 38
$C_{V,\text{komp}}$	Stabilitätszahl bei komponentenweiser Vorwärtsstabilität, Seite 38
$\Delta_x^y(i,i+k)$	der Koeffizient von $p_x^y(i,i+k)$ bei x^k, Seite 142
$\|\cdot\|_2$	Euklidsche Norm, Seite 30
$\|\cdot\|_\infty$	Maximumsnorm, Seite 30
p_i	i-ter Koeffizient im Polynom p, Seite 124
$p_x^y(i,j)$	Interpolationspolynom zu $x_{i:j}$ und $y_{i:j}$, Seite 141
t_i	i-te Position im Tupel t, Seite 47
$t_{i:j}$	Tupel $(t_i,\ldots,t_j)$, Seite 47
$x^{(i)}$	Wert von x im i-ten Schleifendurchlauf, Seite 11
x_i	i-te Koordinate im Vektor x, Seite 107
x^T	zu x transponierter Vektor, Seite 100
O	Landau Notation, Seite 43
o	Landau Notation, Seite 27
pf_p	zu p gehörige Polynomfunktion, Seite 122
Π_K	Menge der Polynomfunktionen über K, Seite 122
Π_K^n	Menge der Polynomfunktionen von Grad $\leq n$ über K, Seite 122
$\mathcal{P}_K$	Datenstruktur für Polynome über K, Seite 123
$\mathcal{P}_K^n$	Datenstruktur für Polynome von Grad $\leq n$ über K, Seite 124
$\mathcal{Q}$	Datenstruktur für rationale Zahlen, Seite 77
$R_{\max}$	größte normalisierte Gleitkommazahl, Seite 84
$R_{\min}$	kleinste normalisierte positive Gleitkommazahl, Seite 84
$\mathrm{T}(a)$	Menge der Teiler von a, Seite 64
$\mathbf{u}$	Rundungseinheit, Seite 16
V_K	Vektorraum über K, Seite 99
$\mathcal{V}_K^m$	Datenstruktur für Vektoren der Dimension m über K, Seite 106
ω	Wortlänge, Seite 52
x	das Polynom $(0,1,0,\ldots)$, Seite 120
$\mathcal{Z}_B$	Datenstruktur für ganze Zahlen beliebiger Länge, Seite 55
$\mathcal{Z}_m^\pm$	Datenstruktur für Kongruenzklassen, Seite 71
$\mathcal{Z}_m^+$	Datenstruktur für Kongruenzklassen, Seite 71
$Z_{\max}$	größte darstellbare ganze Zahl, Seite 53
$Z_{\min}$	kleinste darstellbare ganze Zahl, Seite 53

Index