Informatik-Spektrum (1986) 9: 174–183

Informatik Spektrum
© Springer-Verlag 1986

Sichere Ergebnisse auf Rechenanlagen*

Siegfried M. Rump

IBM Deutschland GmbH, Böblingen

Herrn Prof. Dr. William F. Ames zu seinem 60. Geburtstag gewidmet

Zusammenfassung. Die Rechenergebnisse eines Computers approximieren zwar im allgemeinen das tatsächliche Ergebnis recht gut, doch dies muß nicht der Fall sein. Das Computerergebnis kann prinzipiell beliebig weit vom wahren Ergebnis entfernt liegen, und es können sogar Ergebnisse entstehen, wenn gar keine Lösung existiert. Im vorliegenden Aufsatz wird der Frage nachgegangen, wie sichere, also mit Sicherheit richtige Ergebnisse auf der Rechenanlage zu erzielen sind. Dabei werden die wesentlichen Methoden vorgestellt und erläutert. Die besprochenen Verfahren sind sowohl für die mathematische Anwendung von Interesse als auch für ingenieurmäßige numerische Anwendungen.

Schlüsselwörter: Rundungsfehler, Arithmetik, Genaues Rechnen, Einschließung, Daten-Toleranzen, Intervalle

Summary. The results produced by a computer are in general fairly good approximations of the exact result; however, this need not be the case. The result calculated by the computer can, in fact, be randomly far from the true result, and results are sometimes even produced where in fact the given problem has no solution. The following presents different possibilities to calculate true results on a computer, i.e., results which are verified as correct, and introduces the relevant methods and explanations. The methods presented are of interest for mathematical application as well as for numerical application in technical engineering.

Key words: Rounding errors, Arithmetic, Precision, Accuracy, Inclusion, Data tolerances, Intervals

Computing Reviews Classification: G.1, G.4, B.2

* Überarbeitete Version des Aufsatzes „Mathematik auf dem Rechner", erschienen in „Computer Technik im Profil", Hrsg. H.-R. Schuchmann und H. Zemanek, R. Oldenbourg Verlag, München, Wien 1985.

1. Einleitung

Nach der ersten Einführung elektronischer Rechenanlagen setzte eine stürmische Entwicklung auf dem Gebiet der numerischen Algorithmen ein. Bei der Implementierung numerischer Methoden wurden völlig neue Erfahrungen gemacht. Vor der Einführung von Rechenanlagen waren Methoden gefragt, die sich in ein möglichst einfaches Schema fassen ließen, um dann in Rechensälen mit möglichst geringer Fehlerwahrscheinlichkeit ausgeführt werden zu können. Jetzt war das einfache Rechenschema keine conditio sine qua non mehr; und durch den Wegfall dieser Restriktion wurden auch neue Verfahren entdeckt. Aber erst durch die Einführung höherer Programmiersprachen konnte die Benutzung von Rechnern größeren Kreisen geöffnet werden; der Computer wurde gesellschaftsfähig.

Etwa zu dieser Zeit wurde auch Kritik laut an den Ergebnissen von Computerprogrammen. Das Festpunktsystem hatte sich (entwickelt aus den Analogrechnern) als zu starr erwiesen, und es wurde zum Gleitpunktsystem übergegangen. Hierdurch stellten sich aber auf einmal Fehler ein, die zwar für Spezialisten leicht erklärlich waren, deren Beseitigung jedoch zunächst außer Reichweite lag. Durch Auslöschung und Rundungsfehler konnte nämlich das auf einer Rechenanlage erzielte Ergebnis von dem tatsächlichen Ergebnis beliebig weit abweichen, wie einfache Beispiele wie etwa

$10^{20}+1-10^{20}$ oder
$9x^4-y^4+2y^2$ für $x=10864{,}0$, $y=18817{,}0$

zeigen. Im ersten Beispiel entstehen zwei Summanden $(10^{20}+1)$ und 10^{20}, die in den ersten 20 Dezimalstellen übereinstimmen. Auf jeder Rechenanlage mit weniger als 20 Dezimalen bzw. eine entsprechende Anzahl von Binär- oder Hexadezimalziffern in der Mantisse (und das trifft auf sehr viele zu) muß das Ergebnis der Subtraktion 0 statt 1 sein. Das zweite Beispiel ist ähnlich gelagert, nur nicht mehr ganz so offensichtlich.

Aus der entstandenen Unsicherheit gegenüber

Rechnerergebnissen in bestimmten Situationen entstanden verschiedene Wege zur Lösung. Mit algebraischer, exakter Rechnung wurde immer mit voller Genauigkeit, ohne Rundungsfehler gerechnet (seminumerische Algorithmen), mit symbolischer Manipulation wurde (in Verbindung mit exakter Rechnung) versucht, das Ergebnis einer Rechnung in geschlossener Form anzugeben, und mit naiver Intervallrechnung wurde versucht, Ergebnisse in Schranken einzuschließen. Aus diesen Ansätzen entwickelten sich neue Disziplinen, in denen Mathematik auf dem Rechner betrieben wird. In jüngster Zeit kam eine Richtung neuartiger numerischer Algorithmen hinzu, die mit einer einheitlichen Definition der Gleitpunkt-Arithmetik zunächst mit herkömmlichen Techniken eine Näherung bestimmen und dann auf der Basis einer Einschließungstheorie das richtige Ergebnis einschließen. Die Richtigkeit des Ergebnisses wird immer automatisch verifiziert.

2. Algebraische Manipulation

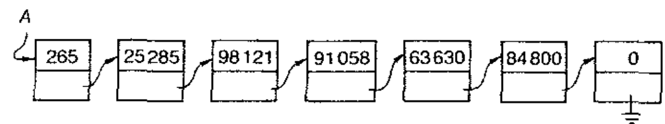
Unter „Algebraischer Manipulation" (algebraic manipulation) versteht man ganz allgemein das Rechnen in und das Manipulieren mit mathematischen Räumen, also etwa dem Körper der rationalen Zahlen. Dabei reicht das Manipulieren mit mathematischen Strukturen bereits weit in den Bereich der symbolischen Rechnung hinein (weshalb beide als "Symbolische und Algebraische Manipulation" oft in einem Atemzug genannt werden). Betrachten wir zunächst das Rechnen in mathematischen Räumen. Die einfachste mathematische Struktur im Bereich der „Algebraischen Manipulation" ist der Ring der ganzen Zahlen. Das Prädikat Ring heißt im wesentlichen, daß Assoziativ-, Kommutativ- und Distributivgesetz(e) gelten und daß zu jedem Element eine additive Inverse existiert. Allein an der Sprechweise wird sofort klar, daß man sich hier von Begriffen wie „Überlauf" oder „Rundungsfehler" zu lösen hat: Im mathematischen Modell gibt es nur ein richtiges Ergebnis und keine (beste) Approximation. Für eine interessante und umfassende Einführung in die „Symbolische und Algebraische Manipulation" auf der Basis von Original-Arbeiten siehe [3].

Ein wichtiges Problem bei der Implementierung des Rechnens im Ring der ganzen Zahlen ist die Darstellung ganzer Zahlen auf dem Rechner. Jede Zahl muß ja immer mit ihrer gesamten Anzahl von Stellen gespeichert werden, und das können (im Prinzip) beliebig viele sein. Zur Lösung des Problems wird die ganze Zahl in Abschnitte gleicher Stellenzahl geteilt und jeder einzelne solcher Abschnitte als Element einer Liste oder als Feldelement abgespeichert. Dabei benötigt man im Fall des Abspeicherns in einem Feld noch die Anzahl der Abschnitte.

Betrachten wir ein Beispiel. Es ist

$A = 30! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot 29 \cdot 30 =$
$265252859812191058636308480000000.$

Nehmen wir an, jeder Abschnitt enthält fünf Dezimalstellen. Dann wäre eine mögliche Darstellung in einer verketteten Liste



Der erste Zeiger „A" zeigt dabei auf den Anfang der Liste, auf das Element mit den am meisten signifikanten Stellen. Die Darstellung in Listen hat (in PASCAL etwa) den Vorteil, daß die Anzahl der Elemente zur Ausführungszeit vergrößert werden kann, indem Speicherplatz vom Haldenspeicher (Heap) angefordert wird. Ein Nachteil ist, daß die Listen selbst Organisationsspeicherplatz benötigen und Sorge zu tragen ist für die Verwaltung des jeweils freien Speichers im System (garbage collection). Auch bei der Speicherung in einem Feld muß diese Verwaltung durchgeführt werden.

Beim Rechnen mit ganzen Zahlen stellt sich heraus, daß im Fall von Addition, Subtraktion und Multiplikation es günstiger ist, die Liste in umgekehrter Reihenfolge aufzubauen. Nur bei der Division ist die oben angegebene Speicherung günstiger. Deshalb wird oft die Zahl in „umgekehrter Reihenfolge" abgespeichert, so daß im obigen Beispiel das erste Listenelement „0" enthält, das zweite „84800" etc. Diese Möglichkeit hat (allerdings nur unbedeutende) Nachteile bei der Vorzeichenbestimmung.

Mathematisch gesehen ist die oben beschriebene Darstellung ganzer Zahlen eine solche im Zahlsystem zur Basis 10^5. Es ist also

$A = a_0 + a_1 \cdot \beta + a_2 \cdot \beta^2 + \ldots + a_6 \cdot \beta^6$ mit $\beta = 10^5$
und $a_0 = 0,\ a_1 = 84800, \ldots, a_6 = 265$

im obigen Beispiel. Die allgemeine Darstellung einer ganzen Zahl B ist

$$B = \sum_{i=0}^{n} b_i \cdot \beta^i,$$

$0 \leq |b_i| < \beta$ für $0 \leq n$ und $b_n \neq 0$.

Ist also $\beta = 10^5$, hat B zwischen $5 \cdot n + 1$ und $6 \cdot n$ Dezimalziffern. Die gebräuchlisten Formen der Darstellung negativer Zahlen sind entweder das Vorzeichen separat zu speichern oder jeder einzelnen „Ziffer", also jeder fünfstelligen Komponente, ein Vorzeichen mitzugeben. Dabei gilt dann die Vereinbarung, daß entweder alle Ziffern nicht-negativ oder alle nicht-positiv sein müssen. Läßt man diese letzte Forderung fallen, erhält man eine redundante Zahldarstellung, auf die wir gleich noch zu sprechen kommen.

Die Algorithmen für Addition und Subtraktion ganzer Zahlen können direkt dem gewöhnlichen Rechnen, wie man es auf der Schule lernt, nachempfunden werden. Entsteht bei der Addition (oder Subtraktion) zweier Ziffern (im Zahlsystem zur Basis β) eine Ziffer, deren Betrag größer oder gleich β ist, wird nur der Anteil klei-

ner als β für diese Ziffer genommen und ein Übertrag zur nächsten Ziffer mitgenommen („eins im Sinn"). In ungünstigen Fällen kann sich ein Übertrag über mehrere, 5-stellige „Ziffern" fortpflanzen. Für $A=10^{100}$, im Zahlensystem zur Basis $\beta=10^5$ eine 21-stellige Zahl, erstreckt sich für $B=A-1$ ein Übertrag über 20 Ziffern. Bei erneuter Addition $C=B+1$ $(=A)$ ergibt sich das gleiche Bild. Um solche, sich fortpflanzenden Überträge zu vermeiden, können für die einzelnen Ziffern unterschiedliche Vorzeichen zugelassen werden. In unserem Beispiel wäre dann

$$B=A-1=1\ 0\ 0\ \dots\ 0\ -1$$

Die Fortpflanzung von Überträgen ist dadurch weitgehend vermieden, dafür wird die Zahldarstellung redundant. Denn es ist $(1)(0)(-1) = (99999)(99999) = (1)(-1)(99999)$, wobei die einzelnen, 5-stelligen „Ziffern" jeweils in Klammern geschrieben wurden. Die weitgehende Vermeidung von Überträgen bei Addition und Subtraktion wird also durch eine etwas kompliziertere Rück-Konvertierung in das gewöhnliche Zahlsystem erkauft. Besonders deutlich wird dies etwa bei $(1)(-99999)(-99999)\dots(-99999)=1$.

In einer hardwaremäßig verdrahteten Gleitpunkt-Arithmetik einer modernen Großrechenanlage werden Überträge durch ein sogenanntes „carry-look-ahead" zumindest dezimiert (vgl. [17]). Eine andere interessante Methode, um Überträge praktisch ganz zu vermeiden, ist die redundante Zahldarstellung von Avizienis (siehe [1], auch [17]). Bezeichnet man einen Additionsschritt oder einen Schritt zur Abarbeitung eines Übertrages als Zyklus, so kommt eine Addition bei dieser Art der Zahldarstellung mit 2 Zyklen aus. Diese konstante Anzahl von Zyklen kann durch paralleles Arbeiten erreicht werden. Die Möglichkeit des parallelen Rechnens liegt in der Zahldarstellung, wodurch diese Avizienissche Zahldarstellung im Zeitalter der Supercomputer wieder an Bedeutung gewinnen könnte.

Auch die Multiplikation langer ganzer Zahlen kann, gemäß der üblichen Art und Weise zu multiplizieren, programmiert werden. Sind Multiplikand und Multiplikator gleich lang und haben beide n Ziffern, so sind insgesamt n^2 Multiplikationen und n^2-1 Additionen auszuführen. Die asymptotische Rechenzeit beträgt also $0(n^2)$ Operationen (das Landausche „0" Symbol beschreibt den asymptotischen Wert für große Werte von n; dabei spielt ein konstanter Faktor keine Rolle, also $n^2=0(n^2)$ und $3n^2=0(n^2)$ aber $0{,}0001\cdot n^{2{,}01}\neq 0(n^2)$, sondern $0(n^{2{,}01})$).

Es gibt Möglichkeiten, große Zahlen schneller zu multiplizieren. Es seien zwei Zahlen, A, B gleicher Länge gegeben. Dann teile man A und B in Hälften auf:

$$A\ \underbrace{a_0\ |\ a_1} \quad * \quad B\ \underbrace{b_0\ |\ b_1}.$$

Dann ist mit einer geeigneten Potenz β^* von β

$$A=a_0\cdot\beta^*+a_1 \quad \text{und} \quad B=b_0\cdot\beta^*+b_1, \quad \text{also}$$
$$A\cdot B= a_0\beta_0\cdot b^{*2}+(a_0b_1+a_1b_0)\cdot\beta^*+a_1b_1.$$

Dabei sind vier Teilprodukte a_0b_0, a_0b_1, a_1b_0, a_1b_1 zu bilden, die je $n^2/4$ Einzelprodukte, also wieder n^2 insgesamt erfordern. Karatsuba schlug vor, durch eine andere Produktbildung die Anzahl der Teilprodukte zu vermindern (siehe [8]). Er berechnete

$$C_1=a_0b_0,\ C_2=a_1b_1 \quad \text{und}$$
$$C_3=(a_0+a_1)(b_0+b_1) \tag{1}$$

mit $a_0b_0=C_1$, $a_0b_1+a_1b_0=C_3-C_2-C_1$ und $a_1b_1=C_2$. Dadurch wurden nur $0{,}75\cdot n^2$ Teilprodukte benötigt auf Kosten zusätzlicher Additionen. Die Idee von Karatsuba war nun, diesen Vorgang zu iterieren, d. h. die Teilprodukte a_0b_0, a_1b_1 und $(a_0+a_1)\cdot(b_0+b_1)$ nach dem gleichen Schema (1) zu berechnen. Er konnte zeigen, daß mit diesem Verfahren die Rechenzeit

von $0(n^2)$ auf $0(n^{\log_2 3})=0(n^{1{,}58})$

reduziert wird. Für einen Rechner, der 1 Million Operationen pro Sekunde ausführt, ergeben sich demnach folgende Rechenzeiten:

n	normale Multiplikation	Karatsuba
100	10 ms	1,5 ms
1000	1 s	57 ms
10000	1,67 min	2,2 s

Diesen idealisierten Rechenzeiten ist noch ein gewisser Zusatzaufwand für Verwaltung etc. hinzuzurechnen. Dieser saldiert in der Praxis tatsächlich erheblich, wie aus der folgenden Tabelle ersichtlich:

n	normale Multiplikation	Karatsuba
100	42 ms	30 ms
1000	4,3 s	1,3 s
10000	7,2 min	51 s

Die absoluten Rechnerzeiten (hier für eine IBM 3081 Anlage) könnten durch Assembler-Programme noch erheblich verbessert werden, die Verhältnisse von normaler gegenüber Karatsuba-Multiplikationen noch einmal um den Faktor 2. Die theoretischen und praktischen Verhältnisse differieren hauptsächlich wegen der zusätzlichen Additionen im Karatsuba-Algorithmus, wegen des zusätzlichen Speicherbedarfs und im besonderen wegen des Organisationsaufwands. Die Vorteile beginnen in der Praxis etwa ab $n=16$.

Es gibt noch schnellere Algorithmen zur Multiplikation langer Zahlen. Entsprechend der Methode von Karatsuba können beide Zahlen in 3 oder 4 oder mehr Teile aufgeteilt werden, und es können allgemeine Formeln angegeben werden, wie die einzelnen Teile zu verknüpfen sind, um möglichst viele Multiplikationen einzusparen. Asymptotisch gesehen können so Algorithmen angegeben werden, deren Rechenzeit $0(n^{1+\varepsilon})$ ist für jedes $\varepsilon>0$. Einen tatsächlichen Zeitgewinn erzielt man dabei natürlich nur für recht große Zahlen.

Es geht aber noch schneller. Schönhage und Strassen (siehe [16]) haben einen Algorithmus angegeben mit der asymptotischen Rechenzeit

$$0(n \cdot \log n \cdot \log \log n).$$

Das Verfahren, das die schnelle Fourier-Transformation benutzt, ist auch insofern von Interesse, da Cook unter gewissen Voraussetzungen gezeigt hat, daß eine asymptotische Rechenzeit von mindestens

$$0\left(\frac{n \cdot \log n}{\log \log n}\right)$$

zur Multiplikation zweier Zahlen der Länge n notwendig ist. Der Faktor f zwischen dem Erreichten und dem maximal Erreichbaren beträgt also asymptotisch $(\log \log n)^2$. Für $n = 1000000000000$ (also für Zahlen mit je einer Billion Ziffern) etwa ist $f = 1{,}16$.

Bei der Division von ganzen Zahlen A, B ist der Quotient Q und Rest R so zu bestimmen, daß $A = Q \cdot B + R$ mit $0 \leq R < |B|$ falls A positiv und $-|B| < R \leq 0$ falls A negativ. Bei der Implementierung ergibt sich die Schwierigkeit, wie eine Ziffer des Quotienten „zu schätzen" ist. Bei der Handrechnung kommen ja für jede Quotientenziffer nur die 10 (Dezimal-)Ziffern von 0 bis 9 in Frage, so daß die richtige leicht zu bestimmen ist. Für $\beta = 10^5$ in obigem Beispiel ist aber eine 5stellige Zahl als „Quotientenziffer" zu bestimmen. Faßt man die beiden ersten Ziffern a_m, a_{m-1} des Dividenden zusammen und dividiert durch die erste Ziffer b_n des Divisors:

$$q_k^* = \lfloor (a_m \cdot \beta + a_{m-1})/b_n \rfloor,$$

so sollte q_k^* eine gute Approximation der wirklichen Ziffer q_k des Quotienten sein ($\lfloor x \rfloor$ bedeutet die größte ganze Zahl kleiner oder gleich x). Setzt man $q_k^* = \beta - 1$ falls $q_k^* \geq \beta$, so fanden Pope und Stein, daß nach Normalisierung des Divisors zu $b_n \geqslant \beta/2$ immer gelten muß

$$q_k \leqq q_k^* \leqq q_k + 2.$$

Die geschätzte (in unserem Beispiel 5stellige) Quotientenziffer ist also um maximal 2 falsch. Die Wahrscheinlichkeit für die geschätzte Quotientenziffer, genau richtig, um 1 oder um 2 falsch zu sein, ist dabei nach Collins/Musser 67%, 32% und 1% (siehe [8]). Nimmt man gar die ersten drei Ziffern des Dividenden und die ersten zwei Ziffern des Divisors mit, so kann die geschätzte Ziffer höchstens um 1 verkehrt sein; und die Wahrscheinlichkeit, daß sie falsch ist, liegt bei $1/\beta$, ist also fast 0 (siehe [8]).

Eine andere Möglichkeit der Division ist, zunächst eine Approximation der Inversen des Divisors zu bestimmen und diese mit dem Dividenden zu multiplizieren. Mit geeigneten Abschätzungen kann man zeigen, daß der Quotient immer richtig bestimmt wird (trotz der Approximation der Inversen des Divisors). Für größere Zahlen ist dieser Algorithmus der gewöhnlichen Division überlegen.

Alle internen Rechnungen bei der Implementierung der Algorithmen zur Verknüpfung ganzer Zahlen sind ganzzahlige Rechnungen. Das heißt, es entstehen keine Rundungsfehler, alle internen und externen Ergebnisse sind mathematisch exakt und richtig.

Bei der praktischen Anwendung (z. B. im Gauß-Algorithmus) fällt auf, daß die Division ganzer Zahlen nur den ganzzahligen Quotienten und Rest liefert. Der genaue Quotient hingegen ist eine rationale Zahl, die als Paar von ganzen Zahlen dargestellt werden kann. Mit den ersten drei Grundrechnungsarten für ganze Zahlen können die vier Grundrechnungsarten für rationale Zahlen erklärt werden:

$$\frac{p}{q} \pm \frac{r}{s} = \frac{ps \pm qr}{q \cdot s}; \quad \frac{p}{q} \cdot \frac{r}{s} = \frac{p \cdot r}{q \cdot s}; \quad \frac{p}{q} / \frac{r}{s} = \frac{p \cdot s}{q \cdot r}.$$

Da sich bei jedem Produkt ganzer Zahlen deren Längen addieren, verlangt man von rationalen Zahlen a/b, daß sie gekürzt sind, also $\mathrm{ggT}(a,b) = 1$, um zu rasches Anwachsen der Zahlenlängen zu vermeiden. Für die Addition rationaler Zahlen beobachtete Henrici, daß $(ps + qr)/qs$ bereits gekürzt ist, wenn $d = \mathrm{ggT}(q,s) = 1$ ist. Andernfalls braucht nur $\mathrm{ggT}(ps/d + qr/d, d)$ berechnet zu werden. Was auf den ersten Blick eher komplizierter aussieht, bringt in der Praxis eine Verbesserung von 1,5 bis 3. Eine effiziente Methode zur ggT-Berechnung geht auf D. H. Lehmer zurück [8].

Mit den bisherigen Vorbereitungen können bereits die Grundoperationen im Ring Z der ganzen Zahlen und im Körper Q der rationalen Zahlen ausgeführt werden. Hiermit können eine Reihe weiterer mathematischer Strukturen exakt auf dem Rechner dargestellt werden. Im folgenden sei ein Auszug angegeben.

Der Ring der Polynome mit ganzzahligen Koeffizienten wird mit $Z[x]$ bezeichnet, bei rationalen Koeffizienten mit $Q[x]$. Es gibt verschiedene Arten der Speicherung von Polynomen mit Grad n. Betrachten wir ein Beispiel:

$$P(x) = 3x^5 - 2x^2 + 7.$$

Entweder werden in einem $(n+2)$-Tupel der Grad und dann die Koeffizienten abgespeichert:

$$(5, 3, 0, 0, -2, 0, 7)$$

oder es werden zur jeweiligen Potenz die einzelnen Koeffizienten abgespeichert:

$$(5, 3, 2, -2, 0, 7).$$

Da zumindest Polynome in mehreren Variablen häufig schwach besetzt sind, d. h. wenige Koeffizienten ungleich 0 sind, wird bei Polynomen in mehreren Variablen meist die zweite Art der Speicherung bevorzugt. Das Rechnen mit Polynomen wird nach den mathematischen Regeln implementiert, also

$$P(x) = \sum_{i=0}^{m} p_i \cdot x^i, \; Q(x) = \sum_{i=0}^{n} q_i \cdot x^i \Rightarrow$$

$$P(x) \pm Q(x) = \sum_{i=0}^{k} (p_i \pm q_i) \cdot x^i \quad \text{und}$$

$$P(x)\cdot Q(x)=\sum_{i=0}^{m+n}\left(\sum_{j=0}^{i}p_jq_{i-j}\right)\cdot x^i$$

mit $k=\max(m,n)$ und $p_i=0$ für $i>m$, $q_i=0$ für $i>n$. Die Division von Polynomen wird gemäß der Standard-Division für Polynome implementiert. Wieder gibt es spezielle, schnellere Algorithmen für Multiplikation und Division von Polynomen und für ggT-Berechnung.

Es war lange Zeit ein offenes Problem, ob Algorithmen zur Faktorisierung von Polynomen existieren mit polynomial beschränkter Rechenzeit in Grad und Logarithmus des Betrags, also

$$0(n^k\cdot\log(|P|)^l)$$

für natürliche Zahlen k, l unabhängig von n. Erst in jüngster Zeit ist dieses Problem von Lenstra/Lenstra/Lovasz positiv entschieden worden [3].

Polynome mit rationalen Koeffizienten speichert man entsprechend denen mit ganzzahligen Koeffizienten mit dem Unterschied, daß das kleinste gemeinsame Vielfache der Nenner nur einmal gespeichert wird mit entsprechend geänderten Koeffizienten. Das führt zu einer erheblichen Speicherplatzersparnis. Das Rechnen erfolgt genauso wie mit Polynomen mit ganzzahligen Koeffizienten.

Rationale Funktionen können gespeichert werden als Paar von Polynomen, wobei das Rechnen wieder gemäß den mathematischen Regeln implementiert wird.

Nach all diesen Überlegungen fragt man sich, ob auch der exakte Umgang mit irrationalen Zahlen auf dem Rechner möglich ist. Ein einfaches Beispiel ist $\sqrt{2}$. Bekanntlich gibt es keine zwei ganze Zahlen p,q, so daß $\sqrt{2}=p/q$ ist, wie man bereits in der Schule beweist [18].

D.h., die unendlich vielen Dezimalstellen von $\sqrt{2}$ weisen keinerlei Periode auf. Trotzdem ist die Zahl $\sqrt{2}$ exakt auf dem Rechner darstellbar durch die Information, daß $\sqrt{2}$ die Nullstelle von $x^2=2$ zwischen 1 und 2 ist. Ist P ein Polynom, welches zwischen den rationalen Zahlen a und b genau eine Nullstelle besitzt, wird also durch das Tripel

$$(a,b,P) \tag{2}$$

eine (algebraische) irrationale Zahl charakterisiert. In unserem Beispiel kann das Polynom dargestellt werden durch $(2,1,0,-2)$, also $\sqrt{2}$ durch $(1,2,2,1,0,-2)$. Interessanterweise kann man mit diesen Darstellungen auch rechnen. Summe, Differenz, Produkt und Quotient algebraischer Zahlen sind ja wieder algebraische Zahlen und durch ein Tripel (2) exakt auf dem Rechner darstellbar. Somit können also beliebige algebraische Zahlen auf dem Rechner dargestellt und verknüpft werden, darunter sogar solche, die durch beliebige Wurzeln und Potenzen rationaler Zahlen nicht ausgedrückt werden können (das sind bestimmte Nullstellen von Polynomen fünften und höheren Grades).

Mit diesen algebraischen Zahlen können dann auch wieder Polynome gebildet werden, also etwa $(\sqrt{7}-1)x^2+\sqrt[3]{2,1}$. Es sei nochmals betont, daß die Wurzelausdrücke nicht etwa angenähert werden, sondern daß die exakten, mathematischen Werte auf dem Rechner wie in (2) repräsentiert werden.

Auf diese Polynome können Verfahren zur Isolierung von Nullstellen angewandt werden (z.B. Sturmsche Kette) und damit Schranken für Nullstellen berechnet werden. Diese Schranken sind dann mit Sicherheit richtig für das gegebene Ausgangspolynom mit den algebraischen Zahlkoeffizienten. Eventuelle Rundungs- oder Konvertierungsfehler, Auslöschungsfehler, gibt es nicht.

Selbst das Rechnen mit transzendenten Zahlen kann unter bestimmten Voraussetzungen exakt erfolgen. Während bei algebraischen Zahlen jedoch insbesondere die algebraische Abhängigkeit für je zwei algebraische Zahlen zwingend festgestellt werden kann, ist kein deterministischer Algorithmus bekannt, der etwa die algebraische Abhängigkeit von e und π zeigt oder widerlegt.

Neben den hier angeführten Grundprinzipien gibt es eine Reihe von Methoden, die hier nicht näher betrachtet werden können. In ganzzahligen Rechnungen mit vorwiegend Additionen, Subtraktionen und Multiplikationen, wenig Divisionen, Vorzeichen- und ggT-Berechnungen arbeitet man z.B. vorteilhaft mit modularer Arithmetik. Bei der Isolierung komplexer Nullstellen wird bei exakter Rechnung Lehmers Methode interessant usw.

3. Symbolische Manipulation

Die „Symbolische Manipulation" wäre nicht möglich ohne die „Algebraische Manipulation". Während bei letzterer z.B. ein Wert zahlenmäßig berechnet wird oder eine Nullstelle durch Schranken eingeschlossen wird, zielt die „Symbolische Manipulation" auf einen formelmäßigen Ausdruck z.B. eines Integralwertes oder auf die symbolische Faktorzerlegung eines Polynoms.

Der eigentliche Durchbruch der symbolischen Manipulation kam mit den großen interaktiven Systemen wie MACSYMA oder SCRATCHPAD. Dabei spielen vor allen Dingen eine wesentliche Rolle die symbolische Integration und die Simplifikation symbolischer Ausdrücke. In jüngerer Zeit sind solche Systeme auch auf Kleinrechnern verfügbar (z.B. MuMath).

In den Jahren 1969 und 1970 legte Risch [13] mit zwei Arbeiten, die die Liouvillesche Theorie benutzen, den Grundstein zur symbolischen Integration. Ein Beispiel ist etwa die Integration von

$$\frac{1}{x^3+2}, \tag{3}$$

entnommen aus einer MACSYMA-Demonstration von R.J. Fateman [5]. Das Ergebnis nach weniger als 2 Sekunden ist

$$-\frac{\log(x^2-2^{1/3}x+2^{2/3})}{6\cdot 2^{2/3}}+\frac{\operatorname{atan}\left(\frac{2x-2^{1/3}}{2^{1/3}\cdot\sqrt{3}}\right)}{2^{2/3}\cdot\sqrt{3}}+\frac{\log(x+2^{1/3})}{3\cdot 2^{2/3}}. \quad (4)$$

Insbesondere die zweidimensionale Anzeige im interaktiven System ist sehr hilfreich. Ein Algorithmus zur automatischen Integration arbeitet in mehreren Stufen. Zunächst wird ein Satz einfacher Regeln wie Produktintegration usw. angewandt. In vielen Fällen führt dies bereits zum Erfolg. Andernfalls werden kompliziertere Regeln wie Substitutionsregel angewandt bis schließlich hin zum allgemeinen Algorithmus von Risch. Dabei kann auch von Integralen nachgewiesen werden, daß sie nicht in geschlossener Form darstellbar sind, ein Nachweis, der ohne Rechner oft schwierig ist. Damit wird der Rechner zum mathematischen Hilfsmittel.

Bei der Differentiation von (4) (nach den üblichen Regeln, ein vergleichsweise einfacher Algorithmus) kann man nicht erwarten, sofort (3) zurückzuerhalten. In MACSYMA ergibt sich

$$\frac{1}{3\left(\frac{(2x-2^{1/3})^2}{3\cdot 2^{2/3}}+1\right)}-\frac{2x-2^{1/3}}{6\cdot 2^{2/3}\,(x^2-2^{1/3}x+2^{2/3})}+\frac{1}{3\cdot 2^{2/3}\cdot(x+2^{1/3})}, \quad (5)$$

ein Ausdruck, dem nicht ohne weiteres anzusehen ist, daß er mit (3) identisch ist. Hier kommen die Simplifikations-Algorithmen zur Anwendung. Auf diesem Gebiet hat in den letzten Jahren eine lebhafte Forschung eingesetzt und zu interessanten Ergebnissen geführt; die Grundlagen zur Simplifikation wurden in der universellen Algebra geschaffen. Zunächst mußte definiert werden, wann ein Ausdruck als „einfacher" anzusehen ist als ein anderer. Dann muß es Ziel jedes Simplifikations-Algorithmus sein, daß mathematisch äquivalente Ausdrücke immer auf denselben, „einfachsten" Ausdruck zurückgeführt werden. Hier ist leider nicht der Raum, auf Einzelheiten näher einzugehen. Das MACSYMA-System vereinfacht (5) übrigens wieder zu (3).

Obige Beispiele zur symbolischen Integration und Simplifikation sind gerechnet auf einer DEC/VAX 11/780. Doch selbst auf Kleinrechnern, auf Personal Computern, gibt es Systeme für symbolische und algebraische Manipulation, wie etwa MuMath. Von einfachen arithmetischen Rechnungen (natürlich in unendlicher Genauigkeit) bietet MuMath auch Gleichungslöser, Integration, Listenverarbeitung etc. Zum Beispiel errechnet MuMath den exakten Wert des Dreifach-Integrals

$$\int_0^{\pi}\int_0^{a}\int_0^{b}\frac{z}{2}\cdot r^3\cdot\sin(2t)\,\mathrm{d}z\,\mathrm{d}r\,\mathrm{d}t$$

schnell zu

$a^4b^2/16$.

Die Zahl π wird dabei nicht durch eine endliche Zahl approximiert, sondern als exakte, reelle Zahl behandelt und im Rechner durch # PI dargestellt.

Die symbolische Manipulation wird mehr und mehr zu einem echten Hilfsmittel des Mathematikers. Aufwendige Papierrechnungen können schnell und fehlerfrei vom Rechner ausgeführt werden. Es genügt ein Knopfdruck, um die Nullstellen von x^3-1 nicht als Gleitpunkt-Näherungen, sondern als exakte Werte

$$x_1=1,\ x_2=\frac{-1+i\sqrt{3}}{2},\ x_3=\frac{-1-i\sqrt{3}}{2}$$

ausgedruckt zu bekommen.

Neben der symbolischen Verarbeitung von Ausdrücken und der Ausführung von Operationen wie Integration gibt es noch andere Anwendungen. Durch die Zuhilfenahme von Rechenanlagen werden z. B. endliche Gruppen berechnet, Differentialgleichungen formal integriert oder sogar mathematische Sätze bewiesen. Ein interessantes Beispiel in diesem Zusammenhang ist die Quantorenelimination (über reell-abgeschlossenen Körpern). Die Aufgabe ist, aus einer Formel, bestehend aus Variablen, logischen Verknüpfungen und Quantoren, die letzteren zu eliminieren. Der erste allgemeine Algorithmus hierfür wurde von Tarski angegeben [19]; in den letzten Jahren hat Collins [4] einen wesentlich effektiveren angegeben und auch implementiert. Betrachten wir ein Beispiel (Kahan [6]). Von einer Ellipse mit Mittelpunkt a, b und Halbachsen c, d und einem Kreis um den Ursprung mit Radius r soll entschieden werden, ob sie sich schneiden oder nicht, d. h.

$$\exists x\exists y\left(x^2+y^2=r^2\wedge\frac{(x-a)^2}{c^2}+\frac{(y-b)^2}{d^2}=1\right). \quad (6)$$

Ein Algorithmus zur Elimination von Quantoren macht aus (6) eine Formel, in die man nur noch a, b, c, d, r einzusetzen braucht, um (mit mathematischer Sicherheit) festzustellen, ob Ellipse und Kreis sich schneiden oder nicht. Man braucht nicht erst auf dem Papier ein Beispiel für (6) durchzurechnen, um die Bedeutung eines solchen Algorithmus zu erkennen.

Die Symbolische und Algebraische Manipulation haben sich mittlerweile zu selbständigen Gebieten entwickelt. In der Behandlung großer numerischer Probleme (number crunching) spielen sie aus naheliegenden Gründen kaum eine Rolle. Der Speicherplatzbedarf und die Rechenzeit für große numerische Probleme ist enorm und kann durch die exakten Ergebnisse oft nicht aufgewogen werden.

4. Sichere Schranken

In den frühen 60er Jahren wurde mit einem anderen Zugang versucht, numerische Probleme schnell, ohne großen Speicherbedarf und mit Sicherheit (ohne Fehler) zu lösen: mit naiver Intervallrechnung. Ein Intervall

ist ein Paar von Zahlen $[a, b]$ mit $a \leq b$. Die mathematische Interpretation ist die Menge aller *reellen* Zahlen x mit $a \leq b$. Intervalle können verknüpft werden gemäß den folgenden Regeln:

$[a, b]+[c, d]=[a+c, b+d]$;
$[a, b]-[c, d]=[a-d, b-c]$;

$[a, b]\cdot[c, d]=[\min(a\cdot c, a\cdot d, b\cdot c, b\cdot d),$
$\max(a\cdot c, a\cdot d, b\cdot c, b\cdot d)]$;

$[a, b]/[c, d]=[\min(a/c, a/d, b/c, b/d),$
$\max(a/c, a/d, b/c, b/d)]$,

wobei im Fall der Division $0 \notin [c, d]$ vorausgesetzt ist. Für Zahlen $x \in [a, b]$ und $y \in [c, d]$ gilt offenbar immer

$x * y \in [a, b] * [c, d]$,

wobei * für +, −, · oder / steht (für einen Übersichtsartikel zur Intervallrechnung siehe [11]). Ersetzt man also in einem Algorithmus alle Zahlen durch Intervalle und alle Operationen durch die zugehörigen Intervalloperationen, müssen die Ergebnisintervalle immer die tatsächliche Lösung enthalten. Viel bringt diese so ermunternd klingende Aussage unser Problem einer Lösung zunächst nicht näher. Denn entweder (bis auf triviale Probleme) kann die Rechnung nicht bis zu Ende geführt werden, weil durch ein Intervall dividiert werden muß, welches die Null enthält, oder die Resultatintervalle sind von so großem Durchmesser, daß sie nur mehr wenig Aussagekraft besitzen.

Die Intervallrechnung hat sich zwischenzeitlich zu einer mathematischen Disziplin entwickelt und ist längst über dieses Anfangsstadium hinaus. Wegen der großen Ergebnisintervalle der eben beschriebenen sogenannten „naiven Intervallrechnung" trifft man gelegentlich noch auf eine gewisse Zurückhaltung gegenüber den Intervallen per se.

Bei *numerischen* Problemstellungen, die aus der Praxis kommen, ist die Frage nach der Richtigkeit des Ergebnisses oder der Sicherheit ohnehin zu relativieren. Häufig sind die Daten des Problems ihrerseits mit Fehlern behaftet, so daß der exakten Lösung des gestellten, des spezifizierten Problems ein anderer Stellenwert zukommt. Außerdem müssen die gegebenen Daten zunächst in das Datenformat der verwendeten Maschine gerundet werden, wodurch sich *ursprünglich gestelltes* und der Rechenanlage *spezifiziertes* Problem bereits i. a. unterscheiden. Diese Probleme der Rundung, d.h. der Verfälschung der Eingabedaten gibt es aufgrund der exakten Darstellung rationaler, algebraischer Zahlen usw. bei der Symbolischen und Algebraischen Manipulation, wenn überhaupt, nur in geringerem Maße.

Die Intervallrechnung bietet für diesen Fall die interessante Möglichkeit der Spezifikation von „ungenauen" Eingabedaten. Es werden statt Gleitpunktzahlen dann ganze Intervalle, also Toleranzen für die Eingabedaten spezifiziert; die Lösung umfaßt dann die Menge *aller* Lösungen für beliebige Daten aus den Ausgangstoleranzen. Wie vorhin geschildert, können u.U. die Ergebnisintervalle jedoch recht weit ausfallen.

Für die numerische Lösung praktischer Problemstellungen kommen noch weitere Fehlerquellen wie Modellfehler, Diskretisierungsfehler usw. in Betracht. Für die Eingrenzung der Rechnungsfehler während eines Rechenvorgangs wurde die „*Rückwärtsanalyse*" entwickelt (siehe [12]). Die Rückwärtsanalyse fragt, um wieviel das spezifizierte Problem *abgeändert* werden muß, damit die errechnete Approximation *exakte* Lösung des perturbierten Problems ist.

Ist die Lösbarkeit eines gegebenen Problems bekannt, läßt diese Sensitivitätsanalyse häufig wichtige Rückschlüsse auf die Empfindlichkeit des Problems gegenüber Datentoleranzen zu und damit darauf, ob die Problemstellung überhaupt sinnvoll ist. Sie ist daher von wichtiger, praktischer Bedeutung. Die Bestimmung des absoluten Fehlers einer Näherungslösung oder der Beweis der Lösbarkeit eines gegebenen Problems ist nicht das Ziel der Rückwärtsanalyse.

In jüngerer Zeit wurden neue Verfahren entwickelt, um schnell und mit wenig Speicherbedarf sichere Ergebnisse zu erhalten. Voraussetzung dafür war eine saubere Definition der Rechnerarithmetik, wie sie in den 70er Jahren von Kulisch gegeben wurde. Nachdem früher die Gleitpunkt-Arithmetik mehr oder weniger intuitiv implementiert wurde, gibt Kulisch eine umfassende mathematische Definition der Gleitpunkt-Arithmetik. Er beschränkt sich dabei nicht auf die einfach- oder doppeltlangen Gleitpunktzahlen als Operanden, sondern definiert die Gleitpunkt-Arithmetik für die im numerischen Rechnen auftretenden Räume (die beiden rechten Spalten in dem folgenden Schema):

		R	$\supset$	D	$\supset$	S
		VR	$\supset$	VD	$\supset$	VS
		MR	$\supset$	MD	$\supset$	MS
PR	$\supset$	IR	$\supset$	ID	$\supset$	IS
PVR	$\supset$	IVR	$\supset$	IVD	$\supset$	IVS
PMR	$\supset$	IMR	$\supset$	IMD	$\supset$	IMS
		C	$\supset$	CD	$\supset$	CS
		VC	$\supset$	VCD	$\supset$	VCS
		MC	$\supset$	MCD	$\supset$	MCS
PC	$\supset$	IC	$\supset$	ICD	$\supset$	ICS
PVC	$\supset$	IVC	$\supset$	$IVCD$	$\supset$	$IVCS$
PMC	$\supset$	IMC	$\supset$	$IMCD$	$\supset$	$IMCS$

Dabei bezeichnen R die reellen Zahlen, D die doppeltlangen und S die einfachlangen Gleitpunktzahlen, der Präfix C die komplexe Erweiterung, V die Vektoren, M die Matrizen, I die Intervallräume und P die Potenzmengen über den jeweils rechts davon stehenden Räumen. Für alle Operationen in diesen Räumen wurden effiziente Implementierungen angegeben. Dabei ist insbesondere das genaue Skalarprodukt hervorzuheben, das für *jedes* Skalarprodukt aus einfach- oder doppeltlangen Gleitpunktzahlen *die* Gleitpunktzahlen (mit Sicherheit) berechnet, die dem exakten Ergebnis am nächsten liegen, und zwar auch, wenn während der Berechnung Über- oder Unterlauf eintritt, das Endergebnis jedoch noch darstellbar ist. Alle Operationen einschließlich Skalarprodukt liefern Ergebnisse von maximaler Genauigkeit.

Es scheint sehr schwierig, wenn nicht gar unmöglich, für *beliebige* Skalarprodukte, also insbesondere Skalarprodukte beliebiger Länge, das Resultat mit maximaler Genauigkeit in annehmbarer Rechenzeit zu bestimmen. Bohlender hat hierfür verschiedene Algorithmen angegeben (siehe [2]); nachstehend soll ein besonders einfacher skizziert werden.

Um die Beschreibung nicht unnötig zu komplizieren, nehmen wir einen dezimalen Rechner an mit 5-stelliger Mantisse und Exponentenbereich -99 bis $+99$. Eine Gleitpunktzahl auf diesem Rechner hat also die Gestalt

$$\pm 0, m_1\, m_2\, m_3\, m_4\, m_5 \cdot 10^e \text{ mit } 0 \leq m \leq 9 \text{ und } -99 \leq e \leq 99 \qquad (7)$$

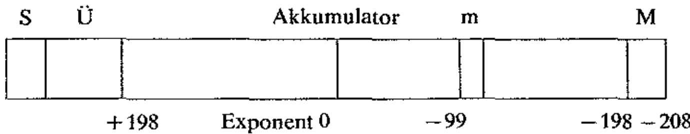
(moderne Großrechenanlagen arbeiten meist mit dualen oder hexadezimalen Zahlen, doch das macht für unsere Erörterungen keinen prinzipiellen Unterschied). Ein Skalarprodukt allgemeiner Form hat die Gestalt

$$a_1\, b_1 + a_2\, b_2 + \ldots + a_n\, b_n = \sum_{i=1}^{n} a_i\, b_i,$$

wobei die a_i, b_i Gleitpunktzahlen sind. Die größte (positive) Gleitpunktzahl in unserem System ist nach (7) $0{,}99999 \cdot 10^{99}$, die kleinste (positive, unnormalisierte) ist $0{,}00001 \cdot 10^{-99}$. Das Produkt zweier positiver Gleitpunktzahlen liegt daher im Bereich

$$0{,}0000000001 \cdot 10^{-198} \ldots 0.9999800001 \cdot 10^{198}. \qquad (8)$$

Stellt man sich jetzt einen langen Akkumulator von folgender Bauart vor:



so kann jede Gleitpunktzahl oder auch jedes Produkt zweier beliebiger Gleitpunktzahlen in diesem Akkumulator an der ihrem Exponenten entsprechenden Stelle abgespeichert werden. Der Bereich „m" bezeichnet dabei den Bereich der Mantisse einer Gleitpunktzahl mit kleinstmöglichen Exponenten, der Bereich „M" den der Mantisse des Produkts zweier Zahlen mit jeweils kleinstem Exponenten -99.

Wird der Akkumulator saldierend gestaltet, können beliebige Produkte von Gleitpunktzahlen aufaddiert werden. Es wird schnell klar, daß ein Unterlauf im Akkumulator niemals eintreten kann. Denn die kleinstmögliche vorkommende Zahl ungleich 0 in einem Produkt ist gemäß (8) $\pm 0{,}0000000001 \cdot 10^{-198} = \pm 10^{-208}$, so daß der Akkumulator als ein Festpunktsystem, jede Zahl im Akkumulator als ein ganzzahliges Vielfaches von 10^{-208} angesehen werden kann. Ein Überlauf tritt dagegen schnell ein, wenn etwa die größte Gleitpunktzahl $0{,}99999 \cdot 10^{99}$ quadriert und das Produkt zu sich selbst addiert wird. Dann wird der Exponent größer als 198.

Daher ist ein Überlaufbereich „Ü" vorgesehen mit, sagen wir, 20 Ziffern. Um in die erste Ziffer des Überlaufbereiches zu gelangen (also zu einer Zahl mit dem Exponenten 199), ist ein Skalarprodukt mindestens der Länge 2 notwendig (zum Zwecke des Überlaufs werden ja immer Produkte der größten Zahlen addiert); um die zweite Ziffer des Überlaufs zu erreichen, ist bereits ein Skalarprodukt mindestens der Länge 11 erforderlich. Um schließlich einen „Überlauf" des Überlaufbereiches zu erzeugen, also eine Zahl mit Exponenten 219, ist ein Skalarprodukt mindestens der Länge

$$10^{219} / (0{,}99999 \cdot 10^{99})^2 > 10^{21}$$

erforderlich. Nimmt man einen Rechner an, der 1 Milliarde Produkte und Additionen in den Akkumulator pro Sekunde ausführen kann, so müßte ein solcher Rechner über 30000 Jahre immer die größten Gleitpunktzahlen multiplizieren und in den Akkumulator addieren, um diesen schließlich zum Überlauf zu bringen. In praktischen Größenordnungen kann man also sagen, daß *jedes* Skalarprodukt im Akkumulator exakt berechnet wird.

In dem zusätzlichen Bereich „S", dem Statuswort, kann Information wie das Vorzeichen abgespeichert werden oder ein Zeiger, von wo bis wo der Akkumulator bisher beschrieben wurde. Die Gleitpunktzahl, die dem Inhalt des Akkumulators (das ist der *exakte* Wert des berechneten Skalarproduktes) am nächsten kommt, wird aus der führenden Ziffer (die am weitesten links stehende Ziffer ungleich Null) und den ihr folgenden Ziffern leicht ermittelt.

Durch das maximal genaue Skalarprodukt wird der Übergang von der genauen Einzeloperation zum genauen Algorithmus vorbereitet. Denn erstaunlicherweise genügen im wesentlichen $+, -, \cdot, /$ und das genaue Skalarprodukt, um in *allen* Räumen (über D und S) der Tabelle, und das sind einige hundert, die Grundoperationen mit maximaler Genauigkeit zu implementieren. Es wurde jedoch bereits durch einfache Beispiele wie

$$10^{20} + 1 - 10^{20}$$

angedeutet, daß trotz maximal genauer Einzeloperationen das Ergebnis mehrerer, zusammengesetzter Operationen stark verfälscht werden kann. Die Algorithmen zur genauen Berechnung von Skalarprodukten liefern zwar bei beliebig langen Skalarprodukten immer Ergebnisse von maximaler Genauigkeit, werden also wie eine Operation behandelt und maximal genau ausgewertet. Das gilt jedoch nur für Skalarprodukte, also Summen von Produkten, aber nicht notwendig für andere Kombinationen von Einzeloperationen. Beim Gaußschen Algorithmus etwa zur Lösung linearer Gleichungssysteme treten alle vier Grundrechenarten $+, -, \cdot, /$ ständig gemischt auf. Trotz maximal genauer Einzeloperationen kann das Endergebnis beliebig verfälscht werden.

Den Ansatz von Kulisch zur Definition der Rechnerarithmetik, der bis in das Jahr 1969 zurückreicht,

darf man wohl den ersten nennen, der eine Rechnerarithmetik rigoros theoretisch fundiert und bis zur praktischen Implementierung geführt hat. Vor kurzem hat der IEEE 754-Standard eine Rechnerarithmetik beschrieben. Gegenüber dem früheren Zustand, daß von den großen Rechnerherstellern kaum eine klare Stellungnahme über die Arbeitsweise der Arithmetik in ihren Anlagen zu erhalten war, ist diese standardmäßige Festlegung der Arithmetik bereits ein gewaltiger Fortschritt. Dadurch wird das Ergebnis jeder einzelnen Operation exakt vorhersagbar. Leider bleibt dieser Ansatz in punkto Skalarprodukt hinter dem vorhin beschriebenen zurück. Die Definition des internen Formats ermöglicht zwar prinzipiell die Implementierung eines langen Akkumulators, zu einer Aufnahme des Skalarproduktes als fünfte Operation (neben $+, -, \cdot, /$) konnte man sich jedoch nicht entschließen.

In jüngerer Zeit wurden Verfahren für Standardprobleme der Numerik entwickelt, in denen numerische Problemstellungen wie lineare Gleichungssysteme als ganzes betrachtet und gelöst werden. Grundlage der neuen Verfahren ist eine neue mathematische Theorie, eine Einschließungstheorie [14, 15, 7]. Die Sätze der Theorie sind so formuliert, daß die Richtigkeit der Voraussetzungen auf dem Rechner verifiziert werden kann. Natürlich benötigt man zu dieser Verifikation eine präzis definierte Rechnerarithmetik, vorzugsweise eine solche mit möglichst hoher Genauigkeit. Sind die Voraussetzungen eines Satzes der Einschließungstheorie erst einmal verifiziert, gelten seine Folgerungen. Und diese besagen dann, daß die Lösung des gestellten Problems existiert und innerhalb einer berechneten Einschließung sogar eindeutig ist. Die Verifikationen der Voraussetzungen kann in einfacher oder doppelter Rechengenauigkeit erfolgen.

Diese Algorithmen weisen also zunächst nach, daß das gegebene Problem lösbar ist, und berechnen dann absolute Fehlerschranken für die Lösung. Wie oben angedeutet, können auch mit Toleranzen behaftete Eingabedaten direkt verarbeitet werden in geringfügig höherer Rechenzeit. Dabei wird dann nachgewiesen, daß jedes der (unendlich vielen) Probleme innerhalb der Toleranzen lösbar ist, und die Lösung aller dieser Probleme wird eingeschlossen. Dadurch ist in gewisser Weise auch eine Sensitivitätsanalyse möglich, indem gegebene Daten mit geringfügigen Toleranzen versehen werden (in der Größenordnung der relativen Rechnungsfehlereinheit), und an den Toleranzen der Lösungsmenge kann dann die Empfindlichkeit des Problems abgelesen werden. Diese Art der Sensitivitätsanalyse unterscheidet sich von der Monte-Carlo-Methode (hier werden Daten zufällig aus den Eingabetoleranzen ausgewählt, das zugehörige Problem gelöst und aus der Streuung der Lösungen auf die Empfindlichkeit des Ausgangsproblems geschlossen) dadurch, daß in einem mathematischen Sinne *alle* Kombinationen von Eingabedaten erfaßt werden statt einer Auswahl, andererseits jedoch Abhängigkeiten in den Eingabedaten schwieriger zu berücksichtigen sind. Dafür benötigt die Monte-Carlo-Methode, je nach Anzahl von Testbeispielen, u. U. um Größenordnungen mehr Rechenzeit als die Einschließungsalgorithmen.

In den auf der Einschließungstheorie beruhenden Algorithmen wird zunächst eine Approximation für eine Lösung des gestellten Problems berechnet. Die Approximation kann mit herkömmlichen numerischen Verfahren berechnet werden; es sind keinerlei Voraussetzungen an die Genauigkeit der Approximation notwendig. Um die berechnete Approximation wird ein potentielles Einschließungsintervall gelegt. Die Einschließungstheorie gibt notwendige und hinreichende Bedingungen dafür an, daß die potentielle Einschließung korrekt ist. Diese Bedingungen sind auf dem Rechner überprüfbar, und so können Ergebnisse auf dem Rechner verifiziert werden. Sollte das erste, potentielle Einschließungsintervall sich als nicht korrekt erweisen, gibt die Einschließungstheorie ein konstruktives Verfahren an zur Gewinnung neuer Einschließungsintervalle. Dieses Verfahren ist so effizient, daß normalerweise nach 2, spätestens 3 Iterationen eine Einschließung erzielt wird. Es kann z. B. für lineare Gleichungssysteme nachgewiesen werden [15], daß genau dann eine Einschließung erzielt wird, wenn die Residueniteration konvergiert. Das ist ein bestmögliches Ergebnis. Sollte das gestellte Problem nicht lösbar sein, wird keine Einschließung berechnet, sondern eine entsprechende Meldung ausgegeben. Umgekehrt kann in einer Gleitpunktiteration eine Konvergenz vorgetäuscht werden, wenn in Wirklichkeit gar keine Lösung existiert [14].

Die Sätze der Einschließungstheorie wurden zunächst mit Hilfe des Brouwerschen Fixpunktsatzes bewiesen. Dementsprechend wurden auch die Einschließungsmengen als konvex vorausgesetzt. Neuerdings gelangen die Beweise ohne Zuhilfenahme eines der traditionellen Fixpunktsätze [15]. Dadurch können die Einschließungsmengen jetzt auch nicht-konvex ausfallen.

Die neuen Algorithmen stellen einen Durchbruch dar. In einfacher oder doppelter Genauigkeit wird in einer Rechenzeit von der Größenordnung wie der eines vergleichbaren Gleitpunkt-Algorithmus die Lösung des gestellten Problems eingeschlossen (letzterer natürlich ohne bewiesenermaßen richtige Einschließung). Der entscheidende Unterschied zu den Verfahren der naiven Intervallrechnung ist der folgende: Es wird nicht von der Problemstellung ausgehend der Fehler jeder weiteren Rechenoperation abgeschätzt und dadurch die Schranken für die Fehler immer weiter vergröbert, sondern es wird immer wieder das *ursprüngliche* Problem in die Rechnung mit einbezogen. Denn das ist die einzige *exakt* gegebene Information. Dadurch gelingt es, die Abschätzungen immer wieder zu verfeinern und schließlich Schranken hoher oder maximaler Genauigkeit zu erzielen.

Theoretisch sind die neuen Algorithmen um den konstanten Faktor 6 langsamer gegenüber einem kon-

ventionellen Algorithmus. Praktisch ist dieser Faktor von der Hardware-Konfiguration abhängig und z. B. auf einem IBM 4361-Prozessor deutlich geringer. Als Faustregel könnte man sagen, daß die neuen Algorithmen in einfacher Genauigkeit etwa so viel Zeit benötigen wie ein herkömmlicher Algorithmus, ausgeführt in einfacher und doppelter Genauigkeit. Letzteres Verfahren wird häufig angewandt, um einen Anhaltspunkt über die Genauigkeit zu gewinnen.

Bei den neuen Algorithmen sind die berechneten Schranken von hoher Genauigkeit, meist sogar von maximaler Genauigkeit, d. h. linke und rechte Grenze aller Ergebnisintervalle sind aufeinanderfolgende Gleitpunktzahlen und die Einschließungen damit bestmöglich. Die bisher erstellten Programmpakete umfassen quadratische, über- und unterbestimmte und spärlich besetzte lineare Gleichungssysteme, Invertierung von Matrizen, Eigenwertprobleme, Nullstellen von Polynomen, nicht-lineare Gleichungssysteme, Auswertung arithmetischer Ausdrücke (mathematische Funktionen), lineare, quadratische und konvexe Optimierungsprobleme, numerische Quadratur bis hin zu Anfangs- und Randwertaufgaben bei gewöhnlichen Differentialgleichungen. Die Arithmetik und ein Teil der Algorithmen wurde 1984 von der Firma IBM als Programmprodukt „ACRITH" auf den Markt gebracht, von dem im Oktober 1985 die dritte Fassung angekündigt wurde.

Literatur

In der nachfolgend angegebenen Literatur finden sich zahlreiche weitere Verweise auf Arbeiten zum Thema „Computer Algebra und Neue Arithmetik" sowie auf Originalaufsätze. Eine kleine Sammlung von Beispielen, die die Schwächen herkömmlicher Rechnerarithmetiken demonstrieren, finden sich im Aufsatz des Autors „Wie zuverlässig sind die Ergebnisse unserer Rechenanlagen?" im Jahrbuch Überblikke Mathematik 1983, Bibliographisches Institut, Mannheim.

1. Avizienis, A.: Signed-Digit Number Representations for Fast Parallel Arithmetic, IEEE Trans. Elect. Comput. *3*, 389 (1961)
2. Bohlender, G.: Genaue Summation von Gleitkommazahlen. Computing, Suppl. 1, 21 (1977)
3. Buchberger, B., Collins, G. E., Loos, R.: Computer Algebra. Wien-New York: Springer 1982
4. Collins, G. E.: Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decomposition. Proc. EUROSAM 1974, SIGSAM Bull. *8*, No. 3, 80 (1974)
5. Fateman, R. J.: Symbolic and Algebraic Computer Programming Systems. SIGSAM Bull. *15*, No. 1 (1981)
6. Kahan, W.: An Ellipse Problem. SIGSAM Bull. *9*, No. 3 (1975)
7. Kaucher, E.: Solving Function Space Problems with Guaranteed close bounds, in: A New Approach to Scientific Computation, August 1982, Hrsg.: U. W. Kulisch, W. L. Miranker; New York: Academic Press 1983
8. Knuth, D. E.: The Art of Computer Programming, Vol. 2. Seminumerical Algorithms. Reading, Mass.: Addison Wesley 1981
9. Kulisch, U. W.: Grundlagen des numerischen Rechnens (Reihe Informatik, 19). Mannheim-Wien-Zürich: Bibliographisches Institut 1976
10. Kulisch, U. W., Miranker, W. L.: Computer Arithmetic in Theory and Practice. New York: Academic Press 1981
11. Nickel, K.: Interval Analysis, in: State of the Art in Numerical Analysis. Hrsg.: D. Jacobs. New York: Academic Press 1977
12. Reinsch, Ch.: Die Behandlung von Rechnungsfehlern in der numerischen Analysis. Jahrbuch Überblick Mathematik 1979. Mannheim-Wien-Zürich: Bibliographisches Institut 1979
13. Risch, R.: The problem of integration in finite terms, Trans. AMS *139*, 167 (1969)
14. Rump, S. M.: Solving algebraic Problems with High Accuracy, in: A New Approach to Scientific Computation, Hrsg.: U. W. Kulisch, W. L. Miranker. New York: Academic Press 1983
15. Rump, S. M.: New Results on Verified Inclusions, to appear in Springer Lecture Notes (1986)
16. Schönhage, A., Strassen, V.: Schnelle Multiplikation großer Zahlen, Computing *7*, 281 (1971)
17. Spaniol, O.: Arithmetik in Rechenanlagen, Stuttgart: Teubner 1976
18. Strubecker, K.: Einführung in die höhere Mathematik, Band I, München: Oldenbourg 1956
19. Tarski, A.: A Decision Method for Elementary Algebra and Geometry, Berkeley: Univ. of California Press 1951
20. Wissenschaftliches Rechnen und Programmiersprachen. Hrsg.: U. Kulisch, Ch. Ullrich (Berichte des German Chapter of ACM 10). Stuttgart: Teubner 1982

Eingegangen 19. 3. 1986

Priv.-Doz. Dr. S. M. Rump
IBM Deutschland GmbH
Schönaicher Straße 220
D-7030 Böblingen