# Kapitel 1
# Einführung

## § 1: Was ist Computeralgebra

Sobald kurz nach dem zweiten Weltkrieg die ersten Computer an Universitäten auftauchten, wurden sie von Mathematikern nicht nur zum numerischen Rechnen eingesetzt, sondern auch für alle anderen Arten mathematischer Routinearbeiten, genau wie auch schon früher alle zur Verfügung stehenden Mittel benutzt wurden: Beispielsweise konstruierte D.H. LEHMER bereits vor rund achtzig Jahren, lange vor den ersten Computern, mit Fahrradketten Maschinen, die (große) natürliche Zahlen in ihre Primfaktoren zerlegen konnten.

Computer manipulieren Bitfolgen; von den meisten Anwendern wurden diese zur Zeit der ersten Computer zwar als Zahlen interpretiert, aber wie wenig später selbst die Buchhalter bemerkten, können sie natürlich auch Informationen ganz anderer Art darstellen. Deshalb wurden bereits auf den ersten Computern (deren Leistungsfähigkeit nach heutigen Standards nicht einmal der eines programmierbaren Taschenrechners entspricht) algebraische, zahlentheoretische und andere abstrakt mathematische Berechnungen durchgeführt wurden. Programmiert wurde meist in Assembler, da die gängigen höhere Programmiersprachen der damaligen Zeit (FORTRAN, ALGOL 60, COBOL, . . . ) vor allem mit Blick auf numerische *bzw.*, im Fall von COBOL, betriebswirtschaftliche Anwendungen konzipiert worden waren.

Eine Ausnahme bildete die 1958 von JOHN MCCARTHY entwickelte Programmiersprache LISP, die speziell für symbolische Manipulation entwickelt wurde, vor allem solche im Bereich der künstlichen Intelligenz. In dieser Sprache wurden Ende der Sechzigerjahre die ersten Computeralgebrasysteme geschrieben: MACSYMA ab 1968 ebenfalls am M.I.T. zunächst vor allem für alle Arten von symbolischen Rechnungen in Forschungsprojekten des M.I.T., REDUCE ungefähr gleichzeitig von ANTHONY C. HEARN vor allem für Berechnungen in der Hochenergiephysik.



Beide Systeme verbreiteten sich schnell an den Universitäten und wurden bald auch schon für eine Vielzahl anderer Anwendungen benutzt; dies wiederum führte zur Weiterentwicklung der Systeme sowohl durch die ursprünglichen Autoren als auch durch Benutzer, die neue Pakete hinzufügten, und es führte auch dazu, daß anderswo neue Computeralgebrasysteme entwickelt wurden, wie beispielsweise Maple an der University of Waterloo (einer der Partneruniversitäten von Mannheim). Mit der zunehmenden Nachfrage lohnte es sich auch, deutlich mehr Arbeit in die Entwicklung der Systeme zu stecken, so daß die neuen Systeme oft nicht mehr in LISP geschrieben waren, sondern in klassischen Programmiersprachen wie MODULA oder C *bzw.* später C++, die zwar für das symbolische Rechnen einen erheblich höheren Programmieraufwand erfordern als LISP, die dafür aber auch zu deutlich schnelleren Programmen führen.

Eine gewisse Zäsur bedeutete das Auftreten von *Mathematica* im Jahr 1988. Dies ist das erste System, das von Anfang an rein kommerziell entwickelt wurde. Der Firmengründer und Initiator STEVE WOLFRAM kommt zwar aus dem Universitätsbereich (bevor er seine Firma gründete, forschte er am *Institute for Advanced Studies* in Princeton über zelluläre Automaten), aber *Mathematica* war von Anfang an gedacht als ein Produkt, das an Naturwissenschaftler, Ingenieure und Mathematiker *verkauft* werden sollte. Ein wesentlicher Aspekt, der aus Sicht dieser Zielgruppe den Kauf von *Mathematica* attraktiv machte, obwohl zumindest damals noch eine ganze Reihe anderer Systeme frei oder gegen nominale Gebühr erhältlich waren, bestand in der Möglichkeit, auf einfache Weise Graphiken zu erzeugen. Bei den ersten Systemen hatte dies nie eine Rolle gespielt, da Graphik damals nur über teure Plotter und (zumindest in Universitätsrechenzentrum) mit Wartezeiten von rund einem Tag erstellt werden konnte. 1988 gab es bereits PCs mit (damals

noch sehr schwachen) graphikfähigen Bildschirmen, und Visualisierung spielte plötzlich in allen Wissenschaften eine erheblich größere Rolle als zuvor.

Der Nachteil der ersten *Mathematica*-Versionen war eine im Vergleich zur Konkurenz ziemlich hohe Fehlerquote bei den mathematischen Berechnungen. (Perfekt ist in diesem Punkt auch heute noch kein Computeralgebrasystem.) Der große Vorteil der einfachen Erzeugung von Graphiken sowie das sehr gute Begleitbuch von STEVE WOLFRAM, das deutlich über dem Qualitätsniveau auch heute üblicher Softwaredokumentation liegt, bescherte *Mathematica* einen großen Erfolg. Da auch Systeme wie MACSYMA und MAPLE mittlerweile in selbständige Unternehmen ausgegliedert worden waren, führte die Konkurenz am Markt schnell dazu, daß Graphik auch ein wesentlicher Bestandteil anderer Computeralgebrasysteme wurde und daß *Mathematica* etwas vorsichtiger mit den Regeln der Mathematik umging; heute unterscheiden sich die beiden kommerziell dominanten Systeme Maple und *Mathematica* nicht mehr wesentlich in ihren Graphikfähigkeiten und ihrer (geringen, aber bemerkbaren) Häufigkeit mathematischer Fehler. Hinzu kam der Markt der Schüler und Studenten, so daß ein am Markt erfolgreiches Computeralgebrasystem auch in der Lage sein muß, die Grundaufgaben der Schulmathematik und der Mathematikausbildung zumindest der ersten Semester der gefragtesten Studiengänge zu lösen.

Da die meisten, die mit dem Begriff *Computeralgebra* überhaupt etwas anfangen können, an Computeralgebrasysteme denken, hat sich dadurch auf die Bedeutung des Worts *Computeralgebra* verändert: Gemeinhin versteht man darunter nicht mehr nur ein Programm, das symbolische Berechnungen ermöglicht, sondern eines, das über ernstzunehmende Graphikfähigkeiten verfügt und viele gängige Aufgabentypen lösen kann, ohne daß der Benutzer notwendigerweise versteht, wie man solche Aufgaben löst.

Hier in der Vorlesung wird es in erster Linie um die Algorithmen gehen, die hinter solche System stehen, insbesondere denen, die sich mit der klassischen Aufgabe des symbolischen Rechnens befassen. In den Übungen wird es allerdings zumindest auch teilweise darum gehen,



Computeralgebrasysteme effizient einzusetzen auch zur Visualisierung mathematischer Sachverhalte.

## § 2: Numerisches Rechnen, exaktes Rechnen und symbolisches Rechnen

Numerisches Rechnen gilt gemeinhin als das Rechnen mit reellen Zahlen. Kurzes Nachdenken zeigt, daß wirkliches Rechnen mit reellen Zahlen weder mit Papier und Bleistift noch per Computer wirklich möglich ist: Die Menge $\mathbb{R}$ der reellen Zahlen ist schließlich überabzählbar, aber sowohl unsere Gehirne als auch unsere Computer sind endlich. Der Datentyp **real** oder **float** oder auch **double** kann daher unmöglich das Rechnen mit reellen Zahlen exakt wiedergeben.

Tatsächlich genügt das Rechnen mit reellen Zahlen per Computer völlig anderen Regeln als denen, die wir vom Körper der reellen Zahlen gewohnt sind. Zunächst einmal müssen wir uns notgedrungen auf eine endliche Teilmenge von $\mathbb{R}$ beschränken; in der Numerik sind dies traditionellerweise die sogenannten Gleichtkommazahlen.

Eine Gleitkommazahl wird dargestellt in der Form $x = \pm m \cdot b^{\pm e}$, wobei die *Mantisse* $m$ zwischen 0 und 1 liegt und der *Exponent* $e$ eine ganze Zahl aus einem gewissen vorgegebenen Bereich ist. Die Basis $b$ ist in heutigen Computern gleich zwei, in einigen alten Mainframe Computern sowie in vielen Taschenrechnern wird auch $b = 10$ verwendet.

Praktisch alle heute gebräuchliche CPUs für Computer richten sich beim Format für $m$ und $e$ nach dem IEEE-Standard 754 von 1985. Hier ist $b = 2$, und einfach genaue Zahlen werden in einem Wort aus 32 Bit gespeichert. Das erste dieser Bits steht für das Vorzeichen, null für positive, eins für negative Zahlen. Danach folgen acht Bit für den Exponenenten $e$ und 23 Bit für die Mantisse $m$.

Die acht Exponentenbit können interpretiert werden als eine ganze Zahl $n$ zwischen 0 und 255; wenn $n$ keinen der beiden Extremwerte 0 und 255 annimmt, wird das Bitmuster interpretiert als die Gleitkommazahl (Mantisse im Zweiersystem)

$$\pm 1{,}m_1 \ldots m_{23} \times 2^{n-127} .$$

Die Zahlen, die in obiger Form dargestellt werden können, liegen somit zwischen $2^{-126} \approx 1{,}175 \cdot 10^{-37}$ und $(2 - 2^{-23}) \cdot 2^{127} \approx 3{,}403 \cdot 10^{38}$. Das führende Bit der Mantisse ist stets gleich eins (sogenannte normalisierte Darstellung) und wird deshalb gleich gar nicht erst abgespeichert. Der Grund liegt natürlich darin, daß man ein führendes Bit null durch Erniedrigung des Exponenten zum Verschwinden bringen kann – es sei denn, man hat bereits den niedrigstmöglichen Exponenten $n = 0$, entsprechend $e = -127$.

Für $n = 0$ gilt daher eine andere Konvention: Jetzt wird die Zahl interpretiert als

$$\pm 0{,}m_1 \ldots m_{23} \times 2^{-126};$$

man hat somit einen (unter Numerikern nicht unumstrittenen) *Unterlaufbereich* aus sogenannten *subnormalen* Zahlen, in dem mit immer weniger geltenden Ziffern Zahlen auch noch positive Werte bis hinunter zu $2^{-23} \times 2^{-126} = 2^{-149} \approx 1{,}401 \cdot 10^{-44}$ dargestellt werden können, außerdem natürlich die Null, bei der sämtliche 32 Bit gleich null sind.

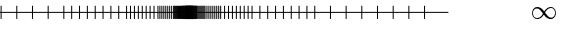
Auch der andere Extremwert $n = 255$ hat eine Sonderbedeutung: Falls alle 23 Mantissenbit gleich null sind, steht dies je nach Vorzeichenbit für $\pm\infty$, andernfalls für NAN *(not a number)*, d.h das Ergebnis einer illegalen Rechenoperation wie $\sqrt{-1}$ oder $0/0$. Das Ergebnis von $1/0$ dagegen ist nicht NAN, sondern $+\infty$, und $-1/0 = -\infty$.

Doppeltgenaue Gleitkommazahlen werden entsprechend dargestellt; hier stehen insgesamt 64 Bit zur Verfügung, eines für das Vorzeichen, elf für den Exponenten und 52 für die Mantisse. Durch die elf Exponentenbit können ganze Zahlen zwischen null und 2047 dargestellt werden; abgesehen von den beiden Extremfällen entspricht dies dem Exponenten $e = n - 1023$.

Der Exponent $e$ sorgt dafür, daß Zahlen aus einem relativ großen Bereich dargestellt werden können, er hat aber auch zur Folge, daß die Dichte der darstellbaren Zahlen in den verschiedenen Größenordnung stark variiert: Am dichtesten liegen die Zahlen in der Umgebung der Null, und mit steigendem Betrag werden die Abstände benachbarter Zahlen immer größer.



Um dies anschaulich zu sehen, betrachten wir ein IEEE-ähnliches Gleitkommasystem mit nur sieben Bit, einem für das Vorzeichen und je drei für Exponent und Mantisse. Das folgende Bild zeigt die Verteilung der so darstellbaren Zahlen (mit Ausnahme von NAN):




Um ein Gefühl dafür zu bekommen, was dies für das praktische Rechnen mit Gleitkommazahlen bedeutet, betrachten wir ein analoges System mit der uns besser vertrauten Dezimaldarstellung von Zahlen (für die es einen eigenen IEEE-Standard 854 von 1987 gibt), und zwar nehmen wir an, daß wir eine dreistellige dezimale Mantisse haben und Exponenten zwischen -3 und 3. Da es bei einer von zwei verschiedenen Basis keine Möglichkeit gibt, bei einer normalisierten Mantisse die erste Ziffer einzusparen, schreiben wir die Zahlen in der Form $\pm 0{,}m_1m_2m_3 \cdot 10^e$.

Zunächst einmal ist klar, daß die Summe zweier Gleitkommazahlen aus diesem System nicht immer als Gleitkommazahl im selben System darstellbar ist: Ein einfaches Gegenbeispiel wäre die Addition der größten darstellbaren Zahl $0{,}999 \cdot 10^3 = 999$ zu $5 = 0{,}5 \cdot 10^1$: Natürlich ist das Ergebnis 1004 nicht mehr im System darstellbar. Der IEEE-Standard sieht vor, daß in so einem Fall eine *overflow*-Bedingung gesetzt wird und das Ergebnis gleich $+\infty$ wird. Wenn man (wie es die meisten Compiler standardmäßig tun) die *overflow*-Bedinung ignoriert und mit dem Ergebnis $+\infty$ weiterrechnet, kann dies zu akzeptablen Ergebnissen führen: Beispielsweise wäre die Rundung von $1/(999 + 5)$ auf die Null für viele Anwendungen kein gar zu großer Fehler, auch wenn es dafür in unserem System die sehr viel genauere Darstellung $0{,}996 \cdot 10^{-3}$ gibt. Spätestens wenn man das Ergebnis mit 999 multipliziert, um den Wert von $999/(999 + 5)$ zu berechnen, sind die Konsequenzen aber katastrophal: Nun bekommen wir eine Null anstelle von $0{,}996 \cdot 10^0$. Ähnlich sieht es auch aus, wenn wir anschließend 500 subtrahieren: $\infty - 500 = \infty$, aber $(999 + 5) - 500 = 504$ ist eine Zahl, die sich in unserem System sogar exakt darstellen ließe!

Auch ohne Bereichsüberschreitung kann es Probleme geben: Beispiels-

weise ist

$$123 + 0{,}0456 = 0{,}123 \cdot 10^3 + 0{,}456 \cdot 10^{-1} = 123{,}0456$$

mit einer nur dreistelligen Mantisse nicht exakt darstellbar. Hier sieht der Standard vor, daß das Ergebnis zu einer darstellbaren Zahl gerundet wird, wobei mehrere Rundungsvorschriften zur Auswahl stehen. Voreingestellt ist üblicherweise eine Rundung zur nächsten Maschinenzahl; wer etwas anderes möchte, kann dies durch spezielle Bits in einem Prozessorstatusregister spezifizieren. Im Beispiel würde man also $123+0{,}0456 = 123$ oder (bei Rundung nach oben) 124 setzen und dabei zwangsläufig einen Rundungsfehler machen.

Wegen solcher unvermeidlicher Rundungsfehler gilt das Assoziativgesetz selbst dann nicht, wenn es keine Bereichsüberschreitung gibt: Bei Rundung zur nächsten Maschinenzahl ist beispielsweise

$$(0{,}456\cdot 10^0 + 0{,}3\cdot 10^{-3}) + 0{,}4\cdot 10^{-3} = 0{,}456\cdot 10^0 + 0{,}4\cdot 10^{-3} = 0{,}456\cdot 10^0,$$

aber

$$0{,}456\cdot 10^0 + (0{,}3\cdot 10^{-3} + 0{,}4\cdot 10^{-3}) = 0{,}456\cdot 10^0 + 0{,}7\cdot 10^{-3} = 0{,}457\cdot 10^0.$$

Ein mathematischer Algorithmus, dessen Korrektheit unter Voraussetzung der Körperaxiome für $\mathbb{R}$ bewiesen wurde, muß daher bei Gleitkommarechnung kein korrektes oder auch nur annähernd korrektes Ergebnis mehr liefern – ein Problem, das keinesfalls nur theoretische Bedeutung hat.

In der numerischen Mathematik ist dieses Problem natürlich schon seit Jahrzehnten bekannt; das erste Buch, das sich ausschließlich damit beschäftigte, war

J.H. WILKINSON: Rounding errors in algebraic processes, *Prentice Hall*, 1963; Nachdruck bei *Dover*, 1994.

Heute enthält fast jedes Lehrbuch der Numerischen Mathematik entsprechende Abschnitte; zwei neuere Bücher in denen es speziell um diese Probleme, ihr theoretisches Verständnis und praktische Algorithmen geht, sind

FRANÇOISE CHAITIN-CHATELIN, VALÉRIE FRAYSSÉ: Lectures on finite precision computations, SIAM, 1996



sowie das sehr ausführlichen Buch

NICHOLAS J. HIGHAM: Accuracy and stability of numerical algorithms, SIAM, 1996.

Eine ausführliche und elementare Darstellung der IEEE-Arithmetik und des Umgangs damit findet man in

MICHAEL L. OVERTON: Numerical Computing with IEEE Floating Point Arithmetic – *Including One Theorem, One Rule of Thumb and One Hundred and One Exercises*, SIAM, 2001.

## §3: Unentscheidbarkeitsprobleme

Ein auch nur moderat koplizierter symbolischer Ausdruck läßt sich praktisch immer auf eine Vielzahl von Arten darstellen, die teils offensichtlich gleich sind, teils aber auch auf den ersten Blick nichts miteinander zu tun haben. Einige Beispiele:

$$\frac{10}{15} = \frac{2}{3}, \quad \sqrt{8} = 2\sqrt{2}, \quad \sqrt{4+2\sqrt{3}} = 1+\sqrt{3}$$

$$(a+b)^2 = a^2 + 2ab + b^2, \quad \frac{x^5-1}{x-1} = 1 + x + x^2 + x^3 + x^4,$$

$$X^5 - 15X^4 + 85X^3 - 225X^2 + 274X - 120$$
$$= (X-1)(X-2)(X-3)(X-4)(X-5),$$

$$\sin x \cos x = \frac{\sin 2x}{2}, \quad 1 + \tan^2 x = \frac{1}{\cos^2 x}$$

Nur in wenigen dieser Fälle ist eine der beiden Darstellung für alle Arten von Anwendungen der anderen vorzuziehen; meist hat mal die eine, mal die andere Form ihre Vorteile.

Andererseits gehört es zu den Grundaufgaben jeglicher Art des Rechnens, daß man entscheiden muß, ob zwei Ausdrücke gleich sind. Dies ist dann am einfachsten, wenn jeder Ausdruck intern durch eine eindeutig bestimmte kanonische Form dargestellt wird. In einem System, daß alle Ergebnisse auf eine solche kanonische Form bringt, lassen sich zwei Ausdrücke einfach dadurch auf Gleichheit testen, daß man ihre

Differenz berechnet; die Ausdrücke sind genau dann gleich, wenn das Ergebnis die kanonische Darstellung der Null ist.

Gegen eine solche Darstellung sprechen sowohl theoretische als auch praktische Gründe: Wenn beispielsweise Polynome stets in ausmultiplizierter Form dargestellt werden, läuft man Gefahr, ein als Produkt von Linearfaktoren gegebenes Polynom zunächst auszumultiplizieren, um dann anschließend mit großer Mühe seine Nullstellen zu bestimmen. Stellt man Polynome dagegen in faktorisierter Form da, so kann es passieren, daß ein als Summe von Potenzen gegebenes Polynom zunächst mit großem Aufwand faktorisiert wird, und wir anschließend beispielsweise eine Stammfunktion suchen, wofür diese Faktorisierung wieder rückgängig gemacht werden muß. Das Ergebnis müßte dann wieder faktorisert werden, wobei je nach Wahl der Integrationskonstanten sehr verschiedene Ergebnisse entstehen können.

In älteren Computeralgebrasystemen wie REDUCE war es üblich, alles auszumultiplizieren; in den heute gebräuchlichen Systemen wie MAPLE und MATHEMATICA werden Umformungen nur noch durchgeführt, wenn es entweder für die jeweilige Rechnung notwendig ist (Zur Berechnung der Stammfunktion eines Polynoms muß dieses in ausmultiplizierter Form vorliegen) oder wenn es der Anwender explizit verlangt. Lediglich in einigen offensichtlichen Fällen bemühen sich auch diese Systeme um Normalisierung: Beispielsweise werden Brüche stets in gekürzter Form dargestellt und bei Summen werden gleichartige Terme zusammengefaßt.

Das theoretische Argument gegen kanonische Darstellungen ist, daß es solche Darstellungen nur für sehr eingeschränkte Klassen von Zahlen und Funktionen gibt: Wie wir gleich sehen werden, ist selbst für reelle Zahlen im allgemeinen unentscheidbar, wann zwei auf unterschiedliche Weise dargestellte Zahlen gleich sind.

## §4: Das zehnte Hilbertsche Problem und der Satz von Richardson

Auf dem Internationalen Mathematikerkongreß 1900 stellt DAVID HILBERT 23 Probleme vor, von denen er glaubte, daß sie für die Mathematik



des 20. Jahrhunderts wichtig sein sollten. Die Probleme kamen aus allen Teilgebieten der Mathematik und hatten auch sehr unterschiedlichen Schwierigkeitsgrad: Einige wurden schon sehr bald gelößt, andere sind auch ein Jahrhundert später noch ungelöst. Das zehnte Problem lautete:

## §5: Wichtige Datenstrukturen der Computeralgebra

## §6: Gängige Computeralgebrasysteme