MC-Computermagazin, Heft 11, 42-59 (1989)

Mathematica
Mathematik auf Mikrocomputern

Markus van Almsick, Klaus Schulten

1. August 1989

1 Einleitung

Viele Kinder wünschen sich einen Zauberkasten, weil sie davon überzeugt sind mit den Utensilien einer solchen Trickkiste wirklich zaubern zu können. Völlig analog verhält es sich bei Schülern und Studenten, die sich einen Computer kaufen, um ihn als Hilfsmittel beim Lösen mathematischer und naturwissenschaftlicher Aufgaben einzusetzen. Leider versagt der Computer-Zauberkasten, sobald in einem Problem symbolische Umformungen gefragt sind, und das ist dummerweise in fast allen allgemeineren Rechnungen, wie Differenzieren, Integrieren und Lösen algebraischer Gleichungen, der Fall. Der Hokuspokus beschränkt sich somit auf Desktop-Publishing, numerische Modellrechnungen und Datenverwaltung (Computerspiele natürlich auch!). Dies sind zwar interessante und wichtige Einsatzgebiete, dennoch wäre für viele eine Art Taschenrechner für algebraische und analytische Rechnungen eine willkommene Ergänzung.

2 Mathematica

Dieser Artikel beschreibt einen derartigen Taschenrechner: *Mathematica* von Wolfram Research Inc. Das *Mathematica* Software-Paket ist seit Juni letzten Jahres in den USA als „System for Doing Mathematics" auf dem Markt. Es ist nicht nur eine gelungene regelbasierte Programmiersprache für symbolische Manipulationen (SM); auch äußerst anspruchsvollen numerischen und grafischen Anforderungen ist es gewachsen. Diese zwei wichtigen Aspekte werden allerdings in diesem Artikel nicht näher in Augenschein genommen, obwohl gerade der Graphikteil von vielen *Mathematica*-Benutzern am häufigsten eingesetzt wird (siehe Abb.[Kugelfkt]). Der Schwerpunkt unseres Artikels soll auf der in der Mikrocomputerwelt noch wenig verbreiteten SM-Programmiersprache liegen. SM-Programmiersprachen sind bereits seit ungefähr 20 Jahren auf Großrechenanlagen zu finden. In die Welt der Mikrocomputer sind derartige Programmiersprachen auf Grund ihrer Komplexität und ihres großen Speicherbedarfs bisher nicht eingedrungen. Die Kapazitäten der kleinen Rechner wachsen jedoch ständig. Daher ist es mittlerweile möglich und auch sinnvoll, auf Mikrocomputern SM-Programme und -Programmiersprachen zu implementieren. Beispiele hierfür sind *Derive, Maple* und *Reduce* [?].

Mathematica selber ist zwar eine Neuentwicklung, aber kein Einzelkind. Der geistige Vater, Stephen Wolfram, war bereits in den Jahren 80-81 Leiter des SMP-Computeralgebra-Projekts am California Institute of Technology. Von dort ist Stephen Wolfram, der vor allem durch seine Theorie der zellulären Automaten in wissenschaftlichen Kreisen berühmt wurde, über Princeton an die University of Illinois

in Urbana-Champaign gekommen. Hier ist er seitdem eigentlich mehr nebenberuflich Direktor des Center for Complex System Research. Die meiste Zeit der letzten drei Jahre hat er in den Aufbau seiner Firma Wolfram Research Inc. gesteckt. Die Entwicklung dieser Firma liest sich wie eine typische amerikanische Erfolgs-Story. Die Gruppe aus hochkarätigen Wissenschaftlern und Programmierern wächst exponentiel und dürfte beim Erscheinen dieses Artikels sicherlich über achtzig Mitarbeiter betragen. Dabei stammt nur ein geringer Anteil dieser Belegschaft aus dem mittleren Westen der USA. Rekrutiert werden die Mitarbeiter aus allen Herren Ländern; von Pakistan bis Australien ist alles vertreten. Das Gros dieser internationalen Gruppe ist Mitte bis Ende zwanzig. Wie sollte es auch anders sein? Der Chef selber, Stephen Wolfram, ist erst gerade dreißig. Es ist daher nicht verwunderlich, wenn dieses junge Unternehmen bei einem Außenstehenden einen unorthodoxen Eindruck hinterläßt. Stephen Wolfram ist diesem Image auch nicht abgeneigt. Ihm selber ist die Aura eines legèren Wissenschaftlers zum Ärgernis einiger marketing-bewußten Mitarbeiter lieber. Dem Produkt ist alledem nichts anzumerken. Es handelt sich um ein sehr solides Software-Paket.

Mathematica läuft auf einer ganzen Palette von Rechnern; bei einer Cray angefangen, bis hinunter in die Niederungen eines Mac Plus (mit 2.5 MByte-Speicher) oder eines 386-er MS-DOS PC's (mit 1.64 MByte-Speicher). Diese untere Hardwaregrenze ist jedoch etwas hypotetischer Natur. Fairerweise sollte man darauf hinweisen, daß *Mathematica* zur Entfaltung aller seiner Fähigkeiten 4-5 MByte in Anspruch nimmt. Diese für Mikrocomputer hohen Anforderungen lassen sich auf einigen Betriebssystemen durch swapping umgehen. *Mathematica* ermöglicht es auch, den eigentlichen Programmkern auf einer größeren Maschine zu implementieren und den Personal-Computer lediglich als *Mathematica*-Terminal zu benutzen. So läuft etwa in unserer Arbeitsgruppe der *Mathematica*-Kernel auf NeXT und Silicon Graphics Rechnern und die Benutzeroberfläche auf Mac II's. Hardwaremäßig verbunden sind diese Computer durch Ethernet. Softwäremäßig wird der Datentransfer mit TCP/IP gesteuert.

Alle Hardwarevoraussetzungen für *Mathematica*-fähige Mikrocomputer sind in Tabelle [Hardware] zusammengefaßt.

Am Beispiel von *Mathematica* sollen nun im Folgenden die Konzepte und Strategien der regelbasierten SM-Programmierung skizziert werden.

3 Strategien in der Mathematik

Mathematik und Informatik sind in vieler Hinsicht verwandte Disziplinen. Weshalb also verhalten sich Computer, die Sprößlinge der Informatiker, so störrisch, wenn es darum geht, symbolische Umformungen an mathematischen Termen vorzunehmen? Es gibt dafür zwei Gründe, die an einem einfachen Beispiel erläutert werden können.

Angenommen man füttert seinen Rechner mit der Gleichung

$$x^2 + 3\,a\,x = 9.$$

Für jemanden, der quadratische bzw. binomische Gleichungen kennt, ist der Lösungsweg evident: entweder man erweitert beide Seiten mit $9a^2$, zieht die Wurzel und löst die Gleichung nach x auf, oder man greift sich eine Formelsammlung und setzt die Koeffizienten der Variable x und den konstanten Term in die Formel für quadratische Gleichungen ein. Der erste vollkommen unscheinbare Schritt in diesem Arbeitsgang ist für einen Computer bereits der schwerste. Wie identifiziert ein

Rechner den Typ eines mathematischen Terms? Eine quadratische Gleichung, z.B.

$$e^x ln a + e^{2x} = ln^2(\frac{1}{\sqrt{a}}),$$

ist nicht immer leicht als solche zu erkennen. Derartige Probleme fallen unter die Rubrik Mustererkennung. Hat ein Rechner den Typ einer Gleichung erkannt und ist der Lösungsweg für diese Gleichung vorgegeben, ist der Rest Fließbandarbeit, wie geschaffen für einen Computer.

Ein zweites Problem beim Lösen mathematischer Terme ergibt sich, wenn der Rechner eine gefragte Lösungsstrategie nicht kennt. Da hilft nur eins: Probieren, auf Teufel komm 'raus! Diese „Strategie" hat einen großen Nachteil. Sie kann extrem oder auch unendlich lange dauern. Meist gibt es eine ganze Reihe von Umformungsmöglichkeiten und jede Umformung ergibt einen neuen Term, auf den wiederum neue mathematische Regeln angewendet werden können. Man muß folglich wie beim Schachspiel eine sich exponentiell verästelnde Baumstruktur von Lösungswegen durchsuchen, eine Aufgabe, die die meisten Hardwarekapazitäten überfordert. Mathematiker können in dieser Situation gegebenenfalls anschaulich argumentieren oder ihren Instinkt zu Rate ziehen. Diese Möglichkeit bietet sich dem Rechner (noch?) nicht.

Nur an einigen Teilabschnitten der mathematischen Forschungsfront kommt die extensive Probierstrategie per Computer zum Einsatz. Zum Beispiel können die vielen Fallunterscheidungen in den Herleitungen von Sylogismen[1] oder die verschiedenen Graphentypen im Beweis des Vier-Farben-Theorems[2] mit einem Computer durchgespielt werden. Vor allem die Programmiersprache PROLOG kommt hier zur Anwendung.

Die meisten alltäglichen anwendungsorientierten mathematischen Probleme lassen sich jedoch ohne Probierstrategie, lösen. Vor allem für dieses Einsatzgebiet ist *Mathematica* geeignet.

Um die Fähigkeiten von *Mathematica* besser erläutern zu können, wird in den folgenden Abschnitten der Aufbau und die Arbeitsweise von SM-Programmen im allgemeinen und von *Mathematica* im besonderen beschrieben. Wie in fast jeder Programmbeschreibung beginnen wir auch hier mit einer Einführung über Datentypen und Strukturen.

4 Datenstruktur

Die Mathematik umfaßt eine riesige Anzahl von Objekten, z.B. Funktionen, Variablen, Konstanten, Matrizen, Graphen, Polynome, Polyeder und Knoten. Diese Objekte müssen alle unter einen Hut gebracht werden. *Mathematica* arbeitet daher mit den in der KI-Forschung sehr verbreiteten Listen als Datenstruktur.

Listen werden folgendermaßen gehandhabt. Die „nackten" Daten, wie z.B. der binäre Code einer reellen Zahl oder die ASCII-Zeichen eines Textes, bezeichnet man als Atome. Mit dieser Art von Daten kann ein Rechner allerdings nichts anfangen. Informationen über den Datentyp und dessen Verarbeitungsweise müssen mit den „nackten" Daten gekoppelt werden. In herkömmlichen Programmiersprachen geschieht dies durch Typen- und Variablen-Definition. Um Atome in die richtige Syntax einzubinden, ver-

[1] Sylogismen sind Theoreme der Logik, d.h. Aussagen, die auf Grund eines logischen Axiomensystems (z.B. Boolsche oder Aristotelische Logik) immer wahr sind. Ein simples Beispiel in der boolschen Logik ist die Aussage $a \wedge \neg a$.

[2] Das Vier-Farben-Theorem besagt: Gegeben sei eine beliebig komplizierte Landkarte auf einer Kugel- oder topologisch äquivalenten Fläche. Es ist dann möglich diese Landkarte mit nur vier Farben so einzufärben, daß Gebiete mit gemeinsamer Grenze jeweils mit unterschiedlichen Farben belegt sind.

wendet *Mathematica* Listen. Listen sind, wie der Name sagt, geordnete Mengen, die Atome oder auch weitere Listen als Elemente enthalten. Nun ist das erste Element einer Liste, der sogenannte Listenkopf, für den Computer eine Art Aushängeschild, das die Bedeutung und Verarbeitungsweise der folgenden Listenelemente bestimmt. Durch diesen Kunstgriff lassen sich mit den jeweiligen Listenköpfen alle gewünschten Datentypen umschreiben.

Beispiele können die Anwendung und Flexibilität dieser Darstellungsweise am besten verdeutlichen.

Notation :
$< Listenkopf > [< 2.elem >, ...]$

$$Real[345.6] \longrightarrow 345.6$$

$$Plus[a, b] \longrightarrow a + b$$

$$Sin[Power[x, -1]] \longrightarrow \sin \frac{1}{x}$$

$$Expand[Power[Plus[x, a], 3]]$$
$$\longrightarrow \quad (x + a)^3 \mapsto x^3 + 3ax^2 + 3a^2x + a^3$$

Drei sehr unterschiedliche Objekte werden durch die obigen Listen dargestellt. Der Listenkopf *Real* identifiziert einen Datentyp, *Plus, Power* und *Sin* fungieren als Operator bzw. Funktion und *Expand* agiert als *Mathematica*-SM-Kommando. Im Endeffekt lassen sich jedoch alle Listenköpfe schlicht als Programmierbefehle interpretieren. Auf diesen Aspekt der syntaktischen Gleichstellung von Daten, Datentypen und Operationen kommen wir weiter unten noch zu sprechen.

Durch Schachtelung von Listen können beliebig komplizierte Strukturen gebildet werden. Eine 2×2-Matrix läßt sich zum Beispiel durch drei verschachtelte Listen beschreiben.

$$List[List[a, b], List[c, d]]$$
$$\longrightarrow \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

Die Datenstrukturierung durch Listen wird in Mathematica ohne Ausnahme beibehalten. Sogar Daten für Grafiken werden in dieser Form gehandhabt. Da die Listen-Notation aber sehr unhandlich ist, wird in den folgenden Abschnitten eine auch in *Mathamtica* verfügbare konventionelle Notation verwendet.

5 Listen als Programmcode

Wie oben bereits erwähnt, können nicht nur Daten, sondern auch Programmcodes in eine Listenstruktur eingebunden werden. Der Kopf einer Liste wird unter diesen Umständen von *Mathematica* als Kommando interpretiert. Mit *Mathematica*-Kommandos der Art *Block, If, Do, Goto* etc. lassen sich so wie in herkömmlichen Programmiersprachen mehrere Befehle in einer vorgegebenen Reihenfolge als Programm oder Prozedur abarbeiten. Eine Gegenüberstellung einer Pascal- und einer *Mathematica*-Prozedur ist in Abb.[Programmierstil] zu finden. Diese prozedurale Programmiertechnik spielt in *Mathematica* allerdings nur eine untergeordnete Rolle. *Mathematica* ist in erster Linie eine regelbasierte Programmiersprache. Bevor jedoch der mit diesem Sprachentyp einhergehende Programmierstil im übernächsten Abschnitt an Hand von Beispielen erläutert werden kann, müssen noch einige grundlegende *Mathematica*-Eigenschaften erwähnt werden.

Wenden wir uns den Bausteinen von *Mathematica*-Programmen, den Befehlen zu. Die elementaren *Mathematica* Befehle, wie z.B. *Sqrt, Expand, Replace, Plot3D* etc., sind von vornherein definiert

und durch in C geschriebene Routinen im Computer verankert. Komplexere Befehle lassen sich aus diesen elementaren Befehlen zusammensetzen. Besonders die große Anzahl von *Mathematica* Befehlen, es sind insgesamt über 600, ermöglicht es, jede gewünschte Operation mit geringem Aufwand zu programmieren. *Mathematica* Befehle lassen sich im wesentlichen in fünf Klassen einordnen:

- Listenköpfe, die lediglich den Typus der folgenden Listenelemente bestimmen: z.B. *Byte, Complex, Polygon*, ...

- Listenköpfe, die Operationen darstellen: Die hinter einem Operator stehenden Listenelemente werden als Parameter oder Variable interpretiert. Numerisch verarbeitbare Symbole, wie *Integer[4]* oder *Infinity[]*, werden von numerischen Prozeduren automatisch verrechnet (z.B. $Power[Infinity[], -1]$ ergibt $Integer[0]$). Beispiele für nicht numerische Operationen sind Prädikatsfunktionen

  $$MemberQ[\{a, b, s, t, x\}, x] \mapsto True,$$

  Umformungen, wie das Differenzieren von Funktionen,

  $$D[x^2] \mapsto 2\,x$$

  und Befehle, wie das Transponieren von Matrizen

  $$Transpose[List[List[a, b], List[c, d]]] \longrightarrow List[List[a, c], List[b, d]].$$

  Falls die Variablen bzw. Parameter keine Verarbeitung zulassen, wird eine Liste unverändert beibehalten; z.B. $Plus[1, a]$ ergibt $Plus[1, a]$.

- Listenköpfe, die Programmabläufe, Programmstrukturierung und Datenstrukturierung ermöglichen: z.B.

  $$If[x >= 0, delta := 1, delta := 0],$$

  $$Do[s := s + i, i, 1, 10].$$

  $$Block[\{local\_var\}, expr1, expr2, ...]$$

- Listenköpfe, die Input/Output-Funktionen triggern: Jedes Ergebnis wird normalerweise von Mathematica unmittelbar in einem Defaultmodus ausgegeben. Um spezielle Ausgabemodi, wie zum Beispiel Graphik, einzuschalten, werden die Ausgabedaten in einer Liste mit dem entsprechenden Ausgabekommando als Listenkopf eingebettet. Es gibt außerdem eine Reihe von Prozeduren, die das Ein- und Auslesen von Daten in Verbindung mit anderen Programmen ermöglichen. Mit diesen Prozeduren ist es etwa möglich, das Resultat einer symbolischen Umformung als C-Befehle, Fortran-Befehle oder als TeX-Source-Text auszugeben, z.B:

  $Integrate[Log[x\hat{}x], x]$
  $\longrightarrow -x\hat{}2/4 + (x\hat{}2 * Log[x])/2$
  $CForm[\%]$
  $\longrightarrow -Power(x, 2)/4$
  $+Power(x, 2) * Log(x)/2$
  $FortranForm[\%]$
  $\longrightarrow -(x**2)/4$
  $+x**2 * Log(x)/2$
  $TeXForm[\%]$
  $\longrightarrow \{\{-\{x\hat{}2\}\} \setminus over4\}$
  $+\{\{\{x\hat{}2\}\log(x)\} \setminus over2\}$

  Diese Einrichtung ist vorallem als Hilfestellung zur Programmentwicklung gedacht.

- Listenköpfe, die Muster zur Mustererkennung definieren: Die Bedeutung und Anwendung derartiger *Mathematica*-Befehle wird im folgenden Abschnitt beschrieben.

6 Mustererkennung

Im mathematischen Alltag sind zwei Arten von Fähigkeiten gefragt. Zum einen sollte man ein gewisses Maß an mathematischen Grundkenntnissen mitbringen und zum anderen sollte man in der Lage sein, sein Wissen auf Probleme anwenden zu können. Das Erlernen von Wissen stellt für Mensch und Maschine keine prinzipielle Schwierigkeit dar. Für den Menschen ist dies eine Fleiß- und für die Maschine eine Speicherplatz-Frage. Ein vollkommen andere Herausforderung ist es, das erlernte Wissen umzusetzen. Mit diesem Prozeß haben nicht nur Computer Probleme. Es gibt eine ganze Menge leidgeprüfter Mathematikschüler, die sich durch Analogieschlüsse von einer Aufgabe zur nächsten hangeln. Ursache für diese Taktik sind Schwierigkeiten, die auftreten, wenn man die einzelnen Objekte in einer Aufgabe mit den entsprechenden Größen in der Theorie zu identifizieren hat. Dieser Arbeitsschritt soll hier mit dem Begriff Mustererkennung belegt werden. Mustererkennung ist unumgänglich, wenn es darum geht, theoretisches Wissen sinnvoll umzusetzen. Wie also erfolgt Mustererkennung in mathematischen Problemen ?

Die Musterumschreibung in *Mathematica* basiert, wie auch in textverarbeitenden Programmen, Datenbanken und Betriebssystemen, auf *wild cards*[3]. Jede *wild card* wird in *Mathematica* durch Unterstrich „_" gekennzeichnet. Der Ausdruck „a_" zum Beispiel paßt als *wild card* auf jede x-beliebige Listenstruktur, jede Liste oder auch jedes Atom. Auf die mit „a_" eingefangene Datenstruktur kann man dann später über den *wild card* Namen „a" zugreifen. Das Zeichen „a" ist sozusagen der Variablenname für alle die Symbole, die durch die *wild card* „a_" aufgenommenen werden können.

Mit Hilfe von *wild cards* werden zum Beispiel Funktionen in *Mathematica* folgendermaßen definiert.

$$f[x\_] := Sin[x]$$

Im Zuge dieser Anweisung wird in allen später folgenden Eingaben eine Listenstruktur der Form $f[<$ *irgendetwas* $>]$ durch die Listenstruktur $Sin[<$ *irgendetwas* $>]$ ersetzt.

Mit Listen lassen sich beliebig komplizierte *wild card*-Strukturen aufbauen, z.B.

$$log[Power[b\_, n\_]] := n\ log[b].$$

Diese Logarithmusregel würde folgende Umformungen verursachen.

$$log[\pi^3] \Longleftrightarrow 3\ log[\pi]$$

Es gibt auch *wild cards* mit kontextabhängigem Default-Wert. Kontextabhängig bedeutet in diesem Zusammenhang, daß leer ausgegangene *wild cards* ein von den Umständen abhängiges neutrales Symbol zugeordnet bekommen. So erfaßt zum Beispiel die Differentiationsregel

$$\frac{d}{dx}x^n = nx^{n-1}$$

in der *Mathematica*-Notation mit der *wild card* n_.

$$D[x\_^{n\_}, x\_] := n\,x^{n-1}$$

auch den Fall

$$D[x, x] \mapsto x^{1-1} \mapsto 1\ .$$

Hier wird der fehlende Exponent n per Default gleich 1 gesetzt. Dies ist zulässig, da 1 das neutrale Element einer jeden Zahlenmenge im Exponenten ist. Im Falle einer Addition wäre für n nicht 1 sondern der Defaultwert 0, das neutrale Element der Addition, eingesetzt worden.

Es gibt noch weitere kontextabhängige Mustereigenschaften. Folgende Muster

[3] Eine *wild card* ist eine Art Jocker-Symbol

sind zum Beispiel wegen der Kommutativität der Addition und wegen der Assoziativität der Multiplikation identisch:

$$Plus[x\_, 1] \Longleftrightarrow Plus[1, x\_],$$

$$Times[x\_, Times[y\_, z\_]] \Longleftrightarrow Times[x\_, y\_, z\_].$$

Derartige kontextbedingte Umformungsmöglichkeiten werden von *Mathematica* automatisch bei der Mustererkennung mit berücksichtigt.

Ohne an dieser Stelle weiter die Möglichkeiten der *wild card* Programmierung auszuloten, läßt sich der Kürze halber folgendes zusammenfassend sagen: *wild cards* sind Schablonen, mit denen man in einem mathematischen Term passende Merkmale aufzufinden versucht. Mit Schablonen lassen sich aber nur syntaktische, d.h. formale Merkmale überprüfen. Damit allein ist es in der Mathematik nicht getan. Die gefundenen – d.h. die in die Schablone passenden Terme – müssen oft noch Eigenschaften erfüllen, die nicht von vornherein mit *wild cards* spezifiziert werden können. Solche semantischen oder auch inhaltlichen Kriterien können als Nebenbedingung dem Rechner bei der Mustersuche mit auf den Weg gegeben werden. Diese Nebenbedingungen werden in *Mathematica* an eine Prozedur oder Regel angehängt und durch „/;“ gekennzeichnet. Die obige Differentiationsregel müßte somit genaugenommen

$$D[x\_^{n\_}, x\_] := n\, x^{n-1} \;\; /; FreeQ[n, x]$$

lauten, denn n sollte x-unabhängig sein. Das bedeutet im *Mathematica*-Kontext, daß in der Listenstruktur von n keine x-Atome und auch keine x-Listen eingebettet seien dürfen. Die boolsche Funktion *FreeQ* überprüft diesen Sachverhalt.

In einigen Fällen der Mustererkennung ist es ratsam vor der Suche mit *wild card*-Schablonen einige Umformungen am gefragten Term vorzunehmen. Liegt ein mathematischer Term in standartisierter Form vor, sind die charakteristischen Merkmale oft leichter mit *wild cards* zu definieren und ausfindig zu machen.

Das eingangs erwähnte Beispiel, das Lösen quadratischer Gleichungen, erfordert alle hier besprochenen Mustererkennungsverfahren. Ein zum Lösen quadratischer Gleichungen geeignetes *Mathematica*-Regelkontingent ist in Abb.[quadrat.Gl.] aufgeführt. Dieses nicht gerade einfache Beispiel ist allerdings erst nach Lesen des folgenden Abschnittes in allen Einzelheiten verständlich. Es besteht aus zwei Teilen. Im ersten Teil, der Implementation des Lösungsverfahrens, wird durch fünf Umformungsregeln ein *Solve*-Kommando zum Lösen quadratischer Gleichungen definiert. Mit diesem *Solve*-Kommando wird dann exemplarisch im zweiten Teil eine binomische Gleichung gelöst.

7 Regelbasiertes Programmieren

Konventionelle Programmiersprachen wie Basic, Fortran, Pascal, C usw. beruhen alle auf ein und derselben Programmiertechnik. Sie unterstützen den Programmierer beim Erstellen einer Befehlskette, die der Computer dann vom Anfang bis zum Ende der Kette abarbeitet. Der Ablauf eines so erstellten Programms wird vom Programmierer bestimmt (Spezialisten für Endlosschleifen werden hier nicht in Betracht gezogen). Diesen altbekannten Programmierstil bezeichnet man als prozedurales Programmieren.

Völlig anders verhält es sich in regelbasierten Programmiersprachen. Anstatt einer Befehlskette wird eine Reihe an Regeln erstellt. Diese Regeln werden im Gegensatz zu Befehlen nicht hintereinander abgearbeitet, sondern je nach Problem und Anwendbarkeit vom Computer selber ein-

gesetzt. Die Strategie, mit welcher die abgespeicherten Regeln angewandt werden, ist von der jeweiligen regelbasierten Programmiersprache abhängig und kann vom Programmierer meist nur in geringem Maße beeinflußt werden. Der Reiz und auch die Gefahr in regelbasierten Programmen besteht in der Unvorhersehbarkeit des Programmablaufes. Man ist oft erstaunt, wie effektiv und „kreativ" ein Rechner die eingegebenen Regeln umsetzt und Lösungswege beschreitet, die man selber nie in Betracht gezogen hätte.

Regelbasiertes Programmieren ist in gewisser Hinsicht „menschlicher" als prozedurales Programmieren. Das soll nicht heißen, daß beim Erstellen eines regelbasierten Programms ein geringeres Maß an Pedanterie als sonst erforderlich wäre. Es ist vielmehr so, daß ein Großteil des menschlichen Wissens in Form von Regeln und nicht in Form von Prozeduren vorliegt. Daher ist es wesentlich einfacher, Wissen, insbesondere mathematisches Wissen, mit regelbasierten Programmiersprachen auf Computern zu implementieren.

Die Philosophie des regelbasierten Programmierens ist in *Mathematica* folgendermaßen verwirklicht. Eine Regel besteht immer aus zwei Teilen; einem Konditional-Teil, der die Anwendbarkeit einer Regel festlegt, und einem Konsekutiv-Teil, der die Instruktionen einer Regel beinhaltet. Der Konditional-Teil wird in *Mathematica* mit Mustern und Nebenbedingungen programmiert. Der Konsekutiv-Teil kann aus einer Prozedur von *Mathematica*-Befehlen oder gegebenenfalls auch aus weiteren untergeordneten Regeln bestehen. Ein typisches Beispiel für ein einfaches Programm sind folgende Differentiationsregeln für algebraische Funktionen.

$$\begin{aligned} D[c\_, x\_] &:= 0 \quad /; FreeQ[c, x] \\ D[x\_^{n\_}, x\_] &:= n\, x^{n-1} \; /; FreeQ[n, x] \\ D[f\_ + g\_, x\_] &:= D[f, x] + D[g, x] \end{aligned}$$

$$D[f\_\ g\_, x\_] := D[f, x]\ g + f\ D[g, x]$$

$$\Rightarrow D[x^3 - x/2] \mapsto 3\,x^2 - \frac{1}{2}$$

Die Strategie, mit der *Mathematica* die zur Verfügung stehenden Regeln anzuwenden versucht, ist denkbar einfach. Zuerst werden die speziellen, d.h. die von ihrem Konditional-Teil her restriktivsten Regeln ausprobiert. Sollte problemspezifisches Wissen nicht vorliegen, versucht *Mathematica* allgemeinere Regeln anzuwenden. Spezielle Regeln enthalten theoretisch mehr und meist auch passendere Information. Die Taktik, spezielle Regeln allgemeinen vorzuziehen, ist daher als heuristisches Kriterium sehr effizient. Die oben aufgeführten Differentiationsregeln sind nach diesem Gesichtspunkt aufgereiht.

Wie weiter oben bereits erwähnt wurde, verfolgt *Mathematica* nur einen Lösungsweg. Das heißt, daß immer nur die jeweils erste passende Regel angewendet wird, und daß nach jeder Umformung die Liste der Regeln auf's Neue von Anfang an durchprobiert wird. Sobald keine Regel mehr anwendbar ist, wird der Umformungsprozeß abgebrochen und das Ergebnis ausgegeben.

8 Objektorientiertes Programmieren

Seit kurzer Zeit gibt es in der Software-Gemeinde ein neues Credo, objektorientiertes Programmieren. Worum geht's? Jede Art der Datenverarbeitung ist ein Wechselspiel zwischen Operationen und Daten bzw. Datentypen. Bisher war es üblich, zwischen diesen beiden Domänen streng zu unterscheiden. Datentypen wurden unabhängig von den mit ihnen assoziierten Operationen definiert. Dieses Konzept ist unflexibel und verhängnisvoll, wenn es darum geht, Veränderungen vorzunehmen.

Ein Beispiel: Angenommen, man hat eine Prozedur zur Multiplikation reeller Zahlen geschrieben. Will man diese Prozedur auf die komplexen Zahlen erweitern, muß man sie weitgehend umschreiben. Das gleiche Drama würde sich für modulare Zahlen[4], Matrizen, usw. wiederholen.

Dies ist nur ein simples Beispiel. Problematisch ist die herkömmlichen Programmiertechnik vor allem in großen und komplexen Programmen. Jede Änderung an einem System mit vielen Interdependenzen ist riskant und oft fatal.

Ein Heilmittel, modulares Programmieren, beschränkt sich leider nur auf die Domäne der Operationen. Entsprechende modulare Programmiertechniken für Datenstrukturen gibt es nicht. Um dieses Manko zu umgehen, muß die Trennung zwischen Datentypen und Operationen aufgehoben werden. Die Devise des objektorientierten Programmierens ist es daher, jedem Datentyp die Information über seine Verarbeitung direkt mit auf den Weg zu geben. Das heißt, daß z.B. jeder Zahlentyp seine Multiplikationregeln mit sich herumträgt und diese nicht einer Multiplikationsprozedur überläßt, die sich ohnehin schon mit dutzenden anderen Zahlentypen abgeben muß. Objektorientiertes Programmieren ist ein durchaus natürlicher Schritt in der Softwareentwicklung, denn das Wesen eines Objektes ist nicht alleine durch seine Struktur, sondern auch durch seine Eigenschaften in den mit ihm assoziierten Operationen bestimmt. Für einen Mathematikstudenten wäre es z.B. ein 'faux pas' in der Definition des Begriffs „Vektor" die Axiome des Vektorraumes, die die Eigenschaften der Vektoroperationen festlegen, nicht zu nennen. So ist ein Zahlentupel von Natur aus noch lange kein Vektor. Erst die auf ein solches Zahlentupel anwendbaren Axiome eines Vektorraumes zeichnet es als Vektor aus.

Zurück zum regelbasierten Programmieren. Objektorientiert bedeutet in diesem Fall, die mit einem Objekt assoziierte Umformungsregeln nicht unter dem Namen der Regel, sondern unter dem Objektnamen einzuordnen.

Ein Beispiel: Wer würde auf die Idee kommen, die trigonometrische Identität $\sin^2 x + \cos^2 x = 1$ als Additionsregel zu bezeichnen? Diese Gleichung ist in Mathematikbüchern selbstverständlich unter der Rubrik „Trigonometrische Funktionen" zu finden.

Um in regelbasierten Programmiersprachen keinen Regelwust aufkommen zu lassen, ist es sinnvoll, die Umformungsregeln objektorientiert zu gliedern. Dieses Konzept ermöglicht es, entsprechende Regeln schnell aufzufinden und ganze Regelkontingente je nach Bedarf ein- und auszuladen. Die letztere Möglichkeit ist vor allem aus Gründen des Speicherbedarfs besonders wichtig. Wirtschaftswissenschaftliche Probleme bedürfen zum Beispiel keiner Theoreme projektiver Geometrien.

Mathematica ermöglicht eine objektorientierte Regelgliederung, indem man die Regeln mit den entsprechenden Objektnamen signiert.

$< Objektname > / :$
$< Muster > \ := < Instruktion >$
$/; < Nebenbedingung >$

Beispiel:

$$Exp\ /:\ \ Exp[a\_]\ Exp[b\_] := Exp[a+b]$$

$$Exp\ /:\ \ Sqrt[Exp[a\_]] := Exp[a/2]$$

In der *Mathematica*-Grundausstattung sind bereits über 60 Regelpakete enthalten. Dieser Grundstock wird sowohl von Wolfram Research als auch von Benutzern ständig verbessert und erweitert. Beispiele

[4] Die natürliche Zahlen modulo 3 bilden z.B. einen modularen Zahlenkörper. 2+2 wäre in diesem Fall nicht gleich 4 sondern gleich 1.

für bestehende Regelkontingente sind Integration, Statistik, Zahlentheorie, Vektoranalysis, und Laplace Transformation. Die *Mathematica*-Gemeinde hofft, daß in Zukunft Experten aus allen Wissensgebieten Beiträge zu einem gemeinsamen Fundus beisteuern werden. Dann würden auch Regelpakete für exotischere Anwendungen, wie etwa Gruppentheorie oder Bundesligastatistik, zu Verfügung stehen.

9 Anwendung

Mathematica ist eine vollwertige Programmiersprache. Diese Bezeichnung wäre jedoch eine Untertreibung und in gewissem Sinne auch mißverständlich. *Mathematica* ist viel mehr ein Zwitterwesen aus Programmiersprache, Expertensystem, Mathematiklabor und Grafikprogramm. Der *Mathematica*-Interpreter ermöglicht es dem Anwender, mit Hilfe eingeladener Regel-Pakete, Aufgaben im Dialog mit dem Computer zu lösen. Jede Eingabe in den Rechner wird durch den jeweiligen Listenkopf als Daten-Eingabe, Regel-Definition[5] oder als *Mathematica*-Kommando interpretiert und sofort ausgeführt.

Um für fremde Anwender die Funktionsweise eines Regel-Paketes gut dokumentieren zu können, bietet *Mathematica* die Möglichkeit, Regelsysteme und Anwendungsbeispiele auf der Benutzeroberfläche, dem sogenannten *Notebook*, in einen Dokumentationstext mit einzubinden. Die in einem solchen Text vorkommenden Formeln können vom Leser direkt ausprobiert und auf eigene Beispiel angewandt werden (siehe Abb.[Fourier]). Man erhält so eine „funktionstüchtige" Beschreibung des jeweiligen Wissensgebietes. Zu Demonstrationzwecken lassen sich sogar kleine Filme auf Mac II's und IBM X-Window Systemen erstellen. Einige Szenen eines solchen *Mathematica*-Films sind in Abbildung [Movie] zu sehen. Besonders im Lehrbetrieb soll diese Einrichtung Furore machen. So wird zum Beispiel an der Universität von Illinois *Mathmatica* in Analysis-Kursen eingesetzt.

Da *Mathematica* ein sehr flexibles und universelles Software-Paket ist, liegen die Anwendungsgebiete weit um die Mathematik herum verstreut. Es gibt dafür zahlreiche Beispiele: sei es nun der Physikprofessor, der seine relativistischen kosmologischen Modelle analytisch in den Griff zu bekommen versucht; sei es der Wall-Street-Aktienmakler, der seine Wirtschaftsmodelle entwickelt oder sei es der Papiermühlenbesitzer in Oregon, der mit *Mathematica* das Regelsystem seiner Papierproduktion steuert [6].

Zum Schluß noch ein etwas verspieltes, aber den Einsatz von *Mathematica* gut illustrierendes Anwendungsbeispiel, das uns in das Dickicht der gewöhnlichen Differentialgleichungen [7] führt:

Ein Förster und sein Hund durchstreifen ihr Jagdgebiet, welches durch einen Fluß schnurgerade in zwei Hälften geteilt wird. Beim Aportieren gelangt der Hund auf die andere Seite dieses Flusses. Um den Förster am diesseitigen Ufer zu erreichen, muß der Hund den Fluß wieder schwimmend durchqueren. Dabei ist er völlig auf sein Herrchen fixiert und schwimmt mit konstanter Geschwindigkeit (relativ zum strömenden Wasser) immer genau in die Richtung des Försters. Einem besorgten Hundefreund stellt sich nun die Frage, ob und wie der Hund bei konstanter Strömung das andere Ufer erreicht.

Kommen wir dem Hund zu Hilfe! Installieren wir ein Koordinatensystem, in dem

[5] Jede Funktion und jede Prozedur kann als Regel interpretiert werden.

[6] Es handelt sich hierbei um wirkliche Anwendungsbeispiele.

[7] gewöhnliche Differentialgleichungen z.B. 1.Ordnung sind Gleichungen des Typs $\frac{dy}{dx} = f(x, y)$.

der Förster im Ursprung steht, in dem der Fluß längs der y-Achse in Richtung $y = +\infty$ fließt und in welchem die Flußbreite auf eine Längeneinheit normiert ist. Der Hund befindet sich dementsprechend zu anfang am rechten Ufer irgendwo auf der Linie, die senkrecht zur x-Achse durch den Punkt (1,0) verläuft. Der Förster steht am linken Ufer im Punkt $(0,0)$, (siehe Abb. [Koordinaten]). Die Fließgeschwindigkeit des Gewässers sei im Vergleich zur Fortbewegungsgeschwindigkeit des Hundes durch den Faktor a gekennzeichnet. In dieser Konfiguration kann die Geschwindigkeit (v_x, v_y) des Hundes in jedem Punkt (x, y) wie folgt bestimmt werden. Da der Hund immer auf den Koordinatenursprung zuschwimmt, ist die Geschwindigkeit gleich dem negativen auf 1 normierten Ortsvektors des Hundes. Außerdem muß für die y-Komponente der Geschwindigkeit noch die Abdrift a in Betracht gezogen werden.

$$\begin{aligned} v_x &= \frac{-x}{\sqrt{x^2+y^2}} \\ v_y &= \frac{-y}{\sqrt{x^2+y^2}} + a \end{aligned}$$

Aus diesem Geschwindigkeitsfeld läßt sich die Trajektorie, d.h. die Route des Hundes, mit den Methoden der gewöhnliche Differentialgleichungen berechnen.

$$\frac{v_y}{v_x} = \frac{\frac{dy}{dt}}{\frac{dx}{dt}} = \frac{dy}{dx}$$

$$\Longrightarrow \frac{dy}{dx} = \frac{a - \frac{y}{\sqrt{x^2+y^2}}}{\frac{-x}{\sqrt{x^2+y^2}}} = \frac{y - a\sqrt{x^2+y^2}}{x}$$

Die Lösung dieser Differentialgleichung mit *Mathematica* ist in Abb.[Förster-Hund-Beispiel] zu sehen.

Es stellt sich heraus, daß der Hund für Fließgeschwindigkeiten $a < 1$ in endlicher Zeit das andere Ufer erreicht, während stärkere Strömungen zu einer Abdrift des Hundes führen. Abschließend läßt sich aber zur Beruhigung eines jeden Hundeliebhabers feststellen, daß nur Pekinesen mit unendlich kleiner Ausdehnung bei einer Abdrift $a \geq 1$ ertrinken. Deutsche Doggen und auch andere Rassen mit endlicher Ausdehnung erreichen früher oder später das andere Ufer.

References

[1] Stephen Wolfram: *Mathematica, A System for Doing Mathematics by Computer*,
Addison-Wesley 1988,
(Dieses Buch soll demnächst auch in deutsch erscheinen. Es handelt sich dabei um die das *Mathematica*-Software-Paket begleitende Anleitung. Da sich dieses Buch mit einigen grundlegenden Fragen der SM-Programmierung beschäftig, ist es als weiterführende Lektüre sehr zu empfehlen!)

[2] Kenneth R. Foster, Haim H. Bau: *Symbolic Manipulation Programs for the Personal Computer*,
Science Vol. 243, Software Reviews page 679, 3. Feb. 1989.

[3] Klaus Jänich: *Analysis für Physiker und Ingenieure*,
Springer-Verlag 1983.

[4] Malcolm MacCallum:
Pocket calculus,
Physics World, page 27, 2. June 1989

Table 1: Abb.[Hardware] : Microcomputer für *Mathematica*, (Juni 1989)

Macintosh Version	**Hardware :** Standard Version läuft auf Macintosh Plus, SE und II; Macintosh II Version unterstützt 68881 Coprozessor; Minimum 2.5 MB RAM, 4 MB RAM empfehlenswert !; 5 MB Harddisk-Platz; System Version 6.02 oder später.
	Kompatibilität : I/O von regulären Textfiles; Graphic : PICT, bitmap, PostScript, encapsulated PostScript; Standard Macintosh editier- und print-features.
	Preis : Standard Version : $495 Mac II Version : $795
MS-DOS 386 Version	**Hardware :** 386 Version (benötigt keinen numerischen Coprozessor); 386/7 Version (benötigt 287 oder 387 Coprozessor); 386/Weitek (benötigt Weitek 1167 oder 3167 Coprozessor); 80386 Prozessor; MS-DOS System Version 3.0 oder später; 640 kB RAM plus 1 MB extended Memory; 6 MB Harddisk-Platz; MS-DOS System Version 3.0 oder später.
	Kompatibilität : Graphic : unterstüzt CGA, EGA, VGA, MCGA, Hercules, 8514 Drucker : PostScript, LaserJet, Epson FX, Toshiba P3
	Preis : 386 Version : $695 386/7 Version : $995 386/Weitek : $1295
NeXT Version	*Mathematica* ist in der Grundausstattung eines jeden NeXT-Computers ohne Aufpreis enthalten.

Table 1: Abb.[Hardware] : Microcomputer für *Mathematica*, (Juni 1989)

Macintosh Version	**Hardware :** Standard Version läuft auf Macintosh Plus, SE und II; Macintosh II Version unterstützt 68881 Coprozessor; Minimum 2.5 MB RAM, 4 MB RAM empfehlenswert !; 5 MB Harddisk-Platz; System Version 6.02 oder später.
	Kompatibilität : I/O von regulären Textfiles; Graphic : PICT, bitmap, PostScript, encapsulated PostScript; Standard Macintosh editier- und print-features.
MS-DOS 386 Version	**Hardware :** 386 Version (benötigt keinen numerischen Coprozessor); 386/7 Version (benötigt 287 oder 387 Coprozessor); 386/Weitek (benötigt Weitek 1167 oder 3167 Coprozessor); 80386 Prozessor; MS-DOS System Version 3.0 oder später; 640 kB RAM plus 1 MB extended Memory; 6 MB Harddisk-Platz; MS-DOS System Version 3.0 oder später.
	Kompatibilität : Graphic : unterstüzt CGA, EGA, VGA, MCGA, Hercules, 8514 Drucker : PostScript, LaserJet, Epson FX, Toshiba P3
NeXT Version	*Mathematica* ist in der Grundausstattung eines jeden NeXT-Computers enthalten.

Tabelle Abb.[Hardware] ohne Preise!

(10)
(10)

Abb.[Förster-Hund]

Gewöhnliche Differentialgleichungen

Gewöhnliche Differentialgleichungen sind je nach Typ mit unterschiedlichen Methoden zu lösen. Im folgenden Beispiel werden zunächst relevante Regeln zur Lösung der in unserem "Förster-Hund-Beispiel" vorkommenden Differentialgleichungen als **DGlSolve**-Befehl definiert . Im zweiten Teil wird dann dieser neu eingeführten **DGlSolve**-Befehl angewandt.

Implementation des DGlSolve-Befehls :

Laden der Integrationsregeln:
(Es handelt sich hier um die in *Mathematica* enthaltenen Regelpakete.)

```
<<integral.m  ;
```

Definition einer Konstanten :

```
const/: NumberQ[const] = True  ;
SetAttribute[const,Constant];
```

1.Fall : y' = a(x) b(y)

Dieser Typ von Differentialgleichung wird mit der Methode der "getrennten Variablen" gelöst. Die Terme, die x bzw. y enthalten, werden getrennt und separat bzgl. x bzw. y integriert.

y' = a(x) b(y) -> Integral[dy / b(y)] = Integral[a(x) dx] + const

```
DGlSolve[y_'[x_] == a_ b_] :=

 (* Fall b = 0 : y' = 0 => y = const  *)                    (* && := "und",  || := "oder" *)
 {{ b == 0 && y == const  ||

 (* Fall b <> 0 :  Integriere  y' / b  =  a *)
    b != 0 && Integrate[1/b,y] == Integrate[a,x] + const }}    /;

 (* Nebenbedingung:  a ist nur x-abhängig und *)
 (*                  b ist nur y-abhängig.    *)
     FreeQ[a,y]  &&  FreeQ[b,x]
```

2.Fall : y' = f(x/y)

Dieser Typ von Differentialgleichung kann durch die Substitution y/x -> u in eine Differentialgleichung des 1. Falles umgewandelt werden.

y' = f(y/x) -> Substituiere u := y/x -> u' := (f(u) - u) * 1/x -> siehe 1.Fall.

```
DGlSolve[y_'[x_] == f_] := Block[{fsub,u,solution},
 (* fsub,u,solution : lokale Variable *)

 (* Subsitution:   y/x -> u  *)
  fsub        := Simplify[ReplaceAll[f,y -> u x]];

 (* Neuer Lösungsversuch mit der substituierten DGl. *)
  solution  := DGlSolve[u'[x] == (fsub - u) / x];

 (* Resubstituion: u    -> y/x *)
  Return[Simplify[ReplaceAll[solution,u -> y/x]]] ]            /;

 (* Nebenbedingung: "f(x/y) = f((r x) / (r y))": f sollte homogen sein *)
  SameQ[Simplify[f],Simplify[ReplaceAll[f,{x -> r x,y -> r y}]]]
```

Der Förster und sein Hund:

Die infinitesimale Bewegung (dx, dy) des Hundes im Punkt (x, y) lautet :
(durch a wird die Abdrift in Relation zur Fortbewegungsgeschwindigkeit des Hundes angegeben)

```
dx = - x / Sqrt[x^2 + y^2] dt     ;       (* Bewegung in Richtung Nullpunkt *)

dy = (- y / Sqrt[x^2 + y^2] + a) dt  ; (* Bewegung in Richtung Nullpunkt
                                          plus Abdrift durch Strömung a   *)
```

Hundetrajektorie als Funktion y(x) :
Die entsprechende Differentialgleichung lautet dy/dx = f(x,y) , d.h.

```
f = dy/dx
```

$$\frac{\sqrt{x^2 + y^2}\left(a - \frac{y}{\sqrt{x^2 + y^2}}\right)}{x}$$

```
f = Simplify[f]
```

$$\frac{y - a\left(x^2 + y^2\right)^{1/2}}{x}$$

Lösung der Differentialgleichung dy/dx = f(x,y) .

```
DGlSolve[y'[x] == f]       (* Löse Differentialgleichung *)
                           (* && := "und",  || := "oder" *)
```

$$\left\{\left\{-a\left(1 + \frac{y^2}{x^2}\right)^{1/2} == 0 \;\&\&\; -\frac{y}{x} == \text{const}\right\} \;\middle|\middle|\; \left\{-a\left(1 + \frac{y^2}{x^2}\right)^{1/2} \neq 0 \;\&\&\; -\frac{\text{Log}\left[-\frac{y}{x} + \left(1 + \frac{y^2}{x^2}\right)^{1/2}\right]}{a} == \text{const} + \text{Log}[x]\right\}\right\}$$

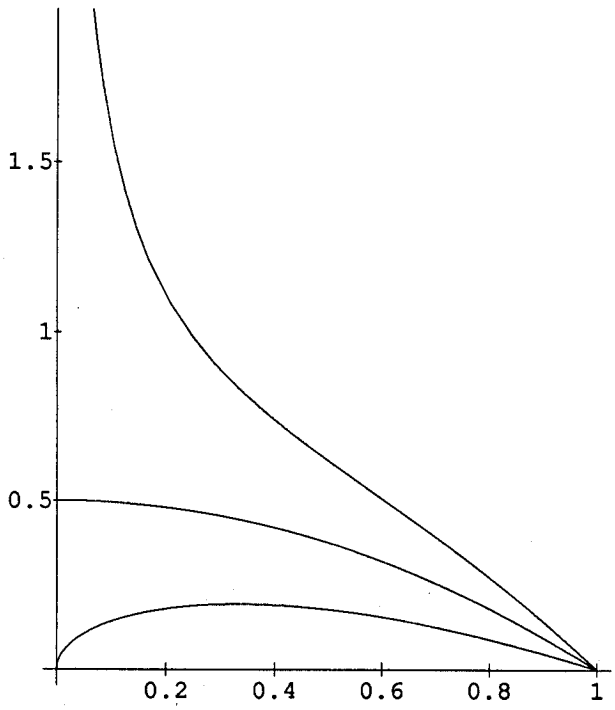
Die obige Gleichung muß per Hand nach y aufgelöst werden, denn in der *Mathematica* - Grundausrüstung sind die Fähigkeiten zur Invertierung transzendenter Funktionen bisher nur in beschränktem Maße vorhanden.
(Tip : ArcSinh[x] = Log[x + Sqrt[1 + x^2]])
Die mit x parametrisierte Trajektorie des Hundes lautet somit:

```
y[x_,a_,const_] := x Sinh[const - a Log[x]]
```

Die Konstante const legt die y-Komponente der Startposition des Hundes fest.

Trajektorien für Strömungsgeschwindigkeiten a = 0, 0.5, 1.0, 1.5 und Startposition (1,0) :

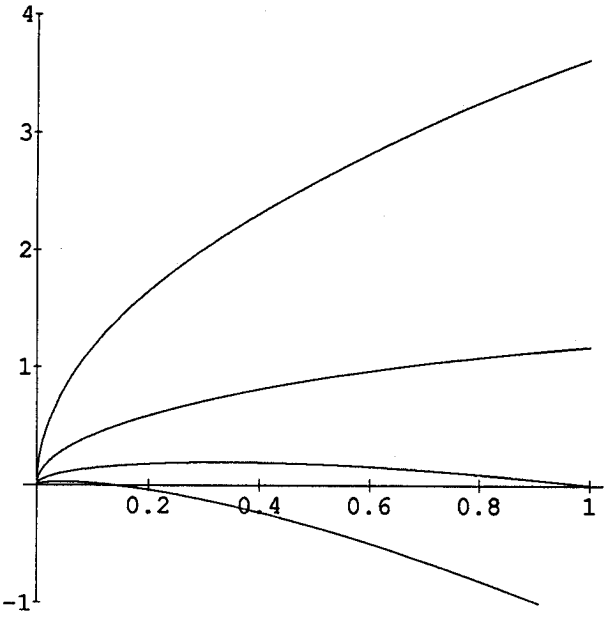
```
Plot[{y[x,0,0],y[x,0.5,0],y[x,1,0],y[x,1.5,0]},
 {x,0.000001,1}]
```



-Graphics-

Trajektorien für verschiedene Startpositionen und Strömungsgeschwindigkeit a=0.5 :

```
Plot[{y[x,0.5,-1],y[x,0.5,0],y[x,0.5,1],y[x,0.5,2]},
 {x,0.000001,1},PlotRange -> {-1,4}]
```



-Graphics-

Abb. [Fourier]

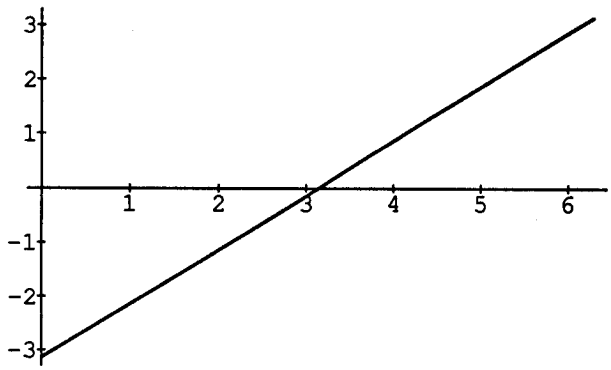
Fourier Analyse

Jede Funktion f(x) auf dem Interval [0, 2 Pi] läßt sich in Funktionen des Typs Sin[n x] und Cos[n x] zerlegen. Dies ist eine Art Spektralzerlegung. Der Anteil einer Funktion, der auf die Frequenz n, also auf die Funktion Sin[n x] und Cos[n x] entfällt, kann mit Hilfe von Fourier-Integralen berechnet werden. Die Fourier-Integrale für Sin[n x] und Cos[n x] lauten :

```
a[f_,n_] := Integrate[Sin[n x] f, {x, 0, 2Pi}] / Pi
b[f_,n_] := Integrate[Cos[n x] f, {x, 0, 2Pi}] / Pi
```

Als Beispiel soll die Funktion f[x] := x - Pi in die ersten 10 Sin- und Cos-Moden zerlegt werden.

```
f := x - Pi
Plot[f, {x, 0, 2 Pi}]
```



-Graphics-

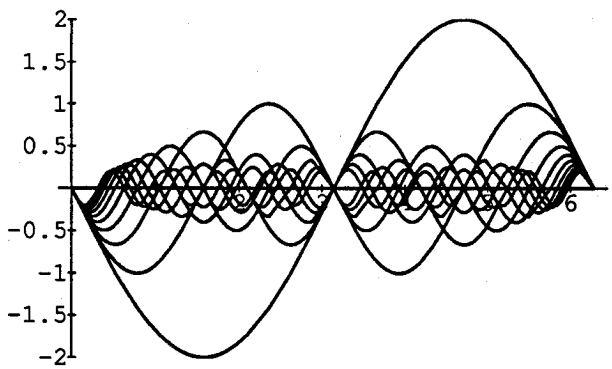
Tabelle der Fourierkoeffizienten von f[x]:

```
MatrixForm[Transpose[Table[{n, a[f,n], b[f,n]}, {n,0,10}]]]
```

0	1	2	3	4	5	6	7	8	9	10
0	-2	-1	$-\frac{2}{3}$	$-\frac{1}{2}$	$-\frac{2}{5}$	$-\frac{1}{3}$	$-\frac{2}{7}$	$-\frac{1}{4}$	$-\frac{2}{9}$	$-\frac{1}{5}$
0	0	0	0	0	0	0	0	0	0	0

Graphisch Darstellung der einzelnen Fourier-Komponenten:

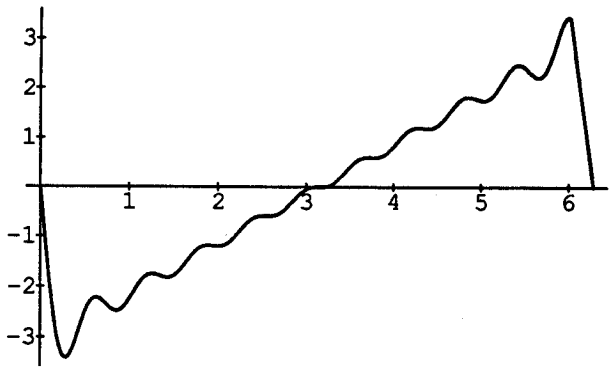
```
Plot[Release[Table[a[f,n] Sin[n x] + b[f,n] Cos[n x], {n, 0, 10}]],
     {x, 0, 2 Pi},PlotRange -> {-2,2}]
```



-Graphics-

Graphische Rekonstruktion von f[x] mit Hilfe der ersten 10 Fourierkomponeneten

```
Plot[Release[Sum[a[f,n] Sin[n x] + b[f,n] Cos[n x], {n, 0, 10}]],
     {x, 0, 2 Pi}]
```



-Graphics-

Abb.[Programmierstil]

Fakultät

Die Fakultät ist eine kombinatorische Grundfunktion.
Def.: fac(n) := n * (n-1) * (n-2) * ... * 2 * 1 , für n > 0 und
fac(0) := 1

Beispielsweise gibt es fac(50) Möglichkeiten 50 verschiedene Bücher in einem Regal anzuordnen.

▩ Pascal Funktion :

(einfacher Text, in *Mathematica* nicht lauffähig !)

```
function fac (n : integer) : integer;
  var
     i, locvar : integer;
 begin
    locvar := 1;
    for i := 1 to n do
       locvar := locvar * i;
    fac := locvar;
 end;
```

▩ *Mathematica* Funktion :

■ Prozedurale Programmierung :

```
fac[n_] := Block[{locvar = 1},
 If[Positive[n] && IntegerQ[n],
  (* True *)   For[i=1, i<=n, i++, locvar = i locvar];
     Return[locvar],
  (* False *) Return[HoldForm[fac[n]]]
 ]
]
```

■ Regelbasierte Programmierung :

```
fac[0]   := 1
fac[n_] := n * fac[n-1]      /; Positive[n] && IntegerQ[n]
                                (Nebenbed.: n ist positiv und ganzzahlig *)
```

▩ Beispiel :

```
fac[50]
```

```
30414093201713378043612608166064768844377641568960512000000000000
```

Abb.[quadrat.Gl.]

Quadratische Gleichungen

▩ Implementierung :

solve[quadratische Gleichung, Unbekannte] wird hier als *Mathematica*-Befehl zum lösen quadratischer Gleichungen definiert.
(Derartige Solve-Funktionen sind in *Mathematica* selbstverständlich Teil der Grundausstattung. Die folgende Implementierung dient nur Demonstrationszwecken.)

■ Der 1. Teil der Umformungsregeln ist dazu bestimmt, eine quadratische Gleichung in die Standardform der Art a x^2 + b x + c == 0 zu transformieren :

Schiebe alle Terme der Gleichung auf die linke Seite :

```
solve[term1_ == term2_, x_] :=
 solve[term1-term2 == 0, x]          /; Not[SameQ[term2,0]]
                                        (* Nebenbed. vermeidet Rekursion ! *)
```

Eliminiere alle Faktoren, die vor einer Summe mit einem x-abhängigen Term stehen, durch Distribution :
(z.B. a (b + x^2) -> a b + a x^2)

```
solve[rest1_. + factor_ (fx_ + rest2_) factor_, x_] :=
     solve[rest1 + factor fx + factor rest2,x]   /;   Not[FreeQ[fx,x]]
```

Fasse alle x-unabhängigen Faktoren desselben x-abhängigen Terms in einer Summe zusammen :
(z.B. a x^2 + b x^2 -> (a + b) x^2)

```
solve[rest_. + a_ fx_ + b_. fx_, x_] := solve[rest + (a+b) fx,x] /;
     FreeQ[{a,b},x] && Not[FreeQ[fx,x]]
```

■ Der 2. Teil der Umformungsregeln ist dazu bestimmt, die binomische Formel auf die in der Standardform befindlichen quadratischen Gleichungen anzuwenden :

1.Fall: quadratische Gleichungen ohne linearem Anteil : (&& := "und")

```
solve[a_. f_^(2 g_.) + c_. == 0,x_] :=
 {{f^g -> Sqrt[-c/a]},{f^g -> -Sqrt[-c/a]}}          /;

 (FreeQ[{a,c},x] && Not[FreeQ[{f},x]])
```

2.Fall: quadratische Gleichungen mit linearem Anteil : (&& := "und")

```
solve[a_. fx2_ + b_. fx_ + c_. == 0, x_] :=
 {{fx -> (-b + Sqrt[b^2 - 4c])/2 },{fx -> -(b + Sqrt[b^2 - 4c])/2}}     /;

 (FreeQ[{a,b,c},x] && Not[FreeQ[{fx2,fx},x]] &&
   SameQ[Simplify[fx2],Simplify[fx^2]])
```

▩ Beispiel :

```
solve[e^x log[a] + e^(2 x) == log[a^(-1/2)]^2, x]
```

$$\left\{\left\{e^x \to \frac{-\log[a] + \sqrt{4\,\log\left[\frac{1}{\sqrt{a}}\right]^2 + \log[a]^2}}{2}\right\},\ \left\{e^x \to \frac{-\left(\log[a] + \sqrt{4\,\log\left[\frac{1}{\sqrt{a}}\right]^2 + \log[a]^2}\right)}{2}\right\}\right\}$$

Die folgende zusätzliche Regel für die Logarithmus-Funktion vereinfacht die Lösung drastisch.

```
log /: log[b_^a_] := a log[b]
solve[e^x log[a] + e^(2 x) == log[a^(-1/2)]^2, x]
```

$$\left\{\left\{e^x \to \frac{-\log[a] + \sqrt{2}\,\log[a]}{2}\right\},\ \left\{e^x \to \frac{-\left(\log[a] + \sqrt{2}\,\log[a]\right)}{2}\right\}\right\}$$

Abb.[Kugelfkt.]

Kugelflächenfunktion

Kugelflächenfunktionen beschreiben die Schwingungsmoden bzgl. Winkelvariablen eines kugelsymmetrischen 3-dimensionalen oszilierenden Systems.

Definition einer Kugelflächenfunktion 3. Grades :

```
Y  :=  SphericalHarmonicY[3,1,theta,phi]
```

Farbkodierung der komplexen Phase :

Laden von Farbprozeduren :

```
<<Colors.m
```

Durch die Funktion **ComplexColor** wird die komplexe Phase des Argumentes in einen Farbbefehl für eine entsprechende Regenbogenfarbe umgewandelt.

```
ComplexColor[z_] :=  HSBColor[Arg[z]/(2 Pi)+1/2, 1, 1]
```

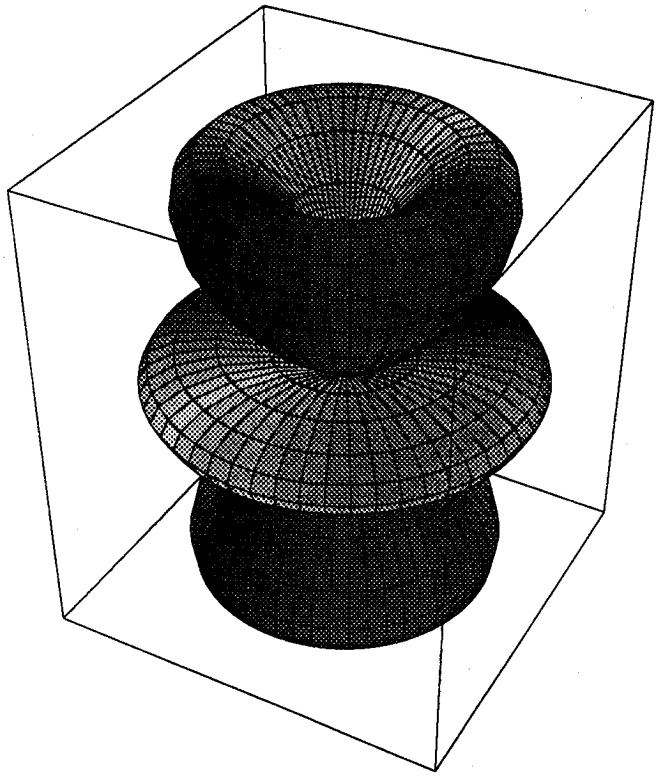
Graphik :

Laden von Graphikprozeduren :

```
<<ParametricPlot3Dnew.m
```

Sphärischer Plot des Absolutbetrags einer Kugelflächenfunktion 3.Grades :
(Die komplexe Phase der Kugelflächenfunktion wird durch die Farbgebung dargestellt)

```
SphericalPlot3D[{Abs[Y],  ComplexColor[Y]},
 {theta,Pi/36,Pi  -  Pi/36,Pi/36},
 {phi,0,2  Pi,Pi/27}]
```



-Graphics3D-