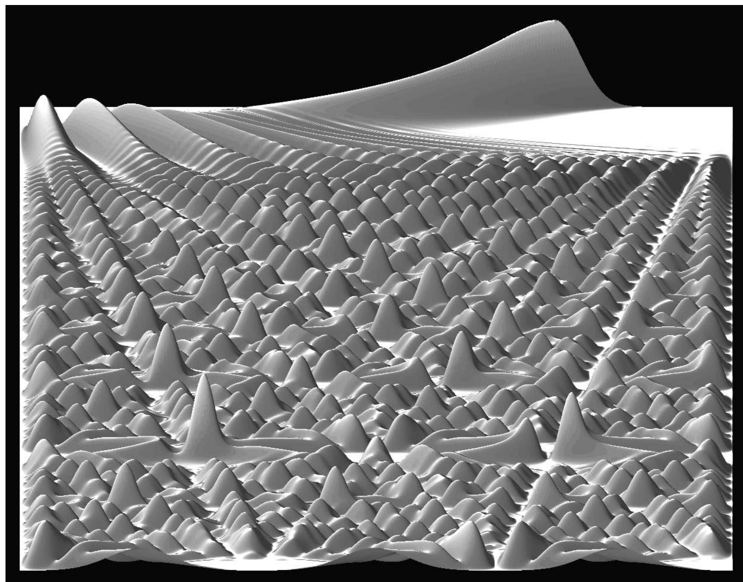
Wolfgang Kinzel / Georg Reents

PHYSIK PER COMPUTER



Programmierung Physikalischer Probleme

mit *Mathematica* und C

© 1996 Spektrum Akademischer Verlag

Vorwort

Der Computer ist heutzutage ein wichtiges Werkzeug der Physik. Die Erfassung und Auswertung umfangreicher Meßdaten und die Steuerung komplexer Experimente sind ohne den Einsatz von elektronischen Rechnern kaum mehr vorstellbar. Auch in der theoretischen Physik hat sich der Computer von einer reinen Rechenmaschine zu einem umfassenden Hilfsmittel gewandelt. Graphische Darstellungen, numerische und algebraische Lösungen von Gleichungen und extensive Simulationen von mikroskopischen Modellen sind wichtige Methoden zur Erforschung physikalischer Gesetzmäßigkeiten geworden.

Aber der Computer ist nicht nur ein Werkzeug, sondern er liefert der Physik auch neuartige Betrachtungsweisen, aus denen neue Fragestellungen resultieren. Früher haben Physiker die Natur hauptsächlich mit Differentialgleichungen beschrieben, heute werden dazu auch diskrete Algorithmen benutzt. Für manche, scheinbar einfache physikalische Modelle gibt es bisher nur eine numerische Antwort. Wir kennen universelle Gesetzmäßigkeiten, die ein Schüler auf dem Taschenrechner nachvollziehen kann, für die es aber (noch?) keine analytische Theorie gibt. Darüber hinaus erschließt der Computer der Physik neue Disziplinen: Neuronale Netzwerke, kombinatorische Optimierung, biologische Evolution, fraktale Strukturbildung und Synergetik sind nur einige Schlagworte aus dem wachsenden Gebiet komplexer Systeme.

Fast jeder Diplomand und Doktorand der Physik nutzt also irgendwann den Computer. Dennoch ist der Einsatz des Rechners in der Lehre und die Ausbildung am Rechner heute noch keinesfalls selbstverständlich, sondern eher die Ausnahme. Die Literatur zu diesem Gebiet ist dementsprechend spärlich. Diese Lücke zu füllen, dazu möchte unser Lehrbuch ein wenig beitragen.

Es entstand aus Vorlesungen an der Universität Würzburg für Studierende der Physik ab dem 5. Semester, also nach den Grundvorlesungen der theoretischen Physik, und es soll der Ergänzung der herkömmlichen Physikvorlesungen dienen, in denen die Möglichkeiten des Computereinsatzes bisher wenig genutzt werden. Wir möchten die Leser dieses Buches anleiten, physikalische Fragestellungen mit dem Rechner zu beantworten. Erfahrungen mit Computern und deren Sprachen sind erwünscht, aber nicht notwendig, denn wir wollen eine Einführung geben und erste

Schritte mit Rechnern und Algorithmen erklären. Dieses Buch enthält keine Details über numerische Mathematik, es bietet keinen Kurs über Programmiersprachen und es lehrt keine fortgeschrittenen Methoden der computerorientierten theoretischen Physik. Aber es führt in eine Vielzahl zum Teil sehr aktueller physikalischer Probleme ein und versucht, diese mit einfachen Algorithmen zu lösen.

Ein Ziel unseres Buches ist, Anregungen zum eigenen Programmieren zu geben. Obwohl fertige Programme beiliegen, sind diese nicht als benutzerfreundliche Experimentierumgebungen gedacht. Wir hoffen, daß man aus ihnen lernen kann, und ermuntern unsere Leser, sie zu modifizieren, oder besser noch, sie neu und effizienter zu schreiben. Manchmal ergeben sich beim Probieren und aus vermeintlichen Fehlversuchen sogar neue Fragestellungen. Man sollte aber die Physik und die zu untersuchenden Gesetzmäßigkeiten nicht aus dem Auge verlieren. Übungsaufgaben zu jedem Abschnitt ergänzen diese Einführung.

Als Programmiersprachen haben wir *Mathematica* und C gewählt. Beides sind moderne Sprachen, die besonders an heutige Arbeitsplatzrechner (z. B. RISC-Workstations unter UNIX) angepaßt sind. Diese Auswahl ist subjektiv und keinesfalls notwendig zur Bearbeitung der Probleme dieses Lehrbuchs. Bei rechenzeitintensiven Computersimulationen wird häufig die Sprache FORTRAN gewählt, deren neue Version FORTRAN90 Elemente der Sprache C übernommen hat. Für die Computeralgebra ist auch MAPLE recht beliebt. Eine Reihe anderer Programmiersprachen wäre also ebenso geeignet. Alle unsere Programme laufen nicht nur auf schnellen Workstations, sondern auch unter DOS oder LINUX auf einem PC.

Von vielen Kollegen und Studenten haben wir Anregungen erhalten, ihnen danken wir hiermit sehr herzlich. Wir möchten erwähnen: M. Biehl, H. Dietz, A. Engel, A. Freking, Th. Hahn, W. Hanke, G. Hildebrand, A. Jung, A. Karch, U. Krey, J. Nestler, M. Opper, M. Schreckenberg und D. Stauffer. Ganz besonderen Dank aber möchten wir drei Personen aussprechen: Martin Lüders hat das Programmpaket *Xgraphics* entwickelt, Martin Schröder hat den Abschnitt über UNIX verfaßt und Ursula Eitelwein hat dieses Buch in LaTeXgeschrieben.

Würzburg, im März 1996

W. Kinzel
G. Reents

Inhaltsverzeichnis

Einleitung **1**

1 Mathematica-Funktionen **3**
1.1 Funktion versus Prozedur 4
1.2 Nichtlineares Pendel 6
1.3 Fouriertransformation 13
1.4 Daten glätten 21
1.5 Nichtlinearer Fit 25
1.6 Multipol-Entwicklung 32
1.7 Wegintegrale 39
1.8 Maxwell-Konstruktion 41
1.9 Beste Spielstrategie 45

2 Lineare Gleichungen **51**
2.1 Quantenoszillator 51
2.2 Elektrisches Netzwerk 56
2.3 Kettenschwingungen 65
2.4 Hofstadter-Schmetterling 69
2.5 Hubbard-Modell 77

3 Iterationen **89**
3.1 Populationsdynamik 89
3.2 Frenkel-Kontorova-Modell 101
3.3 Fraktales Gitter 110
3.4 Neuronales Netzwerk 117

4 Differentialgleichungen **127**
4.1 Runge-Kutta-Methode 127
4.2 Chaotisches Pendel 134
4.3 Stationäre Zustände 144

4.4 Solitonen 150
4.5 Zeitabhängige Schrödinger-Gleichung 159

5 Monte-Carlo-Simulationen 173
5.1 Zufallszahlen 173
5.2 Fraktale Aggregate 180
5.3 Perkolation 189
5.4 Polymer-Ketten 200
5.5 Ising-Ferromagnet 210
5.6 Kürzeste Rundreise 221

A Erste Schritte mit Mathematica 233

B Erste Schritte mit C 247

C Erste Schritte mit UNIX 258

D Erste Schritte mit Xgraphics 268

E Programme 272

Sach- und Namenverzeichnis 308

Einleitung

Dieses Lehrbuch soll den Leser an die Lösung physikalischer Fragestellungen mit dem Computer heranführen. Die Beispiele sind möglichst einfach gehalten, um sie mit wenig Aufwand algebraisch, numerisch und graphisch umzusetzen. Die Anwendung von Computern lernt man hauptsächlich durch eigene Arbeit – ohne ständiges Ausprobieren, ohne spielerische Neugier und ohne den Wunsch zu forschen ist dieses Buch nur von geringem Nutzen.

Fast jeder Abschnitt enthält drei Teile mit den folgenden Überschriften: Physik, Algorithmus und Ergebnisse. Im Physik-Teil wird das Problem formuliert und das Modell eingeführt. Im mittleren Abschnitt wird das Computerprogramm beschrieben, das die Fragestellungen aus dem ersten Teil lösen soll. Dabei werden nur die wesentlichen Befehle erklärt, damit der Leser selbst versuchen kann, das Programm zu vervollständigen. Man kann aber auch den kompletten Computercode aus dem Anhang entnehmen und damit weiterarbeiten; der Quell- und der ausführbare Code sind auf der beiliegenden Diskette vorhanden. Im Ergebnisteil werden dann einige Resultate der Computerrechnungen, meist in Graphikform, erläutert.

Wir haben bewußt eine große Zahl von Beispielen – z.T. auch aus der aktuellen Forschung – behandelt. Die Darstellung der theoretischen Grundlagen kann deshalb nur als erste Einführung angesehen werden. Viele Abschnitte könnten zu ganzen Vorlesungen ausgearbeitet werden, wir müssen auf die weiterführende Literatur verweisen. Dazu haben wir zu jedem Problem einige von uns ausgewählte Lehrbücher oder Originalliteratur zitiert. Die Auswahl dieser Zitate ist naturgemäß sehr subjektiv und erhebt keinerlei Anspruch auf Vollständigkeit. Am Schluß eines jeden Abschnittes wird eine Übungsaufgabe gestellt.

Die PC-Programme sind auf 386-Rechnern mit Koprozessor und VGA-Karte gelaufen. Dies gilt auch für die *Mathematica*-Beispiele (Version 2. 1). Auf dem PC wurde für die C-Programme Turbo C (2.0) von Borland verwendet. Damit kann man bequem Programme entwickeln und Graphik erzeugen.

Von allen Programmen gibt es auch Versionen für Workstations, getestet sind sie auf solchen vom Typ Hewlett Packard HP 9000 Serie 700. Leider ist es dort sehr viel schwieriger als auf dem PC, Graphik zu erzeugen. Schon um nur eine Linie oder einen Kreis auf den Bildschirm zu zeichnen, müssen Displays gesetzt, Fenster

erzeugt, Farbpaletten und gegebenenfalls Ereignisse (Maus- oder Tastaturbefehle) definiert und der Inhalt des Bildspeichers auf den Monitor geschrieben werden. Dazu muß man komplizierte neue Sprachen wie X11, Motif oder PHIGS lernen. Um dies zu erleichtern, hat Martin Lüders das Programmpaket *Xgraphics* auf der Basis der X11-Bibliothek entwickelt, das mit relativ wenig Befehlen elementare Graphik auf hoffentlich jedem Bildschirm einer Workstation erzeugt. Dieses Paket ist auf der beiliegenden Diskette vorhanden. Darüber hinaus kann es sich jeder Interessent über das weltweite Internet von unserem FTP-Server beschaffen und selbst kompilieren. Die Versionen aller C-Programme, die diese Graphik benutzen, werden ebenfalls mitgeliefert.

Alle C-Programme dieses Buches werden in der PC-Version beschrieben. Die Abbildungen dagegen wurden mit *Xgraphics* erzeugt. Meist haben wir die Fenster so gezeigt, wie sie auf dem Bildschirm erscheinen.

Die meisten C-Programme lassen sich während ihres Laufes interaktiv steuern. Sowohl die Versionen für den PC als auch diejenigen für die Workstation erlauben es, mit Tastatureingaben einige Parameter oder Algorithmen zu ändern oder das Programm zu beenden. Auf der Workstation kann man zusätzlich mit der Maus den Lauf steuern.

Wir möchten noch einmal betonen, daß unsere Hard- und Softwareausrüstung keinesfalls für die Bearbeitung der physikalischen Probleme zwingend notwendig ist. Jedes Problem kann auch mit verschiedenen anderen Programmiersprachen gelöst werden. Wir hoffen, daß die Leser dieses Lehrbuchs bald in der Lage sein werden, die Modelle in ihrer Lieblingssprache selbst zu formulieren und zu bearbeiten.

Kapitel 1
Mathematica-Funktionen

Auf der untersten Ebene der Hierarchie der Programmiersprachen arbeitet der Computer kleine Päckchen von An-Aus-Daten (= Bits) schrittweise ab. Jedes Datenpäckchen gibt Anweisungen an elektronische Schalter, wodurch die Ergebnisse von elementaren Rechenoperationen in Datenspeicher geschrieben und neue Daten gelesen werden. Auf dieser Ebene des schrittweisen Abarbeitens elementarer Befehle in Bitform kann der Mensch allerdings kaum schwierigere Probleme formulieren.

Komplexe Algorithmen sollten daher strukturiert sein. Dementsprechend besteht eine höhere Programmiersprache aus Funktionen oder Modulen. Um ein Programm überschaubar zu halten, sollte man Module verwenden, die vom Groben zum Feinen gegliedert sind. Ein guter Programmierstil wird in der Regel mehrere Ebenen davon benutzen, mit dem Vorteil, daß solche Programme leichter modifiziert werden können, weil eventuell nur einzelne Funktionen ausgetauscht werden müssen.

Mathematica ist eine Programmiersprache, die eine große Menge an Funktionen enthält, deren Aufruf aber leichtfällt, da die meisten Parameter Voreinstellungen besitzen und nicht unbedingt angegeben werden müssen. Selbst jede Zeile, die man in *Mathematica* eingibt, kann als eine Funktion betrachtet werden, die sofort ausgeführt wird und für alle folgenden und sogar vorherigen Befehle eine Rolle spielt. In der Hierarchie der Computersprachen ist *Mathematica* daher in diesem Sinne ganz oben anzusiedeln.

Dagegen sind C, FORTRAN, Pascal und Basic zwar ebenfalls höhere Programmiersprachen mit numerischen und graphischen Funktionen, und schließlich ist *Mathematica* in C geschrieben, aber der Aufwand, um sich z. B. schnell einmal den Graphen einer Besselfunktion im Komplexen anzuschauen, ist bei diesen Sprachen wesentlich größer als bei *Mathematica*.

Dieses Kapitel soll in die Nutzung von *Mathematica*-Funktionen einführen. An einfachen Beispielen werden folgende Anwendungen vorgestellt: elliptische Integrale, Reihenentwicklung, Fouriertransformation, χ^2-Verteilung, 3D-Graphik, Vektor-Multiplikation und -Integration, Bestimmung von Nullstellen nichtlinearer Gleichungen und lineares Optimieren.

1.1 Funktion versus Prozedur

Der Unterschied zwischen einem schrittweisen Abarbeiten von Befehlen, das wir hier Prozedur nennen wollen, und einem Aufruf von vorhandenen Funktionen soll an einem einfachen Beispiel erläutert werden: Der Mittelwert von 10000 gleichverteilten Zufallszahlen aus dem Intervall `[0,1]` ist zu bestimmen. Die Daten werden in *Mathematica* wie folgt erzeugt:

```
dataset = Table[Random[],{10000}]
```

Das Ergebnis ist eine Liste von 10000 numerischen Werten.

Zur Bestimmung des Mittelwertes müssen nun alle Zahlen aufsummiert und die Summe durch die Anzahl der Werte geteilt werden. Dies läßt sich schrittweise programmieren und als Funktion `average` schreiben. Die Iteration kann man als `Do`- oder `For`-Schleife schreiben, dabei schirmt die Funktion `Block` die lokalen Variablen `sum` und `average` gegen Definitionen der Außenwelt ab. Dies kann so aussehen:

```
average[data_,length_]:=
              Block[ {sum, average},
               sum = 0. ;
               Do[sum = sum + data[[i]], {i,length}];
               average = sum/length ]
```

Hier wird also Schritt für Schritt eine Zahl aus dem Datensatz `data` zur Variablen `sum` addiert, und dann wird das Ergebnis durch die anzugebende Länge von `data` geteilt. Ganz anders bei der modularen Struktur derselben Aufgabe:

```
average[data_]:= Apply[Plus,data]/Length[data]
```

Diesmal werden Funktionen verwendet, die die Struktur der *Mathematica*-Sprache ausnutzen. `data` wird als Liste von Zahlen eingegeben. `Apply` ersetzt den Kopf `List` der Liste `data` durch den Kopf `Plus` und summiert damit alle Koeffizienten von `data` auf. `Length` bestimmt die Länge der Liste.

Beachten Sie, daß `average` dreimal vorkommt: als abgeschirmte Variable und als Name zweier Funktionen, von denen die eine zwei, die andere dagegen nur ein Argument hat. *Mathematica* erkennt die Unterschiede und verwendet in jedem Fall den richtigen Ausdruck.

Der Leser findet eine Einführung in *Mathematica* im Anhang; trotzdem wollen wir hier einige Besonderheiten dieser neuen Sprache wiederholen: `data_` bedeutet, daß jedes Symbol dafür eingesetzt werden und in die Definition der Funktion übertragen werden kann. Die Zuweisung mit `:=` wertet die rechte Seite jedesmal neu aus, wenn die Funktion aufgerufen wird, während `=` nur einmal auswertet. Zum Beispiel gibt `r:=Random[]` bei jedem Aufruf von `r` einen neuen Wert, während wiederholte Aufrufe von `r` für `r=Random[]` denselben Wert liefern. Indizes von

Listen (Felder, Vektoren, Matrizen, Tensoren usw.) werden durch Doppelklammern `[[i]]` dargestellt.

Um den Unterschied von *Mathematica* zur Programmiersprache C zu erläutern, geben wir hier die prozedurale Version unseres kleinen Beispieles noch einmal in C an. Die Anzahl 10000 erhöhen wir auf eine Million, damit sich Laufzeiten im Sekundenbereich ergeben:

```
#include <stdlib.h>
#include <time.h>

#define LIMIT 1000000

main()
{
 double average(double *,long), dataset[LIMIT];
 clock_t start,end;
 long i;
 for (i=0;i<LIMIT;i++) dataset[i]=(double)rand()/RAND_MAX;
 start=clock();
 printf(" average = %f\n",average(dataset,LIMIT));
 end=clock();
 printf("time= %lf\n",(double)(end-start)/CLOCKS_PER_SEC);
}

double average( double *data,long n )
{
 double sum=0. ;
 long i;
 for(i=0;i<n;i++) sum=sum+data[i] ;
 return sum/n;
}
```

Beachten Sie die Unterschiede zwischen *Mathematica* und C. In C muß alles deklariert werden: Bibliotheksroutinen in den Kopfdateien `<name.h>` und eigene Variable und Funktionen mit `int`, `char`, `float`, `double` usw. C benötigt eine Folge von Anweisungen mit klar definiertem Hauptprogramm und Funktionen, während *Mathematica* immer versucht, alle Symbole mit vorhandenen Definitionen auszuwerten. C ist maschinennah, mit Vor- und Nachteilen. So wird an die Funktion `average` die Maschinenadresse des Feldes `dataset` übergeben, insbesondere ist `dataset` ein Zeiger (=Adresse) auf das Feld, und die einzelnen Werte des Feldes werden mit `dataset[0],...,dataset[999999]` angesprochen. Zeiger werden mit `*` deklariert, also `double *data` bedeutet, daß `data` eine Adresse auf eine `double`-Variable enthält. Vorsicht: `dataset[1000000]` ist irgend etwas

Undefiniertes, dessen Aufruf zwar keine Fehlermeldung hervorruft, aber meistens völlig rätselhafte Ergebnisse produziert.

Leider bedeuten in C und *Mathematica* Komma, Semikolon und verschiedene Klammersymbole jeweils etwas völlig anderes. Aber warum sollen es die Anwender leicht haben?

Was ist nun das Ergebnis der drei Versionen unseres Mittelwert-Programmes? Gibt man `average[dataset]` in *Mathematica* ein, so erhält man auf dem PC nach etwa 10 Sekunden einen Wert in der Nähe von 0.5 . Die dem C-Programm nachempfundene Version `average[dataset, 10000]` dauert dagegen drei- bis viermal länger und liefert natürlich dasselbe Ergebnis. Die Workstation berechnet diese Ergebnisse jeweils etwa zehnmal schneller als der PC.

Das C-Programm dagegen wird zunächst kompiliert, z. B. mit `gcc -o summe summe.c -lm` und dann mit dem File-Namen `summe` aufgerufen. Jetzt dauert es auf dem PC nur 8.5 Sekunden und auf der HP 0.56 Sekunden, bis das Ergebnis vorliegt, obwohl die Anzahl der Rechenschritte um den Faktor 100 erhöht wurde. Hier zeigt sich ein wesentlicher Nachteil von *Mathematica*: Bei numerischen Rechnungen ist es extrem langsam gegenüber C oder vergleichbaren Sprachen.

Literatur

B.W. Kernighan, D.M. Ritchie, *Programmieren in C*, Carl Hanser Verlag, 1990.

Stephen Wolfram, *Mathematica: A System for Doing Mathematics by Computer*, Addison Wesley, 1991.

1.2 Nichtlineares Pendel

Das mathematische Pendel ist ein Standardbeispiel der Physikvorlesung über klassische Mechanik. Schon in der Schule wird die lineare Näherung dieses Problems besprochen. In der Vorlesung wird die exakte Lösung durch ein elliptisches Integral ausgedrückt, aber erst der Computer bietet die Möglichkeit, sich diese nichtelementaren Funktionen anzuschauen und genauer zu analysieren. Daher wollen wir dieses Standardbeispiel der theoretischen Physik an den Anfang unseres Lehrbuches stellen.

Physik

Wir beginnen mit dem physikalischen Modell: Eine punktförmige Masse m an einem masselosen, starren Faden der Länge l schwingt im Schwerefeld. $\varphi(t)$ sei der

Winkel der Auslenkung aus der Ruhelage zur Zeit t, die Reibung sei vernachlässigt. Nach den Gesetzen der Mechanik ist die Energie E des Pendels konstant, also gilt

$$E = \frac{m}{2}\, l^2\, \dot{\varphi}^2 - m\, g\, l\, \cos\varphi = -m\, g\, l\, \cos\varphi_0\,, \tag{1.1}$$

wobei φ_0 der Maximalausschlag mit $\dot{\varphi}_0 = 0$ sein soll und $\dot{\varphi} = \frac{d\varphi}{dt}$ die Winkelgeschwindigkeit ist. Aus dieser Gleichung kann man $\varphi(t)$ mit einem kleinen Trick gewinnen: Man separiert t und φ, indem man (1.1) erst nach $\left(\frac{d\varphi}{dt}\right)^2$ und dann nach dem Differential dt auflöst. Für eine Halbperiode, in der φ mit t zunimmt, erhält man

$$dt = \sqrt{\frac{l}{2g}}\, \frac{d\varphi}{\sqrt{\cos\varphi - \cos\varphi_0}}\,. \tag{1.2}$$

Bei passender Wahl des Zeitnullpunktes ($\varphi(0) = 0,\ \dot{\varphi}(0) > 0$) ergibt sich hieraus durch Integration:

$$t(\varphi) = \sqrt{\frac{l}{2g}} \int_0^{\varphi} \frac{d\varphi'}{\sqrt{\cos\varphi' - \cos\varphi_0}}\,. \tag{1.3}$$

Die Schwingungsdauer T ist offensichtlich viermal die Zeit, die das Pendel bis zum Maximalausschlag φ_0 braucht,

$$T = 4\, t(\varphi_0)\,. \tag{1.4}$$

Da der Integrand von (1.3) für $\varphi' \to \varphi_0$ divergiert, ist es sinnvoll, die Substitution

$$\sin\psi = \sin\frac{\varphi}{2} / \sin\frac{\varphi_0}{2} \tag{1.5}$$

durchzuführen, mit der man folgendes Integral erhält:

$$t(\varphi) = \sqrt{\frac{l}{g}} \int_0^{\psi} \frac{d\psi'}{\sqrt{1 - \sin^2\frac{\varphi_0}{2}\sin^2\psi'}} = \sqrt{\frac{l}{g}}\, F\left(\psi, \sin\frac{\varphi_0}{2}\right). \tag{1.6}$$

Hier divergiert der Integrand nur noch beim Maximalausschlag $\varphi_0 = \pi$. Die Funktion F nennt man elliptisches Integral der ersten Art. Für $\varphi = \varphi_0$ ist nach (1.5) $\psi = \frac{\pi}{2}$; die Schwingungsdauer T berechnet sich daher zu

$$T = 4\sqrt{\frac{l}{g}}\, F\left(\frac{\pi}{2}, \sin\frac{\varphi_0}{2}\right) = 4\sqrt{\frac{l}{g}}\, K\left(\sin\frac{\varphi_0}{2}\right). \tag{1.7}$$

Die Funktion K heißt vollständiges elliptisches Integral erster Art. Sowohl F als auch K sind in *Mathematica* vorhanden.

Für kleine Auslenkungen φ ist die potentielle Energie näherungsweise quadratisch. Bekanntlich ist dann die Kraft linear in φ, und die Lösung der Bewegungsgleichung ist eine Sinusschwingung mit Schwingungsdauer $T = 2\pi\sqrt{l/g}$. Mit größerem Ausschlag φ_0 wird die Kraft eine nichtlineare Funktion von φ, und T hängt von φ_0 ab.

Wir wollen den Einfluß der Nichtlinearität studieren. Um verschiedene Kurven $\varphi(t)$ vergleichen zu können, skalieren wir φ mit φ_0 und t mit T. Außerdem betrachten wir im Phasenraumdiagramm $\dot{\varphi}$ gegen φ für verschiedene Energien E. Die Entstehung höherer harmonischer Schwingungen mit wachsender Nichtlinearität wird durch eine Fouriertransformation sichtbar. Schließlich wollen wir T nach der Größe $\sin^2\frac{\varphi_0}{2}$ entwickeln, die ein Maß für die Nichtlinearität ist.

Algorithmus

Die Funktionen $K(k)$ und $F(\psi, k)$ stehen in *Mathematica* zur Verfügung und werden aufgerufen als

EllipticK[k^2] und EllipticF[ψ,k^2]

Man muß auf die Argumente achtgeben, weil es diesbezüglich leider mehrere gebräuchliche Konventionen gibt. Es gilt also $K(k) =$ `EllipticK[k^2]` und entsprechend für die anderen elliptischen Integrale. Die Umlaufzeit erhält man nach (1.7) einfach mit

```
T[phi0_] = 4 EllipticK[Sin[phi0/2]^2]
```

wobei wir $\sqrt{l/g} = 1$ gesetzt haben. Der Befehl `Plot` zeichnet die Umlaufzeit als Funktion von φ_0 :

```
Plot[ T[phi0], {phi0, 0, Pi}, PlotRange -> {0,30} ]
```

Da T für $\varphi_0 \to \pi$ divergiert, sollte man `PlotRange` setzen und sich nicht daran stören, daß *Mathematica* vor $T(\pi)$ warnt.

$t(\varphi)$ erhält man mit `EllipticF` nach (1.6). Wir wollen jedoch die Umkehrfunktion $\varphi(t)$ berechnen. Aber auch diese Funktionen sind vorhanden; es gilt (wieder mit $\sqrt{l/g} = 1$)

```
psi[t_,phi0_]=JacobiAmplitude[t, Sin[phi0]^2]
```
bzw.
```
sinuspsi[t_,phi0_]=JacobiSN[t, Sin[phi0]^2]
```

Nun ersetzen wir mit (1.5) die Variable ψ durch φ, skalieren φ mit φ_0 und t mit T (wir definieren $x = t/T$) und erhalten schließlich die skalierte Funktion

```
phiscal[x_,phi0_]:=
  2 ArcSin[Sin[phi0/2] sinuspsi[x T[phi0],phi0]]/phi0
```

Die Funktion `JacobiSN[t, Sin[phi0]^2]` hat überdies die richtige Symmetrie, so daß sie nicht nur für $0 \leq x \leq \frac{1}{4}$, sondern für alle x die Lösung liefert. Die Funktion $\varphi(t/T)/\varphi_0$ wollen wir uns für verschiedene Anfangsausschläge φ_0 ansehen. Dazu erzeugen wir uns ein Feld von fünf φ_0-Werten, `phi0[1]=N[0.1 Pi]`,..., `phi0[5]= N[0.999 Pi]` und bilden eine Liste von Funktionen

```
fliste = Table[phiscal[x,phi0[i]],{i,5}]
```

Diese Liste kann man nun mit `Plot` zeichnen, allerdings muß man zuvor den Befehl `Evaluate` darauf anwenden.

Um die Entstehung höherer harmonischer Schwingungsanteile mit wachsender Nichtlinearität φ_0 zu beobachten, soll $\varphi(t)$ in eine Fourierreihe entwickelt werden. Am einfachsten geht dies mit dem eingebauten `Fourier`-Befehl; man muß sich dazu allerdings eine Liste von diskreten Werten $\varphi(t_i), i = 1, \ldots, N$, anlegen (siehe nächsten Abschnitt). Die Beträge der komplexen Fourierkoeffizienten erhält man dann als Liste

```
fouliste = Abs[Fourier[liste]]
```

Setzt man in Gleichung (1.1) die Energie konstant, so erhält man Kurven in der $(\dot{\varphi}, \varphi)$-Ebene, dem sogenannten Phasenraum. Diese Kurven können in *Mathematica* verblüffend einfach gezeichnet werden: Man ruft `ContourPlot` für die Funktion $E(\dot{\varphi}, \varphi)$ auf. Dabei kann man mit der Option `Contours->{E`$_1$`, E`$_2$`, ..., E`$_n$`}` n Höhenlinien mit verschiedenen Werten von `E` zeichnen lassen.

Zuletzt noch einige Bemerkungen zur Reihenentwicklung von $T(\varphi_0)$. Im Prinzip könnte Gleichung (1.7) einfach in eine Taylorreihe nach $m = \sin^2 \frac{\varphi_0}{2}$ mit `Series` entwickelt werden. *Mathematica* schreibt dann allerdings nur formale Ableitungen von `EllipticK` hin, und auch ein `N[...]` Befehl ändert dies nicht.

Man muß daher beim Integral (1.6) für $\psi = \frac{\pi}{2}$ beginnen. Der Integrand ist

```
f = 1/Sqrt[1 - m Sin[psi]^2]
```

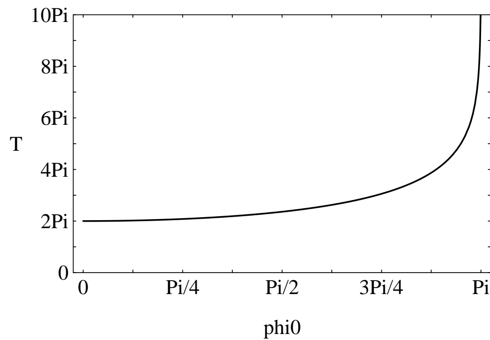
`f` kann man nach `m` entwickeln, z. B. bis zur 10-ten Ordnung in `m` um den Wert 0,

```
g = Series [f, {m,0,10}]
```

und dann jeden einzelnen Ausdruck mit `Integrate` über ψ integrieren. Danach wird `m` wieder mit `/. m -> Sin[phi0/2]^2` auf den Wert $\sin^2 \frac{\varphi_0}{2}$ gesetzt.

Ergebnisse

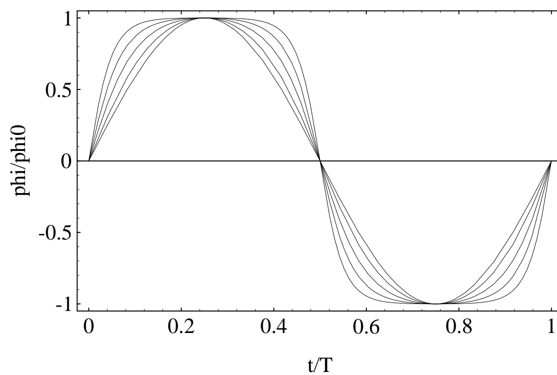
Wir haben untersucht, wie sich die Nichtlinearität, gemessen durch den Maximalausschlag φ_0, auf die Bewegung des Pendels auswirkt. Bild 1.1 zeigt die Umlaufzeit T als Funktion des Maximalausschlages φ_0. Das Pendel braucht eine unendlich



1.1 Periodendauer T als Funktion des Maximalausschlages φ_0.

lange Zeit, um genau am oberen Umschlagpunkt ($\varphi_0 = \pi$) zum Stillstand zu kommen. Man sieht, daß T mit wachsendem φ_0 zunächst nur wenig von seinem Wert $T = 2\pi$ der harmonischen Schwingung abweicht, erst oberhalb der Auslenkung von $90° = \frac{\pi}{2}$ wird T merklich größer.

Der Einfluß von φ_0 auf die Kurve $\varphi(t)$ wird in Bild 1.2 deutlich. φ/φ_0 ist gegen t/T aufgetragen, um verschiedene Kurven miteinander vergleichen zu können



1.2 Skalierte Amplitude des Pendels $\varphi(t/T)/\varphi_0$ für verschiedene Maximalausschläge $\varphi_0 = \frac{1}{10}\pi,\ \frac{4}{5}\pi,\ \frac{19}{20}\pi,\ \frac{999}{1000}\pi$ (von innen nach außen).

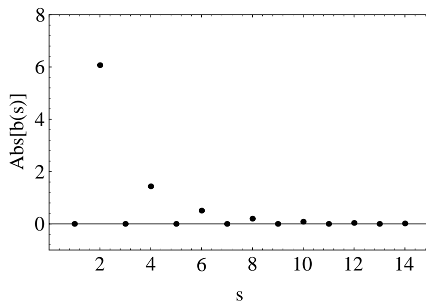
(man beachte: T divergiert für $\varphi_0 \to \pi$). Für kleine φ_0-Werte erhält man eine Sinusschwingung. Mit wachsendem φ_0 wird die Zeit an den oberen Umkehrpunkten $\pm\varphi_0$ immer größer; daher wird $\varphi(t)$ immer flacher, bis man für $\varphi_0 = \pi$ eine Stufenfunktion erhält.

Zerlegt man $\varphi(t)$ in harmonische Schwingungen der Form $b_s e^{i\omega_s t}$, so gilt wegen der Periodizität $\varphi(t) = \varphi(t+T)$, daß die Frequenzen ω_s ganzzahlige Vielfache von

$\frac{2\pi}{T}$ sind. Die in *Mathematica* vorhandene diskrete Fouriertransformation benutzt bei N Datenpunkten $\{\varphi(t_r),\ r = 1, \ldots, N\}$ die Frequenzen $\{\omega_s = \frac{2\pi}{T}(s-1),\ s = 1, \ldots, N\}$ und berechnet die Koeffizienten nach der Formel

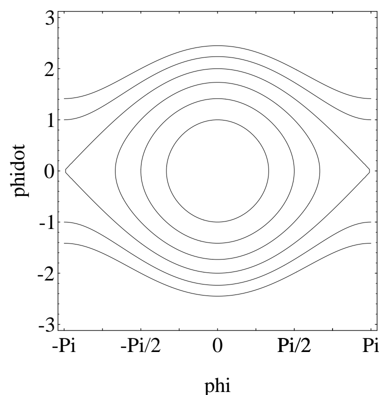
$$b_s = \frac{1}{\sqrt{N}} \sum_{r=1}^{N} \varphi(t_r)\, e^{2\pi i(s-1)(r-1)/N} \, .$$

Wegen der Symmetrie $\varphi(t) = -\varphi(-t)$ verschwinden für alle ungeraden s die Koeffizienten b_s. Der erste nichtverschwindende, nämlich b_2, ist die Amplitude der Grundschwingung $\omega_2 = \frac{2\pi}{T}$. Die weiteren Koeffizienten $b_4, b_6, b_8, \ldots$ geben die Amplituden der Oberschwingungen $3\,\omega_2, 5\,\omega_2, 7\,\omega_2, \ldots$ Der Ausdruck `fouliste` enthält für $\varphi_0 = 0.999\pi$ die Beträge der Koeffizienten b_s, die in Bild 1.3 mit `ListPlot` dargestellt sind.



1.3 Betrag der Fourierkoeffizienten der anharmonischen Schwingung $\varphi(t)$ für den Maximalausschlag $\varphi_0 = 0.999\pi$.

Das Wechselspiel zwischen Auslenkung φ und Winkelgeschwindigkeit $\dot{\varphi}$ kann man im Phasenraum $(\varphi, \dot{\varphi})$ deutlich machen; dabei wird die Zeit t eliminiert. Bild 1.4 zeigt solche Kurven im Phasenraum für verschiedene Energien E. Für kleine φ_0



1.4 Phasenraum-Plot $\dot{\varphi}$ gegen φ für verschiedene Energien $\frac{E}{m\,g\,l} = -\frac{1}{2}$, 0, $\frac{1}{2}$, 1, $\frac{3}{2}$ und 2 (von innen nach außen).

erhält man einen Kreis, der sich mit wachsender Energie verformt. Für $E > m\,g\,l$

schlägt das Pendel über, es bewegt sich nur noch in eine Richtung und hat eine Winkelgeschwindigkeit $\dot{\varphi} \neq 0$ auch am oberen Umschlagpunkt.

Mit dem Befehl

```
tseries = 4 Integrate[g,{psi,0,Pi/2}] /. m -> Sin[phi0/2]^2
```

erhält man die symbolische Entwicklung von T nach $\sin^2(\varphi_0/2)$. *Mathematica* produziert die folgende etwas unübersichtliche Bildschirmausgabe:

```
               phi0 2            phi0 4             phi0 6
        Pi Sin[----]    9 Pi Sin[----]    25 Pi Sin[----]
                 2                 2                  2
  2 Pi + ------------ + -------------- + --------------- +
               2              32               128

             phi0 8              phi0 10               phi0 12
  1225 Pi Sin[----]    3969 Pi Sin[----]     53361 Pi Sin[----]
               2                    2                      2
> ----------------- + ------------------ + ------------------- +
        8192                 32768               524288

               phi0 14                    phi0 16
  184041 Pi Sin[----]     41409225 Pi Sin[----]
                 2                          2
> -------------------- + ---------------------- +
        2097152                536870912

                  phi0 18                    phi0 20
  147744025 Pi Sin[----]     2133423721 Pi Sin[----]
                    2                           2                phi0 2 11
> ----------------------- + ------------------------ + O[Sin[----] ]
        2147483648                34359738368                  2
```

Wir werden deshalb in Zukunft bei solchen Ergebnissen die entsprechende TEX–Form angeben, die man sich übrigens mit dem Befehl `TeXForm` von *Mathematica* erzeugen und mit TEX oder LATEX übersetzen lassen kann:

$$
\begin{aligned}
2\,\pi &+ \frac{\pi\sin^2\frac{\varphi_0}{2}}{2} + \frac{9\,\pi\sin^4\frac{\varphi_0}{2}}{32} + \frac{25\,\pi\sin^6\frac{\varphi_0}{2}}{128} + \frac{1225\,\pi\sin^8\frac{\varphi_0}{2}}{8192} + \\
&+ \frac{3969\,\pi\sin^{10}\frac{\varphi_0}{2}}{32768} + \frac{53361\,\pi\sin^{12}\frac{\varphi_0}{2}}{524288} + \frac{184041\,\pi\sin^{14}\frac{\varphi_0}{2}}{2097152} \\
&+ \frac{41409225\,\pi\sin^{16}\frac{\varphi_0}{2}}{536870912} + \frac{147744025\,\pi\sin^{18}\frac{\varphi_0}{2}}{2147483648} \\
&+ \frac{2133423721\,\pi\sin^{20}\frac{\varphi_0}{2}}{34359738368} + \mathcal{O}(\sin^2\frac{\varphi_0}{2})^{11} \,.
\end{aligned}
$$

Ein Vergleich mit dem exakten $T(\varphi_0)$ zeigt, daß der Fehler dieser Näherungslösung höchstens im Prozentbereich liegt, solange φ_0 kleiner als $120°$ ist, dann aber deutlich ansteigt.

Übung

a) Berechnen Sie die Umlaufzeit T als Funktion von φ_0 einmal mit `EllipticK` und zum anderen mit `NIntegrate` und vergleichen Sie die Rechenzeiten und die Genauigkeiten.

b) Welchen Anteil haben die Oberschwingungen in $\varphi(t)$ als Funktion der maximalen Auslenkung φ_0 ($\widehat{=}$ wachsender Nichtlinearität)? Berechnen Sie die Fourierkoeffizienten $|b_s|/|b_2|$ als Funktion von φ_0.

c) Programmieren Sie folgenden Algorithmus für das vollständige elliptische Integral $K(\sin\alpha)$ und vergleichen Sie das Ergebnis mit `EllipticK[`$\sin^2\alpha$`]`:

Start: $a_0 = 1\,,\qquad b_0 = \cos\alpha\,,$
Iteration: $a_{i+1} = \frac{1}{2}(a_i + b_i)\,,\quad b_{i+1} = \sqrt{a_i b_i}\,,$
Stop: $|\, a_n - b_n \,| < \varepsilon\,,$
Prüfe: `EllipticK[`$\sin^2\alpha$`]` $\simeq \dfrac{\pi}{2a_n}\,.$

Literatur

G. Baumann, *Mathematica in der Theoretischen Physik*, Springer Verlag, 1993.

R. E. Crandall, *Mathematica for the Sciences*, Addison Wesley, 1991.

1.3 Fouriertransformation

Erstaunlich oft kann man physikalische Probleme in guter Näherung durch lineare Gleichungen beschreiben. In solchen Fällen steht dem Forscher ein mächtiges Werkzeug zur Verfügung: die Zerlegung eines Signals in eine Summe von harmonischen Schwingungen. Dieses Werkzeug, das mathematisch gut erforscht ist, wird *Fouriertransformation* genannt. Mit komplexwertigen Funktionen kann man die Transformation besonders kompakt formulieren. Fast jedes Signal, selbst ein unstetiges, kann als Grenzwert einer Summe von stetigen Schwingungen dargestellt werden. Eine wichtige Anwendung der Fouriertransformation ist die Lösung linearer Differentialgleichungen. Nicht nur in der Physik, sondern auch in der Bildverarbeitung,

der Signalübertragung, der Elektronik und in vielen anderen Gebieten spielt die Zerlegung einer Funktion in einfache Schwingungen eine große Rolle.

Häufig stehen Daten nur zu diskreten Zeit- oder Raumpunkten zur Verfügung. In diesem Fall sind die numerischen Algorithmen der Fouriertransformation besonders schnell. Wegen dieses Vorteils wollen wir hier die Fouriertransformation diskreter Daten untersuchen. In den folgenden Abschnitten werden wir sie zur Glättung von Daten, zur Berechnung elektrischer Schaltkreise und zur Analyse von Gitterschwingungen benutzen.

Mathematik

Es sei $a_r,\ r = 1, \ldots, N$, eine Folge komplexer Zahlen. Dann ist deren Fouriertransformierte die Folge $b_s, s = 1, \ldots, N$, mit

$$b_s = \frac{1}{\sqrt{N}} \sum_{r=1}^{N} a_r \, e^{2\pi i(r-1)(s-1)/N} \,. \tag{1.8}$$

Diese Formel hat den Vorteil, fast symmetrisch in a_r und b_s zu sein, denn die inverse Transformation ist

$$a_r = \frac{1}{\sqrt{N}} \sum_{s=1}^{N} b_s \, e^{-2\pi i(r-1)(s-1)/N} \,. \tag{1.9}$$

Das Signal $\{a_1, \ldots, a_N\}$ ist also in eine Summe von Schwingungen

$$c_s(r) = \frac{b_s}{\sqrt{N}} \, e^{-i\omega_s(r-1)} \tag{1.10}$$

mit den Frequenzen $\omega_s = 2\pi(s-1)/N$ zerlegt worden. $b_s = |b_s| e^{i\varphi_s}$ enthält eine Amplitude und eine Phase. Beide Gleichungen (1.8) und (1.9) können auf alle ganzen Zahlen r und s erweitert werden. Wegen $\exp(2\pi i k) = 1$ für $k \in \mathbb{Z}$ sind dann a_r und b_s periodisch in r bzw. s mit der Periode N:

$$a_r = a_{r+k\,N} \;, \quad b_s = b_{s+k\,N} \;, \tag{1.11}$$

so daß man in jeder Summe den Index r oder s über jedes Intervall der Länge N laufen lassen kann. Die kleinste Frequenz der Schwingungen ist $\omega_2 = 2\pi/N$ ($\widehat{=}$ „Wellenlänge“ N), und alle anderen Frequenzen sind ganzzahlige Vielfache von ω_2. Die größte Frequenz ist $\omega_N = 2\pi\frac{N-1}{N}$, denn $c_{N+1}(r)$ hat wegen $b_{N+1} = b_1$ dieselben Funktionswerte wie $c_1(r)$.

Wenn alle Signalwerte a_r reell sind, haben deren Fourierkoeffizienten b_s die folgende Symmetrie:

$$\begin{aligned} b_s^* &= \frac{1}{\sqrt{N}} \sum_{r=1}^{N} a_r \, e^{-2\pi i(r-1)(s-1)/N} \\ &= \frac{1}{\sqrt{N}} \sum_{r=1}^{N} a_r \, e^{2\pi i(r-1)(-s+2-1)/N} = b_{-s+2} \,. \end{aligned}$$

Dabei bedeutet b^* das konjugiert Komplexe von b. Daraus folgt

$$b_s = b^*_{-s+2} = b^*_{N-s+2} \,, \tag{1.12}$$

also sind nur die Werte $b_1, ..., b_{N/2+1}$ relevant, $b_{N/2+2}$ bis b_N sind redundant.

Eine nützliche Eigenschaft der Fouriertransformierten ergibt sich bei der Faltung zweier Folgen $\{f_r\}$ und $\{g_r\}$. Bei dieser Operation werden die Folgenelemente f_r aufsummiert und dabei mit den g_j gewichtet. Man kann dies als lokale Mittelung interpretieren und auf diese Weise z. B. den Einfluß von Meßfehlern mathematisch behandeln. Die Faltung ist folgendermaßen definiert:

$$h_r = \sum_{j=1}^{N} f_{r+1-j} \, g_j \,. \tag{1.13}$$

Die entsprechenden Fouriertransformierten seien die Folgen $\tilde{h}_s, \tilde{g}_s$ und $\tilde{f}_s$. Fügt man die Fourierentwicklungen

$$\begin{aligned} \tilde{h}_s &= \frac{1}{\sqrt{N}} \sum_{r=1}^{N} h_r \, e^{2\pi i(r-1)(s-1)/N} \,, \\ g_j &= \frac{1}{\sqrt{N}} \sum_{m=1}^{N} \tilde{g}_m \, e^{-2\pi i(m-1)(j-1)/N} \,, \\ f_{r+1-j} &= \frac{1}{\sqrt{N}} \sum_{n=1}^{N} \tilde{f}_n \, e^{-2\pi i(n-1)(r-j)/N} \end{aligned}$$

nach Gleichung (1.13) zusammen, ergibt sich

$$\tilde{h}_s = \left(\frac{1}{\sqrt{N}}\right)^3 \sum_{r,j,m,n} \tilde{g}_m \, \tilde{f}_n \, e^{\frac{2\pi i}{N}[(r-1)(s-1)-(m-1)(j-1)-(n-1)(r-j)]} \,.$$

Die Summen über r und j lassen sich mit

$$\sum_{l=1}^{N} e^{2\pi i \, lq/N} = N \, \delta_{q,kN} \,, k \in \mathbb{Z} \,,$$

ausführen, und es verbleibt

$$\tilde{h}_s = \sqrt{N} \sum_{m,n} \tilde{g}_m \, \tilde{f}_n \, e^{\frac{2\pi i}{N}(m-s)} \, \delta_{s-n,0} \, \delta_{n-m,0} = \sqrt{N} \, \tilde{g}_s \, \tilde{f}_s \,. \tag{1.14}$$

Die Faltung wird also nach der Transformation zum einfachen Produkt. Die Rücktransformation gibt dann die gefaltete Funktion

$$h_r = \sum_{s=1}^{N} \tilde{g}_s \, \tilde{f}_s \, e^{-2\pi i(s-1)(r-1)/N} \,. \tag{1.15}$$

Wir werden die Faltung im nächsten Abschnitt zur Glättung von Meßdaten benutzen.

Algorithmus

In *Mathematica* erhält man eine Liste der Fourierkoeffizienten {b₁, ..., bN} mit

```
Fourier[{a1, ..., aN}]
```

und deren Inverses mit

```
InverseFourier[{b1, ..., bN}]
```

Trotzdem ist es vielleicht interessant, sich einmal den Algorithmus der schnellen Fouriertransformation (*Fast Fourier Transform* = *FFT*) anzuschauen. Um alle Folgenglieder b_s aus Gleichung (1.8) zu berechnen, kann man sich vorstellen, daß eine Matrix der Form

$$W_{s,r}(N) = \frac{1}{\sqrt{N}} \, e^{2\pi i(r-1)(s-1)/N}$$

mit einem Vektor $\boldsymbol{a} = (a_1, ..., a_N)$ multipliziert wird; dies erfordert N^2 Rechenschritte. Die *FFT* kann dies aber in einer Anzahl von Schritten berechnen, die nur wie $N \log N$ anwächst. Dazu wird die Summe (1.8) in ungerade und gerade r-Werte aufgespalten:

$$\begin{aligned} b_s(N) &= \sum_{t=1}^{N/2} W_{s,2t-1}(N) \, a_{2t-1} + \sum_{t=1}^{N/2} W_{s,2t}(N) \, a_{2t} \\ &= \sum_{t=1}^{N/2} \frac{1}{\sqrt{2}} W_{s,t}(N/2) \, a_{2t-1} + e^{2\pi i(s-1)/N} \sum_{t=1}^{N/2} \frac{1}{\sqrt{2}} W_{s,t}(N/2) \, a_{2t} \\ &= \frac{1}{\sqrt{2}} \left(b_s^u(N/2) + e^{2\pi i(s-1)/N} \, b_s^g(N/2) \right) . \end{aligned} \tag{1.16}$$

Dabei sind $b_s^u(N/2)$ und $b_s^g(N/2)$ die Fouriertransformierten der Koeffizienten $\{a_1, a_3, \ldots, a_{N-1}\}$ bzw. $\{a_2, a_4, \ldots, a_N\}$, zweier Folgen also mit $N/2$ Gliedern (g und u stehen für gerade und ungerade). Wiederholt man die Aufspaltung für jeden der beiden Teile, so erhält man $b_s^{uu}, b_s^{ug}, b_s^{gu}$ und b_s^{gg}. Diesen Prozeß wiederholt man solange, bis noch jeweils ein a_r-Wert übrig bleibt, der dann irgendeinem Wert, z. B. $b_s^{gugguguu}$ entspricht; die letzte Zuordnung kann angegeben werden. Mit einigen zusätzlichen Buchhaltungstricks läßt sich damit ein schneller Algorithmus programmieren.

Weil dieser Algorithmus nur dann optimal funktioniert, wenn N eine Potenz von 2 ist, gehen wir im folgenden davon aus, daß N von der Form $N = 2^j$ ist. Was die Anzahl $F(N)$ der Rechenschritte betrifft, die benötigt wird, um die Fouriertransformation einer N-gliedrigen Folge zu berechnen, so liest man von Gleichung (1.16) die folgende Rekursionsbeziehung ab:

$$F(N) = 2\,F(N/2) + k\,N\,. \tag{1.17}$$

Der Term $2\,F(N/2)$ ist unmittelbar klar wegen $b_s^u(N/2)$ und $b_s^g(N/2)$. Der Index s in Gleichung (1.16) läuft von 1 bis N, wobei anzumerken ist, daß wegen der Periodizität mit $N/2$ die Werte von $b_s^u(N/2)$ und $b_s^g(N/2)$ dabei zweimal durchlaufen werden. Ein zu N proportionaler Anteil kommt hinzu, weil N Phasenfaktoren $e^{2\pi i(s-1)/N}$ zu berechnen sind und außerdem N Multiplikationen und Additionen auszuführen sind. Auch das Sortieren der $\{a_r\}$ in gerade und ungerade kann man hier unterbringen. Es ist eine leichte Übung zu zeigen, daß $F(N) = N\,[F(1) + k\,\log_2 N]$ die Lösung der Rekursionsgleichung (1.17) ist, mit $F(1) = 0$ als Startwert also $F(N) = k\,N\,\log_2 N$.

Anwendung

Als Beispiel wollen wir uns die Fouriertransformation eines Rechteckimpulses der Breite 2τ anschauen,

$$f(t) = \frac{1}{2}\left(\operatorname{sign}(\tau + t) + \operatorname{sign}(\tau - t)\right). \tag{1.18}$$

Die Funktion f sei zunächst im Intervall $[-\frac{T}{2}, \frac{T}{2}]$ mit $\frac{T}{2} > \tau$ definiert und soll periodisch mit der Periode T fortgesetzt werden. Deren Fourierentwicklung lautet

$$f(t) = \sum_{n=-\infty}^{\infty} \tilde{f}_n\, e^{in\omega t} \tag{1.19}$$

mit $\omega = \frac{2\pi}{T}$, wobei die $\tilde{f}_n$ durch Integrale über $f(t)$ bestimmt werden:

$$\tilde{f}_n = \frac{1}{T} \int_{-T/2}^{T/2} f(t)\, e^{-in\omega t}\, dt\,. \tag{1.20}$$

Dieses Integral läßt sich für den Rechteckimpuls leicht ausführen, mit dem Ergebnis

$$\tilde{f}_n = \frac{2}{T} \frac{\sin(n\omega\tau)}{n\omega}\,. \tag{1.21}$$

Je nach Länge der Periode wird die Funktion $\sin(x)/x$ an verschiedenen Punkten $2\pi n\tau/T$ berücksichtigt, und die Koeffizienten $\tilde{f}_n$ verschwinden, wenn $2n\tau/T$ eine ganze Zahl ist.

Nun tasten wir $f(t)$ innerhalb einer Periode an N diskreten Werten $t_r = -T/2 + r\,T/N$ ab, wenden `Fourier` auf die Folge der Zahlen $\{a_r = f(t_r)\}_{r=1}^{N}$ an und erhalten deren Fouriertransformierte $\{b_1, b_2, \ldots, b_N\}$. Wir wollen den Zusammenhang zwischen den exakten Fourierkoeffizienten $\tilde{f}_n$ und den b_s aus der diskreten Fouriertransformation klären. Dazu ersetzen wir in Gleichung (1.8) die Funktionswerte $a_r = f(t_r)$ durch die unendliche Fourierreihe (1.19) mit $\omega = \frac{2\pi}{T}$. Die r-Summation ergibt Kronecker-Deltas, und für gerades N erhalten wir die folgende Beziehung:

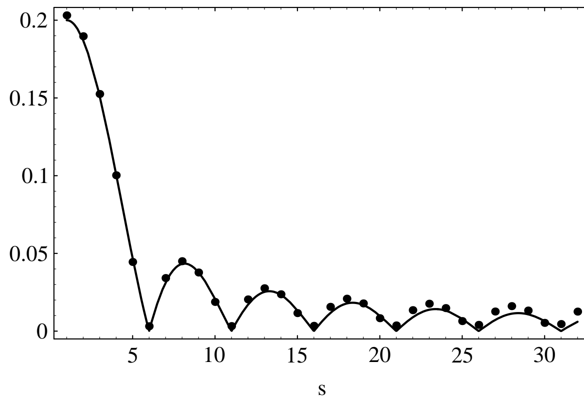
$$\frac{b_s}{\sqrt{N}} = e^{i\,\pi(N-2)(s-1)/N} \sum_{k=-\infty}^{\infty} \tilde{f}_{1-s+k\,N}\,. \tag{1.22}$$

Der Vergleich der Beträge liefert also:

$$\frac{|b_s|}{\sqrt{N}} = \left| \sum_{k=-\infty}^{\infty} \tilde{f}_{1-s+k\,N} \right|. \tag{1.23}$$

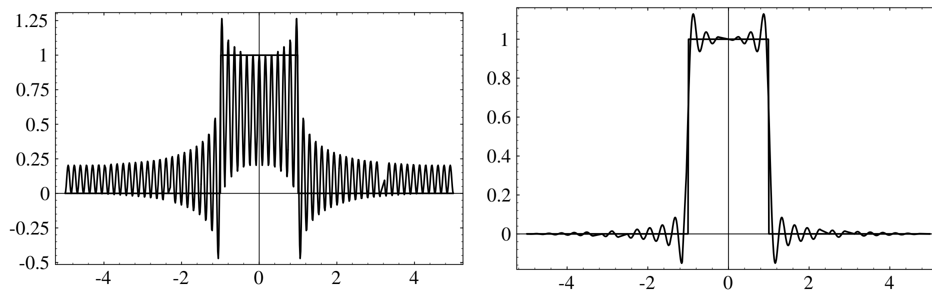
Weil $|\tilde{f}_n| \to 0$ geht für $n \to \infty$, ist $|b_s|/\sqrt{N} \approx |\tilde{f}_{1-s}|$ für $s = 1, 2, \ldots, \frac{N}{2}$. Ob $|b_s|/\sqrt{N} < |\tilde{f}_{1-s}|$ oder $|b_s|/\sqrt{N} > |\tilde{f}_{1-s}|$ gilt, wie sich also das sogenannte *aliasing* auswirkt, läßt sich nicht generell beantworten. Es hängt von der Phasenbeziehung zwischen $\tilde{f}_{1-s}$ und im wesentlichen $\tilde{f}_{1-s\pm N}$ ab. Gerade in dem hier betrachteten Fall kann man durch Variation des Parameters τ/T in (1.18) sowohl den einen als auch den anderen Fall realisieren. Bild 1.5 zeigt für $N = 64$ und $\tau/T = 1/10$ den Betrag von $\tilde{f}_{1-s}$ als durchgezogene Linie und die Beträge der $b_s/\sqrt{N}$ als Punkte.

Die Rücktransformation von den b_s-Koeffizienten zu den ursprünglichen Werten $a_r = f(t_r)$ erfolgt mit Gleichung (1.9) und $r = N\left(\frac{t_r}{T} + \frac{1}{2}\right)$. Diese Gleichung gibt die exakten Werte des Rechteckimpulses an den diskreten Stellen t_r. Sie kann aber



1.5 Kontinuierliche und diskrete Fouriertransformation eines Rechteckimpulses. Die durchgezogene Linie ist $|\tilde{f}_{1-s}|$ als Funktion von s, die Punkte sind die Werte von $|b_s|N^{-\frac{1}{2}}$.

auch als Näherung für die gesamte Funktion $f(t)$ aufgefaßt werden, wenn t_r durch kontinuierliche t-Werte ersetzt wird. Bild 1.6 zeigt, daß dies kein gutes Ergebnis liefert. Der Grund für die schlechte Qualität der Näherung ist der stark oszillatorische Beitrag der hohen Frequenzen für s zwischen $N/2$ und N. Im mathematischen Teil (1.11) wurde gezeigt, daß man die Summe über jedes beliebige Intervall der



1.6 Links die ungünstige und rechts die bessere Fortsetzung der diskreten Fourier-Rücktransformation auf kontinuierliche t-Werte.

Länge N laufen lassen kann. Diese Freiheit nutzen wir, indem wir das Intervall symmetrisch zum Ursprung legen und nur die kleinsten Frequenzen verwenden:

$$
\begin{aligned}
f(t_r) &= \frac{1}{\sqrt{N}} \sum_{s=-N/2+1}^{N/2} b_s\, e^{-2\pi i(r-1)(s-1)/N} \\
&= \frac{1}{\sqrt{N}} \sum_{s=1}^{N/2} b_s\, e^{-2\pi i(r-1)(s-1)/N} +
\end{aligned}
$$

$$
\begin{aligned}
&+ \frac{1}{\sqrt{N}} \sum_{s'=1}^{N/2} b_{s'-N/2}\, e^{-2\pi i(r-1)(s'-N/2-1)/N} \\
&= \frac{1}{\sqrt{N}} \sum_{s=1}^{N/2} b_s\, e^{-2\pi i(r-1)(s-1)/N} \\
&+ \frac{1}{\sqrt{N}} \sum_{s'=1}^{N/2} b^*_{N/2-s'+2}\, e^{-2\pi i(r-1)(s'-N/2-1)/N} \,.
\end{aligned}
$$

Wird diese Version auf kontinuierliche t-Werte erweitert, so sieht man in Bild 1.6, daß die Stufe wesentlich besser approximiert wird. Allerdings können die Sprungstellen des Rechteckimpulses $f(t)$ nur durch starke Oszillationen auf beiden Seiten dargestellt werden.

Übung

1. **Spitzenspannung**

   Eine zeitabhängige Spannung wird an $N = 64$ diskreten Zeitpunkten abgetastet und liefert die Werte $U_1, \ldots, U_N$. Die Fouriertransformierte davon sei die Folge $\{b_1, \ldots, b_N\}$ mit dem Leistungsspektrum $|b_s|^2 = |b_{N-s+2}|^2 = 1$ für $s = 9, 10, \ldots, 17$ und $b_s = 0$ sonst. Damit das Spannungssignal $\{U_1, \ldots, U_N\}$ reell ist, muß $b_{N-s+2} = b_s^*$ gelten.

   a) Wie sieht das Spannungssignal $\{U_1, \ldots, U_N\}$ aus, wenn die Fourierkoeffizienten b_s konstante Phasen haben, z. B. $b_s = 1,$ falls $b_s \neq 0$?
   b) Wählen Sie für die von Null verschiedenen b_s zufällige Phasen, also $b_s = e^{i\varphi_s}$ mit zufälligen φ_s. Wie sieht das Signal jetzt aus?
   c) **Wettbewerb:** Gesucht sind die Phasen φ_s möglichst geringer Spitzenspannung, also $\min\limits_{\{\varphi_s\}} \max\limits_r |U_r|$.

2. **Aperiodischer Kristall**

   Gegeben sei eine Kette von Atomen mit periodischer Anordnung

   $$1\ 1\ 0\ 0\ 1\ 1\ 0\ 0\ 1\ 1\ 0\ 0\ 1\ 1\ 0\ 0.$$

   Mit Röntgenstreuung erhält man den Betrag der Fourierkoeffizienten $|b_s|$ der Funktion a_r mit

$$a_r = \begin{cases} 1 & \text{Atom 1 am Ort } r \\ 0 & \text{sonst} \end{cases}$$

a) Zeichnen Sie das Fourierspektrum b_s.

b) Erzeugen Sie nun eine zufällige Folge von 1 und 0 und zeichnen Sie deren Fourierspektrum.

c) Erzeugen Sie mit folgendem Algorithmus einen aperiodischen Kristall und vergleichen Sie sein Fourierspektrum mit den beiden anderen Kristallen.

   1. Starten Sie mit: 0
1
10
   2. Erzeugen Sie die nächste Zeile, indem Sie die vorletzte an die letzte anhängen.
   3. Iterieren Sie dies bis zur gewünschten Länge.

   Die Zahl der Atome in der n-ten Zeile ist die Fibonacci-Zahl F_n.

Literatur

R. E. Crandall, *Mathematica for the Sciences*, Addison Wesley, 1991.

W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge University Press, 1992.

Paul L. DeVries, *Computerphysik: Grundlagen, Methoden, Übungen*, Spektrum Akademischer Verlag, 1995.

Stephen Wolfram, *Mathematica: A System for Doing Mathematics by Computer*, Addison Wesley, 1991.

1.4 Daten glätten

Experimentelle Daten sind meistens mit einem deutlichen statistischen Fehler behaftet. Die Streuung der Meßwerte kann man bis zu einem gewissen Grad beseitigen, indem man über benachbarte Daten mittelt. Besonders bei der graphischen Aufbereitung von Daten ist eine solche Glättung nützlich. Hier wollen wir das Verfahren an Daten mit einer künstlich erzeugten Streuung demonstrieren. Diese Daten werden geglättet, indem an jedem Punkt die Nachbarpunkte je nach Abstand gewichtet und aufsummiert werden. Damit dabei keine neuen Strukturen entstehen,

wird als Gewicht eine „Gaußglocke“ verwendet. Die Daten werden also mit einer Gaußfunktion gefaltet, eine Glättungsoperation, bei der die Fouriertransformation aus dem vorigen Abschnitt verwendet werden kann.

Physik

Es sei $f(t)$ eine physikalische Größe, die an diskreten Zeitpunkten t_i gemessen wird. Die exakten Werte $f_i = f(t_i)$ sollen mit Zufallszahlen r_i überlagert sein, so daß die Meßdaten durch

$$g_i = f_i + r_i\,,\ i = 1, 2, 3, ..., N\,, \tag{1.24}$$

gegeben sind. Zu jedem i wollen wir neue Werte $\overline{g}_i$ berechnen, die den Mittelwert über eine Nachbarschaft von i bilden. Die Folge der $\overline{g}_i$ ist dann viel glatter als diejenige der g_i-Werte und verdeutlicht besser die ursprüngliche Funktion $f(t)$.

Zur Berechnung von $\overline{g}_i$ könnte man nun einfach die g_j-Werte in einem Intervall um i aufsummieren. Aber es gibt eine bessere Methode, die keine zusätzlichen unerwünschten Strukturen erzeugt. Dabei wird ein gewichtetes Mittel benutzt, das Nachbarn entsprechend ihrem Abstand $|j - i|$ vom Platz i berücksichtigt. Weit entfernte Nachbarn werden nur schwach gewichtet. Als Gewichtsfunktion, auch Kern genannt, wird eine „Gaußglocke“ verwendet. Sei also k_j eine solche Gewichtsfunktion, die wie vorher auf die Werte $j = 1, .., N$ beschränkt wird. Die k_j-Werte für $j < 1$ werden durch die Symmetrie (1.11) ausgedrückt. Der Kern muß natürlich normiert sein:

$$\sum_{j=1}^{N} k_j = 1\,. \tag{1.25}$$

Dann wird $\overline{g}_r$ folgendermaßen konstruiert:

$$\overline{g}_r = \sum_{j=1}^{N} g_{r-j+1}\, k_j\,; \tag{1.26}$$

$\overline{g}$ ist also die diskrete Faltung der Funktionen g und k, die schon in Abschnitt 1.3 besprochen wurde. Daher läßt sie sich einfach als Fouriertransformation darstellen,

$$\overline{g}_r = \sum_{s=1}^{N} \tilde{g}_s\, \tilde{k}_s\, e^{-2\pi i(s-1)(r-1)/N}\,, \tag{1.27}$$

wobei $\tilde{g}$ und $\tilde{k}$ die Fouriertransformierten von g und k sind.

Algorithmus

Die Daten, die wir glätten wollen, erzeugen wir mit *Mathematica* als Liste `data`, und zwar als Werte der Besselfunktion $J_1(x)$, die durch Zufallszahlen verrauscht wurden:

```
data=
Table[N[BesselJ[1,x]+0.2(Random[]-1/2)],{x,0,10,10/255}]
```

Für den Kern wählen wir eine Gaußkurve der Breite σ.

```
kern = Table[N[Exp[-x^2/(2*sigma^2)]], {x,-5,5,10/255}]
```

Allerdings stehen jetzt die größten Funktionswerte in der Mitte der Liste `kern`. Wir müssen deshalb diese Liste zyklisch rotieren, so daß das Maximum an den Beginn der Liste, also nach `kern[[1]]` verschoben wird. Dies geschieht mit dem Befehl

```
kern = RotateLeft[kern,127]
```

Nach (1.25) muß dieser Kern noch normiert werden:

```
kern = kern/Apply[Plus, kern]
```

Die geglätteten Daten (1.27) erhält man dann mit (1.14) ganz einfach durch den Ausdruck

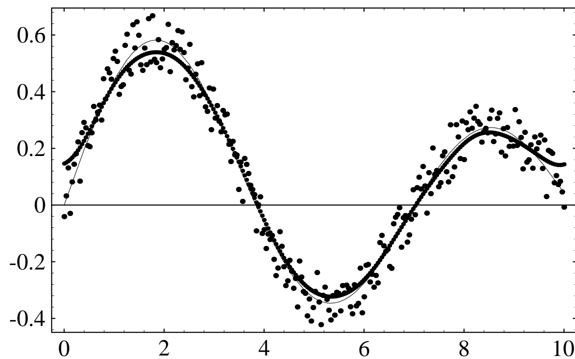
```
Sqrt[256] InverseFourier[Fourier[data] Fourier[kern]]
```

Das Produkt zweier Listen gibt dabei die Liste der Produkte der Elemente, ganz wie Gleichung (1.27) verlangt.

Ergebnisse

Bild 1.7 zeigt das Ergebnis der Rechnungen für $\sigma = 0.4$. Die geglätteten Daten werden mit den verrauschten Daten und mit der ursprünglichen Funktion verglichen. Die Kurve $\overline{g}_i$ ist tatsächlich glatt und hat eine etwas kleinere Amplitude als die Originalkurve. Das muß natürlich so sein, da die geglättete Kurve über die Nachbarschaft mittelt und damit immer kleiner als die Maximalwerte der Daten ist. Insbesondere die bei Anwendung der Fouriertransformation stillschweigend vorausgesetzte Periodizität der Daten führt hier zu einer Verfälschung an den Rändern.

Wir möchten erwähnen, daß genau dieses Beispiel im Abschnitt 3.8.3 des *Mathematica*-Handbuches vorgeführt wird. Allerdings werden dort gleich mehrere Fehler gemacht: Der Kern wird nicht normiert, der Faktor $\sqrt{N}$ wird bei der Faltung vergessen und als Kern wird nur die rechte Hälfte der „Gaußglocke“ genommen. Während



1.7 Ausgangsfunktion (dünne Linie) und verrauschte Daten (Punkte). Die Punkte der geglätteten Daten erscheinen als fette Linie.

sich die ersten beiden Fehler bei den gewählten Parametern gerade aufheben, verschiebt der dritte die geglättete Kurve etwas nach rechts zu größeren x-Werten hin. Abschließend wollen wir noch darauf hinweisen, daß man bei der Mittelung vorsichtig sein muß. Wählt man die Breite des Kerns zu klein, dann erhält man offenbar die ursprünglichen Daten ohne Glättung. Läßt man dagegen eine zu breite Gewichtsfunktion zu, dann wird nicht nur die Streuung weggemittelt, sondern es werden auch kurzwellige Strukturen in der ungestörten Funktion zerstört. Im Extremfall eines unendlich breiten Kerns besteht die geglättete Funktion nur aus einer einzigen Konstanten. Die Breite der Gewichtsfunktion muß daher sorgfältig an die zu glättenden Daten angepaßt werden.

Falls die Statistik der Streuung bekannt ist, so gibt es theoretische Ansätze, mit denen man den optimalen Kern berechnen kann. Alternativ kann man auch eine möglichst glatte Kurve durch die Daten legen, die den korrekten Wert von χ^2 hat (siehe nächsten Abschnitt). Zum Beispiel kann man durch benachbarte Stützpunkte Polynome legen und minimale Krümmungen fordern. Dieses umfangreiche Gebiet der statistischen Analyse von Daten können wir hier nicht behandeln, sondern wir müssen auf die entsprechenden Lehrbücher verweisen.

Übung

Das Ergebnis des oben vorgestellten Algorithmus zum Datenglätten hängt von der gewählten Breite σ des Kerns ab. Berechnen sie die geglätteten Daten für einen großen Bereich von σ-Werten. Wenn die Breite des Kerns kleiner als der Abstand der Datenpunkte ist, erhält man fast die ursprünglichen Daten. Wenn dagegen die Breite größer als das ganze x-Intervall ist, sind die geglätteten Punkte fast konstant. Bestimmen sie einen σ-Wert, der Ihnen geeignet erscheint.

Literatur

W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge University Press, 1992.

Stephen Wolfram, *Mathematica: A System for Doing Mathematics by Computer*, Addison Wesley, 1991.

1.5 Nichtlinearer Fit

Ein Ziel der Naturwissenschaft ist es, mathematische Gesetzmäßigkeiten in gemessenen Daten zu finden. Deshalb werden häufig Modellfunktionen mit freien Parametern an die Daten angepaßt. Die Modelle entstehen – hoffentlich – aus Theorien, und der Computer wird benötigt, um die „besten" Parameter zu finden und ein Maß für die Güte des Fits anzugeben. Dem Wissenschaftler stehen dazu umfangreiche Werkzeuge aus der mathematischen Statistik zur Verfügung mit ebenso umfangreichen Programmpaketen.

Gehen die Parameter linear in die Modelle ein, so ist die mathematische Theorie dazu besonders gut entwickelt. Allerdings sind die Modelle der Physiker oft komplexer, so daß häufig ein nichtlinearer Fit benötigt wird. Hier soll an einem einfachen Beispiel das Prinzip der nichtlinearen Parametersuche erläutert werden. Wir werden dabei die Qualität eines Fits mit dem χ^2-Test abschätzen, wobei wir selbst ein Maß für unser Vertrauen setzen müssen. Niemals werden wir beweisen können, daß unser Modell die Wirklichkeit wiedergibt; aber wir werden damit falsche Modelle mit großer Wahrscheinlichkeit ausschließen können. Für ein akzeptiertes Modell können wir auch ein Maß für die Genauigkeit der verwendeten Parameter angeben.

Am Beispiel einer gedämpften Schwingung, die an nur elf Zeitpunkten gemessen wird und deren Daten stark verrauscht werden, passen wir ein Modell mit vier Parametern an die Daten an und bestimmen deren Fehler.

Theorie

Es seien N Datenpaare $\{Y_i, t_i\}$ gegeben, an die eine Modellfunktion $g(\boldsymbol{a}, t_i)$ mit unbekanntem Parametervektor $\boldsymbol{a}$ mit M Komponenten angepaßt werden soll. Dabei soll bekannt sein, daß zu jedem t_i die Daten Y_i mit einem Fehler ε_i gemessen werden, der im Mittel über viele Experimente verschwindet und eine Varianz σ_i^2 hat, also $\langle \varepsilon_i \rangle = 0$ und $\langle \varepsilon_i^2 \rangle = \sigma_i^2$. Als „besten" Parametersatz $\boldsymbol{a}$ definiert man diejenigen Werte $\boldsymbol{a}_0$, die die quadratische Abweichung χ^2 minimieren, wobei χ^2

wie folgt definiert ist:

$$\chi^2(\boldsymbol{a}) = \sum_{i=1}^{N} \left(\frac{Y_i - g(\boldsymbol{a}, t_i)}{\sigma_i} \right)^2 . \tag{1.28}$$

Wenn $g(\boldsymbol{a}_0, t_i)$ ein angemessenes Modell ist und wenn die Fehler $\varepsilon_i = Y_i - g(\boldsymbol{a}_0, t_i)$ unkorreliert und gaußverteilt sind, dann ist die Verteilung von $\chi^2(\boldsymbol{a}_0)$ bekannt. Wie man durch häufige Wiederholung des Experiments überprüfen könnte, ist die Wahrscheinlichkeit dafür, daß χ^2 kleiner ist als χ_0^2, durch

$$P_{N-M}(\chi_0^2) = \frac{1}{\Gamma\left(\frac{N-M}{2}\right)} \int\limits_0^{\chi_0^2/2} e^{-t}\, t^{\frac{N-M}{2}-1}\, dt \tag{1.29}$$

gegeben. Das Integral ist in der mathematischen Literatur als unvollständige Gammafunktion $\gamma((N-M)/2\,,\,\chi_0^2/2)$ bekannt. $\chi^2(\boldsymbol{a}_0)$ hat nur $N-M$ unabhängige Variable, da M Freiheitsgrade durch die Minimumbedingung fixiert sind. Für große Werte von $N-M$ gilt der zentrale Grenzwertsatz: χ^2 ist gaußverteilt mit Mittelwert $\langle\chi^2\rangle = N-M$ und Varianz $2(N-M)$.

Was bedeutet nun die χ^2-Verteilung (1.29) für die Genauigkeit des Fits? Nehmen wir an, unser Experiment gibt den Wert $\chi_0^2 = \chi^2(\boldsymbol{a}_0)$ für das Minimum von Gleichung (1.28). Ferner sei der Wert $P(\chi_0^2) = 0.99$. Das heißt, wenn das Modell richtig ist, würde in 99% aller Experimente der Wert χ^2 kleiner als χ_0^2 sein. Das könnten wir noch als gültigen Fit akzeptieren, denn wir können nicht ausschließen, daß unser Experiment zu den 1% der Fälle gehört, bei denen χ^2 größer als χ_0^2 ist. Wäre aber $P = 0.9999$, dann wäre unser Fit nur dann richtig, wenn er zu den 0.01% der Experimente mit $\chi^2 > \chi_0^2$ gehören würde. In diesem Fall würde man sicher schließen, daß χ_0^2 viel zu groß ist und daß deshalb unsere Voraussetzungen, insbesondere unser Modell, nicht richtig sind. Andererseits darf der Wert von χ_0^2 nicht zu klein sein, denn unsere Daten haben einen statistischen Fehler. Die Wahrscheinlichkeit, daß ein Experiment einen kleineren Wert als unser χ_0^2 hat, ist wieder durch die Verteilung (1.29) gegeben. Das Intervall für den Wert von χ_0^2, den wir akzeptieren wollen, nennt man *Vertrauensintervall*. Wo wir die Schranke setzen, bei 1% oder bei 0.01%, hängt von uns selbst und unserer Erfahrung mit ähnlichen Problemen ab.

Nehmen wir nun an, daß wir mit der Qualität unseres Fits zufrieden sind, d.h. $\chi^2(\boldsymbol{a})$ hat ein Minimum bei den Parametern $\boldsymbol{a}_0$, und der Wert von $\chi^2(\boldsymbol{a}_0)$ liegt in unserem Vertrauensintervall. Könnten wir das Experiment mehrmals wiederholen, so würden wir selbstverständlich andere Meßfehler ε_i und damit einen anderen Parametervektor $\boldsymbol{a}_0$ erhalten. Wir machen aber nur ein Experiment und wollen dennoch aus den Daten den Fehler von $\boldsymbol{a}_0$ abschätzen.

Hierbei hilft uns wieder die χ^2-Verteilung (1.29), denn Konturen im M-dimensionalen $\boldsymbol{a}$-Raum mit konstantem Wert von $\chi^2(\boldsymbol{a})$ sind ein Maß für den Fehler von $\boldsymbol{a}_0$. Bei kleinen Abweichungen $|\boldsymbol{a} - \boldsymbol{a}_0|$ sind diese Konturen Ellipsoide mit M Hauptachsen, deren Länge ein Maß dafür ist, wie weit man $\boldsymbol{a}_0$ in dieser Richtung ändern kann, bis der Fit nicht mehr akzeptabel ist.

Diese Aussage kann man sogar quantifizieren. Dazu erzeugen wir uns künstliche Daten Y_i, indem wir zum Modell $g(\boldsymbol{a}_0, t_i)$ einen gaußverteilten Fehler ε_i mit $\langle\varepsilon_i^2\rangle = \sigma_i^2$ addieren: $Y_i = g(\boldsymbol{a}_0, t_i) + \varepsilon_i$. Mit diesen Daten lassen wir wiederum die Fitprozedur laufen, suchen also mit den simulierten Daten erneut ein Minimum $\boldsymbol{a}_1$ von $\chi^2(\boldsymbol{a})$. Mehrmals wiederholt liefert dies die Parametervektoren $\boldsymbol{a}_1, \boldsymbol{a}_2, \ldots$, und aus der Breite der Verteilung der Komponenten der $\boldsymbol{a}_k$ gewinnen wir Fehlerbalken für die Fitparameter $\boldsymbol{a}_0$.

Falls die Abweichung $|\boldsymbol{a} - \boldsymbol{a}_0|$ so klein ist, daß man die Entwicklung von $\chi^2(\boldsymbol{a})$ (aus den experimentellen Daten) um $\boldsymbol{a}_0$ nach den quadratischen Gliedern abbrechen kann, so läßt sich zeigen, daß die Größe $\Delta = \chi^2(\boldsymbol{a}) - \chi^2(\boldsymbol{a}_0)$ wieder mit der Verteilungsfunktion P aus (1.29) verteilt ist, in diesem Fall mit M statt mit $N - M$ Freiheitsgraden. Fordern wir daher wie vorher, daß unser Fit zu den 99% aller möglichen Fits beim korrekten Modell gehören soll, dann bestimmt die Ungleichung

$$P_M(\chi^2(\boldsymbol{a}) - \chi^2(\boldsymbol{a}_0)) \leq 0.99 \tag{1.30}$$

ein Gebiet mit erlaubten Werten von $\boldsymbol{a}$. Im Parameterraum sind die Gebiete mit konstantem Δ Ellipsoide. Die Projektion dieser $(M-1)$-dimensionalen Fläche auf die Achse i gibt dann das Fehlerintervall für den Parameter a_i.

Algorithmus

Auch beim nichtlinearen Fit ist es am einfachsten, die vorhandenen *Mathematica*-Funktionen zu nutzen. Zum Auffinden des Minimums von $\chi^2(\boldsymbol{a})$ verwenden wir die Funktion `NonlinearFit` im Paket `Statistics'NonlinearFit'`. Es gibt dort verschiedene Möglichkeiten, Daten und Startbedingungen einzugeben, außerdem kann man die Methode der Minimumsuche ändern und sich die Zwischenergebnisse anzeigen lassen. Natürlich kann man auch $\chi^2(\boldsymbol{a})$ selbst definieren und mit `FindMinimum` den Wert $\boldsymbol{a}_0$ finden.

Zur Erzeugung der Daten und als Modell wählen wir eine gedämpfte Sinusschwingung

```
f[t_]:= a Sin[om t + phi] Exp[-b t]
```

mit 4 Parametern $\boldsymbol{a}$ = {`a, om, phi, b`}. Diese Schwingung wird für den Parametersatz $\boldsymbol{a}$ = {`1,1,0,0.1`} an 11 Zeitpunkten t_i gemessen, und die Daten werden durch gleichverteilte Zufallszahlen verrauscht:

```
daten =
Table[{t,Sin[t]Exp[-t/10.]+0.4*Random[]-0.2}//N,
                                        {t,0,3Pi,0.3Pi}]
```

Für die (nicht gaußverteilten) Fehler ε_i gilt

$$\langle \varepsilon_i \rangle = 0, \;\; \sigma_i^2 = \langle \varepsilon_i^2 \rangle = \frac{1}{0.4} \int\limits_{-0.2}^{0.2} x^2 \, dx = \frac{2}{150} \, .$$

Die Suche nach dem Minimum wird erleichtert, wenn wir einen ungefähren Wert von $\boldsymbol{a}_0$ angeben können. Nach dem Laden der benötigten Statistik-Pakete mit

```
Needs["Statistics'Master'"]
```

rufen wir die Minimumsuche auf:

```
NonlinearFit[daten,f[t],t,
             {{a,1.1},{om,1.1},{phi,.1},{b,.2}},
             ShowProgress->True]
```

Auch die χ^2-Verteilung ist in *Mathematica* vorhanden, und zwar im Paket `Statistics'ContinuousDistributions'`. Sie hat den selbsterklärenden Namen `ChiSquareDistribution[..]`. Als Argument ist die Zahl der Freiheitsgrade einzusetzen, die hier mit $N = 11$ Daten und $M = 4$ Parametern durch $N - M = 7$ bestimmt ist. Mit `PDF` erhält man die Dichte der Verteilung, also den Integranden von $P_7(\chi_0^2)$, Gleichung (1.29), während `CDF` das Integral, also $P_7(\chi_0^2)$ selbst, ergibt. Mit `Quantile` wird die Funktion $P_7(\chi_0^2)$ invertiert.

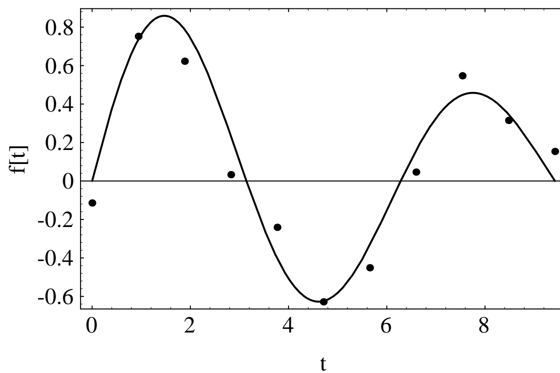
```
Quantile[ChiSquareDistribution[7], 0.95]
```

gibt also dasjenige χ_0^2, für das in 95% aller Experimente der Wert von χ^2 kleiner als χ_0^2 ist. Fixiert man zwei Parameter, so kann man für die übrigen zwei die Fläche in der Parameterebene, für die $\chi^2(\boldsymbol{a}) = \chi^2(\boldsymbol{a}_0) + \Delta$ ist, durch `ContourPlot` darstellen.

Ergebnisse

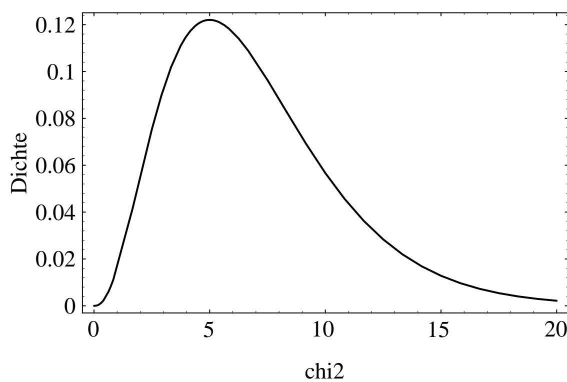
Bild 1.8 zeigt die Funktion `f[t]` für `a=1`, `om=1`, `phi=0` und `b=0.1` und die 11 Daten, die aus `f[t]` mit dem Zufallsfehler ε_i gewonnen wurden. Nach dem Start mit `a=1.1, om=1.1, phi=0.1` und `b=0.2` findet `NonlinearFit` das Minimum $\boldsymbol{a}_0$ von $\chi^2(\boldsymbol{a})$ in etwa fünf Schritten. Das Ergebnis wird von *Mathematica* in Form einer Regel ausgegeben.

```
{a -> 0.825, om -> 0.976, phi -> 0.024, b -> 0.069}   (1.31)
```



1.8 Gedämpfte Schwingung und verrauschte Meßdaten.

Die Dichte der χ^2-Verteilung ist im Bild 1.9 gezeigt. Die Verteilung (1.29) gilt zwar streng genommen nur für gaußverteilte Fehler ε_i, aber wir erwarten keinen großen Unterschied für unsere gleichverteilten Streuungen ε_i. Für den richtigen



1.9 Dichte der χ^2-Verteilung mit 7 Freiheitsgraden.

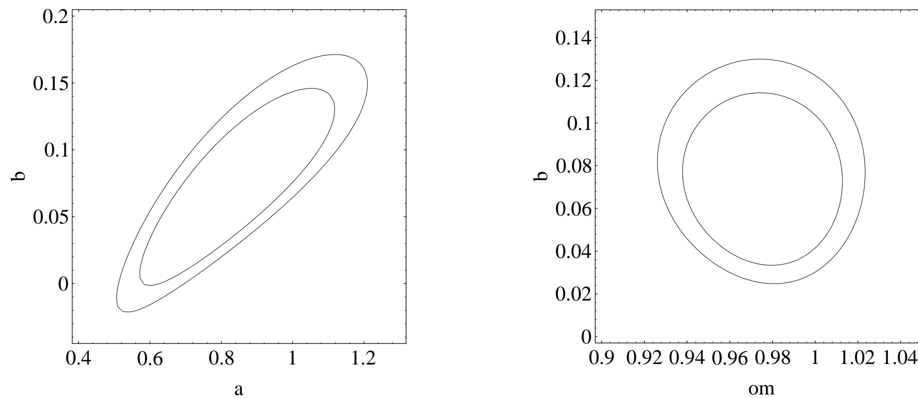
Parametersatz gibt diese Dichte die Verteilung von χ_0^2 für verschiedene Experimente, d. h. für verschiedene Realisierungen der ε_i.

Das Vertrauensintervall für den Wert von χ_0^2 erhalten wir mit

```
grenze[x_] = Quantile[ChiSquareDistribution[7],x]
```

und das Ergebnis von $\{$`grenze[0.05]`, `grenze[0.95]`$\}$ ist $\{2.2, 14.1\}$. Das heißt, für sehr viele Experimente sollten (mit den korrekten Parametern) 5% davon einen Wert $\chi_0^2 \geq 14.1$ und 5% einen Wert $\chi_0^2 \leq 2.2$ haben. In unserem Fall erhalten wir mit $\chi_0^2 = 9.4$ einen Wert weit im Innern des Vertrauensintervalls; wir haben daher keinen Grund, an dem Ergebnis des Fits zu zweifeln.

Konturen mit $P_4(\chi^2(\boldsymbol{a}) - \chi_0^2) = 0.68$ und $P_4(\chi^2(\boldsymbol{a}) - \chi_0^2) = 0.90$ sind in der Abbildung 1.10 zu sehen. Es sind dreidimensionale Flächen im vierdimensionalen Parameterraum, deshalb können wir nur Schnitte zeichnen. Der optimale Parametersatz liegt natürlich genau im Zentrum aller Konturflächen. Der `(a,b)`-Schnitt

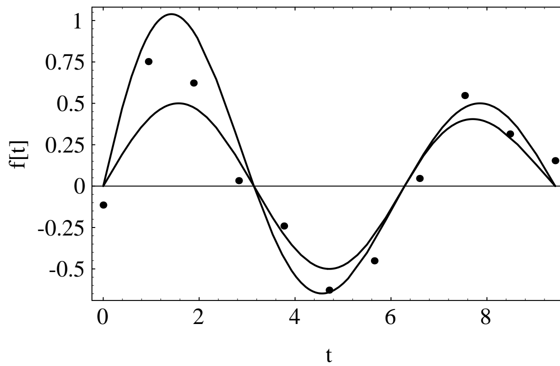


1.10 Konturen mit konstantem $\chi^2(\boldsymbol{a})$. Links: Schnitt mit der `a`-`b`-Parameter-Ebene. Rechts: Schnitt mit der `om`-`b`-Ebene.

zeigt, daß Änderungen in der Amplitude `a` durch eine Änderung der Zeitkonstanten `b`$^{-1}$ der Dämpfung kompensiert werden können, ohne die Qualität des Fits zu verschlechtern. Daher darf man als Fehler für `a` nicht den Schnitt beim optimalen `b` angeben, sondern man muß die Projektion auf die `a`-Achse verwenden. Insbesondere zeigt Bild 1.10 (links), daß es nicht möglich ist, eine obere Schranke für die Zeitkonstante `b`$^{-1}$ zu bestimmen, da die Kontur der äußeren Vertrauensfläche durch den Punkt `b=0` (also `b`$^{-1} = \infty$) geht.

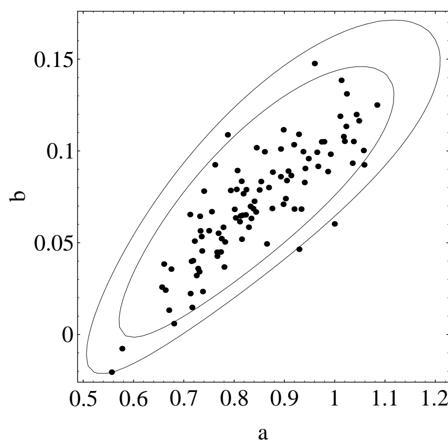
Variiert man die Frequenz `om` anstatt der Amplitude `a`, so tritt die obige Kompensation nicht auf, wie Bild 1.10 (rechts) zeigt. Der wahre Punkt `om=1` und `b=0.1` liegt dicht am Rand der inneren Vertrauensfläche; das Vertrauensintervall sollte daher nicht zu eng gewählt werden. Zeichnet man `f[t]` für zwei Extrema aus Bild 1.10, nämlich `a=1.3`, `b=0.15` und `a=0.5`, `b=0`, so sieht man in Bild 1.11, daß beide Kurven die Daten noch relativ gut wiedergeben. Die wenigen Daten $(N = 11)$ und der große Fehler erlauben keinen besseren Fit.

Wie im Theorie-Teil erläutert wurde, gibt es eine alternative Methode, um den Fehler für die Parameter abzuschätzen. Man nimmt für ein Experiment den optimalen Parametersatz $\boldsymbol{a}_0$ und erzeugt sich damit neue, künstliche Daten, die wiederum gefittet werden. Wird dies mit demselben $\boldsymbol{a}_0$ wiederholt, so erhält man einen Satz von Parametervektoren $\boldsymbol{a}_i$. Bild 1.12 zeigt das Ergebnis für 100 Iterationen mit dem $\boldsymbol{a}_0$ aus Gleichung (1.31) zusammen mit den Konturlinien von Bild 1.10. Man sieht, daß die Werte in der (`a`, `b`)-Projektion sogar zu mehr als 90% innerhalb des äußeren



1.11 Zwei Fitkurven mit Parametern aus der äußeren Kontur von Bild 1.10 (links).

Konturschnitts liegen. Um die Fehlergrenzen von $\boldsymbol{a}_0$ angeben zu können, berechnen wir mit dem ursprünglichen Datensatz $\{Y_i\}$ zu jedem der 100 Parametervektoren $\boldsymbol{a}_i$ das zugehörige $\chi^2(\boldsymbol{a}_i)$ und sortieren die $\boldsymbol{a}_i$ nach wachsendem χ^2. Für ein Vertrau-



1.12 Parameterwerte `a` und `b` aus den Fits zu den künstlichen Daten zusammen mit den Konturlinien von Bild 1.10 (links).

ensniveau von z. B. 68% betrachten wir dann nur noch die ersten 68 dieser Vektoren, von denen der letzte eine χ^2-Grenze $\chi^2_{68} = \chi^2(\boldsymbol{a}_{68})$ festlegt. Alle noch verbleibenden $\boldsymbol{a}_i$ liegen dann innerhalb des Quasi-Ellipsoids $\chi^2(\boldsymbol{a}) \leq \chi^2_{68}$. Wir projizieren die Punkte $\boldsymbol{a}_i$ auf die Achsen des Parameterraumes und erhalten so die äußeren Abmessungen des Ellipsoids. Auf diese Weise können wir schließlich die Aussage treffen, daß mit einer Sicherheit von 68% der wahre Vektor $\boldsymbol{a}_{\text{true}}$ in einem Quasi-Ellipsoid liegt, das in dem Quader

$$\texttt{a} = 0.83 \pm 0.26\,, \quad \texttt{b} = 0.07 \pm 0.06\,, \quad \texttt{om} = 0.98 \pm 0.05\,, \quad \texttt{phi} = 0 \pm 0.2$$

enthalten ist. Natürlich gehört wegen der erwähnten Korrelationen nicht jeder Punkt $\boldsymbol{a}$ des Quaders zu dem 68%-Bereich. Ob er dazugehört, läßt sich an seinem $\chi^2(\boldsymbol{a})$ feststellen.

Übung

Die Daten $\{x_i, f(x_i)\}$ im File `twinpeak.dat` sind das Ergebnis der Funktion

$$f(x_i) = e^{-x_i^2/2} + a\, e^{-(x_i-b)^2/(2\sigma^2)} + r_i\,,$$

wobei die r_i normalverteilte Zufallszahlen mit Mittelwert 0 und Breite $\sigma_0 = 0.05$ sind. Die Daten können Sie mit

```
daten = << twinpeak.dat
```

während einer *Mathematica*-Sitzung einlesen und mit `ListPlot[daten]` ansehen. Finden Sie die Amplitude a, den Ort b und die Breite σ des Zusatzpeaks und geben Sie die Fehler der drei gefitteten Parameter an.

Literatur

S. Brandt, *Datenanalyse: Mit statistischen Methoden und Computerprogrammen*, BI Wissenschaftsverlag, 1992.

J. Honerkamp, *Stochastische Dynamische Systeme: Konzepte, numerische Methoden, Datenanalysen*, VCH Verlagsgesellschaft, 1990.

W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge University Press, 1992.

Stephen Wolfram, *Mathematica: A System for Doing Mathematics by Computer*, Addison Wesley, 1991.

1.6 Multipol-Entwicklung

Ein mächtiges analytisches Werkzeug des Theoretikers ist die Entwicklung einer physikalischen Gleichung nach einer kleinen Größe. Dabei entstehen manchmal Ausdrücke, die kompliziert und unanschaulich sind. Oft hat man kein Gefühl dafür, wie stark die Entwicklung vom wahren Wert abweicht.

Die Multipol-Entwicklung eines elektrostatischen Potentials, ein beliebtes Kapitel aus der Elektrodynamik-Vorlesung, ist ein einfaches Beispiel dafür. Von ferne sieht eine Ladungsverteilung wie eine Punktladung aus. Kommt der Beobachter

näher, so bemerkt er das Dipolmoment, bei größerer Annäherung auch das Quadrupolmoment. Diese approximative Beschreibung läßt sich kompakt durch einen Skalar, einen Vektor und einen Tensor mathematisch formulieren. Jetzt – mit *Mathematica* – können wir dies ebenso kompakt programmieren, aber auch graphisch darstellen und uns die Abweichung vom exakten Potential ansehen.

Physik

Für N punktförmige Ladungen e_i an den Orten $\boldsymbol{r}^{(i)}$ ist das elektrostatische Potential $\Phi(\boldsymbol{r})$ gegeben durch

$$\Phi(\boldsymbol{r}) = \sum_{i=1}^{N} \frac{e_i}{|\boldsymbol{r} - \boldsymbol{r}^{(i)}|} . \tag{1.32}$$

Das elektrische Feld ist durch den Gradienten von Φ bestimmt:

$$\boldsymbol{E} = -\boldsymbol{\nabla}\Phi(\boldsymbol{r}) \, , \boldsymbol{\nabla}\Phi = \left(\frac{\partial\Phi}{\partial x}, \frac{\partial\Phi}{\partial y}, \frac{\partial\Phi}{\partial z} \right) . \tag{1.33}$$

Betrachtet man nun dieses Potential von weiter Ferne, also für $|\boldsymbol{r} - \boldsymbol{r}^{(i)}| \to \infty$, so kann man $\Phi(\boldsymbol{r})$ entwickeln:

$$\Phi(\boldsymbol{r}) = \frac{q}{r} + \frac{\boldsymbol{p} \cdot \boldsymbol{r}}{r^3} + \frac{1}{2} \frac{1}{r^5} \, \boldsymbol{r} \, \widehat{Q} \, \boldsymbol{r} + \mathcal{O}\left(\frac{1}{r^4} \right) \tag{1.34}$$

mit $r = |\boldsymbol{r}|$. Dabei ist q die Gesamtladung, $\boldsymbol{p}$ das Dipolmoment und $\widehat{Q}$ der Quadrupoltensor:

$$q = \sum_{i=1}^{N} e_i \, , \quad \boldsymbol{p} = \sum_{i=1}^{N} e_i \, \boldsymbol{r}^{(i)} \, , \quad \widehat{Q}_{kl} = \sum_{i=1}^{N} e_i \left(3 r_k^{(i)} \, r_l^{(i)} - \delta_{kl} \, {\boldsymbol{r}^{(i)}}^2 \right) , \tag{1.35}$$

wobei $r_k^{(i)}$ die k-te Komponente von $\boldsymbol{r}^{(i)}$ ist.

Algorithmus

Wir wollen diese Entwicklung am Beispiel von fünf positiven und fünf negativen Einheitsladungen untersuchen, die wir in die x-y-Ebene legen, und zwar zufällig verteilt innerhalb eines Quadrats mit Mittelpunkt im Ursprung. Wir erzeugen uns also zehn Vektoren mit

```
rpunkt:={2 Random[]-1, 2 Random[]-1, 0}
Do[r[i] = rpunkt, {i,10}]
```

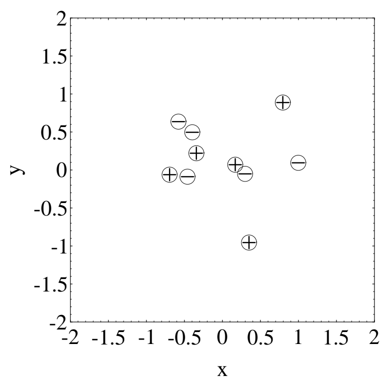
Jeder Vektor `r[i]` ist eine Liste von drei Zahlen. An den ersten fünf Punkten sollen sich positive, an den letzten fünf negative Ladungen befinden. Wir wollen nun die Ladungen graphisch darstellen. Dazu werden zunächst mit der Funktion `Line[...]` an den entsprechenden Stellen Plus- und Minussymbole gezeichnet, wobei man mit `Drop[r[i],-1]` die z-Koordinate beseitigt:

```
p1 = Graphics[Table[Line[{Drop[r[i],-1]-{0.08,0},
                Drop[r[i],-1]+{0.08,0}}],{i,5}]]
p2 = Graphics[Table[Line[{Drop[r[i],-1]+{0,0.08},
                Drop[r[i],-1]-{0,0.08}}],{i,5}]]
p3 = Graphics[Table[Line[{Drop[r[i+5],-1]-{0.08,0},
                Drop[r[i+5],-1]+{0.08,0}}],{i,5}]]
```

Dann wird mit `Circle[ ]` jeweils ein Kreis um die Symbole gezeichnet,

```
p4 = Graphics[{Thickness[0.001],
         Table[Circle[Drop[r[i], -1], 0.1],{i,10}]}]
```

und danach werden mit `Show[p1,p2,p3,p4,`*Optionen*`]` alle vier Graphikobjekte zusammen gezeichnet. Bild 1.13 zeigt das Ergebnis.



1.13 Positive und negative Ladungen zufällig verteilt in der x-y-Ebene.

Um das Potential angeben zu können, definieren wir zunächst den Abstand zwischen zwei Vektoren,

```
dist[r_,s_]=Sqrt[(r-s).(r-s)]
```

Der Punkt zwischen zwei Listen von Zahlen – die volle *Mathematica*-Form davon ist `Dot[l1,l2]` – bewirkt ein Skalarprodukt der beiden Vektoren, also die Summe der Produkte der jeweiligen Komponenten der Vektoren. Ohne den Punkt hingegen würden die Vektoren elementweise miteinander multipliziert werden mit einer Liste als Ergebnis. Nun kann man $\Phi(\boldsymbol{r})$ aus Gl. (1.32) direkt eingeben $(e_i = \pm 1)$:

```
pot[rh_]:=Sum[1/dist[rh,r[i]]-1/dist[rh,r[i+5]],{i,5}]
```

Wir präsentieren drei Möglichkeiten, wie man sich dieses Ergebnis veranschaulichen kann: Zunächst zeichnen wir mit `Plot3D` das Potentialgebirge über der x-y-Ebene, dann mit `ContourPlot` die Höhenlinien dieses Gebirges, und schließlich berechnen wir das elektrische Feld und zeichnen es mit `PlotVectorField` aus dem Paket `Graphics'PlotField'`. Allerdings sieht man im letzten Fall nur etwas, wenn man das Feld normiert, also nur die Richtung anzeigt.

Das Dipol- und Quadrupolmoment kann man ebenso einfach in *Mathematica* formulieren. Nach Gl. (1.35) gilt

```
dipol = Sum[r[i] - r[i+5],{i,5}]
quadrupol[r_]:=
    Table[3 r[[k]] r[[l]] - If[k==l,r.r,0],{k,3},{l,3}]
qsum = Sum[quadrupol[r[i]]-quadrupol[r[i+5]],{i,5}]
```

Dabei wurde das Delta-Symbol δ_{kl} durch folgenden Ausdruck beschrieben

```
If[k==l,1,0]
```

Den Betrag eines Vektors $\boldsymbol{r}$, den wir zwar als `dist[r,0]` ausdrücken könnten, berechnen wir mit der Funktion

```
betrag[r_]=Sqrt[r.r]
```

Damit läßt sich die Entwicklung (1.34) des Potentials $\Phi(\boldsymbol{r})$ direkt definieren. In unserem Beispiel gibt es keine Gesamtladung ($q = 0$), daher ist der erste Beitrag `pot1` der Dipolterm, und `pot2` enthält zusätzlich den Quadrupolbeitrag:

```
pot1[r_] = dipol.r / betrag[r]^3
pot2[r_] = pot1[r]+1/2/betrag[r]^5 r.qsum.r
```

`qsum` ist eine Liste von Listen, in diesem Fall also eine 3×3-Matrix. `r` ist eine Liste von Zahlen, also ein Vektor. Die Funktion `Dot[ , ]` oder kurz mit dem Punkt (**.**) bezeichnet, berechnet Kontraktionen von Tensoren mit beliebig vielen Indizes (= geschachtelten Listen). Hier läßt sich daher die quadratische Form „Vektor mal Matrix mal Vektor" sehr einfach als `r.qsum.r` schreiben. In einer herkömmlichen Programmiersprache wie C müßte man dagegen zwei `for`-Schleifen ineinanderschachteln:

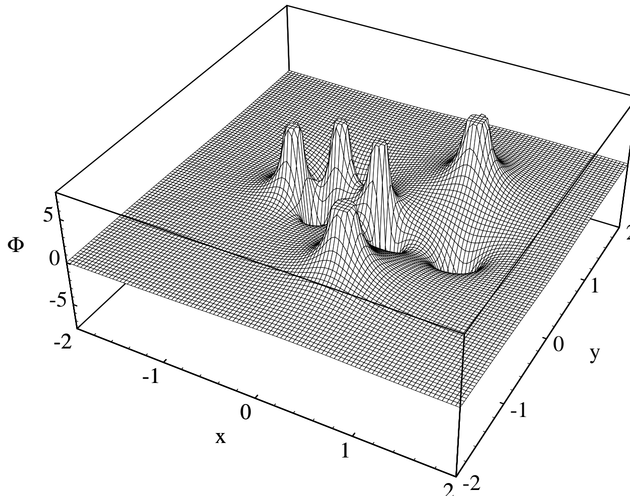
```
sum = 0;
for(i=0; i<3; i++) {
   for(j=0; j<3; j++) {
      sum = sum + r[i]*qsum[i][j]*r[j] }}
```

Das elektrische Feld $\boldsymbol{E}$ erhält man nach Gleichung (1.33) wie folgt:

```
efeld =
-{D[pot[{x,y,z}],x],D[pot[{x,y,z}],y],D[pot[{x,y,z}],z]}
```

und seine Richtung erhält man durch Division durch $|\boldsymbol{E}|$:

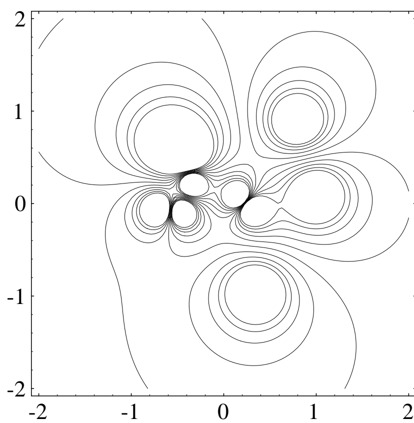
```
richtung = efeld/betrag[efeld]
```



1.14 Potential Φ der 10 Einheitsladungen in der Ebene $z = 0$.

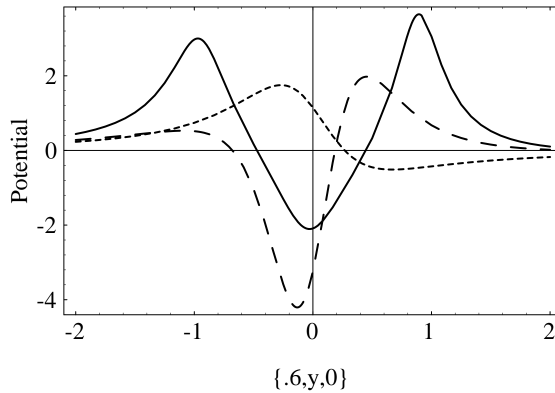
Ergebnisse

In unserem einfachen Beispiel sind fünf positive und fünf negative Ladungen $|e_i| = 1$ zufällig in der x-y-Ebene verteilt (Bild 1.13). Sie erzeugen in dieser Ebene das Potential `pot[{x,y,0}]`, das in Bild 1.14 als Gebirge und in 1.15 durch seine Höhenlinien ($\Phi = \text{const.}$) dargestellt ist.



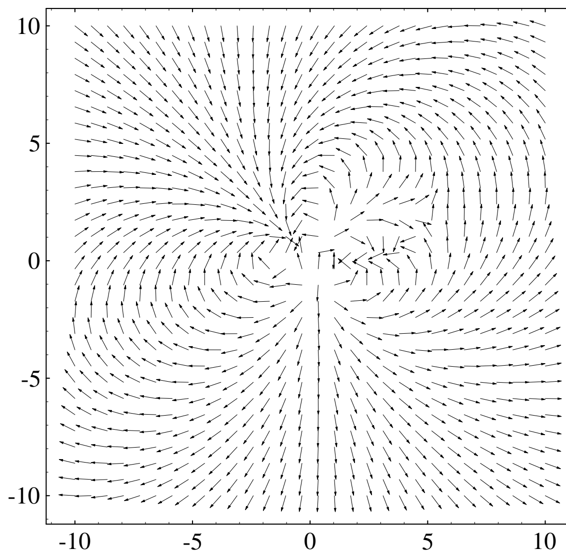
1.15 Wie Bild 1.14, aber als Kontur-Darstellung.

Wir wollen nun die Dipol- und Quadrupolnäherung mit dem exakten Potential vergleichen. Dazu betrachten wir in der x-y-Ebene $(z = 0)$ einen Weg mit $x = 0.6$ parallel zur y-Achse, der dicht an zwei positiven Ladungen bei $y \simeq \pm 1$, einer negativen Ladung und einem Dipol bei $y \simeq 0$ vorbeiläuft. Bild 1.16 zeigt das Ergebnis. Die durchgezogene Kurve ist das exakte Potential. Der Dipolterm (kurz gestrichelt)



1.16 Das exakte Potential (durchgezogen), der Dipolterm (kurz gestrichelt) und die Quadrupolnäherung.

kann wegen $\boldsymbol{p} \cdot \boldsymbol{r}$ nur einmal das Vorzeichen wechseln, er kann also nicht die zwei positiven Maxima bei den positiven Ladungen richtig wiedergeben. Aber auch mit der Quadrupolkorrektur (lang gestrichelt) beschreibt die Näherung nur grob qualitativ das Potential in der Nähe der Ladungen. Die Richtung des elektrischen Feldes ist in Bild 1.17 zu sehen. Weit entfernt von den Ladungen dreht sich das Feld wie



1.17 Die Richtung des elektrischen Feldes der zehn Einheitsladungen in der Ebene $z = 0$.

beim Dipol um den Ursprung. In der Nähe der Ladungen zeigt sich dagegen eine komplizierte Struktur.

Übung

Es soll das Magnetfeld berechnet werden, das eine vom Strom $\boldsymbol{I}$ durchflossene kreisförmige Leiterschleife erzeugt. Die Leiterschleife liege in der x-y-Ebene mit Mittelpunkt im Ursprung und habe den Radius a. Das Vektorpotential $\boldsymbol{A}$, aus dem man das Magnetfeld $\boldsymbol{B} = \boldsymbol{\nabla} \times \boldsymbol{A}$ berechnet, hat in Kugelkoordinaten (r, θ, ϕ) mit den Einheitsvektoren $\boldsymbol{e}_r, \boldsymbol{e}_\theta, \boldsymbol{e}_\phi$ eine besonders einfache Form. Nur seine ϕ-Komponente ist von null verschieden, und im Lehrbuch von J. D. Jackson findet man dafür den Ausdruck

$$A_\phi(r,\theta) = \frac{\mu_0}{\pi} \frac{I\,a}{\sqrt{a^2 + r^2 + 2ar\sin\theta}} \left[\frac{(2-k^2)K(k) - 2E(k)}{k^2}\right]$$

mit

$$k^2 = \frac{4ar\sin\theta}{a^2 + r^2 + 2ar\sin\theta} \,.$$

$K(k) =$ `EllipticK[k²]` und $E(k) =$ `EllipticE[k²]` sind die vollständigen elliptischen Integrale erster und zweiter Art.
Berechnen Sie das Magnetfeld $\boldsymbol{B}(x, y, z)$ und zeichnen Sie in der x-z-Ebene mit der Funktion `PlotVectorField[...]` die Richtung des $\boldsymbol{B}$-Feldes. Versuchen Sie, durch Integration einer geeigneten Differentialgleichung das Feldlinienbild von $\boldsymbol{B}$ in der x-z-Ebene zu zeichnen.
Hinweis: Die Schwierigkeit, daß *Mathematica* mit den Ableitungen `EllipticK'` und `EllipticE'` nichts anzufangen weiß, läßt sich beheben, weil die Ableitungen der elliptischen Integrale wiederum durch elliptische Integrale ausgedrückt werden können. Die zwei Zeilen

```
EllipticK'[x_]=1/(2x)*(EllipticE[x]/(1-x)-EllipticK[x])
EllipticE'[x_]=1/(2x)*(EllipticE[x]-EllipticK[x])
```

in Ihrem Programm lösen das Problem.

Literatur

I.S. Gradshteyn, I.M. Ryzhik, *Table of Integrals, Series, and Products*, Academic Press, 1980.

J. D. Jackson, *Classical Electrodynamics*, John Wiley & Sons, 1975.

R. Schaper, *Grafik mit Mathematica*, Addison Wesley, 1994.

1.7 Wegintegrale

Physik

Arbeit = Kraft × Weg. Diese scheinbar einfache Gleichung aus der Schulphysik wird sofort schwieriger, wenn man sich klarmacht, daß die Kraft ein Vektorfeld $\boldsymbol{K}(\boldsymbol{r})$ ist, also jedem Punkt $\boldsymbol{r}$ im Raum einen Vektor zuordnet, und daß der Weg W eine Kurve $\boldsymbol{r}(t)$ im Raum ist, die durch eine Zeit t parametrisiert werden kann. Die Arbeit A ist damit das Wegintegral

$$A = \int_W \boldsymbol{K} \cdot d\boldsymbol{r} = \int_{t_a}^{t_e} \boldsymbol{K}(\boldsymbol{r}(t)) \cdot \frac{d\boldsymbol{r}}{dt} dt \,. \tag{1.36}$$

Das Skalarprodukt Kraft · Geschwindigkeit wird also über das Zeitintervall $[t_a, t_e]$ integriert. Dieses Integral wird einfach, wenn $\boldsymbol{K}(\boldsymbol{r})$ der Gradient eines Potentials $\Phi(\boldsymbol{r})$ ist,

$$\boldsymbol{K} = -\boldsymbol{\nabla}\Phi \,, \tag{1.37}$$

dann gilt $A = \Phi(\boldsymbol{r}(t_e)) - \Phi(\boldsymbol{r}(t_a))$. Im allgemeinen kann die Auswertung des Wegintegrals allerdings mühsam sein.

Mathematica bietet die Möglichkeit, Vektoranalysis bequem auszuführen, sogar in symbolischer Form. Wir wollen dies an einem Wegintegral demonstrieren und verweisen für weitergehende Rechnungen und Vektoroperationen in anderen Koordinatensystemen auf das Package `Calculus'VectorAnalysis'`.

Algorithmus und Ergebnis

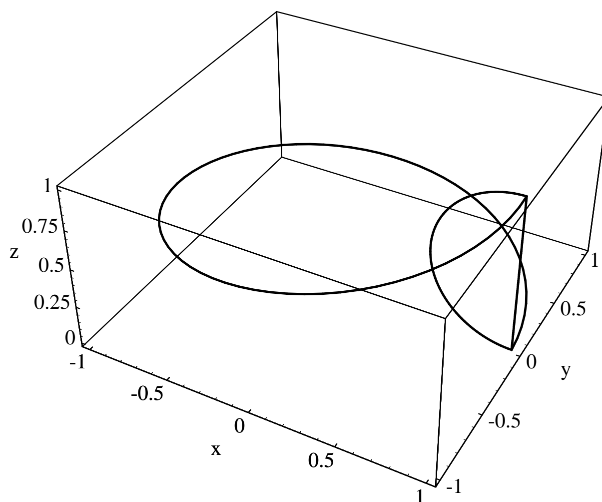
Zunächst definieren wir drei Wege $\boldsymbol{r}(t)$ im Raum, die alle von {1,0,0} nach {1,0,1} führen sollen.

```
r1 = {Cos[2Pi t], Sin[2Pi t], t}
r2 = {1, 0, t}
r3 = {1 - Sin[Pi t]/2, 0 ,(1 - Cos[Pi t])/2}
```

Der erste Weg ist eine Spirale, der zweite die Verbindungsgerade und der dritte ein Halbkreis in der x-z-Ebene. `ParametricPlot3D` zeigt diese Wege in Abbildung 1.18. Dann definieren wir ein Vektorfeld, und zwar

```
k[{x_,y_,z_}]={2x y + z^3, x^2, 3x z^2}
```

Die Geschwindigkeit, mit der der Weg durchlaufen wird, kann einfach mit der Ableitungsoperator `D[...]` definiert werden; `D[...]` ist `Listable`, wirkt also auf Listenelemente:



1.18 Drei verschiedene Integrationswege.

```
v[r_]:=D[r,t]
```

Das Wegintegral kann sogar als Funktion definiert werden, die auf Wege wirkt:

```
int[r_]:=Integrate[k[r].v[r],{t,0,1}]
```

Wir geben nun `int[r1]`, `int[r2]` und `int[r3]` ein und erhalten für alle drei Wege dasselbe Ergebnis, nämlich den Wert 1. Die Unabhängigkeit des Integrals vom Weg läßt vermuten, daß $\boldsymbol{K}$ der Gradient eines Potentials ist. Dann muß nach Sätzen der Vektoranalysis die Rotation von $\boldsymbol{K}$ verschwinden, die definiert ist als

```
rot[{kx_, ky_, kz_}]:= {  D[kz,y] - D[ky,z],
                          D[kx,z] - D[kz,x],
                          D[ky,x] - D[kx,y] }
```

`rot[k[{x,y,z}]]` gibt tatsächlich den Nullvektor `{0,0,0}`.

Wie findet man das Potential $\Phi(\boldsymbol{r})$, das bis auf eine Konstante eindeutig bestimmt ist? Nach Gl. (1.36) gilt

$$\Phi(\boldsymbol{r}) = -\int_{\boldsymbol{0}}^{\boldsymbol{r}} \boldsymbol{K}\cdot d\boldsymbol{s} = -\int_0^1 \boldsymbol{K}\cdot\frac{d\boldsymbol{s}}{dt}dt\,, \tag{1.38}$$

wobei wir `s(t)=r4=t{x,y,z}` als Weg von $\boldsymbol{0}$ bis $\boldsymbol{r}$ wählen. `int[r4]` liefert das Potential $\Phi(x,y,z) = x^2y + xz^3$.

Wir können nun das Feld $\boldsymbol{K}(\boldsymbol{r})$ ein wenig verändern, z. B. durch

```
k[{x,y,z}] = {2x y^2 + z^3, x^3, 3x z^3}
```

und erhalten dann für jeden Weg ein anderes Ergebnis: Nach dem Anwenden von `Simplify` liefert `int[r1]`

$$1 + \frac{3}{4\pi^2} + \frac{3\pi}{4},$$

`int[r2]` gibt $3/4$ und `int[r3]` den Wert $3/4 + 9\pi/256$. Berechnen wir jetzt erneut die Rotation, so ergibt `rot[k[{x,y,z}]]` in Übereinstimmung mit der Abhängigkeit des Integrals vom Integrationsweg einen von $\mathbf{0}$ verschiedenen Wert.

Übung

1. Berechnen Sie die Längen der Kurven `r1`, `r2` und `r3`.
2. Wählen Sie die Parametrisierung $t = \tau^2$ für die Wege `r1`, `r2`, `r3`, berechnen Sie die Wegintegrale und zeichnen Sie die Beträge der drei Beschleunigungen $\left|\frac{d^2\boldsymbol{r}}{dt^2}\right|$ als Funktion von τ.
3. Testen Sie die Gleichheit der Wegintegrale für ein anderes Vektorfeld $\boldsymbol{a}(x, y, z)$ mit $\boldsymbol{a} = \boldsymbol{\nabla}\Phi$ und Φ nach eigener Wahl.

Literatur

S. Großmann, *Mathematischer Einführungskurs für die Physik*, Teubner Studienbücher Physik, 1991.

1.8 Maxwell-Konstruktion

In der Physik treten häufig implizite nichtlineare Gleichungen für die gegenseitige Abhängigkeit von Größen auf, die nur noch numerisch gelöst werden können. Ein Beispiel dafür sind die $p(V, T)$-Kurven des van-der-Waals-Gases mit der Temperatur T als Parameter, die durch die sogenannte *Maxwell-Konstruktion* eine physikalische Bedeutung erlangen.

Die Isothermen der Zustandsgleichung, die als van-der-Waals-Gleichung bekannt ist, haben bei tiefen Temperaturen Schleifen, die aus thermodynamischen Gründen verboten sind. Sie müssen durch Geraden ersetzt werden, die die Schleifen in einem

bestimmten Sinn halbieren. Um diese sogenannten Maxwell-Geraden zu konstruieren, muß man eine nichtlineare Gleichung, die noch ein Integral enthält, numerisch lösen. Auf diese Weise erhält man eine Beschreibung des Phasenübergangs vom Gas zur Flüssigkeit.

Physik

Die Theorie der Wärme gibt für ein ideales Gas aus N wechselwirkungsfreien, klassischen Teilchen ohne innere Freiheitsgrade einen einfachen Zusammenhang zwischen Volumen V, Druck p und Temperatur T:

$$p\,V = N k_{\mathrm{B}} T\,. \tag{1.39}$$

Dabei ist k_{B} die Boltzmannkonstante. Unter Berücksichtigung der Wechselwirkung zwischen den Teilchen erhält man in einer einfachen Näherung die van-der-Waals-Gleichung

$$\left(p + \frac{a}{V^2}\right)(V - b) = N k_{\mathrm{B}} T \tag{1.40}$$

mit Parametern a und b.

Diese Gleichung gibt z. B. die Isothermen an, also den Druck p als Funktion des Volumens V für konstante Temperaturen T. Für hohe Temperaturen fällt der Druck mit wachsendem Volumen, während er bei tiefen Temperaturen $p(V)$ in einem gewissen Bereich wieder ansteigt (siehe Bild 1.19). Es gibt nun eine kritische Temperatur T_{c}, unterhalb derer diese Schleife auftritt. Für $T < T_{\mathrm{c}}$ findet ein Phasenübergang zwischen Gas und Flüssigkeit statt; genau bei T_{c} verschwindet der Unterschied zwischen flüssiger und gasförmiger Phase.

T_{c} und V_{c} sind dadurch bestimmt, daß die erste und zweite Ableitung von $p(V,T)$ nach V verschwinden,

$$\frac{\partial p}{\partial V}(V,T) = 0 \quad \text{und} \quad \frac{\partial^2 p}{\partial V^2}(V,T) = 0\,, \tag{1.41}$$

mit dem Ergebnis:

$$T_{\mathrm{c}} = \frac{8a}{27 N k_{\mathrm{B}} b}\,,\quad V_{\mathrm{c}} = 3\,b\,,\quad p_{\mathrm{c}} = \frac{a}{27\,b^2}\,. \tag{1.42}$$

Skaliert man nun jeweils p, V und T mit $p_{\mathrm{c}}, V_{\mathrm{c}}$ und T_{c}, so erhält man eine Gleichung, die keine Parameter mehr enthält ($\tilde{x} = x/x_{\mathrm{c}}$):

$$\left(\tilde{p} + \frac{3}{\tilde{V}^2}\right)\left(3\tilde{V} - 1\right) = 8\tilde{T}\,. \tag{1.43}$$

Für $\tilde{T} < 1$ gibt die Gleichung also die unphysikalische Schleife, d. h. für einen gewissen Bereich gibt es für jeden Druckwert $\tilde{p}$ drei Volumina $\tilde{V}_1, \tilde{V}_2$ und $\tilde{V}_3$. Der Übergang vom Gas mit großem Volumen $\tilde{V}_3$ zur Flüssigkeit mit kleinem Volumen $\tilde{V}_1$ findet aus thermodynamischen Gründen bei demjenigen Druck $\tilde{p}_\mathrm{t}$ statt, bei dem gilt:

$$\int\limits_{\tilde{V}_1}^{\tilde{V}_3} \tilde{p}(\tilde{V}) d\tilde{V} = \tilde{p}_\mathrm{t} \left(\tilde{V}_3 - \tilde{V}_1 \right) . \tag{1.44}$$

Bei $\tilde{p}_\mathrm{t}$ sind Gas und Flüssigkeit für alle Volumina $\tilde{V}$ mit $\tilde{V}_1 \leq \tilde{V} \leq \tilde{V}_3$ gleichzeitig vorhanden, man beobachtet ein Zweiphasengemisch. Geometrisch bedeutet die obige Gleichung, daß die Fläche zwischen der $\tilde{p}(\tilde{V})$-Kurve und der Geraden $\tilde{p}_\mathrm{t} = \text{const.}$ im Bereich von $\tilde{V}_1$ bis $\tilde{V}_2$ genauso groß ist wie die entsprechende Fläche zwischen $\tilde{V}_2$ und $\tilde{V}_3$ (siehe Bild 1.19). Die Kurve $\tilde{p}_\mathrm{t} = \text{const.}$, die man Maxwell-Gerade nennt, wollen wir mit *Mathematica* konstruieren.

Algorithmus und Ergebnis

Zunächst überprüfen wir die Gleichung (1.42) für den kritischen Punkt. Wir geben die van-der-Waals-Gleichung (1.40) ein $(Nk_\mathrm{B}T = t)$,

```
p=t/(v-b) - a/v^2
```

und definieren die beiden Gleichungen (1.41):

```
gl1=D[p,v]==0
gl2=D[p,{v,2}]==0
```

Dabei bedeutet das Zeichen = eine Zuordnung, während == die Gleichheit überprüft und zuerst ausgeführt wird. Diese Gleichungen lösen wir nach `t` und `v` auf,

```
sol=Solve[{gl1,gl2},{t,v}]
```

mit einer Regel als Ergebnis:

```
          8 a
{{ t -> ----, v -> 3b}}
          27 b
```

Da die Lösung nicht immer eindeutig ist, liefert `Solve` eine Liste von Regeln. Daher stammen die doppelten geschweiften Klammern. Den kritischen Druck erhalten wir, indem wir den inneren Teil der Regel auf `p` anwenden,

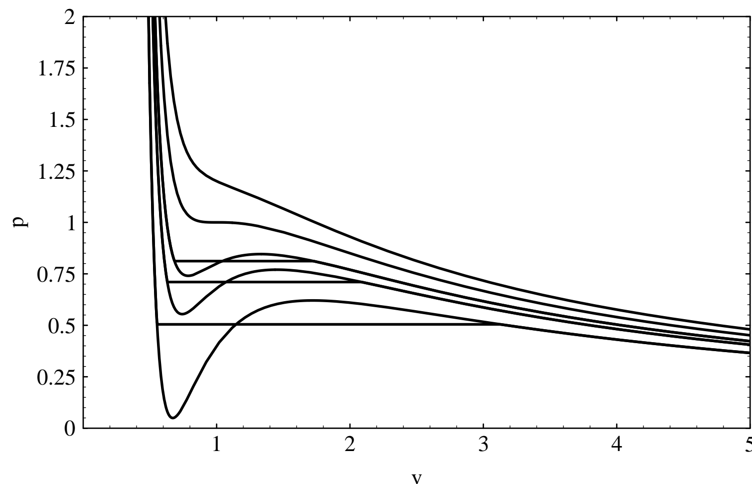
```
pc = p/.sol[[1]]
```

mit dem Ergebnis `a/(27 b^2)`.

Jetzt definieren wir die skalierte $\tilde{p}(\tilde{V})$ Gleichung (1.43):

```
p[v_]=8t/(3v-1)-3/v^2
```

Für die Maxwell-Gerade benötigen wir zwei Gleichungen, um die beiden unbekann-



1.19 Maxwell-Konstruktion für die van-der-Waals-Gleichung. Isothermen zu den skalierten Temperaturen $\tilde{T} = 1.05, 1.0, 0.95, 0.92, 0.85$ (von oben nach unten).

ten Volumina `v1` und `v3`, zwischen denen Gas und Flüssigkeiten koexistieren, zu bestimmen. Die erste Gleichung sagt, daß Koexistenz Gleichheit der Drücke verlangt:

```
gl3=p[v1]==p[v3]
```

Die zweite Gleichung setzt die beiden Flächen nach Gleichung (1.44) gleich:

```
gl4=p[v1]*(v3-v1)==Integrate[p[v],{v,v1,v3}]
```

Ein Versuch mit `Solve` zeigt an, daß *Mathematica* keine analytische Lösung findet. Wir werden deshalb `FindRoot` benutzen, um eine numerische Lösung zu erhalten. Allerdings müssen die Startwerte der `(v1,v3)`-Suche ziemlich genau angegeben werden, damit ein vernünftiges Ergebnis gefunden wird. In der Funktion `plot[T_]` werden deshalb für Temperaturen `T` < 1 zunächst diejenigen `v`-Werte bestimmt, an denen `p[v]` ein relatives Minimum bzw. Maximum hat, und daraus wird das arithmetische Mittel `vtest` gebildet. Mit dem *Mathematica*-Befehl

`Solve[p[v]==p[vtest],v]` erhält man dann drei Lösungen, von denen die beiden äußeren als Startwerte geeignet sind. Die endgültige Maxwell-Funktion ist dann

```
pmax[v_]:= If[v < v1 || v > v3, p[v], p[v1]]
```

Falls `v` zwischen den Lösungen `v1` und `v3` liegt, wird also die Maxwell-Gerade übergeben, sonst `p[v]`. Bild 1.19 zeigt das Ergebnis für verschiedene Werte der skalierten Temperatur $\tilde{T}$ zusammen mit den unphysikalischen `p[v]`-Funktionen. Bei `v` $\to \frac{1}{3}$ divergiert der Druck. Die obere Kurve beschreibt $p(V)$ in der Gasphase, und die darunterliegende gibt die Situation gerade bei der kritischen Temperatur wieder. Die unteren drei $p(V)$-Kurven zeigen bei tiefen Temperaturen eine Trennung in eine flüssige (kleines Volumen) und eine gasförmige Phase. Auf der Maxwell-Geraden koexistieren Flüssigkeit und Gas im thermischen Gleichgewicht.

Übung

Betrachten Sie ein Quantenteilchen der Masse m in einem eindimensionalen Potentialtopf

$$V(x) = \begin{cases} -V_0 & \text{für } -a \leq x \leq a\,, \\ 0 & \text{sonst}\,. \end{cases}$$

Aus der Quantenmechanik-Vorlesung wissen Sie, daß die Energieniveaus E der gebundenen Zustände durch folgende Gleichungen bestimmt sind:

$\epsilon = \dfrac{E}{V_0} = \dfrac{q^2}{x^2} - 1$ mit $x = \dfrac{a}{\hbar}\sqrt{2mV_0}$ und q Lösung der transzendenten Gleichung

$$\tan q = \frac{\sqrt{x^2 - q^2}}{q} \quad \text{oder} \quad -\cot q = \frac{\sqrt{x^2 - q^2}}{q}\,.$$

Berechnen und zeichnen Sie alle Energieniveaus ϵ_n als Funktion des Parameters x.

Literatur

T. Fließbach, *Statistische Physik*, Spektrum Akademischer Verlag, 1995.

E. W. Schmid, G. Spitz, W. Lösch, *Theoretische Physik mit dem Personal Computer*, Springer, 1987.

1.9 Beste Spielstrategie

Als letztes Beispiel für die Anwendung vordefinierter *Mathematica*-Funktionen wollen wir uns eine Optimierungsaufgabe ansehen, die nicht direkt aus der Physik

stammt. Lineares Optimieren wird zwar hauptsächlich in den Wirtschaftswissenschaften (Operations Research) angewandt, trotzdem gibt es auch in der Physik immer wieder Probleme, die diese Methoden – meist mit Benutzung fertiger Programmpakete – benötigen.

Hier wollen wir uns mit einem Spiel beschäftigen, das zwei Personen mit Hilfe einer Auszahlungstabelle spielen. Um die beste Spielstrategie zu finden, brauchen die beiden Kontrahenten einige Sätze der mathematischen Spieltheorie, die in den zwanziger Jahren von J. von Neumann begründet wurde, und außerdem das *Mathematica*-Programm `LinearProgramming`.

Mathematik

Es spielen zwei Personen gegeneinander. Nach jedem Zug muß ein Spieler dem anderen einen Betrag bezahlen, der aus einer Tabelle abgelesen wird. Jeder Spieler versucht, seine Züge so zu wählen, daß die Summe seiner Gewinne und Verluste nach vielen Zügen eine positive Bilanz aufweist.

Für das Spiel wird eine Gewinntabelle $\mathsf{K} = (K_{ij})$ benötigt, z. B.

$$\mathsf{K} = \begin{pmatrix} 0 & 1 & 3 & 1 \\ -1 & 10 & 4 & 2 \\ 7 & -2 & 3 & 7 \end{pmatrix}.$$

Der Spieler Z kann bei jedem Zug eine der drei Zeilen $i = 1, 2, 3$ wählen, während Spieler S vier Wahlmöglichkeiten für die Spalten $j = 1, ..., 4$ hat. Ein Zug besteht darin, daß Spieler Z sich für eine Zeilennummer i und Spieler S sich für eine Spaltennummer j entscheidet, ohne daß der eine von der Wahl des anderen weiß. Dann wird aufgedeckt, und danach erhält Spieler Z gemäß dem Paar (i, j) den Betrag K_{ij} vom Spieler S, bzw. zahlt $|K_{ij}|$ für $K_{ij} < 0$. Da der Gewinn von Z der Verlust von S ist und umgekehrt, nennt man dieses Spiel *Zwei-Personen-Nullsummen-Spiel*. Diese Prozedur wird viele Male wiederholt. Am Ende hat Z im Mittel pro Zug den Betrag K erhalten. Z sucht also eine Strategie, die einen möglichst großen mittleren Gewinn K ergibt.

Natürlich möchte auf der anderen Seite S seine Verluste minimieren, er sucht also das Minimum von K. Für jede Wahl i von Z sucht er also $\min_j K_{ij}$. Da er aber nicht weiß, welche Zeile Z wählen wird, könnte er auf die Idee kommen, diejenige Spalte zu wählen, die im schlechtesten Fall den geringsten Verlust ergibt. Für jede Wahl j verliert S im schlechtesten Fall den Wert $\max_i K_{ij}$. Er wählt also dasjenige j, das zu $\min_j \max_i K_{ij}$ gehört, hier die Spalte $j = 3$. Bei entsprechender Überlegung würde Z die Zeile zum Wert $\max_i \min_j K_{ij}$ wählen, also $i = 1$. Bei dieser Strategie würde nach jedem Zug an Spieler Z der Betrag $K_{13} = 3$ ausgezahlt werden. Die von den jeweiligen Spielern ins Auge gefaßten Grenzen haben

unterschiedliche Werte:

$$0 = \max_i \min_j K_{ij} \neq \min_j \max_i K_{ij} = 4\,. \tag{1.45}$$

Was bedeutet nun ein Gewinn von 3 für Spieler Z, der bei dieser Strategie einen Mindest-„Gewinn“ von 0 erwarten konnte? Und sollte Spieler S, der im ungünstigsten Fall mit einem Verlust von 4 hatte rechnen müssen, mit der 3 zufrieden sein? In der Tat hat Spieler Z allen Grund, seine Strategie zu überdenken. Eine genauere Inspektion der Gewinntabelle sagt ihm nämlich, daß für jede Wahl von S sein Gewinn im Mittel mindestens 3 sein wird, wenn er die Zeilen 2 und 3 zufällig wechselnd gleich häufig auswählt. Und wenn er Zeile 3 dabei ein wenig bevorzugt, wird er den Gewinn von 3 sogar übertreffen. Man sieht an dieser Überlegung, daß es zweckmäßig ist, die Zeilen und Spalten mit einer gewissen Wahrscheinlichkeit p_i bzw. q_j zu wählen. Eine Strategie für den Spieler Z besteht also aus den 3 Häufigkeiten p_1, p_2 und p_3 (mit $p_1 + p_2 + p_3 = 1$), mit denen er unkorreliert die Zeilen wählt. Im Mittel erhält er den Betrag

$$K = \sum_{i,j} p_i\, q_j\, K_{ij}\,. \tag{1.46}$$

Wenn Z nun eine Strategie p_1, p_2, und p_3 wählt, bekommt er im schlechtesten Fall $\min_{q_1 \ldots q_4} \sum_{i,j} p_i\, K_{ij}\, q_j$. Z wird daher ein p_1^0, p_2^0, p_3^0 wählen, das den Gewinn im ungünstigsten Fall maximiert, er sucht also

$$\max_{p_1 \ldots p_3}\ \min_{q_1 \ldots q_4} \sum_{i,j} p_i\, K_{ij}\, q_j\,. \tag{1.47}$$

S sucht analog eine Strategie $q_1^0 \ldots q_4^0$, die den Wert

$$\min_{q_1 \ldots q_4}\ \max_{p_1 \ldots p_3} \sum_{i,j} p_i\, K_{ij}\, q_j \tag{1.48}$$

liefert. Entgegen dem deterministischen Spiel (1.45) geben die optimalen stochastischen Strategien (1.47) und (1.48) denselben Wert:

$$\max_{\boldsymbol{p}}\ \min_{\boldsymbol{q}} \sum_{ij} p_i\, K_{ij}\, q_j = \min_{\boldsymbol{q}}\ \max_{\boldsymbol{p}} \sum_{ij} p_i\, K_{ij}\, q_j = \sum_{ij} p_i^0\, K_{ij}\, q_j^0 = K_0\,. \tag{1.49}$$

Dies ist das berühmte Minimax-Theorem, das J. von Neumann 1926 im Alter von 23 Jahren bewiesen hat. Wir wollen die Bedeutung von (1.49) noch in Worte fassen: Wenn Spieler Z eine optimale Strategie $p_1^0, \ldots, p_3^0$ wählt, so ist sein Gewinn für jede Strategie $q_1, \ldots, q_4$ des Spielers S nach sehr vielen Spielzügen mindestens K_0. Es gilt also für jede Wahl $q_1, \ldots, q_4$ mit $q_i \geq 0$ und $q_1 + q_2 + q_3 + q_4 = 1$:

$$\sum_{i,j} p_i^0\, K_{ij}\, q_j \geq K_0\,. \tag{1.50}$$

Aus dieser Ungleichung kann man ein System von 4 Bedingungen ableiten, denn mit $q_1 = 1$ und $q_2 = q_3 = q_4 = 0$ wird daraus $\sum_i \; p_i^0 \, K_{i1} \geq K_0$ und analog:

$$\sum_i \; p_i^0 \, K_{ij} \geq K_0 \;\text{ für alle } j \,. \tag{1.51}$$

Nehmen wir an, daß $K_0 > 0$ gilt, so wird dies mit $x_i^0 = p_i^0 / K_0$ zu

$$\sum_i \; x_i^0 \, K_{ij} \geq 1 \;\text{ für alle } j \,. \tag{1.52}$$

Nach Sätzen der linearen Optimierung ist nun die optimale Strategie p_1^0, p_2^0, p_3^0 durch den maximalen Wert K_0 im Gleichungssystem (1.51) bestimmt, oder wegen $\sum_i \; x_i^0 = 1/K_0$ durch das Minimum von $\sum_i \; x_i^0$.

Dieser Sachverhalt läßt sich in Vektorschreibweise noch kompakter formulieren: Es seien K die Matrix (K_{ij}) und K^{T} die Transponierte von K, $\boldsymbol{x} = (x_1, x_2, x_3)^{\mathsf{T}}$ mit $x_i \geq 0$, $\boldsymbol{c} = (1,1,1)^{\mathsf{T}}$, $\boldsymbol{b} = (1,1,1,1)^{\mathsf{T}}$. Dann ist die optimale Strategie $\boldsymbol{p}_0 = K_0 \, \boldsymbol{x}_0$ und deren mittlerer Gewinn K_0 durch ein Minimum von $\boldsymbol{c} \cdot \boldsymbol{x}$ unter der Nebenbedingung $\mathsf{K}^{\mathsf{T}} \, \boldsymbol{x} \geq \boldsymbol{b}$ bestimmt, wobei letzte Vektorungleichung komponentenweise gemeint ist. Es gilt $K_0 = 1/(\boldsymbol{c} \cdot \boldsymbol{x}_0)$.

Nach dem Dualitätssatz der linearen Optimierung kann man K_0 auch aus folgendem Problem gewinnen:

Suche das Maximum von $\boldsymbol{b} \cdot \boldsymbol{y}$ unter der Nebenbedingung $\mathsf{K} \, \boldsymbol{y} \leq \boldsymbol{c}$ für den Vektor $\boldsymbol{y} = (y_1, ..., y_4)^{\mathsf{T}}$ mit $y_j \geq 0$.

Die Lösung $\boldsymbol{y}_0$ dieser Aufgabe gibt aber gerade die optimale Strategie für den Spieler S. Denn mit $\boldsymbol{q}_0 = K_0 \, \boldsymbol{y}_0$ erhält man

$$\sum_j \; K_{ij} \, q_j^0 \leq K_0 \;\text{ für alle } i \tag{1.53}$$

und damit analog zu (1.50) und (1.51) für jede Strategie $\boldsymbol{p}$ von Z:

$$\sum_{i,j} \; p_i \, K_{ij} \, q_j^0 \leq K_0 \,. \tag{1.54}$$

Im schlechtesten Fall verliert also der optimal spielende S den Betrag K_0. Spielt S nicht mit der Strategie $\boldsymbol{q}_0$, kann er allerdings höher verlieren.

Ein nützlicher Satz soll noch erwähnt werden: Addiert man zu jedem Matrixelement die Konstante d, so bleiben $\boldsymbol{p}_0$ und $\boldsymbol{q}_0$ optimale Strategien mit dem Spielwert $K_0 + d$. Die obigen Aussagen zu (1.51) gelten streng nur für Matrizen mit positiven Werten K_{ij}. Wegen des Verschiebungssatzes kann aber jede Matrix K ins Positive verschoben werden; danach wird das Optimierungsprogramm angewendet.

Algorithmus und Ergebnis

Das Minimum einer linearen Funktion $\boldsymbol{c} \cdot \boldsymbol{x}$ unter den Nebenbedingungen $\mathsf{K}^{\mathsf{T}}\,\boldsymbol{x} \geq \boldsymbol{b}\,,\ \boldsymbol{x} \geq 0$ kann mit *Mathematica* leicht bestimmt werden. In unserem Beispiel ist

```
c = {1, 1, 1}
b = {1, 1, 1, 1}
k = {{0, 1, 3, 1},{-1, 10, 4, 2},{7, -2, 3, 7}}
```

und

```
LinearProgramming[c,Transpose[k],b]
```

gibt einen optimalen Vektor $\boldsymbol{x}_0$, aus dem $\boldsymbol{p}_0$ durch $\boldsymbol{p}_0 = \boldsymbol{x}_0/(\boldsymbol{c} \cdot \boldsymbol{x}_0)$ und K_0 durch $K_0 = 1/(\boldsymbol{c} \cdot \boldsymbol{x}_0)$ bestimmt werden.

Anstatt das Maximum von $\boldsymbol{b} \cdot \boldsymbol{y}$ können wir auch das Minimum von $-\boldsymbol{b} \cdot \boldsymbol{y}$ suchen. Aus $\mathsf{K}\,\boldsymbol{y} \leq \boldsymbol{c}$ folgt $-\mathsf{K}\,\boldsymbol{y} \geq -\boldsymbol{c}$. Also lösen wir das duale Problem mit

```
LinearProgramming[-b,-k,-c]
```

Dies liefert den Vektor $\boldsymbol{y_0}$ und damit $K_0 = 1/(\boldsymbol{b} \cdot \boldsymbol{y}_0)$ und $\boldsymbol{q}_0 = K_0\,\boldsymbol{y}_0$.

Für unser Beispiel ergibt die Rechnung

$$\boldsymbol{p}_0 = (0,\ 0.45,\ 0.55)^{\mathsf{T}},\ \boldsymbol{q}_0 = (0.6,\ 0.4,\ 0,\ 0)^{\mathsf{T}},\ K_0 = 3.4\,.$$

Damit spielt Z optimal, wenn er die Zeilen 2 und 3 mit den Häufigkeiten 0.45 bzw. 0.55 wählt, während S in jedem Fall nicht mehr als $K_0 = 3.4$ verliert, wenn er die Spalten 1 und 2 mit den Wahrscheinlichkeiten 0.6 bzw. 0.4 nennt. Beide müssen aber dafür Sorge tragen, daß sie die Wahl wirklich zufällig treffen, denn sonst könnte der Gegner auf die Korrelationen zu seinem Vorteil reagieren.

In unserem Beispiel funktioniert der Algorithmus auch für negative K_{ij}-Werte. Bei anderen Matrizen fanden wir aber erst eine Lösung, nachdem wir die Matrix K ins Positive verschoben hatten.

Zur Unterhaltung haben wir schließlich das C-Programm `spiel.c` geschrieben, mit dem die Leser unseres Buches sich im *Zwei-Personen-Nullsummen-Spiel* gegen den Computer versuchen können. Der Rechner erzeugt am Anfang mit Zufallszahlen eine 4×4-Auszahlungsmatrix mit $K_0 = 0$ und berechnet dazu mit dem Programm `simplx` aus den *Numerical Recipes* die optimalen Strategien. Der Spieler wählt bei jedem Zug eine der vier Zeilen, gleichzeitig zieht der Computer eine der Spalten mit den Wahrscheinlichkeiten q_j^0. Nach der obigen Theorie könnte der Leser daher durch eine optimale statistische Auswahl der Zeilen im Mittel verlustfrei spielen. Allerdings wird jeder sofort merken, daß der Computer langfristig gewinnt, es sei denn, der Gegenspieler berechnet seine beste Strategie p_i^0 und handelt danach. Am Ende des Spiels (Taste `e`) wird gezeigt, wie der Spieler hätte spielen müssen, um im Mittel verlustfrei zu bleiben.

Übung

Das wohlbekannte Knobelspiel mit den drei Symbolen Schere, Stein, Papier wollen wir auf die vier Symbole Schere, Stein, Papier und Brunnen erweitern. Zwei Spieler setzen dabei jeweils auf eine dieser vier Möglichkeiten und gewinnen, bzw. verlieren einen Punkt nach folgenden Regeln:

- Brunnen verschluckt Stein, Brunnen verschluckt Schere,
- Papier deckt Brunnen zu, Papier umhüllt Stein,
- Schere zerschneidet Papier,
- Stein zerschlägt Schere.

Formulieren Sie für dieses Spiel eine Auszahlungsmatrix und berechnen Sie damit die optimale Strategie der beiden Spieler.

Literatur

John von Neumann, *Collected Works, Volume VI : Zur Theorie der Gesellschaftsspiele*, Seite 1, Pergamon Press, 1963.

W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge University Press, 1992.

W. Vogel, *Lineares Optimieren*, Akademische Verlagsgesellschaft, 1967.

Kapitel 2
Lineare Gleichungen

Viele Phänomene der Physik können durch lineare Gleichungen beschrieben werden. Doppelte Ursache gibt doppelte Wirkung – dieser Sachverhalt erlaubt es, solche Probleme mathematisch zu lösen. Häufig lassen sich lineare Systeme durch Vektoren beschreiben, Vektoren mit manchmal wenigen, manchmal sehr vielen Komponenten. In den Bewegungsgleichungen tauchen dann Matrizen auf, deren Eigenwerte und Eigenvektoren die Energien und die stationären Zustände des Systems beschreiben. Jede andere Form der Bewegung ist eine Überlagerung dieser Eigenzustände.

Es gibt eine Vielzahl von numerischen Methoden, lineare Gleichungen zu lösen. Diese weit entwickelten Standard-Algorithmen werden in den Lehrbüchern zur numerischen Mathematik ausführlich erklärt. Wir haben in diesem Kapitel, abgesehen vom Hofstadter-Schmetterling, die Beispiele so gewählt, daß sie sich mit vordefinierten Funktionen untersuchen lassen. Beim elektrischen Netzwerk geht es darum, ein lineares Gleichungssystem zu lösen, und außerdem können wir noch einmal die Fouriertransformation anwenden. Andere Fragestellungen, die sowohl der Mechanik als auch der Quantenphysik entstammen, führen uns auf Eigenwertgleichungen, also auf das Problem, Eigenwerte und Eigenvektoren von eventuell großen Matrizen zu bestimmen.

2.1 Quantenoszillator

Die Bewegungsgleichung der Quantenmechanik, die Schrödinger-Gleichung, ist linear: Jede Überlagerung von Lösungen ist wieder eine Lösung. Deshalb hat man Erfolg mit dem Verfahren, die Gleichung in Raum und Zeit durch einen Produktansatz zu separieren und später diese Produktlösungen zu überlagern. Der räumliche Anteil ist die sogenannte stationäre Schrödinger-Gleichung – eine Eigenwertgleichung, die in der Ortsdarstellung die Form einer linearen Differentialgleichung hat. Die Lösungen dieser Gleichung sind Wellenfunktionen $\Psi(\boldsymbol{r})$, die jedem Ort $\boldsymbol{r}$ eine komplexe Zahl Ψ zuordnen, und zwar beschreiben sie diejenigen Zustände des physikalischen Systems, deren Aufenthaltswahrscheinlichkeit $|\Psi(\boldsymbol{r})|^2$ sich zeitlich nicht ändert. Zur numerischen Lösung der Schrödinger-Gleichung kann man entweder die lineare Differentialgleichung näherungsweise diskretisieren und in Matrixform bringen,

oder man kann $\Psi(\boldsymbol{r})$ nach einem vollständigen Satz von Wellenfunktionen $\varphi_n(\boldsymbol{r})$ entwickeln und nur endlich viele davon betrachten. In beiden Fällen führt die stationäre Schrödinger-Gleichung auf eine Eigenwertgleichung einer endlichen Matrix. Wir wollen den zweiten Ansatz am Beispiel des anharmonischen Oszillators studieren.

Physik

Wir behandeln ein eindimensionales Problem, nämlich ein Teilchen der Masse m im quadratischen Potential $V(q) = \frac{1}{2}m\omega^2 q^2$. Dabei ist q die Ortskoordinate des Teilchens. Der Hamiltonoperator dazu lautet in dimensionsloser Form

$$H_0 = \frac{1}{2}\left(p^2 + q^2\right) . \tag{2.1}$$

Dabei werden Energien in Einheiten von $\hbar\omega$, Impulse in Einheiten von $\sqrt{\hbar m\omega}$ und Längen in Einheiten von $\sqrt{\hbar/(m\omega)}$ gemessen. Die Eigenzustände $|j\rangle$ von H_0 findet man in jedem Lehrbuch der Quantenmechanik, sie lauten in der Ortsdarstellung:

$$\varphi_j(q) = \left(2^j\, j!\, \sqrt{\pi}\right)^{-\frac{1}{2}} e^{-\frac{1}{2}q^2}\, H_j(q)\, , \tag{2.2}$$

wobei $H_j(q)$ die Hermitepolynome sind . Es gilt:

$$H_0|j\rangle = \varepsilon_j^0|j\rangle \;\text{ mit }\; \varepsilon_j^0 = j + \frac{1}{2} \;\text{ und }\; j = 0, 1, 2, \ldots .$$

ε_j^0 sind die Energien der Eigenzustände von H_0, und die Matrix $\langle j|H_0|k\rangle$ ist diagonal, weil die Eigenwerte nicht entartet sind.

Wir fügen nun zu H_0 ein anharmonisches Potential hinzu,

$$H = H_0 + \lambda\, q^4, \tag{2.3}$$

und suchen die Matrix $\langle j|H|k\rangle$. Dazu ist es zweckmäßig, q als Summe von Erzeugungs- und Vernichtungsoperatoren $a^\dagger$ und a zu schreiben:

$$q = \frac{1}{\sqrt{2}}(a^\dagger + a)\, , \tag{2.4}$$

wobei a und $a^\dagger$ die folgenden Eigenschaften haben:

$$\begin{aligned} a^\dagger|j\rangle &= \sqrt{j+1}\,|j+1\rangle\, , \\ a|0\rangle = 0 \;\text{ und }\; a|j\rangle &= \sqrt{j}\,|j-1\rangle \;\text{ für }\; j > 0\, . \end{aligned} \tag{2.5}$$

Die Matrixdarstellung von q im Raum der ungestörten Zustände $|j\rangle$ liefert also die Tridiagonalmatrix

$$Q_{jk} = \langle j|q|k\rangle \; = \frac{1}{\sqrt{2}}\,\sqrt{k+1}\,\delta_{j,k+1} + \frac{1}{\sqrt{2}}\sqrt{k}\,\delta_{j,k-1} \; = \; \frac{1}{2}\sqrt{j+k+1}\,\delta_{|k-j|,1}\,. \tag{2.6}$$

Die Näherung, die wir jetzt vornehmen, besteht darin, diese unendlich-dimensionale Matrix nur für $j,k = 0,1,...,n-1$ zu definieren. Der Hamiltonoperator $H = H_0 + \lambda\, q^4$, bzw. seine Matrixdarstellung, soll ebenfalls als $n \times n$-Matrix genähert werden, wobei H_0 durch die Diagonalmatrix mit den Elementen $j + \frac{1}{2}$ und q^4 durch das vierfache Matrixprodukt von Q_{jk} mit sich selbst dargestellt wird. Den Fehler der Approximation kann man abschätzen, indem man die Eigenwerte von H für verschiedene n-Werte miteinander vergleicht.

Algorithmus

Die obigen Matrizen lassen sich in *Mathematica* besonders kompakt definieren. $\langle j|q|k\rangle$ ist nach Gleichung (2.6)

```
q[j_,k_]:= Sqrt[(j+k+1)]/2 /; Abs[j-k]==1
q[j_,k_]:= 0 /; Abs[j-k] != 1
```

Die Konstruktion `lhs := rhs /; test` bedeutet, daß die Definition nur dann benutzt wird, wenn der rechte Ausdruck wahr ist, also den Wert `True` hat. Die Matrix `q` wird als Liste von Listen definiert

```
q[n_]:= Table[q[j,k], {j,0,n-1}, {k,0,n-1}]
```

und H_0 ergibt sich gemäß Gleichung (2.2) zu

```
h0[n_]:= DiagonalMatrix[Table[j+1/2, {j,0,n-1}]]
```

Damit läßt sich H schreiben als

```
h[n_]:= h0[n] + lambda q[n].q[n].q[n].q[n]
```

Die Eigenwerte von H erhält man mit `Eigenvalues[...]`, entweder algebraisch (nur für kleine n-Werte) oder numerisch mit `Eigenvalues[N[h[n]]]`, wobei `lambda` zuvor ein numerischer Wert zugewiesen werden muß.

Ergebnisse

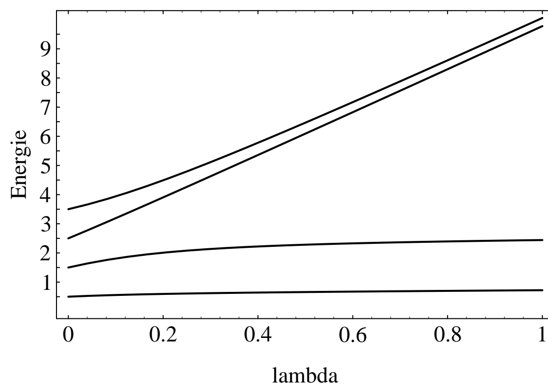
Der Aufruf `h[4] // MatrixForm` liefert die Hamiltonmatrix

$$\begin{pmatrix} \frac{1}{2}+\frac{3}{4}\lambda & 0 & \frac{3}{\sqrt{2}}\lambda & 0 \\ 0 & \frac{3}{2}+\frac{15}{4}\lambda & 0 & 3\sqrt{\frac{3}{2}}\lambda \\ \frac{3}{\sqrt{2}}\lambda & 0 & \frac{5}{2}+\frac{27}{4}\lambda & 0 \\ 0 & 3\sqrt{\frac{3}{2}}\lambda & 0 & \frac{7}{2}+\frac{15}{4}\lambda \end{pmatrix} . \tag{2.7}$$

Deren Eigenwerte lauten

$$\frac{6+15\lambda \pm 2\sqrt{2}\,\sqrt{2+12\lambda+27\lambda^2}}{4} \quad \text{und} \quad \frac{10+15\lambda \pm 2\sqrt{2}\,\sqrt{2+27\lambda^2}}{4} . \tag{2.8}$$

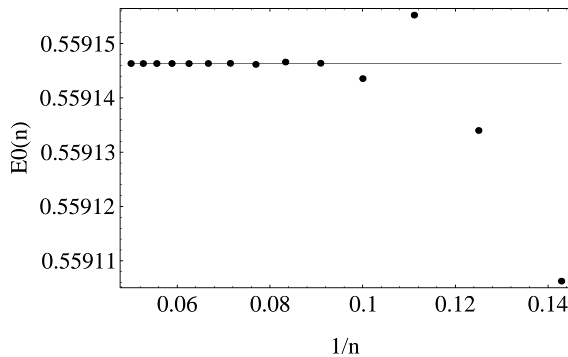
Diese vier Eigenwerte sind in Abbildung 2.1 als Funktion von λ zu sehen. Ohne



2.1 Näherungswerte für die untersten vier Energie-Eigenwerte des Quantenoszillators mit der Anharmonizität λq^4 in Abhängigkeit von λ.

Störung ($\lambda = 0$) erhalten wir die Energien $\frac{1}{2}$, $\frac{3}{2}$, $\frac{5}{2}$ und $\frac{7}{2}$ des harmonischen Oszillators. Mit wachsender Störung λ heben die oberen beiden Eigenwerte sehr stark von den unteren ab. Dies ist ein Effekt der endlichen Größe der Matrizen. Der Operator q^4 verknüpft den Zustand $|j\rangle$ mit $|j \pm 4\rangle$, und daher sollten die Matrixelemente $\langle j|q^4|j \pm 4\rangle$ in der Hamiltonmatrix `h[n]` enthalten sein, wenn die Energie ε_j einigermaßen genau berechnet werden soll. Auf jeden Fall ist es nicht verwunderlich, wenn die Eigenwerte $\varepsilon_n, \varepsilon_{n-1}$ und ε_{n-2} große Abweichungen vom exakten Resultat haben.

Wie gut ist die Näherung? Dies zeigt Abbildung 2.2 am Beispiel der Grundzustandsenergie ε_0. Für die Anharmonizität mit $\lambda = 0.1$ ist ε_0 als Funktion von $1/n$ aufgetragen, und zwar für $n = 7, ..., 20$. Diese Funktion ist offenbar nicht monoton. Obwohl wir ihr asymptotisches Verhalten nicht kennen, läßt sich der Wert für $n \to \infty$ sehr gut angeben. Mit den Befehlen



2.2 Näherungswerte für die Grundzustandsenergie des anharmonischen Oszillators als Funktion der Matrixdimension n für $n = 7, 8, \ldots, 20$, aufgetragen über $1/n$.

```
mat = N[h[n] /. lambda -> 1/10, 20];
 li = Sort[Eigenvalues[mat]]; li[[1]]
```

erhalten wir für $n = 20$ und $n = 40$ die Werte

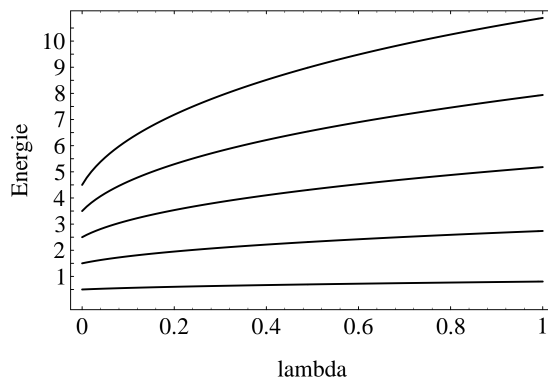
$$\begin{aligned}\varepsilon_0(20) &= 0.559146327396\ldots\,,\\ \varepsilon_0(40) &= 0.559146327183\ldots\,.\end{aligned}$$

Für $n = 20$ haben wir also eine Genauigkeit von etwa neun relevanten Stellen. Höhere Energien sind nur mit geringerer Genauigkeit bestimmt; wir erhalten

$$\begin{aligned}\varepsilon_{10}(20) &= 17.333\ldots\,,\\ \varepsilon_{10}(40) &= 17.351\ldots\,,\end{aligned}$$

also nur eine Genauigkeit von drei Stellen.

Im Bild 2.1 haben wir nur vier Niveaus für unsere Näherung berücksichtigt. Da-



2.3 Die fünf niedrigsten Energie-Eigenwerte als Funktion des Vorfaktors λ der Anharmonizität.

her sind die oberen zwei Energien völlig falsch dargestellt. Im Bild 2.3 dagegen haben wir zwanzig Niveaus mitgenommen. In diesem Fall gibt die numerische Lösung der Eigenwertgleichung ein sehr genaues Ergebnis für die fünf niedrigsten Energien. Sowohl die Energien als auch deren Abstände wachsen mit dem Vorfaktor λ der Anharmonizität.

Übung

Wir betrachten ein Quanten-Teilchen in einer Dimension im Doppelmuldenpotential. Der Hamiltonoperator sei in dimensionsloser Form

$$H = \frac{p^2}{2} - 2q^2 + \frac{q^4}{10} \, .$$

1. Zeichnen Sie das Potential.
2. Berechnen Sie die vier tiefsten Energieniveaus.
3. Zeichnen Sie die Wellenfunktionen der vier tiefsten Niveaus zusammen mit dem Potential.
4. Man kann auch die Matrixelemente von q^2 und q^4 direkt angeben, indem man die Operatoren analog zu (2.4) durch a und $a^\dagger$ ausdrückt. Verbessert das die Ergebnisse?

Literatur

J. Schnakenberg, *Algorithmen in der Quantentheorie und Statistischen Physik*, Zimmermann-Neufang, 1995.

F. Schwabl, *Quantenmechanik*, Springer Verlag, 1988.

2.2 Elektrisches Netzwerk

Das Ohmsche Gesetz ist eine lineare Beziehung zwischen Strom und Spannung. Es gilt sogar noch für monofrequente Wechselströme und Wechselspannungen, wenn Kondensatoren und Spulen durch komplexe Widerstände dargestellt werden. Auch für allgemeine passive elektrische Netzwerke bleiben die Gleichungen linear, wenn man für Ströme und Spannungen eine geeignete Darstellung im Raum der komplexen Zahlen wählt. Sie können bei vorgegebener Frequenz relativ leicht mit dem Computer gelöst werden, und mit Hilfe der Fouriertransformation kann dann zu jedem Eingangssignal die Ausgangsspannung berechnet werden.

Physik

Wir betrachten eine Wechselspannung $V(t)$, den zugehörigen Wechselstrom $I(t)$ und schreiben beide als komplexwertige Funktionen:

$$\begin{aligned} V(t) &= V_0\, e^{i\omega t}\,, \\ I(t) &= I_0\, e^{i\omega t}\,, \end{aligned} \tag{2.9}$$

wobei V_0 und I_0 komplexe Größen sind, deren Phasendifferenz gerade angibt, wie stark die Strom- der Spannungsschwingung vor- oder nacheilt. Eigentlich hat nur der Realteil von (2.9) eine physikalische Bedeutung, aber die Phasenbeziehungen sind in komplexer Schreibweise besonders leicht zu formulieren. Das Ohmsche Gesetz lautet damit

$$V_0 = Z\, I_0 \tag{2.10}$$

mit einem komplexen Widerstand Z. Für einen ohmschen Widerstand R, für eine Kapazität C und für eine Induktivität L gilt

$$Z = R\,,\ Z = \frac{1}{i\omega C}\,,\ Z = i\omega L\,, \tag{2.11}$$

wobei R, C und L reelle Größen sind, z. B. in den Einheiten *Ohm, Farad* und *Henry*. Für ein elektrisches Netzwerk gelten die folgenden Erhaltungssätze, auch als Kirchhoffsche Regeln bekannt:

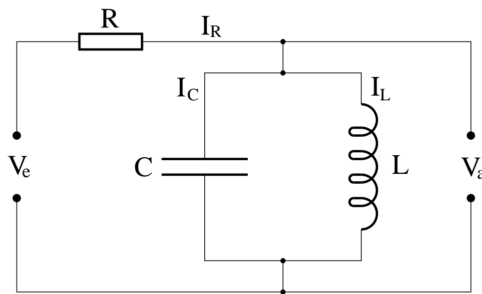
1. Wegen der Ladungserhaltung ist an jedem Knoten die Summe der einlaufenden Ströme ist gleich der Summe der auslaufenden Ströme.

2. Entlang eines jeden Weges addieren sich die Teilspannungen über jedem Bauelement zu der Gesamtspannung über den Weg.

Diese beiden Bedingungen ergeben zusammen mit dem Ohmschen Gesetz ein Gleichungssystem, das alle unbekannten Ströme und Spannungen bestimmt.

Als einfaches Beispiel betrachten wir einen L-C-Schwingkreis, der mit einem Widerstand R in Serie geschaltet ist (Bild 2.4). V_e und V_a seien die komplexen Amplituden der Eingangs- und Ausgangswechselspannung mit der Kreisfrequenz ω, und $I_\mathrm{R}, I_\mathrm{C}$ und I_L seien die Amplituden der Ströme, die nach dem Einschwingen dieselbe Frequenz ω wie die Eingangsspannung haben. Damit gelten folgende Gleichungen:

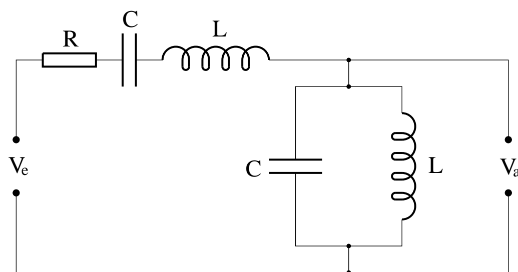
$$\begin{aligned} \text{Spannungsaddition} &: V_\mathrm{R} + V_\mathrm{a} = V_\mathrm{e}\,, \\ \text{Stromerhaltung} &: I_\mathrm{R} = I_\mathrm{C} + I_\mathrm{L}\,, \\ \text{Ohmsches Gesetz} &: V_\mathrm{R} = R\, I_\mathrm{R}\,, \\ & \quad V_\mathrm{a} = \frac{1}{i\omega C} I_\mathrm{C}\,, \\ & \quad V_\mathrm{a} = i\omega L\, I_\mathrm{L}\,. \end{aligned} \tag{2.12}$$



2.4 Serienschaltung von ohmschem Widerstand und L-C-Schwingkreis.

Für eine gegebene Eingangsspannung V_e bestimmen diese 5 Gleichungen die 5 Unbekannten $V_\mathrm{R}, V_\mathrm{a}, I_\mathrm{R}, I_\mathrm{C}$ und I_L, die man in diesem einfachen Fall leicht ohne Computer lösen kann. Der Betrag der Ausgangsspannung V_a nimmt unabhängig von R jeweils ein Maximum bei $\omega = \frac{1}{\sqrt{LC}}$ an; bei dieser Frequenz ist der Widerstand des Schwingkreises unendlich groß.

Im Bild 2.5 haben wir die Schaltung erweitert und einen Serienkreis hinzugefügt. Wenn ein Kondensator C und eine Induktivität L in Reihe geschaltet sind, so ist der Widerstand bei der Frequenz $\omega = \frac{1}{\sqrt{LC}}$ minimal. Daher erwarten wir für diese Schaltung bei dieser Frequenz eine maximale Ausgangsspannung. Dieses Netzwerk



2.5 Serienschaltung von R-C-L-Glied und L-C-Schwingkreis.

wird durch die folgenden zwei Gleichungen beschrieben.

$$\begin{aligned} \text{Spannungsaddition:} \quad & I_\mathrm{R}\left(R + \tfrac{1}{i\omega C} + i\omega L\right) + V_\mathrm{a} = V_\mathrm{e}\,, \\ \text{Stromerhaltung:} \quad & I_\mathrm{R} = \left(i\omega C + \tfrac{1}{i\omega L}\right) V_\mathrm{a}\,, \end{aligned} \tag{2.13}$$

wobei das Ohmsche Gesetz schon eingefügt wurde. Überraschenderweise ist das Ergebnis für $V_\mathrm{a}(\omega)$ ganz anders als das, was wir (als Nichtelektroniker) erwartet haben. Es entstehen nämlich zwei neue Resonanzen ober- und unterhalb von $\omega = \frac{1}{\sqrt{LC}}$, dort wird V_a bei kleinen Widerständen R sehr viel größer als V_e. Das kann man sich durch folgende Überlegung plausibel machen: Für kleine Frequen-

zen wird das Verhalten des Parallelkreises durch die Induktivität dominiert. Vernachlässigen wir die Kapazität im Parallelkreis einmal völlig, so haben wir es mit einer Serienschaltung aus den Elementen R-C-L-L zu tun. Weil die gesamte Induktivität jetzt $2L$ beträgt, erhalten wir eine Resonanz bei $\omega_1 = \frac{1}{\sqrt{2LC}}$. Wird dagegen für hohe Frequenzen im Parallelkreis nur die Kapazität betrachtet, so ergibt sich eine R-C-L-C Serienschaltung mit einer Gesamtkapazität von $C/2$ und dementsprechend eine Resonanzfrequenz von $\omega_2 = \sqrt{\frac{2}{LC}}$. Im Ergebnisteil werden wir diese Näherung mit den exakten Resultaten vergleichen.

Bisher haben wir nur monofrequente sinusförmige Spannungen und Ströme betrachtet. Nun wollen wir ein beliebiges periodisches Eingangssignal $V_{\text{e}}(t)$ an die Schaltung anlegen. Das Netzwerk ist linear, also erhält man aus überlagerten Eingangssignalen die entsprechend überlagerten Ausgangsspannungen. Insbesondere kann man jede periodische Eingangsspannung $V_{\text{e}}(t)$ mit der Periode T in eine Fourierreihe entwickeln:

$$V_{\text{e}}(t) = \sum_{n=-\infty}^{\infty} V_{\text{e}}^{(n)} \, e^{2\pi\, i\, n\, t/T} \; . \tag{2.14}$$

Für jeden Summanden mit der Amplitude $V_{\text{e}}^{(n)}$ erhält man aus der Gleichung (2.13) für die Frequenz $\omega_n = 2\pi\, n/T$ eine Ausgangsspannung $V_{\text{a}}(\omega_n) = V_{\text{a}}^{(n)}$, so daß das gesamte Ausgangssignal gegeben ist durch

$$V_{\text{a}}(t) = \sum_{n=-\infty}^{\infty} V_{\text{a}}^{(n)} \, e^{2\pi\, i\, n\, t/T} \; . \tag{2.15}$$

Algorithmus

In *Mathematica* können die Gleichungen (2.12) und (2.13) direkt eingegeben werden. Obwohl man beide Systeme sofort per Hand lösen kann, wollen wir das Prinzip doch an den einfachen Beispielen demonstrieren. Mit der Normierung $V_{\text{e}} = 1$ lauten die Gleichungen (2.12) daher

```
gl1 = {vr + va == 1, ir == ic + il, vr == ir r,
       va == ic/(I omega c), va == I omega L il}
```

Man beachte, daß das erste Gleichheitszeichen = eine Zuordnung bedeutet, während == einen logischen Ausdruck ergibt. Der Variablen `gl1` wird also eine Liste von Gleichungen zugeordnet. Mit

```
Solve[gl1, {va, vr, ir, ic, il}]
```

wird das Gleichungssystem nach den angegebenen Variablen aufgelöst. Weil Gleichungssysteme im allgemeinen mehrere Lösungen haben, liefert `Solve` eine Liste mit Listen von Regeln. Da es hier aber nur eine Lösung gibt, greifen wir mit `Solve[...][[1]]` davon die erste – und in diesem Fall die einzige – heraus.

Als Anwendung für eine nicht-sinusförmige Eingangsspannung $V_\mathrm{e}(t)$ wählen wir eine Sägezahnspannung mit der Periode T, die wir an N äquidistanten Zeitpunkten abtasten, um die diskrete Fouriertransformation benutzen zu können. Wir definieren also diskrete Spannungswerte durch

$$a_r = V_\mathrm{e}(t_r) \equiv V_\mathrm{e}\left((r-1)\,T/N\right), \quad r = 1, ..., N,$$

und erhalten durch Anwendung der inversen Fouriertransformation die Koeffizienten b_s mit der Eigenschaft

$$a_r = \frac{1}{\sqrt{N}} \sum_{s=1}^{N} b_s \, e^{2\pi i (s-1)(r-1)/N}$$

bzw.

$$V_\mathrm{e}(t_r) = \frac{1}{\sqrt{N}} \sum_{s=1}^{N} b_s \, e^{i \frac{2\pi}{T}(s-1) t_r}.$$

Nicht die Fouriertransformation direkt sondern deren Inverses müssen wir hier benutzen, damit das Vorzeichen im Exponenten der e-Funktion mit Gleichung (2.9) übereinstimmt. Obwohl die Autoren von *Mathematica* erklären, sie wollten, was diese Vorzeichenwahl bei der Fouriertransformation angeht, der Konvention der Physiker folgen, haben sie doch genau das Gegenteil verwirklicht.

Die Amplitude $b_s/\sqrt{N}$ gehört also zur Frequenz $\omega_s = \frac{2\pi}{T}(s-1)$. Am Ausgang wird jede Amplitude b_s mit der Ausgangsspannung $V_\mathrm{a}(\omega_s)$ multipliziert, die wir vorher mit `Solve[...]` aus den obigen Gleichungen erhalten haben. Allerdings gilt das nur für $s = 1, ..., N/2$. Höhere Frequenzen geben eine schlechte Näherung für $V_\mathrm{e}(t)$, wie im Abschnitt 1.3 ausführlich gezeigt wurde. Man muß durch $b_s = b_{s-N}$ die hohen Frequenzen in niedrige negative verschieben, bevor man mit $V_\mathrm{a}(\omega_s)$ transformieren darf. Die transformierten Fourierkoeffizienten sind also

$$\begin{aligned} b_s^\mathrm{t} &= b_s \, V_\mathrm{a}(\omega_s)\,, \qquad s = 1, ..., \frac{N}{2}, \\ b_s^\mathrm{t} &= b_s \, V_\mathrm{a}(\omega_{s-N})\,, \;\; s = \frac{N}{2} + 1, ..., N. \end{aligned}$$

Die Rücktransformation, in diesem Fall also die Fouriertransformation, liefert damit das Ausgangssignal.

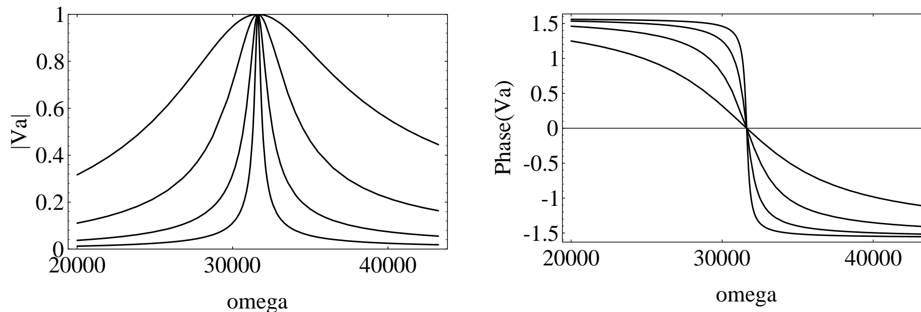
Ergebnisse

Die Lösung des Gleichungssystems (2.12), das die in der Abbildung 2.4 gezeigte Serienschaltung aus ohmschem Widerstand und L-C-Schwingkreis beschreibt, steht im *Mathematica*-Programm in der Variablen `vas` ; in lesbarer Form lautet sie

$$V_\mathrm{a}(\omega) = \frac{-i\omega L}{-i\omega L - R + RCL\omega^2} \,.$$

Offensichtlich hat das Netz eine Resonanz bei $\omega_\mathrm{r} = 1/\sqrt{LC}$, wie man auch in der Abbildung 2.6 für verschiedene R-Werte sieht.

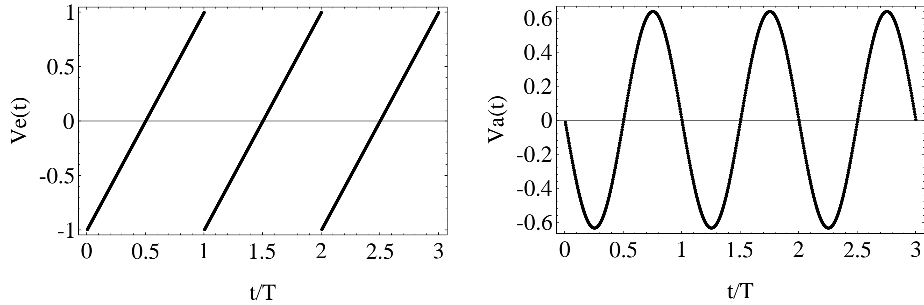
Wir haben $L = 1mH$ und $C = 1\mu F$ gewählt, was eine Resonanzfrequenz ω_r von 31622.8 s^{-1} ergibt. Für $R = 0$ erhält man $V_\mathrm{a}(\omega) = 1$, also keine Resonanz, während für $R \to \infty$ ein scharfes Spannungsmaximum an der Stelle der Resonanzfrequenz entsteht. Bei ω_r geht die Phase der Ausgangsspannung von $\frac{\pi}{2}$ nach $-\frac{\pi}{2}$.



2.6 Frequenzabhängigkeit von Betrag und Phase der Ausgangsspannung $V_\mathrm{a}(\omega)$ für das Netzwerk aus Abbildung 2.4. Die Kurven entsprechen den Widerständen $R = 100\,\Omega$, $300\,\Omega$, $900\,\Omega$, und $2700\,\Omega$. Je größer der Widerstand, um so schärfer ist die Resonanz.

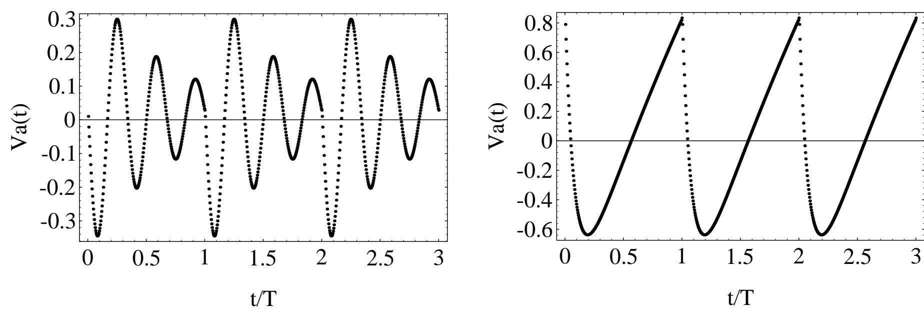
Legen wir nun die Sägezahnspannung $V_\mathrm{e}(t)$ an dieses Filter an, so wird das Ergebnis natürlich von der Schärfe der Resonanz bei ω_r und von dem Verhältnis der Grundfrequenz der Sägezahnspannung zur Resonanzfrequenz abhängen. Wir parametrisieren die Periode T von $V_\mathrm{e}(t)$ deshalb in der Form $\frac{2\pi}{T} = f\,\omega_\mathrm{r}$; f gibt also das Verhältnis von Eingangsgrundfrequenz zu Resonanzfrequenz an. Für $f = 1$ und einer entsprechend schmalen Resonanzkurve sollte aus der sägezahnförmigen Schwingung gerade eine Sinusschwingung mit der Frequenz $\omega = \omega_\mathrm{r}$ herausgefiltert werden.

Für $f < \frac{1}{2}$ und z. B. $R = 200\,\Omega$ wird im wesentlichen nur das Vielfache von $\frac{2\pi}{T}$ in der Nähe von ω_r herausgefiltert. Dagegen werden für $f > 1$ und breiter Resonanz



2.7 Links die ursprüngliche Sägezahnspannung $V_\mathrm{e}(t)$ und rechts die Spannung $V_\mathrm{a}(t)$ am Ausgang des Filters für $f = 1$ und $R = 2700\,\Omega$.

alle Harmonischen von $\frac{2\pi}{T}$ mitgenommen, es entsteht ein verzerrtes Eingangssignal. Diese Erwartung wird in den Abbildungen 2.7 und 2.8 bestätigt.



2.8 Links die Ausgangsspannung für $f = 1/3$ und $R = 200\,\Omega$, rechts das verzerrte Eingangssignal, das sich mit $f = 3$ und $R = 5\,\Omega$ ergibt.

Für das zweite Beispiel, die Serienschaltung von R-C-L-Glied und L-C-Schwingkreis, liefert das *Mathematica*-Programm, wiederum mit $V_\mathrm{e} = 1$, die Lösung:

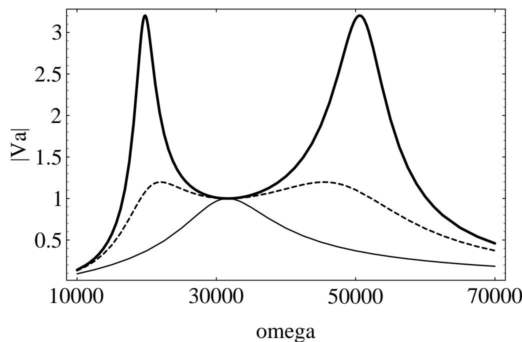
$$V_\mathrm{a} = \frac{CL\omega^2}{CL\omega^2 + (CL\omega^2 - 1)(1 - CL\omega^2 + iCR\omega)}$$

Für $R = 0$, wenn also der ohmsche Widerstand gleich null und damit die Schaltung

verlustfrei ist, verschwindet der Nenner bei

$$\omega = \sqrt{\frac{3 \pm \sqrt{5}}{2LC}}\,.$$

Mit den Werten $L = 1mH, C = 1\mu F$ divergiert $V_{\rm a}$ also bei den Frequenzen $\omega = 19544\ {\rm s}^{-1}$ und $\omega = 51166.7\ {\rm s}^{-1}$. Wir sehen, daß unsere vorherige Abschätzung der beiden Resonanzen bei $\omega_1 = 1/\sqrt{2LC} = 22360\,{\rm s}^{-1}$ bzw. $\omega_2 = \sqrt{2/LC} = 44720\,{\rm s}^{-1}$ nicht ganz falsch war.



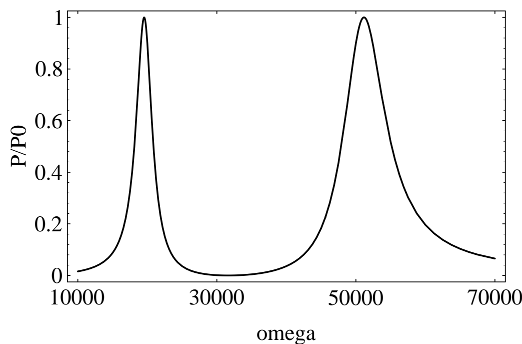
2.9 Betrag der Ausgangsspannung für das Netzwerk aus Serien- und Parallel-Kreis. Die zugehörigen Widerstandswerte sind von oben nach unten $R = 10\,\Omega$, $30\,\Omega$ und $90\,\Omega$.

Abbildung 2.9 zeigt $V_{\rm a}(\omega)$ für $R = 10, 30$ und $90\,\Omega$. Für jeden Wert des Widerstandes R ist bei der Frequenz $\omega = \frac{1}{\sqrt{LC}}$ die Ausgangsspannung gleich dem Eingangssignal, $V_{\rm a} = V_{\rm e}$. Erst bei kleinen Widerständen werden die beiden Resonanzen sichtbar.

Abschließend wollen wir die Leistung berechnen, die bei der Frequenz ω vom Widerstand R in Wärme umgesetzt wird. Auch hierbei erweist sich die komplexe Darstellung von Strom und Spannung als sehr vorteilhaft. Die Rechnung – zunächst für ein allgemeines Bauelement mit komplexem Widerstand Z, das vom Strom $I(t) = I_{\rm Z}\, e^{i\omega t}$ durchflossen wird und an dem die Spannung $V(t) = V_{\rm Z}\, e^{i\omega t}$ abfällt – verläuft folgendermaßen: Die Leistung P ist das Zeitmittel des Produktes aus den Realteilen des Stroms und der Spannung, also $P = \overline{{\rm Re}\ I(t)\ {\rm Re}\ V(t)}$. Wegen der rein harmonischen Zeitabhängigkeit gilt aber ${\rm Re}\ I(t) = {\rm Im}\ I(t + \frac{\pi}{2\omega})$ und analog für $V(t)$. Damit erhalten wir

$$\begin{aligned} P &= \frac{1}{2}\left(\overline{{\rm Re}\ I(t)\ {\rm Re}\ V(t)} + \overline{{\rm Im}\ I(t)\ {\rm Im}\ V(t)}\right) \\ &= \frac{1}{4}\left(\overline{I(t)\, V(t)^*} + \overline{I(t)^*\, V(t)}\right) \\ &= \frac{1}{4}\left(I_{\rm Z}\, V_{\rm Z}^* + I_{\rm Z}^*\, V_{\rm Z}\right) \\ &= \frac{1}{2}|I_{\rm Z}|^2\ {\rm Re}\ Z\,, \end{aligned} \tag{2.16}$$

wobei wir am Schluß $V_Z = Z\, I_Z$ eingesetzt haben. Für den ohmschen Widerstand R beträgt die Leistung demnach $P(\omega) = |I_R(\omega)|^2 R/2$. Wir vergleichen dies mit der Leistung P_0, die sich ergibt, wenn alle Spulen und Kondensatoren in unserer Schaltung überbrückt werden, wenn also die Eingangsspannung V_e direkt am Widerstand R anliegt, so daß man $I_R = V_e/R$ und damit $P_0 = |V_e|^2/(2R)$ erhält. Das Verhältnis der Leistungen $P(\omega)/P_0$ ist für $R = 10\,\Omega$ in der Abbildung 2.10 gezeigt. Bei den beiden Resonanzfrequenzen scheinen Spulen und Kondensatoren völlig durchlässig zu sein, während bei $\omega = \frac{1}{\sqrt{LC}}$ der Parallelkreis einen unendlichen Widerstand bietet.



2.10 Die Leistung $P(\omega)$, die am ohmschen Widerstand $R = 10\,\Omega$ verbraucht wird, bezogen auf diejenige Leistung P_0, die umgesetzt wird, wenn die Eingangsspannung direkt an R anliegt.

Übung

Fügen Sie im zweiten R-C-L-Netzwerk (Bild 2.5) einen ohmschen Widerstand R zum parallelen L-C-Kreis hinzu, und zwar parallel zu Kondensator und Spule.

1. Berechnen und zeichnen Sie $V_a(\omega)$.

2. Welche Form hat eine periodische Rechteckspannung nach dem Durchgang durch dieses Filter?

3. Wie groß ist die Verlustleistung an den beiden Widerständen als Funktion von ω?

Literatur

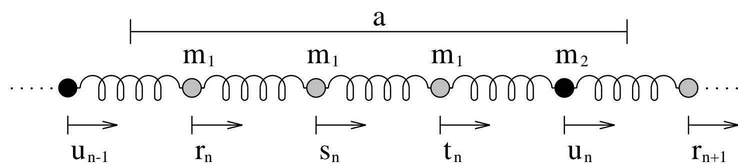
R. E. Crandall, *Mathematica for the Sciences*, Addison Wesley, 1991.

2.3 Kettenschwingungen

Die Bewegung eines Teilchens im quadratischen Potential wird bekanntlich durch eine besonders einfache lineare Differentialgleichung beschrieben. Wenn mehrere Teilchen durch solche linearen Kräfte miteinander wechselwirken, kann deren Bewegung ebenfalls aus linearen Bewegungsgleichungen berechnet werden. Allerdings erhält man dann mehrere solcher Gleichungen, die miteinander verkoppelt sind. Dieses lineare Gleichungssystem läßt sich durch Diagonalisierung einer Matrix lösen. Ein schönes Beispiel dafür ist die lineare Kette mit verschiedenen Massen. Sie ist ein einfaches Modell für die Berechnung von Gitterschwingungen in einem Kristall. Die Eigenwerte einer Matrix geben die Energiebänder der Phononen an, während die Eigenvektoren Aussagen über die Schwingungsformen der Kristallbausteine machen. Durch Überlagerung solcher Eigenschwingungen kann jede mögliche Bewegung des Modell-Festkörpers dargestellt werden.

Physik

Wir betrachten eine Kette, bestehend aus punktförmigen Massen m_1 und m_2, die wir der Einfachheit halber als *leichte* bzw. *schwere* Atome bezeichnen. Die Teilchen seien so angeordnet, daß auf jeweils drei leichte Atome ein schweres folgt. Die Elementarzelle der Länge a enthält also vier Atome. Es sollen nur nächste Nachbarn miteinander wechselwirken. Wir beschränken unsere Betrachtung auf kleine Auslenkungen, das heißt, die Kräfte sollen lineare Funktionen der Massenverschiebungen sein wie in dem nachfolgend abgebildeten Federmodell.



2.11 Lineare Kette, bestehend aus zwei Atomsorten, die durch elastische Kräfte untereinander verbunden sind.

Zur Beschreibung der longitudinalen Schwingungen numerieren wir die Einheitszellen fortlaufend und betrachten die Zelle mit der Nummer n. Innerhalb dieser Zelle seien r_n, s_n, t_n die Auslenkungen der leichten Atome aus der Ruhelage, und u_n sei die Auslenkung des schweren Atoms aus der Ruhelage. Dann lauten die Bewegungsgleichungen

$$m_1 \ddot{r}_n \quad = \quad f\,(s_n - r_n) - f\,(r_n - u_{n-1})$$

$$\begin{aligned} &= f\,(s_n + u_{n-1} - 2r_n)\,, \\ m_1\,\ddot{s}_n &= f\,(t_n + r_n - 2s_n)\,, \\ m_1\,\ddot{t}_n &= f\,(u_n + s_n - 2t_n)\,, \\ m_2\,\ddot{u}_n &= f\,(r_{n+1} + t_n - 2u_n). \end{aligned} \tag{2.17}$$

f ist die Federkonstante. In einer unendlich langen Kette gelten diese Gleichungen für jede Elementarzelle $n \in \mathbb{Z}$. Für die endlich lange Kette aus N Elementarzellen nehmen wir periodische Randbedingungen an, d. h. wir stellen uns eine ringförmige Anordnung aus N identischen Zellen vor. Da sich die Energie der Kette bei Verschiebung um die Länge a nicht ändert und folglich die Bewegungsgleichungen invariant sind gegenüber den Translationen $\{r_n, s_n, t_n, u_n\} \to \{r_{n+k}, s_{n+k}, t_{n+k}, u_{n+k}\}$, $k = 1, 2, \ldots, N$, kann eine Lösung der Gleichung (2.17) durch Fouriertransformation gefunden werden. Wir machen daher den Ansatz

$$\boldsymbol{x}_n(t) = \begin{pmatrix} r_n(t) \\ s_n(t) \\ t_n(t) \\ u_n(t) \end{pmatrix} = \boldsymbol{S}(q)\; e^{iqan \pm i\omega t}, \tag{2.18}$$

wobei wegen der periodischen Randbedingungen q nur die Werte $q_\nu = \frac{2\pi}{Na}\nu,\ \nu = -\frac{N}{2} + 1, \ldots, \frac{N}{2}$ annehmen kann. Dann gilt

$$\ddot{\boldsymbol{x}}_n = -\omega^2\,\boldsymbol{x}_n,\ \boldsymbol{x}_{n\pm 1} = e^{\pm iqa}\,\boldsymbol{x}_n\,. \tag{2.19}$$

Dies in Gleichung (2.17) eingesetzt ergibt

$$\begin{pmatrix} 2f & -f & 0 & -fe^{-iqa} \\ -f & 2f & -f & 0 \\ 0 & -f & 2f & -f \\ -fe^{iqa} & 0 & -f & 2f \end{pmatrix} \boldsymbol{S}(q) = \omega^2 \begin{pmatrix} m_1 & 0 & 0 & 0 \\ 0 & m_1 & 0 & 0 \\ 0 & 0 & m_1 & 0 \\ 0 & 0 & 0 & m_2 \end{pmatrix} \boldsymbol{S}(q), \tag{2.20}$$

also eine verallgemeinerte Eigenwertgleichung vom Typ

$$\mathsf{F}\,\boldsymbol{S} = \lambda\,\mathsf{M}\,\boldsymbol{S}\,. \tag{2.21}$$

Dabei ist M die Massenmatrix, und wir suchen bei vorgegebenem q die Eigenwerte $\lambda(q) = \omega^2(q)$ und die zugehörigen Normalmoden $\boldsymbol{S}(q)$. In unserem Fall läßt sich die Massenmatrix leicht invertieren, und Gleichung (2.21) ist offensichtlich äquivalent zu

$$\mathsf{M}^{-1}\,\mathsf{F}\,\boldsymbol{S} = \lambda\,\boldsymbol{S}\,. \tag{2.22}$$

Allerdings ist $\mathsf{M}^{-1}\,\mathsf{F}$ keine hermitesche Matrix. Daß die Eigenwerte λ dennoch reell und nichtnegativ sind, erkennt man durch folgende Umformung von Gleichung

(2.21). Wir multiplizieren sie von links mit $\mathsf{M}^{-\frac{1}{2}}$, der zu $\mathsf{M}^{\frac{1}{2}}$ inversen Matrix, und erhalten

$$\mathsf{M}^{-\frac{1}{2}}\,\mathsf{F}\,\mathsf{M}^{-\frac{1}{2}}\,\mathsf{M}^{\frac{1}{2}}\,\boldsymbol{S} = \lambda\,\mathsf{M}^{\frac{1}{2}}\,\boldsymbol{S}\,. \tag{2.23}$$

Das ist eine gewöhnliche Eigenwertgleichung für die hermitesche und positiv semidefinite Matrix $\mathsf{M}^{-\frac{1}{2}}\,\mathsf{F}\,\mathsf{M}^{-\frac{1}{2}}$ mit dem Eigenvektor $\mathsf{M}^{\frac{1}{2}}\,\boldsymbol{S}$.

Die so gewonnenen Eigenschwingungen $\boldsymbol{x}_n(t) = \boldsymbol{S}_\ell(q_\nu)\,e^{iq_\nu a\,n \pm i\omega_{\nu\ell}t}$ sind spezielle komplexwertige Lösungen der Bewegungsgleichungen (2.17). Analog zu den elektrischen Filtern (Abschnitt 2.2) erhält man die allgemeine Lösung durch Überlagerung der Eigenschwingungen. Die zu (2.20) konjugiert komplexe Gleichung liefert mit $q \to -q$, daß mit $\boldsymbol{S}(q)$ auch $\boldsymbol{S}(-q)^*$ Eigenvektor zu demselben Eigenwert ist. Die allgemeine reelle Lösung von (2.17) hat deshalb die Form

$$\boldsymbol{x}_n(t) = \sum_{\nu,\ell} \boldsymbol{S}_\ell(q_\nu)\,e^{iq_\nu an}\,\left(c_{\nu\ell}\,e^{i\omega_{\nu\ell}t} + c^*_{-\nu\ell}\,e^{-i\omega_{\nu\ell}t}\right). \tag{2.24}$$

Die noch freien Koeffizienten $c_{\nu\ell}$ werden durch die Anfangsbedingungen festgelegt.

Algorithmus und Ergebnisse

Die 4×4-Matrix (2.22) stellt keine besondere Herausforderung an eine analytische Lösung der Eigenwertgleichung dar. Aber bei mehreren Atomsorten oder dem entsprechenden zweidimensionalen Problem werden die Matrizen so groß, daß nur noch eine numerische Bestimmung der Schwingungsmoden möglich ist. Letzteres wollen wir an unserem einfachen Beispiel demonstrieren.

Zunächst schreiben wir die Matrizen F und M in *Mathematica*-Form, wobei $a = 1$ gewählt wurde,

```
mat1 = { { 2f          , -f ,  0  , -f*Exp[-I q]},
         { -f          , 2f , -f  ,  0          },
         {  0          , -f , 2f  , -f          },
         { -f*Exp[I q],  0 , -f  , 2f          } }

massmat = DiagonalMatrix[{m1,m1,m1,m2}]
```

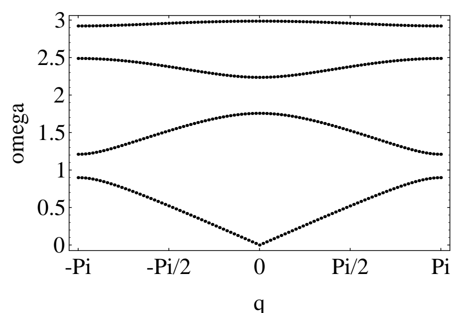
Damit bilden wir das Produkt `mat2 = Inverse[massmat].mat1`. *Mathematica* ist zwar in diesem Fall noch in der Lage, mit `Eigenvalues[mat2]` die Eigenwerte allgemein anzugeben, die Rechnung ist aber langwierig und das Ergebnis besteht aus verschachtelten zweiten und dritten Wurzeln, so daß wir eine numerische Berechnung vorgezogen haben.

```
eigenlist = Table[{x, Chop[Eigenvalues[
             mat2 /.{f->1.,m1->0.4,m2->1.0,q->x}]]},
                    {x,-Pi,Pi,Pi/50}]
```

Dieser Befehl liefert in kurzer Zeit eine Liste von q-Werten und den zugehörigen vier Frequenzquadraten.

```
Flatten[ Table[
         Map[{#[[1]],Sqrt[#[[2,k]]]}&, eigenlist],
         {k,4}], 1]
```

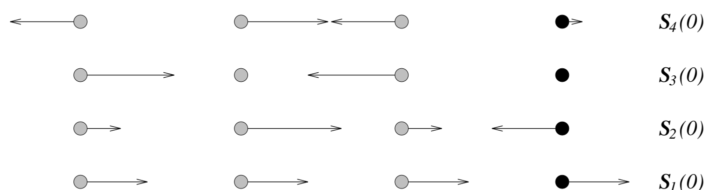
macht daraus eine Liste von (q-ω)-Paaren, die in Abbildung 2.12 mit `ListPlot` gezeichnet wurden. Es entstehen also vier Bänder von erlaubten Frequenzen der Git-



2.12 Wellenzahl q und die zugehörigen vier Frequenzen der Eigenschwingungen der linearen Kette aus Bild (2.11).

terschwingungen. Der unterste Zweig stellt die sogenannten akustischen Phononen dar, bei denen benachbarte Atome fast immer in die gleiche Richtung schwingen. Für $q = 0$ wird die gesamte Kette nur als Ganzes verschoben, was keine Energie kostet ($\omega = 0$). Bei den drei oberen Zweigen, den optischen Phononen, schwingen viele Atome gegeneinander; das kostet auch bei $q = 0$ Energie.

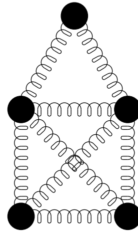
Die einzelnen Auslenkungen kann man an den Eigenvektoren der Matrix $\mathbf{M}^{-1}\mathbf{F}$ ablesen. Abbildung 2.13 zeigt die vier Eigenschwingungen für $q = 0$. Überraschenderweise bleiben im dritten Zweig das schwere und ein leichtes Atom stehen, während die beiden anderen Massen gegeneinander schwingen.



2.13 Eigenschwingungen der Kette für $q = 0$.

Übung

Betrachten Sie die zweidimensionalen Vibrationen Ihres Einfamilienhauses. Fünf gleiche Massen m sind durch Spiralfedern gekoppelt, und das Potential zwischen



benachbarten Massen sei

$$V(\boldsymbol{r}_i, \boldsymbol{r}_j) = \frac{D}{2} \left(|\boldsymbol{r}_i - \boldsymbol{r}_j| - a_{ij}\right)^2 ,$$

wobei a_{ij} der jeweilige Ruheabstand sein soll, also $a_{ij} = l$ für die Kanten und das Dach und $a_{ij} = \sqrt{2}\, l$ für die beiden Diagonalen.
Berechnen Sie für kleine Auslenkungen die Frequenzen der Vibrationen und zeichnen Sie die Eigenschwingungen. Besonders eindrucksvoll ist es, wenn Sie eine Animation der Eigenschwingungen herstellen.

Literatur

H. Goldstein, *Classical Mechanics*, Addison Wesley, 1980.
W. Ludwig, *Festkörperphysik*, Akademische Verlagsgesellschaft, 1978.

2.4 Hofstadter-Schmetterling

Lineare Gleichungen können faszinierende Lösungen haben. Das wird besonders deutlich beim Kristall-Elektron im homogenen magnetischen Feld. Das Spektrum der Energien des Elektrons hat als Funktion der Stärke des Magnetfeldes eine komplizierte Struktur, die an einen Schmetterling erinnert. Es war Douglas Hofstadter, der 1976 dieses Problem untersucht hat. Erstaunlicherweise werden in diesem Spektrum die Unterschiede zwischen rationalen und irrationalen Zahlen sichtbar. Wir wollen die quantenmechanische Gleichung für die Energien des Elektrons herleiten und numerisch lösen. Ein kleines Computerprogramm liefert den faszinierenden Schmetterling.

Physik

Wir modellieren ein Elektron auf einem quadratischen Gitter durch lokalisierte Wellenfunktionen und Hüpfmatrixelemente zwischen nächsten Nachbarn. In solcher *tight-binding* Näherung haben die Einteilchenenergien die Form

$$\varepsilon(\boldsymbol{k}) = \varepsilon_0(\cos k_x a + \cos k_y a)\,, \tag{2.25}$$

wobei a die Gitterkonstante, $2\varepsilon_0$ die Breite des Energiebandes und $\boldsymbol{k} = (k_x, k_y)$ der Wellenvektor eines Elektrons ist. Das Magnetfeld wird durch den *Peierls-Trick* berücksichtigt: $\hbar\boldsymbol{k}$ wird durch $\boldsymbol{p} + \frac{e}{c}\boldsymbol{A}$ ersetzt, wobei $\boldsymbol{A}$ das Vektorpotential und $\boldsymbol{p}$ der Impulsoperator ist. Für ein Magnetfeld vom Betrag B, das senkrecht zur x-y-Ebene steht, kann man $\boldsymbol{A} = (0, B\,x, 0)$ wählen. Damit erhält man die Schrödinger-Gleichung

$$\left[\varepsilon_0 \cos\left(\frac{ap_x}{\hbar}\right) + \varepsilon_0 \cos\left(\frac{ap_y}{\hbar} + \frac{aeBx}{\hbar c}\right)\right] \varphi(x,y) = E\,\varphi(x,y)\,. \tag{2.26}$$

Wegen $\cos\alpha = \frac{1}{2}\left(e^{i\alpha} + e^{-i\alpha}\right)$ und weil $e^{iap_x/\hbar}$ der Translationsoperator ist, der um eine Gitterkonstante a verschiebt, findet man:

$$\begin{aligned} &\frac{1}{2}\varepsilon_0\,[\varphi(x+a,y) + \varphi(x-a,y) + \\ &\quad e^{iaeBx/\hbar c}\,\varphi(x,y+a) + e^{-iaeBx/\hbar c}\,\varphi(x,y-a)] = E\,\varphi(x,y)\,. \end{aligned} \tag{2.27}$$

Die kontinuierliche Wellengleichung wird also zur diskreten Gleichung, die $\varphi(x,y)$ mit den Amplituden an den vier Nachbarplätzen koppelt. Für die y-Abhängigkeit der Wellenfunktion setzen wir eine ebene Welle an:

$$\varphi(x,y) = e^{i\nu y/a}\,\psi(x/a)\,, \tag{2.28}$$

und mit den dimensionslosen Variablen $m = x/a$ und $\sigma = a^2 eB/hc$ erhält man schließlich für $\psi_m \equiv \psi(m)$ die *Harper-Gleichung*

$$\psi_{m-1} + 2\cos(2\pi m\sigma + \nu)\,\psi_m + \psi_{m+1} = E\,\psi_m\,. \tag{2.29}$$

Dabei wird die Energie E in Einheiten von $\varepsilon_0/2$ gemessen. Diese diskrete Schrödinger-Gleichung enthält nur noch die Parameter ν und σ. ν bestimmt den Impuls in y-Richtung, während σ der Quotient aus magnetischem Fluß a^2B durch die Einheitszelle und einem Flußquantum hc/e ist.

Ein zweidimensionales Gitterelektron im Magnetfeld wurde also auf ein Teilchen abgebildet, das auf einer Kette im periodischen Cosinus-Potential springt. ν ist die Phase des Potentials und spielt keine große Rolle, während σ das Verhältnis der Periodenlänge zur Gitterkonstanten ($= 1$ in den m-Variablen) angibt. Hier sieht man

schon den Unterschied zwischen rationalem und irrationalem σ-Wert: Ist σ rational, also $\sigma = p/q$ mit ganzen Zahlen p und q, dann ist das Potential kommensurabel zum Gitter mit der Periode q. Die Wellenfunktion hat in diesem Fall nach dem Bloch-Theorem die Eigenschaft

$$\psi_{m+q} = e^{ikq}\, \psi_m\,. \tag{2.30}$$

Die Eigenzustände und die zugehörigen Energien werden durch k und ν klassifiziert. Schreibt man Gleichung (2.29) in folgender Weise in Matrixform:

$$\begin{pmatrix} \psi_{m+1} \\ \psi_m \end{pmatrix} = \mathbf{M}_1(m,E) \begin{pmatrix} \psi_m \\ \psi_{m-1} \end{pmatrix} \tag{2.31}$$

mit

$$\mathbf{M}_1(m,E) = \begin{pmatrix} E - 2\cos(2\pi m\sigma + \nu) & -1 \\ 1 & 0 \end{pmatrix}, \tag{2.32}$$

so zeigt die q-fach iterierte Gleichung zusammen mit (2.30), nämlich

$$\prod_{r=1}^{q} \mathbf{M}_1(q-r,E) \begin{pmatrix} \psi_0 \\ \psi_{-1} \end{pmatrix} = e^{ikq} \begin{pmatrix} \psi_0 \\ \psi_{-1} \end{pmatrix}, \tag{2.33}$$

daß e^{ikq} Eigenwert der Matrix $\mathbf{M}_q(E) = \prod_{r=1}^{q} \mathbf{M}_1(q-r,E)$ ist. Weil $\det(\mathbf{M}_q) = 1$ ist, ergibt sich der zweite Eigenwert von $\mathbf{M}_q$ zu e^{-ikq}. Daraus folgt für die Spur von $\mathbf{M}_q$:

$$\mathrm{Spur}\, \mathbf{M}_q(E) = 2\cos kq\,. \tag{2.34}$$

Die Spur der Matrix $\mathbf{M}_q(E)$ ist ein Polynom q-ten Grades in E. Dementsprechend hat die Gleichung (2.34) höchstens q reelle Lösungen $E(k,\nu)$. Wir schließen hieraus, daß es maximal q Energiebänder gibt, die man im Prinzip aus (2.34) für rationales $\sigma = p/q$ berechnen kann. Neben jeder rationalen Zahl gibt es aber Zahlen mit beliebig großem q und auch irrationale Zahlen $(q \to \infty)$, so daß es als Funktion von σ neben jedem Spektrum mit wenigen Bändern auch Spektren mit beliebig vielen Energiebändern gibt. Hier zeigt sich also physikalisch der Unterschied zwischen rationalen und irrationalen Zahlen.

Algorithmus

Die numerische Auswertung der Spurbedingung (2.34), die auf eine Nullstellenbestimmung eines Polynoms q-ten Grades hinausläuft, erweist sich als ziemlich schwierig. Wir wählen deshalb einen etwas anderen Weg, der dieses Problem umgeht. Für einen festen Wert von k und ν, und zwar für $k = 0$ und $\nu = 0$, wollen wir die Harper-Gleichung (2.29) numerisch lösen und das Ergebnis graphisch als

Funktion von rationalen Werten $\sigma = p/q$ darstellen. Dazu wählen wir ein festes q, das eine gerade Zahl sein soll, und lassen p von 1 bis $q-1$ laufen. Wir nutzen aus, daß der Potentialterm $V_m = 2\cos(2\pi m p/q)$ in m gerade ist, so daß man von den Eigenvektoren $\{\psi_m\}_{m=0}^{q-1}$ der Harper-Gleichung voraussetzen kann, daß sie entweder gerade oder ungerade in m sind. Die Periodizität $\psi_{m+q} = \psi_m$ reduziert in der Tat das Eigenwertproblem (2.29) auf q unabhängige Komponenten. Durch die erwähnte gerade-ungerade Symmetrie läßt sich dieses Problem auf zwei etwa halb so große Eigenwertaufgaben zurückführen.

Die Voraussetzung, $\{\psi_m\}$ sei z. B. ungerade in m, also $\psi_{-m} = -\psi_m$, liefert uns zunächst $\psi_0 = 0$ und zusammen mit der Periodizität

$$\psi_{\frac{q}{2}+r} = -\psi_{-\frac{q}{2}-r} = -\psi_{\frac{q}{2}-r} \quad \text{für } r = 0, 1, \ldots, \frac{q}{2} - 1\,. \tag{2.35}$$

Insbesondere ist also auch $\psi_{\frac{q}{2}} = 0$, und für die verbleibenden $\frac{q}{2} - 1$ unabhängigen Komponenten $(\psi_1, \psi_2, \ldots, \psi_{\frac{q}{2}-1})^{\mathsf{T}} \equiv \boldsymbol{\psi}_u$ lautet die Gleichung (2.29) in Matrixform:

$$\mathbf{A}_u\,\boldsymbol{\psi}_u = E\,\boldsymbol{\psi}_u \quad \text{mit} \quad \mathbf{A}_u = \begin{pmatrix} V_1 & 1 & 0 & \cdots & 0 & 0 \\ 1 & V_2 & 1 & \ddots & & 0 \\ 0 & 1 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 1 & 0 \\ 0 & & \ddots & 1 & V_{\frac{q}{2}-2} & 1 \\ 0 & 0 & \cdots & 0 & 1 & V_{\frac{q}{2}-1} \end{pmatrix} \tag{2.36}$$

Die möglichen Energien der Harper-Gleichung zu bestimmen bedeutet also in diesem Fall, die Eigenwerte der Tridiagonalmatrix $\mathbf{A}_u$ zu berechnen. Bevor wir auf ein spezielles numerisches Verfahren eingehen, das sich hier als besonders geeignet anbietet, wollen wir kurz zeigen, daß auch die Behandlung der in m geraden Wellenfunktionen ψ_m auf ein ganz ähnliches Problem führt.

Es sei also $\psi_{-m} = \psi_m$ und $\psi_{m+q} = \psi_m$. Dies impliziert zunächst $\psi_{-1} = \psi_1$, und analog zur Gleichung (2.35) erhalten wir jetzt

$$\psi_{\frac{q}{2}+r} = \psi_{-\frac{q}{2}-r} = \psi_{\frac{q}{2}-r} \quad \text{für } r = 1, 2, \ldots, \frac{q}{2} - 1\,. \tag{2.37}$$

Inbesondere gilt also $\psi_{\frac{q}{2}+1} = \psi_{\frac{q}{2}-1}$. Die Matrixform der Harper-Gleichung (2.29) für die $\frac{q}{2} + 1$ unabhängigen Komponenten $(\psi_0, \psi_1, \ldots, \psi_{\frac{q}{2}})^{\mathsf{T}} \equiv \boldsymbol{\psi}_g$ sieht demnach

folgendermaßen aus:

$$\mathsf{A}_g\,\boldsymbol{\psi}_g = E\,\boldsymbol{\psi}_g \quad \text{mit} \quad \mathsf{A}_g = \begin{pmatrix} V_0 & 2 & 0 & \cdots & 0 & 0 \\ 1 & V_1 & 1 & \ddots & & 0 \\ 0 & 1 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 1 & 0 \\ 0 & & \ddots & 1 & V_{\frac{q}{2}-1} & 1 \\ 0 & 0 & \cdots & 0 & 2 & V_{\frac{q}{2}} \end{pmatrix} \tag{2.38}$$

Wie im Fall der ungeraden ψ_m bleibt als Aufgabe, die Eigenwerte einer tridiagonalen Matrix, hier diejenigen von A_g zu bestimmen.

Die Berechnung des charakteristischen Polynoms einer tridiagonalen Matrix läßt sich auf folgende Weise rekursiv formulieren. Wir betrachten z. B. die Matrix $\boldsymbol{A}_u - E\,\mathbf{1}$ mit $\boldsymbol{A}_u$ aus Gleichung (2.36) und davon wiederum die linke obere Ecke, die aus n Zeilen und Spalten bestehen möge. Bezeichnen wir deren Determinante mit $p_n(E)$, so erhalten wir durch Entwickeln nach der letzten Zeile

$$p_n(E) = (V_n - E)\,p_{n-1}(E) - p_{n-2}(E)\,. \tag{2.39}$$

Man vervollständigt diese zweigliedrige Rekursionsformel duch die Anfangswerte

$$p_1(E) = V_1 - E \quad \text{und} \quad p_0(E) = 1 \tag{2.40}$$

und kann auf diese Weise $\det(\boldsymbol{A}_u - E\,\mathbf{1}) = p_{\frac{q}{2}-1}(E)$ rekursiv berechnen.

Die besondere Bedeutung der Polynome $p_n(E)$ liegt aber darin, daß sie eine sogenannte *Sturmsche Kette* bilden: Sturmsche Ketten enthalten Informationen über die Nullstellen des Polynoms mit dem höchsten Grad. Wenn man die Anzahl der Vorzeichenwechsel $w(E)$ betrachtet, die in der Folge

$$p_0(E),\; p_1(E),\; p_2(E), \ldots,\; p_{\frac{q}{2}-1}(E)$$

an der Stelle E auftreten, dann gilt:

Die Anzahl der Nullstellen von $p_{\frac{q}{2}-1}(E)$
im Intervall $E_1 \le E < E_2$ ist gleich $w(E_2) - w(E_1)$.

Weil es bei einer graphischen Darstellung immer eine endliche Auflösung gibt, auf dem Bildschirm z. B. die Pixelbreite, werden wir eine geeignete Rasterung $E_0 < E_1 < E_2 \cdots < E_N$ des überhaupt möglichen Energiebereichs vornehmen. Die Zahl der Vorzeichenwechsel $w(E_i)$, bzw. $w(E_{i+1})$ sagt uns dann, ob zwischen E_i und E_{i+1} Eigenwerte liegen. Der Bereich, in dem überhaupt Energie-Eigenwerte

zu finden sind, läßt sich durch den Satz von Gerschgorin eingrenzen: Für jeden Eigenwert E einer $n \times n$-Matrix $\mathbf{A} = (a_{ij})$ gibt es ein i, so daß gilt:

$$|E - a_{ii}| \leq \sum_{j=1,\, j\neq i}^{n} |a_{ij}| \,. \tag{2.41}$$

Weil in unserem Fall $\sum_{j=1,\, j\neq i}^{n} |a_{ij}| \leq 2$ und $|a_{ii}| \equiv |V_i| \leq 2$ gilt, schließen wir, daß die Eigenwerte zwischen -4 und 4 liegen.

Für die Matrix $\mathbf{A}_g$, die sich leicht von $\mathbf{A}_u$ unterscheidet, lautet der den Gleichungen (2.39, 2.40) entsprechende Algorithmus:

$$\begin{aligned} p_0(E) = 1 \ , \ p_1(E) = V_0 - E \ , \ p_2(E) = (V_1 - E)\, p_1(E) - 2\, p_0(E) \,, \\ p_n(E) = (V_{n-1} - E)\, p_{n-1}(E) - p_{n-2}(E) \ \text{ für } \ n = 3, 4, \ldots, \tfrac{q}{2} \,, \\ p_{\frac{q}{2}+1}(E) = (V_{\frac{q}{2}} - E)\, p_{\frac{q}{2}}(E) - 2\, p_{\frac{q}{2}-1}(E) \,. \end{aligned} \tag{2.42}$$

Wir wollen noch darauf hinweisen, daß die Gleichungen (2.39) bzw. (2.42) für die Sturmschen Ketten fast identisch zur Harper-Gleichung selbst sind. Denn aus (2.29) folgt für die ungeraden Wellenfunktionen $\psi_{-m} = -\psi_m$ mit den Startwerten $\psi_0 = 0,\ \psi_1 = 1$:

$$\psi_2 = (E - V_1) \ , \quad \psi_3 = (E - V_2)\, \psi_2 - 1 \,, \quad \psi_4 = (E - V_3)\, \psi_3 - \psi_2 \,, \ldots$$

Die Wahl von $\psi_1 = 1$ ist beliebig und ändert höchstens die Normierung der Wellenfunktion, nicht aber deren Eigenwert. $\psi_0 = 0$ folgt aus der Symmetrie. Der Vergleich mit (2.39) und (2.40) ergibt

$$\psi_1 = p_0(E) \ , \quad \psi_2 = -p_1(E) \,, \quad \psi_3 = p_2(E) \ , \quad \psi_4 = -p_3(E) \,, \ldots$$

Ändert sich nun die Anzahl der Vorzeichenwechsel in der Folge $\{p_n(E)\}$, so ändert sich die Anzahl der Vorzeichenwechsel in der Folge $\{\psi_n\}$ ebenfalls, und zwar um denselben Betrag. Die Anzahl der Eigenwerte im Intervall $[E_1, E_2]$ ist also durch den Unterschied in der Anzahl der Vorzeichenwechsel von $\{\psi_n(E_1)\}$ und $\{\psi_n(E_2)\}$ bestimmt. Dies erinnert an den Knotensatz der eindimensionalen Quantenmechanik, den wir im Abschnitt 4.3 noch beim anharmonischen Oszillator benutzen: Die Anzahl der Vorzeichenwechsel einer stationären Wellenfunktion $\psi(x)$ bestimmt die Nummer des Energieniveaus über dem Grundzustand.

Der obige Algorithmus läßt sich in wenigen Programmzeilen schreiben. Wir haben in der Abbildung 2.14 eine Diskretisierung des kompletten Energieintervalls $|E| \leq 4$ in 500 Intervalle gewählt. Nach rechts ist der Parameter $\sigma = p/q$ für $q = 500$ und $p = 1, 2, \ldots, 499$ aufgetragen. Die Gesamtzahl der Rechenschritte läßt sich mit etwa $500 \times 500 \times 500 \times 10 \simeq 10^9$ abschätzen, wobei wir zehn Rechenoperationen für die innerste Schleife angenommen haben. Weil *Mathematica* dafür zu langsam wäre, haben wir das Programm in C geschrieben. Es besteht –

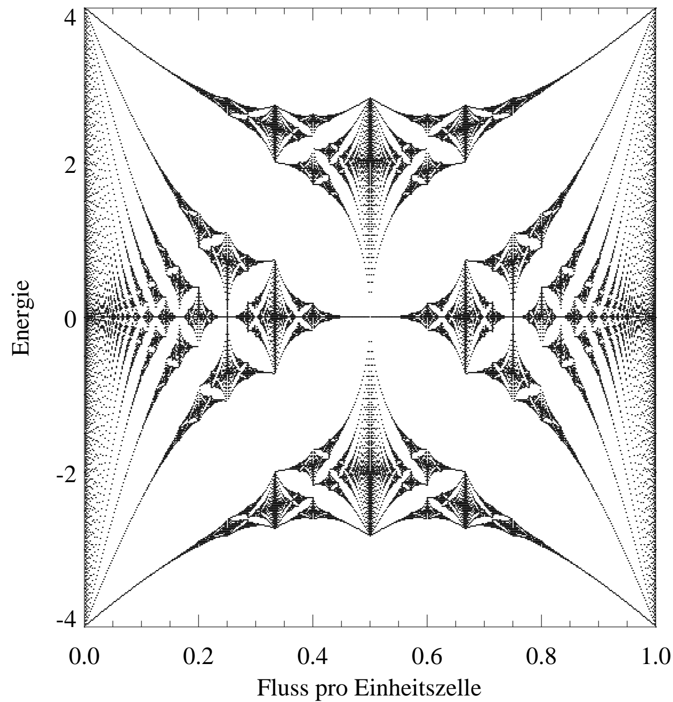
nach der üblichen Initialisierung – im wesentlichen aus drei ineinander geschachtelten Schleifen: Der äußeren `p`-Schleife, in der $\sigma(p)$ variiert wird, der mittleren `ie`-Schleife, innerhalb derer die Energie von -4 bis 4 läuft, und der inneren Schleife über `m`, in der bei vorgegebenem σ und E die Anzahl der Vorzeichenwechsel in der Folge der Polynome $p_0,\, p_1,\, \ldots,\, p_{\frac{q}{2}\pm 1}$ festgestellt wird. Dieser Teil des Programms hat also folgende Form:

```
for(p = 1; p < q; p++)
    {
     sigma = 2.0*pi*p/q;
     nalt = 0;

     for(ie = 0; ie < q+2 ; ie++)
         {
          e = 8.0*ie/q - 4.0 - 4.0/q ;
          n = 0;
          polyalt = 1.0; poly = 2.0*cos(sigma)-e;
          if( polyalt*poly < 0.0 ) n++ ;

          for( m = 2; m < q/2; m++ )
              {
               polyneu = (2.0*cos(sigma*m)-e)*poly-polyalt;
               if( poly*polyneu < 0.0) n++;
               polyalt = poly; poly = polyneu;
              }
         ⋮   Den entsprechenden Programmteil für die geraden
         ⋮   Eigenfunktionen haben wir hier ausgelassen.

          if(n > nalt) aktion;
          nalt = n;
         }
    }
```

Unter dem obigen `aktion` ist ein C-Befehl zu verstehen, mit dem z. B. ein Punkt auf den Bildschirm gezeichnet wird oder die entsprechenden (σ-E)-Werte in ein Datenfile geschrieben werden.

Ergebnis

Das Resultat dieses Programmes ist in der Abbildung 2.14 zu sehen. Die vertikale Achse markiert die Energie und die horizontale den Parameter σ, jeweils in 500 Pixel unterteilt. Die Energie wird in Pixelschritten erhöht. Immer wenn es Eigenwerte in diesem kleinen Intervall gibt, wird der Bildpunkt gezeichnet.



2.14 Eigenwerte der Harpergleichung als Funktion von $\sigma = Ba^2/(\frac{hc}{e})$, d. i. der magnetische Fluß durch die Einheitszelle in Einheiten des elementaren Flußquantums.

Es ist überraschend und faszinierend, welch komplexe, feingezeichnete und ansprechende Struktur dieser *Hofstadter-Schmetterling* hat, der aus der einfachen diskreten Schrödinger-Gleichung entsteht. Zunächst fallen die Symmetrien $E \to -E$, die hier übrigens nur näherungsweise erfüllt ist, und $\sigma \to 1 - \sigma$ auf. Des weiteren sieht der Schmetterling selbstähnlich aus: Teile der Abbildung treten etwas verzerrt auch an vielen anderen Stellen auf und scheinen sich bis ins unendlich Kleine fortzusetzen. Für jeden Wert von σ gibt es 500 Eigenwerte. Allerdings sind diese Werte nur am linken und rechten Rand gleichmäßig verteilt, sonst konzentrieren sie sich auf wenige Bereiche. Bei $\sigma = 1/2$ gibt es offensichtlich zwei solche Bänder, bei $\sigma = 1/3$ drei, bei $\sigma = 1/4$ vier, bei $\sigma = 1/5$ und $\sigma = 2/5$ fünf, bei $\sigma = 1/10$ und $3/10$ zehn usw. Da dicht neben jeder rationalen Zahl $\sigma = m/n$ (m und n teilerfremd) mit einem kleinen Nenner n auch Zahlen σ mit großem Nenner zu finden sind, liegen dicht neben den σ-Werten mit kleinen Nennern immer sehr viele Bänder. Jede irrationale Zahl kann durch ein Folge rationaler Zahlen mit immer

größer werdendem Nenner approximiert werden. Also liegen dicht neben einem irrationalen σ-Wert immer unendlich viele Energiebänder. Das ergibt die kunstvolle Zeichnung, die sich bei unendlich feiner Auflösung unserer Vorstellung widersetzt.

Übung

Schreiben Sie ein Programm, das möglichst schnell einen Hofstadter-Schmetterling auf einem 500×500 Fenster mit $\sigma = p/500$ $(p = 1, ..., 500)$ zeichnet. Nutzen Sie die Symmetrien des Schmetterlings aus und überlegen Sie sich einen Algorithmus, der die vielen großen Löcher überspringt.

Literatur

D. R. Hofstadter, *Energy levels and wave functions of Bloch electrons in rational and irrational magnetic fields*, Physical Review **B14**, 2239 (1976).

J. Stoer, *Numerische Mathematik 1*, Springer Verlag, 1994.

J. Stoer, R. Bulirsch, *Numerische Mathematik 2*, Springer Verlag, 1990.

2.5 Hubbard-Modell

Quantenmechanik gehört sicherlich nicht zu den leichten Gebieten der Physik. Ort und Impuls, Welle und Teilchen verlieren ihre anschauliche Bedeutung. Einzelne Meßergebnisse besagen in der Regel wenig, denn die Quantenmechanik ist im Kern eine statistische Theorie, deren Aussagen als Wahrscheinlichkeitsaussagen formuliert sind. Dazu kommt ein erheblicher mathematischer Apparat, in dem Zustände eingeführt und Operatoren erklärt werden müssen.

Noch schwieriger wird die mathematische Beschreibung, wenn man die Quantenmechanik wechselwirkender Teilchen untersuchen will, z. B. Supraleitung in einem Festkörper durch wechselwirkende Elektronen. Da die einzelnen Elektronen prinzipiell nicht unterscheidbar sind, muß man den Vielteilchenzustand geeignet aus der Überlagerung von Produkten von Einteilchenzuständen aufbauen. Für Elektronen führt dies zum Pauli-Prinzip: Jeder Einteilchenzustand darf nur einmal besetzt sein. In diesem Raum der Vielteilchenwellenfunktionen läßt sich der Hamiltonoperator als Matrix darstellen, deren Eigenwerte die Energien der stationären Zustände angeben.

Nur in wenigen Fällen kann man diese Matrix mit analytischen Methoden diagonalisieren. Aber auch eine numerische Lösung ist nicht leicht zu gewinnen. Die Größe des Systems und das Anwachsen der Zahl der Freiheitsgrade erweisen sich

dabei als das schwierigste Problem.

Als Beispiel für ein quantenmechanisches Vielteilchensystem betrachten wir ein Elektronengas in einem Kristallgitter, bei dem sowohl die gegenseitige Coulombabstoßung der Elektronen als auch deren Wechselwirkung mit den Ionenrümpfen zu berücksichtigen ist. Eine vereinfachende Beschreibung dieser Situation, die dennoch das Wesentliche enthält, wurde 1963 von J. Hubbard vorgeschlagen. Zur Zeit wird dieses Modell im Zusammenhang mit der Hochtemperatur-Supraleitung diskutiert. Es ist allerdings noch nicht sicher, ob das Modell überhaupt Supraleitung zeigt.

Wir wollen hier demonstrieren, wie man die Hamiltonmatrix des eindimensionalen Hubbard-Modells mit dem Computer erzeugen und diagonalisieren kann. Allerdings müssen wir uns dabei auf wenige Teilchen beschränken.

Physik

Im Prinzip werden die Vielteilchenzustände aus Produkten von Einteilchenzuständen aufgebaut. Es gibt aber einen Formalismus, der es erlaubt, ohne die explizite Darstellung der Produktwellenfunktionen auszukommen; es ist die sogenannte *zweite Quantisierung* (eine etwas irreführende Bezeichnung). Sie benutzt Erzeugungs- und Vernichtungsoperatoren, und die Zustände können durch Anwenden von Operatoren auf das Vakuum erzeugt werden. Dabei wird der Fermionen-Charakter durch die Operatoralgebra berücksichtigt. Der Hamiltonoperator wird ebenfalls durch Erzeugungs- und Vernichtungsoperatoren dargestellt, so daß die Hamiltonmatrix sich schließlich algebraisch konstruieren läßt.

Zunächst muß man die Einteilchenzustände numerieren und eine Reihenfolge festlegen. $|0\rangle$ sei das Vakuum. Ein Vielteilchenzustand mit N Teilchen hat dann die Form

$$c_{k_1}^\dagger \, c_{k_2}^\dagger \, ... \, c_{k_N}^\dagger |0\rangle \,, \tag{2.43}$$

wobei $(k_1, k_2, \ldots, k_N)$ ein geordnetes N-Tupel von Indizes und k_i der Index der entsprechenden Einteilchenwellenfunktion ist. Dieser Zustand läßt sich als Determinante von Einteilchenwellenfunktionen $\varphi_k(\boldsymbol{r})$ darstellen. So gilt beispielsweise für zwei Teilchen

$$\langle \boldsymbol{r}_1, \boldsymbol{r}_2 | c_1^\dagger c_2^\dagger | 0 \rangle = \psi(\boldsymbol{r}_1, \boldsymbol{r}_2) = \frac{1}{\sqrt{2}} \left[\varphi_1(\boldsymbol{r}_1)\varphi_2(\boldsymbol{r}_2) - \varphi_2(\boldsymbol{r}_1)\varphi_1(\boldsymbol{r}_2) \right] . \tag{2.44}$$

Wie schon erwähnt, ist diese explizite Darstellung gar nicht erforderlich. Bei orthonormalen Einteilchenzuständen ist allein die Antisymmetrie fermionischer Zustände wichtig, die durch folgende Operatoralgebra dargestellt wird:

$$c_k^\dagger \, c_m + c_m \, c_k^\dagger \quad = \quad \delta_{km} \,,$$

$$\begin{aligned} c_k^\dagger\, c_m^\dagger + c_m^\dagger\, c_k^\dagger &= 0\,, \\ c_k\, c_m + c_m\, c_k &= 0\,. \end{aligned} \tag{2.45}$$

Der Operator $c_k^\dagger$ erzeugt eine Einteilchenwellenfunktion $\varphi_k(\boldsymbol{r})$ im Vielteilchenzustand, falls diese noch nicht vorhanden ist, sonst erhält man 0. Das dazu adjungierte c_k vernichtet $\varphi_k(\boldsymbol{r})$, falls vorhanden, sonst gibt es 0. Das Pauli-Prinzip folgt aus $c_k^\dagger c_k^\dagger = 0$; Doppelbesetzung von $\varphi_k(\boldsymbol{r})$ ist also nicht erlaubt.

Beim Hubbard-Modell sind die Einteilchenzustände $\varphi_k(\boldsymbol{r})$ lokalisierte Wellenfunktionen von Elektronen an Gitterplätzen. Der Index k enthält die Nummer des Platzes und die Quantenzahl der z-Komponente des Spins, dargestellt durch $\sigma =\uparrow$ oder $\sigma =\downarrow$. Die kinetische Energie wird durch ein „Hüpfen“ vom Platz k zum Nachbarplatz m mit dem Operator

$$-t\, c_{m\sigma}^\dagger\, c_{k\sigma} \tag{2.46}$$

berücksichtigt. Befinden sich zwei Elektronen an demselben Platz k, so sollen sie die Coulombabstoßung

$$U\, \boldsymbol{n}_{k\uparrow}\, \boldsymbol{n}_{k\downarrow} \tag{2.47}$$

spüren. Dabei gibt $\boldsymbol{n}_{k\sigma} \equiv c_{k\sigma}^\dagger c_{k\sigma}$ den Wert 1 falls sich ein Elektron mit dem Spin σ am Platz k befindet, sonst gibt $\boldsymbol{n}_{k\sigma}$ den Wert 0. Die Parameter t und U modellieren die Größe der kinetischen bzw. potentiellen Energie.

Im folgenden beschränken wir uns auf eine Kette mit M Plätzen und nehmen periodische Randbedingungen an. Dann lautet der Hamiltonoperator

$$H = -t \sum_{k=1}^{M} \sum_{\sigma} (c_{k\sigma}^\dagger\, c_{k+1,\sigma} + c_{k+1,\sigma}^\dagger\, c_{k\sigma}) + U \sum_{k=1}^{M} \boldsymbol{n}_{k\uparrow}\, \boldsymbol{n}_{k\downarrow} \tag{2.48}$$

mit $c_{M+1,\sigma}^\dagger = c_{1,\sigma}^\dagger$ und $c_{M+1,\sigma} = c_{1\sigma}$. Als Ordnung der Einteilchenzustände wählen wir

$$\{1\uparrow, 2\uparrow, \ldots, M\uparrow\}, \{1\downarrow, 2\downarrow, \ldots, M\downarrow\}\,. \tag{2.49}$$

Jeder Vielteilchenzustand kann also durch die Besetzungszahlen der Einteilchenzustände beschrieben werden. Dann gilt beispielsweise:

$$|\{1,1,0,\ldots,0\},\{0,\ldots,0,1\}\rangle = c_{1\uparrow}^\dagger c_{2\uparrow}^\dagger c_{M\downarrow}^\dagger\, |0\rangle\,.$$

Wendet man den Operator $c_{k\sigma}^\dagger$ auf einen Zustand an, so muß man mit der Algebra (2.45) diesen Operator solange mit den anderen $c_{m\sigma'}^\dagger$ vertauschen, bis die Reihenfolge (2.49) stimmt. Dadurch erhält man auch das Vorzeichen des Zustandes. So ergibt z. B.

$$\begin{aligned} c_{1\downarrow}^\dagger |\{1,0,\ldots,0\},\{0,\ldots,0,1\}\rangle &= c_{1\downarrow}^\dagger\, c_{1\uparrow}^\dagger\, c_{M\downarrow}^\dagger\, |0\rangle \\ &= -\, c_{1\uparrow}^\dagger\, c_{1\downarrow}^\dagger\, c_{M\downarrow}^\dagger |0\rangle \\ &= -\, |\{1,0,\ldots,0\},\{1,0,\ldots,0,1\}\rangle \end{aligned}$$

und

$$\begin{aligned} c_{M\downarrow}|\{1,0,\dots,0\},\{0,\dots,0,1\}\rangle &= c_{M\downarrow}\,c^\dagger_{1\uparrow}\,c^\dagger_{M\downarrow}\,|0\rangle = -c^\dagger_{1\uparrow}\,c_{M\downarrow}\,c^\dagger_{M\downarrow}\,|0\rangle \\ &= -c^\dagger_{1\uparrow}\,|0\rangle + c^\dagger_{1\uparrow}\,c^\dagger_{M\downarrow}\,c_{M\downarrow}\,|0\rangle \\ &= -c^\dagger_{1\uparrow}\,|0\rangle = -|\{1,0,\dots,0\},\{0,\dots,0\}\rangle\,. \end{aligned}$$

Allgemein gilt bei Anwendung von $c^\dagger_{k\sigma}$ oder $c_{k\sigma}$ auf einen Zustand

$$|\boldsymbol{n}\rangle = |\{n_{1\uparrow},\dots,n_{M\uparrow}\},\{n_{1\downarrow},\dots,n_{M\downarrow}\}\rangle\,,$$

daß die Anzahl der Teilchen links von $k\sigma$ das Vorzeichen bestimmt. Wir definieren deshalb die folgende Signum-Funktion:

$$\operatorname{sign}(k\sigma,\boldsymbol{n}) = (-1)^{\delta_{\sigma\downarrow}\sum_{i=1}^{M} n_{i\uparrow}}\;(-1)^{\sum_{j=1}^{k-1} n_{j\sigma}}\,. \tag{2.50}$$

Sie produziert gerade so oft einen Faktor -1 wie es in $\boldsymbol{n}$ von 0 verschiedene Einträge vor der Position $k\sigma$ gibt. Damit können wir die Wirkung der Erzeuger und Vernichter folgendermaßen schreiben:

$$c^\dagger_{k\sigma}|\boldsymbol{n}\rangle = (1-n_{k\sigma})\,\operatorname{sign}(k\sigma,\boldsymbol{n})\,|\{n_{1\uparrow},\dots,1,\dots,n_{M\downarrow}\}\rangle\,. \tag{2.51}$$

Die 1 steht an der Position $k\sigma$, und der Faktor $(1-n_{k\sigma})$ sorgt dafür, daß es keine Doppelbesetzung desselben Zustands gibt. Analog gilt:

$$c_{k\sigma}|\boldsymbol{n}\rangle = n_{k\sigma}\,\operatorname{sign}(k\sigma,\boldsymbol{n})\,|\{n_{1\uparrow},\dots,0,\dots,n_{M\downarrow}\}\rangle\,, \tag{2.52}$$

wobei der Zustand $k\sigma$ vernichtet wurde.

Jeder Platz in der Kette hat vier mögliche Zustände: leer, besetzt mit entweder $\uparrow$ oder $\downarrow$ oder zweifach besetzt mit $\uparrow$ und $\downarrow$, so daß es insgesamt also 4^M Vielteilchenzustände gibt. Der Hamiltonoperator H ist daher in dieser Basis eine $4^M \times 4^M$-Matrix. Diese Matrix ist aber erstens zum großen Teil mit Nullen belegt, und zweitens kann sie durch Symmetrien in Unterblöcke zerlegt werden. Das Hubbard-Modell hat folgende Erhaltungsgrößen: Anzahl der Teilchen mit positiven $(N_\uparrow)$ und negativen $(N_\downarrow)$ Spins, Gesamtspin-Quantenzahl und räumliche Symmetrien. Dadurch reduziert sich die Größe der Blöcke von H erheblich. So kann für 3 Plätze die 64×64-Matrix auf Unterblöcke mit einer maximalen Größe von 3×3 reduziert werden.

Hier wollen wir nur $N_\uparrow$ und $N_\downarrow$ als Erhaltungsgrößen berücksichtigen und auch nicht verschweigen, daß für das eindimensionale Modell eine Reihe von exakten Resultaten bekannt ist, die man mit Bethe-Ansatz-Methoden gewinnen kann. Unser Beispiel dient also nur als Einstieg zu numerischen Rechnungen für zwei- oder dreidimensionale Modelle.

Algorithmus

Wir wollen wieder *Mathematica* benutzen, um die Darstellung der Operatoralgebra (2.45) durch die Gleichungen (2.51) und (2.52) kompakt zu formulieren. Statt des Symbols $|\ldots\rangle$ verwenden wir in *Mathematica* den Kopf `z` und notieren die Vielteilchenzustände in der Form

$$\texttt{z[arg]} \;\text{ mit }\; \texttt{arg} = \{\{n_{1\uparrow}, ..., n_{M\uparrow}\}, \{n_{1\downarrow}, ..., n_{M\downarrow}\}\} \tag{2.53}$$

und $n_{k\sigma} \in \{0, 1\}$. Wir brauchen den Kopf `z`, da wir die Zustände addieren und mit Zahlen multiplizieren müssen. Würden wir dies nur mit der Liste `arg` ausführen, so würden wir falsche Ergebnisse erhalten. Die wesentlichen Manipulationen betreffen aber das Argument von `z`. So erhält man z. B. $n_{k\sigma}$ aus `arg[[sigma, k]]` mit `k` $= k$ und `sigma` $= 1$ für $\sigma = \uparrow$ bzw. `sigma` $= 2$ für $\sigma = \downarrow$.

Nach Vorgabe der Zahlen $M, N_\uparrow$ und $N_\downarrow$ erzeugen wir mit den Funktionen `Permutations[..]` und `Table[..]` die Liste `index`, die alle überhaupt möglichen Zustände enthält. Der dabei benutzte Befehl `Flatten[..]` beseitigt eine Klammerebene. Für $M = 3, N_\uparrow = 2$ und $N_\downarrow = 1$ erhalten wir z. B.

```
index = {{{1,1,0}, {1,0,0}}, {{1,1,0}, {0,1,0}}, {{1,1,0}, {0,0,1}},

         {{1,0,1}, {1,0,0}}, {{1,0,1}, {0,1,0}}, {{1,0,1}, {0,0,1}},

         {{0,1,1}, {1,0,0}}, {{0,1,1}, {0,1,0}}, {{0,1,1}, {0,0,1}}}
```

Der Operator $c^\dagger_{k\sigma}$ muß in dem Argument von `z` eine 1 an der richtigen Stelle erzeugen. Dies geschieht durch die Funktion `plus` :

```
plus[k_,sigma_][arg_]:= ReplacePart[arg,1,{sigma,k}]
```

Entsprechend erzeugt `minus` eine 0. Unter Verwendung der oben eingeführten Vorzeichenfunktion (2.50) können wir jetzt analog zu (2.51) und (2.52) den Operator $c^\dagger_{k\sigma}$ durch seine Wirkung auf Zustände definieren:

```
cdagger[k_,sigma_][factor_.*z[arg_]]:= factor*
                                (1 - arg[[sigma,k]])*
                                sign[k,sigma,arg]*
                                z[plus[k,sigma][arg]]
```

Der Parameter `factor_.` wird in *Mathematica* auf den Wert 1 gesetzt, falls der Zustand keinen zusätzlichen Faktor hat. Da durch diese Operatoren auch immer der Wert 0 erzeugt werden kann, z. B. ist $c^\dagger_{1\uparrow}|\{1, \ldots\}, \{\ldots\}\rangle = 0$, muß `cdagger` auch für die Zahl 0 definiert werden:

```
cdagger[k_,sigma_][0]:= 0
```

Analog wird $c_{k\sigma}$ durch c[k_,sigma_] definiert. Außerdem brauchen wir noch die Operatoren $\boldsymbol{n}_{k\sigma}$:

```
               n[k_,sigma_][0]:= 0
n[k_,sigma_][factor_.*z[arg_]]:= factor*
                                   arg[[sigma,k]]*z[arg]
```

Der Hamiltonoperator (2.48) der Hubbard-Kette lautet dann einfach

```
H[vector_]=Expand[
        -t*Sum[cdagger[k,sigma][c[k+1,sigma][vector]]+
               cdagger[k+1,sigma][c[k,sigma][vector]],
                                {k,sites},{sigma,2} ]+
             u*Sum[n[k,1][n[k,2][vector]] ,{k,sites}]
                 ]
```

Die Länge M der Kette ist hier mit sites bezeichnet. Um die Hamiltonmatrix zu bekommen, brauchen wir die Skalarprodukte $\langle \boldsymbol{n}_i | H | \boldsymbol{n}_j \rangle$ zwischen $|\boldsymbol{n}_i\rangle$ und $H|\boldsymbol{n}_j\rangle$. Da unsere Vielteilchenzustände $|\boldsymbol{n}\rangle$ orthonormiert sind, müssen wir nur noch die Linearität des Skalarproduktes definieren:

```
scalarproduct[a_,0]:= 0
scalarproduct[a_,b_ + c_]:= scalarproduct[a,b] +
                            scalarproduct[a,c]
scalarproduct[z[arg1_],factor_. z[arg2_]]:=
                              factor*If[arg1==arg2,1,0]
```

Die Antilinearität im ersten Argument wird hier nicht benötigt. Man beachte, daß der Vergleich zweier Zustände, in diesem Fall also der Vergleich zweier geschachtelter Listen, in *Mathematica* durch arg1==arg2 sehr kompakt geschrieben werden kann.

Die folgende Tabelle liefert die gesuchte Hamiltonmatrix $\langle \boldsymbol{n}_i | H | \boldsymbol{n}_j \rangle$:

```
h = (hlist = Table[H[z[index[[j]]]], {j, end}];
     Table[ scalarproduct[ z[index[[i]]], hlist[[j]]],
                                   {i,end}, {j,end}])
```

Die obere Grenze der Laufindizes ist vorher durch end = Length[index] bestimmt worden.

Mit Eigenvalues[..] und Eigensystem[..] erhält man dann die Energien und die stationären Zustände des Vielteilchenproblems. Den Grundzustand filtern wir heraus mit der Befehlskombination

```
g[uu_]:= Sort[Thread[ Eigensystem[
               N[ h /. {t -> 1.0, u -> uu} ]]]][[1,2]]
```

Der Befehl `Thread[..]` ordnet das Ergebnis von `Eigensystem[..]` in der Weise, daß Eigenwert und zugehöriger Eigenvektor zusammengefaßt werden. In dem sortierten Resultat steht dann der Grundzustand als der Zustand mit der kleinsten Energie an erster Stelle. Die erste Komponente davon ist die Energie des Grundzustandes, die zweite Komponente, eine Liste, enthält die Koeffizienten $g_{\boldsymbol{n}}$ des Grundzustandes $|g\rangle$ bezüglich der Basis $|\boldsymbol{n}\rangle$. Wir interessieren uns z. B. für die mittlere Doppelbesetzung im Grundzustand, also für

$$\frac{1}{M}\sum_{k=1}^{M}\langle g|\boldsymbol{n}_{k\uparrow}\boldsymbol{n}_{k\downarrow}|g\rangle = \frac{1}{M}\sum_{k=1}^{M}\sum_{\{\boldsymbol{n}\}}|g_{\boldsymbol{n}}|^2\, n_{k\uparrow}(\boldsymbol{n})\, n_{k\downarrow}(\boldsymbol{n})\,. \tag{2.54}$$

Das *Mathematica*-Äquivalent von Gleichung (2.54) ist

```
Sum[Sum[Abs[g[u][[j]]]^2*index[[j,1,k]]*index[[j,2,k]],
        {j,end}], {k,sites}] / sites
```

Dies läßt sich, da die Summen vertauschbar sind und als Skalarprodukte angesehen werden können, auf folgende Weise sehr kompakt formulieren:

```
(Abs[g[u]]^2).Map[#[[1]].#[[2]]&, index] / sites
```

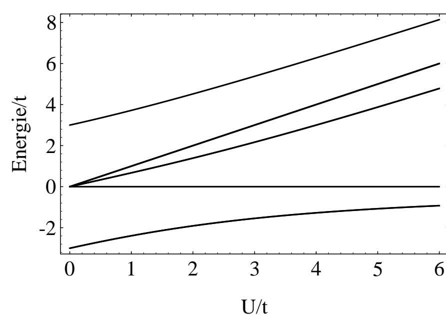
Ergebnisse

Für $M = 3$ Plätze und $N_\uparrow = 2, N_\downarrow = 1$ findet *Mathematica* die Energien noch analytisch. Dazu schreiben wir mit

```
TeXForm[Eigenvalues[h]] >> hubbard.tex
```

das Ergebnis in TEX-Form in eine Datei, fügen die mathematische Umgebung und einige Zeilenumbrüche hinzu und übersetzen sie mit TEX bzw. LATEX. Das Ergebnis sehen Sie auf der nächsten Seite.

Abbildung 2.16 zeigt diese Energien als Funktion des Verhältnisses U/t von Coulomb- und kinetischer Energie. Für $U = 0$ gibt es keine



2.16 Energie-Eigenwerte des Hubbard-Modells in Abhängigkeit von U/t für $M = 3,\ N_\uparrow = 2$ und $N_\downarrow = 1$.

$$\{0,\, u,\, u,$$

$$\frac{2\,u}{3} - \frac{2^{\frac{1}{3}}\left(-27\,t^2-u^2\right)}{3\left(-162\,t^2\,u+16\,u^3-18\,u\left(-9\,t^2+u^2\right)+\sqrt{4\left(-27\,t^2-u^2\right)^3+\left(-162\,t^2\,u+16\,u^3-18\,u\left(-9\,t^2+u^2\right)\right)^2}\right)^{\frac{1}{3}}} + \frac{\left(-162\,t^2\,u+16\,u^3-18\,u\left(-9\,t^2+u^2\right)+\sqrt{4\left(-27\,t^2-u^2\right)^3+\left(-162\,t^2\,u+16\,u^3-18\,u\left(-9\,t^2+u^2\right)\right)^2}\right)^{\frac{1}{3}}}{3\,2^{\frac{1}{3}}},$$

$$\frac{2\,u}{3} - \frac{2^{\frac{1}{3}}\left(-27\,t^2-u^2\right)}{3\left(-162\,t^2\,u+16\,u^3-18\,u\left(-9\,t^2+u^2\right)+\sqrt{4\left(-27\,t^2-u^2\right)^3+\left(-162\,t^2\,u+16\,u^3-18\,u\left(-9\,t^2+u^2\right)\right)^2}\right)^{\frac{1}{3}}} + \frac{\left(-162\,t^2\,u+16\,u^3-18\,u\left(-9\,t^2+u^2\right)+\sqrt{4\left(-27\,t^2-u^2\right)^3+\left(-162\,t^2\,u+16\,u^3-18\,u\left(-9\,t^2+u^2\right)\right)^2}\right)^{\frac{1}{3}}}{3\,2^{\frac{1}{3}}},$$

$$\frac{2\,u}{3} + \frac{\left(1+i\,\sqrt{3}\right)\left(-27\,t^2-u^2\right)}{3\,2^{\frac{2}{3}}\left(-162\,t^2\,u+16\,u^3-18\,u\left(-9\,t^2+u^2\right)+\sqrt{4\left(-27\,t^2-u^2\right)^3+\left(-162\,t^2\,u+16\,u^3-18\,u\left(-9\,t^2+u^2\right)\right)^2}\right)^{\frac{1}{3}}} - \frac{\left(1-i\,\sqrt{3}\right)\left(-162\,t^2\,u+16\,u^3-18\,u\left(-9\,t^2+u^2\right)+\sqrt{4\left(-27\,t^2-u^2\right)^3+\left(-162\,t^2\,u+16\,u^3-18\,u\left(-9\,t^2+u^2\right)\right)^2}\right)^{\frac{1}{3}}}{6\,2^{\frac{1}{3}}},$$

$$\frac{2\,u}{3} + \frac{\left(1+i\,\sqrt{3}\right)\left(-27\,t^2-u^2\right)}{3\,2^{\frac{2}{3}}\left(-162\,t^2\,u+16\,u^3-18\,u\left(-9\,t^2+u^2\right)+\sqrt{4\left(-27\,t^2-u^2\right)^3+\left(-162\,t^2\,u+16\,u^3-18\,u\left(-9\,t^2+u^2\right)\right)^2}\right)^{\frac{1}{3}}} - \frac{\left(1-i\,\sqrt{3}\right)\left(-162\,t^2\,u+16\,u^3-18\,u\left(-9\,t^2+u^2\right)+\sqrt{4\left(-27\,t^2-u^2\right)^3+\left(-162\,t^2\,u+16\,u^3-18\,u\left(-9\,t^2+u^2\right)\right)^2}\right)^{\frac{1}{3}}}{6\,2^{\frac{1}{3}}},$$

$$\frac{2\,u}{3} + \frac{\left(1-i\,\sqrt{3}\right)\left(-27\,t^2-u^2\right)}{3\,2^{\frac{2}{3}}\left(-162\,t^2\,u+16\,u^3-18\,u\left(-9\,t^2+u^2\right)+\sqrt{4\left(-27\,t^2-u^2\right)^3+\left(-162\,t^2\,u+16\,u^3-18\,u\left(-9\,t^2+u^2\right)\right)^2}\right)^{\frac{1}{3}}} - \frac{\left(1+i\,\sqrt{3}\right)\left(-162\,t^2\,u+16\,u^3-18\,u\left(-9\,t^2+u^2\right)+\sqrt{4\left(-27\,t^2-u^2\right)^3+\left(-162\,t^2\,u+16\,u^3-18\,u\left(-9\,t^2+u^2\right)\right)^2}\right)^{\frac{1}{3}}}{6\,2^{\frac{1}{3}}},$$

$$\frac{2\,u}{3} + \frac{\left(1-i\,\sqrt{3}\right)\left(-27\,t^2-u^2\right)}{3\,2^{\frac{2}{3}}\left(-162\,t^2\,u+16\,u^3-18\,u\left(-9\,t^2+u^2\right)+\sqrt{4\left(-27\,t^2-u^2\right)^3+\left(-162\,t^2\,u+16\,u^3-18\,u\left(-9\,t^2+u^2\right)\right)^2}\right)^{\frac{1}{3}}} - \frac{\left(1+i\,\sqrt{3}\right)\left(-162\,t^2\,u+16\,u^3-18\,u\left(-9\,t^2+u^2\right)+\sqrt{4\left(-27\,t^2-u^2\right)^3+\left(-162\,t^2\,u+16\,u^3-18\,u\left(-9\,t^2+u^2\right)\right)^2}\right)^{\frac{1}{3}}}{6\,2^{\frac{1}{3}}}\}$$

2.15 Eigenwerte einer 9×9-Hubbard-Matrix, die von *Mathematica* wegen der Symmetrie der Matrix nicht nur numerisch, sondern allgemein als Funktion von U und t gefunden werden können. Umwandlung in die obige Form mit der Funktion `TeXForm[..]`.

Wechselwirkung zwischen den Elektronen, also können wir die Energieniveaus dadurch berechnen, daß wir Einteilchenniveaus auffüllen. Bei periodischen Randbe-

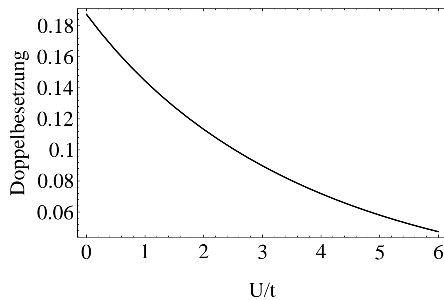
dingungen lassen sich diese Einteilchenenergien für beliebiges M explizit angeben. Wir definieren die Wellenvektoren $k_\nu = \frac{2\pi\nu}{M}$, $\nu = 0, 1, \dots, M-1$. Damit sind die Einteilchen-Eigenzustände des Hubbard-Hamiltonians und deren Energien gegeben durch

$$\begin{aligned} |k_\nu^\sigma\rangle &= \frac{1}{\sqrt{M}} \sum_{l=1}^{M} e^{-i\,k_\nu\,l}\, c_{l\sigma}^\dagger |0\rangle\,, \\ E_\nu &= -2\,t\,\cos k_\nu\,. \end{aligned}$$

Für $M = 3$ erhalten wir als Einteilchen-Energien $-2\,t, t$ und nochmal t. In diese Niveaus können 2 Spin $\uparrow$- und 1 Spin $\downarrow$- Teilchen gesetzt werden, wobei allerdings die Doppelbesetzung zweier $\uparrow$-Spins verboten ist. Dies ergibt folgende 9 Niveaus, in Übereinstimmung mit Abbildung 2.16 :

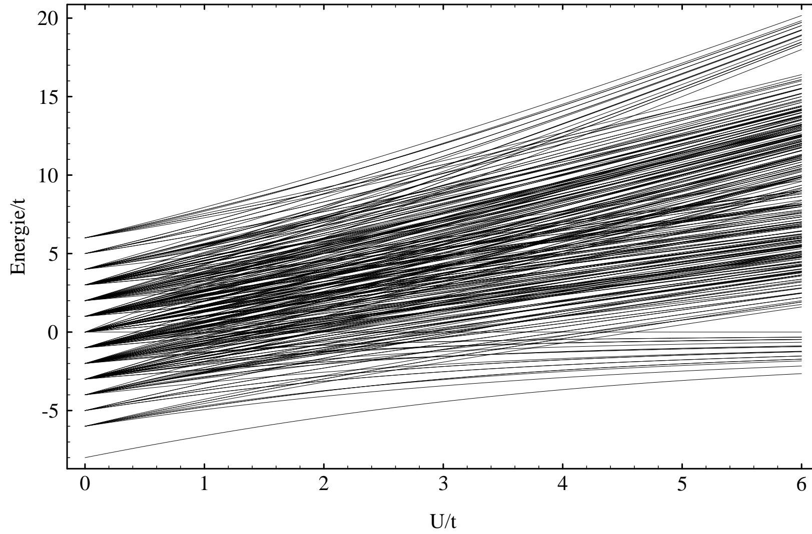
zweifach entartet: $\varepsilon_1 = -3\,t$,
fünffach entartet: $\varepsilon_2 = 0$,
zweifach entartet: $\varepsilon_3 = 3\,t$.

Daß auch für $U \neq 0$ außer dem einen Niveau mit $\varepsilon = 0$ alle anderen zweifach entartet sind, liegt an der Translations- und Spiegelsymmetrie dieses Modells. Die mittlere Doppelbesetzung im Grundzustand für $M = 4$ Plätze und sogenannter *Halbfüllung*, d. h. $N_\uparrow + N_\downarrow = M$, hier für $N_\uparrow = 2$ und $N_\downarrow = 2$, zeigt Abbildung 2.17. Für $U \to \infty$ ist Doppelbesetzung natürlich verboten. Die nächsten



2.17 Mittelwert der Doppelbesetzung im Grundzustand in Abhängigkeit von U/t für $M = 4$, $N_\uparrow = 2$ und $N_\downarrow = 2$.

zwei Abbildungen enthalten Resultate für einen Hubbardring mit $M = 6$ Plätzen, $N_\uparrow = 3$ und $N_\downarrow = 3$. So moderat diese Zahlen erscheinen mögen, die Anzahl der Zustände wächst mit M stark an und beträgt für dieses Beispiel immerhin schon $\binom{6}{3} \cdot \binom{6}{3} = 400$. Die Energien der stationären Zustände zu berechnen bedeutet also, die Eigenwerte einer 400×400-Matrix zu bestimmen. Gäbe es nicht die Symmetrie und die vielfache Entartung, so würde das folgende Diagramm 400 verschiedene Energieniveaus zeigen. Für den Fall des halbgefüllten Bandes, also $\frac{N_\uparrow + N_\downarrow}{M} = 1$,

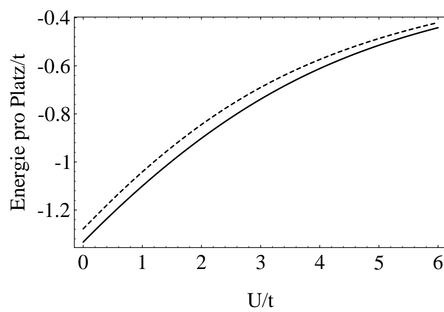


2.18 Energie-Eigenwerte des Hubbardmodells in Abhängigkeit von U/t für $M = 6,\ N_\uparrow = 3$ und $N_\downarrow = 3$.

läßt sich, wenn außerdem noch $\frac{N_\uparrow}{M} = \frac{N_\downarrow}{M} = \frac{1}{2}$ gilt, im Limes $M \to \infty$ die Grundzustandsenergie pro Platz für beliebiges U/t in geschlossener Form angeben. Es gilt:

$$\frac{\varepsilon_0}{M} \quad \longrightarrow \quad -4\,t \int_0^\infty \frac{J_0(\omega)\, J_1(\omega)\, d\omega}{\omega[1 + \exp(\frac{1}{2}\omega U/t)]} \quad \text{für } M \to \infty\,, \tag{2.55}$$

wobei J_0 und J_1 Besselfunktionen sind. Wir werten das Integral numerisch aus und vergleichen das Resultat mit den entsprechenden Werten für $M = 6$. Erstaunlicherweise liegen die beiden Kurven gar nicht so weit auseinander.



2.19 Grundzustandsenergie pro Platz für das halbgefüllte Band. Die gestrichelte Kurve ist das exakte Resultat für $M \to \infty$, die durchgezogene Linie beschreibt die Werte für $M = 6$.

Um größere Systeme zu berechnen, muß man erstens die Symmetrie berücksichtigen und zweitens eine schnelle Computersprache benutzen. So kann man heute Modelle mit bis zu $M = 25$ Plätzen exakt diagonalisieren. Mit der Quanten-Monte-Carlo-Methode, die das Hubbard-Modell bis auf statistische Fehler numerisch exakt behandelt, kann man bis zu $M = 1024$ Gitterplätze untersuchen. Auch mit heutigen Superrechnern sind also nur winzig kleine Quanten-Modelle berechenbar. Die drastische Leistungssteigerung zukünftiger Computer wird die Systemgröße nur geringfügig erhöhen können – es sei denn, es gelingt, neue Algorithmen zu entwickeln.

Übung

Die z-Komponente des Spinoperators am Platz i läßt sich durch die Teilchenzahloperatoren $\boldsymbol{n}_{i\uparrow}$ und $\boldsymbol{n}_{i\downarrow}$ ausdrücken: $S_i^z = 1/2\,(\boldsymbol{n}_{i\uparrow} - \boldsymbol{n}_{i\downarrow})$. Berechnen Sie für den Grundzustand $|g\rangle$ mit $M = 4$ Plätzen und $N_\uparrow = N_\downarrow = 2$ die Spinkorrelationen $\frac{1}{M}\sum_{i=1}^{M}\langle g|S_i^z\,S_{i+1}^z|g\rangle$ und $\frac{1}{M}\sum_{i=1}^{M}\langle g|S_i^z\,S_{i+2}^z|g\rangle$ in Abhängigkeit von U/t.

Literatur

J. E. Hirsch, *Two-dimensional Hubbard model: Numerical simulation study*, Physical Review **B31**, 4403 (1985).

H. Q. Lin, J. E. Gubernatis, *Exact diagonalization methods for quantum systems*, Computers in Physics **7**, 400 (1993).

A. Montorsi [ed.], *The Hubbard model: A Reprint Volume*, World Scientific, 1992.

Kapitel 3
Iterationen

Eine Funktion besteht aus einer Menge von Anweisungen, die gegebene Eingabe- in Ausgabewerte umwandeln. Gehört nun die Ausgabe selbst zur Definitionsmenge der betrachteten Funktion, so kann sie selbst wieder zur Eingabe werden, und die Funktion liefert neue Ausgabewerte. Dies kann dann beliebig oft wiederholt werden. Während es oft keine analytischen Möglichkeiten gibt, die strukturellen Eigenschaften solcher Iterationen von der Form $x_{t+1} = f(x_t)$ zu berechnen, sind sie im Computer leicht zu realisieren. Man muß offenbar nur dieselbe Funktion $f(x)$ immer wieder auf das Ergebnis anwenden. Wir wollen dies an einigen Beispielen demonstrieren.

3.1 Populationsdynamik

Die wohl bekannteste Iteration einer nichtlinearen Funktion ist die sogenannte *logistische Abbildung*. Sie ist eine einfache Parabel, die die reellen Zahlen des Einheitsintervalls in sich selbst abbildet. Wir wollen sie hier vorstellen, weil jeder Wissenschaftler die Eigenschaften einer so elementaren Iteration kennen sollte.

Ein einfacher Mechanismus führt zu komplexem Verhalten – dies ist das Ergebnis der wiederholten Anwendung einer quadratischen Abbildung. Obwohl es nur wenige analytische Ergebnisse dazu gibt, kann jeder die Iteration leicht auf einem Taschenrechner nachvollziehen. Bei gewissen Parameterwerten der Parabel findet man ein unregelmäßiges Umherspringen der iterierten Zahlen im Einheitsintervall. Die Zahlenfolge ist dann extrem empfindlich auf Änderungen des Startwertes der Iteration. Eine wohldefinierte Abbildungsvorschrift liefert eine Zahlenfolge, die schon bei geringer Anfangsungenauigkeit praktisch unvorhersagbar ist. Ein solches Verhalten nennt man *deterministisches Chaos*.

Einige Größen, die man mit dem Computer berechnet, sind nicht nur das Ergebnis einer mathematischen Spielerei, sondern man findet diese Zahlen in vielen Experimenten beim Übergang vom regelmäßigen zum chaotischen Verhalten wieder. Solche Universalität quantitativer Größen ist ein weiterer faszinierender Aspekt dieser Gleichung.

Physik

Wir betrachten die Iteration

$$x_{n+1} = 4\,r\,x_n(1 - x_n) = f(x_n)\,, \quad n = 0, 1, 2, 3, \ldots\,, \tag{3.1}$$

die erstmalig 1845 von dem belgischen Biomathematiker P. F. Verhulst in einer Arbeit zur Populationsdynamik eingeführt wurde. Dabei wird x_n im Intervall $[0, 1]$ iteriert, und r ist ein Parameter, der ebenfalls in $[0, 1]$ variiert werden kann.

Die Funktion (3.1) ist für verschiedene Fragestellungen interessant:

1. Als einfaches Modell zur zeitlichen Entwicklung einer wachsenden Population.

   x_n ist proportional zur Bevölkerungsdichte einer Spezies zur Zeit n. Ihr Wachstum wird beschrieben durch einen linearen Beitrag $4\,r\,x_n$, der allein genommen für $r > \frac{1}{4}$ zum exponentiellen Wachstum, also zur Bevölkerungsexplosion führt. Das Wachstum der Population wird allerdings durch den nichtlinearen Beitrag $-4\,r\,x_n^2$ (z. B. durch begrenzte Futtermenge) gedämpft.

2. Als einfaches Beispiel zur klassischen Mechanik mit Reibung.

   Wir werden im Abschnitt 4.2 sehen, daß die Bewegung eines mechanischen Systems durch einen Poincaré-Schnitt im Phasenraum analysiert werden kann. Dieser Schnitt ergibt eine Iteration einer nichtlinearen Abbildung. Daher ist es nützlich zu wissen, welche Eigenschaften solche Iterationen schon in einer Dimension haben, auch wenn in der Mechanik eindimensionale Poincaré-Schnitte kein Chaos zeigen können.

3. Als Beispiel für eine (schlechte) numerische Lösung einer Differentialgleichung.

   Wenn benachbarte x_n-Werte sich nur wenig ändern, dann können n und x_n kontinuierlich fortgesetzt werden. Setzt man

   $$x_{n+1} - x_n \simeq \frac{dx}{dn}\,,$$

   so wird Gleichung (3.1) zur Näherung der Differentialgleichung

   $$\frac{dx}{dn} = (4r - 1)x - 4r\,x^2$$

   mit der Lösung

   $$x(n) = \frac{4r - 1}{4r + \text{const}\, e^{(1-4r)n}}\,.$$

   Für $r < \frac{1}{4}$ ist $x_\infty = 0$ ein Attraktor, während für $r > \frac{1}{4}$ alle Startwerte $x(0) > 0$ zum Wert $x_\infty = 1 - \frac{1}{4r}$ laufen. Wie in der diskreten Iteration hat diese Gleichung

bei $r_0 = \frac{1}{4}$ einen Phasenübergang von einer aussterbenden zu einer konstanten Population. Es wird sich aber zeigen, daß für größere r-Werte die Differentialgleichung keine Ähnlichkeit mehr zu ihrer diskreten Approximation hat.

Für $r > r_0 = \frac{1}{4}$ hat die Gleichung (3.1) den Fixpunkt $x^* = f(x^*)$ mit $x^* = 1 - \frac{1}{4r}$. Wir betrachten nun eine kleine Abweichung $x^* + \varepsilon_0$ vom Fixpunkt und iterieren diese. Wegen $f(x^* + \varepsilon_0) \simeq f(x^*) + f'(x^*)\varepsilon_0 = x^* + f'(x^*)\varepsilon_0$ gilt für kleine ε_n:

$$\varepsilon_n \simeq f'(x^*)\varepsilon_{n-1} \simeq [f'(x^*)]^n \varepsilon_0 \,. \tag{3.2}$$

Daher explodiert die Störung ε_0 für $|f'(x^*)| > 1$, was $r > r_1 = \frac{3}{4}$ entspricht. Wir werden sehen, daß dann die x_n nach einigen Schritten zwischen den Werten x_1^* und x_2^* hin- und herspringen:

$$\begin{aligned} x_1^* = f(x_2^*)\,,\ x_2^* = f(x_1^*)\,, \\ \text{also}\ \ x_i^* = f(f(x_i^*))\ \ \text{für}\ i = 1,2\,. \end{aligned} \tag{3.3}$$

Die iterierte Abbildung $f^{(2)}(x) = f(f(x))$ hat demnach für gewisse r-Werte zwei stabile Fixpunkte.

Nun gilt $|f^{(2)\prime}(x_i^*)| > 1$ für $r > r_2 = (1 + \sqrt{6})/4$. Daher zeigt dasselbe Argument wie oben, daß der Zweierzyklus für $r > r_2$ instabil wird, die x_n laufen in einen Viererzyklus. Diese Verdoppelung der Perioden setzt sich bis ins Unendliche bei $r_\infty \simeq 0.89$ fort.

Eine Verdoppelung der Periodenlänge bedeutet eine Halbierung der Frequenz; es entstehen also Subharmonische im Frequenzspektrum. Bei r_l wird der 2^{l-1}-Zyklus instabil und ein 2^l- Zyklus entsteht. Für $l \to \infty$ drängen sich die r_l-Werte immer dichter an r_∞ heran, und zwar mit folgender Gesetzmäßigkeit:

$$r_l \approx r_\infty - \frac{c}{\delta^l}\,. \tag{3.4}$$

Daraus folgt

$$\delta = \lim_{l\to\infty} \frac{r_l - r_{l-1}}{r_{l+1} - r_l}\,. \tag{3.5}$$

Die Zahl δ heißt *Feigenbaumkonstante* und hat den Wert $\delta = 4.66920...$ Sie ist deshalb so wichtig, weil sie eine universelle Größe ist; sie wird für viele verschiedene Funktionen $f(x)$ beobachtet. Auch Experimente an realen nichtlinearen Systemen, die durch Periodenverdoppelung ins Chaos übergehen, zeigen dieselbe Zahl.

Für größere r-Werte erhält man chaotische Bänder: x_n scheint zufällig in einem oder mehreren Intervallen ohne Periode hin- und herzuspringen. Dies gilt, obwohl die x_n-Werte nicht zufällig, sondern durch die wohldefinierte Parabel $f(x)$ aus Gleichung (3.1) erzeugt werden. Ein solches Verhalten nennt man, wie schon erwähnt, deterministisches Chaos.

Im chaotischen Bereich kann man an der einfachen Parabel eine wesentliche Eigenschaft des deterministischen Chaos studieren: die Empfindlichkeit auf Anfangsbedingungen. Zwei extrem dicht benachbarte Startwerte trennen sich schon nach wenigen Iterationen und springen scheinbar unabhängig voneinander in den chaotischen Bändern umher. Zur quantitativen Charakterisierung kennzeichnen wir die beiden Trajektorien durch x_n und $x_n + \varepsilon_n$. Dann gilt für $\varepsilon_0 \to 0$

$$\varepsilon_n \simeq \varepsilon_0 \, e^{\lambda \, n} \,. \tag{3.6}$$

λ heißt Ljapunow-Exponent; er mißt die Empfindlichkeit auf den Anfangszustand. Ein positiver Exponent $\lambda > 0$ bedeutet Chaos. Es gilt

$$|f^{(n)}(x_0 + \varepsilon_0) - f^{(n)}(x_0)| \simeq \varepsilon_0 \, e^{\lambda \, n} \,. \tag{3.7}$$

Im Limes $\varepsilon_0 \to 0$ gibt dies gerade die Ableitung der n-fach Iterierten $f^{(n)}(x_0) = f(f(\cdots f(x_0))\cdots)$. Mit der Kettenregel erhält man sofort

$$\lambda = \lim_{n\to\infty} \frac{1}{n} \sum_{i=0}^{n-1} \ln |f'(x_i)| \,. \tag{3.8}$$

Um λ zu bestimmen, muß also nur eine Bahn $\{x_i\}$ berechnet und dabei der Logarithmus der Steigung von f an den Stellen x_i aufsummiert werden.

Algorithmus

Mit *Mathematica* kann man die Funktion `f[x_] = 4 r x (1-x)` einfach mit

```
iterf[n_]:= Nest[f,x,n]
```

n-mal iterieren. Wesentlich schwieriger ist aber die genaue Bestimmung der Verzweigungspunkte r_l und damit der Feigenbaumkonstanten δ. r_1 und r_2 kann man durch Lösen der beiden Gleichungen $f^{(l)}(x^*) = x^*$ und $|\frac{df^{(l)}}{dx}(x^*)| = 1$ für die Unbekannten x^* und r_l finden, entweder per Hand oder mit der *Mathematica*-Funktion `FindRoot`. Aber höhere r_l-Werte konnten wir damit nicht bestimmen. Es genügt jedoch, sogenannte superstabile periodische Bahnen zu betrachten. Dies sind Bahnen, die bei $x_0 = \frac{1}{2}$ starten und nach 2^l Schritten dort wieder ankommen. Wegen $f'\left(\frac{1}{2}\right) = 0$ konvergieren Abweichungen von x_0 sehr schnell gegen den 2^l-Zyklus. In jedem Intervall $[r_l, r_{l+1}]$ gibt es nun eine superstabile Bahn der Periode 2^l bei einem Wert R_l. Die R_l-Werte skalieren dabei wie die r_l-Werte mit der Feigenbaumkonstanten δ gegen r_∞ (siehe Gleichung (3.5)).

Nun ist es wiederum nicht einfach, die Gleichung

$$f^{(2^l)}\left(\frac{1}{2}\right) = \frac{1}{2}$$

numerisch zu lösen. Da diese Gleichung noch weitere $2^l - 1$ Fixpunkte hat, oszilliert die Funktion sehr schnell und `FindRoot` versagt. Aber es gibt einen Trick: Man invertiert die obige Gleichung. Da die Inverse zu f zwei Zweige hat, muß man bei der Iteration angeben, ob x_n rechts (R) oder links (L) von $x_0 = \frac{1}{2}$ liegt. Jede periodische Bahn wird also durch eine Folge aus C, R und L bestimmt, wobei C(= center) den Startwert $x_0 = \frac{1}{2}$ markiert und R, L angeben, ob x_i rechts oder links vom Maximum liegt. Die Iteration wird also durch eine *symbolische Dynamik* beschrieben, die gewissen Regeln unterliegt.

Es gibt nun einen einfachen Algorithmus, um die Sequenz der R und L für eine gegebene Periode 2^l zu bestimmen. Wir geben ihn ohne Beweis an. Der Startwert soll bei $x_0 = \frac{1}{2}$ liegen, also fängt die Sequenz immer mit C an. Für $l = 1$ erhalten wir offensichtlich CR. Die Sequenz für $l + 1$ wird aus derjenigen für l konstruiert, indem man die l-Sequenz verdoppelt und für das mittlere C entweder L setzt, falls die Anzahl der R-Werte in der l-Sequenz ungerade ist, oder R, falls sie gerade ist. Für $l = 2$ gibt die Verdoppelung also $CRCR$, und dies wird $CRLR$, da CR nur ein R enthält. Für $l = 3$ erhält man $CRLRRRLR$.

Die inverse Iteration läßt sich am einfachsten in einem Koordinatensystem ausführen, dessen Ursprung bei $(x, y) = (\frac{1}{2}, \frac{1}{2})$ liegt und dessen beide Achsen gemeinsam so umskaliert werden, daß der Scheitel der Parabel im neuen System bei $(0, 1)$ liegt. In diesem Koordinatensystem hat die ursprüngliche Parabel $f(x) = 4\,r\,x(1 - x)$ die Form

$$g(\xi) = 1 - \mu\,\xi^2\,,$$

wobei μ und r über

$$\mu = r(4r - 2) \;\text{ bzw. }\; r = \frac{1}{4}\left(1 + \sqrt{1 + 4\mu}\right)$$

miteinander zusammenhängen. Wir bezeichnen den linken Ast der Parabel mit $g_L(\xi)$ und den rechten mit $g_R(\xi)$. Die superstabile Bahn für $l = 2$ genügt dann der Gleichung

$$g_R(g_L(g_R(1))) = 0$$

bzw.

$$1 = g_R^{-1}(g_L^{-1}(g_R^{-1}(0)))$$

mit

$$g_R^{-1}(\eta) = \sqrt{\frac{1 - \eta}{\mu}} \quad \text{und} \quad g_L^{-1}(\eta) = -\sqrt{\frac{1 - \eta}{\mu}}\,.$$

Damit erhält man schließlich folgende Bestimmungsgleichung für μ:

$$\mu = \sqrt{\mu + \sqrt{\mu - \sqrt{\mu}}}\,, \tag{3.9}$$

die man durch Iteration oder mit `FindRoot` lösen kann. Die Sequenz $CRLR$ bestimmt die Vorzeichen der geschachtelten Wurzeln in Gleichung (3.9). Nach dem anfänglichen CR, das nicht in der obigen Gleichung auftritt, gibt ein L ein positives und ein R ein negatives Vorzeichen. Dies gilt für sämtliche superstabilen periodischen Bahnen. Die Periode 5 mit der Sequenz $CRLRR$, die zwischen chaotischen Bändern liegt, findet man also bei einem μ-Wert, der folgender Gleichung genügt:

$$\mu = \sqrt{\mu + \sqrt{\mu - \sqrt{\mu - \sqrt{\mu}}}}\,.$$

Die Erzeugung der Sequenz haben wir mit *Mathematica* folgendermaßen programmiert: Für das Symbol R (L) schreiben wir die Zahlen 1 (0), die Sequenz der Länge 2^n wird als Liste `periode[n]` gespeichert. Am Anfang wird für $n = 1$ die Liste `periode[1]= {c,1}` gesetzt, und die Verdoppelung der Sequenz geschieht durch

```
periode[n_]:= periode[n]=
    Join[periode[n-1], correct[periode[n-1]]]
```

Dabei muß noch das Symbol `c` durch `0` oder `1` ersetzt werden, je nachdem, ob die Häufigkeit, mit der die Zahl 1 auftaucht (= Summe der Listenelemente), ungerade oder gerade ist.

```
 correct[list_]:=Block[{sum=0,li=list,l=Length[list]},
                      Do[sum+=li[[i]],{i,2,l}];
                      If[OddQ[sum],li[[1]]=0,li[[1]]=1];
                      li]
```

Die vielfach verschachtelte Wurzel der Bestimmungsgleichung für μ erhält man mit

```
g[n_,mu_]:=Block[{x=Sqrt[mu],l=Length[periode[n]]},
    Do[x=Sqrt[mu+(-1)^(periode[n][[i]]) x],{i,l,3,-1}];
               x]
```

Schließlich wird diese Gleichung numerisch gelöst, wobei die Genauigkeit, mit der `FindRoot` die Lösung berechnet, heraufgesetzt werden muß:

```
genau=30
maxit=30
fr[n_]:=fr[n]=(find=FindRoot[g[n,mu]==mu,
                       {mu,{15/10,16/10}},
                       AccuracyGoal->genau,
                       WorkingPrecision->genau,
                       MaxIterations->maxit];
                       mu/.find)
```

Wenn man sich die Bahnen $x_0, x_1, x_2, x_3...$ für alle Parameter r (bzw. μ) erzeugen und graphisch darstellen will, so sollte man möglichst eine schnelle Programmiersprache verwenden, da für jeden r-Wert etwa 1000 Iterationen notwendig sind. Im Prinzip ist dies aber sehr einfach: Als r-Wert wählt man z. B. die x-Koordinate der Pixel auf dem Bildschirm. Zu jedem r-Wert berechnet man zunächst etwa 100 x_n-Werte, bis die x_n dem Attraktor sehr nahe gekommen sind. Danach berechnet man weitere 1000 x_n-Werte, rechnet sie in y-Koordinaten der Pixel um und zeichnet einen Bildpunkt am Pixel (x, y). In der Sprache C lautet dies:

```
xit = 4.*r*xit*(1.-xit);
putpixel( r*MAXX, (1.-xit)*MAXY,WHITE);
```

Dabei wird der Parameter r über Tastendruck schrittweise erhöht oder erniedrigt.

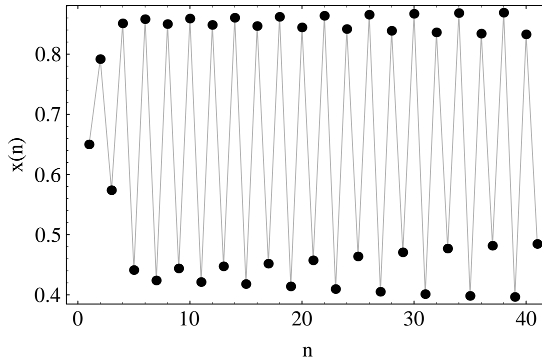
Auch die Dichte der x_n-Werte ist interessant. Um sie darzustellen, unterteilt man die x-Achse in kleine Intervalle, z. B. von der Größe der Pixelabstände. Jedes Intervall erhält eine y-Koordinate, die am Anfang auf den Wert $y = 0$ gesetzt wird. Bei der Iteration der Parabel für einen festen r-Wert fällt jeder x_n-Wert in ein solches Intervall, wobei die jeweilige y-Koordinate um eine Pixeleinheit erhöht wird und am entsprechenden (x, y)-Pixel ein Punkt gezeichnet wird. Somit erhält man ein Histogramm der x_n-Werte, das der Dichte der Punkte auf dem Attraktor entspricht. Dies geschieht mit folgender Funktion:

```
void kanal( double r)
{
    double xit=.5;
    int x,y[600],i;
    clearviewport();
    for (i=0;i<MAXX;i++) y[i]=0;
    rectangle(0,0,MAXX,300);
    while(!kbhit())
    {    xit=4.*r*xit*(1.-xit);
          x=xit*MAXX;
         y[x]++;
         putpixel(x,300-y[x],WHITE);
     }
      getch();getch();clearviewport();return;
}
```

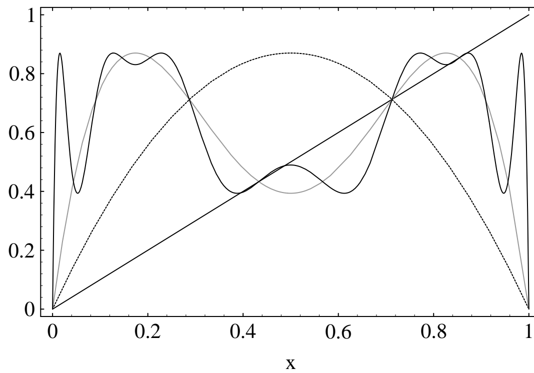
Ergebnisse

Abbildung 3.1 zeigt die Iteration von $x_0 = 0.65$ für $r = 0.87$. Die x_n-Werte laufen in einen Attraktor mit der Periode 4, der durch die Fixpunkte der vierfach iterierten

Abbildung $f(x)$ beschrieben wird. Diese sind in Abbildung 3.2 zu sehen. $f^{(4)}(x)$ schneidet die Gerade $y = x$ achtmal, nur vier davon aber sind stabile Fixpunkte.

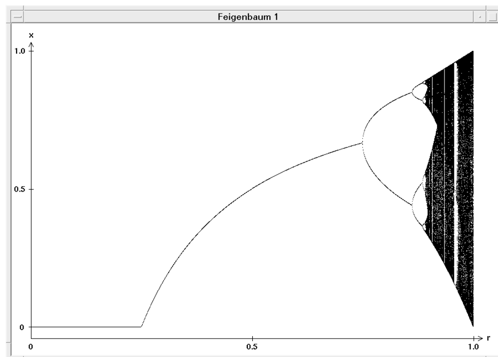


3.1 Iteration des Startwertes $x_0 = 0.65$ für den Parameter $r = 0.87$.Die x_n-Werte laufen zum Attraktor mit der Periode 4.

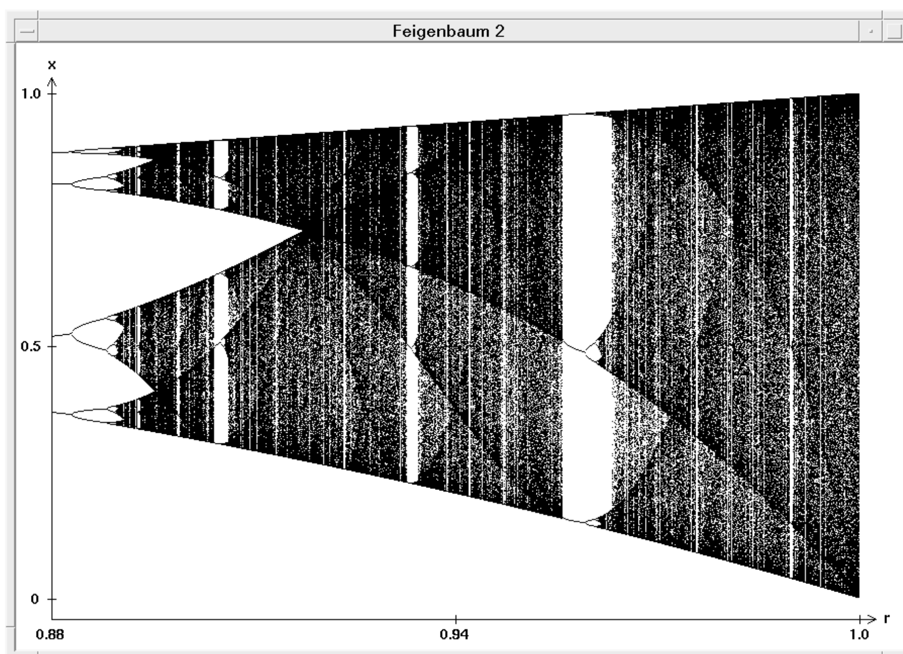


3.2 Die Funktion $f(x)$ und ihre Iterierten $f^{(2)}$ und $f^{(4)}$. Die vierfach iterierte Funktion hat acht Fixpunkte.

Erhöht man r, so entstehen Attraktoren mit den Perioden $8, 16, 32, ...$, bis bei $r_\infty = 0.89...$ der Übergang ins Chaos stattfindet. Dies sieht man deutlich in den Abbildungen 3.3 und 3.4, in der 1000 x_n-Werte nach einer Übergangszeit von etwa 100 Schritten als Funktion von r aufgetragen sind. Attraktoren der Periode p sind also als p Punkte zu sehen, während chaotische Bahnen mehrere Intervalle auf der Vertikalen füllen. Da nur höchstens 1000 Punkte für jeden r-Wert gezeichnet wurden, bekommt man auch einen Eindruck von der Dichte der x_n-Werte. Diese ist aber deutlicher in dem Histogramm in Abbildung 3.5 zu sehen. Periodische Bahnen mit der Periode p erscheinen als Linienschar mit p Spitzen, während chaotische Bahnen zwischen verschiedenen Bändern mit jeweils stetigen Dichten hin- und herspringen. Die Abbildung 3.5 zeigt die chaotische Bewegung dicht unterhalb eines Fünferzyklus; dort gibt es zwar nur ein Band, man sieht aber noch die fünf Spitzen des be-



3.3 Jede vertikale Linie enthält 1000 x_n-Werte der iterierten Parabel, wobei die ersten 100 Punkte nicht gezeigt sind. Der Parameter r variiert auf der Horizontalen von 0 bis 1.



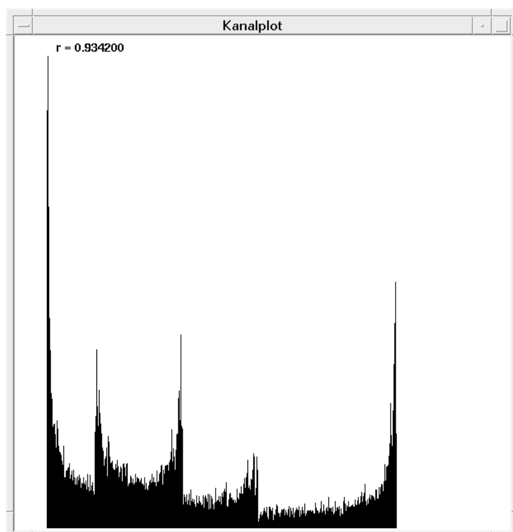
3.4 Ein Ausschnitt aus der vorigen Abbildung für $r = 0.88$ bis $r = 1$.

nachbarten Zyklus. Es läßt sich zeigen, daß für $r = 1$ nur ein Band mit der Dichte

$$\rho(x) = \frac{1}{\pi} \frac{1}{\sqrt{x(1-x)}}$$

existiert. In Abbildung 3.4 sieht man im chaotischen Bereich viele Fenster mit

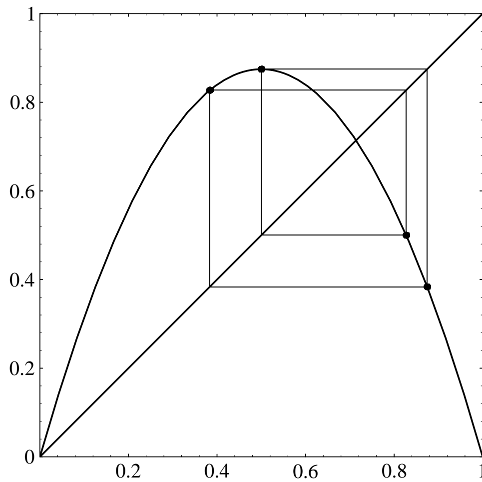
periodischen Bahnen. Das größte Fenster hat die Periode 3, das beim Parameter $r = (\sqrt{8}+1)/4$ beginnt. In diesem Fenster gibt es wieder Periodenverdoppelung: Mit wachsendem r erhält man die Perioden $3, 6, 12, 24, ...$, bis ein chaotischer Bereich erscheint. Tatsächlich kann man zeigen, daß alle Perioden vorkommen.



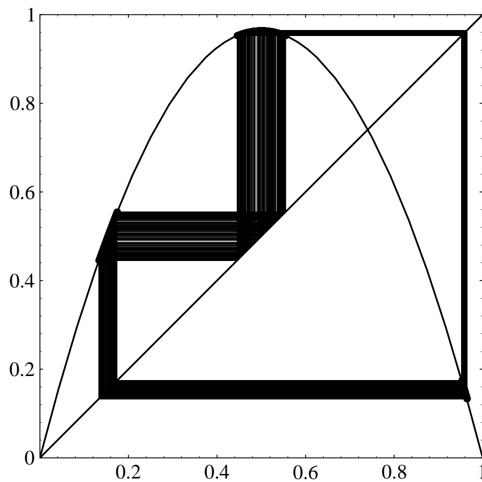
3.5 Häufigkeit der x_n-Werte bei der Iteration der Parabel für einen Parameterwert $r \simeq 0.934$ dicht unterhalb des Fensters mit der Periode 5.

Einen weiteren Eindruck der logistischen Abbildung $x_{n+1} = f(x_n)$ erhält man, indem man in der x-y-Ebene die aufeinanderfolgenden Parabelpunkte $(x_n, f(x_n))$, $(x_{n+1}, f(x_{n+1})), \ldots$ geometrisch konstruiert. Man gewinnt den jeweils nächsten Punkt dadurch, daß man vom aktuellen Punkt horizontal bis zur Winkelhalbierenden $y = x$ geht und von dort aus senkrecht bis zur Parabel $y = 4\,r\,x(1-x)$. Die nächsten zwei Abbildungen zeigen diese Konstruktion, die sich mit *Mathematica* relativ leicht ausführen läßt.
Die Skalengesetze (3.4) und (3.5) bestimmen die universelle Feigenbaumkonstante δ. Dazu berechnen wir die Parameter R_l, bei denen es superstabile Bahnen mit der Periode 2^l gibt. Mit der Methode der symbolischen Dynamik und der inversen Iteration finden wir mit 30 Stellen Genauigkeit, von denen wir hier nur 12 ausdrucken:

$$\begin{aligned}
R_{10} &= 0.892486338871\\
R_{11} &= 0.892486401027\\
R_{12} &= 0.892486414339\\
R_{13} &= 0.892486417190\\
R_{14} &= 0.892486417801\\
R_{15} &= 0.892486417932
\end{aligned}$$



3.6 Superstabiler Viererzyklus mit $r = 0.874640$.



3.7 Drei chaotische Bänder für $r = 0.9642$ kurz oberhalb des großen Fensters mit dem Dreierzyklus.

Bemerkenswert ist, daß der letzte Wert aus der Lösung einer Gleichung mit einer 32768 fach verschachtelten Wurzelfunktion gewonnen wird!
Mit

$$\delta_l = \frac{R_{l-1} - R_{l-2}}{R_l - R_{l-1}}$$

finden wir folgende Näherungswerte für δ:

$$\begin{aligned} \delta_{10} &= 4.669201134601 \\ \delta_{11} &= 4.669201509514 \end{aligned}$$

$$\begin{aligned}
\delta_{12} &= 4.669201587522\\
\delta_{13} &= 4.669201604512\\
\delta_{14} &= 4.669201608116\\
\delta_{15} &= 4.669201608892
\end{aligned}$$

Daraus schätzen wir ab, daß wir δ bis auf 9 Stellen genau berechnet haben. Dem Leser sei es als Übungsaufgabe überlassen, δ_l für $l \to \infty$ zu extrapolieren.

Zum Schluß wollen wir noch eine scheinbar einfache Frage untersuchen: Gegeben seien $x_0 = 1/3$ und $r = 97/100$. Was ist der Wert von x_{100}? Dies scheint sich leicht beantworten zu lassen; mit

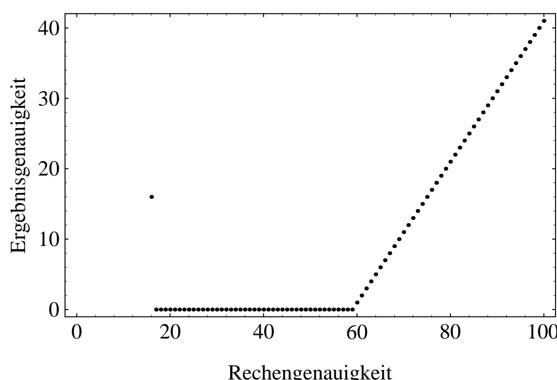
```
h[x_]:= 97/25 x(1-x)
```

gibt

```
Nest[h, N[1/3],100]
```

den Wert $x_{100} = 0.144807$. Dieser Wert aber ist falsch, der richtige Wert ist $x_{100} = 0.410091$.

Die Lösung des Widerspruchs liegt im chaotischen Verhalten der logistischen Abbildung für den gegebenen Parameterwert. Der Ljapunow-Exponent, definiert in den Gleichungen (3.6) bis (3.8), ist positiv. Also führen kleine Ungenauigkeiten in der Rechnung schon nach wenigen Iterationen zu großen Fehlern. Wegen des Be-



3.8 Die Genauigkeit von x_{100} als Funktion der Rechengenauigkeit für $r = 97/100$. Die vertikale Achse gibt die Anzahl der Stellen an, mit der *Mathematica* das Ergebnis der Iteration berechnet.

fehls `N[1/3]` berechnet *Mathematica* alles in Maschinengenauigkeit, die in unserem Beispiel 16 Stellen beträgt. Bei jedem Iterationsschritt wird gerundet, deshalb kann die Abweichung vom wahren Wert, die bei jedem Schritt etwas größer wird, nicht verfolgt werden. Mit `N[1/3,genau]` kann man dagegen mit `genau` die Genauigkeit vorgeben. In jedem Rechenschritt reduziert nun *Mathematica* die errechnete Genauigkeit, bis schließlich bei nur einer Stelle die x_n-Variablen auf den

Wert 0 gesetzt werden. Der Befehl `Precision[x]` fragt die Genauigkeit von x ab. Abbildung 3.8 zeigt das Ergebnis. Für `genau=16` wird gerundet, also ändert sich die Genauigkeit scheinbar nicht. Die richtige Berechnung zeigt aber, daß man erst bei mehr als 60 Stellen Rechengenauigkeit das korrekte Ergebnis erhält. Der Grund für den linearen Zusammenhang zwischen Rechen- und Ergebnisgenauigkeit ist uns nicht bekannt.

Übung

1. Berechnen Sie für die logistische Abbildung (3.1) den Ljapunow-Exponenten als Funktion des Parameters r.
2. Im chaotischen Bereich gibt es in jedem r-Intervall Fenster mit superstabilen periodischen Bahnen. Jede solche Bahn beginnt mit der symbolischen Dynamik CRL. Für einen Fünferzyklus gibt es daher höchstens die vier Möglichkeiten $CRLRR$, $CRLRL$, $CRLLR$ und $CRLLL$. Eine dieser Möglichkeiten kommt aber nicht vor. Berechnen Sie die Parameterwerte der drei verbleibenden Bahnen mit der Periode fünf.

Literatur

R. E. Crandall, *Mathematica for the Sciences*, Addison Wesley, 1991.

Hao Bai-Lin, *Chaos*, World Scientific, 1984.

H. J. Korsch, H.-J. Jodl, *Chaos: A Program Collection for the PC*, Springer Verlag, 1994.

E. Ott, *Chaos in Dynamical Systems*, Cambridge University Press, 1993.

H. O. Peitgen, D. Saupe, *The Science of Fractal Images*, Springer Verlag, 1988.

M. Schröder, *Fraktale, Chaos und Selbstähnlichkeit*, Spektrum Akademischer Verlag, 1994.

H. G. Schuster, *Deterministic Chaos*, Physik Verlag, 1984.

3.2 Kette auf dem Wellblech: Frenkel-Kontorova-Modell

Konkurrieren zwei physikalische Kräfte miteinander, so kann der resultierende Zustand faszinierende Eigenschaften haben. Schon beim Hofstadter-Schmetterling haben wir gesehen, wie zwei miteinander konkurrierende Längen – die vom Gitter und die vom Magnetfeld erzeugte Länge – ein überraschend komplexes Energiespektrum ergaben. Hier wollen wir nun ein einfaches mechanisches Problem mit klas-

sischen Gleichgewichtsbedingungen betrachten: eine Kette aus Teilchen, die durch lineare Kräfte miteinander wechselwirken und sich zusätzlich in einem periodischen Potential aufhalten. Wir untersuchen nur die Ruhelagen der Kettenglieder. Sowohl die Federkräfte als auch die äußeren Kräfte möchten den Teilchen einen eigenen, aber unterschiedlichen Abstand zueinander aufzwingen. Es stellt sich heraus, daß die stabilen Zustände der Kette durch Iterationen einer nichtlinearen zweidimensionalen Abbildung beschrieben werden, deren Bahnen vielfältige Formen annehmen können.

Physik

Unser System ist eine eindimensionale Anordnung von unendlich vielen Punktmassen, deren Ruhelagen durch die Plätze $x_n (n \in \mathbb{Z})$ gekennzeichnet seien. Die Kräfte auf die Teilchen sollen zum einen von einem äußeren periodischen Potential $V(x)$ und zum anderen von einem Wechselwirkungspotential $W(x_n - x_{n-1})$ zwischen benachbarten Teilchen herrühren, so daß die Energie eines langen Teilstückes der Kette, das die Plätze $x_{M+1}, x_{M+2}, \ldots, x_N$ umfaßt, durch

$$H_{MN} = \sum_{n=M+1}^{N} [V(x_n) + W(x_n - x_{n-1})] \tag{3.10}$$

gegeben ist. Alle Längen x werden in Einheiten der Periode des äußeren Potentials gemessen. Das bedeutet

$$V(x+1) = V(x)\,. \tag{3.11}$$

In der Ruhelage kompensieren sich alle Kräfte auf jedes Teilchen, es gilt also

$$0 = \frac{\partial H_{MN}}{\partial x_n} = V'(x_n) + W'(x_n - x_{n-1}) - W'(x_{n+1} - x_n)\,. \tag{3.12}$$

Mit der Definition

$$p_n = W'(x_n - x_{n-1}) \tag{3.13}$$

folgt hieraus

$$\begin{aligned} p_{n+1} &= p_n + V'(x_n)\,, \\ x_{n+1} &= x_n + (W')^{-1}(p_{n+1})\,, \end{aligned} \tag{3.14}$$

oder als Rückwärtsrekursion aufgelöst,

$$\begin{aligned} x_{n-1} &= x_n - (W')^{-1}(p_n)\,, \\ p_{n-1} &= p_n - V'(x_{n-1})\,. \end{aligned} \tag{3.15}$$

Aus (x_0, p_0) – äquivalent dazu ist die Vorgabe von (x_{-1}, x_0) – erhält man die Paare (x_1, p_1) und (x_{-1}, p_{-1}), daraus die Werte (x_2, p_2) und (x_{-2}, p_{-2}) und so fort. Die Gleichungen (3.14) beschreiben daher eine nichtlineare Iteration in der x-p-Ebene. Jeder Startpunkt (x_{-1}, x_0) bestimmt eindeutig einen Zustand $\{\ldots, x_{-2}, x_{-1}, x_0, x_1, x_2, \ldots\}$ der gesamten Kette, bei dem sich jedes Teilchen in Ruhelage befindet.

Die Funktionaldeterminante der Abbildung $(x_n, p_n) \to (x_{n+1}, p_{n+1})$ hat den Wert eins, $\left|\frac{\partial(x_{n+1}, p_{n+1})}{\partial(x_n, p_n)}\right| = 1$; die Abbildung ist daher flächenerhaltend. Obwohl ein Ausschnitt aus der x-p-Ebene durch die Abbildung deformiert wird, hat sein Bild denselben Flächeninhalt wie der ursprüngliche Ausschnitt. Flächenerhaltender Fluß im Phasenraum ist aus der klassischen Mechanik bekannt (Satz von Liouville), und tatsächlich kann man (3.14) als Momentaufnahmen (Poincaré-Schnitt, siehe Abschnitt 4.2) einer kontinuierlichen Dynamik in einem dreidimensionalen Phasenraum auffassen.

Bisher wurden die Potentiale V und W noch nicht spezifiziert, doch nun sollen sie auf ein äußeres Cosinus-Potential und lineare Federkräfte beschränkt werden:

$$\begin{aligned} V(x) &= \frac{K}{(2\pi)^2}\left(1-\cos(2\pi x)\right) , \\ W(\Delta) &= \frac{1}{2}(\Delta-\sigma)^2 . \end{aligned} \tag{3.16}$$

K bestimmt die Stärke des periodischen Potentials, und σ ist der Abstand der Teilchen für $K = 0$ in Einheiten der Periodenlänge des Potentials. Wir modifizieren Gleichung (3.13), die $p_n = x_n - x_{n-1} - \sigma$ liefert, geringfügig und setzen

$$p_n = x_n - x_{n-1} . \tag{3.17}$$

Statt (3.14) erhalten wir dann

$$\begin{aligned} p_{n+1} &= p_n + \frac{K}{2\pi}\sin(2\pi x_n) , \\ x_{n+1} &= x_n + p_{n+1} . \end{aligned} \tag{3.18}$$

Diese Abbildung, die in der Literatur den Namen Standard- oder Chirikov-Abbildung hat, ist in vielen Publikationen untersucht worden. Der Parameter σ kommt nicht mehr in der Iteration, sondern nur noch in der Energie vor,

$$H_{MN} = K \sum_{n=M+1}^{N} \frac{1}{(2\pi)^2}\left(1-\cos 2\pi x_n\right) + \sum_{n=M+1}^{N} \frac{1}{2}(p_n-\sigma)^2 . \tag{3.19}$$

Für gegebene Parameter K und σ ist die mittlere Energie pro Teilchen

$$h = \lim_{N-M\to\infty} \frac{1}{N-M} H_{MN} . \tag{3.20}$$

h ist eine Funktion von x_0 und p_0, wobei der Grundzustand durch den Punkt (x_0, p_0) mit der kleinsten Energie h bestimmt ist.

Eine interessante Eigenschaft eines Zustandes wird durch die sogenannte Windungszahl w beschrieben, die als mittlerer Abstand benachbarter Teilchen definiert ist:

$$w = \langle x_{n+1} - x_n \rangle = \lim_{N-M\to\infty} \frac{x_N - x_M}{N - M} \,. \tag{3.21}$$

Falls ein Zustand relativ zum Gitter periodisch ist (ein kommensurabler Zustand), so gibt es ganze teilerfremde Zahlen P und Q mit $x_{n+Q} = x_n + P$. Daraus folgt

$$w = \frac{x_{n+Q} - x_n}{Q} = \frac{P}{Q} \,. \tag{3.22}$$

Ein kommensurabler Zustand hat daher einen rationalen mittleren Teilchenabstand w. Ein Zustand mit irrationalem w dagegen rastet nicht ins Gitter ein; dies zeigt sich ebenfalls in einigen anderen physikalischen Eigenschaften. Wie beim Hofstadter-Schmetterling unterscheidet die Physik auch hier zwischen rationalen und irrationalen Zahlen!

Algorithmus

Die Programmierung der Abbildung (3.18) ist sehr einfach. In *Mathematica* definieren wir, weil es um numerische Manipulationen geht, `pi=N[Pi]` und die Funktion `t` durch

```
t[{x_,p_}] = {x + p + k/(2 pi) Sin[2 pi x],
              p + k/(2 pi) Sin[2 pi x]}
```

Iterationen können einfach durch `Nest` oder `NestList` ausgeführt werden:

```
list[x0_,p0_]:= NestList[t,{x0,p0},nmax]
```

Das Ergebnis von `list[ ]` ist eine Liste von `nmax+1` Punkten in der x-p-Ebene und beschreibt damit den physikalischen Zustand für die gegebenen Startwerte x_0 und $p_0 = x_0 - x_{-1}$. Nach Vorgabe der Lagen x_0 und x_{-1} zweier benachbarter Teilchen liefert `list[]` also die Positionen der weiteren Teilchen.

Ein anschauliches Bild der resultierenden Zustände erhalten wir, indem wir auf die Kurve $V(x)$ die Teilchen mit den Koordinaten $\{x_n, V(x_n)\}$ legen und diese Punkte mit `ListPlot` zeichnen. Die Koordinaten x_n werden aus dem Ergebnis von `list[..]` mit dem Befehl

```
xlist[x0_,p0_]:= Map[First,list[x0,p0]]
```

herausgezogen.

Schon beim Pendel haben wir gesehen, daß es nützlich ist, anstatt der Funktion x_n das Phasenraumdiagramm in der x-p-Ebene zu betrachten (beim Pendel statt $\varphi(t)$ das $\dot{\varphi}(\varphi)$-Diagramm). Wir brauchen dazu aber nur x und p jeweils im Intervall $[0, 1]$ darzustellen, denn die Abbildung (3.18) gibt dieselben Werte (modulo 1), wenn zu x_n oder p_n jeweils der Wert 1 addiert wird.

Die Modulo-Abbildung kann man mit `Map` in eine Liste hineinziehen und auf die einzelnen Koordinaten wirken lassen:

```
tilde[{x_,p_}]:= {Mod[x,1],Mod[p,1]}
listt[x0_,p0_]:= Map[tilde,list[x0,p0]]
```

Mit `ListPlot` wird das Phasenraumdiagramm erzeugt. Die Energie und die Windungszahl sind leicht aus dem Zustand `ll1 = list[x0,p0]` zu berechnen:

```
de[{x_,p_}]:=k/(2 pi)^2 (1-Cos[2 pi x])+(p-sigma)^2/2

h[x0_:.0838,p0_]:=
        ( ll1=list[x0,p0];
          ll2=Map[de,ll1];
          Apply[Plus,ll2]/Length[ll2])

wind[x0_:.0838,p0_]:=
        ( w1=xlist[x0,p0]; (w1[[-1]] - w1[[1]])/nmax)
```

Allerdings dauert die Berechnung mit *Mathematica* wieder sehr lange. Deshalb wollen wir uns das Phasenraumdiagramm mit einem C-Programm noch einmal auf dem Bildschirm direkt erzeugen. Außerdem sollen die Energie und die Windungszahl berechnet werden. Der Startpunkt wird vom Programm zufällig ausgewählt.

Die Funktion (3.18) lautet in C:

```
pneu= p+K/2./pi*sin(2.*pi*x);
xneu= x+pneu;
```

Eigentlich sollten in dieser innersten Schleife, in der diese Befehle sehr oft ausgeführt werden, die Ausdrücke `K/2./pi` und `2.*pi` nicht jedesmal neu berechnet, sondern durch Konstanten ersetzt werden. Bei unserem Programm benötigt die Graphik aber so viel Computerzeit, daß wir auf die Optimierung des Algorithmus verzichten. Die x-p-Werte werden nun in Bildkoordinaten umgerechnet:

```
xb=fmod(xneu,1.)*xmax+10;
yb=fmod(pneu,1.)*ymax+100;
```

und es wird ein Bildpunkt an die Stelle (`xb`,`yb`) gezeichnet; z. B. auf dem PC mit

```
putpixel(xb,yb,color);
```

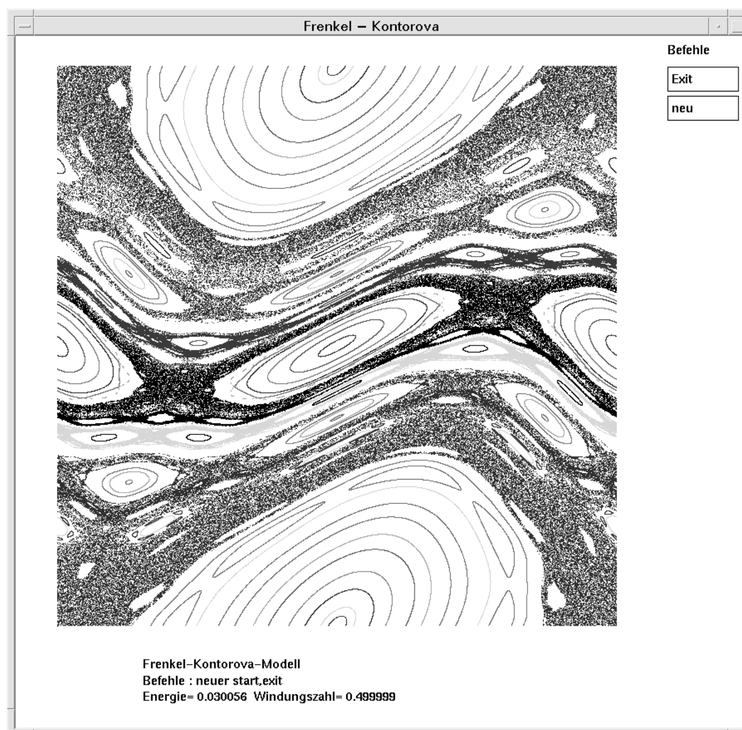
Bei jedem Aufruf wird die Farbe neu gewählt:

```
color= random(getmaxcolor())+1;
```

So erhält man auf dem Bildschirm eine Graphik mit verschiedenen Trajektorien, deren jede einen Zustand repräsentiert, in jeweils unterschiedlichen Farben.

Ergebnisse

Abbildung 3.9 zeigt die Ergebnisse der Iteration der Gleichung (3.18) für $K = 1$. Dabei wurden die Startwerte (x_0, p_0) zufällig im Einheitsquadrat gewählt, und es



3.9 Trajektorien in der x-p-Ebene. Auf der Workstation kann ein neuer Startwert mit der Maus angeklickt werden, und die Trajektorie wird in einer neuen Farbe gezeichnet.

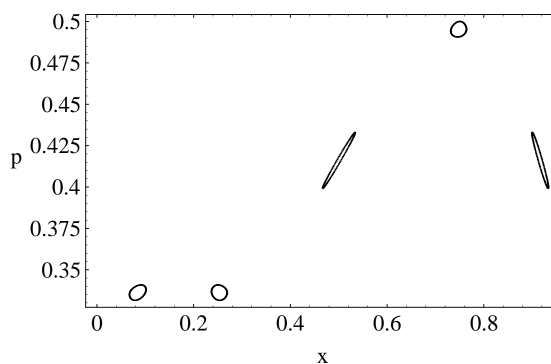
wurden jeweils 10000 Werte für (x_n, p_n) mit dem C-Programm berechnet und gezeichnet. Dieses Bild ist unabhängig von dem Verhältnis σ der konkurrierenden

Längen, da dieser Parameter gar nicht in der iterierten Abbildung vorkommt. Zu jeder Trajektorie gehört eine Windungszahl w, die vom Startpunkt und dem periodischen Potential, nicht aber von σ abhängt. Erst in der Energie pro Teilchen

$$h = \frac{1}{N} \sum_{n=1}^{N} \left[\frac{K}{(2\pi)^2}(1 - \cos(2\pi x_n)) + \frac{1}{2}(p_n - \sigma)^2 \right]$$

taucht die Gleichgewichtskonstante σ auf.

Es gibt drei unterschiedliche Bahnen in der Abbildung 3.9: null-, ein- und zweidimensionale. Die nulldimensionalen Trajektorien bestehen aus endlich vielen Punkten, die z. B. in der Mitte der Inseln zu finden sind. Sie sind die kommensurablen Zustände mit rationaler Windungszahl $w = P/Q$ und der Periode Q. P zeigt an, in welcher Reihenfolge die Q verschiedenen Punkte (x_n, p_n) durchlaufen werden. Um diese kommensurablen Zustände herum sieht man geschlossene Bahnen, also eindimensionale Trajektorien. Diese Zustände sehen fast so aus wie ihre kommensurablen Verwandten in der Mitte der Bahnen, allerdings schwankt jedes x_n um den entsprechenden kommensurablen Wert mit einer Frequenz, die inkommensurabel zur Periode des Potentials ist.



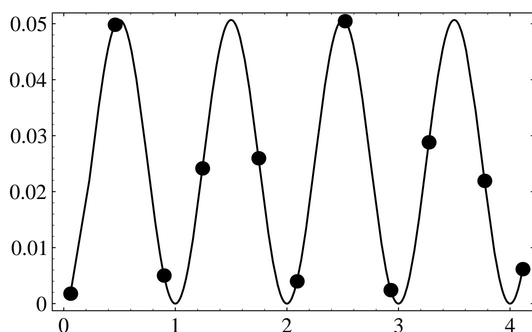
3.10 Die Startwerte $(x_0, p_0) = (0.08, 0.34)$ ergeben eine fastperiodische Bahn im Phasenraum (x, p) (modulo 1).

Wir wollen uns dies mit *Mathematica* für $w = \frac{2}{5}$ ansehen. Zunächst lösen wir mit `FindRoot` die Gleichung

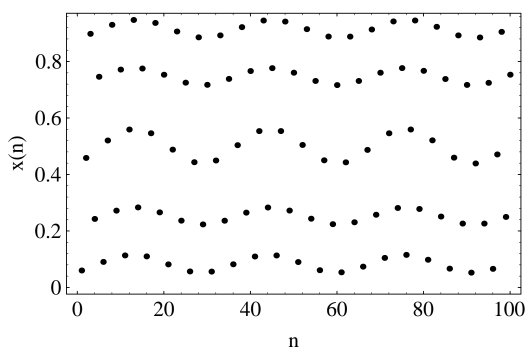
```
Mod[t[t[t[t[t[{x0,p0}]]]]],1] == {x0,p0}
```

Allerdings muß man den Startwert ziemlich dicht bei der Lösung angeben, sonst findet die Routine keine Lösung. Wir erhalten z. B. für (x_0, p_0) den Wert $(0.0838255, 0.336171)$ als einen der $Q = 5$ Fixpunkte der fünffach iterierten Funktion `t[x,p]`. Abbildung 3.10 zeigt nun einen inkommensurablen Zustand mit $(x_0, p_0) = (0.08, 0.34)$, also dicht beim periodischen Zustand in der Mitte der fünf Inseln. Er hat dieselbe Windungszahl $w = 0.4$ und eine etwas höhere Energie als der periodische Zustand. Abbildung 3.11 zeigt den fastperiodischen Zustand im Ortsraum. Man sieht

beim Teilchen in der Potentialmulde, daß die Tallage ein wenig um das Minimum oszilliert. Etwa dieselbe Tallage wird nach $Q = 5$ Iterationen wieder erreicht. Man sieht die Oszillation noch besser in Abbildung 3.12, in der die x_n-Werte direkt gegen n aufgetragen sind. Neben der Periode mit $Q = 5$ gibt es noch eine Welle mit $Q \simeq 30$.



3.11 Dieselbe fastperiodische Trajektorie wie in Abbildung 3.10, jedoch auf das Potential $V(x)$ gezeichnet.

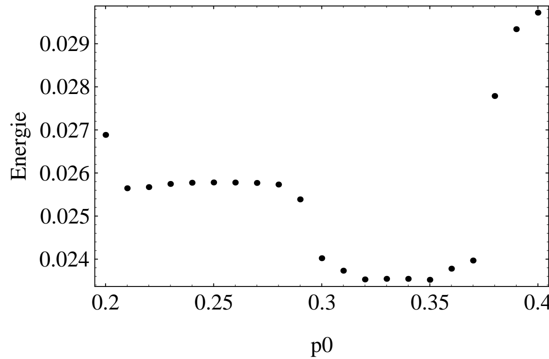


3.12 Wie in den Abbildungen 3.10 und 3.11, jedoch als Auslenkung x (modulo 1) des n-ten Atoms gezeichnet.

Neben den null- und eindimensionalen Bahnen gibt es in Abbildung 3.9 noch Zustände, die ganze Flächenstücke füllen. Solche Bahnen springen chaotisch auf dem Bildschirm hin und her. Manche Bahnen durchmessen offenbar den ganzen x- und p-Bereich. Letzteres gilt nicht für kleine K-Werte. Für $K < K_c = 0.971635...$ gibt es eindimensionale Bahnen, die das gesamte Bild horizontal durchlaufen (sogenannte KAM-Trajektorien). Man kann zeigen, daß chaotische Bahnen die eindimensionalen nicht überspringen können, sie sind also auf ein p-Intervall zwischen zwei KAM-Trajektorien beschränkt. Bei K_c verschwindet die letzte KAM-Trajektorie, und zwar diejenige mit der „irrationalsten“ Windungszahl, dem inversen goldenen Schnitt $w = 2/(1 + \sqrt{5}) = 0.618\ldots$

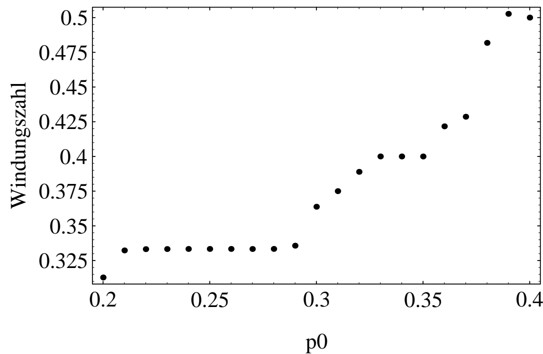
Wir sind aber nicht allein an den Eigenschaften der Standard-Abbildung interes-

siert, sondern suchen denjenigen Zustand, der die tiefste Energie zu einem gegebenen σ-Wert hat. Wir können das Minimum im Prinzip finden, indem wir $h(x_0, p_0)$ für verschiedene (x_0, p_0)-Werte berechnen. In der Praxis gibt es aber bessere Methoden, die wir hier nicht vorstellen (siehe z. B. Griffiths). Abbildung 3.13 zeigt



3.13 Energie für $\sigma = 0.4$ eines kräftefreien Zustandes mit Startwert $x_0 = 0.0838$ als Funktion von p_0.

$h(x_0, p_0)$ in einem Schnitt durch die x_0-p_0-Ebene für festes $x_0 = 0.0838$ und $\sigma = 2/5$. Die zugehörigen Windungszahlen w sind in Abbildung 3.14 zu sehen. Für $p_0 \simeq 0.35$ nimmt h im betrachteten Intervall ein Minimum an. Wenn wir das globale Minimum der Energie finden wollen, müssen wir die Funktion $h(x_0, p_0)$ in der gesamten x_0-p_0-Ebene absuchen. Am Minimum scheint w in den Wert $w = \sigma = 0.4$ „eingerastet" zu sein. Wenn man für alle σ-Werte das genaue



3.14 Mittlerer Nachbarabstand w für die kräftefreien Zustände aus Abbildung 3.13 .

Minimum von $h(x_0, p_0)$ mit der entsprechenden Windungszahl w bestimmt, so findet man ein faszinierendes Verhalten der Funktion $w(\sigma)$, das auch als *Teufelstreppe* bezeichnet wird: w „rastet" dabei in jede rationale Zahl P/Q in Stufen ein, deren Breite mit wachsendem Q-Wert abnimmt. $w(\sigma)$ hat daher unendlich viele Stufen, und zwischen je zwei Stufen liegen wieder unendlich viele. Dennoch ist $w(\sigma)$ stetig;

es gibt auch zu allen irrationalen Windungszahlen einen σ-Wert und einen zugehörigen Grundzustand. Dieses Verhalten kann mathematisch bewiesen werden.

Die Funktion $w(\sigma)$, die sich aller Vorstellungskraft widersetzt, entsteht aus dem Wettbewerb zweier Längen. Die zu den rationalen w-Werten gehörenden kommensurablen Grundzustände entsprechen einzelnen Punkten in der x-p-Ebene, während man zeigen kann, daß die inkommensurablen Grundzustände keinen flächenfüllenden chaotischen Trajektorien sondern nur Linien entsprechen. Inkommensurable Zustände können übrigens ohne Energieänderung über das Cosinus-Potential gezogen werden.

Übung

Zeichnen Sie für verschiedene K-Werte von Gleichung (3.18) Trajektorien in der x-p-Ebene. Für $K = 0$ sollten Sie die Bahnen mit der Hand zeichnen können. Beobachten Sie, wie die geschlossenen Bahnen, die sogenannten KAM-Trajektorien, mit wachsendem Wert von K in Inseln oder chaotische Bahnen übergehen, und berechnen Sie die Windungszahlen der übriggebliebenen KAM-Trajektorien. Versuchen Sie, kurz vor K_c die letzte geschlossene Bahn zu finden.

Literatur

H. J. Korsch, H.-J. Jodl, *Chaos: A Program Collection for the PC*, Springer Verlag, 1994.

E. Ott, *Chaos in Dynamical Systems*, Cambridge University Press, 1993.

H. G. Schuster, *Deterministic Chaos*, Physik Verlag, 1984.

R.B. Griffiths, *Frenkel-Kontorova models of commensurate-incommensurate phase transitions*, Fundamental Problems in Statistical Mechanics VII, p. 69-110, H. van Beijeren, Editor, Elsevier Science Publisher, 1990.

J.M. Greene, *A method for determining a stochastic transition*, J. Math. Phys. **20**, 1183 (1979).

3.3 Fraktales Gitter

Ein einfaches Würfelspiel und eine simple geometrische Wiederholung: Beides führt zu einem seltsamen Gebilde, das weniger als eine Fläche, aber mehr als eine Linie ist, ein sogenanntes Fraktal mit einer Dimension von $D = \log 3/\log 2 = 1.58\ldots$ Überraschenderweise sind Fraktale sehr häufig in der Natur anzutreffen. Küstenlinien, Gebirgszüge, Blutgefäße, Flußläufe, Schwankungen von Börsenkursen und

Flußpegel, alles dies kann durch eine gebrochene Dimension D beschrieben werden. Neben dem geometrischen Spiel werden wir in den folgenden Abschnitten noch Aggregate und Perkolationscluster als weitere Beispiele für diese merkwürdigen selbstähnlichen Gebilde kennenlernen.

Hier wollen wir uns zunächst mit der Definition der fraktalen Dimension beschäftigen. Danach stellen wir zwei Programme vor, die auf verschiedene Arten das Sierpinski-Gerüst konstruieren, das oft als Standard-Beispiel für Fraktale beschrieben wird.

Physik

Die räumliche Dimension D eines Objektes kann man durch die Beziehung zwischen der Anzahl seiner Elemente, die seine Masse M ausmachen, und seiner Ausdehnung L angeben:

$$M \propto L^D \,. \tag{3.23}$$

Wenn man also die Massen zweier gleichartiger homogener Würfel mit den Kantenlängen L und $2L$ vergleicht, erhält man

$$\frac{M_2}{M_1} = \left(\frac{2L}{L}\right)^D = 2^D \tag{3.24}$$

mit $D = 3$. Bei Quadraten gibt dies offensichtlich $D = 2$ und bei Strecken $D = 1$. L kann auch der Radius von Kugeln oder Scheiben sein oder jede andere charakteristische Länge des betrachteten Objektes; die Masse ist natürlich proportional zur Zahl der Teilchen. Auf jeden Fall gilt

$$D = \lim_{L \to \infty} \frac{\log M}{\log L} \,. \tag{3.25}$$

Es gibt auch einen anderen Weg, um die Dimension D zu bestimmen. Dazu überdecken wir das Objekt vollständig mit kleinen Würfeln der Kantenlänge ε. Es sei $N(\varepsilon)$ die kleinste Anzahl solcher Würfel. Dann gilt $N(\varepsilon) \propto \varepsilon^{-D}$ bzw.

$$D = -\lim_{\varepsilon \to 0} \frac{\log N(\varepsilon)}{\log \varepsilon} \,. \tag{3.26}$$

Für einen Würfel benötigen wir $(L/\varepsilon)^3$ überdeckende Kästchen, während die Anzahl für ein Quadrat $(L/\varepsilon)^2$ und für eine Linie $(L/\varepsilon)^1$ beträgt.

Wenn wir also z. B. die Länge einer Grenze auf einer Landkarte bestimmen wollen, so können wir einen kleinen Maßstab mit der Länge ε nehmen und die Zahl

$N(\varepsilon)$ der Schritte bestimmen, die wir zum Abschreiten der Grenzlinie benötigen. Dann sollte eigentlich

$$L = \varepsilon\, N(\varepsilon) \tag{3.27}$$

ein Maß für die Länge sein, denn wir erwarten als Grenze ein eindimensionales Objekt mit $D = 1$. Auf diese Weise haben die Spanier die Länge ihrer Grenze zu Portugal mit 987 km angegeben, während die Portugiesen aber dieselbe Grenze mit 1214 km bestimmt haben. Ein kleinerer Maßstab ε benötigt offenbar mehr überdeckende Schritte als L/ε, also kann nicht $D = 1$ gelten. Deshalb wird Gleichung (3.27) eine Länge L ergeben, die von ε abhängt. Eine genauere Analyse der Daten zeigt folgendes: Eine Grenzlinie ist ein Fraktal mit $1 < D < 2$, daher gibt Gleichung (3.27) $L \propto \varepsilon^{1-D}$, also eine scheinbar divergierende Länge für $\varepsilon \to 0$. Die Grenze hat gar keine wohldefinierte Länge! Dies kann natürlich nur für Maßstäbe ε gelten, die größer als die kleinste (z. B. die Meßlatte der Vermessungsingenieure) und kleiner als die größte Länge (z. B. die Ausdehnung von Portugal) sind.

Ein anderes einfaches Beispiel, bei dem die fraktale Dimension D nicht mit der Einbettungsdimension d, also der Dimension des Raumes übereinstimmt, in dem sich das Objekt befindet, ist eine Zufallsbewegung im d-dimensionalen Raum. Dieser Zufallsweg wird auch als einfaches Modell für ein Polymermolekül diskutiert, das aus vielen Monomeren besteht. Dabei nehmen wir an, daß der Abstand benachbarter Monomere konstant ist, die Winkel zwischen je drei aufeinanderfolgenden aber unabhängig und zufällig verteilt sind. Die Verbindungsvektoren $\boldsymbol{r_i}$ benachbarter Monomere sind dann Zufallsvektoren mit leicht zu berechnenden Eigenschaften. Im Mittel $\langle ... \rangle$ verschiedener Molekülkonfigurationen gilt:

$$\langle \boldsymbol{r}_i \rangle = 0\,, \quad \langle \boldsymbol{r}_i \cdot \boldsymbol{r}_j \rangle = \langle \boldsymbol{r}_i \rangle \cdot \langle \boldsymbol{r}_j \rangle = 0 \text{ für } i \neq j \quad \text{und} \quad \langle {\boldsymbol{r}_i}^2 \rangle = a^2\,,$$

wobei a der Abstand benachbarter Monomere sein soll. Für den Vektor $\boldsymbol{R}$ zwischen Anfang und Ende des Moleküls, das aus $N + 1$ Monomeren bestehen möge, gilt:

$$\boldsymbol{R} = \sum_{i=1}^{N} \boldsymbol{r}_i\,,$$

und damit

$$\langle \boldsymbol{R} \rangle = \sum_{i=1}^{N} \langle \boldsymbol{r_i} \rangle = 0\,,$$

$$\langle \boldsymbol{R}^2 \rangle = \sum_{i=1}^{N} \sum_{j=1}^{N} \langle \boldsymbol{r_i} \cdot \boldsymbol{r_j} \rangle = \sum_{i=1}^{N} \langle {\boldsymbol{r_i}}^2 \rangle + \sum_{i \neq j} \langle \boldsymbol{r_i} \cdot \boldsymbol{r_j} \rangle = N a^2\,.$$

Da die Masse M proportional zur Zahl N der Monomere und die lineare Ausdehnung L proportional zu $\sqrt{\langle \boldsymbol{R}^2 \rangle}$ wächst, gilt also

$$M \propto L^2$$

und mit Gleichung (3.23) folglich $D = 2$. Ein solcher Zufallsweg, im Fachjargon *random walk* genannt, ist daher immer ein zweidimensionales Objekt, unabhängig von der Dimension d des Raumes, in dem er erzeugt wird. Die fraktale Dimension wird allerdings verkleinert, wenn man die Ausdehnung der einzelnen Monomere mit berücksichtigt. Diesen Effekt werden wir im Abschnitt 5.4 besprechen.

Algorithmus und Ergebnis

Nun wollen wir uns die anfangs erwähnten Spiele genauer anschauen. Zunächst der Algorithmus für das Würfelspiel:

- Wähle drei Punkte $\boldsymbol{p}_1, \boldsymbol{p}_2$ und $\boldsymbol{p}_3$ und einen Startpunkt $\boldsymbol{q}_0$ beliebig in der Ebene.
- Ausgehend vom Punkt $\boldsymbol{q}_n$, konstruiere den nächsten Punkt $\boldsymbol{q}_{n+1}$ auf folgende Weise: Wähle zufällig (z. B. mit einem Würfel) einen der drei Punkte $\boldsymbol{p}_i$. Dann berechne die Mitte der Verbindungslinie zwischen $\boldsymbol{q}_n$ und $\boldsymbol{p}_i$, also

$$\boldsymbol{q}_{n+1} = (\boldsymbol{q}_n + \boldsymbol{p}_i)/2 \,.$$

- Iteriere die obige Gleichung unendlich oft.

Welches Muster erzeugen die Punkte $\{\boldsymbol{q}_0, \boldsymbol{q}_1, \boldsymbol{q}_2, \ldots\}$ in der Ebene? Der Algorithmus, den wir wegen der höheren Rechengeschwindigkeit in C schreiben, ist leicht zu programmieren. Jeden Punkt definieren wir als Struktur mit jeweils zwei ganzzahligen Variablen x und y, die die Pixelkoordinaten auf dem Bildschirm angeben.

```
struct{int x, int y} qn={20,20}, pw,
       p[3]={{10,10},{MAXX-10,10},{MAXX/3,MAXY-10}} ;
```

Die Startwerte sind in der Typendeklaration direkt mit angegeben. In jedem Iterationsschritt wird nun einer der drei Punkte `p[i]` durch das Ziehen einer Zufallszahl $i \in \{0, 1, 2\}$ ausgewählt,

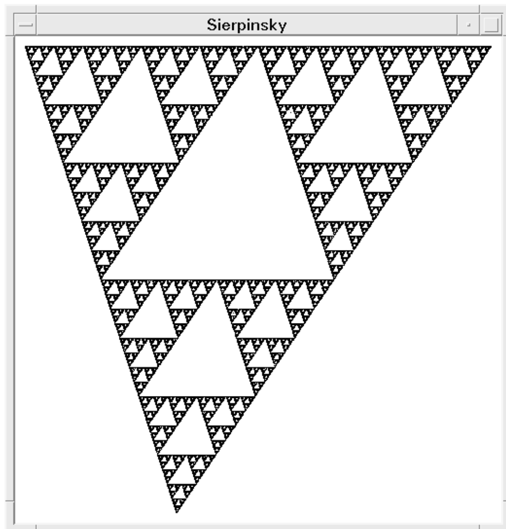
```
pw = p[(3*rand())/RAND_MAX]
```

und ein neuer Punkt berechnet:

```
qn.x = (pw.x + qn.x)/2 ;
qn.y = (pw.y + qn.y)/2 ;
```

Dann wird an dieser Stelle ein Bildpunkt gezeichnet, z. B. auf dem PC mit

```
putpixel(qn.x, qn.y, WHITE) ;
```



3.15 Punkte in der Ebene, die durch das im Text beschriebene Würfelspiel erzeugt wurden. Die Ecken des Sierpinski-Dreiecks sind die Punkte `p[0]`,`p[1]` und `p[2]`.

Das Ergebnis zeigt Abbildung 3.15. Erstaunlicherweise führt die Zufallsbewegung des Punktes zu einer offenbar regelmäßigen Struktur, die aus ineinandergeschachtelten Dreiecken besteht. Die Struktur ist selbstähnlich: Jedes Dreieck sieht gleich aus, unabhängig von seiner Größe (natürlich nur für Dreicke, die wesentlich größer als die Pixelabstände sind). Die Struktur ist aber fast leer, jedes Dreieck hat beliebig viele Löcher auf jeder Längenskala. Jeder Startpunkt landet nach wenigen Iterationen auf diesem seltsamen Gitter; das Gerüst, das auch den Namen *Sierpinski-gasket* hat, ist also ein Attraktor der Dynamik für die gesamte Ebene.

Das zweite geometrische Spiel lautet wie folgt:

- Starte mit einem Dreieck,
- entferne aus seiner Mitte ein Dreieck so, daß drei gleiche Dreiecke übrigbleiben,
- iteriere dies unendlich oft für alle jeweils übrigbleibenden Dreiecke.

Wegen der einfachen Graphikbefehle wollen wir dies in *Mathematica* programmieren. Jedes Dreieck wird durch eine Liste von drei Punkten in der Ebene beschrieben. Das Startdreieck lautet:

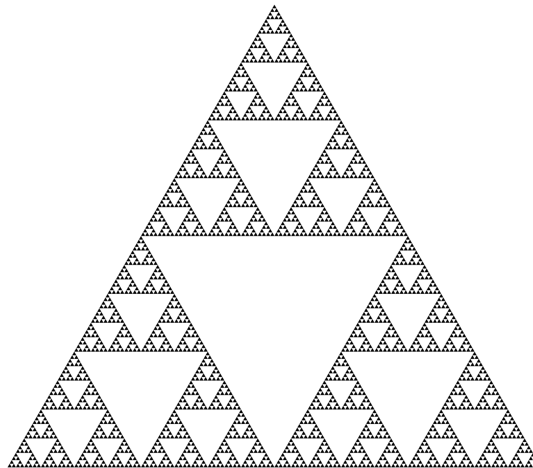
```
list={{{0.,0.},{.5,N[Sqrt[3/4]]},{1.,0.}}}
```

Jedes Dreieck wird nun dadurch verdreifacht, daß man es erst mit einem Faktor $\frac{1}{2}$ staucht und dann um eine Kantenlänge entlang zweier Kanten verschiebt:

```
verv[d_]:= Block[ {d1,d2,d3},
                  d1={d[[1]],(d[[2]]+d[[1]])*.5,
                             (d[[3]]+d[[1]])*.5};
                  d2=d1+Table[(d1[[3]]-d1[[1]]),{3}];
                  d3=d1+Table[(d1[[2]]-d1[[1]]),{3}];
                  {d1,d2,d3}
                ]
```

Diese Funktion wird mit dem Befehl `Map` auf die ganze Liste von Dreiecken wiederholt angewendet. Der Befehl `Polygon` produziert schließlich Dreiecke als Graphikobjekte, die mit `Show` gezeichnet werden:

```
plot1:= Block[ {listzw,plotlist},
                    listzw=Map[verv,list];
                    list=Flatten[listzw,1];
                    plotlist=Map[Polygon,list];
                    Show[Graphics[plotlist],
                         AspectRatio -> Automatic]
             ]
```



3.16 Durch wiederholtes Herausschneiden von Dreiecken wird ein Fraktal mit der Dimension $D \simeq 1.58$ erzeugt.

Abbildung 3.16 zeigt das Ergebnis. Wir erhalten offenbar die gleiche Struktur wie im vorherigen Würfelspiel. Jetzt können wir aber leicht einsehen, daß das selbstähnliche Gerüst im Limes unendlich vieler Iterationen ein Fraktal ist. Wenn wir eine

Kante der Länge L eines Dreiecks halbieren, so reduziert sich offenbar seine Masse M um den Faktor 3. Nach t Schritten gilt

$$M_t = 3^{-t} M\,, \;\; L_t = 2^{-t} L\,.$$

Daraus folgt

$$D = \lim_{t\to\infty} \frac{\ln M_t}{\ln L_t} = \lim_{t\to\infty} \frac{(-t)\;\ln 3 + \ln M}{(-t)\;\ln 2 + \ln L} = \frac{\ln 3}{\ln 2}\,.$$

Das Sierpinski-Gerüst hat daher die gebrochene Dimension $D = 1.58...$; es füllt zwar keine Fläche, ist aber mehr als eine Linie. Seine Dichte $\rho = M/L^2$ ist null, und jedes noch so kleine Dreieck sieht genauso aus wie ein großes.

Es ist bemerkenswert, daß sich die Natur der Fraktale bedient, um z. B. jede Stelle eines Körpers effektiv mit Blut zu versorgen oder um platzsparende direkte Verbindungen von 10^{11} kommunizierenden Nervenzellen in unserem Gehirn zu erzeugen. Die Technik konnte die Fraktale bisher nur wenig nutzen. Das Buch von M. Schröder allerdings beschreibt den Einsatz fraktaler Reflektoren in der Akustik.

Übung

Eine Koch-Kurve ist durch folgende Iteration definiert: Aus einer Strecke wird das mittlere Drittel herausgetrennt und dafür eine Ausbuchtung mit gleicher Kantenlänge wie im folgenden Bild hinzugefügt. Starten Sie mit einem gleichseitigen Dreieck

und iterieren Sie die obige Konstruktion für jede Kante möglichst oft. Sie erhalten dann die Kochsche Schneeflocke.

Literatur

E. Ott, *Chaos in Dynamical Systems*, Cambridge University Press, 1993.

H. O. Peitgen, D. Saupe, *The Science of Fractal Images*, Springer Verlag, 1988.

M. Schröder, *Fraktale, Chaos und Selbstähnlichkeit*, Spektrum Akademischer Verlag, 1994.

H. G. Schuster, *Deterministic Chaos*, Physik Verlag, 1984.

Stan Wagon, *Mathematica in Aktion*, Spektrum Akademischer Verlag, 1993.

3.4 Neuronales Netzwerk

Können Computer unser Gehirn simulieren? Jeder, der darüber nachdenkt, wird wohl zu dem Schluß kommen, daß diese vermessene Frage für die heutigen Computer mit einem deutlichen Nein beantwortet werden muß. Zwar sind heutzutage eine Menge von Fakten über die Arbeitsweise unseres Gehirns bekannt, trotzdem weiß man noch nicht, wie aus dem Informationsaustausch von 10^{11} Nervenzellen (Neuronen) und ihren 10^{14} Kontaktstellen (Synapsen) solche uns selbstverständlichen Vorgänge wie Lernen, Erkennen und Denken entstehen können.

Wir wissen wenig, aber dennoch versuchen zur Zeit einige tausend Wissenschaftler und Ingenieure, Computerprogramme zu entwickeln, die sowohl von der Architektur als auch von der Funktion des Gehirns lernen. Solche Algorithmen und deren Hardware-Realisierung nennt man *Neuronale Netzwerke* bzw. *Neurocomputer*; in der Tat haben sie Eigenschaften, die sich deutlich von den heutigen Computern unterscheiden.

Ein Neuronales Netz erhält kein Computerprogramm, sondern es lernt anhand von Beispielen, indem es die Stärken seiner Synapsen daran anpaßt. Nach der Lernphase kann es verallgemeinern, d. h. es hat aus den Beispielen Regeln erkannt. Die gelernte Information ist nicht wie im Computer in numerierten Schubladen gespeichert, sondern verteilt im gesamten Netz der Synapsen. Die Logik ist kein strenges „Ja oder Nein“ , sondern ein verschwommenes „Mehr oder Weniger“ .

Viele eindrucksvolle Anwendungen der Neuronalen Netze sind in den letzten Jahren demonstriert worden: englischen Text aussprechen, Ziffern erkennen, Backgammon spielen, Motordefekte an den Laufgeräuschen erkennen, Bankkunden nach ihrer Kreditwürdigkeit beurteilen, Börsenkurse vorhersagen usw. Es hat sich herausgestellt, daß Neuronale Netze oft mit anderen, viel komplizierteren Methoden konkurrieren können. Dabei sind die Lernregeln verblüffend einfach und universell.

Das einfachste Neuronale Netz, das sogenannte *Perzeptron*, wurde schon in den sechziger Jahren angewendet und mathematisch untersucht. Es soll hier vorgestellt und programmiert werden. Unser Perzeptron lernt, die Tastatureingabe 1 und 0 vorherzusagen. Selbst wenn der Leser sich bemüht, die beiden Tasten völlig zufällig zu wählen, wird das Netzwerk im Mittel besser als 50% richtig vorhersagen. Zwar ist diese Fähigkeit noch recht bescheiden, doch immerhin kann ein Rechner mit einem Fünf-Zeilen-Algorithmus eines C-Programmes lernen, menschliches Verhalten vorherzusagen; das sollte uns doch zu denken geben!

Physik

Das Perzeptron besteht aus einer Eingabeschicht von „Neuronen“ S_i, einer weiteren Schicht von N „synaptischen“ Gewichten w_i $(i = 1, ..., N)$ und aus einem Aus-

gabeneuron S_0, das mit allen Eingabeneuronen direkt über Synapsen verkoppelt ist. Wie bei realen Nervenzellen reagiert das Element S_0 auf die Summe der Aktivitäten derjenigen Neuronen, die über die synaptischen Verbindungen direkt auf S_0 einwirken können. Schon 1943 wurde dieser recht komplexe zeitabhängige Vorgang durch eine einfache mathematische Gleichung beschrieben:

$$S_0 = \operatorname{sign}\left(\sum_{j=1}^{N} S_j\, w_j\right) . \tag{3.28}$$

Dabei kann jedes Neuron nur zwei Zustände annehmen: Es ruht $(S_i = -1)$ oder es sendet Impulse $(S_i = 1)$. Die Koeffizienten $w_i \in \mathbb{R}$ modellieren die Stärke, mit der das ankommende Signal des Neurons S_i durch die Synapse in ein elektrisches Potential im Kern von S_0 umgewandelt wird. Die reelle Zahl w_i beschreibt komplexe biochemische Prozesse; eine Synapse kann beispielsweise hemmend $(w_i < 0)$ oder erregend $(w_i > 0)$ sein. Wenn die Summe der Potentiale den Schwellenwert 0 überschreitet, dann feuert das Neuron $(S_0 = +1)$, sonst ruht es $(S_0 = -1)$.

Dies ist die biologische Motivation für die Gleichung (3.28). Mathematisch definiert sie eine Boolesche Funktion $\{+1, -1\}^N \to \{+1, -1\}$, die jede der möglichen Eingaben $\boldsymbol{S} = (S_1, S_2, \ldots, S_N)$ nach $+1$ oder -1 klassifiziert. Es existieren 2^N verschiedene Eingaben $\boldsymbol{S}$. Im allgemeinen kann man jede davon mit $+1$ oder -1 markieren, also gibt es $2^{(2^N)}$ Boolesche Funktionen; für $N = 10$ erhält man die unvorstellbar große Zahl 10^{308}.

Das durch Gleichung (3.28) charakterisierte Perzeptron definiert aber nur spezielle Boolesche Funktionen. Geometrisch gesehen sind $\boldsymbol{S}$ und $\boldsymbol{w} = (w_1, w_2, \ldots, w_N)$ Vektoren im N-dimensionalen Raum. Gleichung (3.28) trennt die Ausgaben $S_0 = 1$ und $S_0 = -1$ durch die $(N-1)$-dimensionale Hyperebene $\boldsymbol{w} \cdot \boldsymbol{S} = 0$. Daher nennt man das Perzeptron auch eine linear separable Boolesche Funktion. Man kann zeigen, daß deren Anzahl kleiner als $2^{(N^2)}$ ist, für $N = 10$ aber immerhin noch 10^{30}.

Was bedeutet nun Lernen und Verallgemeinern für das durch Gleichung (3.28) beschriebene Perzeptron? Ein Neuronales Netzwerk erhält weder Regeln noch Programme, sondern Beispiele. In unserem Fall bestehen die Beispiele aus einer Menge von Eingabe/Ausgabe-Paaren, $(\boldsymbol{x}_\nu, y_\nu)$ mit $\boldsymbol{x}_\nu = (x_{\nu 1}, x_{\nu 2}, \ldots, x_{\nu N})$, $y_\nu \in \{+1, -1\}$ und $\nu = 1, ..., M$, die von einer unbekannten Abbildung geliefert werden. Wie bei realen Nervenzellen lernt unser Netzwerk durch „synaptische Plastizität“ , d. h. es paßt die Stärke seiner Gewichte w_i an die Beispiele an. Im günstigsten Fall kann das Perzeptron nach der Lernphase alle Eingaben richtig klassifizieren:

$$y_\nu = \operatorname{sign}(\boldsymbol{w} \cdot \boldsymbol{x}_\nu) . \tag{3.29}$$

Schon seit den sechziger Jahren sind effektive Lernregeln für das Perzeptron bekannt. Bereits 1949 hat D. Hebb postuliert, daß Synapsen sich an die Aktivität ihrer

beiden Ein- und Ausgangsneuronen anpassen. Mathematisch wurde diese Aussage später so formuliert:

$$\Delta \boldsymbol{w} = \frac{1}{N} y_\nu \, \boldsymbol{x}_\nu \,. \tag{3.30}$$

Bei der Präsentation des ν-ten Beispiels ändert sich jede Synapsenstärke w_i ein wenig, und zwar proportional zum Produkt aus Eingabe- und Ausgabeaktivität. Wenn nur ein Beispiel $(\boldsymbol{x}, y)$ gelernt werden soll, sieht man leicht, daß Synapsen aus Gleichung (3.30) die zu lernende Bedingung (3.29) perfekt erfüllen $(x_j \in \{+1, -1\})$:

$$\boldsymbol{w} \cdot \boldsymbol{x} = \sum_{j=1}^{N} \left(\frac{y}{N} x_j \right) x_j = \frac{y}{N} \sum_{j=1}^{N} x_j \, x_j = y \,. \tag{3.31}$$

Allerdings geht das nicht mehr, wenn N Neuronen $M = \mathcal{O}(N)$ Beispiele lernen sollen; in diesem Fall muß man Gleichung (3.30) ein wenig abändern. Jedes Beispiel soll nur dann mit Gleichung (3.30) gelernt werden, wenn es noch nicht richtig durch das momentane $\boldsymbol{w}$ abgebildet wird:

$$\Delta \boldsymbol{w} = \frac{1}{N} y_\nu \, \boldsymbol{x}_\nu \ \text{ wenn } \ (\boldsymbol{w} \cdot \boldsymbol{x}_\nu) \, y_\nu \leq 0, \ \text{ sonst } \ \Delta \boldsymbol{w} = 0 \,. \tag{3.32}$$

Dies ist die bekannte Perzeptron-Lernregel, für die Rosenblatt 1960 einen mathematischen Konvergenzbeweis gefunden hat: Wenn die M Beispiele $(\boldsymbol{x}_\nu, y_\nu)$ überhaupt durch ein Perzeptron richtig abgebildet werden können, dann stoppt der Algorithmus (3.32). In diesem Fall findet die Lernregel (3.32) also immer einen Gewichtsvektor $\boldsymbol{w}$, einen von vielen, der für alle Beispiele die Gleichung (3.29) erfüllt.

Da dieses Lehrbuch Algorithmen betont, wollen wir den Beweis hier vorstellen. Mit der Definition $\boldsymbol{z}_\nu = y_\nu \, \boldsymbol{x}_\nu$ suchen wir also ein $\boldsymbol{w}$ mit $\boldsymbol{w} \cdot \boldsymbol{z}_\nu > 0$ für $\nu = 1, ..., M$. Nach Voraussetzung soll es ein solches $\boldsymbol{w}_*$ mit $\boldsymbol{w}_* \cdot \boldsymbol{z}_\nu \geq c > 0$ für alle ν geben. Die Konstante $c > 0$ entspricht dem kleinsten Abstand der Punkte $\boldsymbol{z}_\nu$ zu der durch $\boldsymbol{w}_*$ definierten Hyperebene.

Wir starten nun den Algorithmus mit $\boldsymbol{w}(t = 0) = \boldsymbol{0}$. t soll diejenigen Lernschritte zählen, bei denen die Gewichte geändert werden, bei denen also $\boldsymbol{w}(t) \cdot \boldsymbol{z}_\nu \leq 0$ gilt. Bei jedem solchen Schritt t liefert die Lernregel

$$\boldsymbol{w}(t+1) = \boldsymbol{w}(t) + \frac{1}{N} \boldsymbol{z}_\nu \,, \tag{3.33}$$

Wenn wir Gleichung (3.33) vektoriell quadrieren, erhalten wir

$$\begin{aligned} \left(\boldsymbol{w}(t+1)\right)^2 &= \left(\boldsymbol{w}(t)\right)^2 + \frac{2}{N} \boldsymbol{w}(t) \cdot \boldsymbol{z}_\nu + \frac{1}{N^2} \left(\boldsymbol{z}_\nu\right)^2 \\ &\leq \left(\boldsymbol{w}(t)\right)^2 + \frac{1}{N} \,. \end{aligned} \tag{3.34}$$

Beachten Sie, daß $(\boldsymbol{z}_\nu)^2 = \sum_j z_{\nu_j}^2 = N$ ist.

Iteriert man Gleichung (3.34) von 0 bis t, so erhält man mit $\boldsymbol{w}(0) = \mathbf{0}$:

$$(\boldsymbol{w}(t))^2 \leq \frac{t}{N} . \tag{3.35}$$

Eine Abschätzung für das Skalarprodukt $\boldsymbol{w}_* \cdot \boldsymbol{w}(t+1)$ ergibt

$$\boldsymbol{w}_* \cdot \boldsymbol{w}(t+1) = \boldsymbol{w}_* \cdot \boldsymbol{w}(t) + \frac{1}{N} \boldsymbol{w}_* \cdot \boldsymbol{z}_\nu \geq \boldsymbol{w}_* \cdot \boldsymbol{w}(t) + \frac{c}{N} , \tag{3.36}$$

die Iteration von 0 bis t also

$$\boldsymbol{w}_* \cdot \boldsymbol{w}(t) \geq \frac{c\,t}{N} . \tag{3.37}$$

Nun setzen wir (3.35) und (3.37) in die Schwarzsche Ungleichung

$$(\boldsymbol{w}(t))^2 (\boldsymbol{w}_*)^2 \geq (\boldsymbol{w}(t) \cdot \boldsymbol{w}_*)^2 \tag{3.38}$$

ein und erhalten

$$\frac{t}{N} (\boldsymbol{w}_*)^2 \geq \frac{c^2 t^2}{N^2} , \tag{3.39}$$

oder

$$t \leq N \frac{(\boldsymbol{w}_*)^2}{c^2} = t_* . \tag{3.40}$$

Damit ist die Zahl t der durchgeführten Lernschritte beschränkt, der Algorithmus stoppt nach höchstens t_* Schritten. Er endet aber nur, wenn alle Beispiele richtig abgebildet werden. Die Perzeptron-Regel kann also alle lernbaren Probleme auch wirklich perfekt lernen.

Wenn sehr viele Beispiele gelernt werden sollen, wird der Abstand c zur Hyperebene klein und damit die Zahl t_* der Lernschritte sehr groß. Das bedeutet, daß man eventuell jedes Beispiel sehr oft lernen muß, bis schließlich alle richtig klassifiziert werden.

Wieviele Beispiele kann ein Perzeptron überhaupt lernen? Das hängt offenbar von der unbekannten Abbildung ab, die die Beispiele erzeugt. Wenn diese von einem anderen Perzeptron stammen, dann kann nach dem obigen Konvergenz-Theorem jede Anzahl M gelernt werden. Wenn dagegen die Klassifikationsbits y_ν zufällig gewählt sind, dann gibt es wieder exakte Aussagen: Für $M < N$ können alle Beispiele perfekt gelernt werden, im Limes $N \to \infty$ sogar (mit Wahrscheinlichkeit 1) für $M < 2N$.

Nach der Lernphase kann das Perzeptron verallgemeinern. Das bedeutet, daß es eine Eingabe $\boldsymbol{S}$, die es vorher nicht gelernt hat, genauso klassifiziert wie die unbekannte Abbildung, von der es gelernt hat. Wenn die „Lehrer“-Funktion für eine Eingabe $\boldsymbol{S}$ das Resultat S_0 liefert, dann ist ein Maß g für die Verallgemeinerungsfähigkeit folgendermaßen definiert

$$g = \langle \Theta \left(S_0 \, \boldsymbol{w} \cdot \boldsymbol{S} \right) \rangle_{\boldsymbol{S}} \ , \tag{3.41}$$

wobei über viele Eingaben $\boldsymbol{S}$ gemittelt wird. $g = 1/2$ ist das Ergebnis von zufälligem Raten, das Perzeptron kann in diesem Fall nicht verallgemeinern. Bei $g > 1/2$ dagegen hat das Perzeptron in den gelernten Beispielen eine gewisse Regelmäßigkeit erkannt und stimmt mit der Wahrscheinlichkeit g mit dem „Lehrer“ überein.

Wir wollen nun das Perzeptron benutzen, um Zeitreihen zu analysieren und Vorhersagen für den nächsten Zeitschritt zu machen. Es sei eine Folge von Bits gegeben, z. B.

$$\boldsymbol{F} = (1, -1, -1, 1, 1, -1, -1, -1, 1, 1, 1, -1, 1, 1, -1, ...) \ . \tag{3.42}$$

Das Perzeptron tastet jeweils ein Fenster von N Bits ab

$$\boldsymbol{x}_\nu = (F_\nu, F_{\nu+1}, ..., F_{\nu+N-1}) \tag{3.43}$$

und macht eine Vorhersage für das Bit $F_{\nu+N}$:

$$\tilde{F}_{\nu+N} = \operatorname{sign}\left(\boldsymbol{w} \cdot \boldsymbol{x}_\nu\right) \ . \tag{3.44}$$

Dann lernt das Perzeptron das Beispiel $(\boldsymbol{x}_\nu, F_{\nu+N})$ mit der Rosenblatt-Regel (3.32) bzw. (3.33), schiebt das Fenster um ein Bit nach rechts und macht eine neue Vorhersage für die Eingabe $\boldsymbol{x}_{\nu+1}$. Dies wird iteriert und eine Trefferrate g bestimmt.

Wenn die Folge $\boldsymbol{F}$ eine Periode M hat, dann gibt es offenbar M Beispiele und für $M < N$ kann das Perzeptron diesen „Rhythmus“ perfekt lernen. Eine Zufallsfolge dagegen wird immer eine Trefferrate von $g = 50\%$ liefern. Wir wollen selbst eine Bitfolge eingeben und testen, ob und wie gut ein einfaches Perzeptron deutliche bzw. unbewußte Regelmäßigkeiten daran erkennen kann.

Algorithmus

Dazu programmieren wir das Perzeptron in der Sprache C. Die Neuronenaktivität wird im Feld `neuron[N]` und die synaptischen Gewichte in `weight[N]` gespeichert. Die Variable `input` erhält das nächste Bit. Es wird immer nur das letzte Beispiel gelernt. Der Lern- und Vorhersage-Algorithmus (3.32) und (3.44) lautet dann einfach:

```
while (1) {
```

1. Einlesen:
```
if (getch() == '1') input=1; else input = -1; runs++;
```
2. Potential berechnen:
```
for (h=0., i=0; i<N; i++) h += weight[i] * neuron[i];
```
3. Treffer der Vorhersage zählen:
```
if(h*input >0) correct++;
```
4. Lernen:
```
if(h*input < 0)
for (i=0; i<N; i++)
weight[i] += input*neuron[i]/(float)N;
```
5. Fenster verschieben:
```
for (i=N-1; i>0; i--) neuron[i] = neuron[i-1];
neuron [0]= input;
} /* end of while */
```

Der wesentliche Teil des Programmes `nn.c` besteht also wirklich nur aus fünf Zeilen. Alles übrige ist Initialisierung und Bildschirmgraphik. Die Trefferrate in der Form `correct/ runs*100.0` wird jedes Mal ausgedruckt, ebenfalls die Stärke der Gewichte und das Ergebnis der Vorhersage. Nach der Taste `n` wird neu gezählt, und mit `e` verläßt man das Programm.

Die Version `nnf name` übernimmt die Eingaben `1` und `0` aus der Datei `name`. Wir wollen hier kurz zeigen, wie man Argumente aus der Kommandozeile übernehmen und eine Datei lesen kann. Die Funktion `main` enthält nun Argumente

```
main(int argc, char *argv[])
```

`argc` enthält die Anzahl der Eingabeworte, hier `argc=2`. Die beiden Feldkomponenten `argv[0]` und `argv[1]` enthalten die Adressen des Programmnamens `(nnf)` und des Arguments `(name)`. Wird kein Name eingegeben, so soll er abgefragt und eingelesen werden. Mit diesem Namen wird dann die entsprechende Datei zum Lesen geöffnet.

```
FILE *fp;
char str[100];
if(argc==1)   { printf("Welche Eingabedatei?");
                scanf("%s", str);
              }
   else       strcpy(str, argv[1]);
if((fp=fopen(str,"r"))==NULL)
              { printf("Datei nicht vorhanden");
                exit(1); }
```

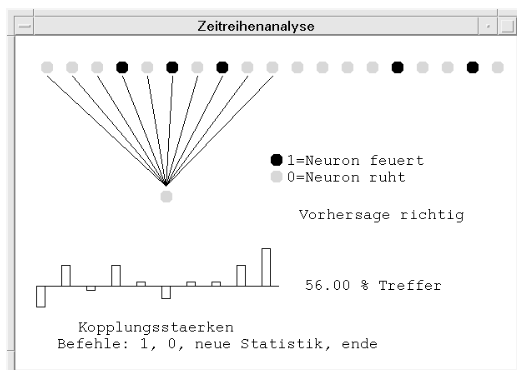
Das Einlesen der Zeichen `1` und `0` lautet dann folgendermaßen:

```
while(feof(fp)==NULL)
    {   switch(fgetc(fp))
        {   case'1': input=1; break;
            case'0': input=-1; break;
            default: continue;
        }
       ...
```

`feof` signalisiert das Dateiende und `fgetc` liest das nächste Zeichen aus der Datei, die durch `fopen` den Zeiger `fp` erhalten hat. `break` springt aus dem `switch`-Befehl heraus, und `continue` überspringt den Rest der `while`-Schleife, die wie vorher die Schritte zwei bis fünf enthält. Das Programm `nnf` nimmt nur die Eingaben `1` und `0` als Bitsequenz, während `nn` die Eingabe `1` oder „verschieden von `1`" in `+1` und `-1` umwandelt.

Ergebnis

Der Aufruf `nn` stellt das Perzeptron auf dem Bildschirm graphisch dar (Bild 3.17). Die Bitfolge aus der Tastatureingabe `1` oder `0`, dargestell als roter bzw. grüner Punkt,



3.17 Neuronales Netz zur Vorhersage einer Zeitreihe.

ist zwar 20 Schritte weit zu sehen, das Perzeptron greift aber nur die letzten 10 Bits ab und macht eine Vorhersage für die folgende Eingabe. Diese wird natürlich nicht angezeigt, sondern das Ergebnis der Übereinstimmung wird erst nach der Eingabe ausgedruckt. Den Lernvorgang sieht man an der Änderung der synaptischen Gewichte, die als violette Balken gezeichnet werden. Die Trefferrate g ist das wesentliche Ergebnis; $g > 50\%$ bedeutet, daß das Perzeptron eine Struktur in der Bitfolge erkannt hat.

Zunächst tippen wir einen Rhythmus ein, z. B.

$$1010101010...$$

Nach wenigen Schritten hat das Netzwerk diese zwei Muster gelernt und liefert nur noch richtige Vorhersagen. Wechselt man dann die Folge zu einer anderen Periode, z. B.

$$111001110011100\ldots,$$

so hat das Perzeptron ebenfalls nach wenigen Schritten diese Folge perfekt gelernt. Eine Rückkehr zum vorherigen Rhythmus gelingt natürlich wieder perfekt. Da die Stärke $\boldsymbol{w}$ der Synapsen aber durch den gesamten Lernvorgang beeinflußt wird, hat $\boldsymbol{w}$ andere Koeffizienten als vorher.

Nun versuchen wir, 1 und 0 zufällig einzutippen. Wir haben dies mit Studenten unserer Vorlesung getestet, die jeweils etwa 1000 mal die beiden Zeichen in möglichst zufälliger Folge in eine Datei geschrieben haben. Die Auswertung mit `nnf` ergab eine mittlere Trefferrate von $g = 60\%$, mit Werten von $50\% \leq g \leq 80\%$. Offenbar ist es recht schwierig, Zufallsfolgen per Hand zu erzeugen. Oft gibt schon die Vorhersage $1 \to 0$ bzw. $0 \to 1$ eine Trefferrate $g > 50\%$. Das Perzeptron erkennt recht schnell solche Rhythmen.

Fünf Zeilen eines Computerprogrammes sind also in der Lage, unsere Reaktionen mit einer gewissen Wahrscheinlichkeit g vorherzusagen. Umgekehrt sollten wir allerdings auch in der Lage sein, das Netzwerk zu überlisten, denn das Perzeptron ist ein einfaches deterministisches System, das wir genau berechnen können. Wir können daher im Prinzip ausrechnen, welche Vorhersage das Netzwerk machen wird und dann genau das Gegenteil eintippen. So sollte es uns gelingen, die Trefferrate g weit unter 50% zu drücken.

In der Tat lieferte ein Student eine Bitfolge ab, die $g = 0$ ergab. Später räumte er allerdings ein, das Programm bereits zu kennen, so daß er sich das Gegenbit vom Netzwerk selbst erzeugen lassen konnte. Eigene Versuche, auf die Vorhersagen des Modells entsprechend zu reagieren, konnten g auf etwa 40% drücken. Wir laden unsere Leser herzlich ein, beides auszuprobieren: die Erzeugung von Zufallsbits und die Überlistung des Perzeptrons.

Übung

Schreiben Sie per Hand in eine Datei `name.dat` eine Folge von etwa 2000 Zeichen, und zwar 1 oder 0, die möglichst zufällig eingetippt werden sollen. Fremdzeichen werden vom Programm ignoriert. Lassen Sie diese Daten mit `nnf name.dat` auswerten. Sind Sie ein guter Zufallszahlengenerator? Überprüfen Sie, ob Zufalls-

bits, die vom Computer erzeugt werden, eine Trefferrate von etwa 50% liefern. Wie groß darf die Abweichung von 50% sein?

Literatur

J. Hertz, A. Krogh, R. G. Palmer, *Introduction to the Theory of Neural Computation*, Addison-Wesley, 1991.

B. Müller, J. Reinhardt, M. T. Strickland, *Neural Networks: An Introduction*, Springer Verlag, 1995.

Kapitel 4
Differentialgleichungen

Seit mehr als 300 Jahren beschreiben Physiker die Bewegung von Massen durch Gesetzmäßigkeiten im unendlich Kleinen: Massenelemente bewegen sich durch Kräfte in einem winzigen Zeitschritt eine winzige Strecke vorwärts. Eine solche Beschreibung führt auf Differentialgleichungen, die auch heute noch das wichtigste Werkzeug in der Physik sind. Wenn doppelt so starke Ursachen eine doppelt so große Wirkung haben, kann man die Gleichungen meist lösen. Bei nichtlinearen Differentialgleichungen dagegen ist man oft auf einige wenige lösbare Spezialfälle oder auf numerische Lösungen angewiesen. In diesem Kapitel wollen wir eine Einführung in einige der numerischen Methoden geben und dazu einfache physikalische Beispiele lösen.

Es gibt viele fertige Programmpakete zur numerischen Lösung von Differentialgleichungen. So kann man beispielsweise in *Mathematica* mit `NDSolve` gewöhnliche Differentialgleichungen lösen, ohne die Einzelheiten des Programms zu kennen. Trotzdem wollen wir hier nicht ganz auf die Beschreibung der Algorithmen verzichten, denn erstens muß man bei großen und schwierigen Problemen immer selbst einen Kompromiß zwischen Rechenzeit und Rechengenauigkeit suchen, und zweitens gibt es für viele Spezialfälle besondere Methoden, die auch heute noch erforscht werden.

4.1 Runge-Kutta-Methode

Grundlagen

Zunächst wollen wir die Lösung einer gewöhnlichen Differentialgleichung diskutieren. Dazu betrachten wir eine Funktion $y(x)$, deren n-te Ableitung $y^{(n)}(x)$ sich als Funktion von x und allen vorherigen Ableitungen schreiben läßt,

$$y^{(n)} = f(x, y, y', y'', \dots, y^{(n-1)}) \,, \tag{4.1}$$

also eine explizite Differentialgleichung n-ter Ordnung. Diese Gleichung ist äquivalent zu n Gleichungen erster Ordnung, denn mit $y_1 = y,\ y_2 = y', \dots, y_n = y^{(n-1)}$

erhält man das Gleichungssystem

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= y_3 \\ &\vdots \\ y_{n-1}' &= y_n \\ y_n' &= f(x, y_1, ..., y_n) \, , \end{aligned} \tag{4.2}$$

oder in Vektorform mit $\boldsymbol{y} = (y_1, y_2, ..., y_n)$,

$$\boldsymbol{y}'(x) = \lim_{\Delta x \to 0} \frac{\boldsymbol{y}(x + \Delta x) - \boldsymbol{y}(x)}{\Delta x} = \boldsymbol{f}(x, \boldsymbol{y}) \, . \tag{4.3}$$

Kennt man den Vektor $\boldsymbol{y}$ am Ort x, so kann man seinen Wert nach einem winzigen Schritt $x + \Delta x$ mit Gleichung (4.3) berechnen. Dies ist auch die numerische Methode: Die Ortskoordinate x wird diskretisiert und mit Gleichung (4.3) versucht man, $\boldsymbol{y}$ am Nachbarort $x + \Delta x$ möglichst genau zu berechnen.

Zur Beschreibung der numerischen Methode beschränken wir uns auf die Dimension $n = 1$, d. h. $\boldsymbol{y}$ und $\boldsymbol{f}$ haben nur je eine Komponente y und f. Der vorzustellende Algorithmus läßt sich später leicht auf beliebige Dimensionen erweitern. Sei also

$$y' = f(x, y) \tag{4.4}$$

gegeben. Für den Start der Rechnung benötigen wir offenbar einen Anfangswert $y(x_0) = y_0$. Zunächst diskretisieren wir die x-Achse:

$$x_n = x_0 + n\, h \, , \tag{4.5}$$

wobei n eine ganze Zahl und h die Schrittweite ist. Seien $y_n = y(x_n)$ und $y_n' = y'(x_n)$. Dann gibt die Taylor-Entwicklung

$$y_{n\pm1} = y(x_n \pm h) = y_n \pm h\, y_n' + \frac{h^2}{2} y_n'' \pm \frac{h^3}{6} y_n''' + \mathcal{O}(h^4) \, . \tag{4.6}$$

y_n' erhält man aus Gleichung (4.4). In der einfachsten Näherung, dem sogenannten *Euler-Verfahren*, erhält man damit:

$$y_{n+1} = y_n + h\, f(x_n, y_n) \, . \tag{4.7}$$

Der Fehler ist von der Größenordnung h^2. Diese Methode wird nicht für den praktischen Einsatz empfohlen. Mit nur geringem Aufwand kann sie wesentlich verbessert werden.

Gleichung (4.7) berücksichtigt nur die Steigung f am Punkt (x_n, y_n), obwohl sie sich auf dem Weg zum nächsten Punkt (x_{n+1}, y_{n+1}) schon wieder geändert hat.

Die Taylorreihe (4.6) zeigt, daß es günstiger ist, die Steigung auf dem halben Weg zu diesem Punkt zu benutzen, denn dann gewinnt man eine Größenordnung in der Schrittweite h. Doch dazu muß man zunächst eine geeignete Schätzung für $y_{n+\frac{1}{2}} = y\left(x_n + \frac{h}{2}\right)$ finden. Dies ist die Idee des Runge-Kutta-Verfahrens, das schon im Jahr 1895 entwickelt wurde. Nimmt man die Entwicklung (4.6) an der Stelle $x_n + \frac{h}{2}$ mit der Schrittweite $\frac{h}{2}$, so gilt:

$$\begin{aligned} y_n &= y_{n+\frac{1}{2}} - \frac{h}{2}\, y'_{n+\frac{1}{2}} + \left(\frac{h}{2}\right)^2 \frac{1}{2}\, y''_{n+\frac{1}{2}} + \mathcal{O}(h^3)\,, \\ y_{n+1} &= y_{n+\frac{1}{2}} + \frac{h}{2}\, y'_{n+\frac{1}{2}} + \left(\frac{h}{2}\right)^2 \frac{1}{2}\, y''_{n+\frac{1}{2}} + \mathcal{O}(h^3)\,. \end{aligned} \tag{4.8}$$

Daraus folgt:

$$y_{n+1} = y_n + h\, y'_{n+\frac{1}{2}} + \mathcal{O}(h^3)\,. \tag{4.9}$$

$y'_{n+\frac{1}{2}} = f\left(x_{n+\frac{1}{2}}, y_{n+\frac{1}{2}}\right)$ approximieren wir durch $f\left(x_{n+\frac{1}{2}}, y_n + \frac{h}{2}\, f(x_n, y_n)\right) + \mathcal{O}(h^2)$, so daß wir zu folgendem Algorithmus gelangen:

$$\begin{aligned} k_1 &= h\, f(x_n, y_n)\,, \\ k_2 &= h\, f\left(x_n + \frac{h}{2}, y_n + \frac{k_1}{2}\right), \\ y_{n+1} &= y_n + k_2\,. \end{aligned} \tag{4.10}$$

Dieses Runge-Kutta-Verfahren zweiter Ordnung macht also einen Fehler der Größenordnung h^3 bei nur zwei Berechnungen der Funktion f. Wenn man diesen Rechenschritt mehrmals iteriert, kann man noch höhere Ordnungen von h auslöschen. Das in der Praxis bewährte Verfahren benötigt vier Rechnungen bei einem Fehler von der Ordnung h^5. Wir geben nur das Ergebnis an:

$$\begin{aligned} k_1 &= h\, f\,(x_n, y_n)\ , \\ k_2 &= h\, f\left(x_n + \frac{h}{2}, y_n + \frac{k_1}{2}\right), \\ k_3 &= h\, f\left(x_n + \frac{h}{2}, y_n + \frac{k_2}{2}\right), \\ k_4 &= h\, f\,(x_n + h, y_n + k_3)\,, \\ y_{n+1} &= y_n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6}\,. \end{aligned} \tag{4.11}$$

Es bleibt das Problem, die Größe der Schrittweite h zu wählen. Eine Möglichkeit, einen geeigneten Wert für h zu finden, besteht darin, das Ergebnis für verschiedene

h-Werte zu prüfen. Bei schwacher Änderung von y kann man sich große Schrittweiten leisten, während man im umgekehrten Fall viele kleine Rechenschritte braucht. Wenn also die Steigung von y stark variiert, sollte das Programm selbsttätig die Schrittweite h reduzieren.

Um den Fehler abzuschätzen, berechnet man parallel zu je zwei Runge-Kutta-Schritten (4.11) mit der Schrittweite h (Ergebnis $y_{2\times 1}$) einmal einen Runge-Kutta-Schritt mit der Schrittweite $2h$ (Ergebnis $y_{1\times 2}$). Dies kostet weniger als 50 % Mehraufwand, da die Ableitung am Startpunkt ohnehin schon berechnet ist. Bezeichnen wir die exakte Lösung für einen Schritt von x nach $x+2h$ mit $y(x+2h)$, so beträgt die Abweichung der Näherungswerte $y_{2\times 1}$, bzw. $y_{1\times 2}$ von $y(x+2h)$, da wir ja eine Methode vierter Ordnung verwenden:

$$\begin{aligned} y(x+2h) &= y_{2\times 1} + 2\,h^5\,\phi + \mathcal{O}(h^6)\,, \\ y(x+2h) &= y_{1\times 2} + (2h)^5\,\phi + \mathcal{O}(h^6)\,. \end{aligned} \tag{4.12}$$

Die Konstante ϕ ist durch die fünfte Ableitung der Funktion $y(x)$ bestimmt. Für den Unterschied $\Delta = y_{1\times 2} - y_{2\times 1}$ folgt hieraus:

$$\Delta \propto\; h^5\,. \tag{4.13}$$

Wenn wir also zwei Schrittweiten h_1 und h_0 wählen, so gilt:

$$h_0 = h_1 \left|\frac{\Delta_0}{\Delta_1}\right|^{0.2}. \tag{4.14}$$

Aus dem berechneten Fehler Δ_1 zur Schrittweite h_1 kann damit die Schrittweite h_0 für die gewünschte Genauigkeit Δ_0 berechnet werden. Bezieht man die gewünschte Genauigkeit auf den globalen statt wie in (4.12) bis (4.14) auf den lokalen Fehler, so muß in Gleichung (4.14) der Exponent $1/5$ durch $1/4$ ersetzt werden. In der Praxis hat sich ein gemischtes Verfahren als verläßlich erwiesen, das bei einer möglichen Vergrößerung von h vorsichtshalber den Exponenten $1/5$ und bei einer nötigen Verkleinerung den Exponenten $1/4$ benutzt.

Oft gibt es in der Physik Bewegungsgleichungen der Form

$$\begin{aligned} \frac{dx}{dt} &= v(t)\,, \\ \frac{dv}{dt} &= f(x(t),t)\,. \end{aligned} \tag{4.15}$$

$v(t)$ ist die Geschwindigkeit eines Teilchens mit der Bahn $x(t)$. In diesem Fall gibt es einen Trick, um die Steigung zwischen zwei Punkten abzuschätzen: Man berechnet x zu den Zeiten $t_n = t_0 + n\,h$ mit $n = 1, 2, 3...$ und v zu den Zwischenzeiten

$t_{n+\frac{1}{2}}$ durch

$$\begin{aligned} v_{n+\frac{1}{2}} &= v_{n-\frac{1}{2}} + h\, f(x_n, t_n)\,, \\ x_{n+1} &= x_n + h\, v_{n+\frac{1}{2}}\,. \end{aligned} \quad (4.16)$$

Der Algorithmus springt also immer hin und her zwischen der Berechnung des Ortes x und der Geschwindigkeit v an den Zwischenplätzen. Deshalb hat er den Namen *leapfrog (Bocksprung)*. Da die Ableitungen immer in der Mitte zwischen jeweils zwei aufeinanderfolgenden Werten berechnet werden, ist der Fehler von der Ordnung h^3. Obwohl die Leapfrog-Methode denselben Rechenaufwand wie der Euler-Schritt erfordert, ist sie genauer und in der Praxis viel robuster. Sie wird daher gern bei der *Molekulardynamik* verwendet, also bei der Lösung der Bewegungsgleichungen von vielen miteinander wechselwirkenden Teilchen.

Abschließend soll noch darauf hingewiesen werden, daß das Runge-Kutta-Verfahren zwar einfach und robust ist, daß es aber je nach Problemstellung noch bessere Methoden mit geringerer Rechenzeit geben kann. Beispielsweise kann man das Ergebnis von mehreren einfachen Schritten für verschiedene h-Werte berechnen und für $h \to 0$ geeignet extrapolieren; dies ist die Idee der Richardson Extrapolation und der Bulirsch-Stoer-Methode. Eine andere Möglichkeit ist, verschiedene y_i-Werte mit einem Polynom zu extrapolieren und damit eine Vorhersage für einen y_n-Wert zu erhalten, mit dem die Steigung y'_n berechnet wird. Durch Integration der y'_i-Werte korrigiert man dann den Wert y_n. Auch für solche sogenannten Predictor-Corrector-Methoden gibt es in den Lehrbüchern geeignete Algorithmen.

Algorithmus

Gleichung (4.11) kann man einfach programmieren, selbst wenn y und f wieder durch Vektoren $\boldsymbol{y}$ und $\boldsymbol{f}$ ersetzt werden. In *Mathematica* sind dann $\boldsymbol{y}$ und $\boldsymbol{f}$ Listen von Zahlen bzw. Funktionen. Falls $\boldsymbol{f}$ nur von $\boldsymbol{y}$ und nicht von x abhängt, kann Gleichung (4.11) geschrieben werden als

```
RKStep[f_, y_, yp_, h_]:=
     Module[{k1, k2, k3, k4},
             k1 = h N[f/.Thread[y -> yp]];
             k2 = h N[f/.Thread[y -> yp+k1/2]];
             k3 = h N[f/.Thread[y -> yp+k2/2]];
             k4 = h N[f/.Thread[y -> yp+k3]];
             yp + (k1 + 2*k2 + 2*k3 + k4)/6 ]
```

Die Funktion `Thread` weist jedem Element der Liste `y` den entsprechenden Wert der Liste `yp` zu. Man beachte, daß `k1`, `k2`, `k3` und `k4` wieder Listen sind. Die $\boldsymbol{y}_i$-Werte im Intervall $[0, x]$ erhält man dann aus dem Anfangswert $\boldsymbol{y}_0$ durch

```
RungeKutta[f_List,y_List,y0_List,{x_, dx_}]:=
  NestList[RKStep[f,y,#,N[dx]]&,N[y0],Round[N[x/dx]]]
```

Zum Vergleich programmieren wir auch das Euler-Verfahren:

```
EulerStep[f_,y_,yp_,h_]:= yp+h N[f/.Thread[y -> yp]]
Euler[f_,y_,y0_,{x_, dx_}]:=
NestList[EulerStep[f,y,#,N[dx]]&,N[y0],Round[N[x/dx]]]
```

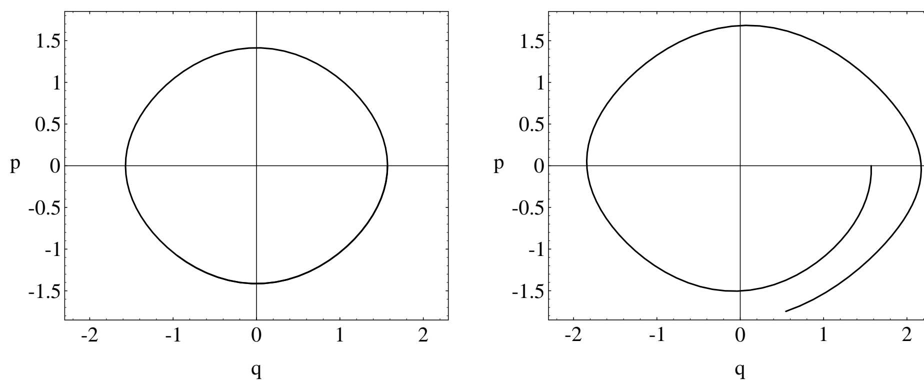
Ergebnisse

Wir wollen die Differentialgleichung für ein Problem lösen, das wir schon im Abschnitt 1.2 ausführlich untersucht haben: das mathematische Pendel. Die Hamiltonfunktion dazu lautet

```
hamilton = p^2/2 - Cos[q]
```

Dabei ist p die Winkelgeschwindigkeit $\dot{\varphi}$ und q der Ausschlagswinkel φ des Pendels. Energie und Zeit werden wie vorher in Einheiten von mgl und $\sqrt{l/g}$ gemessen. Die Bewegungsgleichungen erhält man aus den partiellen Ableitungen der Hamiltonfunktion. In *Mathematica* lautet damit deren Lösung

```
RungeKutta[{D[hamilton,p], -D[hamilton,q]},
                    {q,p}, {phi0,p0}, {tmax,dt}]
```



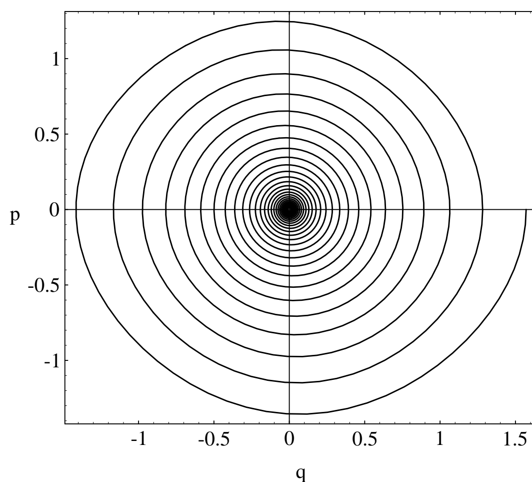
4.1 Schwingung des Pendels im Phasenraum $q = \varphi, p = \dot{\varphi}$. Die Kurve links zeigt das Ergebnis des Runge-Kutta-Verfahrens, während die Kurve der rechten Graphik mit der Euler-Methode berechnet wurde und offenbar ein falsches Resultat liefert.

Abbildung 4.1 zeigt das Ergebnis. Die Kurve der linken Graphik ist das richtige Ergebnis für den Anfangswinkel $\varphi_0 = \frac{\pi}{2}$. Ohne Reibung ist die Energie erhalten und die Kurve muß sich wieder schließen (siehe auch Abbildung 1.4). Zum Vergleich enthält die Abbildung rechts auch das Ergebnis des Euler-Verfahrens mit derselben Schrittweite $h = 0.1$. Man sieht, daß diese Methode ein völlig falsches Ergebnis liefert, da die Energie wächst und sich die Kurve nicht schließt.

Natürlich können wir auch sehr einfach eine Reibungskraft hinzufügen. Wir addieren zur Gravitationskraft einen Term der Form $-r\,p$ und erhalten die Lösung der Bewegungsgleichung mit

```
RungeKutta[{p,-Sin[q]-r p},{q,p},{phi0,p0},{tmax,dt}]
```

In Abbildung 4.2 sieht man, daß das Pendel seine Energie verliert und in die Ruhelage $q = p = 0$ relaxiert.



4.2 Wie Abbildung 4.1 links, aber mit Reibung $r = 0.05$.

Übung

Ein Teilchen der Masse m soll sich in einem eindimensionalen Doppelmuldenpotential bewegen und außerdem eine Reibungskraft spüren, die proportional zur Geschwindigkeit ist. Die Bewegungsgleichung für den Ort $x(t)$ als Funktion der Zeit t lautet also

$$m\,\ddot{x} = -r\,\dot{x} + a\,x - b\,x^3$$

mit positiven Konstanten r, a und b. Je nach Startzustand endet das Teilchen entweder in der linken ($x < 0$) oder in der rechten Potentialmulde.

Für die Konstanten $r/m = 0.1$, $a/m = b/m = 1$ (in geeigneten Einheiten) sollen Sie in der Ebene der Startzustände $(x(0), \dot{x}(0))$ diejenigen Gebiete berechnen und zeichnen, deren Punkte in der linken Mulde, also bei $x(\infty) = -1$ landen.

Literatur

R. E. Crandall, *Mathematica for the Sciences*, Addison Wesley, 1991.

S. E. Koonin, D. C. Meredith, *Physik auf dem Computer, Band 1+2*, R. Oldenbourg Verlag, 1990.

H. J. Korsch, H.-J. Jodl, *Chaos: A Program Collection for the PC*, Springer Verlag, 1994.

W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge University Press, 1992.

E. W. Schmid, G. Spitz, W. Lösch, *Theoretische Physik mit dem Personal Computer*, Springer Verlag, 1987.

J. Schnakenberg, *Algorithmen in der Quantentheorie und Statistischen Physik*, Zimmermann-Neufang, 1995.

J. Stoer, R. Bulirsch, *Numerische Mathematik 2*, Springer Verlag, 1990.

Paul L. DeVries, *Computerphysik: Grundlagen, Methoden, Übungen*, Spektrum Akademischer Verlag, 1995.

4.2 Chaotisches Pendel

Schon am Ende letzten Jahrhunderts hat der französische Mathematiker Henri Poincaré gezeigt, daß ein einfaches mechanisches System eine komplexe Bewegung ausführen kann. Die Vorstellung, daß man nur die Anfangsbedingungen möglichst genau angeben muß und daß man dann die Bewegung eines mechanischen Systems im Prinzip durch seine Newtonschen Bewegungsgleichungen genau vorausberechnen kann, wird in der Praxis schon bei einfachen Modellen ad absurdum geführt. Im allgemeinen reagiert ein System so empfindlich auf die Anfangsbedingungen, daß eine winzige Ungenauigkeit der Startwerte schon nach kurzer Zeit zu einer großen Ungenauigkeit der Vorhersage führt. Das gilt nicht nur für die Wettervorhersage, sondern auch schon für ein einfaches Pendel mit äußerem Antrieb.

Obwohl diese Tatsache schon seit langem bekannt ist, hat es fast 100 Jahre gedauert, bis ihre Bedeutung in der Wissenschaft erkannt und erforscht wurde. Erst mit dem Computer konnte chaotische Bewegung im Detail untersucht und – was vielleicht ebenso wichtig ist – graphisch dargestellt werden.

In diesem Abschnitt wollen wir ein einfaches Beispiel numerisch berechnen: das nichtlineare Pendel mit Reibung, das von einer äußeren periodischen Kraft angetrieben wird.

Physik

Wir betrachten zunächst das Pendel aus Abschitt 1.2 mit einer zusätzlichen Reibungskraft. Die Bewegungsgleichung für den Winkel $\varphi(t)$ der Auslenkung lautet in dimensionsloser Form:

$$\ddot{\varphi} + r\,\dot{\varphi} + \sin\varphi = 0 \tag{4.17}$$

mit einem Reibungsparameter $r \geq 0$. Diese Differentialgleichung zweiter Ordnung kann man durch die Einführung von $\omega(t) = \dot{\varphi}(t)$ in zwei Gleichungen erster Ordnung umschreiben:

$$\begin{aligned} \dot{\varphi} &= \omega\,, \\ \dot{\omega} &= -r\,\omega - \sin\varphi\,. \end{aligned} \tag{4.18}$$

Die Bewegung des Pendels läßt sich daher in einem zweidimensionalen Phasenraum darstellen. Jeder Wert (φ, ω) bestimmt eindeutig seine Änderung $(d\varphi, d\omega)$ im Zeitintervall dt. Daher kann sich eine Trajektorie $(\varphi(t), \omega(t))$ nicht selbst kreuzen.

Man kann zeigen, daß es aus diesem Grund keine chaotische Bewegung geben kann. In zwei Dimensionen gibt es einfach keinen Platz für Trajektorien, die für lange Zeiten etwas anderes tun, als geschlossene Bahnen zu bilden oder sich solchen anzunähern. In unserem Beispiel relaxiert für $r > 0$ das Pendel, eventuell nach einigen Schwingungen und Überschlägen, zur Ruhelage $(\varphi = 0, \omega = 0)$. Der Ursprung des Phasenraumes ist ein Attraktor, der fast alle Anfangspunkte $(\varphi(0), \omega(0))$ anzieht. Jede Trajektorie läuft schließlich spiralförmig zur Ruhelage (siehe Abbildung 4.2).

Das ändert sich, wenn wir eine dritte Richtung im Phasenraum erlauben. Dazu treiben wir das Pendel mit einem periodischen Drehmoment der Stärke a und der Frequenz ω_{D} an,

$$\ddot{\varphi} + r\,\dot{\varphi} + \sin\varphi = a\,\cos(\omega_{\mathrm{D}}\,t)\,. \tag{4.19}$$

Mit $\theta = \omega_{\mathrm{D}}\,t$ erhält man

$$\begin{aligned} \dot{\varphi} &= \omega\,, \\ \dot{\omega} &= -r\,\omega - \sin\varphi + a\,\cos\theta\,, \\ \dot{\theta} &= \omega_{\mathrm{D}}\,. \end{aligned} \tag{4.20}$$

Die Bewegung des Pendels wird nun im dreidimensionalen Raum $(\varphi, \omega, \theta)$ beschrieben. Es gibt drei Parameter $(r, a, \omega_{\mathrm{D}})$. Ohne Antrieb, ohne Reibung $(a =$

$r = 0$) und für kleine Winkel $\varphi \ll 1$ erhalten wir eine harmonische Schwingung mit der Frequenz $\omega_0 = 1$. Es gibt deshalb im allgemeinen Fall drei Zeitskalen, die miteinander konkurrieren: die Antriebsperiode $2\pi/\omega_D$, die Eigenperiode $2\pi/\omega_0$ und die Relaxationszeit $1/r$. Wie schon beim Hofstadter-Schmetterling und beim Frenkel-Kontorova-Modell führt der Wettbewerb zwischen verschiedenen Längen oder Zeiten auch hier zu interessanten physikalischen Phänomenen.

Die dreidimensionale Bewegung $(\varphi(t), \omega(t), \theta(t))$ ist nur schwer zu analysieren, auch wenn wie in unserem Fall θ nur linear mit der Zeit t wächst. Um die Vielfalt der möglichen Bewegungsformen auf die wesentlichen Strukturen zu reduzieren, beobachtet man wie bei einer stroboskopischen Beleuchtung die Bewegung nur nach jeweils festen Zeitintervallen. Das Ergebnis wird als *Poincaré-Schnitt* bezeichnet.

Als Zeitintervall benutzen wir die Antriebsperiode und betrachten

$$(\varphi(t_j), \omega(t_j)) \quad \text{mit} \quad t_j = \frac{2\pi j}{\omega_D} \quad \text{und} \quad j = 0, 1, 2, \dots \, . \tag{4.21}$$

Damit erhalten wir eine Folge von Punkten in der Ebene, wobei jeder Punkt durch seine Vorgänger eindeutig bestimmt wird. Wie bei der Kette auf dem Wellblech (Abschnitt 3.2) reduzieren wir das Problem auf eine zweidimensionale diskrete Abbildung, allerdings mit dem Unterschied, daß für $r > 0$ die Abbildung nicht flächenerhaltend ist. Analog zur eindimensionalen logistischen Abbildung (Abschnitt 3.1) wird ein Flächenstück bei der Iteration kleiner und es entstehen Attraktoren.

Wie sehen nun Trajektorien $(\varphi(t_j), \omega(t_j))$ im Poincaré-Schnitt aus? Bei einer periodischen Bewegung mit der Periodenlänge $2\pi/\omega_0$ sieht man in der stroboskopischen Aufnahme entweder einzelne Punkte oder eine geschlossene Kurve. Für $\omega_0 = \omega_D$ gibt es nur einen einzigen Punkt in der (φ, ω)-Ebene. Für $\omega_0 = (p/q)\,\omega_D$ mit teilerfremden ganzen Zahlen p und q erhält man q verschiedene Punkte, und p bestimmt die Reihenfolge, in der die q Punkte durchlaufen werden. Ist dagegen ω_0 ein irrationales Vielfaches von ω_D, so füllen unendlich viele Punkte $(\varphi(t_j), \omega(t_j))$ eine geschlossene Kurve. Aus periodischen Bewegungen resultieren im Poincaré-Schnitt also entweder einzelne Punkte oder geschlossene Kurven. Solche Bahnen können Attraktoren sein, d. h., startet man mit Punkten $(\varphi(0), \omega(0))$ in einem gewissen Einzugsgebiet, so laufen alle diese Werte zu einem solchen periodischen Attraktor. Es sind auch mehrere Attraktoren mit dazugehörigen Einzugsgebieten möglich.

Es gibt aber noch andere Arten von Attraktoren, sogenannte *seltsame Attraktoren*. Sie kann man sich als eine Art Blätterteig vorstellen, den man durch fortwährendes Strecken und Falten erhält. Solche Attraktoren entsprechen chaotischen Bahnen, die scheinbar unberechenbar durch den Phasenraum laufen. Die seltsamen Attraktoren sind Fraktale (siehe Abschnitt 3.3), sie sind mehr als eine Linie aber weniger als ein Flächenstück.

Die fraktale Dimension D kann man wie im Abschnitt 3.3 durch Überdeckung

des Attraktors mit N Quadraten der Kantenlänge ε bestimmen.

$$D_{\text{Ü}} = - \lim_{\varepsilon \to 0} \frac{\log N(\varepsilon)}{\log \varepsilon} \,. \tag{4.22}$$

Numerisch effektiver aber ist eine Methode, die von Grassberger und Procaccia vorgeschlagen wurde. Dazu erzeugen wir uns N Punkte $\boldsymbol{x}_i$ auf dem Attraktor, die möglichst unkorreliert sein sollen. Dann zählen wir die Anzahl der Punkte $\boldsymbol{x}_j$, die einen Abstand zu $\boldsymbol{x}_i$ haben, der kleiner ist als R. Diese Zahl mitteln wir über $\boldsymbol{x}_i$. Formal können wir diese Korrelation mit der Stufenfunktion $\Theta(x)$ ausdrücken:

$$C(R) = \lim_{N \to \infty} \frac{1}{N(N-1)} \sum_{i \neq j}^{N} \Theta(R - |\boldsymbol{x}_i - \boldsymbol{x}_j|) \,. \tag{4.23}$$

$C(R)$ kann als die mittlere Masse eines Ausschnittes des Attraktors aufgefaßt werden, und die Masse-Länge-Beziehung aus Abschnitt 3.3 definiert eine fraktale Dimension D_{C} durch:

$$C(R) \propto R^{D_{\text{C}}} \,. \tag{4.24}$$

Der log-log-Plot dieser Gleichung sollte also eine Gerade mit der Steigung D_{C} geben. Das gilt aber nur für R-Werte, die größer als der mittlere Abstand der Datenpunkte und kleiner als die Ausdehnung des Attraktors sind.

Eine andere Methode zur Definition der fraktalen Dimension nutzt den Begriff der Informationsentropie. Dazu überdeckt man den Attraktor wieder mit Quadraten der Kantenlänge ε. Dann zählt man, wieviele Punkte der erzeugten Trajektorie in jedem Kästchen liegen. Sei also p_i die Wahrscheinlichkeit, daß ein Punkt (φ, ω) des Attraktors im Quadrat mit der Nummer i liegt. Die Entropie ist dann definiert als

$$I(\varepsilon) = - \sum_i p_i \ln p_i \,, \tag{4.25}$$

und die Informationsdimension D_{I} erhält man als

$$D_{\text{I}} = - \lim_{\varepsilon \to 0} \frac{I(\varepsilon)}{\ln \varepsilon} \,. \tag{4.26}$$

Man kann nun zeigen, daß folgende Beziehung zwischen den drei Dimensionen besteht:

$$D_{\text{Ü}} \leq D_{\text{I}} \leq D_{\text{C}} \,. \tag{4.27}$$

In der Praxis stimmen die drei Dimensionen oft innerhalb des Beobachtungsfehlers überein.

Algorithmus

Zur numerischen Lösung der Differentialgleichung (4.20) wählen wir das Runge-Kutta-Verfahren vierter Ordnung aus dem vorherigen Abschnitt. Um die Bewegung schon während der numerischen Rechnung auf dem Bildschirm sichtbar zu machen, verwenden wir die Sprache C mit der entsprechenden Graphikumgebung. In unserem Programm kann während des Laufes durch Tastendruck die Stärke a des Antriebes geändert und zwischen der kontinuierlichen Bewegung und dem Poincaré-Schnitt hin- und hergeschaltet werden.

Es ist nicht schwer, den Runge-Kutta-Schritt Gleichung (4.11) selbst zu programmieren, aber wir wollen hier demonstrieren, wie die Routine `odeint` aus den *Numerical Recipes* in das eigene Programm eingebunden wird.

`odeint` integriert das Gleichungssystem (4.20) mit adaptiver Schrittweitenkontrolle. Die rechnerische Erzeugung der Trajektorie gibt deshalb nicht deren tatsächlichen zeitlichen Verlauf wieder. Wer diesen Zusammenhang erhalten möchte, sollte direkt die Routine `rk4` aus den *Numerical Recipes* benutzen.

Zunächst müssen alle Variablen der verwendeten Routinen deklariert und die Routinen zum eigenen Programm hinzugefügt werden. Dabei muß beachtet werden, daß `odeint` die Programme `rk4` und `rkqc` benutzt und daß alle Routinen gewisse Fehlerroutinen aufrufen, die in `nrutil` enthalten sind. Man kann alle benötigten Programme direkt in den eigenen Programmtext hineinkopieren oder mit `#include` hinzufügen, z. B. durch

```
#define float double
#include "\tc\recipes\nr.h"
#include "\tc\recipes\nrutil.h"
#include "\tc\recipes\nrutil.c"
#include "\tc\recipes\odeint.c"
#include "\tc\recipes\rkqc.c"
#include "\tc\recipes\rk4.c"
```

Der Pfadname hängt natürlich davon ab, wo der Benutzer die Programme gespeichert hat. Den ersten Befehl fügen wir hinzu, damit alle reellen Variablen im Programm und in den *Recipes* vom selben Typ `double` sind. `odeint` wird in folgender Form aufgerufen:

```
odeint(y,n,t1,t2,eps,dt,0.,&nok,&nbad,derivs,rkqc)
```

Dabei ist `y` ein Vektor mit den Variablen $(\varphi(t_1), \omega(t_1))$. Es wird über das Intervall $[t_1, t_2]$ integriert und danach wird `y` durch die Endwerte $(\varphi(t_2), \omega(t_2))$ ersetzt. `n` ist die Anzahl der Variablen. Da wir in Gleichung (4.20) θ wieder durch $\omega_\text{D}\, t$ ersetzen, gibt es nur `n` $= 2$ Variablen. Mit `eps` geben wir die gewünschte Genauigkeit und mit `dt` eine Abschätzung der benötigten Schrittweite an. Die Variablen `nok`

und `nbad` enthalten bei der Ausgabe Informationen über die Anzahl der benötigten Schritte. `derivs` ist der Name einer Funktion, mit der die rechte Seite von Gleichung (4.20) ausgewertet wird. Mit `derivs(t,y,f)` werden aus `t` und `y` die Komponenten von `f` berechnet. Für unser Beispiel ist

$$\begin{aligned} \texttt{y[1]} &= \varphi\texttt{,} \\ \texttt{y[2]} &= \omega\texttt{,} \end{aligned}$$

und dementsprechend

$$\begin{aligned} \texttt{f[1]} &= \texttt{y[2],} \\ \texttt{f[2]} &= \texttt{-r}*\texttt{y[2]-sin(y[1])+a}*\texttt{cos(}\omega_{\text{D}}*\texttt{t).} \end{aligned}$$

Schließlich übergibt man der Routine `odeint` noch die Funktion `rkqc`, die wiederum den Runge-Kutta-Schritt `rk4` aufruft. Man kann aber auch andere, eventuell eigene Integrationsprozeduren übergeben.

Zusammenfassend besteht also der wesentliche Teil des Programmes `pendel.c` aus den Befehlen:

```
main()
{
y[1]=pi/2.;
y[2]=0.;

while(done==0)
 {
  odeint(y,2,t,t+3.*pi,eps,dt,0.,&nok,&nbad,derivs,rkqc);
  xalt=fmod(y[1]/2./pi +100.5,1.)*xbild;
  yalt=y[2]/ysc*ybild/2+ybild/2;
  rectangle(xalt,yalt,xalt+1,yalt+1);
  t=t+3.*pi;
 }
}

void derivs(double t,double *y,double *f)
 {
        f[1]=y[2];
        f[2]=-r*y[2]-sin(y[1])+a*cos(2./3.*t);
 }
```

Auch hier sollte ein guter Programmierer alle Konstanten vorher berechnen lassen. Die Vektoren `y` und `f` wurden einen Platz größer als erforderlich deklariert, da die *Recipes* die Indizes mit 1 anstatt mit 0 (wie in C) beginnen lassen.

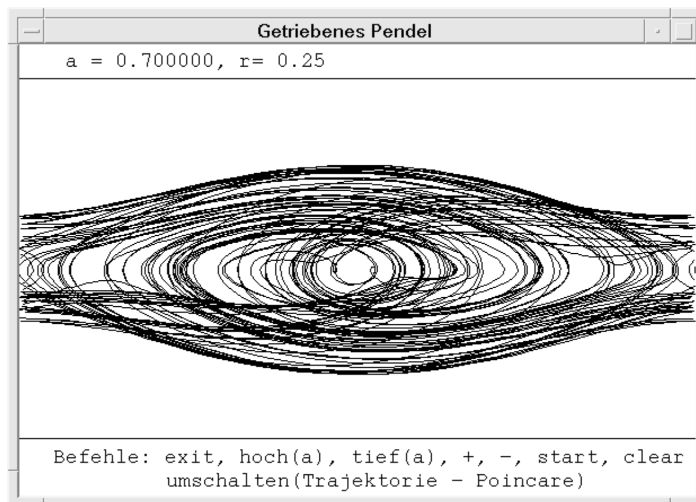
Das obige Programm benutzt $\omega_{\text{D}} = 2/3$ und zeichnet die Punkte $(\varphi(t_i), \omega(t_i))$ (als kleine Rechtecke) jeweils nach einer Antriebsperiode $[t, t + 3\pi]$. Will man die

Bahn kontinuierlich laufen sehen, so muß man $t + 3\pi$ durch $t + dt$ ersetzen, wobei die Bahn im Zeitraum dt möglichst nur wenige Pixel überspringen sollte. Im letzten Fall wird natürlich kein Punkt sondern eine Linie vom Anfangs- zum Endpunkt gezeichnet.

Ergebnisse

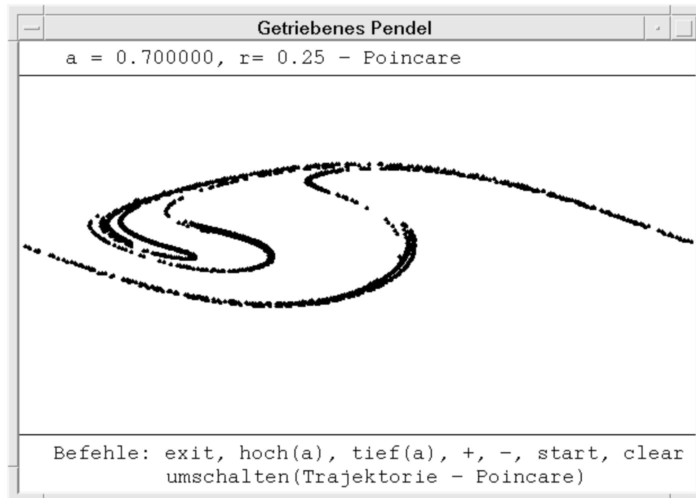
Als Beispiel betrachten wir auf dem Bildschirm das getriebene Pendel mit dem Reibungskoeffizienten $r = 0.25$ und der Antriebsfrequenz $\omega_{\mathrm{D}} = 2/3$. Als Startwerte wählen wir die 90°-Auslenkung, $\varphi(0) = \pi/2$, und die Anfangswinkelgeschwindigkeit $\omega(0) = 0$. Mit der Taste `h` bzw. `t` können wir die Antriebskraft a um jeweils 0.01 verstärken bzw. verringern.

Ohne Antrieb $(a = 0)$ führt das Pendel eine gedämpfte Schwingung aus und hält schließlich in der Ruhelage $(\varphi, \omega) = (0, 0)$ an. Für kleine a-Werte erhalten wir nach einer Einschwingphase eine periodische Bewegung mit der Periode $2\pi/\omega_{\mathrm{D}}$, also einen Punkt im Poincaré-Schnitt. Bei $a = 0.68$ ist der Antrieb so stark, daß sich das Pendel überschlagen kann. Die Bahn verläßt den Bildschirm und tritt wegen der Periodizität des Winkels φ (= horizontale Achse) auf der gegenüberliegenden Seite wieder ein.

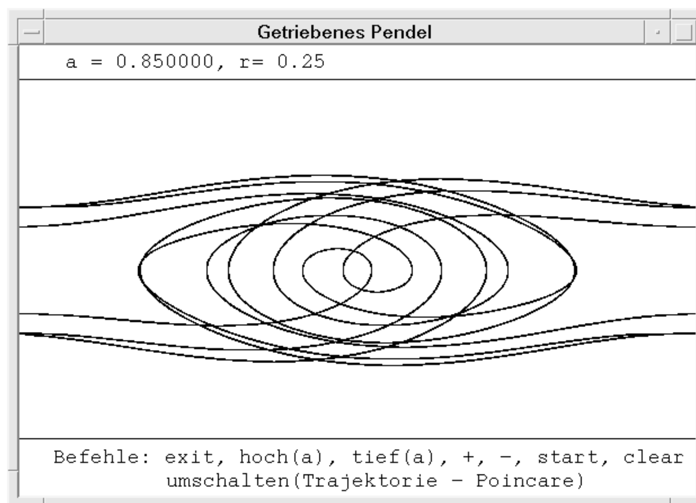


4.3 Bewegung des getriebenen Pendels im Phasenraum (φ, ω), Antriebsstärke $a = 0.7$.

Für $a = 0.7$ beobachten wir einen chaotischen Attraktor (Abbildung 4.3). Im Poincaré-Schnitt deutet sich ein fraktaler „Blätterteig“ an (Abbildung 4.4). Auch für

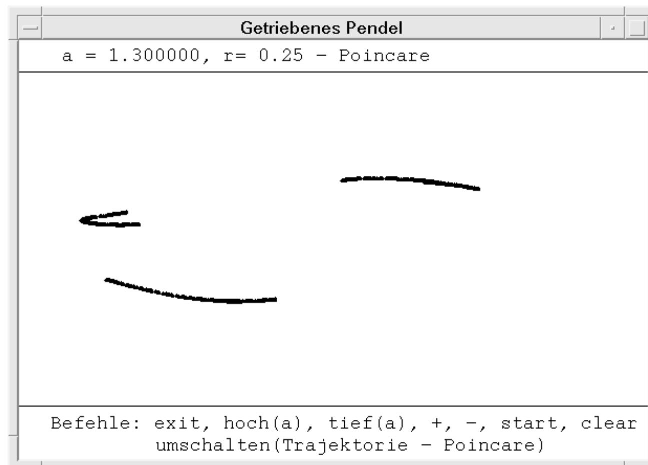


4.4 Wie Abbildung 4.3, aber als Poincaré-Schnitt jeweils zu den Zeiten $2\,\pi\,j/\omega_{\text{D}}$.



4.5 Phasenraum-Plot für $a = 0.85$. Die Bewegung ist ein Zyklus der Periode $7\,\frac{2\pi}{\omega_{\text{D}}}$. Der zugehörige Poincaré-Schnitt zeigt genau 7 Punkte.

$a = 0.85$ sieht die Bahn immer noch regellos aus (Abbildung 4.5), aber der Poincaré-Schnitt zeigt deutlich einen Zyklus der Länge $7 \cdot 2\pi/\omega_{\text{D}}$. Auch Periodenverdoppelung kann man beobachten. Der einfache Zyklus mit Überschlag bei $a = 0.97$



4.6 Poincaré-Schnitt für $a = 1.3$. Die Bewegung wechselt zwischen drei chaotischen Bändern.

hat sich für $a = 0.98$ und $a = 0.99$ verdoppelt und bei $a = 1.0$ vervierfacht. Bei $a = 1.01$ sieht die Bahn schon wieder chaotisch aus. Bei $a = 1.2$ sieht man drei Punkte und bei $a = 1.3$ eine chaotische Bahn mit drei Bändern, die in Abbildung 4.6 als Poincaré-Schnitt zu sehen ist.

Mit wachsender Antriebsstärke a beobachten wir also einen fortwährenden Wechsel von periodischer und chaotischer Bewegung.

Übung

Zwischen zwei Platten ist eine Flüssigkeit mit gegebener Viskosität μ, Wärmeleitfähigkeit κ und Dichte ρ eingesperrt. Die untere Platte hat eine Temperatur $T + \Delta T$, die obere T. Ist der Temperaturunterschied klein, so findet ausschließlich Wärmeleitung statt, die konvektive Bewegung wird durch die viskose Reibung gebremst. Bei Anwachsen der Temperatur beginnt die heiße Flüssigkeit in einigen Bereichen aufzusteigen und in anderen wieder abzusinken. Es entstehen sogenannte *Konvektionsrollen*. Bei weiterer Temperaturerhöhung brechen diese statischen Rollen auf und es entsteht eine chaotische Bewegung.

Unter der Annahme, daß die Konvektionsrollen in y-Richtung unendlich ausgedehnt sind, hat der Meteorologe Lorenz durch Fourierentwicklung in x- und z-Richtung und Vernachlässigung höherer Glieder der Fourierreihe ein Modell ent-

wickelt, das ein solches Bénard-Experiment am Übergang vom geordneten zum chaotischen Verhalten gut beschreibt.

Das System wird in dieser Näherung durch folgende drei Gleichungen beschrieben:

$$\begin{aligned}\dot{X} &= -\sigma X + \sigma Y,\\ \dot{Y} &= -X Z + r X - Y,\\ \dot{Z} &= X Y - b Z.\end{aligned}$$

X stellt dabei die Geschwindigkeit der Zirkularbewegung, Y die Temperaturdifferenz zwischen aufsteigender und fallender Flüssigkeit und Z die Abweichung des resultierenden Temperaturprofils von seinem Gleichgewichtsverlauf dar. b und σ werden allein durch die Materialparameter (μ, ρ, κ) und durch die geometrischen Abmessungen bestimmt und sind somit als Konstanten aufzufassen. r ist proportional zur angelegten Temperaturdifferenz und dient somit als äußerer Steuerparameter, der das Verhalten des Systems bestimmt.

Dieses System ist für diverse Anwendungen interessant:

- in der Meteorologie: Bewegung der Luft,
- in der Astronomie: Sternaufbau bei konvektiven Sternen,
- in der Energietechnik: Wärmeleitung von Dämmstoffen.

Integrieren Sie die Differentialgleichungen des Lorenzmodells mit dem Runge-Kutta-Verfahren. Wählen Sie $\sigma = 10$ und $b = 8/3$. Der Parameter r soll im Programm frei veränderbar sein. Stellen Sie das Ergebnis für X und Y graphisch dar, und zwar

- als Projektion der kontinuierlichen Bewegung auf die X-Y-Ebene,
- als Poincaré-Schnitt zu $Z = \text{const.} = 20.0$.

Um den Schnittpunkt mit der Ebene $Z = \text{const.} = 20$ genauer zu bestimmen, sollte man linear zwischen dem letzten Wert über 20 und dem ersten unter 20 interpolieren.

Ab wann setzt chaotisches Verhalten ein? Welche Dimension hat der seltsame Attraktor im oben genannten Poincaré-Schnitt für $r = 28$?

Literatur

G. L. Baker, J. P. Gallup, *Chaotic Dynamics: An Introduction*, Cambridge University Press, 1991.

H. J. Korsch, H.-J. Jodl, *Chaos: A Program Collection for the PC*, Springer Verlag, 1994.

E. Ott, *Chaos in Dynamical Systems*, Cambridge University Press, 1993.

W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge University Press, 1992.

4.3 Stationäre Zustände

Die Newtonschen Bewegungsgleichungen, so erfolgreich sie unsere makroskopische Welt beschreiben, erweisen sich als nicht geeignet, wenn es um die Bewegung von Elektronen in Atomen, Molekülen und Festkörpern geht. Im mikroskopisch Kleinen sind nur Wahrscheinlichkeitsaussagen für den Ort und den Impuls eines Teilchens möglich, deren zeitliche Änderungen durch die quantenmechanische Schrödinger-Gleichung beschrieben wird. Unter bestimmten Bedingungen reduziert sich diese dynamische Gleichung auf die sogenannte stationäre Schrödinger-Gleichung, eine Eigenwertgleichung für den Energieoperator, die in der Ortsdarstellung die Form einer linearen Differentialgleichung hat. Im Prinzip kann sie mit den Methoden der vorigen Abschnitte numerisch gelöst werden. Es gibt jedoch eine effektivere Methode zur numerischen Lösung der Schrödinger-Gleichung, die wir hier an einem einfachen Beispiel, dem anharmonischen Oszillator, erläutern wollen. Zusätzlich haben wir bei der Schrödinger-Gleichung noch das Problem, daß die Energie nicht alle reellen, sondern nur bestimmte diskrete Werte annehmen kann, die erst aus der Gleichung selbst bestimmt werden müssen.

Physik

Ein Quantenteilchen wird durch eine im allgemeinen komplexwertige Wellenfunktion $\psi(\boldsymbol{r}, t)$ beschrieben, die einer partiellen Differentialgleichung zweiter Ordnung genügt. $|\psi(\boldsymbol{r}, t)|^2$ ist die Wahrscheinlichkeitsdichte, zur Zeit t das Teilchen am Ort $\boldsymbol{r}$ zu messen.

Wenn die Mittelwerte aller Meßgrößen in diesem Zustand zeitunabhängig sind, so gehorcht die Wellenfunktion ψ der stationären Schrödinger-Gleichung, die in einer Raumdimension folgendermaßen lautet:

$$-\frac{\hbar^2}{2m}\psi''(x) + V(x)\,\psi(x) = E\,\psi(x)\,. \tag{4.28}$$

Da das Potential $V(x)$ reell ist, kann auch ψ als reellwertige Funktion gewählt werden. Gleichung (4.28) beschreibt den stationären Zustand eines Teilchens in einer Dimension mit der Masse m im Potential $V(x)$, sie ist linear und enthält eine zweite Ableitung. Aber es gibt eine Besonderheit gegenüber der Newtonschen Bewegungsgleichung: Die Energie E des Teilchens kann nicht alle Werte annehmen, sondern die Gleichung (4.28) bestimmt die möglichen Werte erst dadurch, daß die Wellenfunktion normierbar sein muß, daß also $\int dx\ |\psi(x)|^2$ einen endlichen Wert annimmt. Es stellt sich heraus, daß schon bei einer winzigen Abweichung des Wertes E von einem erlaubten Energiewert die Wellenfunktion für große x-Werte exponentiell anwächst. Diese Tatsache muß bei der numerischen Lösungssuche berück-

sichtigt werden. Wie wir sehen werden, kann man sich andererseits gerade dieses Verhalten zunutze machen, um die gesuchten Energie-Eigenwerte mit großer Genauigkeit numerisch zu bestimmen.

Bei symmetrischen Potentialen $V(x) = V(-x)$ hat auch der stationäre Zustand die Symmetrie $\psi(x) = \pm\psi(-x)$, wobei sich die Vorzeichen mit wachsenden Energiewerten abwechseln. Der Grundzustand ist symmetrisch, $\psi_0(x) = \psi_0(-x)$, und hat keine Nullstelle, $|\psi_0(x)| > 0$. Bei jedem höheren Energie-Eigenwert kommt eine Nullstelle (= Knoten) der Wellenfunktion hinzu. Die Anzahl der Knoten von $\psi(x)$ gibt daher die Nummer des Energieniveaus an.

In unserem Beispiel betrachten wir den anharmonischen Oszillator aus Abschnitt 2.1, dessen Schrödinger-Gleichung die folgende dimensionslose Form hat:

$$\psi''(x) + (-x^2 - \lambda\, x^4 + 2E)\, \psi(x) = 0\,. \tag{4.29}$$

Die Energie E und der Ort x werden dabei in Einheiten von $\hbar\omega$ bzw. $\sqrt{\hbar/(m\omega)}$ gemessen. Im harmonischen Fall $(\lambda = 0)$ kennen wir die Eigenwerte und die Eigenzustände aus analytischen Rechnungen. E^0 kann relativ zur Ruheenergie $E_0^0 = 1/2$ nur ganzzahlige nicht-negative Werte annehmen,

$$E_n^0 = n + \frac{1}{2} \quad \text{mit} \quad n = 0, 1, 2, \, \ldots \tag{4.30}$$

Diese Energieniveaus werden durch einen anharmonischen Beitrag $\lambda\, x^4$ mit $\lambda > 0$ nach oben verschoben. Für negative λ-Werte gibt es keine stationären Zustände, da dann für große Werte von $|x|$ der negative x^4-Term dominiert und das Potential $V(x)$ immer stärker negativ wird. Aufgrund des Tunneleffektes wird jede Wellenfunktion ins Unendliche zerfließen. Das bedeutet, daß $E_n(\lambda)$ nicht analytisch bei $\lambda = 0$ sein kann.

Algorithmus

Mehrere Probleme müssen gelöst werden: Wie findet man die Eigenwerte E ? Welchen Algorithmus benutzt man für die Integration der Schrödinger-Gleichung? Mit welchen Anfangswerten $\psi(x_0)$ und $\psi(x_1)$ startet man die numerische Integration?

Das erste Problem lösen wir mit dem sogenannten Schießverfahren: Wir wählen einen Energiewert E, von dem wir sicher wissen, daß er unterhalb der Grundzustandsenergie E_0 liegt, z. B. $E = 0.5$, und integrieren die Schrödinger-Gleichung vom Startwert x_0 bis zu einem x-Wert, bei dem $|\psi(x)|$ unrealistisch groß geworden ist. Dann erhöhen wir die Energie schrittweise, bis $\psi(x)$ das Vorzeichen wechselt. Jetzt haben wir aufgrund des Knotensatzes ein Energieintervall gefunden, in dem E_0 liegen muß. Nun integrieren wir mit einem E-Wert in der Mitte des Intervalls;

wenn $\psi(x)$ das Vorzeichen wechselt, liegt E_0 in der unteren, sonst in der oberen Hälfte des Intervalls. Auf diese Weise können wir E_0 sehr schnell einschachteln.

Wir „schießen" sozusagen die Wellenfunktion $\psi(x)$ für verschiedene E-Werte solange zu großen x-Werten hin, bis die Randbedingung $|\psi(x)| \to 0$ für große x-Werte einigermaßen erfüllt ist. Allerdings bleibt $\psi(x)$ nur in einem gewissen Intervall dicht beim Wert Null, dann explodiert es wieder ins Unendliche.

Obwohl das zweite Problem, die Integration der Schrödinger-Gleichung, im Prinzip mit dem Runge-Kutta-Verfahren aus Abschnitt 4.1 gelöst werden kann, ist es bei dieser Art von Gleichung möglich, eine größere Genauigkeit mit einer einfacheren Methode, dem Numerov-Verfahren, zu erzielen. Gleichung (4.29) hat die Form:

$$\psi''(x) + k(x)\,\psi(x) = 0 \tag{4.31}$$

mit $k(x) = 2\,(E - V(x))$. Wir diskretisieren mit der Schrittweite h die x-Achse in der Form $x_n = (n - \frac{1}{2})\,h$ für gerade, bzw. $x_n = n\,h$ für ungerade Wellenfunktionen und kennzeichnen die Wellenfunktion und auch deren Ableitungen an den Stellen x_n mit einem unteren Index n, also $\psi_n = \psi(x_n)\,, \psi'_n = \psi'(x_n)$ und analog für die höheren Ableitungen. Aus der Taylor-Entwicklung

$$\psi_{n\pm1} = \psi_n \pm h\psi'_n + \frac{h^2}{2}\psi''_n \pm \frac{h^3}{6}\psi_n^{(3)} + \frac{h^4}{24}\psi_n^{(4)} \pm \ldots \tag{4.32}$$

folgt:

$$\psi_{n+1} + \psi_{n-1} = 2\psi_n + h^2\psi''_n + \frac{h^4}{12}\psi_n^{(4)} + \mathcal{O}(h^6)\,. \tag{4.33}$$

ψ''_n können wir mit Gleichung (4.31) ersetzen,

$$\psi''_n = -k_n\,\psi_n\,. \tag{4.34}$$

Nun kommt der entscheidende Trick, bei dem die vierte Ableitung durch die diskretisierte zweite ersetzt wird:

$$\begin{aligned}\psi_n^{(4)} &= \frac{d^2}{dx^2}\psi''(x)\bigg|_{x_n} = -\frac{d^2}{dx^2}\left(k(x)\,\psi(x)\right)\bigg|_{x_n} \\ &= -\frac{k_{n+1}\,\psi_{n+1} - 2\,k_n\,\psi_n + k_{n-1}\,\psi_{n-1}}{h^2} + \mathcal{O}(h^2)\,.\end{aligned} \tag{4.35}$$

Setzen wir dies in Gleichung (4.33) ein, so erhalten wir:

$$\begin{aligned}&\left(1 + \frac{h^2}{12}k_{n+1}\right)\psi_{n+1} - 2\left(1 - \frac{5}{12}h^2 k_n\right)\psi_n \\ &\qquad + \left(1 + \frac{h^2}{12}k_{n-1}\right)\psi_{n-1} = 0 + \mathcal{O}(h^6)\,.\end{aligned} \tag{4.36}$$

Aus zwei benachbarten Startwerten können damit sämtliche ψ_n-Werte berechnet werden.

Wie werden die beiden Startwerte gewählt? Wegen der Symmetrie des Potentials sind, wie schon erwähnt, die Eigenfunktionen mit aufsteigenden Energiewerten abwechselnd gerade und ungerade. Dementsprechend wählen wir $\psi_0 = \psi_1 \neq 0$ für die geraden und $\psi_0 = 0\,,\psi_1 \neq 0$ für die ungeraden Wellenfunktionen. Der Wert von ψ_1 kann beliebig gewählt werden, denn er ändert wegen der Linearität der Schrödinger-Gleichung nicht die Energie E_k sondern nur den Wert der Normierung $\int |\psi(x)|^2 dx$.

Damit ist der Algorithmus definiert, und wir können ihn leicht programmieren. Als Programmiersprache wählen wir C, um die Wellenfunktion auf dem Bildschirm direkt darstellen und die Energiewerte E_k interaktiv durch Tastendruck einschachteln zu können.

Der Integrationsschritt Gleichung (4.36) wird zur Funktion `step(&x,dx,&y,&ym1)`, die als Ergebnis `yp1` ($= \psi_{n+1}$) liefert. Dabei ist `x` der aktuelle Wert von x_n, `dx` die Schrittweite, und `y`, `ym1` entsprechen ψ_n und ψ_{n-1}. Da wir in `step` die Werte von `x`, `y`, `ym1` verändern wollen, müssen wir die Adressen dieser Variablen übergeben, die jeweils mit einem `&` markiert werden. Man beachte, daß eine Funktion in C nur Werte übergibt und keine Werte an die Argumente zurückgibt.

Damit lautet die Berechnung von ψ_{n+1} :

```
double step(double *xa, double dx, double *ya,
            double *ym1a)
{
   long i,n;
   double k(double);
   double yp1, x, y, ym1, xp1, xm1;
   x=*xa; y=*ya; ym1=*ym1a;
   n=ceil(10./dx/500.);
   for(i=1;i<=n;i++)
      {
            xp1=x+dx;  xm1=x-dx;
            yp1=(2.*(1.-5./12.*dx*dx*k(x))*y
                 -(1.+dx*dx/12.*k(xm1))*ym1)/
                 (1.+dx*dx/12.*k(xp1));
            xm1=x;  x=xp1;
            ym1=y;  y=yp1;
      }
   *xa=x;*ya=y;*ym1a=ym1;
   return yp1;
}
```

Die in dieser Schleife immer wiederkehrenden Rechnungen kann man dadurch beschleunigen, daß man Konstanten wie `5./12.*dx*dx` vorher in eine Variable schreibt. Die Iterationsschleife `for` (...) läuft soweit, daß gerade ein Pixel auf der x-Achse des Bildschirms überbrückt wird. (`xb`, `yb`) sind die Koordinaten auf dem Bildschirm. `xb` wird mit einer `for`-Schleife jeweils um den Wert 1 erhöht, während `ybneu` aus `yp1` berechnet wird. Als Startwert für ψ_1 wählen wir 1.0. Die Hauptschleife für die geraden Wellenfunktionen lautet damit:

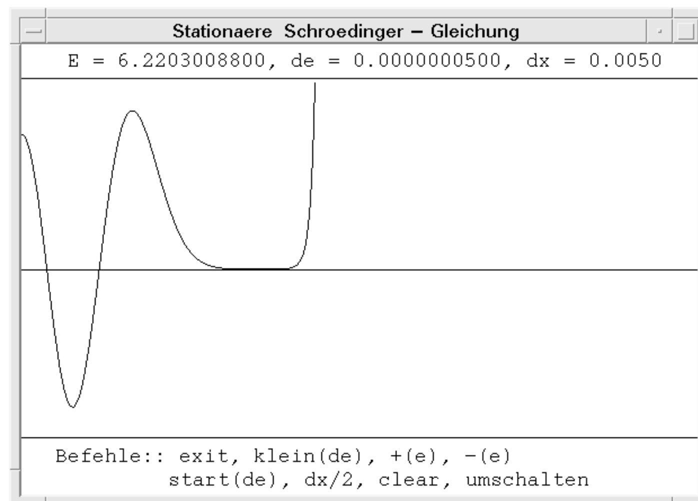
```
x=dx/2.; y=ym1=1.; ybalt=1;
for(xb=1; xb<XMAX; xb++)
   {
     yp1=step(&x,dx,&y,&ym1);
     ybneu=(1.-yp1)*YMAX;
     if(abs(ybneu)>10000) break;
     line(xb-1,ybalt,xb,ybneu);
     ybalt=ybneu;
    }
```

Dabei wird als Explosionskriterium für ψ_n benutzt, daß die y-Koordinate den Wert 10000 überschreitet. Auf dem PC wird eine Linie von `(xb-1,ybalt)` nach `(xb, ybneu)` gezeichnet. Nach dieser Schleife wird auf eine Tastatureingabe mit `ch= getch()` gewartet, und je nach Eingabe werden der Wert von `e` $(= E)$, die Energieschrittweite `de` oder die Schrittweite der Integration `dx` $(= h)$ geändert.

Ergebnis

Das Programm `schroedinger` berechnet und zeichnet die Wellenfunktion $\psi(x)$ für die Stärke $\lambda = 0.1$ des anharmonischen Potentials. Bild 4.7 zeigt das Ergebnis für $E = 6.22030088$. Wegen der Symmetrie $\psi(x) = \psi(-x)$ (nur der Bereich $x \geq 0$ ist gezeichnet) hat die Wellenfunktion vier Nullstellen, wir suchen also das vierte Energieniveau E_4 über dem Grundzustand. Für $\lambda = 0$ gilt $E_4^0 = 4.5$. Der anharmonische Term mit $\lambda = 0.1$ verschiebt daher dieses Niveau erheblich nach oben. Die Wellenfunktion $\psi(x)$ aus Bild 4.7 liegt schon sehr dicht an dem gebundenen Zustand $\psi_4(x)$. Das sieht man daran, daß sich $\psi(x)$ sehr lange an die x-Achse anschmiegt, bevor es exponentiell stark nach $+\infty$ schießt. Erhöht man E um den Wert 0.00000005, so divergiert das entsprechende $\psi(x)$ nach $-\infty$. Die Energie E_4 liegt also im Intervall

$$[\,6.22030088, 6.22030093\,].$$



4.7 Wellenfunktion $\psi_4(x)$ mit vier Knoten für den fünften Energie-Eigenwert.

Im Abschnitt 2.1 haben wir die Energieniveaus mit einer anderen Methode berechnet. Die unendliche Matrix des Hamiltonoperators wurde dort durch eine $N \times N$-Untermatrix genähert und diagonalisiert. Für $N = 50$ fanden wir den Wert

$$E_4 = 6.220300900006516\,,$$

der mit dem obigen Resultat übereinstimmt. Der Vorteil der direkten Integration liegt darin, daß wir eine obere und untere Schranke erhalten. Deren Genauigkeit können wir prüfen, indem wir die Schrittweite `dx` der Integration halbieren.

Übung

Im Abschnitt 2.1 sollten die vier niedrigsten Energieniveaus des Doppelmuldenpotentials mit der Matrix-Methode berechnet werden. Hier soll dasselbe Problem durch Integration der Schrödinger-Gleichung gelöst werden. Das Potential V sei also in dimensionsloser Form

$$V(x) = -2x^2 + x^4/10\,.$$

Berechnen sie mit dem Schießverfahren die niedrigsten Energien und die dazugehörigen Wellenfunktionen, und vergleichen Sie Ihre Ergebnisse mit denen aus dem Abschnitt 2.1.

Literatur

S. Brandt, H. D. Dahmen, *Quantum Mechanics on the Personal Computer*, Springer Verlag, 1992.

S. E. Koonin, D. C. Meredith, *Physik auf dem Computer, Band 1+2*, R. Oldenbourg Verlag, 1990.

E. W. Schmid, G. Spitz, W. Lösch, *Theoretische Physik mit dem Personal Computer*, Springer Verlag, 1987.

J. Schnakenberg, *Algorithmen in der Quantentheorie und Statistischen Physik*, Zimmermann-Neufang, 1995.

J. Stoer, R. Bulirsch, *Numerische Mathematik 2*, Springer Verlag, 1990.

4.4 Solitonen

Bisher haben wir nur Differentialgleichungen von Funktionen betrachtet, die von *einer* Variablen (Zeit oder Ort) abhängen. Jetzt wollen wir eine Welle betrachten, die sich mit der Zeit fortbewegt und eventuell ihre Form ändert. Die zeitliche Änderung ist dabei mit der räumlichen gekoppelt, man erhält eine partielle Differentialgleichung.

Wir wollen uns in diesem Abschnitt mit der *Korteweg-de-Vries-Gleichung* (KdV-Gleichung) beschäftigen und ausnahmsweise eine analytisch bekannte Lösung dieser Gleichung auch auf numerischem Wege gewinnen. Die KdV-Gleichung ist eine nichtlineare Differentialgleichung mit speziellen Lösungen, den sogenannten Solitonen, die einige überraschende Eigenschaften zeigen. So können z. B. zwei Solitonen aufeinander zulaufen, sich gegenseitig durchdringen und danach ohne Formänderung mit der ursprünglichen Geschwindigkeit wieder getrennt weiterlaufen.

Physik

In der Theorie von Wasserwellen in seichten Kanälen wird die Auslenkung $u(x,t)$ über dem mittleren Wasserspiegel zur Zeit t am Ort x durch eine nichtlineare partielle Differentialgleichung beschrieben. In dimensionsloser und geeignet skalierter Form lautet die KdV-Gleichung

$$\frac{\partial u}{\partial t} = 6\,u\,\frac{\partial u}{\partial x} - \frac{\partial^3 u}{\partial x^3}\,. \tag{4.37}$$

Im Term $u\frac{\partial u}{\partial x}$ ergibt eine doppelte Auslenkung einen vierfachen Beitrag, die Gleichung ist also nichtlinear.

Unter den vielen Lösungen der KdV-Gleichung gibt es eine ganz charakteristische, nämlich

$$u(x,t) = -2\,\mathrm{sech}^2(x-4t) \tag{4.38}$$

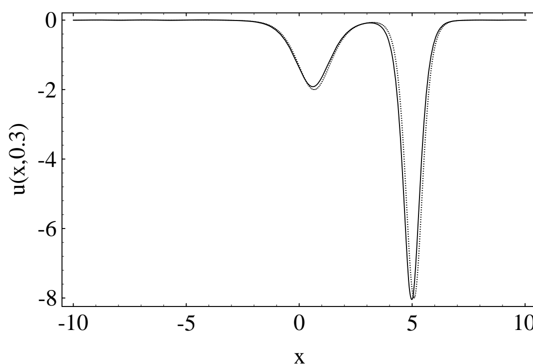
mit $\mathrm{sech}\,x = 1/\cosh x$. Diese Funktion $u(x,t)$ beschreibt ein Wellental, das sich mit der Geschwindigkeit $v = 4$ nach rechts bewegt und dabei seine Form nicht ändert. Eine solche Lösung wird als Soliton bezeichnet.

Man kennt aber auch komplexere Lösungen. Die mathematische Theorie dazu ist nicht einfach, so daß wir hier nur das Ergebnis angeben. Aus dem Anfangszustand

$$u(x,0) = -N(N+1)\,\mathrm{sech}^2(x) \tag{4.39}$$

entstehen N Solitonen, die sich mit unterschiedlichen Geschwindigkeiten fortbewegen. Für $N = 2$ findet man z. B.

$$u(x,t) = -12\,\frac{3+4\cosh(2x-8t)+\cosh(4x-64t)}{[3\cosh(x-28t)+\cosh(3x-36t)]^2}\,. \tag{4.40}$$



4.8 Zwei Solitonen entfernen sich voneinander. Die numerische Lösung (durchgezogen) stimmt gut mit der exakten (gepunktet) überein ($t = 0.3$, $dx = 0.18$, $dt = 0.002$).

Bild 4.8 zeigt, daß diese Lösung zwei Solitonen enthält. Eine Welle mit großer Amplitude hat eine breitere Welle mit kleiner Amplitude überholt. Zur Zeit $t = 0$ überlagern sich beide zu dem Wellenpaket (4.39), während nach dem Überholvorgang beide Solitonen wieder die gleiche Form wie vorher haben.

Es soll noch ein interessanter Zusammenhang mit der Quantenmechanik erwähnt werden: Die Schrödinger-Gleichung mit dem Potential $V(x) = -N(N+1)\,\mathrm{sech}^2(x)$ hat N gebundene Zustände, aus denen man mit den Methoden der inversen Streutheorie die Lösung der KdV-Gleichung konstruieren kann. Da dies nicht einfach zu erklären ist, verweisen wir auf die Literatur zu Solitonen.

Algorithmus und Ergebnis

Um die KdV-Gleichung (4.37) numerisch zu lösen, müssen wir zunächst den Raum x und die Zeit t diskretisieren,

$$u_n^j = u(j\,dx, n\,dt)\,. \tag{4.41}$$

j und n sind ganze Zahlen, dx und dt die Schrittweiten in x- und t-Richtung. Wir wollen zunächst demonstrieren, was passiert, wenn wir die Gleichung (4.37) zu naiv in Differenzenform schreiben. Zum Beispiel können wir konsequent für jede Ableitung $f'(x_k)$ die sogenannte Vorwärts-2-Punkt-Formel verwenden, d. h. $f'(x_k)$ durch $(f(x_k+h) - f(x_k))/h$ approximieren. Aus den partiellen Ableitungen wird dann:

$$\frac{\partial u}{\partial t} \;\to\; \frac{1}{dt}\left(u_{n+1}^j - u_n^j\right), \quad \frac{\partial u}{\partial x} \;\to\; \frac{1}{dx}\left(u_n^{j+1} - u_n^j\right), \tag{4.42}$$

Indem wir die Regel dreimal iterieren, erhalten wir

$$\frac{\partial^3 u}{\partial x^3} \;\to\; \frac{1}{(dx)^3}\left(u_n^{j+3} - 3u_n^{j+2} + 3u_n^{j+1} - u_n^j\right). \tag{4.43}$$

Mit diesen Ersetzungen lösen wir Gleichung (4.37) nach u_{n+1}^j auf und erhalten:

$$u_{n+1}^j = u_n^j + dt\left(6u_n^j\,\frac{u_n^{j+1} - u_n^j}{dx} - \frac{u_n^{j+3} - 3u_n^{j+2} + 3u_n^{j+1} - u_n^j}{(dx)^3}\right). \tag{4.44}$$

Diese Gleichung ist leicht zu programmieren. In *Mathematica* stellen wir $\{u_n^j\}_{j=0}^{\max}$ als Liste mit `max` $+1$ Elementen dar. Zur Initialisierung verwenden wir Gleichung (4.39) mit $N = 2$,

```
ustart:=Table[-6 Sech[(j-max/2)dx]^2//N,{j,0,max}]
```

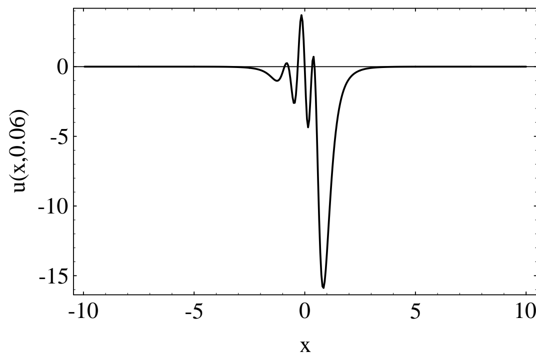
und nehmen periodische Randbedingungen an. Wir verschaffen uns mit `uplus[k] =RotateLeft[u,k]` für $k = 1, 2$ und 3 die verschobenen Listen $\{u_n^{j+k}\}$ und können dann den Integrationsschritt $\{u_n^j\} \to \{u_{n+1}^j\}$ folgendermaßen formulieren:

```
step[u_]:=(Do[uplus[k]=RotateLeft[u,k],{k,3}];
       u+dt(6 u(uplus[1]-u)/dx -
         (uplus[3]-3 uplus[2]+3 uplus[1]-u)/dx^3))
```

Mit `plot2[i_:3]` zeichnen wir für $dx = 0.05$ und $dt = 0.02$ das Ergebnis nach `i` Integrationsschritten.

```
plot2[i_:3]:= (dx=0.05; dt=0.02; upast=ustart; zeit=0;
 Do[upres=step[upast];
    upast=upres;Print["Zeit ",zeit=zeit+dt],{i}] ;
    xulist = Table[{(j-max/2)dx,upres[[j]]},{j,0,max}];
    ListPlot[xulist, PlotJoined->True, PlotRange->All])
```

Abbildung 4.9 zeigt das Resultat. Schon nach drei Zeitschritten treten in $u(x,t)$ numerisch bedingte Oszillationen auf, die nach zwei weiteren Schritten „explodieren“ ; dies ist natürlich unphysikalisch und auf den schlechten Algorithmus zurückzuführen.



4.9 Integration mit `step[u]` und Schrittweiten $dx = 0.05$, $dt = 0.02$. Schon nach drei Integrationsschritten erhält man ein unphysikalisches Ergebnis. Man vergleiche mit Bild 4.8.

Was läuft falsch? Wir haben mehrere Fehler gemacht, die wir durch folgende Modifikationen vermeiden können:

1. Durch besseres Diskretisieren der Ableitungen kann man den Fehler der Approximation von $\frac{\partial}{\partial t}$ und $\frac{\partial^n}{\partial x^n}$ reduzieren,

2. Mittelwertbildung stabilisiert den Algorithmus,

3. dx muß immer relativ groß zu dt gewählt werden.

Punkt 1 kann man sich auf folgende Weise klarmachen: Wir betrachten die Taylorreihe von $u(x,0)$ um $x=0$ mit $u_0' = \frac{\partial}{\partial x} u(x,0)|_{x=0}$ usw.

$$u_{\pm 1} = u_0 \pm dx\, u_0' + \frac{(dx)^2}{2}\, u_0'' \pm \frac{(dx)^3}{6}\, u_0''' + \cdots \,.$$

Dann gilt:

$$\begin{aligned} u_0' &= \frac{u_1 - u_0}{dx} - \frac{dx}{2}\, u_0'' + \cdots \,, \\ u_0' &= \frac{u_1 - u_{-1}}{2\,dx} - \frac{(dx)^2}{6}\, u_0''' + \cdots \,. \end{aligned} \tag{4.45}$$

Man sieht, daß man im zweiten Fall u_0' so berechnen kann, daß $\frac{dx}{2}\, u_0''$ gerade kompensiert wird. Dadurch wird der Fehler um eine Größenordnung reduziert. Auch

bei höheren Ableitungen kann man so durch geeignete Wahl der Koeffizienten den Fehler reduzieren. Man erhält

$$u_0'' = \frac{u_1 - 2u_0 + u_{-1}}{dx^2} + \mathcal{O}(dx^2)\,,$$
$$u_0''' = \frac{u_2 - 2u_1 + 2u_{-1} - u_{-2}}{2\,dx^3} + \mathcal{O}(dx^2)\,. \tag{4.46}$$

Bei dem in (4.44) verwandten Algorithmus wurde also hinsichtlich der Güte der Approximation bei sämtlichen Ableitungen eine Größenordnung in dx und dt verschenkt.

Punkt 2 und 3 wollen wir an einer einfachen linearen Differentialgleichung erläutern, und zwar wählen wir

$$\frac{\partial u}{\partial t} = -v\,\frac{\partial u}{\partial x}$$

mit einer konstanten Geschwindigkeit v. Wir diskretisieren diese Gleichung in der Form

$$\frac{u_{n+1}^j - u_n^j}{dt} = -v\,\frac{u_n^{j+1} - u_n^{j-1}}{2\,dx}\,, \tag{4.47}$$

wobei wir für $\frac{\partial}{\partial x}$ schon die verbesserte Version (4.45) eingesetzt haben. Aufgelöst nach den Komponenten $(u_{n+1}^1, u_{n+1}^2, \ldots)$, die wir zu $\boldsymbol{u}_{n+1}$ zusammenfassen, läßt sich (4.47) als Matrixgleichung

$$\boldsymbol{u}_{n+1}^\mathsf{T} = \mathsf{M}\,\boldsymbol{u}_n^\mathsf{T} \tag{4.48}$$

mit der tridiagonalen Matrix M formulieren. Die Lösung von (4.48) ist offensichtlich $\boldsymbol{u}_n^\mathsf{T} = \mathsf{M}^n\boldsymbol{u}_0^\mathsf{T}$, an der wir erkennen, daß für die Stabilität des Algorithmus der Betrag der Eigenwerte von M entscheidend ist. Die Eigenmoden $\boldsymbol{w}$ von Matrizen dieses Typs sind immer von der Form $w^j = \exp(i\,k\,j\,dx)$ mit einem Wellenvektor k, dessen mögliche Werte mit den Randbedingungen zusammenhängen. Wählen wir eine Eigenmode $\boldsymbol{w} = \boldsymbol{w}_0$ als Anfangsauslenkung und bezeichnen den zugehörigen Eigenwert mit a, so ergibt die n-fache Anwendung der Matrix M :

$$w_n^j = a^n\,e^{i\,k\,j\,dx} \tag{4.49}$$

Setzen wir dies in (4.47) ein, so erhalten wir eine Bestimmungsgleichung für die Amplitude a:

$$\frac{a^{n+1} - a^n}{dt} = -v\,a^n\,\frac{e^{ikdx} - e^{-ikdx}}{2dx}\,. \tag{4.50}$$

Die Lösung ist:

$$a = 1 - \mathrm{i}\,\frac{vdt}{dx}\sin(k\,dx)\,. \tag{4.51}$$

a ist also komplex, und der Betrag ist außer für $k = 0$ immer größer als eins. Das heißt, daß jede Anfangsauslenkung, die nicht gerade konstant ist, mit der Zeit exponentiell anwächst. Der Algorithmus ist unbrauchbar!

Eine scheinbar unwesentliche Änderung kann das Verfahren aber stabilisieren: Wir ersetzen in Gleichung (4.47) u_n^j durch den Mittelwert:

$$u_n^j \to \frac{1}{2}\left(u_n^{j+1} + u_n^{j-1}\right) . \tag{4.52}$$

Dann erhalten wir

$$u_{n+1}^j = \frac{1}{2}\left(u_n^{j+1} + u_n^{j-1}\right) - \frac{vdt}{2dx}\left(u_n^{j+1} - u_n^{j-1}\right) , \tag{4.53}$$

und die Amplitudengleichung ergibt:

$$a = \cos(kdx) - \mathrm{i}\,\frac{vdt}{dx}\sin(kdx) . \tag{4.54}$$

Nun ist $|a| \leq 1$ für

$$|v|dt \leq dx . \tag{4.55}$$

Diese Ungleichung heißt Courant-Bedingung. Sie besagt, daß Störungen nur dann stabil bleiben, wenn der Zeitschritt dt kleiner als die Ausbreitungszeit $dx/|v|$ gewählt wird. Oder umgekehrt: Die räumliche Diskretisierung darf nicht zu klein gewählt werden!

Nach Gleichung (4.46) ist es besser, auch die Zeitableitung symmetrisch zu diskretisieren:

$$u_{n+1}^j - u_{n-1}^j = -\frac{vdt}{dx}\left(u_n^{j+1} - u_n^{j-1}\right) . \tag{4.56}$$

Dies gibt die Amplitudengleichung

$$\left(a - \frac{1}{a}\right) = -2\,i\,\frac{vdt}{dx}\,\sin kdx . \tag{4.57}$$

Die Lösung

$$a = -i\,\frac{vdt}{dx}\,\sin kdx \pm \sqrt{1 - \left(\frac{vdt}{dx}\,\sin kdx\right)^2} \tag{4.58}$$

zeigt, daß $|a|$ den Wert 1 hat, wenn die Courant-Bedingung (4.55) gilt. Auch hier muß man dt relativ zu dx klein genug wählen.

Gleichung (4.56) enthält nun drei Zeitschritte, nämlich $n+1, n$ und $n-1$. Um die Zukunft zu berechnen, müssen wir Gegenwart und Vergangenheit kennen. Im ersten Schritt kennen wir nur die Gegenwart, die Zukunft muß also in einem Schritt,

z. B. mit Gleichung (4.53) berechnet werden. Alle weiteren Schritte können dann mit Gleichung (4.56) durchgeführt werden.

Diese Erkenntnisse wollen wir nun für die numerische Lösung der KdV-Gleichung nutzen. Wir setzen:

$$\begin{aligned}
\frac{\partial u}{\partial t} &\rightarrow \frac{u_{n+1}^{j} - u_{n-1}^{j}}{2dt}\,, \\
\frac{\partial u}{\partial x} &\rightarrow \frac{u_{n}^{j+1} - u_{n}^{j-1}}{2dx}\,, \\
u &\rightarrow \frac{1}{3}\left(u_{n}^{j+1} + u_{n}^{j} + u_{n}^{j-1}\right), \\
\frac{\partial^3 u}{\partial x^3} &\rightarrow \frac{1}{2(dx)^3}\left(u_{n}^{j+2} - 2u_{n}^{j+1} + 2u_{n}^{j-1} - u_{n}^{j-2}\right).
\end{aligned}$$

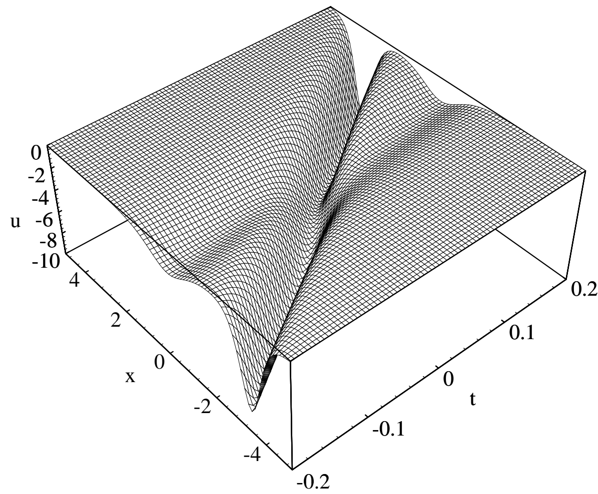
$n-1$ nennen wir Vergangenheit, n Gegenwart und $n+1$ Zukunft. Dann lautet ein Zeitschritt

```
step2[u_,w_]:=(up1=RotateLeft[u]; up2=RotateLeft[up1];
               um1=RotateRight[u];um2=RotateRight[um1];
               w+dt(2(um1+u+up1)*(up1-um1)/dx -
                    (up2-2 up1+2 um1-um2)/dx^3 ) )
```

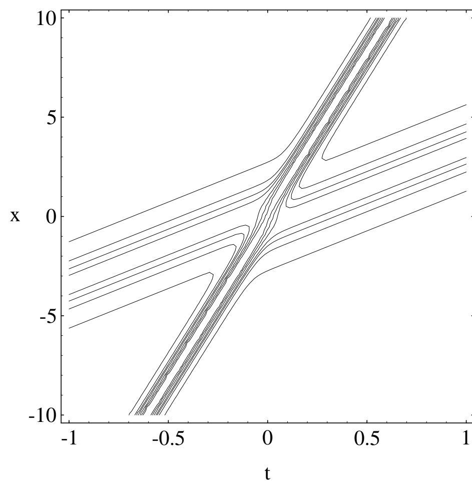
Mit `step2` erzeugen wir die Liste `ufut` der Zukunft, wenn wir die Gegenwart (Liste `upres`) und die Vergangenheit (Liste `upast`) einsetzen. Mit `plot3[i_:9]` wird dies iteriert und nach `i` Zeitschritten gezeichnet, wobei wir vorher mit `Interpolation[xulist]` durch die x-u-Werte eine glatte Kurve hindurchgelegt haben.

```
plot3[i_:9]:=
  (Do[ufut=step2[upres,upast];
      upast=upres;upres=ufut; zeit = zeit+dt, {i}];
   Print["Zeit ",zeit];
   xulist=Table[{(j-max/2)*dx,ufut[[j+1]]},{j,0,max}];
   uu=Interpolation[xulist];
   Plot[uu[x],{x,-10.,10.},PlotRange->All,
               Frame -> True, Axes -> None ] )
```

Bild 4.8 zeigt das Ergebnis für den Anfangszustand $u(x,0) = -6\,\mathrm{sech}^2(x)$. Die Übereinstimmung zwischen numerischer und exakter Lösung, die als punktierte Kurve ebenfalls eingezeichnet ist, kann als gut bezeichnet werden. Aus dem Wellental $u(x,0)$ entstehen zwei Solitonen, die sich mit unterschiedlicher Geschwindigkeit nach rechts bewegen. Dies wird besonders im 3D-Plot (Bild 4.10) von $u(x,t)$ und

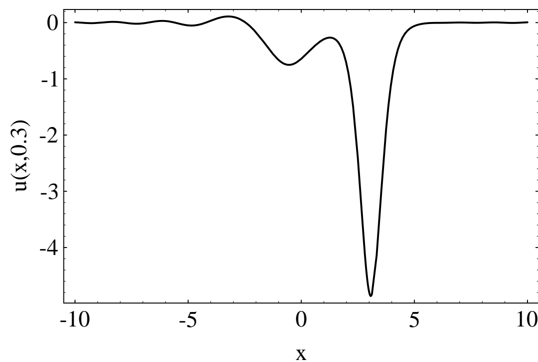


4.10 Zwei Solitonen wie in Bild 4.8, aber als 3D-Plot für verschiedene Zeiten. Ein großes, schnelles Soliton überholt ein kleines breites.



4.11 Wie Bild 4.10, aber als Contour-Plot mit anderen Bereichen von Ort und Zeit. Beim Überholvorgang erfährt das kleine und breite Soliton eine Verzögerung, während das große beschleunigt wird.

im dazugehörigen Contour-Plot (Bild 4.11) deutlich. Für negative Zeiten nähert sich das schnelle Soliton mit großer Amplitude dem langsamen. Bei $t = 0$ überlagern sich beide zu einem einzigen Wellental, und nach dem Überholvorgang haben beide wieder die ursprüngliche Gestalt. Beim Überholen erfährt das kleine Soliton eine Verzögerung, während das große beschleunigt wird. Dies ist besonders gut im Contour-Plot Bild 4.11 zu erkennen.



4.12 Wie Bild 4.8, aber mit dem Startzustand $u(x,0) = -4\,\mathrm{sech}^2\,x$. Zusätzlich zum nach rechts laufenden Soliton werden Wellen abgestrahlt.

Die eingangs erwähnten analytischen Lösungen haben als Anfangszustand $-N(N+1)\,\mathrm{sech}^2\,x$, der in N Solitonen zerfällt. Die Amplituden -2 und -6 ergeben also die Ein- bzw. Zwei-Soliton-Lösung. Was aber geschieht mit Anfangszuständen, deren Amplitude dazwischen liegt? Bild 4.12 zeigt das numerische Ergebnis für den Startzustand $u(x,0) = -4\,\mathrm{sech}^2\,x$. Zusätzlich zum Soliton, das sich nach rechts bewegt, zerfließen nach links weitere Wellen.

Selbstverständlich läßt sich durch Verkleinern der Schrittweite dx die Güte der numerischen Approximation verbessern. Der Aufwand steigt dabei aber beträchtlich. Klar ist, daß die Länge der Listen $\{u_n^j\}_{j=0}^{\max}$ wie $1/dx$ anwächst. Daß aber aus Stabilitätsgründen zugleich das maximale dt proportional zu $(dx)^3$ verkleinert werden muß, wollen wir in der folgenden Übung behandeln.

Übung

Die Stabilität des in `step2[u_,w_]` programmierten Algorithmus wird entscheidend durch die diskretisierte dritte Ableitung begrenzt.

a) Streichen Sie in der vollen KdV-Gleichung den nichtlinearen Term. Untersuchen Sie also die Gleichung

$$\frac{\partial u}{\partial t} = -\frac{\partial^3 u}{\partial x^3}\,. \tag{4.59}$$

Die hier benutzte Diskretisierung ergibt

$$u_{n+1}^j = u_{n-1}^j - dt\,(u_n^{j+2} - 2u_n^{j+1} + 2u_n^{j-1} - u_n^{j-2})/(dx)^3\,. \tag{4.60}$$

Untersuchen Sie diesen Algorithmus auf seine Stabilität, indem Sie in (4.60) den Ansatz (4.49) verwenden. Zeigen Sie, daß analog zu (4.57) hieraus die Amplitu-

dengleichung

$$a = \frac{1}{a} - 2\,i\,dt\,(\sin(2kdx) - 2\sin(kdx))\,/(dx)^3$$

folgt. Bestimmen Sie numerisch das Maximum μ von $|\sin(2kdx) - 2\sin(kdx)|$ ($\mu \simeq 2.6$) und zeigen Sie, daß sich hieraus und aus der Forderung $|a| \leq 1$ die folgende Stabilitätsbedingung ergibt:

$$dt \leq \frac{1}{\mu}\,(dx)^3\,. \tag{4.61}$$

b) Benutzen Sie den Algorithmus `step2[u,w]` zur Integration und die Anfangsbedingung $u(x,0) = -6\,\mathrm{sech}^2\,x$. Variieren Sie dx zwischen 0.1 und 0.2 und bestimmen Sie zu gegebenem dx die Stabilitätsgrenze hinsichtlich dt, d. h. bestimmen Sie ungefähr das maximale dt in Abhängigkeit von dx. Bestätigen Sie in einem log-log-Plot die Gleichung (4.61).

Literatur

G. Baumann, *Mathematica in der Theoretischen Physik*, Springer Verlag, 1993.

R. E. Crandall, *Mathematica for the Sciences*, Addison Wesley, 1991.

4.5 Zeitabhängige Schrödinger-Gleichung

Wenn sich ein Teilchen in einem Kasten ohne Reibung und ohne sonstige Kräfte bewegt, so ändert sich seine Bewegung nur dadurch, daß es an den Wänden reflektiert wird. In einer Dimension läuft es also gleichmäßig hin und her. Dieses klassische Bild, das von der Vorstellung einer Punktmasse mit scharfem Ort und Impuls ausgeht, trifft aber nicht mehr zu, wenn der Kasten mikroskopisch klein ist. Dann müssen wir das Teilchen durch eine Wellenfunktion beschreiben. In der Quantenmechanik-Vorlesung lernt man, daß die stationären Zustände in diesem Fall stehende Wellen sind. Was aber passiert mit einem anfangs lokalisierten Wellenpaket, das gegen die Wände des Kastens läuft?

Dieses Problem hat sowohl analytische als auch numerische Aspekte. Wir werden die Entwicklung nach Eigenfunktionen benutzen, um Aussagen über Symmetrien, Wiederkehrzeiten und weitere charakteristische Längen- und Zeitskalen zu gewinnen. Andererseits wollen wir anhand der zeitabhängigen Schrödinger-Gleichung demonstrieren, wie man partielle Differentialgleichungen numerisch mit einem impliziten Schema löst. Dabei ist eine tridiagonale Matrix zu invertieren, wofür es ein schnelles numerisches Verfahren gibt. Außerdem ist bei der Diskretisierung der

quantenmechanischen Zeitentwicklung darauf zu achten, daß die Norm der Wellenfunktion sich zeitlich nicht ändert.

Wir werden sehen, daß das scheinbar einfache und wohlverstandene Lehrbuch-Beispiel ein überraschend komplexes Verhalten zeigt. Das Wellenpaket zerfließt, es entstehen wilde Interferenzmuster, die sich plötzlich wieder zu glatten Wellenpaketen rekonstruieren. Schließlich wiederholt sich der ganze Prozeß periodisch mit der Zeit. Das Titelbild dieses Lehrbuchs zeigt, welche Komplexität und Schönheit aus elementarer Quantenmechanik entstehen kann.

Physik

Es sei $\psi(x,t)$ die komplexwertige Wellenfunktion eines Teilchens mit der Masse m, das sich in einer Dimension in einem Potential $\tilde{V}(x)$ bewegt. $|\psi(x,t)|^2 dx$ ist dann die Wahrscheinlichkeit, das Teilchen zur Zeit t im Intervall $[x, x+dx]$ anzutreffen. Die zeitliche Änderung von $\psi(x,t)$ wird durch die Schrödinger-Gleichung in der Ortsdardarstellung beschrieben:

$$i\,\hbar\,\frac{\partial\psi}{\partial t} = -\frac{\hbar^2}{2m}\,\frac{\partial^2\psi}{\partial x^2} + \tilde{V}(x)\,\psi\,. \tag{4.62}$$

Um diese Gleichung in eine dimensionslose Form zu bringen, skalieren wir die Zeit mit t_0 und den Ort mit x_0:

$$i\,\frac{\hbar}{t_0}\,\frac{\partial\psi}{\partial(t/t_0)} = -\frac{\hbar^2}{2mx_0^2}\,\frac{\partial^2\psi}{\partial(x/x_0)^2} + \tilde{V}(x)\,\psi\,. \tag{4.63}$$

Nun wählen wir t_0 und x_0 so, daß folgende Gleichung erfüllt ist:

$$\hbar\,t_0 = 2mx_0^2\,. \tag{4.64}$$

Wenn wir nun $\tilde{V}(x)\,t_0/\hbar = V(x/x_0)$ setzen und Ort und Zeit in Einheiten von x_0 und t_0 messen, so erhalten wir die dimensionslose Gleichung

$$i\,\frac{\partial\psi}{\partial t} = H\psi = \left[-\frac{\partial^2}{\partial x^2} + V(x)\right]\psi\,. \tag{4.65}$$

H ist der skalierte Hamiltonoperator des Teilchens. Zur numerischen Lösung dieser Gleichung gibt es zwei Wege: Erstens können wir die Eigenzustände und Eigenwerte der stationären Gleichung $H\psi = E\psi$ berechnen, $\psi(x,0)$ nach diesen Zuständen entwickeln und dann für $\psi(x,t)$ eine Reihendarstellung angeben. Zweitens können wir die zeitabhängige Gleichung wie im vorigen Abschnitt direkt integrieren. Der zweite Weg ist auch für Probleme anwendbar, bei denen die erste Methode versagt, so daß wir im Algorithmus-Teil diesen Weg beschreiben wollen. Wenn andererseits

die Eigenzustände bekannt sind, so kann man mit dem analytischen Ansatz zumindest einige Eigenschaften der Wellenfunktion $\psi(x,t)$ direkt herleiten.

Wir wollen ein Teilchen im unendlich hohen Kasten betrachten. Weil im folgenden Symmetrieüberlegungen eine beträchtliche Rolle spielen, legen wir das Koordinatensystem so, daß sich diese Symmetrie einfach ausdrücken läßt. Das Potential $V(x)$ habe also die Form

$$V(x) = \begin{cases} 0 & \text{für} \quad -\frac{1}{2} \le x \le \frac{1}{2}\,, \\ \infty & \text{sonst}\,. \end{cases} \tag{4.66}$$

Dabei wird die Länge x in Einheiten der Breite a des Kastens und die Energie in Einheiten von $\hbar^2/2ma^2$ gemessen. Die Energien E_n der stationären Zustände $\psi_n(x)$ sind aus der Quantenmechanik-Vorlesung bekannt:

$$\begin{aligned} E_n &= n^2\,\pi^2 \quad \text{mit } n = 1,2,3,\dots\,, \\ \psi_n(x) &= \begin{cases} \sqrt{2}\,\cos(n\pi x) & \text{für } n \text{ ungerade} \\ \sqrt{2}\,\sin(n\pi x) & \text{für } n \text{ gerade} \end{cases} \quad , \; -\tfrac{1}{2} \le x \le \tfrac{1}{2}\,. \end{aligned} \tag{4.67}$$

Diese Zustände sind stehende Wellen, deren Betrag sich zeitlich nicht ändert. Der mittlere Ort des Teilchens in diesen Zuständen $\langle x \rangle = 0$ ändert sich ebenfalls nicht mit der Zeit.

Wir wollen nun ein Wellenpaket $\psi(x,t)$ betrachten, das im Kasten hin- und herläuft. $\psi(x,0)$ kann nach den Eigenzuständen entwickelt werden, so daß wir für $\psi(x,t)$ erhalten:

$$\begin{aligned} \psi(x,t) &= \sum_{m=1}^{\infty} \hat{c}_{2m-1}\, e^{-i\,\pi^2(2m-1)^2 t} \cos((2m-1)\pi x) + \\ &\quad \sum_{m=1}^{\infty} \hat{c}_{2m}\, e^{-i\,\pi^2(2m)^2 t} \sin(2m\pi x) \qquad (4.68) \\ &= \psi_{\mathrm{g}}(x,t) + \psi_{\mathrm{u}}(x,t) \qquad (4.69) \end{aligned}$$

mit Entwicklungskoeffizienten $\{\hat{c}_n\}$. Wegen der Symmetrie der Basisfunktionen (4.67) liefert diese Entwicklung zugleich die Zerlegung von $\psi(x,t)$ in den geraden (ψ_{g}) und ungeraden (ψ_{u}) Anteil.

Der allgemeine Zustand ist also eine unendliche Überlagerung aus stehenden Wellen, bei jeder Welle aber entwickelt sich der Phasenfaktor mit einer anderen Geschwindigkeit. Auf diese Weise werden komplexe Interferenzmuster erzeugt, die sich zeitlich sehr schnell ändern können (siehe Ergebnisse). Allerdings gibt es im Kasten eine Besonderheit, die mit der Struktur der Energien E_n zusammenhängt. Jede Frequenz $\omega_n = n^2\,\pi^2$ ist ein ganzzahliges Vielfaches der Grundfrequenz

$\omega_1 = \pi^2$. Daraus folgt, daß $\psi(x,t)$ periodisch in der Zeit t ist mit der Periodendauer $T = 2\pi/\omega_1 = 2/\pi$. Nach der Zeit $t = T$ verschwinden die wilden Interferenzmuster und das Wellenpaket $\psi(x,t)$ kehrt vollständig in seinen Ausgangszustand $\psi(x,0)$ zurück.

Wir werden im Ergebnisteil sehen, daß auch schon vorher zu bestimmten Zeiten einfache Muster auftauchen. So entsteht z. B. nach der Zeit $t = T/2$ die an der Mitte des Kastens gespiegelte Anfangsverteilung, die gerade in die entgegengesetzte Richtung läuft. Dies wollen wir kurz analytisch zeigen. Zur Zeit $t = T/2 = 1/\pi$ ergeben alle Phasenfaktoren in $\psi_{\mathrm{g}}(x,t)$ den Faktor -1, denn $\exp(-i\,\pi\,(2m-1)^2) = -1$, während die e-Faktoren in $\psi_{\mathrm{u}}(x,t)$ für $t = 1/\pi$ alle den Wert 1 haben. Also erhalten wir

$$\begin{aligned} \psi(x,\tfrac{T}{2}) &= -\psi_{\mathrm{g}}(x,0) + \psi_{\mathrm{u}}(x,0) \\ &= -\psi_{\mathrm{g}}(-x,0) - \psi_{\mathrm{u}}(-x,0) = -\psi(-x,0)\,. \end{aligned} \tag{4.70}$$

Nach der Zeit $T/2$ entsteht also die Aufenthaltswahrscheinlichkeit $|\psi(x,0)|^2$ zur Anfangszeit, nur an der Mittelachse des Kastens, $x = 0$, gespiegelt. Daß die Welle zu diesem Zeitpunkt in die entgegengesetzte Richtung läuft, erkennt man am besten an der Stromdichte

$$j(x,t) = i\,\left(\psi(x,t)\,\frac{\partial\psi^*}{\partial x}(x,t) - \psi^*(x,t)\,\frac{\partial\psi}{\partial x}(x,t)\right)\,. \tag{4.71}$$

Setzen wir hier das Ergebnis von Gleichung (4.70) ein, so ergibt sich unmittelbar

$$j(x,\tfrac{T}{2}) = -j(-x,0)\,. \tag{4.72}$$

Wir haben gezeigt, daß sich zu den Zeiten $T/2$ und T aus den komplexen Interferenzmustern die ursprüngliche Form zurückbildet. Der Ergebnisteil zeigt aber, daß sich auch schon früher einfache Wellen bilden. So entstehen z. B. bei $T/4$ zwei Wellenpakete, die gegeneinander laufen und später miteinander interferieren. Dies ist eine Folge der von uns gewählten Anfangsbedingung

$$\psi(x,0) = e^{ikx}\,\exp\left[-\frac{1}{2\sigma^2}\left(x+\frac{1}{4}\right)^2\right]\,. \tag{4.73}$$

ψ ist eine Gaußfunktion der Breite σ , die bei $x = -1/4$ konzentriert ist, multipliziert mit $\exp(ikx)$. Dieser Faktor bewirkt, daß das Teilchen am Anfang eine mittlere Geschwindigkeit

$$\langle v\rangle_0 = \langle\psi_{t=0}\,|\,\frac{p}{m}\,|\,\psi_{t=0}\rangle/\parallel\psi\parallel^2 = \frac{\hbar\,k}{m} = 2\,k \tag{4.74}$$

hat. Die Breite σ muß genügend klein sein, damit die Randbedingung $\psi(\pm\frac{1}{2}, 0) = 0$ noch mit ausreichender Genauigkeit als erfüllt angesehen werden kann. In der numerischen Simulation verwenden wir $\sigma = 0.05$, also $|\psi(\pm\frac{1}{2}, 0)| \leq \exp(-12.5) \lesssim 4 \cdot 10^{-6}$ im Vergleich zu $|\psi(-1/4, 0)| = 1$.

Wir setzen $t = T/4 = 1/(2\pi)$ in Gleichung (4.68) ein – die Phasenfaktoren ergeben jetzt $\exp(-i\pi^2(2m-1)^2/(2\pi)) = -i$ bzw. $\exp(-i\pi^2(2m)^2/(2\pi)) = 1$ – und erhalten

$$\psi(x, \tfrac{T}{4}) = -i\,\psi_{\rm g}(x, 0) + \psi_{\rm u}(x, 0)\,. \tag{4.75}$$

Das Betragsquadrat der Wellenfunktion für $t = T/4$ ist demnach

$$\begin{aligned} |\psi(x, \tfrac{T}{4})|^2 \;\; &= \;\; |\psi_{\rm g}(x, 0)|^2 + |\psi_{\rm u}(x, 0)|^2 + \\ & \quad\; i\, \left(\psi_{\rm g}^*(x, 0)\psi_{\rm u}(x, 0) - \psi_{\rm g}(x, 0)\psi_{\rm u}^*(x, 0)\right). \end{aligned} \tag{4.76}$$

Der letzte Summand dieser Gleichung, der sich als $2\,{\rm Im}(\psi_{\rm g}(x, 0)\,\psi_{\rm u}^*(x, 0))$ schreiben läßt, ist verschwindend klein. Eine einfache Rechnung unter Benutzung von (4.73) ergibt:

$$2\ {\rm Im}\,(\psi_{\rm g}(x, 0)\,\psi_{\rm u}^*(x, 0)) = -2\ \exp\left[-\frac{1}{2\sigma^2}\left(\frac{1}{8} + 2x^2\right)\right]\ \cos kx\ \sin kx\,, \tag{4.77}$$

einen Term also, der für $\sigma = 0.05$ von der Größenordnung $\exp(-25)$ ist. Der Rest läßt sich in folgender Weise durch $\psi(\pm x, 0)$ ausdrücken:

$$\begin{aligned} &|\psi_{\rm g}(x, 0)|^2 + |\psi_{\rm u}(x, 0)|^2 \\ &\qquad = \frac{1}{2}\left(|\psi_{\rm g}(x, 0) + \psi_{\rm u}(x, 0)|^2 + |\psi_{\rm g}(x, 0) - \psi_{\rm u}(x, 0)|^2\right) \\ &\qquad = \frac{1}{2}\left(|\psi_{\rm g}(x, 0) + \psi_{\rm u}(x, 0)|^2 + |\psi_{\rm g}(-x, 0) + \psi_{\rm u}(-x, 0)|^2\right) \\ &\qquad = \frac{1}{2}\left(|\psi(x, 0)|^2 + |\psi(-x, 0)|^2\right). \end{aligned}$$

Zusammengefaßt erhalten wir:

$$|\psi(x, \tfrac{T}{4})|^2 \cong \tfrac{1}{2}\left(|\psi(x, 0)|^2 + |\psi(-x, 0)|^2\right), \tag{4.78}$$

also gerade die symmetrisierte Anfangsverteilung. Eine entsprechende Rechnung für die Stromdichte ergibt

$$j(x, \tfrac{T}{4}) \cong \tfrac{1}{2}\,j(x, 0) - \tfrac{1}{2}\,j(-x, 0) \tag{4.79}$$

und zeigt uns, daß die beiden Wellenpakete (4.78) aufeinanderzulaufen.

Auch zu noch kürzeren Zeiten entstehen Wellen mit einer einfachen Struktur. Setzen wir $t = T\,p/q$ mit ganzen teilerfremden Zahlen p und q, dann kann die Phase $\exp(-2\pi i\; n^2 p/q)$ höchstens q verschiedene Werte annehmen. $\psi(x,t)$ ist daher nach (4.68) eine Überlagerung von q Wellen, wobei jede Welle eine unendliche Teilsumme aus $\psi(x,0)$ ist. Nimmt man an, daß jede Teilwelle eine glatte Funktion ist, so ist deren Überlagerung ebenfalls glatt, und $\psi(x,t)$ zeigt zu diesen Zeiten eine einfache Struktur. Genau dies werden wir bei der numerischen Integration beobachten.

Damit schließen wir die analytischen Betrachtungen ab und wenden uns der Beschreibung der numerischen Lösung der zeitabhängigen Schrödinger-Gleichung zu.

Algorithmus

Zunächst wird $\psi(x,t)$ diskretisiert:

$$\psi_n^j = \psi(j\,dx, n\,dt)\,. \tag{4.80}$$

j und n sind ganze Zahlen, dx und dt sind die Schrittweiten in Ort und Zeit. Wie vorher schreiben wir die zweite Ableitung mit einem Fehler der Größenordnung $\mathcal{O}(dx^2)$ als

$$\frac{\partial^2 \psi}{\partial x^2} \to \frac{1}{(dx)^2}\,\left(\psi_n^{j+1} - 2\psi_n^j + \psi_n^{j-1}\right). \tag{4.81}$$

Nun könnten wir die zeitliche Ableitung geeignet diskretisieren, analog zu den Solitonen aus dem vorigen Abschnitt. Allerdings erhält dieser Algorithmus nicht die Gesamtwahrscheinlichkeit $\int |\psi(x,t)|^2 dx = 1$, die sich zeitlich nicht ändern darf. Es ist in der Tat möglich, eine diskrete Näherung der partiellen Differentialgleichung zu finden, bei der die Gesamtwahrscheinlichkeit erhalten bleibt. Dazu muß der diskretisierte Zeitentwicklungsoperator unitär gewählt werden.

Die Lösung der Schrödinger-Gleichung (4.65) kann man auch in der Form

$$\psi(x,t+dt) = e^{-iHdt}\psi(x,t) \tag{4.82}$$

schreiben. Die Näherung

$$e^{-iHdt} \simeq (1 - iHdt) + \mathcal{O}(dt^2)\,, \tag{4.83}$$

die dem einfachen Euler-Schritt entspricht, ist nicht mehr unitär. Dagegen erhält man mit

$$e^{-iHdt} = \left(e^{iHdt/2}\right)^{-1} e^{-iHdt/2} \simeq \left(1 + iH\frac{dt}{2}\right)^{-1}\left(1 - iH\frac{dt}{2}\right) + \mathcal{O}(dt^3) \tag{4.84}$$

einen unitären Operator für die diskrete Zeitentwicklung:

$$\psi_{n+1}^j = \left(1 + \frac{i}{2} H\, dt\right)^{-1} \left(1 - \frac{i}{2} H\, dt\right) \psi_n^j \,. \tag{4.85}$$

Nun bleibt die Gesamtwahrscheinlichkeit $\sum_j |\psi_n^j|^2$ als Funktion der Zeit n konstant. Der inverse Operator kann auch auf die linke Seite der Gleichung gebracht werden,

$$\left(1 + \frac{i}{2} H\, dt\right) \psi_{n+1}^j = \left(1 - \frac{i}{2} H\, dt\right) \psi_n^j \,. \tag{4.86}$$

In dieser Form wird die Differenzengleichung implizit; das heißt, die Wellenfunktion im folgenden Zeitschritt ist durch ein Gleichungssystem bestimmt, das man erst lösen muß. Der entsprechende Algorithmus kostet zwar mehr Rechenzeit, führt aber fast immer zu einem stabilen numerischen Verfahren. Setzen wir nun die spezielle Form des Hamiltonoperators ein, so erhalten wir

$$(H\psi)_n^j = -\frac{1}{(dx)^2} \left(\psi_n^{j+1} - 2\psi_n^j + \psi_n^{j-1}\right) + V^j\, \psi_n^j \,. \tag{4.87}$$

Damit erhält man aus Gleichung (4.86)

$$\begin{aligned} \psi_{n+1}^{j+1} + \left(i\,\frac{2\,(dx)^2}{dt} - (dx)^2 V^j - 2\right) \psi_{n+1}^j + \psi_{n+1}^{j-1} &= \\ -\psi_n^{j+1} + \left(i\,\frac{2\,(dx)^2}{dt} + (dx)^2 V^j + 2\right) \psi_n^j - \psi_n^{j-1} &= \Omega_n^j \,. \end{aligned} \tag{4.88}$$

Die zweite Zeile, in der nur Größen des Zeitschritts n vorkommen, haben wir mit Ω_n^j abgekürzt. Diese Gleichung hat die Form

$$\mathsf{T}\, \boldsymbol{\psi}_{n+1} = \boldsymbol{\Omega}_n \,, \tag{4.89}$$

wobei die Matrix T Tridiagonalgestalt hat. Wir müssen also zu jedem Zeitschritt ein lineares Gleichungssystem lösen. Bei Tridiagonalmatrizen gibt es dafür ein spezielles Verfahren. Wir machen den Ansatz

$$\psi_{n+1}^{j+1} = a^j\, \psi_{n+1}^j + b_n^j \,. \tag{4.90}$$

Setzen wir dies in Gleichung (4.88) ein, so finden wir

$$\psi_{n+1}^j = \left[2 + (dx)^2 V^j - i\,\frac{2(dx)^2}{dt} - a^j\right]^{-1} \left[\psi_{n+1}^{j-1} + b_n^j - \Omega_n^j\right] . \tag{4.91}$$

Der Vergleich mit (4.90) ergibt Gleichungen für a^{j-1} und b_n^{j-1}, die wir nach a^j und b_n^j auflösen können. Das Ergebnis ist

$$\begin{aligned} a^j &= 2 + (dx)^2 V^j - i\frac{2(dx)^2}{dt} - \frac{1}{a^{j-1}} \,, \\ b_n^j &= \Omega_n^j + b_n^{j-1}/a^{j-1} \,. \end{aligned} \tag{4.92}$$

a^j ist also, wie vom Ansatz (4.90) gefordert, von n unabhängig, während b_n^j aus der Wellenfunktion ψ_n^j berechnet wird. Zur Lösung dieser Gleichungen benötigen wir den Anfangszustand ψ_0^j und Randbedingungen. Da wir das Teilchen in einen Kasten mit unendlich hohen Wänden sperren wollen, muß ψ am Rand verschwinden,

$$\psi_n^0 = 0 \quad \text{und} \quad \psi_n^J = 0\,, \tag{4.93}$$

wobei wir jetzt, da es sich einfacher programmieren läßt, den Rand durch $x = 0$ und $x = 1$ definiert haben. Die Anzahl J der Stützpunkte ist damit durch $J = 1 + 1/dx$ festgelegt. Gleichung (4.88) lautet bei $j = 1$:

$$\psi_{n+1}^2 + \left(i\,\frac{2\,(dx)^2}{dt} - (dx)^2\,V^1 - 2 \right) \psi_{n+1}^1 = \Omega_n^1\,. \tag{4.94}$$

Der Vergleich mit (4.90) ergibt:

$$a^1 = 2 + (dx)^2\,V^1 - i\,\frac{2\,(dx)^2}{dt}\;,\;\; b_n^1 = \Omega_n^1\,. \tag{4.95}$$

Mit diesen Startwerten und Gleichung (4.92) können alle a^j und nach jedem Zeitschritt alle b_n^j berechnet werden. Die Wellenfunktion berechnet man dann durch Rückwärts-Iteration aus Gleichung (4.90):

$$\psi_{n+1}^j = \left(\psi_{n+1}^{j+1} - b_n^j\right)/a^j\,. \tag{4.96}$$

Da der rechte Randwert $\psi_{n+1}^J = 0$ festliegt, kann so der gesamte Vektor ψ_{n+1}^j zur Zeit $n + 1$ berechnet werden.

Alle Ergebnisse werden nun zu einem Algorithmus zusammengefaßt:

1. Wähle einen Startzustand ψ_0^j und berechne damit Ω_0^j aus Gleichung (4.88).
2. Berechne den Vektor a^j mit dem Startwert aus Gleichung (4.95) und der Rekursion (4.92).
3. Berechne für alle Orte j die Variable b_n^j mit dem Startwert aus (4.95) und der Rekursion (4.92).
4. Berechne aus Gleichung (4.96) die Werte der Wellenfunktion ψ_{n+1}^j mit dem Startwert $\psi_{n+1}^J = 0$ für den nächsten Zeitschritt $n + 1$. Damit erhält man auch Ω_{n+1}^j aus Gleichung (4.88).
5. Iteriere die Schritte 3 und 4 für $n = 0, 1, 2, \ldots$

Als Anfangszustand wählen wir, wie schon erwähnt, ein Gaußsches Wellenpaket, das sich mit dem mittleren Impuls k bewegt:

$$\psi(x,0) = e^{i\,k\,x}\,\exp\left[-\frac{(x-x_0)^2}{2\,\sigma^2}\right]. \tag{4.97}$$

Die obigen Gleichungen können leicht in jeder Programmiersprache formuliert werden. Dabei muß allerdings beachtet werden, daß die Größen $\{\psi_n^j, a^j\}$ und $\{b_n^j\}$ komplexe Zahlen sind. Während *Mathematica* und auch andere Sprachen wie FORTRAN direkt mit solchen Zahlen rechnen können, muß man in C entweder die entsprechenden Routinen, z. B. aus den *Numerical Recipes*, hinzufügen oder selbst eine Struktur mit Real- und Imaginärteil definieren:

```
typedef struct{double real, imag; } complex;
```

Multiplikation und Division müssen damit selbst geschrieben werden. Die für komplexe Zahlen $a = \operatorname{Re} a + i \operatorname{Im} a$ und $b = \operatorname{Re} b + i \operatorname{Im} b$ bekannten Formeln der Multiplikation und Division

$$a \cdot b = \operatorname{Re} a\ \operatorname{Re} b - \operatorname{Im} a\ \operatorname{Im} b + i\,(\operatorname{Im} a\ \operatorname{Re} b + \operatorname{Re} a\ \operatorname{Im} b)\,, \tag{4.98}$$

$$\frac{a}{b} = \frac{\operatorname{Re} a\ \operatorname{Re} b + \operatorname{Im} a\ \operatorname{Im} b + i\,(\operatorname{Im} a\ \operatorname{Re} b - \operatorname{Re} a\ \operatorname{Im} b)}{(\operatorname{Re} b)^2 + (\operatorname{Im} b)^2} \tag{4.99}$$

finden sich zum Beispiel in der als Funktion `calculate_b` geschriebenen Iteration der Gleichung (4.92) wieder:

```
void calculate_b(complex *b, complex *omega, complex *a)
{
 int j ; double a2;

 b[1].real=omega[1].real;
 b[1].imag=omega[1].imag;
 for (j=2;j<J;j++)
  {a2=a[j-1].real*a[j-1].real+a[j-1].imag*a[j-1].imag;
   b[j].real=omega[j].real+
    (b[j-1].real*a[j-1].real+b[j-1].imag*a[j-1].imag)/a2;
   b[j].imag=omega[j].imag+
    (b[j-1].imag*a[j-1].real-b[j-1].real*a[j-1].imag)/a2;
  }
}
```

Alle anderen Schritte des obigen Algorithmus lassen sich ebenso einfach programmieren. Wir verweisen auf den Quellcode auf der beiliegenden Diskette.

Zur numerischen Integration der Schrödinger-Gleichung haben wir für den Kasten das Intervall $[0,1]$ gewählt, unter anderem auch, weil sich damit die x-Koordinate leichter in positive Bildkoordinaten umrechnen läßt. Die der Gleichung (4.68) entsprechende Eigenfunktionsentwicklung lautet dann:

$$\psi(x,t) = \sum_{n=1}^{\infty} c_n \, e^{-in^2\pi^2 t} \sin(n\pi x) \,, \tag{4.100}$$

$$c_n = 2 \int_0^1 \sin(n\pi x) \, \psi(x,0) \, dx \,. \tag{4.101}$$

Der Startzustand in unserer Simulation ist das gaußförmige Wellenpaket (4.97) mit der Breite $\sigma = 0.05$ und dem mittleren Impuls $k = 40$, das bei $x_0 = 0.25$ zentriert ist. Wie wir schon gesehen haben, muß die Welle nach der Zeit $T = 2/\pi \simeq 0.6366$ in ihren Anfangszustand zurückkehren. Dies gibt uns einen Test für die Genauigkeit, mit der wir die Schrödinger-Gleichung integriert haben.

T ist natürlich auch die größte Zeitskala, die für dieses Problem relevant ist. Zusätzlich aber gibt es drei weitere charakteristische Zeiten, die hier eine Rolle spielen. Da ist zum einen die Zeit t_1, die das Teilchen benötigt, um aufgrund seines mittleren Impulses den Kasten einmal zu durchlaufen. Mit $k = 40$ erhalten wir aus Gleichung (4.74) $t_1 = 1/80 = 0.0125$. Darüber hinaus müssen wir den typisch quantenmechanischen Effekt des Zerfließens eines Wellenpaketes berücksichtigen. Eine freie gaußförmige Anfangsverteilung mit der Breite σ hat zur Zeit t die Breite $\sigma \sqrt{1 + t^2 \hbar^2/(4 m^2 \sigma^4)}$. Wenn wir fragen, wann diese Breite mit der Kastenbreite übereinstimmt, so erhalten wir daraus die Zeit $t_2 \cong 1/20 = 0.05$.

Die vierte Zeitskala schließlich, die uns auch einen Anhaltspunkt für die Wahl der Schrittweiten dx und dt gibt, ist durch die größte relevante Phasengeschwindigkeit festgelegt, zu deren Bestimmung wir die Gleichungen (4.100) und (4.101) heranziehen. Drücken wir den Sinus in diesen Gleichungen durch die Exponentialfunktion aus, so sind die Summanden in (4.100) von der Form

$$c_n \, e^{-i\,n\,\pi(n\,\pi\,t \,\pm\, x)} \,, \tag{4.102}$$

so daß die Phasengeschwindigkeit v_n^{ph} für diese Teilwelle den Wert

$$v_n^{\mathrm{ph}} = \pm n\,\pi \tag{4.103}$$

hat. Um abzuschätzen, bis zu welchem n die Koeffizienten c_n nennenswerte Beiträge liefern, erinnern wir daran, daß die Fouriertransformierte einer Gaußfunktion wiederum eine Gaußfunktion ist, allerdings mit reziproker Breite. Der Faktor $\exp(i\,k\,x)$ bedeutet im Fourierraum eine Verschiebung um k. Daraus folgt für die n-Abhängigkeit von c_n:

$$|c_n| \propto e^{-\frac{1}{2}\sigma^2(n\,\pi - k)^2} \,. \tag{4.104}$$

Auf dem dem Bildschirm können wir nur solche Amplituden sichtbar machen, die mindestens $1/1000$ der maximalen Amplitude betragen. Dies führt uns zur Abschätzung

$$\frac{1}{2}\sigma^2(n\,\pi - k)^2 \leq \ln(1000) \quad \Rightarrow \quad n_{\max} \simeq 35\,, \; v_{\max}^{\mathrm{ph}} \simeq 100\,. \tag{4.105}$$

Wenn auf den kürzesten Sinusbogen noch etwa 20 bis 30 Teilpunkte entfallen sollen, so ergibt diese Überlegung

$$dx \lesssim 10^{-3} \;\text{ und }\; dt \lesssim 10^{-5}\,, \tag{4.106}$$

weil dt jedenfalls nicht größer als $dx/v_{\max}^{\mathrm{ph}}$ sein sollte.

Will man auch den globalen Aspekt berücksichtigen, daß die Integration zumindest bis zur Dauer einer Periode korrekt sein soll, so läßt sich die Schrittweite dt durch folgende Überlegung genauer festlegen. Der von uns benutzte numerische Algorithmus garantiert zwar, daß die Norm der Wellenfunktion erhalten bleibt, stellt aber trotzdem eine Näherung dar. Während nämlich die exakte Zeitentwicklung der Eigenmode $\psi_n(x)$ nach ℓ Zeitschritten durch den Faktor $\exp(-i\,n^2\pi^2\,\ell\,dt)$ gegeben ist, berechnen wir sie numerisch gemäß (4.85) als

$$\left(\frac{1 - i\,\frac{dt}{2}n^2\pi^2}{1 + i\,\frac{dt}{2}n^2\pi^2}\right)^{\ell} = e^{-i\,\ell\,d\varphi} \;\text{ mit }\; d\varphi = 2\,\arctan\left(\frac{dt}{2}n^2\pi^2\right). \tag{4.107}$$

Für den Betrag der Differenz dieser beiden Ausdrücke liefert die Taylor-Entwicklung von $d\varphi$

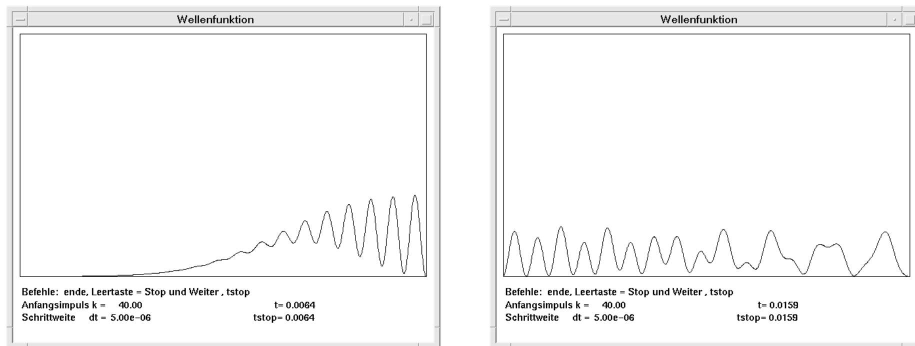
$$\Delta \simeq \frac{1}{12}\,\ell(dt\,n^2\pi^2)^3\,. \tag{4.108}$$

Fordern wir $\Delta < 1/2$ für $\ell = T/dt$ und $n = n_{\max}$, so folgt hieraus $dt \lesssim 2 \cdot 10^{-6}$. Es stellt sich heraus, daß unser Algorithmus in der Tat mit $dx = 10^{-3}$ und $dt = 5 \cdot 10^{-6}$ passable Ergebnisse produziert, die allerdings mit wachsender Integrationszeit auch Abweichungen von der exakten Zeitentwicklung aufweisen.

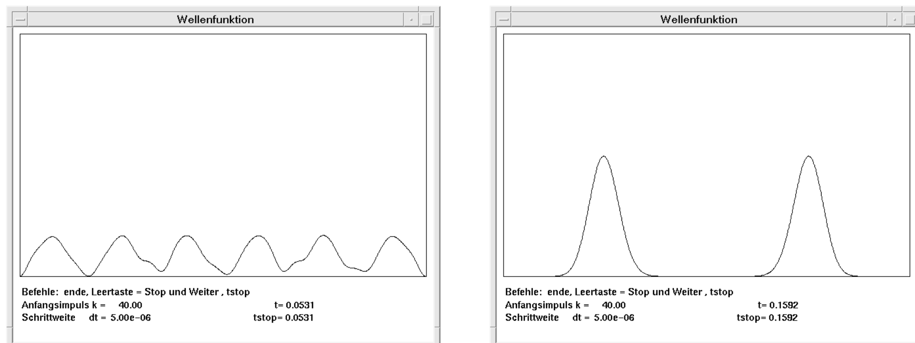
Ergebnisse

Am Anfang merkt die Welle noch nichts vom Kasten. Sie bewegt sich nach rechts und zerfließt dabei. Sobald aber ein Teil der Welle von der Wand reflektiert wird, interferiert er mit dem noch einlaufendem Teil zu einem Wellenmuster (Bild 4.13, links). Auf der rechten Seite dieser Abbildung sieht man, daß sich schon nach kurzer Zeit ($t = T/40$) ein unregelmäßiges Interferenzmuster über den ganzen Kasten ausgebildet hat. Dieses Muster ist typisch für fast alle Zeiten.

Doch plötzlich entstehen aus den scheinbar chaotischen Bewegungen regelmäßige Figuren, wie in Bild 4.14 und 4.15 zu den Zeiten $T/12$, $T/4$, $T/3$ und $T/2$ zu sehen ist. Wie schon gezeigt wurde, bilden sich zur Zeit $t = T/4$ zwei Wellenpakete, die gegeneinander laufen und dann miteinander interferieren (Bild 4.14, rechts), und bei $t = T/2$ entsteht plötzlich wieder die ursprüngliche Welle, allerdings an der Mitte des Kastens gespiegelt und nach links laufend (Bild 4.15, rechts). Diese einfachen Muster zerfließen und interferieren aber schnell wieder zu wilden Bewegungen.

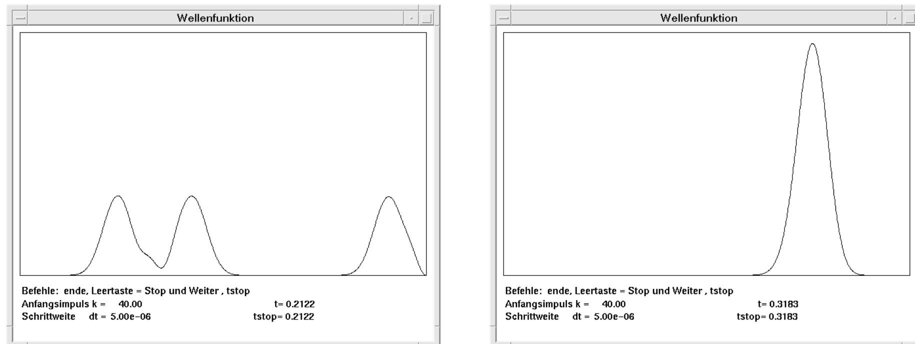


4.13 Links: Wellenpaket $|\psi(x,t)|^2$ kurz nach dem ersten Aufprall auf die Wand ($t = T/100$). Rechts: Typisches Interferenzmuster, $|\psi(x,t)|^2$ für $t = T/40$.



4.14 $|\psi(x,t)|^2$ zu den Zeiten $t = T/12$ (links) und $t = T/4$ (rechts).

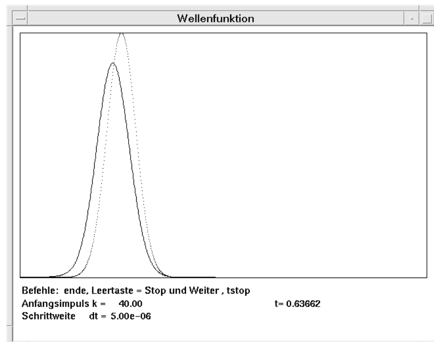
Die Einzelbilder zu verschiedenen Zeiten werden im Titelbild dieses Lehrbuches als Gebirge über der Ort-Zeit-Ebene dargestellt. Die Ortskoordinate wurde horizontal von rechts nach links und die Zeit von hinten nach vorn aufgetragen. Ganz hinten sieht man, wie der Startzustand nach links läuft, dabei zunächst zerfließt und dann



4.15 Wellenpaket $|\psi(x,t)|^2$ zu den Zeiten $t = T/3$ (links) und $t = T/2$ (rechts).

an der Wand reflektiert wird. Ganz vorn ist die Aufenthaltswahrscheinlichkeit kurz vor der Zeit $T/6$ zu sehen; eine glatte Welle mit drei Bergen. Dazwischen entsteht eine faszinierende Hügellandschaft von unerwarteter Vielfalt und Regelmäßigkeit. Insbesondere die vielen Täler, die sich sternförmig ausbreiten, sind uns bei der Beobachtung der Welle direkt auf dem Bildschirm nicht aufgefallen. Wir haben bis jetzt diese Täler noch nicht analytisch berechnen können.

Bild 4.16 schließlich zeigt sowohl die Anfangsverteilung $|\psi(x,0)|^2$ (punktiert) als auch das numerisch integrierte Wellenpaket nach der vollen Periode T. Wollten



4.16 Die Anfangsverteilung $|\psi(x,0)|^2$ (punktiert) und das numerisch integrierte Wellenpaket $|\psi(x,T)|^2$ nach der vollen Periode T.

wir die offensichtliche Diskrepanz zwischen numerischer Lösung und exakter Zeitentwicklung beseitigen, so müßten wir die Diskretisierung der Ortskoordinate noch weiter verfeinern. Bei unseren Überlegungen zur Periodendauer sind wir bisher nämlich von den Energiewerten $E_n = n^2\pi^2$ des exakten Hamiltonoperators H ausgegangen, wohingegen für die numerische Integration die Matrixversion von H, Gleichung (4.87) mit $V^j = 0$, relevant ist. Die Eigenwerte E_n^{d} dieses Operators und die zugehörigen Eigenvektoren ϕ_n^j, die natürlich die Randbedingungen (4.93)

erfüllen müssen, sind aber auch bekannt:

$$\phi_n^j = \sin(n\,\pi\,j\,dx)\,, \quad E_n^{\mathrm{d}} = \frac{4}{(dx)^2}\sin^2\left(\frac{n\pi}{2}dx\right). \tag{4.109}$$

n ist hier der Index der Eigenmode und nicht die Nummer des Zeitschrittes. Der obigen Gleichung und der Entwicklung von E_n^{d} für $dx \ll 1$,

$$E_n^{\mathrm{d}} \simeq n^2\pi^2 - \frac{1}{12}n^4\pi^4\,(dx)^2\,, \tag{4.110}$$

entnehmen wir erstens, daß die Frequenzen nicht mehr exakt ganzzahlige Vielfache einer Grundfrequenz sind, und zweitens, daß sie alle kleiner sind als die vorher betrachteten E_n. Hierdurch erklärt sich sowohl die Verbreiterung als auch die zeitliche Verzögerung des Wellenpaketes in Bild 4.16. Die Berücksichtigung der modifizierten Energiewerte E_n^{d} (4.109) in Gleichung (4.107) ergibt $dx \lesssim 10^{-4}$ und $dt \lesssim 10^{-6}$, wenn diese Korrektur nach einer Periode noch nicht sichtbar sein soll.

Übung

a) Verwenden Sie die beschriebene Integrationsroutine, um die Mittelwerte $\langle x\rangle$, $\langle p\rangle$ und deren Streuungen Δx, Δp in Abhängigkeit von der Zeit zu berechnen und sichtbar zu machen.

b) Der obige Programmcode läßt sich leicht so abwandeln, daß damit der Tunneleffekt demonstriert werden kann. Ändern Sie das Potential innerhalb des Kastens, indem Sie in die Mitte des Kastens eine gaußförmige Barriere der Form

$$V_0\, e^{-(x-\frac{1}{2})^2/(2d^2)}$$

setzen. Versuchen Sie, ein numerisches Kriterium für die Tunnelzeit t_{T} zu finden. Zum Beispiel ist die Wahrscheinlichkeit w_n, das Teilchen zur Zeit $t_n = n\,dt$ rechts von der Barriere zu finden, durch

$$w_n = \frac{\sum_{j=J/2}^{J} |\psi_n^j|^2}{\sum_{j=1}^{J} |\psi_0^j|^2}$$

berechenbar. Bestimmen Sie numerisch die Abhängigkeit der Tunnelzeit von V_0 und der Barrierenbreite d.

Literatur

S. Brandt, H. D. Dahmen, *Quantum Mechanics on the Personal Computer*, Springer Verlag, 1992.

S. E. Koonin, D. C. Meredith, *Physik auf dem Computer, Band 1+2*, R. Oldenbourg Verlag, 1990.

Kapitel 5
Monte-Carlo-Simulationen

Monte Carlo → Spielbank → Roulette → Zufallszahlen: Das sind die Assoziationen, die einer wichtigen Methode der Computersimulation den Namen gaben. Mit Hilfe von Zufallszahlen kann man mit dem Computer z. B. die Bewegung eines wechselwirkenden Vielteilchensystems, das sich im Wärmebad befindet, simulieren. Wie im realen Experiment lassen sich Temperatur und andere Parameter variieren. Die Modellmaterialien können aufgeheizt oder abgekühlt werden, dabei kann man bei hinreichend tiefen Temperaturen beobachten, wie Gase flüssig werden, wie sich magnetische Atome ordnen oder wie Metalle ihren elektrischen Widerstand verlieren.

Wir wollen hier an einfachen Beispielen studieren, wie sich mit Hilfe von Zufallszahlen interessante physikalische Phänomene beschreiben lassen. Einige dieser Modelle haben sogar universelle Eigenschaften: Die Werte der kritischen Exponenten, die die Singularitäten an Phasenübergängen beschreiben, gelten für viele verschiedene Modelle und werden sogar an realen Materialien gemessen. Daher ist die Computersimulation besonders wichtig für das Verständnis der kooperativen Eigenschaften wechselwirkender Teilchen.

5.1 Zufallszahlen

Ein Computer kann keine Zufallszahlen erzeugen. Er arbeitet nach einem wohldefinierten Programm, nach Regeln also, die auf Eingabedaten angewendet werden und Ausgabedaten erzeugen. Ein Computer funktioniert daher wie eine deterministische Abbildung, die dem Zufall keinen Raum läßt. Dennoch gibt es Algorithmen, die „Pseudozufallszahlen“ erzeugen. Bei vielen statistischen Tests führt eine solche Folge von Zahlen zu ähnlichen Ergebnissen wie sie Zufallszahlen erzeugen würden, die die mathematische Definition von „zufällig“ erfüllen. Wir wollen hier solche Algorithmen kurz vorstellen.

Algorithmus und Ergebnis

Im Rechner werden Zahlen durch eine Anzahl von Bits (0 oder 1) dargestellt. Stehen z. B. pro Zahl 32 Bits zur Verfügung, so können damit maximal 2^{32} verschiedene

Zahlen dargestellt werden. Eine Abbildung f auf diesen Zahlen,

$$r_n = f(r_{n-1}) \,, \tag{5.1}$$

erzeugt demnach eine Folge $r_0, r_1, r_2, \dots$, die sich nach höchstens 2^{32} Schritten wiederholen muß. Ein Computer kann daher nur periodische Sequenzen von Zahlen erzeugen. Bei maximaler Periodenlänge kommt jede Zahl in einer sehr langen Folge gleich oft vor, diese Zahlen sind demnach gleichverteilt. Wenn die computererzeugte Zahlenreihe viele Tests auf Zufälligkeit besteht, so nennt man sie Zufallszahlen.

Eine in diesem Zusammenhang häufig verwendete Abbildung $f(r)$ ist die Modulo-Funktion mit drei Parametern a, c und m:

$$r_n = (a\, r_{n-1} + c) \bmod m \,. \tag{5.2}$$

Ruft man die vom System bereitgestellten Zufallszahlen-Generatoren auf, z. B. `rand()` in C oder `Random[]` in *Mathematica*, so werden fast immer diese sogenannten linear kongruenten Generatoren benutzt. Das Ergebnis der Modulo-Funktion (= Rest bei Division durch m) ist auf höchstens m verschiedene Werte beschränkt; daher hat die maximale Periode die Länge m. Für manche Parameter a und c wird dieses Maximum m auch erreicht. Einige Generatoren mit solchen Parametern haben bei den Tests auf Zufälligkeit relativ gut abgeschnitten. In den *Numerical Recipes* sind einige Werte der günstigen Parameter (m, a, c) angegeben, z. B. $m = 6075, a = 106$ und $c = 1283$. Mit diesem Generator haben wir Punkte auf der dreidimensionalen Kugeloberfläche aus je drei aufeinanderfolgenden Zufallszahlen erzeugt,

$$\boldsymbol{e}_n = (r_n, r_{n+1}, r_{n+2}) / \sqrt{r_n^2 + r_{n+1}^2 + r_{n+2}^2} \,. \tag{5.3}$$

Jeder Generator benötigt einen Startwert r_0, den wir hier auf $r_0 = 1234$ gesetzt haben. Ein kurzes *Mathematica*-Programm

```
         r0 = 1234
          a = 106
          c = 1283
          m = 6075
  zufall[r_] = Mod[a r+c,m]
     uniform = NestList[zufall,r0,m+1]/N[m]
```

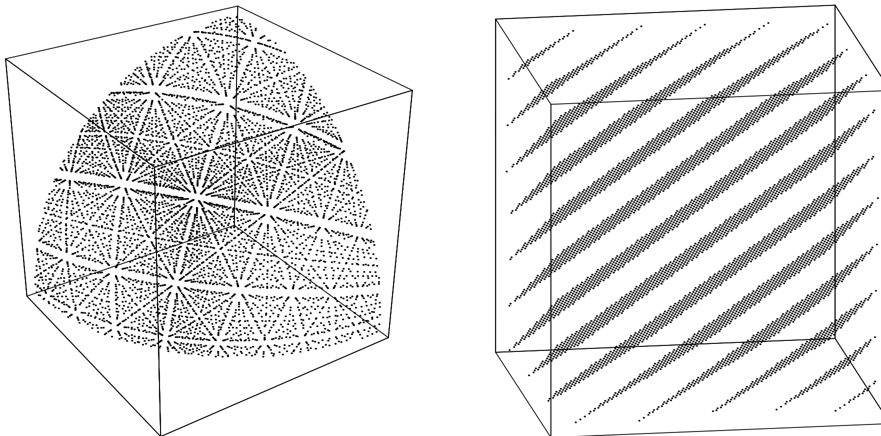
verschafft uns zunächst sämtliche im Einheitsintervall liegenden Zufallszahlen, aus denen wir mit

```
      tripel = Table[Take[uniform,{n,n+2}],{n,m}]
 unitvectors = Map[(#/Sqrt[#.#])&,tripel]
```

die in Gleichung (5.3) beschriebenen Einheitsvektoren $\boldsymbol{e}_n$ gewinnen. Der folgende Befehl macht daraus Graphikobjekte und stellt sie auf dem Bildschirm dar:

```
Show[Graphics3D[Map[Point,unitvectors]],
     ViewPoint -> {2,3,2}]
```

Der Vektor `{2,3,2}` weist in die Richtung des Beobachters. Bild 5.1 (links) zeigt das Ergebnis. Wären die Punkte $\boldsymbol{e}_n$ wirklich zufällig verteilt, so müßte bis auf



5.1 Tripel aus Pseudozufallszahlen. Links: Projektion auf die Kugeloberfläche. Rechts: Dieselben Punkte im Einheitswürfel bei geeigneter Blickrichtung.

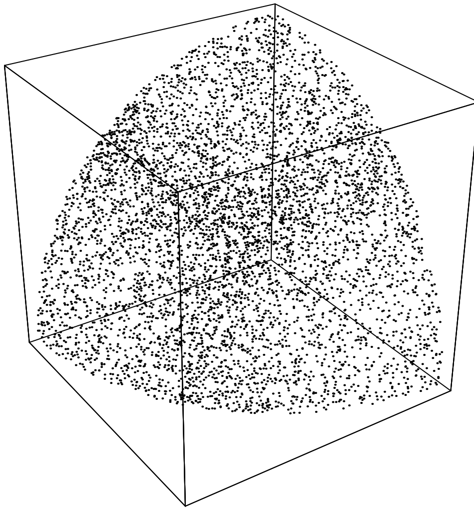
einen Geometriefaktor, der von der Projektion des Würfels auf die Kugel herrührt, dieser Ausschnitt der Oberfläche gleichmäßig mit Punkten überdeckt sein. Man sieht aber deutliche Strukturen, die aus der Korrelation zwischen drei aufeinanderfolgenden Zahlen r_n entstehen. Tatsächlich läßt sich zeigen, daß die Punkte $\boldsymbol{u}_n = (r_n, r_{n+1}, r_{n+2})/m$ wie die Bausteine eines Kristalls auf einem regelmäßigen Gitter liegen. Ausgehend von einem Gitterpunkt kann man die nächstgelegenen Punkte aufsuchen und auf diese Weise die primitiven Gittervektoren bestimmen. Blickt man aus genügend großer Entfernung und geeigneter Richtung auf die Zufallspunkte, so wird diese Gitterstruktur sichtbar. Bild 5.1 (rechts) ist das Resultat von

```
Show[Graphics3D[Map[Point, tripel]],
     ViewPoint -> {-325., 2000., -525.}]
```

Es ist klar zu erkennen, daß insgesamt 14 Ebenen ausreichen, um alle Punkte unterzubringen. Dieser Wert ist noch als relativ gut anzusehen, denn er ist zu vergleichen

mit der völlig regelmäßigen kubischen Struktur, die auf $m^{1/3} = 6075^{1/3} \simeq 18$ Ebenen führt. Bei einer Periode von $m = 6075$ können die Vektoren also nicht gleichförmiger als auf 18 Ebenen verteilt sein. Eine schlechte Wahl von a und c hat zur Folge, daß weniger Ebenen ausreichen, um alle Punkte aufzunehmen.

In Abbildung 5.2 sind die Zufallszahlen in derselben Art aufgetragen wie in Bild 5.1 (links), nur wurden sie diesmal mit dem *Mathematica*-Generator `Random[]`



5.2 Wie Bild 5.1 (links), aber die Vektoren wurden mit dem *Mathematica*-Generator erzeugt.

erzeugt. Jetzt erkennt man keine Struktur mehr auf der Kugeloberfläche. Das bedeutet aber nicht, daß es keine anderen Korrelationen zwischen den Zufallszahlen gibt. Bei geeigneter Auftragung könnten auch höhere Korrelationen oder Abhängigkeiten zwischen weit voneinander entfernten Zahlen dem Auge sichtbar werden.

Es gibt verschiedene Tricks, um möglichst große Perioden und gute Zufallszahlen zu erhalten. Man kann mehrere verschiedene Generatoren ineinander verschachteln, oder man kann andere Abbildungen f benutzen. Ebenso kann man sich Sequenzen von einzelnen Bits erzeugen, indem man Bits in einem festem Abstand miteinander logisch verknüpft. Nach dieser Vorschrift wird ein neues Bit erzeugt, und danach wird das Verknüpfungsfenster um eine Stelle verschoben. Es ist aber bekannt, daß auch solche Schieberegister-Generatoren Korrelationen in den Bitsequenzen liefern.

Wie kann man überhaupt mit einfachen Abbildungen aus m verschiedenen Zahlen Perioden erzeugen, die länger sind als m? Um dies zu erreichen, kann man Gleichung (5.1) z. B. dadurch erweitern, daß r_n nicht nur aus einer, sondern aus zwei oder mehreren schon erzeugten Zahlen r_j berechnet wird:

$$r_n = f\left(r_{n-t}, r_{n-s}\right) . \tag{5.4}$$

Dabei sind t und s geeignet gewählte natürliche Zahlen mit $t > s$. Jetzt wiederholt sich die Folge erst, wenn alle Zahlen $\{r_{n-t}, r_{n-t+1} \ldots, r_{n-1}\}$ schon einmal vorgekommen sind. Benutzt man ganze Zahlen mit 32 Bits, so kann der Generator daher maximal $2^{32\,t}$ Zahlen erzeugen. Auf diese Weise lassen sich also Zahlenfolgen mit riesigen Perioden erzeugen. Schon für $t = 2$ kann ein Programm, das pro Sekunde eine Million Zufallszahlen erzeugt, 584942 Jahre laufen, bevor sich die Zahlenfolge wiederholt.

Natürlich muß man in der Iteration (5.4) noch eine Funktion f finden, die auch wirklich die maximale Periode $2^{32\,t}$ erzeugt. Vor etwa 10 Jahren haben Marsaglia und Zaman eine Klasse von einfachen Funktionen gefunden, die fast die maximale Periode liefern. Sie nennen diese Funktionen *subtract-with-borrow generators*:

$$r_n = (r_{n-s} - r_{n-t} - c) \bmod m \,. \tag{5.5}$$

Dabei ist c ein Bit, das für den folgenden Schritt auf $c = 1$ oder $c = 0$ gesetzt wird, je nachdem, ob $r_{n-s} - r_{n-t}$ negativ oder positiv ist. Mit zahlentheoretischen Argumenten geben die Autoren Werte von s, t und m an, bei denen die Periodenlänge berechnet werden kann. So hat der Generator

$$r_n = (r_{n-2} - r_{n-3} - c) \bmod (2^{32} - 18) \tag{5.6}$$

die Periodenlänge $(m^3 - m^2)/3 \simeq 2^{95}$. Dieser Generator eignet sich außerdem hervorragend für die Sprache C. In modernen Rechnern ist die Operation *Modulo* 2^{32} automatisch bei der Arithmetik von ganzen Zahlen mit 32 Bit Länge eingebaut. Da C vorzeichenlose Integerzahlen erlaubt (unsigned long int), kann man Gleichung (5.6) sehr einfach programmieren.

Marsaglia und Zaman kombinieren diesen Generator mit

$$r_n = (69069\, r_{n-1} + 1013904243) \bmod 2^{32} \,, \tag{5.7}$$

der die Periode 2^{32} hat. Die Gleichungen (5.6) und (5.7) können leicht in der C-Sprache formuliert werden.

```
#define N 1000000
typedef unsigned long int unlong;
unlong x=521288629, y=362436069, z=16163801,
      c=1, n=1131199209;

unlong mzran()
{ unlong s;
    if(y>x+c) {s=y-(x+c); c=0;}
          else      {s=y-(x+c)-18; c=1;}
    x=y; y=z; z=s; n=69069*n+1013904243;
```

```
     return (z+n);
  }

main()
{
      double r=0.;
      long i;
      for (i=0;i<N; i++)
         r+=mzran()/4294967296.;
      printf ("r= %lf \n",r/(double)N);
}
```

x, y, z und n werden auf Anfangswerte gesetzt, die der Benutzer beliebig ändern kann. Das Hilfsbit c muß auf c = y > z gesetzt werden, in C also auf 1 (wahr) oder 0 (falsch). Da dieses Programm für einen Rechner geschrieben ist, der die Modulo-2^{32}-Operation automatisch durchführt, muß bei mod $(2^{32} - 18)$ der Wert 18 noch von der Differenz $(r_{n-2} - r_{n-3} - c)$ abgezogen werden. Das geschieht natürlich nur, wenn diese Differenz negativ ist, denn sonst kommt die Modulo-Operation nicht vor. Der nächste r_n-Wert ist die Summe der beiden Generatoren (5.6) und (5.7).

Weil diese beiden Generatoren teilerfremde Perioden haben, ist die Länge des zusammengesetzten Generators das Produkt der beiden einzelnen Längen, also etwa 2^{127}. Dieser Generator kann etwa 10^{24} Jahre laufen, bevor sich die Zufallszahlen wiederholen. Da die Zahlenfolge außerdem eine Menge statistischer Tests bestanden hat, kann man bei vielen Anwendungen davon ausgehen, daß sich diese Zahlen so verhalten wie „echte" Zufallszahlen.

Dennoch müssen wir den Leser warnen: auch die obigen Pseudozufallszahlen entstehen aus einem deterministischen Algorithmus, der sogar zahlentheoretisch analysiert werden kann. Man kann also nicht ausschließen, daß Korrelationen bei statistischen Anwendungen stören. Jedes Problem reagiert anders auf unterschiedliche Korrelationen. Daher können die Experten auch nach einigen Jahrzehnten Erfahrung mit Zufallszahlen nur den Rat geben, verschiedene Generatoren zu benutzen und die Ergebnisse zu vergleichen. Wenn möglich, sollte man sich ein ähnliches Problem suchen, das exakt gelöst werden kann und die numerischen Ergebnisse mit den exakt berechneten vergleichen. Dann hat man eine gewisse Sicherheit über die Qualität der Zufallszahlen. Da die zukünftigen Computersimulationen immer umfangreicher und genauer werden, darf die Qualität der Zufallszahlen nicht vernachlässigt werden.

Die Beispiele in unserem Lehrbuch dienen nur zur Demonstration und nicht zur genauen quantitativen Analyse. Deshalb werden wir im folgenden nur die in den Programmiersprachen vorhandenen Generatoren rand() bzw. Random[] benutzen.

Übung

a) Programmieren Sie die beiden Zufallszahlengeneratoren

$$\begin{aligned}(I) \quad r_n &= (r_{n-2} - r_{n-5}) \bmod 10\,, \\ (II) \quad r_n &= (r_{n-2} - r_{n-5} - c) \bmod 10\,,\end{aligned}$$

wobei $c = 1$ oder $c = 0$ gesetzt wird, falls $(r_{n-2} - r_{n-5}) < 0$ bzw. > 0 ist. Beide Generatoren erzeugen ganze Zahlen $0, 1, \ldots, 9$ und wiederholen die Zahlenfolge, wenn die letzten fünf Zahlen schon einmal vorgekommen sind. Beide müssen daher nach maximal 10^5 Schritten die erzeugte Zahlenfolge wiederholen.

Starten Sie mit verschiedenen Sätzen von fünf Anfangsziffern und berechnen Sie die tatsächliche Periodenlänge der erzeugten Zahlenfolgen.

Tip: Die Periodenlänge der Generatoren läßt sich z. B. dadurch berechnen, daß man zwei Folgen parallel erzeugt. Bei der zweiten Folge erzeugt man jeweils zwei Schritte und beobachtet dann, wann die zweite die erste Folge überholt.

b) Erzeugen Sie sich die Summe s von N gleichverteilten Zufallszahlen r aus dem Einheitsintervall. Der Mittelwert der Summe ist dann $m = 0.5N$ und die mittlere quadratische Abweichung, die Varianz, ist

$$\sigma^2 = N\left(\langle r^2\rangle - \langle r\rangle^2\right) = N\left[\int_0^1 r^2 dr - \left(\int_0^1 r\,dr\right)^2\right] = \frac{N}{12}\,.$$

Wiederholen Sie das Experiment sehr oft und zeichnen Sie die Häufigkeit der Summen in ein Histogramm. Vergleichen Sie das Ergebnis mit einer Gaußverteilung mit Mittelwert m und Varianz σ^2,

$$P(s) = \frac{1}{\sqrt{2\pi}\,\sigma}\,\exp\left[\frac{-(s-m)^2}{2\,\sigma^2}\right].$$

Überprüfen Sie die Übereinstimmung für $N = 10, 100$ und 1000.

Literatur

K. Binder, D. W. Heermann, *Monte Carlo Simulation in Statistical Physics*, Springer Verlag, 1992.

H. Gould, J. Tobochnik, *An Introduction to Computer Simulation Methods: Applications to Physical Systems, Parts I and II*, Addison Wesley, 1988.

D. E. Knuth, *The Art of Computer Programming*, Vols. I, II, and III, Addison Wesley, 1973.

G. Marsaglia, A. Zaman, *Some portable very-long period random number generators*, Computer in Physics **8**, 117 (1994).

W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge University Press, 1992.

5.2 Fraktale Aggregate

Im Abschnitt 3.3 haben wir Gebilde konstruiert, die mehr als eine Linie aber weniger als eine Fläche sind. Sie werden durch eine gebrochene Dimension charakterisiert und haben den Namen Fraktale. Sie sind selbstähnlich, ein Teil sieht ähnlich aus wie das ganze Gebilde, und sie können durch eine einfache deterministische Regel erzeugt werden.

Gibt es solche Gebilde in der Natur? In der Tat lassen sich viele Strukturen in unserer Umgebung mit Hilfe von fraktalen Dimensionen quantitativ charakterisieren. Küstenlinien, Gebirgszüge, Flußläufe, Blutgefäße, Nervenzellen, Blattoberflächen, Schwankungen von Börsenkursen und viele andere Phänomene lassen sich durch ein Potenzgesetz der Art „die Masse M wächst wie die Länge L zur Potenz D“ beschreiben. Allerdings sind natürliche Strukturen nicht so streng regelmäßig wie die Sierpinski-Packung aus Abschnitt 3.3, sondern sie sind auch durch zufällige Mechanismen entstanden.

Ein einfaches Beispiel für solche durch Zufall und Regel entstandenen Fraktale sind Aggregate. Wenn Teilchen an einen Kern herandiffundieren und dort haften bleiben, so entsteht eine lockere, körnige Struktur mit einer gebrochenen Dimension zwischen 2 und 3. Mit einem einfachen Computermodell, das 1981 von Witten und Sander vorgeschlagen und untersucht wurde, wollen wir demonstrieren, wie mit wenig Aufwand solche fraktalen Gebilde mit dem Computer erzeugt werden können.

Physik

Wir beschreiben im folgenden ein Modell eines Wachstumsprozesses, bei dem die ungerichtete Diffusion von Teilchen, bevor sie angelagert werden, die entscheidende Rolle spielt. Das Abscheiden von Material aus einer Elektrolytlösung bei sehr geringer Spannung zwischen den Elektroden kann als Beispiel hierfür dienen. Man muß nur dafür sorgen, daß die Zufallsbewegung der diffundierenden Ionen wichtiger ist als ihre Drift aufgrund des elektrischen Feldes. Die Materialablagerung an

der Elektrode führt dann nicht zu einem kompakten Gebilde, sondern es entsteht ein filigranartig verästelter Cluster, den wir wieder durch eine fraktale Dimension D beschreiben können. D ist etwa $5/6$ mal so groß wie die Raumdimension.

Den Prozeß nennt man *Diffusion Limited Aggregation (DLA)*. Obwohl DLA einfach auf dem Computer simuliert und analysiert werden kann, gibt es unseres Wissens bis heute keine analytische Theorie dazu. Dagegen existiert für das Diffusionsproblem bzw. die Zufallsbewegung eines Teilchens eine gut entwickelte mathematische Theorie. Einige Aussagen und Ergebnisse, auf die wir im Algorithmus-Teil zurückkommen, wollen wir hier kurz erläutern.

Wir betrachten einen *random walk* auf einem quadratischen, oder auch allgemeiner, auf einem d-dimensionalen kubischen Gitter. Die Gittervektoren bezeichnen wir mit $\boldsymbol{x}$, und die Verbindungsvektoren zu den $2\,d$ nächsten Nachbarn nennen wir $\Delta\boldsymbol{x}_i\,, i = 1, 2, \ldots, 2\,d$. Die Bewegung unseres Zufallswanderers ist nun dadurch bestimmt, daß er in jedem Zeitschritt Δt, wenn er zur Zeit t am Gitterplatz $\boldsymbol{x}$ ist, von den nächsten Nachbarplätzen $\boldsymbol{x} + \Delta\boldsymbol{x}_i$ einen zufällig auswählt und dort hinspringt. Setzen wir jetzt viele Teilchen auf das Gitter, die unabhängig voneinander alle nach demselben Zufallsrezept hüpfen, und bezeichnen mit $u(\boldsymbol{x}, t)$ den Bruchteil derer, die zur Zeit t am Platz $\boldsymbol{x}$ sind, so erhalten wir folgende Bilanzgleichung:

$$u(\boldsymbol{x}, t + \Delta t) - u(\boldsymbol{x}, t) = \frac{1}{2d} \sum_{i=1}^{2d} \left(u(\boldsymbol{x} + \Delta\boldsymbol{x}_i, t) - u(\boldsymbol{x}, t) \right) . \tag{5.8}$$

Den Term $u(\boldsymbol{x}, t)$ haben wir auf beiden Seiten subtrahiert, damit man leichter sieht, daß dies gerade die diskrete Version der Diffusionsgleichung ist. Dividieren wir nämlich (5.8) sowohl durch ein geeignetes kleines Volumen, so daß eine Dichte entsteht, als auch durch Δt und lassen alle kleinen Größen, insbesondere alle $\Delta\boldsymbol{x}_i$ proportional zu $\sqrt{\Delta t}$, gegen 0 gehen, so erhalten wir für die Wahrscheinlichkeitsdichte die Gleichung

$$\frac{\partial u}{\partial t} = \eta\, \boldsymbol{\nabla}^2 u \tag{5.9}$$

mit einer Diffusionskonstanten η.

Wir wollen zwei charakteristische Lösungen dieser Diffusionsgleichung diskutieren, und zwar zum einen die Lösung eines Anfangswertproblems, die uns Auskunft gibt über Art und Geschwindigkeit, mit der sich im Mittel ein Zufallsweg ausbreitet, und zum anderen eine stationäre Lösung, aus der wir eine Abschätzung der fraktalen Dimension D des DLA-Clusters, an den sich die diffundierenden Teilchen anlagern, gewinnen können.

Gleichung (5.9) mit der Anfangsbedingung $u(\boldsymbol{x}, t_0) = \delta(\boldsymbol{x} - \boldsymbol{x}_0)$ und der Randbedingung $u \to 0$ für $|\boldsymbol{x}| \to \infty$ läßt sich durch Fouriertransformation lösen. Man

erhält für $t \geq t_0$:

$$u(\boldsymbol{x}, t) = (4\pi\eta(t - t_0))^{-\frac{d}{2}} \exp\left(-\frac{|\boldsymbol{x} - \boldsymbol{x}_0|^2}{4\eta(t - t_0)}\right). \tag{5.10}$$

Wir sehen hieran, daß die Wahrscheinlichkeit, das hüpfende Teilchen, das zur Zeit t_0 bei $\boldsymbol{x}_0$ war, zu einer späteren Zeit t bei $\boldsymbol{x}$ anzutreffen, nur vom Abstand $r = |\boldsymbol{x} - \boldsymbol{x}_0|$, nicht aber von der Richtung abhängt. Das ist nicht weiter verwunderlich und drückt nur die räumliche Isotropie des Diffusionsprozesses aus, die wir hineingesteckt haben, indem wir alle Raumrichtungen gleichbehandelt haben. Der Mittelwert $\langle(\boldsymbol{x} - \boldsymbol{x}_0)^2\rangle$ ergibt sich zu

$$\langle(\boldsymbol{x} - \boldsymbol{x}_0)^2\rangle = 2\, d\, \eta\, (t - t_0) \tag{5.11}$$

und besagt, daß die mittlere Ausdehnung des Zufallsweges wie $\sqrt{t - t_0}$ anwächst. Dieses Gesetz ist bis auf den Vorfaktor unabhängig von der Dimension d des Raumes, in den die Zufallsbewegung eingebettet ist.

Das Modell von Witten und Sander beschreibt ein diffusionsbestimmtes Wachstum, das nach folgender Vorschrift abläuft. Man beginnt mit einem minimalen Cluster, in der Regel mit einem einzelnen Teilchen im Ursprung des Koordinatensystems. Dann läßt man in großer Entfernung und unter einer zufälligen Richtung ein weiteres Teilchen starten, das solange frei diffundiert, bis es an den Kern im Ursprung anstößt und dort haften bleibt. Daraufhin startet weit draußen, wiederum unter einer zufällig ausgewürfelten Richtung das nächste Teilchen, und dann immer so weiter. Im Mittel über viele Realisierungen wird der zentrale Cluster radialsymmetrisch wachsen. Für eine konkrete Realisierung bezeichnen wir die maximale radiale Ausdehnung des Clusters mit $R_{\max}(t)$ und bestimmen seine fraktale Dimension D durch den Zusammenhang zwischen $R_{\max}(t)$ und seiner Masse $M(t)$, der Zahl der bis zum Zeitpunkt t angelagerten Teilchen:

$$M \propto R_{\max}^D\,. \tag{5.12}$$

Wenn wir dieses Modell durch eine Wahrscheinlichkeitsdichte $u(\boldsymbol{x}, t)$ der diffundierenden Teilchen beschreiben wollen, so müssen wir als Randbedingung berücksichtigen, daß die Dichte am gerade entstandenen Cluster verschwindet. Ein weiteres Teilchen wird mit einer Wahrscheinlichkeit, die proportional zum Teilchenstrom ist, am Cluster angelagert. Näherungsweise können wir aber auch annehmen, daß alle Teilchen, die einen gewissen Einfangradius R_c unterschreiten, durch Absorption weggefangen werden. Nach Witten und Sander ist aber R_c proportional zu $R_{\max}$, und wir machen für die nachfolgende Dimensionsbetrachtung keinen Fehler, wenn wir R_c durch $R_{\max}$ ersetzen. Außerdem können wir davon ausgehen, daß die Zeit, die den Diffusionsprozeß charakterisiert, z. B. die Zeit, die ein diffundierendes Teilchen benötigt, um die Strecke zum Nachbarplatz zurückzulegen, sehr viel kleiner ist

als die Zeit, die das Anwachsen des Clusters charakterisiert. Mit anderen Worten, wir suchen eine radialsymmetrische, zeitunabhängige Lösung der Diffusionsgleichung (5.9) mit der Randbedingung $u(R_{\max}) = 0$. Die Gleichung

$$\Delta u = \frac{1}{r^{d-1}} \frac{\partial}{\partial r} r^{d-1} \frac{\partial}{\partial r} u = 0 \tag{5.13}$$

können wir aber leicht integrieren. Wir erhalten für $d > 2$ die Lösung

$$u(r) = u_0 \left(1 - \left(\frac{R_{\max}}{r} \right)^{d-2} \right), \tag{5.14}$$

und für $d = 2$ ist u proportional zu $\ln(r/R_{\max})$. Die radiale Komponente der Stromdichte, $j_r = -\eta\, \partial u / \partial r$, ist in jedem Fall proportional zu $R_{\max}^{d-2}/r^{d-1}$, woraus wir nach Integration über die Oberfläche erhalten, daß der Gesamtteilchenstrom J, der vom Cluster absorbiert wird, proportional zu $R_{\max}^{d-2}$ ist. J ist proportional zur Massenzunahme, so daß sich die folgende Gleichung ergibt:

$$R_{\max}^{d-2} \propto J = \frac{dM}{dt} = \frac{dM}{dR_{\max}} \frac{dR_{\max}}{dt} \propto R_{\max}^{D-1}\, v\,. \tag{5.15}$$

Die Geschwindigkeit v, mit der die Clustergröße zunimmt, ist also proportional zu $R_{\max}^{d-1-D}$. Stecken wir jetzt noch die plausible Annahme hinein, daß v jedenfalls nicht mit der Clustergröße anwachsen kann, so darf der Exponent $d-1-D$ nicht positiv sein, es muß also $d-1 \leq D$ gelten. Andererseits ist der DLA-Cluster in den d-dimensionalen Raum eingebettet, so daß wir die Relation

$$d - 1 \leq D \leq d \tag{5.16}$$

erhalten, die durch numerische Simulationen für $d = 2, 3, \ldots, 8$ bestätigt wurde.

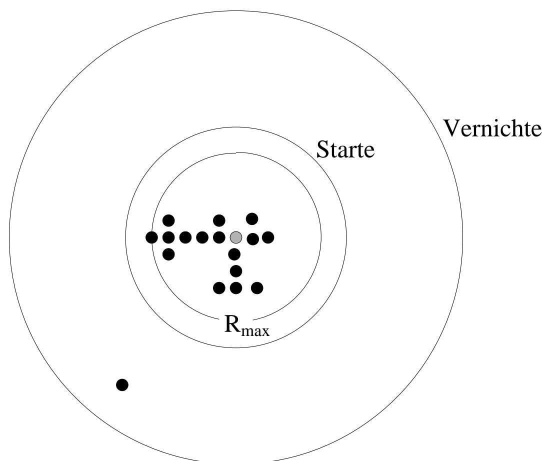
Algorithmus

Wir wollen einen DLA-Cluster auf einem 2-dimensionalen quadratischen Gitter erzeugen. Dazu besetzen wir den Platz in der Mitte des Gitters, setzen ein weiteres Teilchen im Abstand R_s auf das Gitter und lassen es diffundieren, bis es entweder am besetzten Platz hängenbleibt oder einen Abstand $R_k > R_s$ dazu überschritten hat. Dies wird iteriert. Der Algorithmus lautet also:

1. Starte mit einem leeren Quadratgitter und besetze den mittleren Gitterplatz mit einem Teilchen.
2. Die aktuelle Größe des Aggregates sei durch den Radius $R_{\max}$ gegeben. Setze ein Teilchen auf einen zufällig gewählten Platz, der einen Abstand $R_s \gtrsim R_{\max}$ vom Ursprung hat, und lasse es zu einem zufällig gewählten Nachbarplatz hüpfen.

3. Wenn das Teilchen einen Nachbarplatz des Aggregates erreicht, so wird es hinzugefügt und $R_{\max}$ wird eventuell erhöht. Wenn das Teilchen den Abstand R_k ($R_\mathrm{k} > R_\mathrm{s} > R_{\max}$) überschreitet, dann wird es vernichtet.
4. Iteriere 2 und 3 und berechne $D = \ln N / \ln R_{\max}$, wobei N die Anzahl der Aggregatteilchen ist.

Die Skizze 5.3 zeigt noch einmal den kleinsten Kreis mit dem Radius $R_{\max}$ um das Aggregat, den Startkreis und den Vernichtungskreis. Allerdings verfälscht die Einführung eines Vernichtungskreises die Eigenschaften des Aggregates, denn das diffundierende Teilchen kann auch für $R > R_\mathrm{k}$ wieder zum Aggregat zurückkehren. Unser Algorithmus beschreibt also erst im Limes $R_\mathrm{k} \to \infty$ das eigentliche Problem. Große R_k-Werte bedeuten aber große Rechenzeiten, und wir erwarten keine wesentlichen Abweichungen für endliche R_k-Werte. Man kann die Wahrscheinlichkeit, daß das Teilchen an eine bestimmte Stelle des Startkreises zurückkehrt, analytisch berechnen und damit das Teilchen in einem Schritt zurücksetzen. Damit vermeidet man den Vernichtungskreis. Bei großen Aggregaten lohnt sich dieser Aufwand, doch wir wollen hier darauf verzichten.

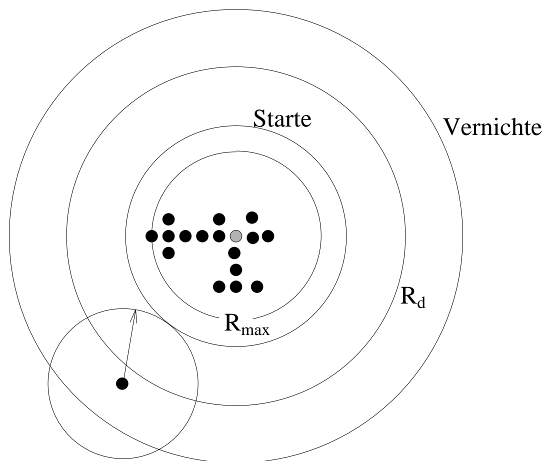


5.3 Das diffundierende Teilchen startet am mittleren Kreis. Wenn es den äußeren Kreis überschreitet, wird es vernichtet, und ein neues Teilchen wird auf den Startkreis gesetzt.

Der Algorithmus läßt sich darüber hinaus noch wesentlich beschleunigen, wenn man um das Teilchen, das sich vom Cluster wegbewegt hat, einen möglichst großen Kreis konstruiert, der gerade den Rand des Aggregates berührt. Da das diffundierende Teilchen nach Gleichung (5.10) mit gleicher Wahrscheinlichkeit an jedem Punkt dieses Kreises ankommt, kann man es zufällig auf den Kreis setzen, ohne die zeitaufwendige Diffusion zu simulieren. Dies wird solange wiederholt, bis das Teilchen einen Nachbarplatz des Aggregates erreicht hat oder den Vernichtungskreis überschreitet. Der Algorithmus dazu lautet also:

1. Starte mit einem leeren Quadratgitter und besetze den mittleren Gitterplatz mit einem Aggregatteilchen.
2. Die aktuelle Größe des Aggregates ist $R_{\max}$. Setze ein Teilchen auf einen zufällig gewählten Platz auf den Kreis mit dem Radius $R_\mathrm{s} \gtrsim R_{\max}$ und lasse es zu einem zufällig gewählten Nachbarplatz hüpfen.
3. Wenn das Teilchen einen Nachbarplatz des Aggregates erreicht, so wird es hinzugefügt und $R_{\max}$ wird eventuell erhöht. Wenn das Teilchen den Abstand $R_\mathrm{d} > R_\mathrm{s} \gtrsim R_{\max}$ überschreitet – seine Entfernung vom Ursprung sei R –, so wird es auf einen zufällig gewählten Platz auf einem Kreis um das Teilchen mit dem Radius $(R - R_\mathrm{s})$ gesetzt. Wenn das Teilchen den Abstand $R_\mathrm{k} > R_\mathrm{d} > R_\mathrm{s} \gtrsim R_{\max}$ überschreitet, so wird es vernichtet.
4. Iteriere 2. und 3. und berechne $D = \ln N / \ln R_{\max}$, wobei N die Anzahl der Aggregatteilchen ist.

Die Skizze 5.4 soll diesen Algorithmus noch einmal verdeutlichen. Zur schnellen



5.4 Wie Bild 5.3, doch außerhalb des Kreises mit dem Radius R_d legt das Teilchen mit einem Sprung die Strecke $R - R_\mathrm{s}$ zurück.

Simulation des DLA-Clusters wählen wir die Programmiersprache C. Das Quadratgitter wird durch das zweidimensionale Feld `xf[rx][ry]` beschrieben, das die Werte 1 und 0 annehmen kann. `xf[rx][ry]=1` bedeutet, daß am Ort mit den Koordinaten `rx` und `ry` ein Aggregatteilchen sitzt, während die Plätze mit `xf=0` zur Diffusion zur Verfügung stehen. Bei sehr großen Aggregaten wäre es sicherlich sinnvoll, nicht das ganze Gitter, sondern nur die Aggregat- und Randplätze zu speichern, um aber die dazu erforderliche Buchhaltung zu vermeiden, wählen wir den einfachen Weg.

Der Startplatz auf dem Kreis mit Radius R_s wird mit der Funktion `besetze()` gewählt:

```
void besetze()
{
  double phi;
  phi=(double)rand()/RAND_MAX*2.*pi;
  rx=rs*sin(phi);
  ry=rs*cos(phi);
}
```

Das Hüpfen auf einen der vier Nachbarplätze wird mit einer zufällig gewählten Zahl $0, 1, 2$ oder 3 gesteuert:

```
void huepfe()
{
   int r;
   r=random(4);
   switch(r)
   {
     case 0: rx+=1;break;
     case 1: rx+=-1;break;
     case 2: ry+=1;break;
     case 3: ry+=-1;break;
   }
}
```

Nach jedem Sprung muß geprüft werden, ob das Teilchen vernichtet wird ($R > R_{\mathrm{k}}$), ob es einen Randplatz des Aggregates erreicht hat, ob ein Hüpfsprung ($R < R_{\mathrm{d}}$) oder ein Kreissprung ($R \geq R_{\mathrm{d}}$) gemacht wird. Diese vier „wenn" im Schritt Nr. 3 des obigen Algorithmus werden durch die Funktion `pruefe()` abgefragt:

```
char pruefe()
{
  double r,x,y;
  x=rx;
  y=ry;
  r=sqrt(x*x+y*y);
  if (r > rkill)  return 'v';
  if (r >= rd)    return 'k';
  if (xf[rx + 1 + lmax/2][ry + lmax/2] +
     xf[rx - 1 + lmax/2][ry + lmax/2] +
     xf[rx + lmax/2][ry + 1 + lmax/2] +
     xf[rx + lmax/2][ry - 1 + lmax/2] > 0)
    return 'a';
  else
```

```
    return 'h';
}
```

`lmax` ist die Größe des Zeichenfensters. Die folgende Funktion fügt das Teilchen zum Aggregat hinzu:

```
void aggregate()
{
  double x,y;
  xf[rx+lmax/2][ry+lmax/2]=1;
  x=rx;y=ry;
  rmax= max(rmax,sqrt(x*x+y*y));
  if(rmax>lmax/2.-5.) {printf("\7");stop=1;}
  circle(4*rx+340,4*ry+240,2);
}
```

und beim Kreissprung wird ein Zufallsvektor mit dem Betrag $R - R_\mathrm{s}$ zum Ort des Teilchens addiert,

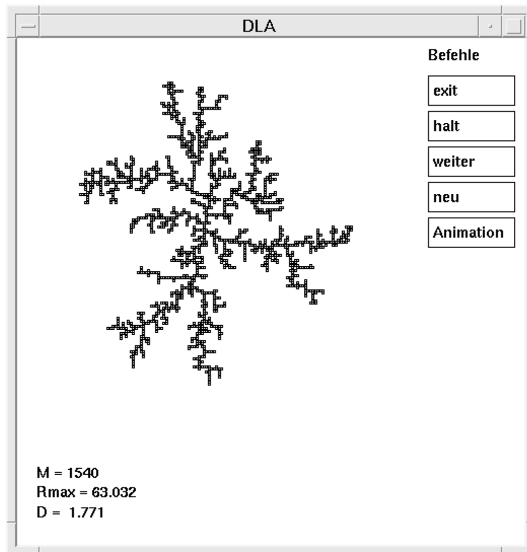
```
void kreissprung()
{
  double r,x,y,phi;
  phi=(double)rand()/RAND_MAX*2.*pi;
  x=rx; y=ry; r=sqrt(x*x+y*y);
  rx+=(r-rs)*sin(phi);
  ry+=(r-rs)*cos(phi);
}
```

Die Koordinaten `rx` und `ry` des Teilchens und das Aggregatfeld `xf[rx][ry]` werden vor dem Hauptteil `main()` global deklariert, so daß alle Funktionen auf sie zugreifen können. Mit den obigen fünf Funktionen lautet dann der wesentliche Teil des Hauptprogrammes

```
while(stop==0)
 {
  if(kbhit()) stop=1;
  switch(pruefe())
   {
    case 'v':besetze();huepfe();break;
    case 'a':aggregate();besetze();huepfe();break;
    case 'h':huepfe();break;
    case 'k':kreissprung();break;
   }
 }
```

Ergebnis

In Bild 5.5 ist das Ergebnis der Simulation zu sehen. Auf der Workstation haben sich innerhalb einer Minute 1540 Teilchen zu einem fraktalen Aggregat angelagert. Mit der maximalen Ausdehnung von 63 Gitterabständen erhält man die Abschätzung $D = \ln 1540/\ln 63 \simeq 1.77$ für die fraktale Dimension.



5.5 DLA-Cluster auf dem Quadratgitter.

Die Simulationen zeigen außerdem, daß durch die Einführung der großen Sprünge für $R > R_{\mathrm{d}}$ die Rechenzeit sich nicht wesentlich erhöht, wenn man den Radius des Vernichtungskreises vergrößert.

Übung

Verbesserte Dimensionsanalyse

Bei der Bestimmung der fraktalen Dimension haben wir bisher vernachlässigt, daß der momentan erzeugte Cluster noch nicht fertig ist. Während sich die „Arme“ des Clusters weiter ausbilden, können immer noch weitere Teilchen in seine inneren Bereiche gelangen. Durch unser willkürliches Abbrechen beim Erreichen eines gewissen maximalen Radius vernachlässigen wir diese Teilchen. Der tatsächliche Cluster sollte etwas dichter, die fraktale Dimension etwas höher sein.

Berechnen Sie die Masse des Clusters unter Vernachlässigung aller Teilchen, die einen Abstand größer als r haben. Tragen Sie in einem log-log-Diagramm die Masse über r auf und versuchen Sie, das Verhalten zu interpretieren.

Variation der Modell-Parameter

Es ist interessant zu beobachten, inwieweit sich Form und fraktale Dimension des Clusters verändern, wenn man das ursprüngliche DLA-Modell leicht variiert.

Führen Sie in das Programm `dla.c` einen zusätzlichen Parameter w_{haft} ein, der die Wahrscheinlichkeit darstellt, daß ein Teilchen, das auf einem dem Cluster benachbarten Platz ankommt, auch tatsächlich haften bleibt.

Für den Fall, daß das Teilchen nicht befestigt wird, soll es weiter diffundieren bis es erneut den Cluster trifft, wo es wieder mit der Wahrscheinlichkeit w_{haft} haften bleibt. Beim Weiterdiffundieren ist natürlich darauf zu achten, daß dem Teilchen verboten wird, auf einen bereits besetzten Platz zu hüpfen.

Literatur

H. Gould, J. Tobochnik, *An Introduction to Computer Simulation Methods: Applications to Physical Systems, Parts I and II*, Addison Wesley, 1988.

E. Sander, L. M. Sander, R. M. Ziff, *Fractals and fractal correlations*, Computers in Physics **8**, 420 (1994).

5.3 Perkolation

In einem idealen Kristall sind alle Atome regelmäßig auf einem Gitter angeordnet. Für diesen Fall können wir die Fouriertransformation benutzen, um die periodischen Strukturen und auch die Kristallanregungen, z. B. die Phononen, zu beschreiben. Weil aber unsere Natur nur selten aus idealen Kristallen besteht, bemühen sich die Festkörperphysiker seit Jahrzehnten, auch Materialien mit unregelmäßigen Strukturen zu verstehen.

Es gibt eine einfache Beschreibung eines ungeordneten Materials, das Perkolationsmodell. Das englische Wort *to percolate* bedeutet *durchsickern*. Das Modell soll ein poröses Material beschreiben, durch das gewisse Teilchen hindurchdiffundieren können, falls es durchgehende Wege durch die Poren gibt. Die Geometrie der zufällig entstanden Poren wird durch die Perkolationstheorie beschrieben. Das Modell kann jedem Schüler erklärt werden, es kann auf einem Computer leicht programmiert werden, und dennoch enthält es interessante Physik, die mit Phasenübergängen, kritischen Phänomenen, Universalität und selbstähnlichen Fraktalen zusammenhängt.

In diesem Abschnitt wollen wir das Perkolationsmodell vorstellen und simulieren.

Physik

Nehmen wir an, wir haben eine Legierung, die aus zwei Atomsorten besteht, die wir A- und B-Atome nennen wollen. Die Konzentration der A-Atome sei p, und beide Teilchensorten sollen zufällig verteilt sein und zusammen ein regelmäßiges Kristallgitter bilden. Wir wollen weiterhin annehmen, daß die A-Atome magnetisch sind mit einer kurzreichweitigen magnetischen Wechselwirkung, die nur bei unmittelbarer Nachbarschaft zweier A-Atome zum Tragen kommt. Wenn es keine Kopplung zu den unmagnetischen B-Atomen gibt, so kann der Kristall nur dann eine magnetische Ordnung bilden, wenn ein unendlich großes Netz von zusammenhängenden A-Atomen existiert. Phasenübergänge gibt es zwar streng genommen nur in unendlich großen Systemen mit hinreichend großer Vernetzung, in unseren Computersimulationen werden wir aber auch für endlich große Systeme bei einer kritischen Konzentration der A-Atome deutliche Anzeichen eines Phasenübergangs erkennen können. Die Größe der Strukturen von A-Atomen, die alle untereinander verbunden sind, spielt dabei eine wichtige Rolle. Eine solche Menge von A-Atomen, die durch ihre Kopplungen zusammenhängen, wird *Cluster* genannt.

Bei kleinen Konzentrationen p gibt es nur eine Menge kleiner Cluster ohne magnetische Ordnung. Mit wachsendem p wird die mittlere Größe der zusammenhängenden A-Atome ebenfalls wachsen, bis sich ab einem Schwellenwert p_c ein Cluster durch den ganzen Kristall ausdehnt. p_c nennt man Perkolationsschwelle oder kritische Konzentration. Für $p > p_c$ koexistiert der unendliche mit vielen kleinen Clustern, erst bei $p = 1$ sind alle Plätze mit A-Atomen besetzt, und es gibt nur einen (unendlichen) Cluster.

Für den unendlich großen Kristall gibt es nur *einen* wohldefinierten Wert für die Perkolationsschwelle p_c. Beispielsweise findet man auf dem quadratischen Gitter mit vier nächsten Nachbarn den numerischen Wert $p_c = 0.59275 \pm 0.00003$, auf dem Dreiecksgitter mit sechs Nachbarplätzen ist $p_c = 1/2$ exakt bekannt. p_c hängt zwar vom Gittertyp und der Reichweite der Kopplungen ab, nicht aber von der speziellen Realisierung der Probe, wenn man die Größe des Kristalls gegen Unendlich gehen läßt. Im endlichen Kristall dagegen gibt es, abhängig von p, natürlich nur eine gewisse Wahrscheinlichkeit dafür, daß ein Cluster existiert, der zwei gegenüberliegende Seiten verbindet.

Man kann nun viele Fragen zu den Eigenschaften der Cluster stellen und dazu mathematische Gesetzmäßigkeiten erarbeiten. Wie groß ist p_c für verschiedene Gitterstrukturen und Clusterdefinitionen? Wie wächst die mittlere Ausdehnung der endlichen Cluster, wenn man die Konzentration p bis zur Schwelle p_c erhöht? Wie wächst die Dichte des unendlichen Clusters oberhalb p_c? Wie sieht die Verteilung der Clustergrößen aus? Hat der unendliche Cluster bei p_c eine Struktur?

Besonders interessant sind die Gesetzmäßigkeiten in der Nähe der Perkolationsschwelle p_c. Wir definieren z.B. eine mittlere Ausdehung $R(s)$ eines Clusters mit s

Teilchen durch

$$R^2(s) = \frac{1}{s(s-1)} \sum_{i \neq j} (\boldsymbol{r}_i - \boldsymbol{r}_j)^2 \,, \tag{5.17}$$

wobei i und j die Teilchen des Clusters durchnumerieren und $\boldsymbol{r}_i$ die Position des i-ten Atoms auf dem Gitter ist. Die mittlere Ausdehnung ξ der endlichen Cluster ist dann definiert durch

$$\xi = \sqrt{\langle R^2(s) \rangle_{s<\infty}} \tag{5.18}$$

dabei bezeichnet $\langle ... \rangle_{s<\infty}$ den Mittelwert über alle endlichen Cluster .

ξ divergiert an der Perkolationsschwelle, und dicht bei p_c gilt:

$$\xi \sim |p - p_c|^{-\nu} \,. \tag{5.19}$$

Dieses Gesetz erhält man für sämtliche Perkolationsmodelle. Überraschenderweise ist der Exponent ν universell. Das heißt, er hat einen Wert, der nur von der Raumdimension abhängt. In der Ebene ist für verschiedene Gittertypen und Reichweiten der Kopplungen, auch für das entsprechende Problem ohne Gitter, und in realen zweidimensionalen Legierungen der Wert $\nu = 4/3$ exakt bekannt. Für drei Dimensionen erhält man numerisch $\nu \simeq 0.88$.

Die Wahrscheinlichkeit $P(p)$, daß ein beliebig herausgegriffenes A-Atom zum unendlichen Cluster gehört, ist null für $p < p_c$ und wächst stetig von $P(p_c) = 0$ bis $P(1) = 1$ an. Dicht bei p_c wird der Anstieg wieder durch ein Potenzgesetz mit einem universellen Exponenten β beschrieben:

$$P(p) \sim (p - p_c)^{\beta}, \qquad (p \gtrsim p_c) \,. \tag{5.20}$$

In zwei Dimensionen gilt $\beta = 5/36 \simeq 0.14$; $P(p)$ wächst also bei p_c sehr steil an.

Direkt an der Perkolationsschwelle verschwindet zwar die Dichte des unendlichen Clusters, $P(p_c)\, p_c = 0$, trotzdem existiert er, und er hat eine interessante Struktur: Er ist ein Fraktal. Wenn wir einen quadratischen Ausschnitt des Gitters mit der Kantenlänge L betrachten, und wenn $M(L)$ die Anzahl der Teilchen des unendlichen Clusters in diesem Ausschnitt ist, dann gilt bei p_c:

$$\langle M(L) \rangle \propto L^D \,, \tag{5.21}$$

wobei $\langle ... \rangle$ den Mittelwert über verschiedene Ausschnitte bezeichnet. Die Masse des Clusters wächst also mit einer Potenz der Länge, und nach Abschnitt 3.3 ist D seine fraktale Dimension. In der Ebene gilt $D = 91/48 \simeq 1.89$. Der kritische Perkolationscluster ist also wesentlich kompakter als das durch Diffusion erzeugte Aggregat aus dem vorherigen Abschnitt.

Bisher haben wir drei universelle kritische Exponenten ν, β und D kennengelernt, aber noch viele andere Eigenschaften werden bei p_c durch Potenzgesetze

mit weiteren Exponenten beschrieben. Solche Singularitäten sind aber nicht unabhängig voneinander, sondern hängen durch Skalengesetze miteinander zusammen. Wir wollen hier nur die Idee skizzieren und für das tiefere Verständnis auf die Lehrbücher zu den kritischen Phasenübergängen und der Renormierungsgruppentheorie verweisen.

Sei $M(p, L)$ die mittlere Anzahl der Teilchen eines Clusters, der ein Quadrat der Länge L durchquert. Für jeden Wert von L erhält man die Anzahl M als Funktion von p ; dies ergibt eine ganze Kurvenschar. Die Skalentheorie besagt nun, daß diese Kurvenschar in der Nähe des kritischen Punktes nach geeigneter Skalierung durch eine einzige universelle Funktion f beschrieben wird. Die Skalen für die Größen M und L können durch Potenzen von entweder $p - p_\mathrm{c}$ oder mit Gleichung (5.19) von ξ ausgedrückt werden. L wird dabei in Einheiten von ξ und M in Einheiten einer Potenz von ξ, z. B. ξ^x, gemessen. Das bedeutet:

$$M(p, L) \sim \xi^x f\left(\frac{L}{\xi}\right), \tag{5.22}$$

wobei f eine zunächst unbekannte Funktion und x ein kritischer Exponent ist. Setzt man $L = k\,\xi$ mit einer Konstanten k, so folgt aus Gleichung (5.21) $x = D$. L wird also in Einheiten von ξ und M in Einheiten von ξ^D gemessen, daher der Name Skalentheorie.

Aus $L = k\,\xi$ folgt auch:

$$\frac{M(p, L)}{L^d} \sim \xi^{D-d} f(k), \tag{5.23}$$

wobei L^d die Anzahl der Gitterpunkte im d-dimensionalen Würfel mit der Kantenlänge L ist. M/L^d ist aber die Wahrscheinlichkeit, daß ein Gitterplatz zum perkolierenden Cluster gehört. Für große L-Werte gilt daher:

$$\frac{M(p, L)}{L^d} \simeq p\,P(p) \sim (p - p_\mathrm{c})^\beta . \tag{5.24}$$

Aus den Gleichungen (5.19), (5.23) und (5.24) folgt

$$(p - p_\mathrm{c})^\beta \sim (p - p_\mathrm{c})^{-\nu\,(D-d)} . \tag{5.25}$$

Daraus kann man schließen:

$$\beta = (d - D)\,\nu . \tag{5.26}$$

Die drei kritischen Exponenten sind damit durch ein Skalengesetz miteinander verkoppelt. Tatsächlich genügt die Kenntnis zweier Exponenten, um alle anderen zu berechnen.

Zunächst sind die kritischen Exponenten nur für das unendlich große Gitter definiert. Mit dem Computer können wir aber nur endliche Gitter mit Teilchen besetzen. Im endlichen System gibt es jedoch keinen Phasenübergang, keine scharf definierte Perkolationsschwelle und keine Divergenzen. Wie kann man von den Eigenschaften eines endlichen Systems auf die kritischen universellen Gesetze des unendlichen Gitters schließen?

Die Theorie des *finite size scaling* beantwortet diese Frage. Sie basiert auf den oben skizzierten Skalengesetzen, die auch etwas über die Abhängigkeit der Singularitäten von der Gittergröße L aussagen. Benutzt man das durch Gleichung (5.19) beschriebene Verhalten der Korrelationslänge in der Skalenbeziehung (5.22), so erhält man mit $f(x) = \tilde{f}(x^{1/\nu})$:

$$M(p, L) \cdot |p - p_c|^{\nu D} \sim \tilde{f}\left((p - p_c) L^{1/\nu}\right) . \tag{5.27}$$

Die Konzentration p und die Systemlänge L sind also in der Nähe des kritischen Punktes $p = p_c, L = \infty$ miteinander verknüpft.

Die fraktale Dimension D erhält man schon aus Gleichung (5.21). Denn für $L \ll \xi$ sieht das endliche System wie das unendlich große aus. Deshalb folgt aus Gleichung (5.21) $M(p_c, L) \sim L^D$. Aus dem Anstieg der Zahl $M(p_c, L)$ der Teilchen im perkolierenden Cluster mit der Gittergröße L kann man also die fraktale Dimension D numerisch berechnen. Wie aber bestimmt man p_c und den Exponenten ν?

Dazu stellen wir ein Skalengesetz analog zu Gleichung (5.27) für die Wahrscheinlichkeit $\pi(p, L)$ auf, daß es in einer Probe der Größe L einen Cluster gibt, der zwei gegenüberliegende Seiten verbindet, der also perkoliert. Offenbar gilt

$$\pi(p, \infty) = \begin{cases} 0 & \text{für} \quad p < p_c , \\ 1 & \text{für} \quad p > p_c . \end{cases} \tag{5.28}$$

Für endliche L-Werte wird diese Stufenfunktion in der Nähe von p_c abgerundet. Da π im unendlichen System für $p > p_c$ eine Konstante ist, hat der entsprechende Skalenexponent den Wert Null. Die Skalengleichung für $\pi(p, L)$, analog zu Gleichung (5.27) für $M(p, L)$, lautet also mit einer unbekannten Funktion g:

$$\pi(p, L) = g\left((p - p_c)\, L^{1/\nu}\right) . \tag{5.29}$$

Wenn wir nun die Plätze eines Gitters mit der Wahrscheinlichkeit p mit Atomen besetzen und dann p erhöhen, so wachsen alle Cluster. Bei einem Schwellenwert $p_c(L)$ wird ein perkolierender Cluster auftreten.

Die Wahrscheinlichkeit, daß $p_c(L)$ im Intervall $[p, p + dp]$ liegt, ist durch die Ableitung $(d\pi/dp)\, dp$ an der Stelle $p_c(L)$ gegeben. $d\pi/dp$ zeigt ein Maximum, das für $L \to \infty$ divergiert und zur Perkolationsschwelle p_c des unendlichen Gitters läuft. Wegen der großen statistischen Fluktuationen läßt sich das Maximum von

$d\pi/dp$ nur schlecht numerisch bestimmen. Es ist besser, den Mittelwert und die Fluktuationen von $p_c(L)$ zu berechnen. Der Mittelwert der Perkolationsschwelle ist gegeben durch

$$\langle p_c(L)\rangle = \int p \, \frac{d\pi}{dp} \, dp \,. \tag{5.30}$$

Aus Gleichung (5.29) erhält man dafür das Skalengesetz

$$\langle p_c(L)\rangle = \int pL^{1/\nu} g'\left((p-p_c)L^{1/\nu}\right) \, dp \,. \tag{5.31}$$

Mit der Substitution $z = (p - p_c)L^{1/\nu}$ und mit $\int \frac{d\pi}{dp} \, dp = \pi(1, L) - \pi(0, L) = 1$ folgt daraus:

$$\langle p_c(L)\rangle - p_c \sim L^{-1/\nu} \,. \tag{5.32}$$

Wenn wir nun für viele Simulationen eines Systems der Länge L den Wert $p_c(L)$ bestimmen und dies für viele möglichst große L-Werte wiederholen, so kann man den Mittelwert $\langle p_c(L)\rangle$ durch das Gesetz (5.32) fitten und dadurch p_c und ν bestimmen. Man kann auch gleichzeitig den Mittelwert des Quadrates von $p_c(L)$ berechnen und erhält damit ein Maß Δ für die Breite von $d\pi/dp$. Führen wir nämlich für

$$\begin{aligned} \Delta^2 &= \langle (p_c(L) - \langle p_c(L)\rangle)^2\rangle = \langle p_c(L)^2\rangle - \langle p_c(L)\rangle^2 \\ &= \langle (p_c(L) - p_c)^2\rangle - \langle p_c(L) - p_c\rangle^2 \end{aligned} \tag{5.33}$$

die zu (5.30) und (5.31) analoge Rechnung durch, so erhalten wir

$$\Delta \sim L^{-1/\nu} \,. \tag{5.34}$$

Aus der Standardabweichung von $p_c(L)$ läßt sich somit der Parameter ν direkt gewinnen.

Mit Hilfe der Skalengesetze gelingt es uns also, die kritischen Werte des unendlichen Gitters aus den Eigenschaften endlicher Systeme zu berechnen. Dieses *finite size scaling* ist nicht nur auf das Perkolationsproblem beschränkt, sondern es gilt auch für andere Arten von Phasenübergängen, z. B. für den magnetischen Übergang des Ising-Ferromagneten, den wir in Abschnitt 5.5 behandeln werden. Wir haben damit ein wichtiges Werkzeug kennengelernt, um universelle, also modellunabhängige Größen zu berechnen.

Schließlich sollten wir erwähnen, daß es neben der hier besprochenen *site percolation* auch die sogenannte *bond percolation* gibt. Dabei werden nicht die Plätze des Gitters, die *sites*, sondern die Verbindungslinien benachbarter Plätze, die *bonds*, mit der Wahrscheinlichkeit p besetzt. Solche Systeme haben zwar eine andere Perkolationsschwelle, jedoch dieselben universellen kritischen Eigenschaften wie das besprochene Modell.

Algorithmus

Auf einem Gitter läßt sich die Perkolationsstruktur numerisch leicht erzeugen. Wir wählen ein Quadratgitter mit Gitterplätzen (i,j), $i = 0,\ldots,L-1$ und $j = 0,\ldots,L-1$, und benutzen gleichverteilte Zufallszahlen $r \in [0,1]$. Der Algorithmus lautet damit:

1. Durchlaufe alle (i,j).
2. Ziehe eine Zufallszahl r.
3. Falls $r < p$, zeichne einen Punkt am Platz (i,j).

Das C-Programm dazu lautet:

```
double p = 0.59275;
int i, j, L = 500, pr;
pr = p*RAND_MAX;
for (i = 0; i < L; i++)
for (j = 0; j < L; j++)
    if (rand() < pr) putpixel(i,j,WHITE);
```

Bild 5.6 zeigt das Ergebnis. Man sieht eine scheinbar regellose Struktur. Es ist deshalb überraschend, daß man daraus durch geeignete Fragen und deren quantitative Beantwortung so vielfältige mathematische Gesetzmäßigkeiten herausholen kann. Allerdings ist die Auswertung der Perkolationsstruktur nicht mehr so einfach zu programmieren wie deren Erzeugung. Die Frage z. B., ob die Struktur in Bild 5.6 perkoliert, ob es also einen Weg über besetzte Nachbarplätze gibt, der zwei gegenüberliegende Seiten verbindet, läßt sich nicht leicht beantworten. Auch unser Auge ist mit dieser Frage überfordert.

Wir brauchen einen Algorithmus, der Cluster von zusammenhängenden Teilchen identifizieren kann. Eine naive Methode wäre, von jedem besetzten Platz aus wachsende Kreise zu ziehen und alle verbundenen Plätze zu markieren. Dies kostet aber sehr viel Rechenzeit. Ein schneller Algorithmus wurde 1976 von Hoshen und Kopelman entwickelt. Dabei wird das Gitter nur einmal durchlaufen, und es werden Clusternummern an alle besetzten Plätze vergeben. Jeder Platz, der nicht mit vorher besuchten verbunden ist, erhält eine neue Nummer. Allerdings werden dabei manchmal auch Teile desselben Clusters mit unterschiedlichen Nummern belegt. Durch einen Buchhaltungstrick können solche Kollisionen am Ende leicht aufgelöst werden. Die Details dieses Algorithmus sind in den Lehrbüchern von Stauffer/Aharony und Gould/Tobochnik gut beschrieben.

Hier wollen wir einen anderen Weg verfolgen, der von Leath ebenfalls im Jahr 1976 vorgeschlagen wurde. Cluster sollen direkt durch einen Wachstumsprozeß erzeugt werden. Dazu besetzen wir zunächst nur das Zentrum eines leeren Gitters

und definieren alle Nachbarplätze als besetzt oder unbesetzt mit der Wahrscheinlichkeit p bzw. $1 - p$. Dieser Prozeß wird iteriert, wobei jeweils alle undefinierten Randplätze in besetzte oder unbesetzte umgewandelt werden. Wenn es keine undefinierten Randplätze des Clusters mehr gibt, dann wurde ein Perkolationscluster erzeugt. Viele Wiederholungen geben eine Verteilung von Clustern, die man mit den Methoden des *finite size scaling* auswerten kann. Da jedes erzeugte Teilchen das statistische Gewicht p und jeder Leerplatz das Gewicht $(1 - p)$ erhält, und da der gesamte Cluster mit dem Produkt aller dieser Faktoren gewichtet wird, erzeugt der Wachstumsprozeß die Cluster mit derselben Wahrscheinlichkeit wie der vorherige Algorithmus.

Der Wachstumsalgorithmus für einen Cluster lautet damit:

1. Markiere alle Plätze eines Quadratgitters mit der Seitenlänge L als undefiniert und besetze das Zentrum. Der Cluster besteht am Anfang also nur aus einem Teilchen.
2. Wähle einen undefinierten Randplatz des Clusters aus.
3. Ziehe eine gleichverteilte Zufallszahl $r \in [0, 1]$ und besetze diesen Platz, falls $r < p$. Sonst markiere diesen Platz als unbesetzt.
4. Iteriere 2 und 3, bis es keine undefinierten Randplätze des Clusters mehr gibt.

Damit der Wachstumsprozeß am Rand des Gitters in definierter Weise beendet wird, markieren wir die Gitterrandplätze mit „unbesetzt“ . Zu Punkt 2 werden wir eine Liste erzeugen, aus der wir die Positionen der noch undefinierten Randplätze des Clusters entnehmen können. Natürlich muß diese Liste während des Wachtumsprozesses ständig aktualisiert werden.

Da wir wieder möglichst große Gitter simulieren und auf dem Bildschirm den wachsenden Cluster direkt beobachten wollen, haben wir den Algorithmus in der Sprache C geschrieben. Zunächst vereinbaren wir, daß die Markierung der drei verschiedenen Fälle mit den Zahlen 0, 1 und 2 erfolgen soll:

```
#define UNDEFINIERT 0
#define BESETZT 1
#define UNBESETZT 2
```

Für das Gitter, das nur diese drei Werte anzunehmen braucht, benutzen wir einen Datentyp, der wenig Speicherplatz benötigt, nämlich `char feld[L][L]`, und definieren außerdem mit

```
struct { int x; int y ;} liste[PD];
```

eine Liste, die die Positionen `(i,j)` der undefinierten Clusterrandplätze aufnehmen soll. Diese Liste, die anfangs nur die vier Nachbarplätze des Zentrums enthält,

wird sukzessive abgearbeitet und erhält dabei neue Einträge. Weil die bearbeiteten Plätze nicht mehr benötigt werden, kann diese Liste periodisch beschrieben und durchlaufen werden. Ihre Länge PD braucht deshalb nicht gleich L^2 zu sein, sondern es reicht aus, wenn wir PD$= 4\,L$ wählen. Die Initialisierung sieht folgendermaßen aus:

```
for(i=0;i<L;i++)
   feld[0][i]=feld[i][0]=feld[i][L-1]=
              feld[L-1][i]=UNBESETZT;
for(i=1;i<L-1;i++)
for(j=1;j<L-1;j++)
   feld[i][j]=UNDEFINIERT;
feld[L/2][L/2]=BESETZT;
liste[0].x=L/2+1; liste[0].y=L/2;
liste[1].x=L/2  ; liste[1].y=L/2+1;
liste[2].x=L/2-1; liste[2].y=L/2;
liste[3].x=L/2  ; liste[3].y=L/2-1;
count=3;
putpixel(L/2, L/2, WHITE);
```

Dabei ist `count` eine reine Zählvariable, deren Wert jeweils um 1 erhöht wird, wenn ein neuer undefinierter Randplatz des Clusters entsteht. Die Hauptschleife, Punkt 4, lautet:

```
while(! done )
      {
       event();
       for(k=0; k<=count; k++)
          {
           i=liste[k%PD].x; j=liste[k%PD].y;
           definiere(i,j);
          } /* for */
      } /* while */
```

Das Programm durchläuft und bearbeitet also die Liste der noch undefinierten Randplätze des Clusters. Die Funktion `definiere(i,j)` prüft zuerst, ob der Platz `(i,j)` schon definiert ist. Das ist nötig, weil derselbe Randplatz eventuell mehrfach in der Liste auftaucht. Wenn der Platz noch undefiniert ist, wird Schritt 3 durchgeführt und der Platz `(i,j)` besetzt, falls die Zufallszahl `r` kleiner als die Konzentration `p` ist. Wenn dieser jetzt besetzte Platz noch undefinierte Nachbarn hat, werden deren Positionen in die obige Liste aufgenommen und der Wert der Variablen `count` wird entsprechend erhöht.

Das sieht im Programm folgendermaßen aus:

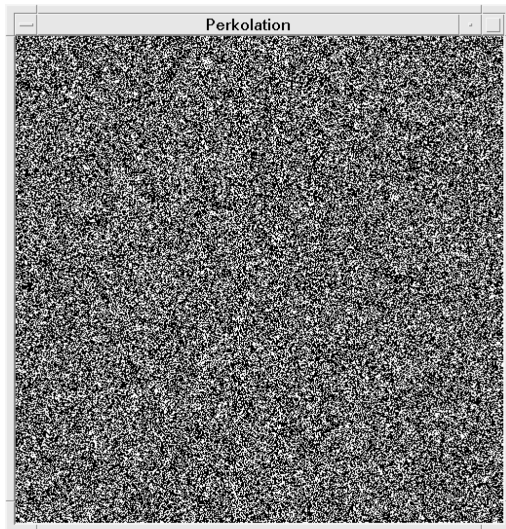
```
void definiere(int i, int j)
 {
  double r;
  if( feld [i][j] != UNDEFINIERT ) return;
  r = rand() / (double) RAND_MAX;
  if( r < p )
   {
    feld[i][j] = BESETZT;
    putpixel (i, j, WHITE);
    if( feld[i][j+1]==UNDEFINIERT )
     {count++;liste[count%PD].x=i;liste[count%PD].y=j+1;}
    if( feld[i][j-1]==UNDEFINIERT )
     {count++;liste[count%PD].x=i;liste[count%PD].y=j-1;}
    if( feld[i+1][j]==UNDEFINIERT )
     {count++;liste[count%PD].x=i+1;liste[count%PD].y=j;}
    if( feld[i-1][j]==UNDEFINIERT )
     {count++;liste[count%PD].x=i-1;liste[count%PD].y=j;}
      }
    else feld[i][j] = UNBESETZT;
   }
```

Das Programm ist noch durch die Deklarationen, die graphischen Initialisierungen und die Tastaturabfrage `event()` zu ergänzen.

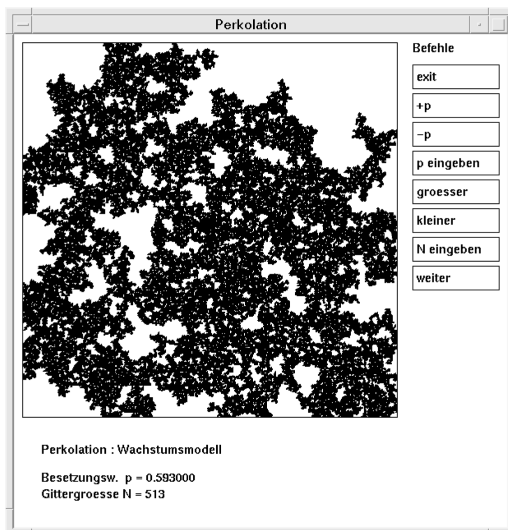
Ergebnis

Zunächst wenden wir den ersten Algorithmus zur Erzeugung der gesamten Perkolationsstruktur an und besetzen ein Quadratgitter mit 500×500 Plätzen mit der Konzentration $p = 0.59275$. Die besetzten Plätze (A-Atome) sind schwarz markiert, das Gitter und die unbesetzten Plätze werden nicht gezeigt. Bild 5.6 zeigt das Ergebnis. Man sieht nur ein Rauschen, das noch nichts von den mathematischen Gesetzen ahnen läßt, die man nach quantitativer Auswertung dieses Bildes erhalten kann. Wir haben die Konzentration p gerade an der Perkolationsschwelle gewählt, doch ein perkolierender Cluster und kritische Eigenschaften sind nicht zu erkennen. Nur ein geeignetes Computerprogramm kann aus dieser Struktur die Cluster bestimmen und quantitativ auswerten.

Bild 5.7 zeigt einen Cluster, der durch den Wachstumsalgorithmus erzeugt wurde. Die Konzentration p und die Gittergröße stimmen mit denen von Bild 5.6 überein. Jetzt sieht man einen perkolierenden Cluster, der Strukturen auf allen Längenskalen von der Gitterkonstanten bis zur Gesamtgröße des Gitters zeigt. Er ist ein fraktales selbstähnliches Gebilde. Natürlich erzeugt das Programm nicht nur perkolierende



5.6 Perkolationsstruktur auf einem Quadratgitter an der Perkolationsschwelle.



5.7 Perkolierender Cluster an der Perkolationsschwelle.

Cluster, sondern sehr oft stoppt der Wachstumsprozeß, bevor der Cluster den Rand erreicht hat. Für $p < p_c$ tritt dieser Fall sehr häufig auf, während für $p > p_c$ fast nur noch perkolierende Cluster erzeugt werden.

Übung

Es sei s die Anzahl der Teilchen in einem Perkolationscluster. Die mittlere Clustermasse $\langle s \rangle$ der *endlichen* Cluster divergiert im unendlich großen System an der Perkolationsschwelle mit dem Potenzgesetz

$$\langle s \rangle \sim |p - p_c|^{-\gamma}.$$

Berechnen Sie den Mittelwert $\langle s \rangle$ für alle numerisch erzeugten Cluster, die nicht den Rand berühren. Tragen Sie diese Werte als Funktion der Konzentration p auf, und versuchen Sie, damit die kritische Konzentration p_c zu bestimmen. Berechnen Sie das Maximum dieser Funktion für verschiedene Werte der Gitterlänge L. Versuchen Sie, mit Hilfe von *finite size scaling* den kritischen Exponenten γ zu berechnen. Sie dürfen dabei den oben angegebenen Wert von ν benutzen.

Literatur

H. Gould, J. Tobochnik, *An Introduction to Computer Simulation Methods: Applications to Physical Systems, Part II*, Addison Wesley, 1988.

J. Schnakenberg, *Algorithmen in der Quantentheorie und Statistischen Physik*, Zimmermann-Neufang, 1995.

D. Stauffer, A. Aharony, *Perkolationstheorie: Eine Einführung*, VCH Verlagsgesellschaft, 1995.

5.4 Polymer-Ketten

Polymere spielen eine große Rolle sowohl in der Chemie als auch in der Biologie und der Medizin. Aber auch die Physik interessiert sich seit langem für allgemeine mathematische Gesetzmäßigkeiten der Eigenschaften von Polymeren. Schon ein einziges Molekül, das aus einer langen Kette von identischen Einheiten (*Monomeren*) aufgebaut ist, hat interessante Eigenschaften. Um solche Ketten von vielen tausend Monomeren mathematisch beschreiben zu können, muß man versuchen, die wesentlichen Mechanismen und Strukturen möglichst einfach zu modellieren. Ein bewährtes Modell dazu ist der zufällig erzeugte Weg, der in der Fachliteratur mit *random walk* bezeichnet wird. Der Weg besteht aus vielen kleinen Strecken, die in zufälligen Richtungen aneinandergereiht sind.

Im Abschnitt 3.3 haben wir schon festgestellt, daß ein Zufallsweg als Modell für ein Polymermolekül angesehen werden kann, bei dem in einem Wärmebad alle Konfigurationen gleich wahrscheinlich sind. Die Masse der Kette wächst dabei mit dem Quadrat ihrer mittleren Ausdehnung. Die mathematische Theorie dazu ist

sehr gut entwickelt. Allerdings wird bei diesem Modell ein wichtiger Mechanismus vernachlässigt: Die Kette darf sich nicht selbst durchdringen. Zufallswege mit einer solchen Einschränkung, die deshalb auch SAW's *(self-avoiding walks)* heißen, kann man nur näherungsweise analytisch berechnen. Sie lassen sich aber relativ leicht auf einem Computer simulieren.

Physik

Am einfachsten kann ein Zufallsweg auf einem Gitter definiert werden. Da wir uns nur für die globalen Eigenschaften sehr langer Polymere interessieren, insbesondere für die fraktale Dimension, können wir annehmen, daß es keine Rolle spielt, ob wir einen *random walk* mit kontinuierlichen Schrittweiten und Richtungen oder einen solchen auf dem Gitter betrachten. Im Abschnitt 3.3 haben wir gezeigt, daß ein Zufallsweg ohne Einschränkungen ein Knäuel eines Polymermoleküls beschreibt, dessen Masse wie das Quadrat der mittleren Ausdehnung wächst. Sei nämlich N die Anzahl der Schritte und R_N der Abstand der beiden Enden des Weges, dann gilt

$$N = \langle R_N^2 \rangle / a^2 \,. \tag{5.35}$$

Dabei ist $\langle ... \rangle$ ein Mittelwert über alle Zufallswege und a ist die Länge der Kettenglieder. Definiert man eine mittlere Länge durch $L = \sqrt{\langle R_N^2 \rangle}$, so erhält man

$$N \propto L^D \tag{5.36}$$

mit der Dimension $D = 2$. Dieses Ergebnis gilt nicht nur in der Ebene, sondern in allen Raumdimensionen.

Nun betrachten wir Zufallswege, die sich nicht selbst schneiden dürfen. Bei realen Polymeren ist es das Eigenvolumen der Ketten, das diese Durchdringung verbietet. Ein solcher *self-avoiding walk* wird offenbar im Vergleich zum *random walk* im Mittel nicht so stark verknäult sein, der mittlere End-zu-End-Abstand L wird größer werden. Das bedeutet, die Struktur des entsprechenden Polymers ist nicht mehr so kompakt und die Dimension D sollte kleiner sein. Tatsächlich findet man beim SAW für große N-Werte wieder ein Gesetz der Form (5.36), nun aber mit einer fraktalen Dimension D, die von der Raumdimension d abhängt. Schon 1949 hat Flory mit einer Art Molekularfeldnäherung eine Formel für D angegeben,

$$D = \frac{d+2}{3} \,. \tag{5.37}$$

Spätere Untersuchungen haben gezeigt, daß diese Formel in $d = 1, 2$ und 4 Raumdimensionen exakt ist und daß die numerischen Ergebnisse für $d = 3$ einen nur wenig höheren Wert für D geben. In $d = 4$ Raumdimensionen stimmt der Wert $D = 2$ mit

demjenigen des *random walk* überein, dort spielt also das Verbot des Selbstkreuzens keine Rolle mehr bei den globalen Eigenschaften. In allen höheren Raumdimensionen bleibt der Wert $D = 2$. Wie bei vielen anderen Phasenübergängen werden ab einer oberen kritischen Dimension $d = 4$ die kritischen Exponenten durch die Molekularfeldtheorie beschrieben.

Tatsächlich gibt es verdünnte Lösungen von Polymeren, bei denen bei hohen Temperaturen ein SAW-Gesetz wie in den Gleichungen (5.36) und (5.37) beobachtet wurde. Bei realen Molekül-Ketten gibt es auch attraktive Wechselwirkungen zwischen entfernten Monomeren eines einzelnen Polymermoleküls, so daß die Kette bei tiefen Temperaturen zu einem kompakten Knäuel mit $D = d$ kollabieren kann. Noch komplexer wird es natürlich, wenn viele Molekülketten miteinander wechselwirken. Gerade in den letzten Jahren haben extensive numerische Simulationen wesentlich zum Verständnis der Polymerdynamik in einer solchen „Spaghetti-Suppe“ beitragen können. Hier wollen wir uns dagegen mit der Simulation des relativ einfachen SAW befassen.

Algorithmus

Einen Zufallsweg auf einem Gitter kann man mit dem Computer leicht erzeugen. Bei jedem Schritt wählt man mit einer Zufallszahl einen der Nachbarplätze aus und macht dorthin einen weiteren Schritt. Auch beim Zufallsweg, der sich nicht überkreuzen darf, scheint der Algorithmus offensichtlich zu sein: Man wählt zufällig einen der bisher nicht besuchten Nachbarplätze aus und macht einen Schritt dorthin. Doch überraschenderweise gibt dieser Algorithmus falsche Resultate: Im Mittel wird das Polymer kompakter als das korrekte statistische Mittel. Ketten, die sich an mehreren Stellen berühren, werden öfter erzeugt als gestreckte Ketten mit wenigen Berührungspunkten.

Der Grund dafür soll hier kurz erläutert werden. Sei Z_i die Zahl der noch nicht besuchten Nachbarplätze am Ende einer Kette. Dann ist $1/Z_i$ die Wahrscheinlichkeit, daß ein Monomer an einem dieser Plätze hinzugefügt wird. Die gesamte Wahrscheinlichkeit W, eine spezielle Kettenkonfiguration mit N Gliedern zu erzeugen, ist damit

$$W_N = \prod_{i=1}^{N} \frac{1}{Z_i} \,. \tag{5.38}$$

Man sieht also, daß Ketten mit vielen Berührungspunkten (Z_i klein) eine hohe Wahrscheinlichkeit erhalten. Dies darf nicht sein, denn im korrekten statistischen Mittel muß jede erlaubte Konfiguration mit derselben Wahrscheinlichkeit auftreten.

Gleichung (5.38) zeigt aber auch sofort, wie man die unerwünschte Bevorzugung kompakter Ketten korrigieren kann. Man muß jeder Bindung das statistische Ge-

wicht Z_i geben und die Eigenschaften des Polymers damit wichten. Dadurch erhält jede Kette dasselbe statistische Gewicht

$$W_N \cdot \prod_{i=1}^{N} Z_i = 1\,, \tag{5.39}$$

und der korrekte Wert des mittleren End-zu-End-Abstandes L ist bei dieser Simulation also

$$L^2 = \langle R_N^2 \rangle = \frac{\sum\limits_l R_{N,l}^2 \prod\limits_{i=1}^{N} Z_{i,l}}{\sum\limits_l \prod\limits_{i=1}^{N} Z_{i,l}}\,, \tag{5.40}$$

wobei l die erzeugten Konfigurationen durchnumeriert.

Es gibt aber auch eine effektive Methode, um direkt Polymerkonfigurationen mit konstanter Wahrscheinlichkeit zu erzeugen. Während der vorherige Algorithmus Ketten wachsen läßt, arbeitet diese Methode, die mit *Reptation* bezeichnet wird, mit konstanter Kettenlänge N. Dabei „schlängelt" sich das Polymer über das Gitter und ändert auf diese Weise ständig seine Form und Richtung.

Bevor wir diesen Reptationsalgorithmus und seine Eigenschaften beschreiben, müssen wir kurz erläutern, was man unter einer Konfiguration des Polymers versteht und wann zwei solche Konfigurationen auf dem ebenen Quadratgitter als verschieden gelten. Es ist beispielsweise sinnvoll, alle diejenigen Konfigurationen als gleich anzusehen, die durch die möglichen Translationen einer gegebenen Form der Polymerkette entstehen. Zum Vergleich eventuell verschiedener Konfigurationen können wir diese dann so verschieben, daß alle an demselben festen Gitterpunkt beginnen. Dabei ist implizit schon die Festlegung getroffen worden, daß die Kette einen Anfang hat und wir also von der ersten Monomereneinheit, der zweiten, der dritten usw. reden können.

Eine weitere elementare Symmetrie des Quadratgitters sollten wir noch zur Vereinfachung nutzen, nämlich die $90°$-Drehungen. Die Richtung des ersten Monomers könnte nach Norden, nach Westen, nach Süden oder nach Osten zeigen.

Wenn wir von der Richtung des ersten Kettengliedes absehen, so ist zur eindeutigen Charakterisierung einer Konfiguration nur erforderlich zu wissen, ob nach dem ersten Schritt der nächste nach links, geradeaus oder nach rechts erfolgt, und dementsprechend für die folgenden Schritte. Wir können also eine Konfiguration eines Polymers, das aus N Gliedern besteht, durch eine Folge von $N-1$ Richtungsangaben beschreiben, wobei deren mögliche Werte *links, geradeaus* und *rechts* sind. Wichtig in diesem Zusammenhang ist, daß diese Folge in einer bestimmten Richtung durchlaufen wird. Wie oben schon gesagt, benötigt die Polymerkette eine Orientierung. Dazu markieren wir die Enden der Kette mit *Kopf* bzw. *Schwanz*

und verabreden z. B., daß die Orientierung vom Schwanzende zum Kopfende hin erfolgen soll.

Aber nicht jede Folge von $N-1$ Richtungsangaben stellt einen SAW dar, sondern es sind nur diejenigen Folgen erlaubt, die keine Selbstüberschneidungen aufweisen, eine Nebenbedingung, die sich nicht lokal formulieren läßt. Das ist auch der Grund dafür, daß es zur Statistik der SAWs bisher nur so wenige analytische Resultate gibt und man auf Computersimulationen angewiesen ist.

Der Algorithmus für die Reptation einer Polymerkette mit N Gliedern lautet folgendermaßen:

1. Starte mit einer Konfiguration auf dem Gitter, wobei die Enden der Kette mit *Kopf* bzw. *Schwanz* markiert sind.

2. Entferne das letzte Monomer am Schwanzende und wähle am Kopfende einen der Nachbarplätze zufällig aus (drei Möglichkeiten auf dem Quadratgitter).

3. Falls dieser Platz unbesetzt ist, füge dort ein neues Kopfmonomer hinzu. Falls dieser Platz besetzt ist, restauriere die alte Konfiguration, vertausche die Markierungen *Kopf* und *Schwanz* und berücksichtige die ansonsten unveränderte Konfiguration als neue Konfiguration bei der Mittelwertbildung.

4. Iteriere 2 und 3.

Man kann zeigen, daß diese Methode alle Konfigurationen, die überhaupt aus der Startkonfiguration entstehen können, mit der gleichen Wahrscheinlichkeit erzeugt. Dazu numerieren wir die verschiedenen erreichbaren Zustände mit $l = 1, 2, 3, \ldots, \mathcal{N}$ durch und bezeichnen mit $p_l(t)$ die Besetzungswahrscheinlichkeiten, die sich nach entsprechend langer Zeit einstellen. Wenn wir zudem noch die Übergangswahrscheinlichkeiten $W(l \to k)$ kennen, mit der pro Zeitschritt Konfiguration l in Konfiguration k transformiert wird, so können wir für die Zeitentwicklung der $p_l(t)$ folgende Mastergleichung aufschreiben:

$$\dot{p}_l = \sum_{k=1}^{\mathcal{N}} \left(W(k \to l)\, p_k - W(l \to k)\, p_l \right) . \tag{5.41}$$

Wegen $\sum_k W(l \to k) = 1$ läßt sich die Summation im zweiten Term ausführen. Weil wir an der stationären Verteilung interessiert sind, d. h. $\dot{p}_l = 0$, suchen wir also die Lösung des Gleichungssystems

$$\sum_{k=1}^{\mathcal{N}} W(k \to l)\, p_k = p_l . \tag{5.42}$$

Bei manchen Problemen dieser Art sind die Übergangswahrscheinlichkeiten $W(l \to k)$ mit Hilfe einer stationären Verteilung w_k konstruiert, und zwar so, daß die Gleichung

$$W(k \to l)\, w_k = W(l \to k)\, w_l \tag{5.43}$$

gilt. Man spricht dann von detailliertem Gleichgewicht (*detailed balance*), und aus Gleichung (5.41) ersieht man sofort, daß dann $p_k = w_k$ eine Lösung der stationären Mastergleichung ist.

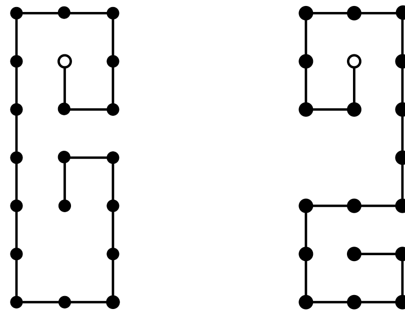
Detailed balance gilt aber für unseren Fall nicht, wie man sich an folgendem Beispiel klarmacht: Die völlig gestreckte Konfiguration hat mit Wahrscheinlichkeit $1/3$ nach dem nächsten Zeitschritt am Kopfende einen Rechtsknick, während die Wahrscheinlichkeit, daß dieser neue Zustand wieder in den gestreckten übergeht, null ist.

Statt des detaillierten Gleichgewichts gilt aber für den Reptationsalgorithmus die folgende Beziehung:

$$W(k \to l) = W(l_{\mathsf{inv}} \to k_{\mathsf{inv}})\,, \tag{5.44}$$

wobei wir mit l_{inv} den Zustand bezeichnen, den man aus l durch Vertauschung von *Kopf* und *Schwanz* erhält. Für $l = k_{\mathsf{inv}}$ ist die Gleichung (5.44) trivialerweise erfüllt. Das betrifft alle die Fälle, in denen der Algorithmus eine Kopf-Schwanz-Vertauschung verlangt. Für die anderen Übergänge gilt folgendes: Wenn es möglich ist, am Schwanzende ein Monomer zu entfernen und dafür am Kopfende eines hinzuzufügen, so ist auch der umgekehrte Prozeß möglich, nämlich dieses neue Kopfmonomer zu entfernen und es ans Schwanzende zurückzulegen. Daraus schließen wir, daß beide Seiten von Gleichung (5.44) entweder gleich null sind oder ungleich null sind. Wenn sie aber ungleich null sind, so sind sie beide gleich $1/3$, denn das Hinzufügen eines neuen Kopfmonomers einer bestimmten Richtung geschieht, wenn es überhaupt möglich ist, immer mit der Wahrscheinlichkeit $1/3$. Verwenden wir (5.44) in Gleichung (5.42), so erhalten wir ähnlich wie oben, daß $p_l = \text{const.} = 1/\mathcal{N}$ eine Lösung ist, denn natürlich ist auch $\sum_k W(l_{\mathsf{inv}} \to k_{\mathsf{inv}}) = 1$. D. h., die erreichbaren Konfigurationen werden mit gleicher Wahrscheinlichkeit erzeugt.

Allerdings gibt es ganze Klassen von Konfigurationen, die mit dem Reptationsalgorithmus nicht erreichbar sind, wenn man als Startzustand den völlig gestreckten oder einen äquivalenten benutzt. Das Bild 5.8 zeigt Beispiele aus zwei verschiedenen solcher Klassen. Wir können aber davon ausgehen, daß die Anzahl dieser Konfigurationen klein ist im Vergleich zu den Zuständen in der Hauptklasse. Andererseits hängt es natürlich von der Fragestellung ab, ob die anderen Konfigurationen relevant sind oder nicht. Wenn wir mit dem Reptationsalgorithmus den mittleren End-zu-End-Abstand berechnen, so werden wir einen geringfügig zu großen Wert erwarten, weil wir die stärker eingerollten Zustände der anderen Klassen nicht berücksichtigt haben.



5.8 Polymerkonfigurationen, die nicht von der gestreckten Startkonfiguration aus erreichbar sind. Der offene Kreis bezeichnet den Kopf.

Wir wollen uns das schlängelnde Polymer auf dem Computerbildschirm erzeugen. Dazu programmieren wir den Reptationsalgorithmus auf dem Quadratgitter in der Sprache C. Zunächst deklarieren wir den Variablentyp `vector`, der Strukturen mit den Ortskoordinaten (x, y) definiert.

```
typedef struct { float x,y;} vector;
 vector richtung[3], polymer[NMAX];
```

`polymer[NMAX]` ist ein Feld von Vektoren, das alle Ortskoordinaten des Polymers enthält; `polymer[5].y` gibt z. B. die y-Koordinate des 5-ten Monomers an. Am Anfang wird als Konfiguration für das Polymer die gestreckte Kette festgelegt und mit der Funktion `kreis` gezeichnet.

```
for(i=0;i<N;i++)
 {
  polymer[i].x=i-N/2;
  polymer[i].y=0;
  kreis(polymer[i]);
 }
```

Jede andere äquivalente Startkonfiguration ist genauso möglich. Die Markierungen *Kopf* und *Schwanz* werden am Anfang auf die Plätze $(N - 1)$ bzw. 0 gelegt, und bei jeder Bewegung werden diese Zeiger um den Wert 1 (modulo N) erhöht.

Die Funktion `wahl()` wählt eine der drei möglichen Nachbarplätze des Kopfes zufällig aus (Punkt 2). Die Funktion `kreuzung(w)` prüft, ob der Nachbarplatz durch die Kette besetzt ist. Falls ja, wird *Kopf* und *Schwanz* vertauscht, falls nein, wird der neue *Kopf* akzeptiert und ein Schwanzelement überschrieben. Punkt 3 lautet also im Programm:

```
while(!done)
  {
    event();
    w=wahl();
    if(kreuzung(w))
          { kopf=schwanz; incr=-incr;
            schwanz=(kopf+incr+N)%N;
          }
     else akzeptiere(w);
  }
```

Anstatt das Polymer auf seinem Speicherplatz zu verschieben, verwenden wir Indizes für den *Kopf* und den *Schwanz*. Die Variable `incr = ± 1` gibt die Richtung des Polymers im Feld `polymer[N]` an. Um negative Indizes zu vermeiden, wird bei der Modulo-Operation `%` noch die Zahl N addiert.

Die Funktion `event()` fragt wie in jedem unserer C-Programme Tasten- bzw. Maus-Aktionen des Beobachters ab und gibt eventuell `done=1` zurück, wodurch die `while`-Schleife beendet wird.

Die Funktion `wahl()` gibt einen (x, y)-Vektor zurück, der vom *Kopf* auf einen der drei Nachbarplätze zeigt. Zunächst werden die drei möglichen Richtungen erzeugt, indem die Differenz ($\boldsymbol{r}$(*Kopf*) $-\,\boldsymbol{r}$(*Hals*)) gebildet wird (`richtung[0]`) und dann die beiden Vektoren senkrecht dazu berechnet werden. Wenn z. B. der Vektor `richtung[0]=` $(0,1)$ ist, so enthalten die Variablen `richtung[1]` und `richtung[2]` die Vektoren $(1,0)$ bzw. $(-1,0)$. Mit einer gleichverteilten Zufallszahl aus dem Intervall `[0,3)` wird dann durch Umwandlung in eine ganze Zahl r eine der drei Richtungen zufällig ausgewählt und zurückgegeben.

```
vector wahl()
{
  int km,r;
  km=(kopf-incr+N)%N;
  richtung[0].x=polymer[kopf].x-polymer[km].x;
  richtung[0].y=polymer[kopf].y-polymer[km].y;
  richtung[1].x=richtung[0].y;
  richtung[1].y=richtung[0].x;
  richtung[2].x=-richtung[1].x;
  richtung[2].y=-richtung[1].y;
  r=random(3);
  return richtung[r];
}
```

Die Funktion `kreuzung(w)` gibt den Wert 1 bzw. 0 zurück, falls der gewählte Nachbarplatz auf den alten Kettengliedern liegt bzw. nicht liegt. Dies läßt sich

folgendermaßen leicht überprüfen:

```
int kreuzung(vector w)
{
  int i;
  for(i=0;i<N;i++)
    if(w.x+polymer[kopf].x==polymer[i].x  &&
       w.y+polymer[kopf].y==polymer[i].y &&
       i != schwanz)  return 1;
  return 0;
}
```

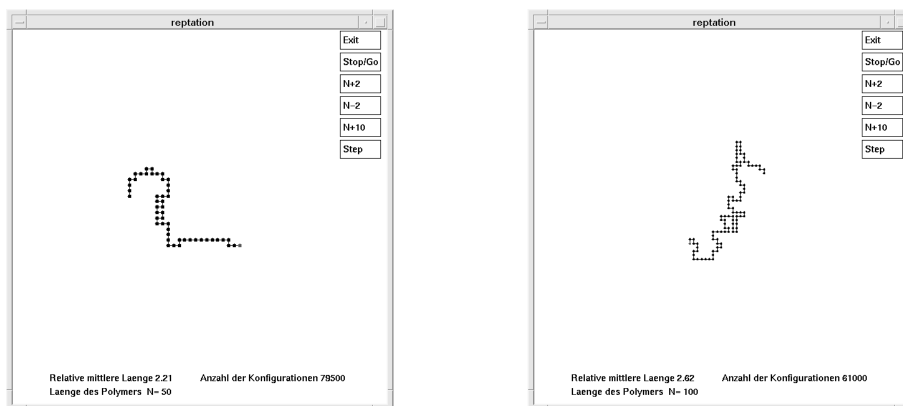
Ohne Kreuzung wird der gewählte Platz als neuer *Kopf* akzeptiert und die Indizes werden um den Wert `incr` verschoben. Auf dem Bildschirm wird der Kreis am Ort des Schwanzes mit Hintergrundfarbe übermalt und der neue *Kopf* wird als Kreis hinzugezeichnet. So entsteht eine Schlangenbewegung auf dem Gitter. All dies macht folgende Funktion:

```
void akzeptiere (vector w)
{
 int kp;
 void verschiebe();
 setcolor(BLACK);
 kreis(polymer[schwanz]);
 kp=(kopf+incr+N)%N;
 polymer[kp].x=w.x+polymer[kopf].x;
 polymer[kp].y=w.y+polymer[kopf].y;
 kopf=kp;
 schwanz=(kopf+incr+N)%N;
 setcolor(WHITE);
 kreis(polymer[kopf]);
 if(abs(polymer[kopf].x)>2*N ||
    abs(polymer[kopf].y)>2*N) verschiebe()
}
```

Um den mittleren End-zu-End-Abstand L zu berechnen, müssen wir nach jedem Schritt, auch nach der Richtungsumkehr, den Betrag des Differenzvektors $\boldsymbol{r}$(*Kopf*) $-$ $\boldsymbol{r}$(*Schwanz*) berücksichtigen, alle Beträge aufsummieren und schließlich durch die Anzahl der Reptationsschritte dividieren. Zusätzlich haben wir noch eine Funktion `verschiebe()` definiert, die das Polymer in die Mitte schiebt, wenn es das Fenster verlassen will.

Ergebnisse

Der Aufruf unseres C-Programms erzeugt das Polymer auf dem Bildschirm (Bild 5.9). Gleichzeitig wird der mittlere End-zu-End-Abstand ausgedruckt, und zwar relativ zum Ergebnis $L = \sqrt{N}$ des *random walk* ohne Einschränkungen. Man erkennt, daß das Polymer im Mittel gestreckter ist als der *random walk* und daß diese relative mittlere Länge mit N anwächst, in Übereinstimmung mit der kleineren fraktalen Dimension $D = 4/3$. Tatsächlich ergibt schon der Fit aus den beiden Werten für $N = 50$ und $N = 100$ recht genau, daß das Verhältnis der Längen, wie es die Theorie vorhersagt, proportional zu $N^{0.25}$ anwächst.



5.9 Polymerketten der Längen $N = 50$ und $N = 100$ auf dem Quadratgitter.

Übung

Programmieren Sie den anfangs erwähnten Algorithmus, indem Sie, wie in den Gleichungen (5.39), (5.40) angegeben, die Anzahl Z_i der bei jedem Schritt verfügbaren freien Plätze in korrekter Weise berücksichtigen. Vergleichen Sie diese Implementierung mit dem Reptationsalgorithmus in bezug auf die Rechengeschwindigkeit und den Wert für den mittleren End-zu-End-Abstand.

Literatur

K. Binder, D. W. Heermann, *Monte Carlo Simulation in Statistical Physics*, Springer Verlag, 1992.

H. Gould, J. Tobochnik, *An Introduction to Computer Simulation Methods: Applications to Physical Systems, Parts I and II*, Addison Wesley, 1988.

5.5 Ising-Ferromagnet

Wenn sehr viele Teilchen zusammenwirken, können qualitativ neue Eigenschaften entstehen. Zum Beispiel werden Gase bei tiefen Temperaturen oder hohem Druck plötzlich flüssig, Flüssigkeiten werden zum Festkörper, magnetische Atome ordnen sich zu einem magnetischen Material und Metalle verlieren plötzlich ihren elektrischen Widerstand. Es gibt sehr viele solcher Systeme, die im Wärmegleichgewicht bei einer wohldefinierten Temperatur ihre makroskopischen Eigenschaften ändern. In der mathematischen Modellierung treten diese Phasenübergänge nur im unendlich großen System auf.

Kann man mit dem Computer ein Wärmegleichgewicht simulieren und ein System langsam abkühlen? Kann die Simulation endlich großer Systeme Phasenumwandlungen beschreiben? Mit solchen Fragen wollen wir uns in diesem Abschnitt befassen, exemplarisch wieder an einem einfachen Modell, dem Ising-Ferromagneten auf dem Quadratgitter. Dieses Modell hat am Phasenübergang universelle kritische Eigenschaften, die auch bei vielen anderen Modellen und bei realen Materialien gemessen werden. Wir lassen die mikroskopischen Magnete auf dem Bildschirm flackern und heizen dabei mit einem Tastendruck den Magneten auf oder kühlen ihn ab.

Physik

Schon seit Anfang dieses Jahrhunderts ist bekannt, wie man das Wärmegleichgewicht von wechselwirkenden Teilchen mathematisch beschreiben kann. In der kanonischen Formulierung führt die statistische Mechanik die thermischen Eigenschaften auf die Berechnung einer Zustandssumme Z zurück,

$$Z = \sum_{\underline{S}} e^{-H(\underline{S})/k_\mathrm{B}T} \,. \tag{5.45}$$

Z ist die Summe über alle möglichen Vielteilchenzustände $\underline{S}$, und jeder Zustand wird entsprechend seiner Energie $H(\underline{S})$ gewichtet. T ist die Temperatur und k_B die Boltzmann-Konstante.

Wir wollen das Ising-Modell beschreiben, ein einfaches Modell für ein magnetisches System. Dazu betrachten wir ein Quadratgitter, auf dessen Gitterplätzen Spins sitzen, die nur zwei Einstellmöglichkeiten haben. Außerdem gibt es zwischen benachbarten Spins eine Wechselwirkung. Mathematisch läßt sich das folgendermaßen formulieren: An jedem Gitterplatz $i = 1, ..., N$ gibt es eine Variable $S_i \in \{+1, -1\}$, so daß ein Vielteilchenzustand durch $\underline{S} = (S_1, S_2, ..., S_N)$ gekennzeichnet ist. In die Berechnung der Gesamtenergie $H(\underline{S})$ geht eine Summation

sowohl über alle Nachbarpaare $(i,j)_{nn}$ als auch über alle Plätze i ein:

$$H = -J \sum_{(i,j)_{nn}} S_i S_j - h \sum_i S_i \, . \tag{5.46}$$

Dies bedeutet, parallele Nachbar-Spins haben die Paar-Wechselwirkungs-Energie $-J < 0$, während der Term proportional zu h ein äußeres Magnetfeld beschreibt, in dem jeder zum Feld parallele Spin die Energie $-h < 0$ hat.

Das Problem ist damit wohldefiniert: Wir brauchen „nur" die Summe (5.45) auszurechnen. Wir wollen dies mit einem Computer versuchen. Nehmen wir an, wir hätten 100 Stunden Rechenzeit auf einer Maschine zur Verfügung, die für einen Rechenschritt etwa 10^{-6} s benötigt. Insgesamt können wir also 3.6×10^{11} Rechenschritte nutzen, wobei wir von einer schnellen Maschine ausgegangen sind, wenn man beachtet, daß ein Rechenschritt, also z. B. die Auswertung der exp-Funktion, viele Zeittakte benötigt.

Wieviele Rechnungen brauchen wir bei N Spins für die Auswertung der Gleichung (5.45)? Da jeder Spin S_i zwei Einstellmöglichkeiten hat, gibt es insgesamt 2^N verschiedene Spinkonfigurationen. Bei jedem dieser Zustände muß die Energie H aus Gleichung (5.46) berechnet werden. Dies kostet $2N$ Rechenschritte. Daraus folgt

$$2N \, 2^N = 3.6 \cdot 10^{11} \tag{5.47}$$

mit der Lösung $N \simeq 32$. Das Ergebnis ist entmutigend. Selbst mit heutigen Supercomputern können wir nur einen winzigen Magneten der Größe $3 \times 3 \times 3$ berechnen, dabei wollen wir ein reales Material mit $10^7 \times 10^7 \times 10^7$ Spins beschreiben!

Mit Transfermatrix-Methoden gelingt es, die Zustandssumme Z für Stäbe der Größe $4 \times 4 \times \infty$ exakt zu berechnen. Allerdings ist dies immer noch sehr wenig, wenn wir Phasenübergänge beobachten wollen.

Man könnte nun auf die Idee kommen, nur einige wenige zufällig erzeugte Zustände $\underline{S} = (S_1, S_2, S_3, \ldots, S_N)$ in der Zustandssumme Gleichung (5.45) zu berücksichtigen. Das funktioniert aber nicht. Denn zufällig erzeugte Zustände haben nach dem zentralen Grenzwertsatz für große N-Werte die Energie $H \simeq 0 + \mathcal{O}(\sqrt{N})$. Die physikalisch wichtigen Zustände haben aber eine Energie der Größenordnung $\mathcal{O}(N)$, sie werden also bei der obigen Methode überhaupt nicht erzeugt.

Schon zu den Zeiten der ersten Computer haben sich Physiker überlegt, wie man die physikalisch relevanten Zustände erzeugen kann. Dazu startet man mit einer Konfiguration $\underline{S}(t = 0)$ und erzeugt daraus eine Folge $\underline{S}(1), \underline{S}(2), ..., \underline{S}(t)$, die in das thermische Gleichgewicht relaxiert. Dazu wird eine Übergangswahrscheinlichkeit $W(\underline{S} \to \underline{S}')$ definiert, die – man vergleiche mit dem vorigen Abschnitt – folgendem *detaillierten Gleichgewicht* genügt

$$W(\underline{S} \to \underline{S}') \, e^{-H(\underline{S})/k_{\mathrm{B}}T} = W(\underline{S}' \to \underline{S}) \, e^{-H(\underline{S}')/k_{\mathrm{B}}T} \, . \tag{5.48}$$

Diese Gleichung kann man sich folgendermaßen klarmachen: Im Gleichgewicht ist

$$P(\underline{S}) = \frac{1}{Z}\, e^{-H(\underline{S})/k_\mathrm{B}T} \tag{5.49}$$

die Wahrscheinlichkeit, einen Zustand $\underline{S}$ vorzufinden. Die Forderung (5.48) nach detaillierter Balance hat zur Folge, daß bei der durch W definierten Dynamik das thermische Gleichgewicht $P(\underline{S})$ ein stationärer Zustand ist. Die Besetzungswahrscheinlichkeit $P(\underline{S})$ ändert sich nicht mit der Zeit. Wie im Abschnitt 5.4 kurz beschrieben, kann man dies mathematisch präzise mit einer Mastergleichung formulieren und beweisen.

Im Algorithmus-Teil wird eine Funktion $W(\underline{S} \to \underline{S}')$ definiert, die der Gleichung (5.48) genügt und sehr einfach zu programmieren ist. Im Prinzip gibt uns Gleichung (5.48) noch viele Freiheiten zur Wahl von W, wir müssen nur beachten, daß W auch durch den ganzen Konfigurationsraum (= Menge der Zustände $\underline{S}$) führen kann. Zwei der gängigsten Algorithmen, die beide Bedingungen erfüllen, sind als *Metropolis-Algorithmus* bzw. *Heat-Bath-Algorithmus* bekannt.

Der durch W definierte stochastische Prozeß führt zu einer Folge von physikalisch relevanten Konfigurationen $\underline{S}(0), \underline{S}(1), ..., \underline{S}(t)$, die z. B. bis auf Fluktuationen die richtige Energie und Magnetisierung ergeben. Mit diesen Zuständen $\underline{S}(t)$ können Mittelwerte von Größen $A(\underline{S})$ berechnet werden.

$$\langle A \rangle_{t_0,t_1} = \frac{1}{t_1} \sum_{t=t_0}^{t_0+t_1} A(\underline{S}(t))\,. \tag{5.50}$$

Im Grenzwert $t_0 \to \infty$ und $t_1 \to \infty$ stimmt dieses Zeitmittel mit dem statistischen Mittel

$$\langle A \rangle = \sum_{\underline{S}} P(\underline{S})\; A(\underline{S}) \tag{5.51}$$

überein. In der Praxis kann man aber nur mit endlichen Werten für t_0 und t_1 rechnen. t_0 muß dabei so groß sein, daß das System aus dem eventuell unphysikalischen Startzustand $\underline{S}(0)$ ins thermische Gleichgewicht relaxiert ist, und t_1 muß so groß sein, daß die statistischen Fluktuationen von $\langle A \rangle_{t_0,t_1}$ klein sind. Beides hängt vom Modell, seinen Parametern und der Systemgröße ab.

Mit den heutigen Computern kann man etwa folgende Werte erreichen:

$$N \simeq 10^6 ... 10^{12}\,,\; t_0 \simeq t_1 \simeq (10^4 ... 10^6) N\,. \tag{5.52}$$

Daher werden nur etwa 10^{10} bis 10^{15} Konfigurationen $\underline{S}(t)$ erzeugt, während das Modell 2^N, also $10^{300\,000}$ bis $10^{300\,000\,000\,000}$ verschiedene Zustände $\underline{S}$ hat. Welchen Algorithmus man auch immer verwendet, er kann nur einen winzigen Bruchteil aller möglichen Konfigurationen erzeugen. Aber er muß die „richtigen“, die typischen bzw. die physikalischen Zustände erzeugen.

Bevor wir zur Programmierung des durch Gleichung (5.48) beschriebenen stochastischen Prozesses kommen, wollen wir kurz die Eigenschaften des Ising-Ferromagneten und deren Abhängigkeit von der Systemgröße N diskutieren. Das unendlich große System ($N \to \infty$) hat ohne äußeres Magnetfeld ($h = 0$) einen Phasenübergang in zwei und mehr Raumdimensionen. Unterhalb einer kritischen Temperatur T_c ordnen sich im thermischen Mittel die Spins parallel zueinander und bilden eine makroskopische Magnetisierung M,

$$M(T) = \frac{1}{N} \sum_i \langle S_i \rangle \,. \tag{5.53}$$

M kann positiv oder negativ sein; das System wählt durch zufällige Fluktuationen beim Abkühlen eine Richtung aus. Da für $h = 0$ die Energie symmetrisch in M ist, $H(\underline{S}) = H(-\underline{S})$, nennt man diesen Vorgang *spontane Symmetriebrechung*. Ein äußeres Magnetfeld $h \neq 0$ hebt diese Symmetrie auf und zerstört dadurch den Phasenübergang.

Der Phasenübergang ist kritisch, d. h. die Magnetisierung $M(T)$ geht mit wachsender Temperatur stetig gegen den Wert $M(T) = 0$ für $T \geq T_c$. Wie bei der Perkolation aus Abschnitt 5.3 gibt es eine divergierende Korrelationslänge $\xi(T)$, die durch den Zerfall der Spinkorrelationen für große Abstände r definiert ist:

$$\langle S_i S_j \rangle \sim \exp(-r/\xi) \;\text{ mit }\; r = |\boldsymbol{r}_i - \boldsymbol{r}_j| \,. \tag{5.54}$$

In der Nähe des Phasenübergangs $T \simeq T_c, h \simeq 0$, gibt es wieder Skalengesetze und Potenzsingularitäten mit universellen kritischen Exponenten. Bei T_c gelten folgende Beziehungen für die Magnetisierung M, die Korrelationslänge ξ, die magnetische Suszeptibilität χ und die spezifische Wärme C:

$$\begin{aligned} M(T) &\sim (T_c - T)^{\beta} \;\; (T \nearrow T_c) \\ \chi(T) &\sim |T_c - T|^{-\gamma} \\ \xi(T) &\sim |T_c - T|^{-\nu} \\ C(T) &\sim |T_c - T|^{-\alpha} \,. \end{aligned} \tag{5.55}$$

Die Begründung für diese universellen Eigenschaften gibt die mathematisch anspruchsvolle Theorie der *Renormierungsgruppe* aus den siebziger Jahren (Nobelpreis für K. G. Wilson im Jahre 1982). Das zweidimensionale Modell (für $h = 0$) wurde 1944 von Onsager gelöst. Die universellen Größen sind daher für $d = 2$ exakt bekannt, und zwar gibt die Onsager-Lösung

$$\begin{aligned} T_c &= \frac{J}{k_B} \frac{2}{\ln\left(\sqrt{2}+1\right)} \simeq 2.269 ... \frac{J}{k_B} \,, \\ \beta &= \frac{1}{8},\, \nu = 1,\, \gamma = \frac{7}{4},\, \alpha = 0 \,. \end{aligned} \tag{5.56}$$

Die spezifische Wärme C divergiert wie $\ln|T - T_\mathrm{c}|$. Der Wert $\alpha = 0$ bedeutet also nicht, daß die spezifische Wärme bei T_c endlich bliebe, sondern nur, daß sie schwächer als jede Potenz von $|T - T_\mathrm{c}|$ divergiert. Der Wert für T_c gilt nur für das Ising-Modell auf dem Quadratgitter, die Exponenten hingegen gelten für alle zweidimensionalen Systeme, deren kritischer Phasenübergang dieselben Symmetrien wie das Ising-Modell hat.

Bei der Perkolation haben wir schon gesehen, daß man von einem endlich großen auf das Verhalten des unendlichen Systems schließen kann. Dieses *finite size scaling*, das ebenfalls mit der Renormierungstheorie begründet wird, gilt auch für den Ising-Ferromagneten. Ein endliches System hat keinen Phasenübergang in eine magnetische Ordnung, aber aus der N-Abhängigkeit von χ und C kann man die kritischen Exponenten berechnen.

Sei L die Kantenlänge eines Quadrates mit $N = L^2$ Spins. Analog zur Gleichung (5.27) gilt asymptotisch dicht bei T_c:

$$\chi(T, L) = |T - T_\mathrm{c}|^{-\gamma} f\left(|T - T_\mathrm{c}|L^{1/\nu}\right) . \tag{5.57}$$

Setzt man $|T - T_\mathrm{c}|\ L^{1/\nu} = \mathrm{const.}$, so erhält man

$$\chi(T_\mathrm{c}, L) \sim L^{\gamma/\nu} . \tag{5.58}$$

Aus der Divergenz der Suszeptibilität als Funktion der Systemgröße erhält man also den Wert γ/ν. Entsprechend findet man aus der Divergenz der spezifischen Wärme $C(T_\mathrm{c}, L)$ den Exponenten α/ν. Aus diesen beiden Werten kann man mit Hilfe der Skalengesetze alle anderen Exponenten berechnen. Es gilt (d = Raumdimension $2, 3$ oder 4)

$$\nu = \frac{2}{d + \alpha/\nu} , \quad \gamma = \frac{\gamma}{\nu} \frac{2}{d + \alpha/\nu} , \quad \beta = \frac{d - \gamma/\nu}{d + \alpha/\nu} . \tag{5.59}$$

Algorithmus

Der stochastische Prozeß benötigt eine Übergangswahrscheinlichkeit $W(\underline{S} \to \underline{S}')$, die dem detaillierten Gleichgewicht (5.48) genügt. Wir wollen hier eine einfache Methode verwenden, und zwar den Metropolis-Algorithmus. W soll nur Übergänge erlauben, bei denen höchstens ein Spin geändert wird. Sei also

$$\underline{S} = (S_1, S_2, ..., S_i, ..., S_N) ,$$

und

$$\underline{S}' = (S_1, S_2, ..., -S_i, ..., S_N) . \tag{5.60}$$

Außerdem sollen Übergänge in energetisch günstigere Zustände immer stattfinden $(W = 1)$. Mit den Gleichungen (5.48) und (5.49) gilt damit

$$W(\underline{S} \to \underline{S}') = \begin{cases} 1 & \text{für } H(\underline{S}') < H(\underline{S}) \\ \exp(\frac{H(\underline{S})-H(\underline{S}')}{k_\mathrm{B}T}) & \text{sonst .} \end{cases} \tag{5.61}$$

$\Delta E = H(\underline{S}) - H(\underline{S}')$ läßt sich einfach berechnen. Es gilt nämlich:

$$H(\underline{S}) = -JS_i \sum_{j\in\mathcal{N}(i)} S_j - h\,S_i + \mathit{Rest}\,, \tag{5.62}$$

wobei die j-Summe nur über die Nachbarn von i läuft und *Rest* denjenigen Anteil von H enthält, der nicht von S_i abhängt. $\underline{S}'$ hat dieselben Spins $S_j' = S_j$ bis auf $S_i' = -S_i$, also hat $H(\underline{S}')$ auch denselben Rest. Daraus folgt

$$\Delta E = -2S_i \left(J \sum_{j\in\mathcal{N}(i)} S_j - h \right) = -2S_i h_i\,. \tag{5.63}$$

h_i nennt man das innere Feld des Spins S_i. Damit können wir den Metropolis-Algorithmus schließlich formulieren:

1. Wähle einen Startzustand $\underline{S}(0) = (S_1, ..., S_N)$.
2. Wähle ein i (zufällig oder sequentiell) und berechne $\Delta E = -2S_i h_i$.
3. Wenn $\Delta E \geq 0$, dann drehe den Spin $S_i \to -S_i$. Wenn $\Delta E < 0$, dann ziehe eine gleichverteilte Zufallszahl $r \in [0,1]$. Wenn $r < \exp(\Delta E/k_\mathrm{B}T)$, dann $S_i \to -S_i$, sonst berücksichtige die alte Konfiguration noch einmal.
4. Iteriere 2 und 3.

Dieser Algorithmus gilt nicht nur für das Ising-Modell, sondern ganz allgemein für alle Modelle der statistischen Mechanik. Wir werden ihn auch bei schwierigen Optimierungsproblemen im nächsten Abschnitt anwenden. Weil diese Methode Zufallszahlen benutzt, nennt man sie *Monte-Carlo-Simulation*.

Der obige Algorithmus ist nicht die einzige Möglichkeit, detailliertes Gleichgewicht zu erfüllen und damit thermisches Gleichgewicht zu erreichen. Gerade in den letzten Jahren gab es eine Vielzahl von neueren Entwicklungen zu der seit vier Jahrzehnten bekannten Methode. So kann man große, geeignet ausgewählte Cluster von Spins gleichzeitig drehen und dadurch, insbesondere bei T_c, die Simulation erheblich beschleunigen. Diese Methode kann man noch mit einer hierarchischen Überstruktur, die man aus den Mehrgitter-Verfahren (*Multi Grid Method*) der Informatik kennt, beschleunigen. Ein weiteres Verfahren nutzt die Informationen über

die Fluktuation der Energie und der Magnetisierung aus, um mit einer Simulation Eigenschaften eines ganzen Temperaturintervalls zu erhalten. Man kann auch Simulationen bei konstanter Energie (mikrokanonisch) durchführen, oder mit einem Trick hohe Energiebarrieren überwinden. Der Metropolis-Algorithmus dagegen ist einfach und universell, ein Grund, weshalb wir ihn hier programmieren wollen.

Wir sind daran interessiert, die Systemgröße N, die Relaxationszeit t_0 und die Meßzeit t_1 so groß wie nur möglich zu machen. Wir brauchen also eine schnelle Computersprache. In der aktuellen Forschung und für die Computersimulation großer Systeme wird meistens die Sprache FORTRAN benutzt, da die FORTRAN-Compiler der Superrechner den höchsten Optimierungsgrad haben. Aber auch C-Programme laufen auf Höchstleistungsrechnern effektiv. Wir wollen im folgenden die Monte-Carlo-Simulation des Ising-Ferromagneten in C programmieren.

Die Spins S_i werden auf einem quadratischen Feld `s[x][y]`, `x` $= 1, \ldots, L$, `y` $= 1, \ldots, L$, gespeichert und mit den Werten $S_i = 1$ initialisiert. Man könnte jeden beliebigen Startzustand nehmen, da das System immer ins thermische Gleichgewicht relaxiert. Bei tiefen Temperaturen können andere Startzustände aber lange Relaxationszeiten haben. Die Diffusion von Domänenwänden – im Ergebnisteil ist ein Bild davon zu sehen – macht dies deutlich.

Randeffekte können einen starken Einfluß auf die Eigenschaften des Magneten haben, insbesondere bei kleinen Systemen. Daher verwenden wir hier periodische Randbedingungen, d. h. jeder Randspin wechselwirkt mit dem Spin auf der gegenüberliegenden Seite. Damit hat das System keinen Rand, sondern es ist topologisch zu einem Torus (Autoreifen) aufgewickelt. Um in der innersten Schleife möglichst viel Rechenzeit einzusparen, verzichten wir darauf, mit `if`-Abfragen oder `Modulo`-Rechnungen die Randindizes festzustellen. Statt dessen kopieren wir die Werte der Randspins nach jedem Durchlauf in die vier Zusatzfelder `s[0][i]`, `s[L+1][i]`, `s[i][0]` und `s[i][L+1]` mit `i` $= 1, \ldots, L$.

Der Algorithmus vergleicht Zufallszahlen mit $\exp(-2S_i\, h_i / k_\mathrm{B} T)$, dem Boltzmann-Gewicht bei der Temperatur T. Mit einem kleinen Trick können wir die ständige Berechnung der e-Funktion vermeiden. Wir führen ihn für $h = 0$ vor, er funktioniert aber auch für $h \neq 0$. Auf dem Quadratgitter kann nämlich die Summe $S_i \sum_{j \in \mathcal{N}(i)} S_j$ über die Nachbarn von S_i nur die Werte $-4, -2, 0, 2$ und 4 annehmen. Bei den negativen Werten wird S_i immer gedreht. Deshalb gibt es nur zwei verschiedene Boltzmann-Gewichte $\exp(-4/T)$ und $\exp(-8/T)$, wobei die Temperatur T in Einheiten der Wechselwirkungsenergie J/k_B gemessen wird. Beide Zahlen werden vor den Monte-Carlo-Schritten mit folgender Funktion berechnet:

```
void setT(double t)
  {
    temp=t;
    bf[0]= 0.5*RAND_MAX;
```

```
    bf[1]= exp(-4./temp)*RAND_MAX;
    bf[2]= exp(-8./temp)*RAND_MAX;
  }
```

Die Variable `temp` und das Feld `bf[...]` müssen natürlich vor dem Hauptprogramm `main()` deklariert sein, sonst kennt die Funktion `setT()` diese Größen nicht. Falls das innere Feld den Wert Null hat, verwenden wir die Übergangswahrscheinlichkeit $W(\underline{S} \to \underline{S}') = W(\underline{S}' \to \underline{S}) = 1/2$ (anstatt 1). Dies vermeidet Oszillationen, die bei der sequentiellen Wahl der besuchten Plätze auftreten können.

Mit den so definierten Boltzmann-Gewichten `bf[...]` lautet die innerste Schleife

```
for(x=1;x<L+1;x++) for(y=1;y<L+1;y++)
 {
  e=s[x][y]*(s[x-1][y]+s[x+1][y]+s[x][y-1]+s[x][y+1]);
  if( e<0 || rand()<bf[e/2] )
   {
    s[x][y]=-s[x][y];
    v=2*(x*80+2*(y-1)+2);
    ch=(s[x][y]+1)*15;
    poke(VSEG,v,0xf00|ch);
   }
 }
for(x=1;x<L+1;x++)
 {
  s[0][x]   = s[L][x];
  s[L+1][x] = s[1][x];
  s[x][0]   = s[x][L];
  s[x][L+1] = s[x][1];
 }
```

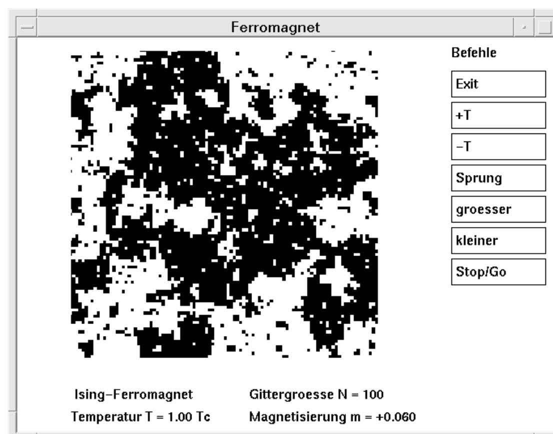
In der letzten `for`-Schleife werden die zusätzlichen Hilfsfelder mit den aktuellen Randspins besetzt, um die periodischen Randbedingungen zu erfüllen. Mit dem `poke`-Befehl sprechen wir auf dem PC den Bildspeicher im Textmodus direkt an. Dies ist sehr schnell, während ein Graphik- oder ein `putch()`-Befehl das Programm etwa um einen Faktor 2 langsamer macht. Allerdings funktioniert dieser Befehl nicht auf allen Maschinen.

Die Monte-Carlo-Simulation läßt sich mit wenig Aufwand programmieren. Mit kleinen Programmen und schnellen Computern erhält man leicht Resultate. Aber es ist nicht einfach, die Daten auszuwerten. Immerhin möchte man aus stark fluktuierenden Ergebnissen für relativ kleine Systeme und – verglichen mit dem Experiment – relativ kurzen Rechenzeiten allgemeine Gesetze für die Eigenschaften des

unendlichen Gitters im thermischen Gleichgewicht ($\widehat{=}$ unendliche Relaxationszeiten) ableiten. Auch mit der oben erläuterten Theorie des *finite size scaling*, die ja nur asymptotisch für $N \to \infty$ gilt, bleibt dies eine schwierige und mühsame Aufgabe. Hinzu kommt, daß versteckte Korrelationen in scheinbar guten Zufallszahlengeneratoren schon oft zu einem systematischen Fehler geführt haben.

Ergebnisse

Das PC-Programm ISING simuliert einen 20×20-Ising-Ferromagneten auf dem Quadratgitter und zeigt die Stellung der Spins auf dem Bildschirm an. Mit einem Tastendruck läßt sich während des Laufes das System aufheizen bzw. abkühlen. Die Anfangstemperatur $T = 2.269 J/k_{\mathrm{B}}$ ist nach Onsager die kritische Temperatur T_{c} des unendlichen Magneten. Zusätzlich zur Temperatur T und zur Zahl der Monte-Carlo-Schritte pro Spin (MCS) wird die Zeit ausgedruckt, die der Rechner für einen Monte-Carlo-Schritt benötigt, auf unserem PC etwa 4×10^{-5} s. Also selbst auf einem PC ist ein Schritt so schnell, daß die Spins im simulierten Wärmebad flackern. Bild 5.10 zeigt ein System der Größe 100×100, dessen Verhalten mit dem ent-

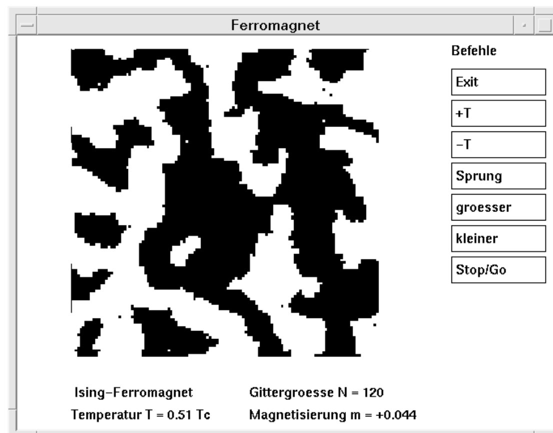


5.10 Zustand des Ferromagneten im thermischen Gleichgewicht bei T_{c}.

sprechenden Programm auf der Workstation simuliert wurde. Bei T_{c} sieht das Bild selbstähnlich aus, d. h. es treten Cluster von parallelen Spins auf allen Größenordnungen auf. Die Korrelationslänge ξ ist am kritischen Punkt so groß wie die Gitterausdehnung L, und die Korrelationen $\langle S_i S_j \rangle$ zerfallen für $a \ll r = |\boldsymbol{r}_i - \boldsymbol{r}_j| \ll L$ wie eine Potenz von r (a = Gitterkonstante). Bei $T = 0.8\, T_{\mathrm{c}}$ ist das System magnetisiert, $(\sum S_i)/N \neq 0$. Durch spontane Fluktuationen hat sich der Magnet eine der beiden möglichen Richtungen gewählt und die Spins S_i sind bis auf wenige Ausnahmen parallel dazu ausgerichtet. Wir beobachten also eine von null verschiedene

spontane Magnetisierung, und selbst wenn wir $\sum S_i$ über ein Zeitintervall t_1 mitteln würden, würde $\langle(\sum S_i)/N\rangle_{t_0,t_1}$ nicht verschwinden. Widerspricht dies der exakten Aussage, daß ein endliches System keinen Phasenübergang haben kann ($M(T) = 0$ für $T > 0$)?

Hierzu müssen wir uns die Zeitskala klarmachen, in der das System die Magnetisierungsrichtung ändern kann. Bei tiefen Temperaturen kann diese Zeit mit dem Arrhenius-Gesetz abgeschätzt werden. Um das Vorzeichen von $\sum S_i$ zu ändern, muß eine Energiebarriere der Ordnung $\Delta E = JL$ überwunden werden, denn es muß sich eine Wand zwischen positiver und negativer Magnetisierung bilden, die durch das gesamte Gitter diffundiert. Das braucht eine Zeit der Größenordnung $\tau \simeq t_0 \exp(\Delta E/k_\mathrm{B}T)$ wobei t_0 die Relaxationszeit eines einzelnen Spins ist. In unserem Fall ($L = 100, k_\mathrm{B}T = J, t_0 \simeq 10^{-4}$s) ist diese Zeit also etwa $\tau \simeq 10^{40}$ Sekunden, also wesentlich länger als das Universum existiert. Wir werden daher bei $T = J/k_\mathrm{B}$ keine Umkehr der Magnetisierung beobachten können, und es gilt $\langle M\rangle_{t_0,t_1} \neq 0$.



5.11 Wandernde Domänenwände des Ferromagneten nach sprunghaftem Abkühlen.

Bei $T = 1.5\,T_\mathrm{c}$ scheinen die Spins fast zufällig ausgewürfelt zu sein, die Korrelationslänge ξ hat fast den Wert a angenommen. Mit einem Tastendruck kann man nun den Magneten sprungartig auf eine Temperatur deutlich unterhalb T_c abkühlen. Nach einigen Versuchen sollte es gelingen, Bereiche mit positiver und negativer Magnetisierung gleichzeitig zu erzeugen (Bild 5.11). Dann entstehen diffundierende Domänenwände, die sich so lange gegenseitig vernichten oder zusammenziehen, bis das thermische Gleichgewicht mit nur einer einzigen Domäne übrig bleibt. Die Magnetisierung zeigt als Funktion der Zeit am kritischen Punkt starke Fluktuationen (Programm `ising_g`). Nach der statistischen Mechanik bestimmen diese Schwankungen von M gerade die magnetische Suszeptibilität χ, die bei T_c nach Gleichung

(5.58) mit L divergiert. Das gleiche gilt für die Fluktuationen der Energie, die die spezifische Wärme C bestimmen.

Für $T > T_\mathrm{c}$ und für lange Zeiten t zerfallen die zeitlichen Korrelationen (ebenso wie die räumlichen) exponentiell schnell

$$\langle S_i(t_0)\, S_i(t_0 + t)\rangle \sim e^{-t/\tau} \, . \tag{5.64}$$

Die Relaxationszeit $\tau(T)$ divergiert bei T_c wie

$$\tau \sim |T - T_\mathrm{c}|^{-z/\nu} \, , \tag{5.65}$$

wobei z ein neuer universeller kritischer Exponent ist, dessen Wert auch beim zweidimensionalen Modell nur numerisch bekannt ist $(z \simeq 2.1)$. Die Fluktuationen der Magnetisierung werden bei T_c also nicht nur größer, sondern auch langsamer. Deshalb nennt man diesen Effekt *critical slowing down*.

Übung

Es soll der Phasenübergang eines Gittergases mit der Monte-Carlo-Methode simuliert werden. Dazu betrachten wir ein Modell, bei dem auf einem $N \times N$-Gitter die Teilchen nur auf den Gitterpunkten sitzen können. Dies wird durch die Variable n_i beschrieben, die die Werte $n_i = 1(0)$ annimmt, falls der Gitterplatz i besetzt (unbesetzt) ist. Die Anzahl der Teilchen wird durch ein chemisches Potential kontrolliert. Die Dynamik besteht darin, daß Teilchen verschwinden und wieder auftauchen können. Jeder Platz i wird mit der Wahrscheinlichkeit proportional zu z^{n_i} besetzt, wobei z die *Fugazität* genannt wird. Benachbarte Teilchen sollen sich so stark abstoßen, daß keine Nachbarplätze besetzt werden können. Das Modell enthält sonst keine Wechselwirkungen und deshalb auch keine Temperatur, der einzige Kontrollparameter ist die Fugazität z, die die Teilchendichte des Gitters einstellt.

Wenn das Gitter maximal bedeckt ist, befinden sich $N^2/2$ Teilchen auf einem der beiden Untergitter, wie beim Schachbrett entweder auf den weißen oder auf den schwarzen Plätzen. Aber auch bei unvollständiger Bedeckung bevorzugen die Teilchen eines der beiden Untergitter; im thermischen Mittel ist die Differenz der Anzahl der Teilchen zwischen den weißen und schwarzen Plätzen ungleich Null. Diese Differenz ist der Ordungsparameter des Systems, er entspricht der Magnetisierung des Ising-Modells. Erst wenn die Bedeckung einen gewissen Wert unterschreitet, werden im thermischen Mittel beide Untergitter gleich stark belegt, und der Ordnungsparameter hat den Wert Null. Wie beim Ising-Modell gibt es diesen Phasenübergang streng genommen nur im unendlich großen System; in der Simulation muß man *finite size scaling* anwenden, um die Eigenschaften des Übergangs zu berechnen.

Programmieren Sie folgenden Monte-Carlo-Schritt für das Gittergas:

1. Suche zufällig einen Gitterplatz aus.
2. Ist der Gitterplatz leer, besetze ihn, wenn dies erlaubt ist, d. h., wenn alle nächsten Nachbarplätze leer sind, ansonsten akzeptiere die alte Konfiguration.
3. Ist der Gitterplatz besetzt, ziehe eine Zufallszahl $r \in [0, 1]$ und vergleiche sie mit z^{-1}. Ist sie kleiner, entferne das Teilchen. Ist sie größer, akzeptiere die alte Konfiguration.

Berechnen Sie den Mittelwert des Ordnungsparameters und dessen Varianz, die Suszeptibilität, als Funktion der Fugazität z. Bestimmen Sie deren kritischen Wert z_c und die zugehörige Bedeckung ρ_c des Gitters.
Hinweis: In der Literatur findet man $z_c = 3.7959$ und $\rho_c = 0.36776$.

Literatur

K. Binder, D. W. Heermann, *Monte Carlo Simulation in Statistical Physics*, Springer Verlag, 1992.

H. Gould, J. Tobochnik, *An Introduction to Computer Simulation Methods: Applications to Physical Systems, Parts I and II*, Addison Wesley, 1988.

J. Honerkamp, *Stochastische Dynamische Systeme: Konzepte, numerische Methoden, Datenanalysen*, VCH Verlagsgesellschaft, 1990.

S. E. Koonin, D. C. Meredith, *Physik auf dem Computer, Band 1+2*, R. Oldenbourg Verlag, 1990.

J. Schnakenberg, *Algorithmen in der Quantentheorie und Statistischen Physik*, Zimmermann-Neufang, 1995.

5.6 Kürzeste Rundreise

Ein Handlungsreisender möchte eine große Anzahl von Städten in möglichst kurzer Zeit besuchen. Dazu plant er eine Reiseroute, d. h. eine Reihenfolge der Städte mit möglichst kurzem Reiseweg. Welches ist die kürzeste Rundreise, bei der jede Stadt genau einmal besucht wird? Dieses scheinbar einfache Problem, das *travelling salesman problem (TSP)*, ist das Standardbeispiel der schwierigen kombinatorischen Optimierungsprobleme. Eine Generation von Mathematikern und Ingenieuren hat es mathematisch analysiert und Algorithmen zu seiner Lösung entwickelt.

Überraschenderweise kann man die Minimierung der Rundreise auch als ein Problem der statistischen Mechanik auffassen. Die Weglänge der Reise entspricht der Energie eines Vielteilchenproblems, und mit der Monte-Carlo-Methode aus dem

vorherigen Abschnitt können wir im Prinzip die kürzeste und in der Praxis eine sehr kurze Rundreise finden.

Physik

Verteile N Punkte – jeder Punkt entspricht einer Stadt – auf ein Quadrat mit der Seitenlänge L und markiere einen Rundweg, der jeden Punkt genau einmal berührt. Wieviele Rundwege gibt es? Eine bestimmte Route ist durch die Reihenfolge der Punkte festgelegt, und offensichtlich gibt es $N!$ verschiedene Anordnungen, nämlich gerade alle Permutationen der N Punkte. Da jeweils N Startpunkte und zwei Richtungen dieselbe Weglänge geben, müssen $(N-1)!/2$ verschiedene Wege betrachtet werden. Für $N = 100$ erhält man etwa 10^{155} Wege, also viel mehr, als man jemals auf einem Computer wird durchprobieren können.

$N!$ wächst schneller als e^N und sehr viel schneller als jede Potenz von N. Man kann sich zwar leicht eine regelmäßige Anordnung der Städte überlegen, bei der die kürzeste Reise angegeben werden kann, aber für den ungünstigsten Fall sind sich die Mathematiker fast sicher, daß es keinen Algorithmus gibt, der den kürzesten Rundweg in einer Anzahl von Schritten berechnet, die wie ein Polynom von N wächst. $N!$ wächst wie $N^{\frac{1}{2}}(N/e)^N$. Mit den Methoden der Informatik kann man jedoch das exakte Abzählen der Wege auf etwa $N^2\,2^N$ Rechenschritte reduzieren, für $N = 100$ also auf etwa 10^{34} Rechnungen. Das ist aber immer noch zuviel selbst für einen heutigen Superrechner.

Obwohl die numerische Lösung des Problems fast aussichtslos scheint, hat die Informatik aber Methoden entwickelt, mit denen in einem konkreten Fall die kürzeste Rundreise eines Handlungsreisenden, der etwa $N = 2500$ Städte besuchen soll, exakt berechnet wurde. Mathematisch läßt sich das *TSP* so formulieren: c_{ij} sei der Abstand der Punkte i und j, und es sei $x_{ij} = 1$, falls die Rundreise die Städte i und j direkt verbindet, $x_{ij} = 0$ sonst. Sowohl c_{ij} als auch x_{ij} sind also symmetrisch in den Indizes i und j. Dann suchen wir die kleinste Länge

$$E_0 = \min_{\{x_{ij}\}} \sum_{i<j} c_{ij}\, x_{ij} \tag{5.66}$$

unter der Nebenbedingung

$$\sum_{j=1}^{N} x_{ij} = 2 \ \text{ für alle } i\,. \tag{5.67}$$

Gleichung (5.67) besagt, daß die Reise jede Stadt genau einmal berücksichtigt, die

dementsprechend genau zwei unmittelbare Nachbarn hat. Um disjunkte Teilwege auszuschließen, muß man noch fordern:

$$\sum_{i \in S} \sum_{j \notin S} x_{ij} \geq 2 \ \text{ für alle } \ S \, , \tag{5.68}$$

wobei S eine beliebige echte Teilmenge der Indizes $(1, 2, \ldots, N)$ ist, ausgenommen die leere Menge. Die erfolgreichste Idee zur Lösung des *TSP* erweitert dieses diskrete Optimierungsproblem $(x_{ij} \in \{0, 1\})$ auf ein kontinuierliches $(x_{ij} \in \mathbb{R})$, das mit Standardmethoden der linearen Optimierung gelöst wird, wie wir sie im Abschnitt 1.9 besprochen haben. Durch ständiges Hinzufügen von Nebenbedingungen kann man schließlich alle kontinuierlichen Variablen x_{ij} auf die erlaubten Werte 0 und 1 einschränken. Damit hat man das Problem exakt gelöst.

Der Algorithmus ist sehr kompliziert und erfordert umfangreiche Programmierarbeit. Hier wollen wir eine andere Methode zur Lösung des *TSP* vorstellen, die 1982 aus der statistischen Mechanik ungeordneter Magnete (*Spingläser*) entstanden und relativ einfach zu programmieren ist. Dazu fassen wir die Variablen $x_{ij} \in \{0, 1\}$ als Freiheitsgrade eines Vielteilchensystems mit der Energie $H(\underline{S})$ auf:

$$H(\underline{S}) = \sum_{i<j} c_{ij} \, x_{ij} \, . \tag{5.69}$$

$\underline{S} = (x_{12}, x_{13}, ... x_{N-1,N})$ kennzeichnet eine der $(N-1)!/2$ möglichen erlaubten Rundreisen und H ist deren Länge. Wir suchen den Zustand mit der niedrigsten Energie. Die Physik, insbesondere die Thermodynamik des Festkörpers sagt uns, wie wir vorgehen sollten: Wir müssen das System aufheizen und es dann sehr langsam abkühlen. Wenn sich das System immer im thermischen Gleichgewicht befindet, dann muß es für $T \to 0$ den Zustand tiefster Energie erreichen. Der vorige Abschnitt sagt uns auch, wie wir bei diesem Problem das Aufheizen und Abkühlen bewerkstelligen können, nämlich mit Hilfe der Monte-Carlo-Simulation. Dabei wird ein stochastischer Prozeß mit einer Übergangswahrscheinlichkeit $W(\underline{S} \to \underline{S}')$ konstruiert, die das detaillierte Gleichgewicht mit dem Boltzmann-Faktor $P(\underline{S}) = \exp(-H(\underline{S})/T)/Z$ erfüllt (Gleichung (5.49)). T ist eine formale „Temperatur“, die dieselbe Einheit wie $H(\underline{S})$, also die einer Länge haben muß.

Im vorigen Abschnitt haben wir gesehen, daß die so erzeugte Folge von Rundreisen $\underline{S}(0), \underline{S}(1), ..., \underline{S}(t)$ ins thermische Gleichgewicht relaxiert. Das heißt, die Wahrscheinlichkeit, den Weg $\underline{S}(t)$ vorzufinden, ist $P(\underline{S}(t))$, wenn t nur groß genug ist. Nun können wir das System so langsam abkühlen, daß es sich immer im Gleichgewicht befindet. Bei $T = 0$ ist offenbar $P(\underline{S})$ nur dann von Null verschieden, wenn $H(\underline{S})$ die niedrigste Energie (= den kürzesten Weg) annimmt.

Wenn wir also nur langsam genug abkühlen, dann finden wir mit Sicherheit die kürzeste Rundreise. Was aber ist „langsam genug“ ? Das ist nur schwer zu beant-

worten. Wenn wir durch den Raum aller Wege $\underline{S}$ wandern und $H(\underline{S})$ als Höhe ansehen, dann ist $H(\underline{S})$ ein komplexes Gebirge mit vielen Tälern, Seitentälern, Schluchten und Schächten. Laufen wir also nur den Berg hinab, so werden wir mit hoher Wahrscheinlichkeit in einem Nebental weit weg vom absoluten Minimum landen. Die Monte-Carlo-Methode (bei $T > 0$) erlaubt hingegen auch mit einer gewissen Wahrscheinlichkeit, die vom Höhenunterschied abhängt, Schritte bergauf. Wie lange aber dauert es, einen hohen Bergkamm zu überschreiten, um in ein tieferes Tal zu kommen?

Diese Zeit kann wie im vorigen Abschnitt mit dem Arrhenius-Gesetz abgeschätzt werden. Sei $\Delta E > 0$ die zu überwindende Höhendifferenz. Dann ist eine typische Zeit

$$\tau = \tau_0 \, e^{\Delta E/T} \,. \tag{5.70}$$

Beim Abkühlen müssen wir dem Wanderer Zeit lassen, die Bergkämme zu überschreiten. Durch Umkehren der Gleichung (5.70) sehen wir, daß ein für die Simulation entworfener Abkühlplan $T(t)$ – dabei sei t die aktuelle Rechenzeit – möglichst die Ungleichung

$$T(t) \geq \frac{\Delta E}{\ln\left(\frac{t}{\tau_0}\right)} \,. \tag{5.71}$$

erfüllen sollte. Die Temperatur T darf daher als Funktion der Zeit t nur extrem langsam sinken, um thermisches Gleichgewicht zu garantieren. In der Praxis wird man nie so viel Rechenzeit aufwenden können, um das thermische Gleichgewicht perfekt zu simulieren. Aber wenn wir nur dicht beim Gleichgewicht bleiben können, werden wir dennoch eine Energie finden, die fast die Grundzustandsenergie erreicht. Die Weglänge wird also fast optimal sein, während der tatsächliche Weg ganz anders als der optimale aussehen kann.

Mit wenig Programmieraufwand und geringem Wissen über das Problem, aber mit viel Rechenzeit, kann man daher mit dieser Methode des „Simulierten Ausglühens“ (*simulated annealing*) ein gutes lokales Minimum eines komplexen kombinatorischen Optimierungsproblems finden. Die Methode kann leicht auf viele andere Probleme dieser Art angewandt werden, sie konkurriert zur Zeit allerdings mit sogenannten genetischen Optimierungsalgorithmen, die mit Populationen von Wegen arbeiten.

Bevor wir die Einzelheiten des Algorithmus besprechen, wollen wir die kürzeste Rundreise für den Fall abschätzen, daß N Städte zufällig und unkorreliert im Quadrat der Kantenlänge L verteilt sind. Wenn wir zu jeder Stadt i den Abstand r_i zum nächsten Nachbarn bestimmen, dann gilt offenbar für die kleinste Länge:

$$E_0 \geq \sum_{i=1}^{N} r_i \,. \tag{5.72}$$

Für $N \to \infty$ kann die rechte Seite als $N\langle r\rangle$ geschrieben werden, wobei $\langle r\rangle$ der mittlere Nächste-Nachbar-Abstand ist. Wir wollen nun $\langle r\rangle$ berechnen, dabei aber den Effekt des Randes vernachlässigen und dazu einen genügend großen Ausschnitt der Ebene betrachten, auf der Städte mit der Dichte $\rho = N/L^2$ verteilt sind. Sei $w(r)dr$ die Wahrscheinlichkeit, daß die nächste Nachbarstadt einen Abstand zwischen r und $r + dr$ hat. $w(r)dr$ ist also die Wahrscheinlichkeit, zwischen null und r keinen nächsten Nachbarn zu finden, multipliziert mit der Wahrscheinlichkeit, in dem Kreisring mit der Fläche $2\pi r\, dr$ eine Stadt zu finden:

$$w(r)\, dr = \left[1 - \int\limits_0^r w(r')\, dr'\right] \rho\, 2\pi r\, dr\,. \tag{5.73}$$

Diese Integralgleichung für $w(r)$ läßt sich unmittelbar in eine Differentialgleichung für $f(r) = 1 - \int\limits_0^r w(r')\, dr'$ umwandeln, wenn man $f'(r) = -w(r)$ benutzt, nämlich

$$f' = -2\pi\, \rho\, r\, f \quad \text{mit} \quad f(0) = 1\,. \tag{5.74}$$

Als Lösung erhalten wir

$$f(r) = e^{-\pi\,\rho\, r^2} \quad \text{und} \quad w(r) = 2\pi\rho r\, e^{-\pi\rho r^2}\,. \tag{5.75}$$

Der mittlere Abstand der nächsten Nachbarn ist durch $w(r)$ gegeben:

$$\langle r\rangle = \int\limits_0^\infty r\, w(r)\, dr = 2\pi\rho \int\limits_0^\infty e^{-\pi\rho r^2}\, r^2 dr = \frac{1}{2\sqrt{\rho}} = \frac{1}{2}\,\frac{L}{\sqrt{N}}\,. \tag{5.76}$$

Damit gilt nach Gleichung (5.72) für die Länge E_0 der kürzesten Rundreise (im Limes $N \to \infty$)

$$E_0 \geq \frac{1}{2}\, L\sqrt{N}\,. \tag{5.77}$$

Berücksichtigt man entsprechend die übernächsten Nachbarn, so ändert sich der Vorfaktor zu $0.599\ldots$ In jedem Fall erhält man aber das Skalenverhalten $L\sqrt{N}$, so daß man verschiedene Systemgrößen L und Städtezahlen N miteinander vergleichen kann. Den genauen Wert der skalierten Länge

$$l = \frac{E_0}{L\sqrt{N}} \tag{5.78}$$

kann man bis heute nur numerisch bestimmen, z. B. finden Percus und Martin $l = 0.7120 \pm 0.0002$ im Limes $N \to \infty$.

Algorithmus

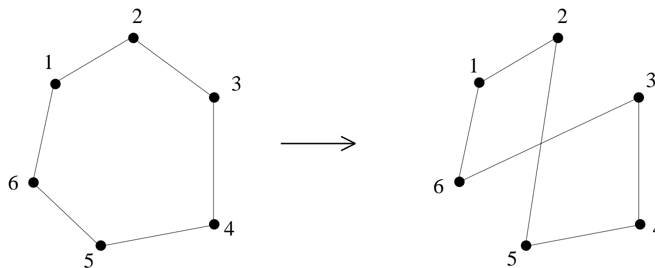
Wie im vorigen Abschnitt müssen wir uns einfache Änderungen des Zustandes $\underline{S}$ überlegen. Erlaubte Wege kann man durch ein Feld darstellen, das die Reihenfolge enthält, mit der die Städte besucht werden,

$$\underline{S} = (i_1, i_2, ..., i_N)\,, \tag{5.79}$$

wobei $i_k \in \{1, ..., N\}$ die Nummer der Stadt ist. Es gibt viele Möglichkeiten, aus einer Rundreise $\underline{S}$ eine etwas geänderte $\underline{S}'$ zu konstruieren. Hier wollen wir der Einfachheit halber nur die folgende vorstellen: Wir wählen zufällig eine Position p und eine Länge $l \leq N/2$, trennen den Abschnitt $i_p...i_{p+l}$ aus dem Weg $\underline{S}$ heraus und fügen ihn in umgekehrter Reihenfolge wieder ein,

$$\underline{S}' = (i_1, ..., i_{p-1}, i_{p+l}, i_{p+l-1}, ..., i_p, i_{p+l+1}, ..., i_N)\,. \tag{5.80}$$

Dabei ist $p + l$ modulo N zu nehmen. Dieser Zug ist in Bild 5.12 dargestellt. Mit $N = 6\,,\, p = 3$ und $l = 2$ wird die Rundreise $\underline{S} = (1, 2, 3, 4, 5, 6)$ zu $\underline{S}' = (1, 2, 5, 4, 3, 6)$. Wiederholte Anwendung dieses Zuges führt offenbar durch die gesamte Menge aller möglichen Wege und kann alle Kreuzungen beseitigen. Damit haben wir alles, was wir zur Anwendung der im vorigen Abschnitt beschrie-



5.12 Schrittweise Änderung der Rundreise bei der Monte-Carlo-Simulation.

benen Monte-Carlo-Simulation benötigen. Jeder versuchte Zug wird akzeptiert, falls die neue Rundreise kürzer ist $\left(H(\underline{S}') \leq H(\underline{S})\right)$ oder eine Zufallszahl $r \in [0, 1]$ kleiner als der Boltzmann-Faktor $\exp\left[-\left(H(\underline{S}') - H(\underline{S})\right)/T\right]$ ist.

Im letzteren Fall müssen wir jedes Mal sowohl eine Zufallszahl r als auch die Funktion $\exp(...)$ berechnen. Beides kann man vermeiden, indem man den Boltzmann-Faktor durch eine Stufenfunktion $\theta\left[T - \left(H(\underline{S}') - H(\underline{S})\right)\right]$ ersetzt. Der Zug $\underline{S} \to \underline{S}'$ wird also immer akzeptiert, wenn die Energiedifferenz $H(\underline{S}') - H(\underline{S})$ kleiner als T ist. Für diese Vereinfachung kennen wir keine theoretische Begründung,

sie wird aber seit kurzem bei der kombinatorischen Optimierung erfolgreich angewandt.

Das C-Programm `travel.c` berücksichtigt beide Methoden. Im Quadrat der Länge `L` werden `N` Städte zufällig verteilt. Die Koordinaten (x, y) jeder Stadt sind im Feld `map` mit der Struktur `city` deklariert:

```
typedef struct {int x,y;} city;
city map[MAXCITIES];
```

`N` ist auf den Wert 100 gesetzt, es kann aber auch als Parameter beim Aufruf von `travel` übergeben werden. Dies geschieht mit

```
main(int argc,char *argv[])
{
    ...
  if(argc>1) N=atoi(argv[1]);
    ...
```

Die Reihenfolge der Städte wird im Feld `int path[N]` festgelegt. Am Anfang werden die Koordinaten (x, y) der Städte zufällig und der Startweg als $\underline{S} = (0, 1, \ldots, N-1)$ gewählt. Die Hauptschleife sieht dann folgendermaßen aus:

```
while (!done)
   {
    event();
    anneal(path);
    if(count++ > DRAW)
     {
      drawpath(path);
      oldlength = length(path);
      print(oldlength);
      if(aut) temp = .999*temp;
      count=0;
     }
   }
```

Die Funktion `event()` fragt den Tastaturpuffer ab. Bei laufendem Programm sind folgende Aktionen möglich: Herauf- oder Heruntersetzen der Temperatur um 10%, Wahl der Methode *Boltzmann-Faktor* oder *Schwellenfunktion*, Umschalten auf automatische Temperaturerniedrigung und das Speichern der aktuellen Rundreisekonfiguration. Schließlich kann man noch die Temperatur auf den Wert `temp=0` springen lassen. Dann werden nur noch solche Züge akzeptiert, die die Weglänge verringern.

Die Funktion `anneal(path)` berechnet den nächsten Monte-Carlo-Schritt. Es werden zufällig die Position `pos` und die Länge `len` gewählt, die benötigt werden,

um mit `change(newpath,oldpath,pos,len)` die neue Rundreise erzeugen zu können. Weiterhin definieren wir eine Funktion `changed_length(...)`, um die Länge des neuen Weges zu bestimmen. Am effektivsten programmiert man diese Funktion, indem man nur die Längenänderung gegenüber der alten Konfiguration berechnet. Um der Anhäufung von Rundungsfehlern vorzubeugen, wird in regelmäßigen Abständen mit `length(path)` die Weglänge von Grund auf neu berechnet. Der ausgewürfelte Weg wird akzeptiert, wenn für den Unterschied `de` der beiden Weglängen gilt:

```
(ann == 1 && (de < 0 || frand() < exp(-de/temp))) ||
              (ann == 0 && de < temp)
```

Falls diese Größe wahr ist, also den Wert 1 hat, wird durch die Standardfunktion `memcpy` der neue Weg in den alten kopiert, der Zug $\underline{S} \to \underline{S}'$ wird akzeptiert. Insgesamt lautet die Funktion

```
void anneal( int oldpath[] )
 {
  double newlength,de,lscal;
  int pos,len,newpath[MAXCITIES];

  pos=rand()*f1;
   len=rand()*f2;
  change(newpath,oldpath,pos,len);
  newlength=changed_length(oldlength,oldpath,pos,len);
  de=newlength-oldlength;
  if(de==0.) return;
  if((ann==1 && (de < 0 || frand() < exp(-de/temp))) ||
     (ann==0 && de < temp))
   {
    memcpy(oldpath,newpath,N*sizeof(int));
    oldlength=newlength;
    flipcount++;
   }
 }
```

`f1` und `f2` sind vorher definierte Skalierungsfaktoren und `frand()` ist ein Zufallszahlen-Generator, der gleichverteilte reelle Zahlen im Einheitsintervall erzeugt. Er kann z. B. am Anfang des Programms mit

```
#define frand() (double)rand()/(RAND_MAX+1.)
```

definiert werden.

Die neue Rundreise wird erzeugt, indem zunächst die alte Rundreise (`op`) in die neue (`np`) kopiert wird. Dann wird zwischen `pos` und `pos+len` die Reihenfolge der Städte umgedreht. Das sieht so aus:

```
void change(int np[],int op[],int pos,int len)
 {
  int i,j;

  memcpy(np,op,N*sizeof(int));
  j=len;
  for(i=0;i<=len;i++)
    {
     np[(pos+i) % N]=op[(pos+j) % N];
     j--;
    }
 }
```

Die Berechnung der Weglänge ist offensichtlich:

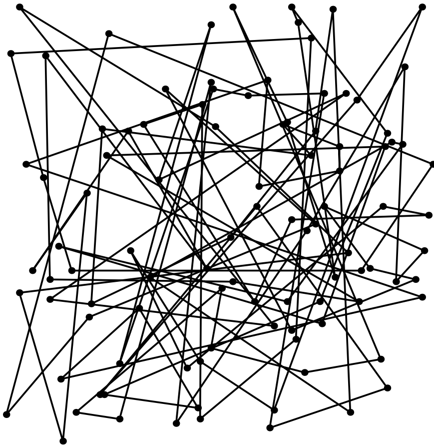
```
double length(int path[] )
  {
   int i,j;
   double l=0.,dx,dy;

   for(i=0;i<N;i++)
     {
      j=(i+1)%N;
      dx=map[path[i]].x-map[path[j]].x;
      dy=map[path[i]].y-map[path[j]].y;
      l+= sqrt(dx*dx+dy*dy);
     }
   return (l);
  }
```

Weitere Funktionen `drawpath` und `print` zeichnen den Weg bzw. schreiben Text in das Graphikfenster. Nach dem Kompilieren können wir mit dem Aufruf `travel` die Suche nach der kürzesten Rundreise auf dem Bildschirm beobachten und dabei die Temperatur hoch- und herunterfahren.

Ergebnisse

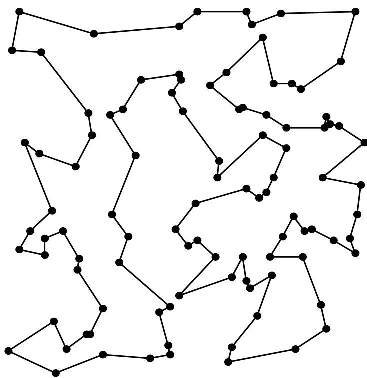
Wir haben die Daten aus `travel.dat` mit *Mathematica* eingelesen und gezeichnet. Der zufällig erzeugte Startzustand von etwa 100 Städten ist in Bild 5.13 zu se-

hen. Die skalierte Weglänge l, Gleichung (5.78), hat den Wert $l \simeq 4.8$. Wir starten



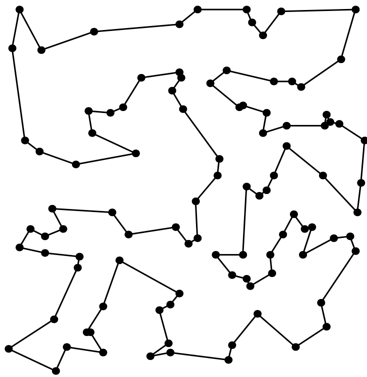
5.13 Zufällig gewählter Startzustand mit der skalierten Länge $l = 4.8$

bei einer Temperatur von `L/8`. Schon bei dieser relativ hohen Temperatur relaxiert das System schnell auf den Wert $l \simeq 2.2$, dabei werden etwa 20% der versuchten Züge akzeptiert. Durch langsames automatisches Abkühlen, wobei die Temperatur nach jeweils 100 Versuchen um 1‰ erniedrigt wird, gelangt das System nach etwa 30 Minuten Rechenzeit auf unserem PC in die Zustände Bild 5.14 bzw. Bild 5.15. Beide Rundreisen sind lokale Minima im Energiegebirge, d. h. alle unsere



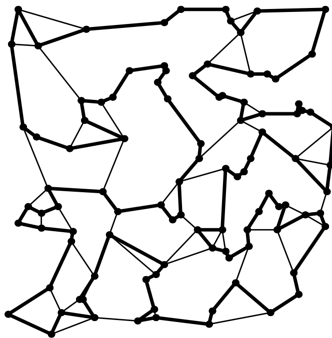
5.14 Rundreise nach dem simulierten Ausglühen mit dem Boltzmann-Algorithmus. Die skalierte Länge ist nun $l = 0.883$.

Wegänderungen $\underline{S} \to \underline{S}'$ geben immer nur längere Reisen, die bei $T = 0$ nicht mehr akzeptiert werden. In diesem Fall – und in den meisten anderen – ergab der schnelle Schwellenalgorithmus einen etwas niedrigeren Wert ($l = 0.858$) als die langsamere Metropolis-Methode ($l = 0.883$). Geht man von Anfang an im komplexen Energiegebirge $H(\underline{S})$ nur bergab (`temp=0`), so erhält man einen längeren



5.15 Wie Bild 5.14, aber mit dem Schwellen-Algorithmus. Das Ergebnis ist eine etwas kürzere Rundreise mit $l = 0.858$

Weg ($l = 0.909$), allerdings sehr viel schneller als mit den beiden Abkühlverfahren. Bild 5.16 vergleicht die beiden Resultate der Abbildungen 5.14 und 5.15. Obwohl



5.16 Vergleich der beiden optimierten Rundreisen aus den Abbildungen 5.14 (dünn) und 5.15 (dick). Obwohl die beiden Kurven fast gleich lang sind, sind sie doch deutlich voneinander verschieden.

die Längen der beiden Rundreisen fast gleich sind, sind die Reiserouten deutlich voneinander verschieden. Wiederholt man die Simulation mit anderen Zufallszahlen (bei denselben Städten), so erhält man fast die gleiche Weglänge aber wieder eine ganz andere Reihenfolge der Städte.

Wir haben die Skalierung Gleichung (5.77) überprüft, indem wir eine Rundreise mit 300 Städten simuliert und mit dem Schwellenalgorithmus langsam abgekühlt haben. Das Ergebnis $l = 0.826$ bestätigt die Skalierung $E_0 \propto \sqrt{N}$.

Natürlich wissen wir nie, ob wir wirklich die kürzeste Rundreise gefunden haben. Um diese Frage zu beantworten, benötigen wir die relativ aufwendigen Methoden der Informatik. Um aber bei einem noch wenig verstandenen Optimierungsproblem mit geringem Aufwand eine gute Lösung zu erhalten, kann man *simulated annealing* wohl empfehlen.

Übung

Hier soll versucht werden, das Optimierungsproblem aus Abschnitt 1.3 mit der Methode des *simulated annealing* zu lösen. Es soll ein Signal mit gegebenem Leistungsspektrum übertragen werden, und dabei soll die Spitzenspannung möglichst klein gehalten werden.

Von der an $N = 64$ diskreten Zeitpunkten abgetasteten Spannung $U_1, \ldots, U_N$ sei deren Fouriertransformierte $\{b_1, \ldots, b_N\}$, bzw. das Leistungsspektrum bekannt: $|b_s|^2 = |b_{N-s+2}|^2 = 1$ für $s = 9, 10, \ldots, 17$ und $b_s = 0$ sonst. Damit das Spannungssignal $\{U_1, \ldots, U_N\}$ reell ist, muß $b_{N-s+2} = b_s^*$ gelten.

Gesucht sind die Phasen φ_s, die mit $b_s = e^{i\varphi_s}$ eine minimale Spitzenspannung geben, es soll also der Wert von $\min_{\{\varphi_s\}}(\max_r |U_r|)$ bestimmt werden.

Der Zustand $\underline{S}$ aus Abschnitt 5.5 ist nun eine Konfiguration möglicher Phasen $(\varphi_9, .., \varphi_{17})$, und der Energie entspricht hier die Spitzenspannung $H(\underline{S}) = \max_r |U_r|$. In jedem Monte-Carlo-Schritt wird die Konfiguration der Phasen zufällig geändert, z. B. durch Variation einer einzelnen, beliebig herausgegriffenen Phase. Dies liefert eine Energieänderung, und der Algorithmus aus Abschnitt 5.5 entscheidet daraufhin, ob die neue Konfiguration akzeptiert wird.

Das bisher beste, von einem unserer Studenten erzielte Ergebnis liegt bei $H = 0.739545$. Wer diesen Wert unterbietet, möge uns dies bitte mitteilen.

Literatur

A. G. Percus, O. C. Martin, *Dimensional Dependence in the Euclidean Travelling Salesman Problem*, Physical Review Letters **76**, 1188 (1996).

W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge University Press, 1992.

J. Schnakenberg, *Algorithmen in der Quantentheorie und Statistischen Physik*, Zimmermann-Neufang, 1995.

Anhang A
Erste Schritte mit Mathematica

Auf fast allen Rechnern wird *Mathematica* mit dem Befehl **`math`** aufgerufen. Bei Systemen mit einer graphischen Oberfläche geschieht dies manchmal indirekt durch das Anklicken eines Symbols. *Mathematica* meldet sich danach mit `In[1]:=` und erwartet eine Eingabe, die mit RETURN abgeschickt wird. Die Notebook-Version von *Mathematica* verhält sich insofern etwas anders, als hier einzelne oder mehrere Befehle mit SHIFT-RETURN übergeben werden. *Mathematica* ist eine interpretierende Programmiersprache, bearbeitet also sofort die Eingabe und präsentiert das Ergebnis in der Form `Out[..]=` ... auf dem Bildschirm. Dabei werden alle Ausdrücke, die vorher in derselben Sitzung definiert wurden, berücksichtigt. Man beendet eine *Mathematica*-Sitzung mit `In[..]:=` **`Quit`**. Der Befehl **`Exit`** bzw. CONTROL-D auf Unix-Systemen bewirkt dasselbe.

> Das Kommando **`math`** startet eine *Mathematica*-Sitzung.
>
> `In[..]:=` **`Quit`** beendet die Sitzung.

Simple Arithmetik

`In[1]:= 4 + 7`	
`Out[1]= 11`	
`In[2]:= 3 4.2`	*Leerstelle = Multiplikation*
`Out[2]= 12.6`	*Dezimalzahlen mit Dezimalpunkt*
`In[3]:= 2*3*4`	*Auch * = Multiplikation*
`Out[3]= 24`	
`In[4]:= 24/8`	*/ = Division*
`Out[4]= 3`	
`In[5]:= 2^3`	*^ = Potenz*
`Out[5]= 8`	

In[6]:= **(3+4)^2−2(3+1)**	
Out[6]= 41	
In[7]:= **%/5**	% = *das letzte ausgegebene Resultat*
Out[7]= $\frac{41}{5}$	
In[8]:= **N[%]**	N[x] = *numerischer Wert von x*
Out[8]= 8.2	
In[9]:= **22.4/(6.023 10^23)**	
Out[9]= 3.71908 10^{-23}	
In[10]:= **3^−2*36+4**	*„Potenz" vor „Punkt" vor „Strich"*
Out[10]= 8	

Die wichtigsten Funktionen

In[11]:= **Sqrt[16]**	Sqrt[x] = $\sqrt{x}$
Out[11]= 4	
In[12]:= **N[Pi/2]**	Pi = *Mathematica-Form von* π
Out[12]= 1.5708	
In[13]:= **Sin[%]**	Sin[x] = $\sin x$
Out[13]= 1.	1. = 1.0

Alle in *Mathematica* vorhandenen Funktionen, Prozeduren, Konstanten, ... beginnen mit einem Großbuchstaben.
Funktionsargumente stehen jeweils in eckigen Klammern [ ] .

In[14]:= **Cos[Pi/4]**	Cos[x] = $\cos x$
Out[14]= $\frac{\text{Sqrt[2]}}{2}$	*Exakter Wert*
In[15]:= **Exp[0.5]**	Exp[x] = *Exponentialfunktion von x*
Out[15]= 1.64872	*Numerischer Wert*
In[16]:= **Log[%]**	Log[x] = $\ln x$
Out[16]= 0.5	

In[17]:= **?Log**
Log[z] gives the natural Logarithm of z (logarithm to base E). Log[b, z] gives the logarithm to base b.

In[17]:= **Log[E, %15]**

%15 *ist dasselbe wie* Out[15]. *Alle während einer Sitzung erzeugten* In[n] *und* Out[n] *sind wieder aufrufbar.*

Out[17]= 0.5

Hilfen in *Mathematica* erhält man mit dem Fragezeichen (?)

In[..]:= **?Log** ⟶ Informationen zu Log[x].
In[..]:= **??Log** ⟶ Mehr Informationen zu Log[x].
In[..]:= **?L*** ⟶ Alle Funktionen, die mit L beginnen.
In[..]:= **?*** ⟶ „Alles“ wird aufgelistet.

In[18]:= **Sqrt[-1]**

Out[18]= I

In[19]:= **?I**
I represents the imaginary unit Sqrt[-1].

In[19]:= **Exp[I*N[Pi/4]]**

Out[19]= 0.707107 + 0.707107 I

In[20]:= **Re[%]+I*Im[%]**

Re[z] = *Realteil von z*
Im[z] = *Imaginärteil von z*

Out[20]= 0.707107 + 0.707107 I

In[21]:= **?Arc***
ArcCos ArcCot ArcCsc ArcSec ArcSech ArcSin ArcSinh ArcTan ArcTanh ArcCosh ArcCoth ArcCsch

In[21]:= **ArcTan[1]**

ArcTan[x] = $\arctan x$

Out[21]= $\frac{\text{Pi}}{4}$

Graphik

Wir verzichten im folgenden auf die Kennzeichnung von Ein- und Ausgaben durch das *In[..]* bzw. *Out[..]* am Zeilenanfang. Eingaben an den Rechner sind durch Fettdruck kenntlich gemacht, alles andere sind Ausgaben vom Rechner oder unsere Kommentare.

`Plot[Sin[x], {x, 0, 2Pi}]` *Zeichnung der Funktion* $\sin x$.

`Plot[{BesselJ[0,x], BesselJ[1,x]}, {x, 0, 10}]` *Die Besselfunktionen* $J_0(x)$ *und* $J_1(x)$ *in einem Plot.*

`Plot3D[Exp[-(x^2+y^2)], {x, -2, 2}, {y, -2, 2}]` *Dreidimensionaler Plot der 2-dimensionalen Gaußfunktion* $\exp[-(x^2+y^2)]$.

`ParametricPlot[{Cos[t], Sin[t]}, {t, 0, 2Pi}]` *Parameter-Plot eines Kreises. Auf dem Bildschirm erscheint eine Ellipse, weil das Achsenverhältnis ungleich 1 ist.*

`Show[%, AspectRatio -> Automatic]` *Jetzt ist es ein Kreis.* `Show[..]` *zeigt Graphikobjekte mit den entsprechenden Optionen.*

`ParametricPlot3D[{Cos[t],Sin[t],t/6}, {t,0,6Pi}]`
Eine Schraubenlinie in 3 Dimensionen.

```
ParametricPlot3D[{r Cos[t],r Sin[t],r^2},
                                    {r,0,2},{t,0,2Pi}]
```

Ein Rotationsparaboloid als Fläche in 3 Dimensionen.

```
ContourPlot[Exp[-(x-1)^2-y^2]+ Exp[-(x+1)^2-y^2],
                  {x,-3,3},{y,-3,3}, PlotPoints -> 60]
```

Höhenlinien. `PlotPoints -> 60` *ist eine optionale Angabe.*

`Options[Plot]` *Eine Liste der Optionen zu* `Plot[..]`.

Die wichtigsten Plot-Befehle:

```
Plot[f, {x, xmin, xmax}]
Plot3D[f, {x, xmin, xmax}, {y, ymin, ymax}]
ParametricPlot[{fx, fy}, {t, tmin, tmax}]
ParametricPlot3D[{fx,fy,fz}, {t, tmin, tmax}]
ParametricPlot3D[{fx,fy,fz}, {s,s1,s2}, {t,t1,t2}]
ContourPlot[f, {x, xmin, xmax}, {y, ymin, ymax}]
```

Symbole und eigene Funktionen

`n = 20!` $\longrightarrow$ `2432902008176640000`, `k!` = `Faktorial[k]` = *k-Fakultät.*

`N[n]` $\longrightarrow$ `2.4329` 10^{18}, *die Variable* `n` *hat den Wert* `20!`

`a = Random[]` $\longrightarrow$ `0.296851`, `Random[]` *erzeugt Zufallszahlen zwischen* 0 *und* 1. *Der Variablen* `a` *wurde die Zahl* `0.296851` *zugewiesen.*

`N[a, 13]` $\longrightarrow$ `0.2968512129099`, *die Zahl* `a` *mit 13 Stellen. Falls nicht anders angegeben, werden Dezimalzahlen intern mit 16 Stellen verarbeitet.*

`Clear[a,n]` *löscht die Belegung der Variablen* `a,n`.

`c = (a+b)^3` $\longrightarrow$ `(a + b)`^{3}

`c = Expand[c]` $\longrightarrow$ `a`^{3} `+ 3 a`^{2} `b + 3 a b`^{2} `+ b`^{3}

`f[x_]:= Exp[-0.25 x`^{2}`]` **`Cos[4.5 x]`** *So definiert man Funktionen.* `x_` *ist ein ‚Platzhalter' mit dem Namen* `x`. *Der Unterstreichungsstrich bei* `x_` *ist wichtig. Er darf nur auf der linken Seite der Gleichung auftauchen. Statt des Zeichens* `:=` *hätten wir hier auch das Zeichen* `=` *verwenden können.*

`f[1.5]` $\longrightarrow$ `0.50882`, *Funktionswert von* `f` *für* $x = 1.5$.

`Plot[f[x],{x,-Pi,Pi}]` *Plot der oben definierten Funktion* `f`. *Statt* `x` *kann man hier auch einen beliebigen anderen Variablennamen verwenden. Mit dem Befehl* `Plot[f[t],{t,-Pi,Pi}]` *erreicht man dasselbe.*

`fermi[e_,b_]:= 1/(Exp[b(e-1)]+1)` *Eine Funktion mit zwei Variablen.*

`Plot[{fermi[e,10],fermi[e,Infinity]},{e,0,1.7}]` *Plot zweier Fermifunktionen.* `Infinity` $= +\infty$.

> Der Unterschied von **`:=`** und **`=`** an einem Beispiel:
> **`a = Random[]`**
> Die rechte Seite wird ausgewertet, und der Variablen **`a`** wird diese Zufallszahl zugewiesen. Wiederholter Aufruf von **`a`** liefert jedesmal diese Zahl.
>
> **`r:= Random[]`**
> Die rechte Seite wird zunächst nicht ausgewertet, sondern erst dann, wenn die Variable **`r`** erneut aufgerufen wird. Bei jedem Aufruf von **`r`** wird die rechte Seite erneut ausgewertet. Es wird jedesmal eine neue Zufallszahl produziert.

Listen

`numlist = {2,3,4}` *Eine Liste von drei Zahlen. Listen werde mit geschweiften Klammern* { } *geschrieben.*

Head[numlist] ⟶ `List`. *Jedes Ding hat einen Kopf.* `numlist` *ist eine Liste.*

numlist^2 ⟶ `{4,9,16}`. *Fast alle Funktionen sind* `Listable`.

Log[numlist] // N ⟶ *Liste der Logarithmen*

Table[2, {7}] ⟶ `{2,2,2,2,2,2,2}`. `Table[..]` *macht Listen.*

Table[i^3, {i,5}] ⟶ `{1, 8, 27, 64, 125}`

Table[1/j, {j,3,8}] ⟶ $\{\frac{1}{3}, \frac{1}{4}, \frac{1}{5}, \frac{1}{6}, \frac{1}{7}, \frac{1}{8}\}$

Table[x, {x,0,2,0.25}] ⟶ `{0, 0.25, 0.5, ..., 2.}`

> **Table[expr, {..}]** erzeugt Listen. Die geschweifte Klammer mit der allgemeinen Form **{i, imin, imax, di}** heißt `'iterator'`. Verkürzungen der allgemeinen Form sind möglich.

liste = Table[{x, Gamma[x]}, {x,2,4,0.05}];

> Ein Semikolon (;) nach einem Ausdruck unterdrückt das nächste `Out[..]`. Trotzdem haben die Variablen die zugewiesenen Werte.

ListPlot[liste] `ListPlot[..]` *plottet Daten.*

letters = Table[FromCharacterCode[j], {j,122,97,-1}]
⟶ `{z,y,x,w, ...}`

Sort[letters] ⟶ `{a,b,c,d, ... }`. `Sort[..]` *sortiert, nicht nur Buchstaben, sondern auch Zahlen.*

letters[[3]] ⟶ `x`, *das 3. Element der Liste* `letters`. *Teile von Listen oder von Ausdrücken bezeichnet man mit doppelten eckigen Klammern* `[[..]]`.

Length[liste] ⟶ `41`, *die Länge der Liste* `liste`.

liste[[20]] ⟶ `{2.95, 1.91077}`

liste[[20,1]] ⟶ `2.95`

liste[[20,2]] ⟶ `1.91077`

abc = Reverse[letters] ⟶ `{a,b,c,d,...}`

abc[[{2,4,6}]] ⟶ `{b,d,f}`

`abc[[-1]]` ⟶ `z`. *Ein negativer Index entspricht der Zählung von hinten.*

`Permutations[{1,2,3}]` ⟶ *alle Permutationen von* `{1,2,3}`.

Speichern, Einlesen, Fitten von Daten

`Clear["Global'*"]` *Löscht alle selbstdefinierten Variablen und Funktionen.*

`f[x_] = 1.0 x^4 - 2.0 x^2 + 0.5` *Ein Polynom vierten Grades.*

`?f` ⟶ *die Definition von* `f[x]`.

`p1 = Plot[f[x], {x,-1.5,1.5}]`

`r:= Random[Real, {-0.1,0.1}]` `Random[]` *(ohne Argument) liefert eine reelle Zufallszahl zwischen 0 und 1.* `Random[Type,Range]` *ist die allgemeinere Form.*

`daten = Table[{x, f[x]+r}, {x,-1.5,1.5,0.05}] // Chop`
Verrauschte Daten. `Chop` *ersetzt reelle Zahlen, die sich von 0 um weniger als* 10^{-10} *unterscheiden, durch 0.*

`p2 = ListPlot[daten]`

`Show[p1,p2]` `Show[..]` *kann Graphikobjekte kombinieren.*

`g[x_] = Fit[daten, {1,x,x^2,x^3,x^4}, x]` `Fit[..]` *liefert einen linearen* least square *Fit.*

```
Plot[{f[x],g[x]},{x,-1.5,1.5},
        PlotStyle -> {RGBColor[0,0,0],RGBColor[1,0,0]}]
```
Originalkurve schwarz, Fitkurve rot.

`daten >> liste.dat` *Mathematica-Objekte lassen sich mit* `>>filename` *speichern.*

`!ls` *So schickt man von einer Mathematica-Sitzung aus Befehle ans Betriebssystem.*

`!Befehl` bewirkt, daß der Befehl ans Betriebssystem geht. `!!filename` bringt den Inhalt von `filename` auf den Bildschirm.

`!!liste.dat`

`daten2 = << liste.dat` *Das Gegenstück zu* `>>` *ist der Befehl* `<<` *. Man*

kann damit gespeicherte Mathematica-Objekte einlesen.

daten2 == daten ⟶ True. == *ist das logische Gleichheitszeichen.*

OutputForm[TableForm[daten]] >> messdaten.dat

!!messdaten.dat messdaten.dat *könnte ein Datenfile sein, das bei einer Messung erzeugt wurde.*

data=ReadList["messdaten.dat",{Real,Real}] ReadList[..] *macht aus Daten eine Mathematica-Liste, hier eine Liste von Zahlenpaaren.*

model[x_] = a + b x + c x^2 + d x^3 + e x^4 *Wir definieren eine Modellfunktion, deren Parameter wir anpassen wollen.*

Needs["Statistics‘NonlinearFit‘"] *Auf diese Weise wird das* package NonlinearFit.m *geladen.*

regel = NonlinearFit[data,model[x],x,{a,b,c,d,e}] *Die Prozedur* NonlinearFit[..] *liefert eine Liste von Regeln, in der die optimalen Parameterwerte stehen.*

h[x_] = model[x] /. regel *So wendet man Regeln an. Die Funktion* h[x] *ist in diesem Fall nahezu identisch mit* g[x].

Einfache *Mathematica*-Programme

Wir gehen davon aus, daß mit einem Texteditor eine Datei programm1.m mit dem folgenden Inhalt erzeugt wurde:

```
(* So schreibt man Kommentare in MATHEMATICA *)

Clear["Global‘*"]      (* Löscht alte Definitionen *)
f[x_]:= Tan[x/4]-1
Plot[f[x],{x,2,4}]
r = FindRoot[f[x]==0,{x,3.1}]  (* 3.1 = Startwert *)
pi = x /. r
```

<< programm1.m ⟶ *Plot von* f[x] *und* 3.14159*, eine Approximation von* π. FindRoot[..] *findet numerische Lösungen von Gleichungen.*

N[pi-Pi] ⟶ 8.88178 10^{-16}

Das nächste Programm, gespeichert als programm2.m, verwendet die *Mathematica*-Funktion Block[..] und berechnet die ersten k Nullstellen der Besselfunk-

tion $J_n(x)$. Es benutzt die Tatsache, daß für $n > 0$ die erste Nullstelle von $J_n(x)$ etwa bei $n + 1.9\,n^{1/3}$ liegt und daß die folgenden Nullstellen alle ungefähr den Abstand π haben.

```
besselnull[n_,k_]:=
    Block[{f,start,regel,nullstelle,liste},
          f[x_] = BesselJ[n,x];
          start = If[n==0, 2.5, n + 1.9 n^(1/3)];
          regel = FindRoot[f[x]==0,{x,start}];
          nullstelle = x /. regel; liste =
{nullstelle};
          While[Length[liste] < k,
                start = liste[[-1]] + Pi;
                regel = FindRoot[f[x]==0,{x,start}];
                nullstelle = x /. regel;
                AppendTo[liste,nullstelle] ];
          If[n > 0, PrependTo[liste,0]];
          Plot[f[x],{x,0,liste[[-1]]+1}];
          liste ]
```

`<< programm2.m`

`besselnull[3,4]` $\longrightarrow$ *Plot von $J_3(x)$ und eine Liste der ersten vier Nullstellen von $J_3(x)$.*

Schleifen und *if*-Anweisungen

`Do[Print["Hello World"],{20}]` $\longrightarrow$ *Schreibt 20 mal* `Hello World` *auf den Bildschirm.*

Schleifen:

```
Do[expr, {'iterator'}]
For[start, test, incr, body]
While[test, body]
Nest[f, expr, n]
NestList[f, expr, n]
FixedPoint[f, expr]
```

`s1 = "  "; s2 = "  ASCII-Code  ";` s1 *und* s2 *sind Strings.*

`Do[Print[s1,FromCharacterCode[j],s2,j],{j,65,75}]`
Schreibt die ersten 11 Großbuchstaben und deren ASCII-Code auf den Bildschirm.

`For[i=1, i < 11, i++, Print[i,"  ",i^2]]` $\longrightarrow$ *Produziert die Zahlen von 1 bis 10 und deren Quadrate.*

`test:= (a = Random[]; a < 0.8)`

`While[test, Print[a]]` $\longrightarrow$ *Im Mittel vier Zufallszahlen zwischen 0 und 0.8.*

`Nest[Sin, t, 3]` $\longrightarrow$ `Sin[Sin[Sin[t]]]`

`Nest[Sin, 1.5, 30]` *Wie oben, jedoch 30 mal iteriert und mit Startwert 1.5.*

`NestList[Sin, 1.5, 30]` `NestList[..]` *ergibt eine Liste aller bei der Iteration berechneten Werte.*

`f[x_]:= N[Cos[x],18]` *Numerischer Wert von* $\cos x$ *mit 18 Stellen.*

`FixedPoint[f,1/10]` $\longrightarrow$ `0.739085133215160642`*, dasjenige* x*, für das mit hinreichender Genauigkeit* $\cos x = x$ *gilt.*

`If[Random[] < 0.5, Print["Zahl < 0.5"]]`

`beep:= Print["\^G"]; beepbeep:=(beep;Pause[1];beep)`

`If[Random[] < .5, beep, beepbeep]`

`theta[x_]:= If[x <= 0, 0, 1]` *Definition der* Θ*-Funktion.*

`Plot[theta[x],{x,-2,2},AspectRatio -> Automatic]`

Differentiation, Integration, Taylorreihen

`Clear["Gobal'*"]`

`D[Exp[x^2], x]` $\longrightarrow$ *die Ableitung von* $\exp(x^2)$ *nach* x.

`Integrate[1/(x^4-1), x]` $\longrightarrow$ *das unbestimmte Integral von* $\dfrac{1}{x^4-1}$.

`Simplify[D[%,x]]` $\longrightarrow$ *der Integrand des obigen Integrals.* `Simplify[..]` *bringt auf den Hauptnenner, kürzt gemeinsame Faktoren und versucht, das Ergebnis auf eine möglichst einfache Form zu bringen.*

`Integrate[Exp[c*x],{x, a, b}]` $\longrightarrow$ *das bestimmte Integral von* $e^{c\,x}$

bezüglich x *in den Grenzen von* a *bis* b.

`Integrate[Sin[Sin[x]],{x, 0, 1}]` *Ohne Erfolg.*

`NIntegrate[Sin[Sin[x]],{x, 0, 1}]` ⟶ `0.430606`, *numerische Integration mit* `NIntegrate[..]`.

`Integrate[x*y*Exp[-(x^2 + y^2)],{x,0,2.0},{y,0,x}]` *Ein Doppelintegral mit dem Wert* `0.120463`. *Es wird zuerst über* y *integriert. Die Grenzen dieser* y*-Integration dürfen die innere Integrationsvariable* x *enthalten.*

`reihe = Series[Tan[x], {x, 0, 10}]` ⟶ *die Reihenentwicklung von* $\tan x$ *um* $x = 0$ *bis zum Term der Ordnung* x^{10} *plus* `O[x]`^{11}, *das die weggelassenen Terme höherer Ordnung repräsentiert.*

`Head[reihe]` ⟶ `SeriesData`, *ein eigenes Mathematica-Objekt, kein Polynom.*

`Normal[reihe]` `Normal[..]` *macht daraus ein Polynom.*

`Timing[Series[Exp[Sin[x]], {x,0,80}];]` ⟶ `{11.54 Second, Null}`

`Timing[Series[Exp[Sin[1.0*x]],{x,0,80}];]` ⟶ `{0.47 Second, Null}`, *numerische Ergebnisse sind oft viel schneller berechenbar als exakte.*

Vektoren, Matrizen, Eigenwerte

`Remove["Global'*"]` `Remove[..]` *ist noch radikaler als* `Clear[..]`.

`v = {x,y,z}` *Vektoren schreibt man als Listen.*

`r = Table[x[j],{j,3}]` ⟶ `{x[1], x[2], x[3]}`

`b = Table[1/(i+j),{i,3},{j,3}]` *Matrizen stellt man dar als Listen von Listen.*

`MatrixForm[b]` ⟶ *die Matrix* `b`.

`d = DiagonalMatrix[{d1,d2,d3}]` ⟶ *eine Diagonalmatrix mit den Diagonalelementen* `d1`, `d2`, `d3`.

`IdentityMatrix[3]` ⟶ *die* 3×3*-Einheitsmatrix.*

`Transpose[b] == b` ⟶ `True`, *weil* `b` *symmetrisch ist.* `Transpose[b]`

ergibt die zu `b` *transponierte Matrix.*

`Det[b]` $\longrightarrow$ *die Determinante von* `b`.

`v.r` $\longrightarrow$ *das Skalarprodukt von* `v` *mit* `r`.

`b.v` $\longrightarrow$ *die Matrix* `b` *angewandt auf den Vektor* `v`.

`b.d` $\longrightarrow$ *das Matrixprodukt* `b` *mal* `d`.

`b.Inverse[b]` $\longrightarrow$ *die Einheitsmatrix,* `Inverse[b]` *liefert die zu* `b` *inverse Matrix.*

`Eigenvalues[b]` $\longrightarrow$ *die Eigenwerte der Matrix* `b`. *Es sind die Nullstellen eines Polynoms 3. Grades, die Mathematica mit der Cardanischen Formel berechnen kann.*

`nb = N[b]` $\longrightarrow$ *die numerische Approximation der Matrix* `b`. *Die exakten rationalen Zahlen sind durch reelle Zahlen mit einer Genauigkeit von* 10^{-16} *ersetzt worden.*

`Eigenvalues[nb]` $\longrightarrow$ `{0.875115, 0.0409049, 0.000646659}`, *eine Liste der Eigenwerte der Matrix* `nb`.

`u = Eigenvectors[nb]` $\longrightarrow$ *eine Liste der Eigenvektoren von* `nb`.

`Chop[u.Transpose[u]]` $\longrightarrow$ *die Einheitsmatrix, denn die Eigenvektoren von* `nb` *sind orthonormal, bis auf numerische Ungenauigkeiten von der Größenordnung* 10^{-16}.

`Sort[Thread[Eigensystem[nb]]]` $\longrightarrow$ *Eigenwerte und zugehörige Eigenvektoren, sortiert nach der Größe der Eigenwerte.*

Lösen von Gleichungen

`FindRoot[Cos[x]==x,{x,1}]` $\longrightarrow$ `{x -> 0.739085}`, *die Lösung der Gleichung* $\cos x = x$. *Die Klammer* `{x,1}` *bedeutet: suche eine Lösung bezüglich* `x` *und starte die Suche bei* `x=1`.

`FindRoot[{Cos[a*x]==x,-a*Sin[a*x]==-1}, {x,1}, {a,1}]`
Auch Gleichungssysteme lassen sich mit `FindRoot[..]` *lösen.*

`Plot[Product[(1+x/j^2),{j,Infinity}],{x,1,4}]` $\longrightarrow$ *eine glatte Funktion, die im betrachteten Bereich monoton ansteigt.*

`FindRoot[Product[(1+x/j^2),{j,Infinity}]==20,{x,3}]` $\longrightarrow$

eine Fehlermeldung, weil sich die Funktion nicht symbolisch differenzieren läßt.

`FindRoot[Product[(1+x/j^2),{j,Infinity}]==20,{x,{2,3}}]`
So geht's. Wenn man statt eines Startwertes ein Startintervall angibt, wird das Sekantenverfahren benutzt.

> `FindRoot[lhs == rhs, {x, x0}]` findet numerische Lösungen.
>
> `Solve[eqns, vars]` versucht, exakte Lösungen zu finden.
>
> `NSolve[eqns, vars]` Numerische Lösungen algebraischer Gleichungen.
>
> `LinearSolve[m, b]` löst das Gleichungssystem `m.x == b`
>
> `DSolve[eqn, y[x], x]` löst Differentialgleichungen.
>
> `NDSolve[eqns, y, {x, xmin, xmax}]` findet numerische Lösungen von Differentialgleichungen.

`Solve[a*x^2+b*x+c==0,x]` $\longrightarrow$ *die 2 Lösungen der quadratischen Gleichung.*

`Solve[ArcSin[x]==a,x]` $\longrightarrow$ `{{x -> Sin[a]}}`

`Solve[x^5-5x^2+1 == 0, x]` *Zu schwierig.*

`NSolve[x^5-5x^2+1 == 0, x]` $\longrightarrow$ *5 numerische Nullstellen.*

`DSolve[y''[x]==y[x], y[x], x]` $\longrightarrow$ *die allgemeine Lösung dieser Differentialgleichung.*

`DSolve[y'[x]==Cos[x*y[x]], y[x], x]` *Fehlanzeige.*

`NDSolve[{y'[x]==Cos[x*y[x]],y[0]==0},y[x],{x,-5,5}]` $\longrightarrow$ *ein Mathematica-Objekt, das* InterpolatingFunction *heißt.*

`f[x_] = y[x] /. %` *So kann man aus dem Objekt* InterpolatingFunction *eine normale Funktion machen, die sich mit* `Plot[f[x],{x,-5,5}]` *plotten läßt.*

Muster, Ordnen, Sortieren

`Remove["Global'*"]`

`liste = {1, 2, f[a], g[b], x^n, f[b], 3.4, Sin[p]^2}`

`Position[liste, f[_]]` $\longrightarrow$ `{{3},{6}}`, *nämlich eine Liste aller Plätze mit* `f[`*irgendwas*`]`.

Cases[liste,_^2] ⟶ {Sin[p]^2}, *alle Fälle mit Potenz 2.*

Count[liste,_^_] ⟶ 2, *die Anzahl aller Fälle mit irgendeiner Potenz.*

DeleteCases[liste,_[b]] ⟶ {1,2,f[a],x^n,3.4,Sin[p]^2}, *irgendwas* [b] *fällt heraus.*

Select[liste,IntegerQ] ⟶ {1,2}, *die Fälle, für die* IntegerQ[..] *den Wert* True *gibt. Es gibt weit über 30 Ausdrücke der Form* *Q, *mit denen man Fragen stellen kann.*

?*Q ⟶ *alle diese Fragen.*

Select[liste,NumberQ] ⟶ {1,2,3.4}. NumberQ[*Zahl*] *ergibt* True.

Select[liste,!NumberQ[#]&] ⟶ *alles, außer den Zahlen. Das Ausrufezeichen (* ! *) ist die logische Verneinung. Die Konstruktion* expr[#]& *macht aus* expr *einen Operator, eine sogenannte* pure function. *Für* # *wird das Argument eingesetzt.*

term = Expand[3*(1+x)^3*(1-y-x)^2] Expand[..] *multipliziert aus.*

Factor[term] ⟶ *das Argument vom vorigen* Expand[]. *In gewisser Weise ist* Factor *das Gegenstück zu* Expand.

FactorTerms[term] *klammert Zahlenfaktoren aus.*

Collect[term,y] *ordnet nach Potenzen von* y.

Coefficient[term,x^2*y^2] ⟶ 2, *der Koeffizient von* x^2 y^2.

CoefficientList[term,x] → *die Liste der Koeffizienten der* x*-Potenzen.*

Apart[1/(x^4-1)] ⟶ *die Partialbruchzerlegung von* $\frac{1}{x^4-1}$.

Together[%] Together[..] *bringt auf den Hauptnenner.*

N[Sort[{Sqrt[2],2,E,Pi,EulerGamma}]] ⟶ *Ergebnis:*
{2., 1.41421, 2.71828, 0.577216, 3.14159}. *Nach welchen Gesichtspunkten wird hier sortiert?*

kleiner:= (N[#1] < N[#2])& *ordnet nach dem numerischen Wert.*

N[Sort[{Sqrt[2],2,E,Pi,EulerGamma},kleiner]] *Jetzt stimmt's.*

Anhang B
Erste Schritte mit C

C ist die Programmiersprache der UNIX-Rechner. Während FORTRAN hauptsächlich für extensive numerische Rechnungen benutzt wird, erlaubt C einen übersichtlichen, kompakten und funktionsorientierten Programmierstil und die Benutzung maschinennaher Graphikumgebungen. Ferner hat C (ebenso wie PASCAL) den Vorteil, daß es für den PC eine bequeme Umgebung zur Programmentwicklung mit eingebauter Graphikbibliothek gibt.

C hat jedoch auch Nachteile: Zum einen ist es nicht für wissenschaftliches Rechnen entwickelt worden. Potenzen werden mit einer Funktion berechnet, die Mathematik-Bibliothek muß beim Compilieren immer mit aufgerufen werden und komplexe Zahlen, Vektor- und Matrixmultiplikationen muß man sich selbst definieren. Zum anderen muß man oft Speicheradressen verwenden, und C kontrolliert nicht, ob Indizes versehentlich den vorher definierten Bereich überschreiten. Insbesondere der Anfänger muß sich sehr vor dem letzten Punkt in acht nehmen: C toleriert sehr viele Fehler, und man hat daher große Mühe, die Ursache für offensichtlich falsche Ergebnisse herauszufinden. Andererseits läßt C dem Programmierer viele Freiheiten, die er zu seinem Vorteil nutzen kann.

Wir wollen hier, hautpsächlich mit einfachen Beispielen, eine erste Einführung in C geben. Ein Programm muß immer die Funktion `main( )` enthalten, deren Anweisungen in den darauffolgenden Klammern { ... } stehen. Der Typ aller Variablen und Funktionen muß deklariert werden sein, z. B. mit

```
int i, antwort;
char ch;
float f, rationale_Zahl;
long Langes_i;
double Langes_f;
```

C unterscheidet zwischen Groß- und Kleinbuchstaben. Die Variablen hinter `int`, `char` und `long` sind ganze Zahlen (Integer), diejenigen nach `float` und `double` sind rationale Zahlen. Die verschiedenen Typen markieren verschiedene Genauigkeiten bzw. Größen des zugewiesenen Speicherplatzes, die von der Maschine abhängen. Normalerweise sind für `char` ein Byte (= 8 Bits = Speicherbedarf für ein ASCII-Zeichen), für `int` zwei und für `long` vier Bytes reserviert.

Als erstes Beispiel wollen wir die Zahlen 4, 5, .., 15 auf dem Bildschirm ausdrucken: Wir schreiben in eine Datei `druck.c` die Zeilen

```
main( )
 {
    int i;
    for( i=4; i<16; i=i+1) printf("i=%d \n",i);
 }
```

Jeder Befehl muß durch ein Semikolon abgeschlossen werden. Die Iterationsschleife `for` enthält in runden Klammern die Struktur: (Start; Abbruchbedingung; Befehl nach jedem Durchlauf). Hinter der runden Klammer erscheint der auszuführende Befehl; mehrere solcher Befehle müssen mit {...} geklammert werden, wobei hinter jedem Befehl das Semikolon stehen muß.

Dieses Programm wird mit

```
cc  -o druck druck.c
```

compiliert, und der Befehl `druck` schreibt die Zahlen auf den Bildschirm. Die Druckfunktion `printf` erwartet als erstes Argument eine Zeichenkette `"..."`, die sowohl den Text, als auch die Formatanweisung für die zu druckende Variable `i` enthält. Je nach Typ der Variablen sind folgende Druckanweisungen gebräuchlich:

	Variablentyp	
%d	int	
%ld	long	
%c	char	
%f	float	(Fließkomma-Darstellung)
%e	float	(Exponential-Darstellung)
%lf	double	
%s	string	

Das Symbol `\n` ist ein Steuerzeichen, das einen Zeilenvorschub bewirkt. Es entspricht der RETURN-Taste.

Die obigen Druckanweisungen sind auch für das Einlesen zu verwenden, das die Funktion `scanf( )` übernimmt:

```
main( )
  {      double i = 0.;
        for(; i<100.;)
         {   printf( "Bitte Zahl eingeben \n");
```

```
                scanf( "%lf", &i);
                printf( "Sie haben %lf eingegeben",i);
            }
    }
```

Dieses Programm zeigt einige Besonderheiten: 1. In der Typ-Deklaration kann die Variable auch gleich initialisiert werden. 2. In der `for`-Schleife können auch leere Start- und Inkrementanweisungen stehen. 3. Übergibt man eine Variable an eine Funktion, hier `i` an `scanf`, so wird nur der Wert übergeben. Also kann `scanf` keine neue Zahl auf den Speicherplatz von `i` schreiben. Das geht nur, wenn die Funktion die Addresse (und nicht den Wert) der Variablen enthält: dann kann `scanf` auf den Speicherplatz von `i` die eingetippte Zahl schreiben. Adressen erhält man durch ein vorgestelltes `&`.

Das Programm liest so lange Zahlen ein und schreibt sie dann, bis eine Zahl größer als 100 ist. Wenn `scanf` die eingegebenen Zeichen nicht in eine reelle Zahl umwandeln kann, wartet die Funktion auf die folgende Eingabe.

Man kann natürlich auch Text einlesen und ausdrucken. Dazu muß man für jedes Zeichen einen Speicherplatz vom Typ `char` reservieren, z. B. als Vektor.

```
main()
 {   char str[100];
     printf( "Wie heissen Sie?");
     scanf( "%s", str);
     printf( "Guten Tag, %s", str);
 }
```

`str[100]` ist ein Vektor mit Platz für 100 Zeichen. `str[0]` ist das erste und `str[99]` das letzte Zeichen. `str` ist die Adresse des Vektors (= Adresse des ersten Platzes, also `&str[0]`), deshalb reicht es, in `scanf` den Namen des Vektors ohne `&` zu übergeben. `scanf` liest alle Zeichen ein bis zu einem Leerzeichen oder einem RETURN, dann wird das Ende der Zeichenkette mit dem ASCII-Zeichen `\0` (= Null) markiert. `printf` druckt alle Zeichen bis zu dem Zeichen `\0`.

Nach diesen beiden ersten Versuchen wollen wir eine Liste von Befehlen und Bemerkungen folgen lassen, mit denen erste kleine Programme machbar sein sollten.

```
#include <math.h>
#include "myfile"
#define  N 1000
```

Diese Anweisungen werden vor dem Compilieren ausgeführt. `#include` fügt an dieser Stelle die benannten Dateien hinzu, <···> sucht in speziellen Pfaden, `"···"` sucht im momentanen Pfad, in dem sich das Programm befindet. Alle benutzten Funktionen müssen deklariert sein, das geschieht in den entsprechenden *Header*-Dateien `math.h, stdlib.h, graphics.h, time.h` usw. Durch die Zeile

`#define N 1000` wird im gesamten Programm das Symbol `N` durch das Symbol `1000` ersetzt.

```
summe = a+b;
a = b = a/b * e;
mod = a % b;
```

Das Gleichheitszeichen ordnet der linken Variablen den rechten Wert zu. `%` ist die Modulo-Operation, `mod` erhält also den Rest von `a/b`.

```
summe += a;
a++; b--;
```

C enthält einige Abkürzungen: Die obigen Befehle ersetzen `summe = summe+a; a = a+1; b = b-1;`

`a>b;`	größer als
`a>=b;`	größer oder gleich
`a==b;`	gleich
`a!=b;`	ungleich
`a<=b;`	kleiner oder gleich
`a<b;`	kleiner als

Wenn die obigen Relationen wahr sind, erhält der Ausdruck den Wert 1, sonst den Wert 0. Für logische Ausdrücke gibt es keinen speziellen Datentyp, C arbeitet einfach mit ganzen Zahlen vom Typ `int`.

`a<b || c!=a`	oder
`a<b && c!=a`	und
`!(a<b)`	nicht

Logische Operatoren verknüpfen logische Werte und erzeugen 1 (wahr) oder 0 (falsch). Die logische Negation ! macht aus 0 eine 1 und umgekehrt.

```
if(b==0)   printf ("Division ist nicht definiert");
   else    printf ("Ergebnis = %lf", a/b);
if (b>1.e-06)
     {  scanf("%lf",&a);
        printf("a geteilt durch b=%lf", a/b);
     }
```

Die Anweisung `if` führt den folgenden Befehl nur dann aus, wenn der Klammerausdruck wahr ist (einen Wert ungleich 0 hat). Falls die Klammer den Wert 0 hat,

wird der Befehl nach `else` ausgeführt. Der `else`-Zweig muß nicht angegeben werden, und mehrere Befehle muß man als Block {...} zusammenfassen, wie das zweite Beispiel zeigt.

```
switch (ch=getch())
    {
     case'e':  exit(0);
     case'f':  scanf("%lf",&a); break;
     case's':  scanf("%s",str); break;
     default:  printf("Sie haben %c eingegeben",ch);
               break;
    }
```

Mehrere, eventuell verschachtelte `if...else...if...else...` Anweisungen kann man übersichtlicher durch einen `switch` Befehl ersetzen. Es wird das nächste Tastaturzeichen durch die Funktion `getch()` aufgerufen, an die Variable `ch` übergeben und diejenige `case` Anweisung ausgeführt, die den entsprechenden Wert von `ch` enthält. Falls der Wert nicht vorhanden ist, wird die `default` Anweisung ausgeführt. Jeweils der letzte Befehl sollte `break;` sein. Damit springt das Programm an das Ende des `switch` Blockes {...}.

```
for (summe=0., i=0; i<N; i++) summe +=feld[i]; for(;;);
```

Die `for`-Schleife haben wir schon kennengelernt. Die ersten zwei Anweisungen, durch ein Komma getrennt, werden beim Start ausgeführt. Die Schleife bricht ab, wenn der Wert nach dem ersten Semikolon Null (also falsch) ist. `i++` wird nach jedem Schleifendurchgang ausgeführt. Das zweite Beispiel gibt eine unsinnige Endlosschleife, die nur durch den gewaltsamen Abbruch des Programmes mit `Ctrl-c` (Control- und c-Taste gleichzeitig drücken) gestoppt werden kann.

```
while(ch!='e')
  { ch=getch();
    printf ("Das Zeichen %c wurde eingetippt \n",ch);
  }
```

Die `while`-Schleife ist eine weitere wichtige Iteration von Befehlen. Sie wird solange ausgeführt, wie der Wert in der runden Klammer ungleich 0, also wahr ist. Die Funktion `getch()` liest ein Zeichen von der Tastatur und gibt es an die Variable `ch`. Dann wird es gedruckt. Falls das Zeichen `'e'` eingetippt wird, wird die Schleife beendet.

```
double v[6];
double u[5]={1.0, 1.5, 0., 6, -1.};
      v[5]=u[1] * u[0];
```

Bei der Deklaration von Vektoren muß der Typ und die Länge des Vektors angegeben werden. Die Indizes laufen von 0 bis (Länge-1) und werden mit eckigen Klammern angesprochen. `v[5]` hat also den Wert 1.5, `v[6]` ist nicht definiert. Vektoren können bei der Deklaration initialisiert werden.

```
double mat[6][5];
     for(i=0;i<6;i++)
       {
         for(sum=0., j=0;j<5;j++)
             sum+= mat[i][j]*u[j];
         v[i]=sum;
        }
```

Matrizen werden durch zwei Indizes deklariert und angesprochen. Für jeden Index ist eine separate eckige Klammer erforderlich. Der erste Index gibt die Zeile, der zweite die Spalte der Matrix an. Im obigen Beispiel ist der Vektor `v` das Ergebnis der Multiplikation der Matrix `mat` mit dem Vektor `u`. Die Matrix hat 6 Zeilen $i = 0, ..., 5$ und 5 Spalten $j = 0, ..., 4$. `mat` enthält die Adresse des ersten Elementes `mat[0][0]` und `mat[i]` diejenige der i-ten Zeile, also des Elementes `mat[i][0]`. Das ist wichtig, wenn man Matrizen oder Zeilen davon an Funktionen übergeben will.

```
double *a, wert, b;
    a = &b;
    scanf ("%lf", a);
    wert = *a;
```

Es gibt in C Variablen, die die Adresse und nicht den Wert eines Speicherplatzes enthalten. Sie werden durch `Typ *` deklariert, d. h. die Variable `a` enthält die Adresse eines Speicherplatzes vom angegebenen Typ, oder `a` zeigt auf einen Platz vom Typ `double`. Solche Variablen werden auch Zeiger genannt, und ihr Wert wird mit `*a` angesprochen. Im obigen Beispiel wird die Adresse von `b` an `a` übergeben, dann wird auf diesen Platz eine reelle Zahl eingelesen, und der Wert von `a` wird an `wert` übergeben. Dasselbe Ergebnis für `wert` hätte auch der Befehl `scanf("%lf", &wert)` geliefert.

In der folgenden Tabelle sind `x` und `y` vom Typ `double`. Winkel werden bei trigonometrischen Funktionen im Bogenmaß angegeben.

`sin(x)`	Sinus von x
`cos(x)`	Kosinus von x
`tan(x)`	Tangens von x
`asin(x)`	$\arcsin(x)$ im Bereich $[-\pi/2, \pi/2]$, $x \in [-1, 1]$.
`acos(x)`	$\arccos(x)$ im Bereich $[0, \pi]$, $x \in [-1, 1]$.

atan(x)	$\arctan(x)$ im Bereich $[-\pi/2, \pi/2]$.
atan2(y,x)	$\arctan(y/x)$ im Bereich $[-\pi, \pi]$.
sinh(x)	Sinus Hyperbolicus von x
cosh(x)	Cosinus Hyperbolicus von x
tanh(x)	Tangens Hyperbolicus von x
exp(x)	Exponentialfunktion e^x
log(x)	natürlicher Logarithmus $\ln(x)$, $x > 0$.
log10(x)	Logarithmus zur Basis 10, $\log_{10}(x)$, $x > 0$.
pow(x,y)	x^y
sqrt(x)	$\sqrt{x}$, $x \geq 0$.
fabs(x)	absoluter Wert $\|x\|$
fmod(x,y)	Rest von x/y

Diese mathematischen Funktionen geben einen Funktionswert vom Typ `double` zurück. Sie müssen vorher mit `#include <math.h>` deklariert werden. Falls das Argument nicht vom Typ `double` ist, wird es automatisch umgewandelt.

```
double r;
r = rand() / (RAND_MAX +1.);
```

Die Funktion `rand()` liefert ganzzahlige gleichverteilte Pseudo-Zufallszahlen zwischen `0` und `RAND_MAX`. Braucht man reelle Zufallszahlen im Intervall `[0,1]`, so kann man `rand()` durch `RAND_MAX` dividieren. Allerdings muß man dazu den Nenner in eine reelle Zahl umwandeln, sonst ist das Ergebnis die ganze Zahl 0. Die Addition von `RAND_MAX` und der reellen Zahl `1.` ergibt als Resultat eine Zahl vom Typ `float`. Die Funktion `rand()` und die Konstante `RAND_MAX` müssen mit `#include <stdlib.h>` deklariert bzw. definiert werden.

```
#include <math.h>
main()
{
  double f(double),x;
  for(x=0.;x<10.;x+=.5)
         printf("\n f(%3.1lf) = %4.2e ",x,f(x));
}

double f(double xxx)
{
   double x2;
   x2=xxx*xxx;
   return (x2*exp(-x2/2));
}
```

Man kann eigene Funktionen definieren, hier $f(x) = x^2\,e^{-x^2/2}$. Dazu muß die Funktion in der aufrufenden Funktion (hier `main()`) deklariert und dann definiert werden, jeweils mit den Typenangaben, auch für die Argumente. Die Befehle der Funktion $f(x)$ stehen in einem Block $\{...\}$, und der Wert wird mit `return` an den aufrufenden Funktionsnamen zurückgegeben. Wird kein Wert zurückgegeben oder kein Argument übergeben, so wird der Typ `void` benutzt. Es wird nur der Wert des Argumentes `x` übergeben, `f` kann hier also nicht den Wert der Variablen `x` ändern. Wird dagegen die Adresse von `x` , also `&x` übergeben, dann kann `f` auch den Wert von `x` ändern:

```
#include <math.h>
main()
{
  double f(double *),x=0.;
  while (x<100.) printf(" \n f(%lf)=%lf",x,f(&x));
}

double f(double *adresse)
{
  double x2;
  x2=(*adresse)*(*adresse);
  *adresse+=.5;
  return x2*exp(-x2/2.);
}
```

Hier bedeutet `double*` einen Zeiger auf eine Variable vom Typ `double`. Die Übergabe der Adresse anstatt des Wertes ist notwendig, wenn Vektoren, Matrizen oder sogar Funktionen an eine Funktion übergeben werden sollen.

```
main()
{
   double u[100],v[100];
   double ScalarProduct(double *, double *,int);
   int n=20,i;

   for(i=0;i<n;i++) u[i]=v[i]=i;
   printf(" \n %lf \n",ScalarProduct(u,v,n));

}
```

```
double ScalarProduct(double u[], double v[], int n)
{
  int i;
  double summe;
  for(i=0,summe=0.;i<n;i++) summe += u[i]*v[i];
  return summe;
}
```

Dieses Beispiel zeigt, wie Vektoren an Funktionen übergeben werden. `double u[]` zeigt dem Compiler, daß die Funktion `ScalarProduct` eine Adresse auf das erste Element eines Vektors vom Typ `double` erhält; `double *u` ist identisch damit. Es wird in der Funktion kein Speicherplatz für einen Vektor reserviert, sondern nur mit der Adresse des ersten Platzes gearbeitet. `u[3]` zählt diese Adresse drei Plätze weiter und gibt den Wert dieses Platzes zurück. Die Funktion weiß nicht, wie groß der in `main()` reservierte Speicherplatz ist; man muß daher die Länge n der Vektoren angeben.

```
double MatrixFunktion(double mat[][10],int zeilen,
     int spalten)
{ ... }
```

Bei Matrizen wird zwar auch nur die Adresse des ersten Elementes übergeben, also `&mat [0][0]`, aber das Programm muß die Länge der Zeilen (hier 10) wissen, um jedes Element ansprechen zu können. Denn das Element `mat[i][j]` steht im Speicherplatz mit der Adresse `&mat[0][0] + i*10 + j`. Die Anzahl der reservierten Zeilen muß nicht bekannt sein.

```
/* So
     besser
     nicht */
double u[100],v[100],ScalarProduct(void);int n=20,i=0;
main(){for( ; i<n; i++) u[i]=v[i]=i*i;
printf("%lf",ScalarProduct());}
double ScalarProduct(void){double sum=0.;
for(i=0;i<n;i++)sum+=u[i]*v[i];return sum;}
    /* Das ist
       ein Kommentar */
```

Variable können nur innerhalb der Funktion benutzt werden, innerhalb derer sie definiert sind. Sollen sie in mehreren Funktionen gültig sein, so müssen sie vor diesen Funktionen außerhalb der geschweiften Klammern deklariert worden sein. `u,v,n,i` und `ScalarProduct` kann man also (auch mit Werten) in allen Funktionen benutzen, `sum` dagegen nur in `ScalarProduct`. `void` bedeutet, daß keine Variablen übergeben werden. Natürlich kann die Funktion jetzt nicht auf andere

Vektoren x und y angewendet werden, sie steht also nur als Abkürzung für einen Block von Befehlen.

Vorsicht: Fallen!

Wir wollen hier einige Standardfehler auflisten, die leider oft keine Fehlermeldung produzieren.

```
b = mat[i,j];
```

Matrixelemente müssen durch zwei Indizes jeweils in eckigen Klammern angesprochen werden, also `mat[i][j]`. Der obige Befehl ruft `i` auf, dann `j` und weist das Element `mat[j]` der Variablen `b` zu. `mat[j]` ist aber die Adresse der j-ten Zeile (= `&mat[j][0]`)!

```
if(a=b)   dosomething();
```

Der Programmierer meinte sicherlich `a==b`. Der obige Befehl weist `b` der Variablen `a` zu und prüft dann, ob der Wert von `a` gleich Null ist. Falls nicht, wird die Funktion `dosomething()` aufgerufen.

```
ganze_zahl = 3 * rand() / RAND_MAX;
```

Alle Variablen und Konstanten sollen vom Typ *Integer* sein. Manche Compiler werten diesen Ausdruck von rechts nach links aus. In diesem Fall erhält der Quotient und damit die gesamte Rechnung den Wert Null, da beim Ergebnis der Division zweier ganzer Zahlen der Rest verloren geht. Also sollte man das Produkt in Klammern setzen, dann erhält man auf jeden Fall die Zufallszahlen 0, 1 und 2.

```
int vector[10],i;
for (i=1; i<=10; i++)  vector[i]=i*i;
```

Vektoren der Länge N werden mit `i=0` bis `i=N-1` indiziert, `vector[10]=100;` weist ohne jede Warnung dem Speicherplatz nach dem für `vector` reservierten Bereich den Wert 100 zu, und der Programmierer weiß nicht, was er da gerade überschreibt!

```
main()
{
     char *name;
     printf("Wie heissen Sie? ");
     scanf( "%s",name);
}
```

`name` ist als Adresse (=Zeiger) auf eine Zeichenvariable deklariert, aber nirgends wird der Zeiger initialisiert. Das heißt, in der Variablen `name` steht irgendeine unbekannte Zahl, die als Adresse interpretiert wird, und ab dieser Adresse wird ihre Eingabe geschrieben und zerstört damit andere Daten. Vor diesem Fehler warnt der Compiler, er führt ihn aber aus.

```
    char  msg[10];
    msg ="Hallo";
```

Namen von Vektoren sind zwar Zeiger, aber keine Variablen. `"Hallo"` markiert die Adresse der Zeichenkette, jedoch darf die Adresse des Vektors `msg` nicht geändert werden. Der Compiler erzeugt eine Fehlermeldung. Folgendes Programm beseitigt beide vorherigen Fehler

```
main()
{
    char name[20],*msg;
    printf(" \n Wie heissen Sie ?");
    scanf("%s",name);
    msg="Hello";
    printf("\n %s %s \n", msg,name);
}
```

Der Unterschied zwischen `"x"` und `'x'`

Der erste Ausdruck ist eine Zeichenkette, er besteht aus den beiden Zeichen `'x'` und `\0`. Der zweite Ausdruck ist ein einzelnes Zeichen.

```
    int a=100;
    scanf("%d",a);
```

`scanf` interpretiert `a` als Adresse Nr. 100 und überschreibt das, was dort gerade steht. Richtig lautet der Einlesebefehl `scanf("%d",&a);`.

Literatur

B.W. Kernighan, D.M. Ritchie, *Programmieren in C*, Carl Hanser Verlag, 1990.

Anhang C
Erste Schritte mit UNIX

Einführung

Dieser Anhang soll dem Anfänger einen kurzen Überblick über die wichtigsten UNIX-Befehle geben. Zusätzlich wird erklärt, wie man über das Internet an die in diesem Buch beschriebenen Beispielprogramme kommt, wie man diese editieren und kompilieren kann.

Unter UNIX gibt es mittlerweile viele *public domain* Programme, d.h. Programme, die man für nichtkommmerzielle Zwecke frei kopieren darf. Davon haben sich etliche zu Standardprogrammen entwickelt. Einige werden auch hier beschrieben, wie `less`, `gzip` und `gcc`, `g++`. Wenn diese Programme bei Ihnen nicht verfügbar sind, können Sie den Systembetreuer bitten, diese zu installieren. Diese Programme sind mit (*) gekennzeichnet.

UNIX ist ein Multiuser- und Multitasking-Betriebssystem. Es können mehrere Benutzer unabhängig voneinander zur selben Zeit arbeiten und mehrere Programme „im Hintergrund gleichzeitig“ gerechnet werden. „Gleichzeitig“ bedeutet, daß diese Prozesse sich die Rechenzeit aufteilen.

Um an einem UNIX System arbeiten zu können, braucht man einen Benutzernamen *Username* und ein Kennwort *Password*. Diese bekommt man vom Systembetreuer. Jeder Benutzer hat ein eigenes Verzeichnis *Home-Directory* auf der Festplatte, in dem er seine Programme und Dateien speichern kann.

Nachdem man am Rechner den Benutzernamen und sein Passwort eingetippt hat, befindet man sich entweder in einer *shell* – das heißt, der Rechner wartet auf Kommandos – oder es wird eine Graphikoberfläche (üblicherweise X11) gestartet. Falls im letzteren Fall nicht automatisch ein Eingabefenster geöffnet wurde, muß man mit der Maus ein Fenster erzeugen. Dann können die UNIX-Befehle in das Fenster eingetippt und mit RETURN abgeschickt werden. Fenster können mit den Maustasten verschoben, in alle Richtungen vergrößert und verkleinert und als *icon* in den Hintergrund gebracht werden.

Folgende Befehlsgruppen werden in diesem Anhang beschrieben:

- Manipulieren von Dateien

- Kommunikation mit anderen Rechnern

- Compiler (zur Übersetzung eigener Programme)

Nichtbeschriebene Befehlsgruppen sind:

- Kommunikation mit der Peripherie (z. B. drucken)
- Überwachung und Steuerung der im „Hintergrund“ laufenden Programme
- Kommunikation mit anderen Benutzern (z. B. Electronic Mail)

Es sei darauf hingewiesen, daß es bei den Graphikoberflächen oft auch Hilfsprogramme (sogenannte *Filemanager*) gibt, die die Dateiverwaltung leichter machen.

Dateien

Eine Datei ist eine Menge von Zeichen. Diese können Textdaten, Zahlenmengen oder auch Programme bedeuten. Die Einheit, in der meistens die Länge von Dateien gemessen wird, ist das *Byte*. Ein Byte ist eine Zahl zwischen 0 und 255. Üblicherweise benötigt ein Zeichen ein Byte und eine Zahl zwei bis vier Bytes Speicherplatz in einer Datei.

Jede Datei hat einen Namen. Dieser kann unter UNIX auf fast allen Systemen mehr als 8 Buchstaben Länge haben. Groß- und Kleinschrift wird unterschieden. Dateien besitzen ferner Attribute wie: Name des Eigentümers, Name der Gruppe, der sie zugewiesen ist und Rechte des Eigentümers, der Gruppe und aller anderen Benutzer. Man kann zum Beispiel allen anderen Benutzern (abgesehen vom Systembetreuer) verbieten, die Datei zu lesen.

Die Dateien sind hierarchisch in Verzeichnisse (*Directories*) gegliedert. Diese kann man mit Schubladen vergleichen. Die Wurzel (*root*) dieses Verzeichnisbaumes hat das Symbol /. Unterverzeichnisse werden mit / getrennt.

Eine tyische Baumstruktur wäre zum Beispiel:

```
/  —  /etc    —  /etc/motd
                  /etc/passwd
                  ...
      /users  —  /users/user1  —  /user/user1/.cshrc
                                  /user/user1/.login
                                  ...
                  ...
      ...
      /unix
```

Dabei merkt der Benutzer nicht, ob die Verzeichnisse eigene Platten oder gar Platten an entfernten Rechnern sind. Die Shell merkt sich ein aktuelles Verzeichnis *current directory*. Dieses ist am Anfang das eigene Verzeichnis. Dateinamen, die mit / beginnen, werden absolut von der Wurzel aus bezeichnet. Beginnt ein Dateiname mit

˜/, so wird stattdessen das eigene Verzeichnis eingesetzt. Beginnt er mit ˜*user*/, wird er relativ zu dem angegebenen Benutzer *user* gesehen. Alle anderen Dateinamen werden relativ zu dem aktuellen Verzeichnis gesehen. Ein Punkt **.** bezeichnet das aktuelle und zwei Punkte **..** das übergeordnete Verzeichnis.

Die wichtigsten Befehle

Befehle in UNIX setzen sich aus Kommandos, Optionen und Parametern zusammen. Dabei wird den Optionen das Zeichen - vorangesetzt. Sollen mehrere Optionen benutzt werden, können diese meistens hinter ein einziges Zeichen - geschrieben werden. Die Optionen müssen vor den Parametern kommen. Zum Beispiel gibt `ls -a dir` alle Dateien des Verzeichnisses `dir` aus.

Dateinamen können *Joker* enthalten. Diese werden von der shell ergänzt. Das Zeichen * steht für beliebig viele beliebige Zeichen. So werden mit dem Befehl `ls m*.c` alle Programme ausgegeben, die mit dem Buchstaben `m` beginnen und mit `.c` aufhören.

Die Dateinamen werden oft so gewählt, daß man erkennt, um was es sich handelt: Dazu wird üblicherweise ein Punkt und ein Kürzel *extension* an den Namen angefügt. Die wichtigsten Extensions sind:

.c C-Programmtext (*source code*)

.f Fortran-Programmtext

.p Pascal-Programmtext (manchmal auch .pas)

.o Objektdatei (übersetzter Programmtext, noch nicht aufrufbar)

.m Textdatei mit *Mathematica*-Befehlen

.tar ein Programm-Archiv mit `tar` erstellt

.gz eine mit `gzip`(*) gepackte Datei

.z eine mit `compress` gepackte Datei

Hilfen

Mit dem Befehl `man` kann man sich die Beschreibung eines Befehles seitenweise auf dem Bildschirm anzeigen lassen. Dabei werden Syntax und auch Optionen und Parameter erklärt. Zum Beispiel erhält man mit dem Kommando `man man` die Beschreibung des `man` Kommandos.

Weiß man nur ein Stichwort, nicht aber den genauen Befehl, kann man mit `man -k stichwort` eine Liste der Befehle erhalten, die in ihrer einzeiligen Beschreibung das Wort `stichwort` enthalten.

Dateimanipulation

Eine Datei kann auf komfortable Weise mit einem sogenannten Editor erstellt, bzw. geändert werden. Dieses wird später beschrieben. Textfiles können mit `cat`, `more` oder `less`(*) auf dem Bildschirm angezeigt werden. Dabei wird bei `more` die Datei seitenweise angezeigt, und bei `less` kann man sogar mit Tastendruck beliebig in dem Dokument umherspringen.

Das Kopieren einer Datei kann mit dem Unix-Befehl *copy* geschehen. Die Syntax ist: `cp eingabe ausgabe`. Hierbei wird die Datei `eingabe` in die Datei `ausgabe` kopiert. Um Verzeichnisse zu kopieren muß die Option `-R` angegeben werden. Dabei werden die tieferliegenden Dateien rekursiv kopiert.

Dieselbe Syntax hat der *move* Befehl, der es ermöglicht, Dateien umzubenennen, bzw. innerhalb verschiedener Directories zu verschieben: `mv alt neu` benennt die Datei `alt` in `neu` um. Dies kann auch mit Directories geschehen.

Eine Datei kann man mit dem `rm` Befehl löschen (*remove*). `rm file` löscht die Datei `file`. Diese kann dann nicht mehr wiederhergestellt werden. `rm *.txt` löscht im aktuellen Verzeichnis alle Dateien mit der Extension `txt`. Ganze Directories einschließlich der darin enthaltenen Dateien kann man mit dem Befehl `rm -R` rekursiv löschen. Hierbei sollte man aber sehr aufpassen, daß nicht aus Versehen wichtige Daten gelöscht werden. Anfangs sollte man zusätzlich die Option `-i` benutzen. Hier wird „interaktiv“ bei jeder Datei explizit nachgefragt, ob diese auch wirklich gelöscht werden soll, also zum Beispiel: `rm -Ri verzeichnis`.

Information über Verzeichnisse, aktuelles Verzeichnis

Das aktuelle Verzeichnis kann man mit `pwd` (*present working directory*) anzeigen lassen. Alle Dateinamen, die nicht mit / beginnen, werden wie schon beschrieben relativ zu diesem gesehen. `cd dir` setzt das aktuelle Verzeichnis auf `dir`. Wird der Parameter weggelassen, so wird das eigene Verzeichnis benutzt.

Verzeichnisse kann man mit `mkdir` (für *make directory*) erstellen und mit dem Befehl `rmdir` (für *remove directory*) löschen. Der Befehl `mkdir temp` erstellt also das Verzeichnis `temp`, der Befehl `rmdir temp` löscht das leere (!) Verzeichnis `temp`. Verzeichnisse, in denen sich Dateien befinden, können nur mit `rm -R` gelöscht werden.

Um den Inhalt eines Verzeichnisses anzugeben, gibt es den List-Befehl `ls`. Mit `ls dir` wird der Inhalt des Verzeichnisses `dir` ausgegeben. Durch die Option `-a` werden auch Dateien angezeigt, deren Dateinamen mit einem Punkt beginnen, zum Beispiel die Datei `.cshrc`, die beim Starten der Shell abgearbeitet wird. Bei Verwendung von `-l` wird eine lange Liste mit Angaben über Zugriffsrechte, Benutzer, Gruppe, Datum der letzten Modifikation und Größe ausgegeben.

Passwort

Das eigene Passwort kann mit dem Kommando `passwd` geändert werden. Hierzu muß man das alte Passwort, das neue und (zur Sicherheit, daß man sich nicht vertippt hat) noch einmal das neue eintippen. Die Passwörter erscheinen nicht auf dem Bildschirm.

Arbeitet man auf mehreren miteinander vernetzten Rechnern, wird oft *NIS (Network Information Service)* – früher *Yellow Pages* – verwendet. Dabei existiert nur eine Passwortdatei für alle Rechner. Es gibt dann einen *Server*, der die Passwörter verwaltet. In so einem Fall muß man das Kommando `yppasswd` verwenden.

Umleiten der Aus- und Eingabe

Die Shell, also der Kommandointerpreter, erlaubt auch das Umlenken von Ein- und Ausgaben durch Einfügen des Zeichens $<$ bzw. $>$. Durch das $|$-Symbol (*pipe*) wird die Ausgabe eines Programmes zur Eingabe eines weiteren Programmes. Zum Beispiel gibt `ls dir | more` den Inhalt des aktuellen Verzeichnisses seitenweise an. `ls dir > temp` erstellt die Datei `temp` und beschreibt sie mit der Ausgabe des `ls`-Befehls.

Der Standard Editor `vi`

Für das Erstellen und das Ändern von Dateien stehen einige sogenannte Editoren zur Verfügung. `vi` (*visual*) ist der am weitesten verbreitete. Er besitzt sehr viele Möglichkeiten, ist allerdings anfangs nicht leicht zu bedienen. Der Editor `emacs` von GNU wird ebenfalls häufig benutzt. Hier soll nur kurz auf den `vi` eingegangen werden, da er in jedem UNIX System mitgeliefert wird.

Aufgerufen wird der Editor mit `vi file`. Jetzt wird die Datei `file` (falls vorhanden) geladen oder sonst neu erstellt. Beim Arbeiten mit dem `vi` werden drei Modi unterschieden:

- Der Kommando-Modus
- Der Einfüge-Modus
- Der Kommandozeilen-Modus

Nach dem Starten befindet man sich im Kommando-Modus. Das heißt, die getippten Tasten werden als Befehle interpretiert. Wichtige `vi`-Befehle in diesem Modus sind:

A   Springt an das Ende der Zeile und geht in den Einfüge-Modus (*Append*)

I   Springt an den Anfang der Zeile und geht in den Einfüge-Modus (*Insert*)

i	Geht in den Einfüge-Modus (*Insert*)
ESC	schaltet vom Einfüge-Modus zurück in den Kommando-Modus
J	fügt die nächste Zeile zur aktuellen hinzu (*Join*)
x	löscht das Zeichen unter dem Cursor
dd	löscht die aktuelle Zeile
D	löscht ab der Cursorposition bis zum Ende der Zeile
dw	löscht ein Wort
nG	springt in Zeile n
G	springt in die letzte Zeile
/*pattern*	sucht nach *pattern*.

Mit einem Doppelpunkt **:** wird in den Kommandozeilen-Modus umgeschaltet. Die wichtigsten Befehle sind:

:x	Abspeichern des Dokumentes und verlassen des Editors (*Exit*)
:q	Verlassen des Editors, falls nichts geändert wurde (*Quit*)
:q!	Verlassen des Editors, Änderungen werden verworfen (*Quit*)
:w	Abspeichern des Dokumentes (*Write*)
:wq	Abspeichern des Dokumentes und verlassen des Editors
:r *file*	Einlesen der Datei *file* (*Read*)
:set nu	Zeilennumerierung einschalten (*Set Numbers*)

Es gibt noch wesentlich mehr Kommandos. So kann man natürlich Blöcke markieren und kopieren oder auch ganze Blöcke einrücken.

Der C-Compiler

Das Übersetzen eines C-Programmes in einen ausführbaren Code geschieht in zwei Schritten. In dem ersten wird das Programm in ein sogenanntes Objektfile übersetzt (*compiling*). Dieses enthält dann das übersetzte Programm. Im zweiten Schritt wird dieses Objekt mit Standardbibliotheken und eventuell anderen Objektdateien zu einem ausführbaren Programm zusammengefügt (*linking*).

Der C-Compiler, der bei einem UNIX System mitgeliefert wird, heißt `cc`. Mit dem Aufruf `cc file.c` wird das Programm `file.c` übersetzt und auch schon gleich gebunden. Das fertige, ausführbare Programm heißt dann `a.out`. Sollen Mathematikroutinen, z. B. trigonometrische Funktionen, verwendet werden, muß die Mathematikbibliothek dazugebunden werden. Dieses geschieht mit der Option `-lm`.

Wichtige Optionen sind:

`-o name` Das Ausgabefile heißt `name` statt `a.out`

`-O` optimiert den Code

`-c` erzeugt nur eine Objektdatei

`-I dir` Include-Dateien werden zusätzlich in `dir` gesucht

`-l`*x* Der Linker bindet die Bibliothek `lib`*x* dazu

`-L dir` Der Linker sucht Programmbibliotheken zusätzlich in `dir`

Ein Standard C-Compiler, der frei verfügbar ist, ist der `gcc`(*) von GNU. Er liefert oftmals schnelleren Code als der vom System mitgelieferte. Weiterhin unterstützt er C++ mit dem Aufruf `g++`.

Make

Um größere Programmpakete zu übersetzen, benutzt man häufig das Hilfsprogramm `make`. Es liest eine Datei namens `Makefile` und arbeitet die darin angegebenen Anweisungen ab. Das Format soll hier nicht besprochen werden.

Netzwerke

Viele UNIX-Rechner sind über das *Internet* vernetzt. In diesem Abschnitt werden die wichtigsten Befehle beschrieben.

Jeder am Internet angeschlossene Rechner hat eine Adresse. Diese besteht aus vier Zahlen, die jeweils zwischen 0 und 255 liegen dürfen. Jeder Adresse ist ein Name zugeordnet. So hat `ftp.physik.uni-wuerzburg.de` zur Zeit die Adresse

132.187.40.15. Die einzelnen Zahlengruppen lassen sich nicht eindeutig in Namen umsetzen. So beginnt zwar jede Würzburger Uni-Adresse (.uni-wuerzburg.de) mit 132.187, die Physik hat aber noch andere Unternummern außer 40. Da bei Netzwerkumstellungen die Nummern geändert werden können, sollte man möglichst die Namen verwenden.

Verbindung zu anderen Rechnern

Man kann durch den Befehl `telnet rechner` eine Verbindung zu einem anderen Rechner im Netz aufbauen. Dabei kann `rechner` entweder der Name oder die Internet Adresse sein.

`telnet ftp.physik.uni-wuerzburg.de` gibt, falls eine physikalische Verbindung besteht und unser Rechner arbeitet, als Ausgabe:

```
Trying...
Connected to wptx15.physik.uni-wuerzburg.de.
Escape character is '^]'.

HP-UX wptx15 A.09.03 A 9000/712 (ttys0)

login:
```

An dieser Stelle muß der Benutzer seine Benutzerkennung und sein Passwort eingeben.

Man sollte Telnet-Sitzungen immer korrekt schließen, d.h. den fremden Rechner mit `logout` verlassen. Ist mal alles schief gegangen, kommt man mit dem angegebenen Steuerzeichen *Escape character* in ein Telnet-Menü. Dabei bedeutet das `^]` Zeichen das gleichzeitige Drücken der Control- und der] -Taste. In dem Telnet-Menü erhält man mit dem Kommando `?` eine kurze Hilfe. Mit `close` kann die Verbindung beendet werden.

Eine Möglichkeit, auf einem anderen UNIX-Rechner zu arbeiten, bietet das Kommando `rlogin`. Hierbei wird der gleiche Benutzername angenommen. Mit der Option `-l` lässt sich dieser explizit ändern.

Programme von anderen Rechnern kopieren

Um Dateien zwischen Rechnern auszutauschen, gibt es `ftp`, ein *File Transfer Protocol*. Mit `ftp Rechner` wird eine Verbindung aufgebaut. Dabei kann `Rechner` wieder entweder die Internetadresse oder der Name sein. Auch hier muß sich der Benutzer identifizieren. Der Rechner fragt nach Benutzernamen und Passwort. Hat das `login` geklappt, so können unter anderem folgende Befehle benutzt werden:

`bi`	setzt den Übertragungsmodus auf *binär*, d.h. die Daten werden ohne Änderung übertragen
`as`	setzt den Übertragungsmodus auf *ASCII* (Textmodus). Hierbei können Steuerzeichen unterdrückt werden. Es kann eine systemabhängige Konvertierung der Textzeichen vorgenommen werden.
`ls`	zeigt die Dateien in dem fremden Verzeichnis an (*list*)
`cd dir`	setzt das fremde Verzeichnis auf `dir`
`get file`	kopiert die Datei `file` vom fremden auf den lokalen Rechner
`put file`	kopiert die Datei `file` vom lokalen auf den fremden Rechner
`mget files`	wie `get`, nur dürfen Joker im Dateinamen verwendet werden
`mput files`	wie `put`, nur dürfen Joker im Dateinamen verwendet werden

Internet-Dateitransfer

Viele Einrichtungen bieten Programme an, die sich jeder frei kopieren darf. Diese kann man sich über das Internet mit Hilfe von `ftp` beschaffen. Spezielle *ftp-server* bieten dazu die Möglichkeit, sich als Benutzer `ftp` oder `anonymous` einzuloggen. Als Passwort verlangen sie die eigene E-Mail Adresse. So sind auch die in diesem Buch angegebenen Programme auf dem ftp-Server

`ftp.physik.uni-wuerzburg.de`

(derzeitige IP-Adresse: 132.187.40.15) im Verzeichnis `/pub/cphys` erhältlich.

Datenkomprimierung

Zum Archivieren von Daten ist es nützlich, mehrere Programme unter einem Namen zusammen zu speichern. Dieses wird oft mit dem Programm `tar` (*tape archiver*) getan. Mit `tar -cf file.tar dir` archiviert man das Verzeichnis `dir` mit all seinen Dateien und Unterverzeichnissen in die eine Datei `file.tar`. Mit dem Befehl `tar -tf file.tar` erhält man ein Inhaltsverzeichnis des Archivs und mit dem Befehl `tar -xf file.tar` wird das Verzeichnis wieder hergestellt.

Um dann noch Plattenplatz zu sparen, werden Komprimierungsprogramme eingesetzt. Ein häufig verwendetes ist gzip[(*)]. `gzip file` komprimiert die Datei `file`. Das Ausgabefile hat den Namen `file.gz`. Zum Expandieren dienen die Befehle `gzip -d file.gz` oder `gunzip file.gz`.

X11 über ein Netzwerk

Es ist kein Problem, X11-Anwendungen über ein Netzwerk zu betreiben, also auf einem Rechner zu rechnen und auf einem anderen die Ausgaben anzeigen zu lassen. Nehmen wir an, wir wollen auf der Maschine `rechner` ein X-Programm starten, welches die Fenster auf dem X-Bildschirm unserer lokalen Maschine `local` darstellen soll. Dann müssen wir als erstes unserer Maschine mitteilen, daß sie Bilder von der anderen Maschine akzeptieren soll. Dies geschieht mit: `xhost + rechner`. Auf der Maschine `rechner` müssen wir die Umgebungsvariable `DISPLAY` setzen. Dieses geschieht in der `csh`, bzw. `tcsh` durch `setenv DISPLAY local:0`. Das Symbol `:0` steht für die erste X-Sitzung. Alle jetzt aufgerufenen X-Programme senden ihre Graphikausgaben auf den Bildschirm von `local`.

Literatur

UNIX: Eine Einführung, Regionales Rechenzentrum für Niedersachsen, Universität Hannover, 1993.

J. Gulbins, *Unix*, Springer Verlag, 1988.

Anhang D
Erste Schritte mit Xgraphics

In unseren Beispielprogrammen werden oft Bewegungen berechnet, die unmittelbar nach dem Vorliegen der Daten auf dem Bildschirm sichtbar gemacht werden. Eine solche graphische Darstellung von Resultaten, die im C-Programm berechnet werden, ist auf der Workstation im UNIX-Betriebssystem nicht einfach. Es steht zwar auf fast allen Rechnern die Benutzeroberfläche X-Windows zur Verfügung, mit der elementare Graphikbefehle programmiert werden können, aber die Benutzung ist für Anfänger sehr kompliziert, weil selbst einfache Programme mehrere Seiten Quellcode benötigen.

Um die Erstellung von Graphikprogrammen zu erleichtern, hat Martin Lüders Xgraphics entwickelt. Es verwendet ausschließlich die Routinen von Xlib, der Standard-Bibliothek des X-Windows Systems, und stellt einfache Befehle zum Verwalten von Fenstern und zum Zeichnen in Fenster bereit. Dabei werden die vielen Parameter, die für Aufrufe der Xlib-Befehle übergeben werden müssen, intern verwaltet, so daß beim Programmieren nur die wirklich notwendigen Parameter angegeben werden müssen. Vor allem wird dem Programmierer aber ein Großteil der Verwaltung der Fenster abgenommen. So muß man sich beispielsweise in einfachen Fällen nicht darum kümmern, ob ein Fenster vom Anwender vergrößert oder im anderen Fall zum Icon verkleinert wird. Dadurch wird es möglich, tatsächlich die Physik in den Mittelpunkt der Programme zu stellen. Dennoch sind die Datenstrukturen der Xlib direkt zugänglich, so daß man jederzeit Xlib-Befehle oder auch auf Xlib aufbauende Erweiterungen hinzufügen kann.

Ein wichtiger Grundbaustein von Xgraphics sind die Zeichenbereiche, *Worlds* genannt, die es erlauben, ein lokales Koordinatensystem zu definieren, welches dem zu programmierenden Problem angepaßt ist und sich nicht nach der jeweiligen Größe des aktuellen Fensters zu richten hat. Die wesentlichen Befehle von Xgraphics sind: Erzeuge ein Fenster (`CreateWindow`), erzeuge einen Zeichenbereich mit selbstdefinierten Koordinaten im Fenster (`CreateWorld`), erzeuge mit der Maus bedienbare Steuerungsknöpfe (`InitButtons`), reagiere auf Maus- oder Tastenbewegungen (`GetEvent`), lies eine Zahl ein (`GetNumber`), zeichne einen Punkt mit Pixelkoordinaten (`DrawPoint`) oder mit selbstdefinierten Koordinaten (`WDrawPoint`), zeichne einen Kreis (`DrawCircle` bzw. `WDrawCircle`), schreibe Text (`DrawString` bzw. `WDrawString`) usw. Der Quellcode des C-

Programms `Xgraphics.c` muß zusammen mit dem eigenen Programm übersetzt werden. Dabei müssen die X11-Bibliothek und die entsprechenden Pfade angegeben werden. Dies macht das Programm `compile`. Im eigenen Programm muß die Header-Datei `Xgraphics.h` mit `#include` hinzugefügt werden. Eine ausführliche Beschreibung gibt das Postscript-File `Xgraphics.ps`.

Dieses Programmpaket inklusive der Dokumentation, einiger Demonstrationsprogramme und der X-Windows-Versionen der Beispielprogramme finden sich auf der Diskette. Sie sind mit dem Unix-Befehl `tar` zu Gruppen zusammengefaßt. Die Archive enthalten:

- `xgraphics.tar` Das eigentliche Xgraphics Paket mit der Dokumentation.
- `demo.tar` Einige Demonstrationsprogramme.
- `physics.tar` Die Beispiele aus dem Buch.

Das Entpacken erreicht man mit folgendem Befehl:

```
tar xf file.tar
```

Außerdem kann der Leser sie über das Computernetz auf seinen Rechner kopieren. Das geschieht wie folgt:

```
ftp  ftp.physik.uni-wuerzburg.de
```

Der Rechner fragt nach dem Namen:

```
ftp
```

Der Rechner fragt nach dem Passwort:

```
leser@rechner.uni-excellent.de
```

Das richtige Verzeichnis muß gewählt werden:

```
cd pub/Xgraphics
```

Jetzt werden sämtliche Files kopiert.

```
mget *
```

Dabei fragt der Rechner bei jedem File, ob er es kopieren soll (Antwort `y` oder `n`). Abschließend wird die Verbindung beendet:

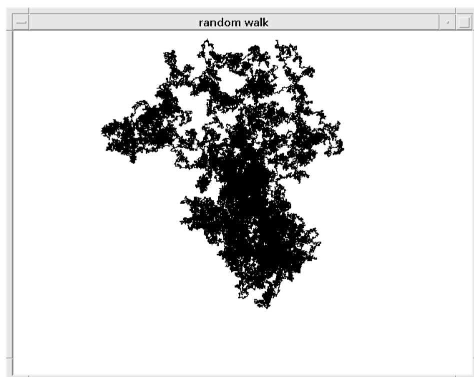
```
bye
```

Außerdem existieren im World Wide Web einige Seiten zu Xgraphics, mit der kompletten Dokumentation und gegebenenfalls Hinweisen zu neuen Versionen. Die Adresse im World Wide Web ist:

```
http://www.physik.uni-wuerzburg.de/TP3/Xgraphics.html
```

An einem einfachen Beispiel wollen wir demonstrieren, wie man schon mit ganz wenigen Befehlen ein physikalisches Problem auf dem Bildschirm darstellen kann, nämlich die Zufallsbewegung eines Teilchens (*random walk*), für die es eine weitentwickelte mathematische Beschreibung gibt. Im Modell hüpft ein Teilchen zufällig auf einen der vier Nachbarplätze im Quadratgitter. Dies wird durch das folgende C-Programm dargestellt:

```
#include <stdlib.h>
#include <math.h>
#include "Xgraphics.h"

#define MAXX 640
#define MAXY 480
#define N 250000

main()
{
   int x=MAXX/2,y=MAXY/2,r,i;

   InitX();
   mywindow=CreateWindow(MAXX,MAXY,"random walk");
   ShowWindow(mywindow);

         for(i=0;i<N;i++)
          {    DrawPoint(mywindow,x,y,1);
               r=rand()/(RAND_MAX+1.)*4.;
               switch(r)
                    {   case 0: x++;break;
                        case 1: x--; break;
                        case 2: y++;break;
                        case 3: y--;break;       }
          }
  getchar();
  ExitX();
}
```



D.1 Zufallsweg mit 250 000 Schritten auf dem Quadratgitter.

x und y sind die Pixel-Koordinaten des Teilchens im Fenster, dessen Größe durch `MAXX` und `MAXY` festgelegt ist. Gemäß der Zufallszahl $r \in \{0, 1, 2, 3\}$ werden sie jeweils um den Wert eins erhöht oder erniedrigt. Mit fünf Graphikbefehlen kann man also schon interessante Bewegungen auf dem Bildschirm erzeugen. Das Programm mit dem Namen `walk.c` wird mit `compile walk` übersetzt und mit `walk` aufgerufen. Dabei müssen eventuell im File `compile` die entsprechenden Pfade der eigenen Maschine eingetragen werden, und das File muß mit `chmod u+x compile` ausführbar gemacht werden. Das Ergebnis sollte das Bild D.1 sein. Wenn man dann im aufrufenden Fenster ein Zeichen eingibt, verschwindet das Bild mit dem Zufallsweg.

Weitere Demonstrationsprogramme, die auch die anderen Fähigkeiten von Xgraphics zeigen, finden sich auf der Diskette.

Literatur

M. Lüders, *Einführung in Xgraphics*, Postscript File auf beiliegender Diskette, außerdem erhältlich über FTP unter `ftp.physik.uni-wuerzburg.de` als `Xgraphics.ps` im Verzeichnis `/pub/Xgraphics/`.

O. Jones, *Einführung in das X-Window System*, Hanser Verlag, 1991.

Anhang E
Programme

In diesem Abschnitt wird ein Teil der Computer-Programme abgedruckt, und zwar fast alle *Mathematica*-Programme und einige der C-Programme für den PC. Aus Platzgründen können wir leider nicht alle Quellcodes drucken, aber alle Programme sind auf der beiliegenden Diskette vorhanden und können selbst ausgedruckt werden. Die Namen der *Mathematica*-Programme sind durchgehend von der Form `name.m`. Die PC-Versionen der C-Programme wurden mit *Turbo C* von Borland kompiliert und sind als `name.exe` auf der beiliegenden Diskette vorhanden.
Einige der C-Programme kann man mit Tasten steuern, die auf dem Bildschirm in der Zeile *Befehle* angezeigt werden. Der erste Buchstabe in dem erklärenden Wort steuert den Algorithmus; so bedeutet *exit*, daß die Taste `e` das Programm beendet.
Alle Beispiele werden auch als Programme für UNIX oder LINUX auf der Diskette mitgeliefert. Zusätzlich können sie, dann vielleicht in verbesserter Version, vom FTP-Server des Institutes für Theoretische Physik der Universität Würzburg über das Internet kopiert werden (siehe Anhang C). Die Verzeichnisse dort sind:
`/pub/buch/cphys/dos/`
`/pub/buch/cphys/mathematica/`
`/pub/buch/cphys/unix/`
`/pub/Xgraphics/`
Zusätzlich gibt es im World Wide Web Informationen zu den Programmen unter der Adresse

`http://www.physik.uni-wuerzburg.de/TP3/cphys.html`

1.1 Funktion gegen Prozedur: summe.m und summe.c

```
Print["           Funktion gegen Prozedur"]
dataset=Table[Random[],{10000}];
average[data_,length_]:= Block[{sum,average},
                          sum=0.;
                          Do[sum=sum+data[[i]],{i,length}];
                          average=sum/length
                               ]
average[data_]:=Apply[Plus,data]/Length[data]
```

summe.c

```
#include <stdlib.h>
#include <time.h>

main()
{
  float average(float *,int);
  int i;
  clock_t  start,end;
  float dataset[10000];
  clrscr();
  for (i=0;i<10000;i++) dataset[i]=random(1000)/1000.;
  start=clock();
  for(i=0;i<99;i++) average(dataset,10000);
  printf( "  average = %f\n",average(dataset,10000));

  end=clock();
  printf( " Zeit= %f sec %d ",(end-start)/CLK_TCK,CLK_TCK);
  getch();
}

float average( float* data,int n )
{
   float sum=0. ;
   int i;
   for(i=0;i<n;i++) sum=sum+data[i] ;
   return sum/n;
}
```

1.2 Pendel: pendel.m

```
Print[ " Nichtlineares Pendel "]
T[phi0_]=4 EllipticK[Sin[phi0/2]^2]
plot1:=Plot[T[phi0],{phi0,0,Pi},PlotRange->{0,30},
            Frame -> True, FrameLabel->{"phi0","T"}]
sinuspsi[t_,phi0_]=JacobiSN[t,Sin[phi0/2]^2]
phiscal[x_,phi0_]=2 ArcSin[Sin[phi0/2]*
                           sinuspsi[x T[phi0],phi0]]/phi0
phi0[1]=N[.1 Pi]
phi0[2]=N[.8 Pi]
phi0[3]=N[.95 Pi]
phi0[4]=N[.99 Pi]
phi0[5]=N[.999 Pi]
fliste=Table[phiscal[x,phi0[i]],{i,5}]
```

```
plot2:=Plot[Evaluate[fliste],{x,0.,1.},
            Frame -> True,
            FrameLabel->{"t/T","phi/phi0"},
            PlotStyle -> Thickness[0.001]]
liste=Table[phiscal[x,N[.999 Pi]],{x,0,.99,.01}]
fouliste:=Take[Abs[Fourier[liste]],15]
plot3:= (gr1 = ListPlot[fouliste,PlotRange->{{0,15},{-1,8}},
                        Frame -> True,
                        PlotStyle -> PointSize[0.02],
                        FrameLabel->{"s","Abs[b(s)]"},
                        DisplayFunction -> Identity] ;
               Show[gr1, Graphics[Line[{{0,0},{0,15}}]],
                    DisplayFunction -> $DisplayFunction ] )
f=1/Sqrt[1-m Sin[psi]^2]
g=Series[f,{m,0,10}]
tseries=4 Integrate[g,{psi,0,Pi/2}] /. m->Sin[phi0/2]^2
e=phidot^2/2-Cos[phi]
plot4:=ContourPlot[e,{phi,-Pi,Pi},{phidot,-3,3},
                   Contours->{-.5,0,.5,1,1.5,2},
                   ContourShading->False,PlotPoints->100,
                   FrameLabel -> {"phi", "phidot"}]
```

1.3 Fouriertransformation: fourier.m

```
Print["        Fouriertransformation"]
f[t_]=(Sign[1-t]+Sign[1+t])/2
fs[w_] = Integrate[Exp[I*w*t], {t, -1, 1}]/T // ComplexExpand
T=10
plot1:=Plot[fs[w],{w,-5Pi,5Pi}, PlotRange -> All]
p2liste:= p2liste = Table[ Abs[ fs[ k*2 Pi/T] ],{k,64}]
plot2:=(g1=ListPlot[p2liste,PlotJoined->True,PlotRange->All,
                    PlotStyle -> Thickness[0.0],
                    DisplayFunction -> Identity];
        g2=ListPlot[p2liste,PlotRange->All,
                    PlotStyle -> PointSize[0.015],
                    DisplayFunction -> Identity];
           Show[g1,g2,DisplayFunction -> $DisplayFunction] )
fliste := fliste = Table[N[f[-5+10 r/64]],{r,64}]
fsliste := fsliste = Fourier[fliste]/Sqrt[64]
p3liste:= p3liste = Abs[fsliste ]
plot3:=(g1=ListPlot[p3liste, PlotJoined->True,PlotRange->All,
               PlotLabel->"Diskrete Fouriertransformation",
                    PlotStyle -> Thickness[0.0],
                    DisplayFunction -> Identity];
        g2=ListPlot[p3liste,PlotRange->All,
```

```
                    PlotStyle -> PointSize[0.015],
                    DisplayFunction -> Identity];
         Show[g1,g2, DisplayFunction -> $DisplayFunction] )
fapp1[t_]:=Sum[fsliste[[s]] Exp[-2 Pi I /64 (s-1)(64t/T+31)],
               {s,64}]//N
plot4:=Plot[{Re[fapp1[t]],f[t]},{t,-5,5}]
fapp2[t_]:=Sum[N[fsliste[[s]]*
                 Exp[-2 Pi I /64 (s-1)(64t/T+31)]],{s,32}]+
           Sum[N[Conjugate[fsliste[[34-s]]]*
                 Exp[-2 Pi I /64 (s-33)(64t/T+31)]],{s,32}]
plot5:=Plot[{Re[fapp2[t]],f[t]},{t,-5,5}]
```

1.4 Datenglätten: glaetten.m

```
Print["      Daten glaetten  mit Faltung"]
data = Table[N[BesselJ[1,x]+.2 (Random[]-1/2)],
                                  {x,0,10,10./255}]
xdata = Range[0,10,10./255]
plot1 := p1 = ListPlot[Thread[Join[{xdata},{data}]],
                       PlotStyle->PointSize[.01]]
sigma= 0.4
kern = Table[ N[ Exp[-x^2/(2*sigma^2)]],{x,-5,5,10./255} ]
kern = RotateLeft[kern,127]
kern = kern/Apply[Plus,kern]
plot2:=p2=ListPlot[Thread[Join[{xdata},{kern}]],
                   PlotRange->All]
glatt=Sqrt[256]*
      InverseFourier[Fourier[data]*Fourier[kern]] //Chop
plot3:=p3=ListPlot[Thread[Join[{xdata},{glatt}]]]
plot4:=p4=Plot[BesselJ[1,x],{x,0,10},
               PlotStyle ->Thickness[0.001]]
plot5:=Show[p1,p3,p4]
```

1.5 Nichtlinearer Fit: chi2.m

```
Print[        " Nichtlinearer Fit"]
Needs["Statistics‘Master‘"]
SeedRandom[12345]
daten = Table[{t,Sin[t] Exp[-t/10.]+.4 Random[]-.2}//N,
              {t,0,3Pi,.3 Pi}]
plot1:= (gr1 = ListPlot[daten,PlotStyle -> PointSize[0.02],
                        DisplayFunction -> Identity];
         gr2 = Plot[Sin[t] Exp[-t/10.],{t,0,3Pi},
                    DisplayFunction -> Identity];
         Show[gr1,gr2, DisplayFunction -> $DisplayFunction])
```

```
f[t_]=a Sin[om t + phi] Exp[-t b]
sigma2=NIntegrate[x^2,{x,-.2,.2}]/.4
sigma = Sqrt[sigma2]
quadrat[{t_,y_}]=(y-f[t])^2/sigma2
chi2=Apply[Plus,Map[quadrat,daten]]
find:=FindMinimum[chi2,{a,0.9},{om,1.1},{phi,0.1},{b,.2}]
fit:=NonlinearFit[daten,f[t],t,
                  {{a,1.1},{om,1.1},{phi,.1},{b,.2}},
                  ShowProgress->True]
pvalue[x_]=1.-CDF[ChiSquareDistribution[7],x]
intervall={Quantile[ChiSquareDistribution[7],.05],
           Quantile[ChiSquareDistribution[7],.95]}
grenze[x_]=Quantile[ChiSquareDistribution[4],x]

plot2:=Plot[PDF[ChiSquareDistribution[7],x],{x,0,20}]
rule = NonlinearFit[daten,f[t],t,
                    {{a,1.1},{om,1.1},{phi,.1},{b,.2}}]
a0=a/.rule; b0=b/.rule; om0=om/.rule; phi0=phi/.rule;
chi2min = chi2/.rule

plot3:=ContourPlot[chi2/.{phi->phi0,om->om0},
         {a,.4,1.3},{b,-.04,.2},
         ContourShading->False,
         Contours->{chi2min+grenze[.683],chi2min+grenze[.9]},
         PlotPoints->50,FrameLabel->{"a","b"}]

plot4:=ContourPlot[chi2/.{phi->phi0,a->a0},
         {om,.90,1.05},{b,0.0,.15},
         ContourShading->False,
         Contours->{chi2min+grenze[.683],chi2min+grenze[.9]},
         PlotPoints->50,FrameLabel->{"om","b"}]
g[t_]= f[t]/.rule
step:=(daten2=N[Table[{t,g[t] + .4 Random[]-.2},
                 {t,0,3Pi,.3 Pi}]];
       NonlinearFit[daten2,f[t],t,
                    {{a,a0},{om,om0},{phi,phi0},{b,b0}}] )
tab:= tab = Table[{chi2,a,b,om,phi}/.step,{100}]
abtab := Map[Take[#,{2,3}]&,tab]

plot5:= ListPlot[abtab,PlotRange->{{.4,1.2},{-0.045,.2}},
                 AspectRatio -> 1, Frame -> True,
                 FrameLabel ->{"a","b"}, Axes -> None,
                 PlotStyle -> PointSize[0.01]]
```

1.6 Multipole: multipole.m

```
Print[" Multipole fuer ein statisches Potential "]
SeedRandom[123456789]
rpunkt:={2Random[]-1,2Random[]-1,0}
Do[r[i]=rpunkt,{i,10}]
p1=Graphics[Table[Line[{Drop[r[i],-1]-{0.08,0},
                        Drop[r[i],-1]+{0.08,0}}],{i,5}]]
p2=Graphics[Table[Line[{Drop[r[i],-1]+{0,0.08},
                        Drop[r[i],-1]-{0,0.08}}],{i,5}]]
p3=Graphics[Table[Line[{Drop[r[i+5],-1]-{0.08,0},
                        Drop[r[i+5],-1]+{0.08,0}}],{i,5}]]
p4=Graphics[{Thickness[0.001],
             Table[Circle[Drop[r[i], -1], 0.1],{i,10}]}]
plot1:=Show[p1, p2, p3, p4, Frame ->True,
            AspectRatio -> Automatic,
            PlotRange -> {{-2, 2}, {-2, 2}},
            FrameLabel -> {"x", "y"}]
dist[r_,s_]=Sqrt[(r-s).(r-s)]
pot[rh_]:=Sum[ 1/dist[rh,r[i]]-1/dist[rh,r[i+5]] ,{i,5}]
plot2:=Plot3D[pot[{x,y,0}] ,{x,-2,2},{y,-2,2},
           AxesLabel->{"x","y","Phi"},
           PlotRange -> {-8,8},
           PlotPoints->40]
plot3:=ContourPlot[pot[{x,y,0}] ,{x,-2,2},{y,-2,2},
              PlotPoints->40,
              ContourShading->False,AxesLabel->{"x","y"} ]
quadrupol[r_]:=Table[3 r[[k]] r[[l]] - If[k==l,r.r,0],
                     {k,3},{l,3}]
qsum=Sum[quadrupol[r[i]]-quadrupol[r[i+5]],{i,5}]
betrag[r_]=Sqrt[r.r]
dipol=Sum[r[i]-r[i+5],{i,5}]
pot1[r_] = dipol.r/betrag[r]^3
pot2[r_] = pot1[r] + 1/2/betrag[r]^5 * r.qsum.r
weg={.6,y,0}
plot4:=Plot[{ pot1[weg],pot2[weg],pot[weg]},{y,-2,2},
            Frame -> True,
            FrameLabel -> {"{.6,y,0}","Potential"},
            PlotStyle -> {Dashing[{.01,.01}],
                      Dashing[{.03,.03}],Dashing[{1.0,0}]}]
Needs["Graphics`PlotField`"]
efeld=-{D[pot[{x,y,0}],x],D[pot[{x,y,0}],y]}
richtung=efeld/betrag[efeld]
plot5:=PlotVectorField[richtung,{x,-10,10},{y,-10,10},
                       PlotPoints->30]
```

1.7 Wegintegrale: wegintegrale.m

```
Print["     Wegintegrale"]
r1={Cos[2Pi t],Sin[2Pi t],t}
r2={1,0,t}
r3={1-Sin[Pi t]/2,0,1/2(1-Cos[Pi t])}
p1:=ParametricPlot3D[Evaluate[r1],{t,0,1},
                     DisplayFunction->Identity]
p2:=ParametricPlot3D[Evaluate[r2],{t,0,1},
                     DisplayFunction->Identity]
p3:=ParametricPlot3D[Evaluate[r3],{t,0,1},
                     DisplayFunction->Identity]
plot1:=Show[p1,p2,p3,PlotLabel->" Wegintegrale",
                     AxesLabel->{"x","y","z"},
                     DisplayFunction->$DisplayFunction]
k[{x_,y_,z_}]={2 x y +z^3,x^2,3 x z^2}
v[r_]:=D[r,t]
int[r_]:=Integrate[k[r].v[r],{t,0,1}]
r4=t{x,y,z}
rot[{kx_,ky_,kz_}]:={ D[kz,y]-D[ky,z],
                      D[kx,z]-D[kz,x],
                      D[ky,x]-D[kx,y]  }
```

1.8 Maxwell-Konstruktion: maxwell.m

```
Print [ " Maxwell-Konstruktion fuer das van der Waals-Gas "]
pp=t/(v-b)-a/v^2
gl1=D[pp,v]==0
gl2=D[pp,{v,2}]==0
sol=Solve[{gl1,gl2},{t,v}]
pc=pp/.sol[[1]]
p[v_] = 8 t/(3 v-1)-3/v^2
Off[Integrate::gener]
gl3 = p[v1]==p[v3]
gl4 = p[v1]*(v3-v1)==Integrate[p[v],{v,v1,v3}]
On[Integrate::gener]
pmax[v_]:=If[v < vleft || v > vright, p[v], p[vleft]]
plot[T_]:= Block[{}, t = T;
                 If[ t >= 1,
                 Plot[p[v], {v, .34, 5.},
                      PlotRange -> {{0,5},{0,2}},
                      Frame -> True,
                      FrameLabel -> {"v","p"}],
            vmin = v /. FindMinimum[ p[v], {v,0.4}][[2]];
            vmax = v /. FindMinimum[-p[v], {v,1.}][[2]];
```

```
            vtest = (vmin + vmax)/2 ;
            r = Solve[ p[v] == p[vtest], v];
            v1start = v /. r[[1]];
            v3start = v /. r[[3]];
            frs = Chop[FindRoot[{gl3,gl4},{v1,v1start},
                                {v3,v3start}]];
            vleft=v1/.frs; vright=v3/.frs;
            Plot[{pmax[v],p[v]},{v,0.34,5.},
                 PlotRange->{{0,5},{0,2}},
                 Frame -> True, FrameLabel -> {"v","p"} ]]]
```

1.9 Beste Spielstrategie: spiel.c

Nur auf Diskette vorhanden.

2.1 Quantenoszillator: quantenoszi.m

```
Print["    Quantenoszillator"]
q[j_,k_]:= Sqrt[(j+k+1)]/2 /; Abs[j-k]==1
q[j_,k_]:= 0 /; Abs[j-k]!= 1
q[n_]:= Table[q[j,k], {j,0,n-1}, {k,0,n-1}]
h0[n_]:= DiagonalMatrix[Table[i+1/2,{i,0,n-1}]]
h[n_]:= h0[n] + lambda q[n].q[n].q[n].q[n]
ev[n_]:= Eigenvalues[h[n]]
plot1:= Plot[Evaluate[ev[4]], {lambda, 0, 1},
             Frame -> True,
             FrameTicks -> {Automatic,
       Table[{0.5*j,If[EvenQ[j],ToString[j/2],""]},{j,19}]},
             FrameLabel -> {"lambda","Energie"},
             PlotLabel ->"Eigenwerte von h[4]" ]
evnum[n_,la_]:= Sort[Eigenvalues[N[h[n]/. lambda->la]]]
evdimlist[n_,la_]:= Table[{1/i, evnum[i,la][[1]]}, {i,7,n}]
plot2:=( gr1=ListPlot[evdimlist[20,.1],
                      Axes -> None, Frame -> True,
                      PlotRange -> All,
                      FrameLabel -> {"1/n","E0(n)"},
                      PlotStyle -> PointSize[0.02],
                      DisplayFunction -> Identity] ;
         gr2 = Graphics[{Thickness[0.001],
                        Line[{{1/20,0.559146327},
                              {1/7 ,0.559146327}}]}];
         Show[gr1,gr2,DisplayFunction -> $DisplayFunction] )
evlist:= evlist= Table[{la,evnum[20,la]},{la,0,1,.05}]
ev2[k_]:=Table[{evlist[[i,1]],evlist[[i,2,k]]},
               {i,Length[evlist]}]
plot3:= ( Table[gr[k]=ListPlot[ev2[k],PlotJoined ->True,
```

```
                        DisplayFunction -> Identity,
                        Frame -> True,
                        FrameLabel -> {"lambda","Energie"},
                        FrameTicks -> {Automatic,
               Table[{0.5*j,If[EvenQ[j],ToString[j/2],""]},
                        {j,21}]}],{k,5}];
           Show[gr[1],gr[2],gr[3],gr[4],gr[5],
                DisplayFunction -> $DisplayFunction] )
```

2.2 Elektrische Schwingkreise: netz.m

```
Print["    Elektrische Schaltkreise"]
gl1={vr+va==1,ir==ic+il,vr==ir r,
     va==ic/(I omega c),va==I omega l il}
sol=Solve[gl1,{va,vr,ir,ic,il}][[1]]
vas=(va/.sol)//Simplify
numwert = {c -> N[10^(-6)], l -> N[10^(-3)]}
vasnum = vas /. numwert
fliste=Table[vasnum /.r->100.0*3^s, {s,0,3}]
plot1:=Plot[Evaluate[Abs[fliste]],{omega,20000,43246},
            PlotRange->{0,1}, Frame -> True,
            FrameLabel -> {"omega","Abs[va]"},
            FrameTicks -> {{20000,30000,40000},Automatic} ]
plot2:=Plot[Evaluate[Arg[fliste]],{omega,20000,43246},
            Frame -> True,
            FrameLabel -> {"omega","Phase(va)"},
            FrameTicks -> {{20000,30000,40000},Automatic} ]
vsaege[t_] := -1+2 t/T
omegares=1/Sqrt[l c] /. numwert //N
aliste=Table[N[vsaege[(n-1)/256 T]],{n,256}]
bliste=InverseFourier[aliste]
plot3[fac_,w_]:=Block[{valiste,vtrans,omegai,plotlist},
                 omegai = fac*omegares;
                 valiste=Join[
                  Table[vasnum/.{omega->omegai*(s-1),r->w},
                        {s,128}],{0.0},
                  Table[vasnum/.{omega->omegai*(s-128),r->w},
                        {s,1,127}] ];
             vtrans=Fourier[bliste valiste]//Chop;
             plotlist=Table[{k/256,vtrans[[Mod[k-1,256]+1]]},
                            {k,768}];
             ListPlot[plotlist,
                      Frame -> True,
                      FrameLabel->{"t/T","va(t)"},
                      PlotRange -> All]]
```

```
gl2={ir(r + 1/(I omega c) + I omega l) + va ==1,
     ir == (I omega c + 1/(I omega l)) va }
sol2=Solve[gl2,{va,ir}][[1]]
vas2=(va/.sol2)//Simplify
irs = Simplify[ir /. sol2]
leistung = r^2 Abs[irs]^2
fliste3=Table[Abs[vas2] /. numwert /. r->10*3^s ,{s,0,2}]
plot4:=Plot[Evaluate[fliste3],{omega,10000,70000},
        PlotRange->All,Frame -> True,
        PlotStyle->{Thickness[0.006],Dashing[{.01,.008}],
                    Thickness[0.003]},
        FrameTicks -> {{10000,30000,50000,70000},Automatic},
        FrameLabel ->{"omega","|va|"}]
plot5:=Plot[(leistung/.numwert)/.r->10.0,{omega,10000,70000},
            PlotRange->All,Frame -> True,
            FrameTicks -> {{10000,30000,50000,70000},
                            Automatic},
            FrameLabel ->{"omega","P/P0"}]
```

2.3 Kettenschwingungen: kette.m

```
Print["  Kettenschwingungen "]
mat1 = { { 2f          , -f ,  0  , -f*Exp[-I q]  },
         { -f          , 2f , -f  ,  0            },
         {  0          , -f , 2f  , -f            },
         { -f*Exp[I q] , 0  , -f  ,  2f           } }
massmat=DiagonalMatrix[{m1,m1,m1,m2}]
mat2=Inverse[massmat].mat1
eigenlist=Table[{x,
            Chop[Eigenvalues[
             mat2/.{f -> 1., m1 -> 0.4, m2 -> 1.0, q-> x}]]},
                {x,-Pi,Pi,Pi/50}]
plotlist=N[Flatten[Table[
                    Map[{#[[1]],Sqrt[#[[2,k]]]}&,
                        eigenlist],{k,4}],1]]
plot1 := ListPlot[plotlist,FrameLabel -> {"q","omega"},
             Frame -> True,Axes -> None,
             FrameTicks ->{{{-Pi,"-Pi"},{-Pi/2,"-Pi/2"},
                            {0,"0"},{Pi/2,"Pi/2"},{Pi,"Pi"}},
                           Automatic}]
eigensys := Thread[Chop[Eigensystem[mat2 /.
                   {f -> 1., m1 -> 0.4, m2 -> 1.0, q->0.0}]]]
```

2.4 Hofstadter-Schmetterling: hofstadt.c

```
/***** Hofstadter-Schmetterling *****/

#include <math.h>
#include <graphics.h>

#define q 200
#define maxy 200
#define SHIFT 150

main()
{
  int ie, n, nalt, m, p;
  double sigma, pi, e, polyalt, poly, polyneu;
  int gdriver=DETECT,gmode;

  initgraph(&gdriver,&gmode,"\\tc");
  pi=acos(-1.);
  outtextxy(250,400,"Befehl: exit");
  for(p = 1; p < q; p++)
    {
     sigma = 2.0*pi*p/q;
     nalt = 0;
     for(ie = 0; ie < maxy+2 ; ie++)
       {
        e = 8.0*ie/maxy - 4.0 - 4.0/maxy ;
        n = 0;
        polyalt = 1.0; poly = 2.0*cos(sigma) - e;
        if( polyalt*poly < 0.0 ) n++ ;

        for( m = 2; m < q/2; m++ )
          {
           polyneu = ( 2.0*cos(sigma*m) - e )*poly - polyalt;
           if( poly*polyneu < 0.0) n++;
           polyalt = poly; poly = polyneu;
          }
   /*   Die geraden Eigenfunktionen sind auskommentiert.
        polyalt = 1.0; poly = 2.0 - e;
        if( polyalt*poly < 0.0 ) n++ ;
        polyneu = ( 2.0*cos(sigma) - e)*poly - 2.0*polyalt;
        if( poly*polyneu < 0.0) n++;
        polyalt = poly; poly = polyneu;
        for( m = 2; m < q/2; m++ )
          { polyneu = ( 2.0*cos(sigma*m) - e)*poly - polyalt;
```

```
            if( poly*polyneu < 0.0) n++;
            polyalt = poly; poly = polyneu;}
        polyneu = ( 2.0*cos(sigma*q/2)-e)*poly-2.0*polyalt;
        if( poly*polyneu < 0.0) n++;
    */
        if(n > nalt) putpixel(p+SHIFT,ie+SHIFT,WHITE);
        nalt = n;

       }; /* ie-Schleife */
       if(kbhit()) {getch();break;}
    }; /* p-Schleife */
    getch(); closegraph();
}
```

2.5 Hubbard Modell: hubbard.m

```
Print["    Hubbard-Modell"]; Print[""]
sites = Input["   Anzahl der Plaetze: "] ; Print[""]
particles = Input["   Anzahl der Teilchen: "]; Print[""]
spinup = Input["   Davon SpinUp-Teilchen: "] ; Print[""]
spindown = particles - spinup
tkin = -2.*t*(Sum[N[Cos[2*Pi/sites k]],
               {k,-Floor[(spinup-1)/2],Floor[spinup/2]}]+
              Sum[N[Cos[2*Pi/sites k]],
               {k,-Floor[(spindown-1)/2],Floor[spindown/2]}])
Print[""]
Print["   Die Grundzustandsenergie fuer U = 0 ist: " ,tkin]
left = Permutations[ Table[ If[ j <= spinup, 1, 0],
                           {j,sites}] ]
right = Permutations[ Table[ If[ j <= spindown, 1, 0],
                             {j,sites}] ]
index = Flatten[ Table[ {left[[i]],right[[j]]},
                 {i,Length[left]}, {j,Length[right]} ],1]
end = Length[index]
plus[k_,sigma_][arg_]:= ReplacePart[arg,1,{sigma,k}]
minus[k_,sigma_][arg_]:= ReplacePart[arg,0,{sigma,k}]
sign[k_,sigma_,arg_] := (-1)^(spinup*(sigma-1))*
                        (-1)^(Sum[ arg[[sigma,j]],{j,k-1}])
cdagger[sites+1,sigma_][any_] := cdagger[1,sigma][any]
c[sites+1,sigma_][any_] := c[1,sigma][any]
cdagger[k_,sigma_][0]:= 0
c[k_,sigma_][0]:= 0
cdagger[k_,sigma_][factor_. z[arg_]]:=
                   factor*(1 - arg[[sigma,k]])*
                   sign[k,sigma,arg]*z[plus[k,sigma][arg]]
```

```
c[k_,sigma_][factor_. z[arg_]]:=
             factor*arg[[sigma,k]]*
             sign[k,sigma,arg]*z[minus[k,sigma][arg]]
n[k_,sigma_][0]:= 0
n[k_,sigma_][factor_. z[arg_]]:= factor*arg[[sigma,k]]*z[arg]
scalarproduct[a_,0] := 0
scalarproduct[a_,b_ + c_]:=scalarproduct[a,b]+
                           scalarproduct[a,c]
scalarproduct[z[arg1_],factor_. z[arg2_]]:=
                        factor* If[arg1==arg2,1,0]
H[vector_] = Expand[
             -t*Sum[cdagger[k,sigma][c[k+1,sigma][vector]] +
                    cdagger[k+1,sigma][c[k,sigma][vector]] ,
                    {k,sites},{sigma,2} ] +
                u*Sum[n[k,1][n[k,2][vector]] ,{k,sites}] ]
Print[""];Print[" "]
Print["         Die  Hamilton - Matrix wird berechnet. " ]
Print[""]
Print["         Die Dimension dieser Matrix ist ",end,
                                             " x ",end,"."]
h = ( hlist = Table[H[z[index[[j]]]], {j, end}];
   Table[ scalarproduct[ z[index[[i]]], hlist[[j]] ],
          {i,end}, {j,end}] )
Print[""]
  Print["         Die Energie - Eigenwerte",
                              " werden berechnet."]
zustand = Chop[Table[Flatten[
               {x,Sort[Eigenvalues[h/.{t -> 1.,u -> x}]]}],
               {x, 0., 6., .1}]]
Print[""];Print[" "]
Print["         Die folgenden Befehle sind moeglich: "]
Print[" "]
Print["         plot1  => Vielteilchen-Spektrum in",
                             " Abhaengigkeit von U/t "]
Print[" "]
Print["         plot2  => Doppelbesetzung im",
                   " Grundzustand als Funktion von U/t "]
Print[" "]
Print["         plot3  => Vergleich: exaktes Resultat",
                               " fuer N -> Unendlich"]
Print["                   mit endlichem N, Energie pro",
                              " Platz im Grundzustand"]
plot1 :=  Show[
  Table[ListPlot[Map[{#[[1]],#[[j]]}&,zustand],
                 Frame->True,Axes->None,
```

```
                 DisplayFunction -> Identity,
                 PlotStyle->{PointSize[0], Thickness[0.0]},
                 PlotJoined->True,
                 FrameLabel->{"U/t","Energie/t"} ],
        {j,2,end+1}], DisplayFunction -> $DisplayFunction]
g[uu_]:= Chop[Sort[Thread[
            Eigensystem[N[ h/.{t -> 1.0,u -> uu}]]]]][[1,2]]]
nnsum = Map[#[[1]].#[[2]]&,index]/sites
md[u_]:= (Abs[g[u]]^2).nnsum
plot2:= (ll = Sort[Eigenvalues[ h /. {t -> 1.0, u -> 1.0}]];
           If[ll[[1]]==ll[[2]],
              text = "Achtung: Grundzustand ist entartet!",
              text = ""];
    Plot[md[u], {u,0.0,6},
         Axes -> None, Frame -> True, FrameLabel->
         {"U/t","Doppelbesetzung",text,""}] )
plot3:= (grundzustand = Map[{#[[1]],#[[2]]/sites}&,zustand];
         g1 = ListPlot[grundzustand,Frame->True,Axes->None,
               PlotStyle->PointSize[0],PlotJoined->True,
               FrameLabel->{"U/t","Energie pro Platz/t"},
               DisplayFunction -> Identity ];
f[u_,0] = 1/4 ;
f[u_,0.0] = 0.25 ;
f[u_,x_] = BesselJ[0,x]*BesselJ[1,x]/(x*(1+Exp[x u/2]));
plotlist = Table[{u,-4.0*
                 NIntegrate[f[u,x],{x,0,5.52}]},{u,0,6,.2}];
g2 = ListPlot[plotlist,
              PlotJoined -> True,DisplayFunction -> Identity,
              PlotStyle -> Dashing[{0.01,0.01}]];
Show[g1, g2, DisplayFunction -> $DisplayFunction] )
```

3.1 Populationsdynamik: logabb.c und logabb.m

```
/*****  Populationsdynamik *****/

#include <stdlib.h>
#include <graphics.h>
#include <math.h>

int maxx,maxy;

main()
{
 void kanal(double);
 int gdriver=DETECT,gmode;
```

```
 char ch,str[100];
 double xit=0.4,rmin=.88,r;
 int x,y=50,i,ir;
 initgraph(&gdriver,&gmode,"\\tc");
 maxx=getmaxx();
 maxy=getmaxy();
 setcolor(YELLOW);
 settextstyle(1,0,1);
 outtextxy(100,maxy-80,
 " Logistische Abbildung x = 4 r x (1-x)");
 settextstyle(0,0,1);
 outtextxy(50,maxy-30,
" Befehle : tief,hoch,clearscreen,print r,kanal,exit ");
 setviewport(1,1,maxx,maxy-100,1);
 r=1.-(1.-rmin)*y/(maxy-100);
 while(1)
  {
   xit=4.*r*xit*(1.-xit);
   x=xit*maxx;
   putpixel(x,y,WHITE);
   if(kbhit())
    {
     switch(getch())
      { case 't': if(y<(maxy-100)) y+=1;break;
        case 'h': if(y>1)   y-=1;break;
        default : getch();break;
        case 'e': closegraph(); exit(1);
        case 'k': kanal(r);break;
        case 'p': sprintf(str," r= %lf",r);
                  outtextxy(1,y,str);
                  break;
        case 'c':  clearviewport();break;
      } /* switch */
     r=1.-(1.-rmin)*y/(maxy-100);
     for(i=0;i<100;i++)  xit=4.*r*xit*(1.-xit);
    } /* if */
  }/* while */
}/* main */

void kanal( double r)
{
    double xit=.4;
    int x,y[1000],i;
    clearviewport();
    for (i=0;i<maxx;i++) y[i]=0;
```

```
    line(0,maxy-100,maxx,maxy-100);
    while(!kbhit())
    {    xit=4.*r*xit*(1.-xit);
         x=xit*maxx;
         y[x]++;
         putpixel(x,maxy-100-y[x],WHITE);
     }
      getch();getch();clearviewport();return;
}
```

logabb.m

```
Print [" Logistische Abbildung"]
f[x_]=4 r x(1-x)
iterf[n_]:=Nest[f,x,n]
plot1:=Plot[Evaluate[{x,f[x],iterf[2],iterf[4]}/.r->.87],
                     {x,0,1}, Frame -> True,
        PlotStyle -> {{Thickness[0.002],GrayLevel[0]},
                      {Thickness[0.002],
                       Dashing[{0.003,0.003}]},
                      {Thickness[0.002],GrayLevel[.6]},
                      {Thickness[0.002],GrayLevel[0]}},
        FrameLabel -> {"x","f[f[f[f[x]]]]"} ]
nf[x_,r_]=N[f[x]]
list1[r_,n_]:=NestList[nf[#,r]&,.65,n]
plot2:= (liste=list1[.87,40];
         g1=ListPlot[liste,PlotStyle -> PointSize[0.02],
                     DisplayFunction -> Identity ];
         g2=ListPlot[liste,PlotJoined->True,
             PlotStyle -> {Thickness[0.002],GrayLevel[.7]},
                      FrameLabel -> {"n","x(n)"},
                      DisplayFunction -> Identity ];
         Show[g2, g1, Frame -> True,
              DisplayFunction -> $DisplayFunction] )
plot3[r_:0.87464]:=(g1=Plot[nf[x,r],{x,0,1},
                   DisplayFunction -> Identity];
           xx=Nest[nf[#,r]&,0.3,500];
           li1=NestList[{nf[#[[1]],r],nf[#[[2]],r]}&,
                         {xx,nf[xx,r]},1000];
           li2 = Flatten[Map[{{#[[1]],#[[2]]},
                               {#[[2]],#[[2]]}}&,li1],1];
           li3=Flatten[Map[{{#[[1]],#[[2]]}}&,li1],1];
           g2=ListPlot[li3,PlotRange -> {{0,1},{0,1}},
                       PlotStyle -> PointSize[0.015],
                       DisplayFunction -> Identity ];
```

```
           g3=ListPlot[li2,PlotRange -> {{0,1},{0,1}},
                        PlotJoined -> True,
                        PlotStyle -> Thickness[0],
                        DisplayFunction -> Identity ];
           g4=Graphics[Text[
               StringJoin["r = ",ToString[r]],{0.5,0.05}]];
           g5=Graphics[Line[{{0,0},{1,1}}]];
             Show[g1,g2,g3,g4,g5,PlotRange -> {{0,1},{0,1}},
                  AspectRatio -> Automatic,
                  Frame -> True,
                  DisplayFunction -> $DisplayFunction])
h[x_]=97/25 x(1-x)
hl[n_]:=NestList[h,N[1/3],n]
hl[n_,prec_]:=NestList[h,N[1/3,prec],n]
tab:=Table[{prec,Precision[Last[hl[100,prec]]]},
           {prec,16,100}]
plot4:=ListPlot[tab, Frame -> True,
                PlotStyle -> PointSize[0.01],
                FrameLabel ->
                {"Rechengenauigkeit","Ergebnisgenauigkeit"}]
periode[1]={c,1}
genau=30
maxit=30
periode[n_]:=periode[n]=Join[periode[n-1],
                             correct[periode[n-1]]]
correct[list_]:=Block[{sum=0,li=list,l=Length[list]},
                      Do[sum+=li[[i]],{i,2,l}];
                      If[OddQ[sum],li[[1]]=0,li[[1]]=1];
                      li]
g[n_,mu_]:=Block[{x=Sqrt[mu],l=Length[periode[n]]},
                Do[x=Sqrt[mu+(-1)^(periode[n][[i]]) x],
                   {i,l,3,-1}];  x]
fr[n_]:=fr[n]=(find=FindRoot[g[n,mu]==mu,{mu,{15/10,16/10}},
                      AccuracyGoal->genau,
                      WorkingPrecision->genau,
                      MaxIterations->maxit];
                      mu/.find)
rr[n_]:= rr[n] = (1+Sqrt[1+4*fr[n]])/4
delta[1]:= Print["   Der Wert von n ist zu klein"]
delta[2]:= Print["   Der Wert von n ist zu klein"]
delta[n_]:= delta[n]=(Sqrt[1+4*fr[n-1]]-Sqrt[1+4*fr[n-2]])/
                     (Sqrt[1+4*fr[n]]-Sqrt[1+4*fr[n-1]])
feigenbaum :=
Do[Print["       n = ",n,
         "  Feigenbaum-Konstante = ",delta[n]],{n,3,20}]
```

3.2 Kette auf dem Wellblech: frenkel.c und frenkel.m

```
/*****  Frenkel-Kontorova-Modell  *****/

#include <math.h>
#include <stdlib.h>
#include <graphics.h>

main()
{
 int gdriver=DETECT,gmode;
 int xb,yb,color=WHITE,maxx,maxy;
 long nsum=0;
 double k=1.,sig=.4,p=0.,x=0.,pi=M_PI,xneu,
        pneu,h=0.,wind=0.,xalt,palt;
 char ch,str[1000];

 initgraph(&gdriver,&gmode,"\\tc");
 maxx=getmaxx();
 maxy=getmaxy();
 outtextxy(100,maxy-70," Frenkel-Kontorova-Modell");
 outtextxy(300,maxy-70," Befehle: neu, print, exit");
 xalt=x; palt=p;
 while(1)
     {
      pneu=p+k/2./pi*sin(2.*pi*x);
      xneu=pneu+x;
      xb=fmod(xneu+100.,1.)*maxx;
      yb=(1.-fmod(pneu+100.,1.))*(maxy-100);
      putpixel(xb,yb,color);
      wind+=xneu-x;
      x=xneu; p=pneu;
      h+=k/4./pi/pi*(1.-cos(2.*pi*x))+(p-sig)*(p-sig)/2.;
      nsum++;

      if(kbhit())
       { switch(getch())
          {case 'e' : closegraph(); exit(1);
           case 'n' : x=xalt=(double) rand()/RAND_MAX;
                      p=palt=(double) rand()/RAND_MAX;
                      color=random(getmaxcolor())+1;
                      h=0.;wind=0;nsum=0;break;
           case 'p' :
                      sprintf(str,
       " Energie= %lf  Windungszahl= %lf  (x,p)= %lf,%lf",
```

```
                        h/nsum,wind/nsum,xalt,palt);
                    setviewport(10,maxy-50,maxx,maxy,1);
                    clearviewport();
                    outtextxy(1,1,str);
                    setviewport(1,1,maxx,maxy,1); break;
           default  : getch();break;
            }/* switch */
         }/* if */
      }/* while */
}/* main */
```

frenkel.m

```
Print[" Frenkel- Kontorova- Modell"]
pi = N[Pi]
k=1.
sigma=.4
nmax=1000
t[{x_,p_}] = {x + p + k/(2 pi) Sin[2 pi x],
                  p + k/(2 pi) Sin[2 pi x] }
list[x0_,p0_]:= NestList[t,{x0,p0},nmax]
xlist[x0_,p0_]:= Map[First,list[x0,p0]]
tilde[{x_,p_}]:= {Mod[x,1],Mod[p,1]}
listt[x0_,p0_]:= Map[tilde,list[x0,p0]]
plot1[x0_:.06,p0_:.34]:= ListPlot[listt[x0,p0],
                         Frame -> True,  Axes -> None,
                         FrameLabel->{"x","p"},
                         RotateLabel -> False,
                         PlotStyle -> PointSize[0.00]]
plot2[x0_:.06,p0_:.34]:=
      (xl=Map[First, NestList[t,{x0,p0},10]];
       tab=Table[{xl[[m]],k/(2 pi)^2*(1-Cos[2 pi xl[[m]]])},
                 {m,11}];
       p1=ListPlot[tab, PlotStyle -> PointSize[0.03],
                        DisplayFunction->Identity];
       p2=Plot[k/(2 pi)^2*(1-Cos[2 pi x]),
               {x,xl[[1]],xl[[11]]},
               DisplayFunction->Identity];
       Show[p1,p2,DisplayFunction->$DisplayFunction,
                 Frame -> True] )
plot3[x0_:.06,p0_:.34]:=
      (xl=Map[First, NestList[t, {x0,p0}, 99]];
       ListPlot[Mod[xl,1],
                Frame -> True,FrameLabel->{"n","x(n)"},
                PlotStyle -> PointSize[0.013] ])
```

```
de[{x_,p_}]:=k/(2 pi)^2*(1-Cos[2 pi x])+0.5*(p - sigma)^2
h[x0_:.0838,p0_]:=(ll1=list[x0,p0]; ll2 = Map[de,ll1] ;
                   Apply[Plus,ll2]/Length[ll2])
wind[x0_:.0838,p0_]:=(w1=xlist[x0,p0];
                      (w1[[-1]]-w1[[1]])/nmax )
hp0:=Table[{p0,h[.0838,p0]},{p0, .2, .4, .01}]
plot4:= ListPlot[hp0, Frame -> True,
                 FrameLabel->{"p0","Energie"},
                 PlotStyle -> PointSize[0.013]]
hx0:=Table[{x,h[x,.336]},{x,.15,.2,.001}]
plot5:=ListPlot[hx0,Frame -> True,Axes -> None,
                FrameLabel->{"x0","Energie"},
                PlotStyle -> PointSize[0.013]]
windp0:=Table[{p0,wind[p0]},{p0,0.2,.4,.01}]
plot6:=ListPlot[windp0, Frame -> True,
                FrameLabel->{"p0","windungszahl"},
                PlotStyle -> PointSize[0.013] ]
```

3.3 Fraktale Packung: sierp.c und sierp.m

```
/*****  Fraktales Gitter  *****/

#include <graphics.h>
#include <stdlib.h>
#include <math.h>

main()
{
  int gdriver=DETECT,gmode;
  struct {int x;int y;} pt={10,10},pw,
                       p[3]={{1,1},{500,30},{200,300}};
  initgraph(&gdriver,&gmode,"\\tc");

  while(!kbhit())
    {
      pw=p[random(3)];
      pt.x=(pw.x+pt.x)/2;
      pt.y=(pw.y+pt.y)/2;
      putpixel(pt.x,pt.y,WHITE);
    }
  getch();getch();
  closegraph();
}
```

sierp.m

```
Print[ "Erzeugt und plotted einen Sierpinsky gasket "]
list={{{0.,0.},{.5,N[Sqrt[3/4]]},{1.,0.}}}
verv[d_]:=Block[ {d1,d2,d3},
          d1={d[[1]],(d[[2]]+d[[1]])*.5,(d[[3]]+d[[1]])*.5};
          d2=d1+Table[(d1[[3]]-d1[[1]]),{3}];
          d3=d1+Table[(d1[[2]]-d1[[1]]),{3}];
                  {d1,d2,d3} ]
plot1:= Block[{listzw,plotlist},
              listzw=Map[verv,list];
              list=Flatten[listzw,1];
              plotlist=Map[Polygon,list];
              Show[Graphics[plotlist],
                   AspectRatio -> Automatic]]
```

3.4 Neuronales Netzwerk: nnf.c und nn.c

nn.c ist nur auf Diskette vorhanden.

```
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <stdio.h>

#define N 10
#define N2 20


float runs=0,correct=0;

main(int argc, char * argv[] )
{
 int neuron[N2],input,i;
 float weight[N2],h,wsum,kappa=1.;
 char ch,str[100];
 FILE * fp;

 if(argc==1){
            printf(" Welche Eingabedatei ? ");
            scanf("%s",str);
           }
 else strcpy(str,argv[1]);

 if((fp=fopen(str,"r"))==NULL)
  { printf(" Datei nicht vorhanden ! ");exit(1); }
```

```
    printf(" Auswertung der Datei %s laeuft ",str);
    for(i=0;i<N2;i++) { neuron[i]=1; weight[i]=(float)i/N;
  }
 while(feof(fp)==NULL)
  {
   switch(fgetc(fp))
    {
     case '1' : input=1; runs++;break;
     case '0' : input=-1; runs++;break;
     default  : continue;
    }
   for(h=0.,i=0 ;i<N ;i++)  h+=weight[i]*neuron[i];
   for(wsum=0.,i=0; i<N; i++) wsum+=weight[i]*weight[i];
   if(h*input>0.)  correct++;
   if( h*input < kappa*sqrt(wsum) )
   for(i=0;i<N;i++)  weight[i]+=input*neuron[i]/(float)N;
   for(i=N2-1;i>0;i--) neuron[i]=neuron[i-1];
   neuron[0]=input;
  }
 if(runs!=0) printf("\n Anzahl der Eingaben: %6.2f",runs);
 printf(
  "\n %6.2f %% richtige Vorhersagen",correct/runs*100.);
 fclose(fp);
 getch();
}
```

4.1 Runge-Kutta-Methode: rungek.m

```
Print["    RungeKutta: Pendel"]
RKStep[f_, y_, yp_, dt_]:=
       Module[{ k1, k2, k3, k4 },
                k1 = dt N[ f /. Thread[y -> yp] ];
                k2 = dt N[ f /. Thread[y -> yp + k1/2] ];
                k3 = dt N[ f /. Thread[y -> yp + k2/2] ];
                k4 = dt N[ f /. Thread[y -> yp + k3] ];
                yp + (k1 + 2 k2 + 2 k3 + k4)/6 ]
RungeKutta[f_List, y_List, y0_List, {x_, dx_}] :=
 NestList[RKStep[f,y,#,N[dx]]&,N[y0],Round[N[x/dx]] ] /;
                Length[f] == Length[y] == Length[y0]

EulerStep[f_,y_,yp_,h_]:= yp + h N[f /. Thread[y -> yp]]
Euler[f_,y_,y0_,{x_,dx_}]:=
 NestList[EulerStep[f, y, #, N[dx]]&, N[y0], Round[N[x/dx]]
                                       ]
hamilton = p^2/2 - Cos[q]
```

```
tmax = 200
dt = 0.1
phi0 = Pi/2
p0 = 0
r = 0.05
phase:=RungeKutta[{D[hamilton,p],-D[hamilton,q]},
                   {q, p}, {phi0,p0}, {10,dt}]
plot1:=ListPlot[phase,PlotJoined->True,Frame -> True,
                AspectRatio->Automatic,
                FrameLabel -> {"q","p"},
                RotateLabel -> False,
                PlotRange -> {{-2.3,2.3},{-1.85,1.85}}]
phase2:=RungeKutta[{p,-Sin[q]-r p},
                   {q,p},{phi0,p0},{tmax,dt}]
plot2:= ListPlot[phase2,PlotJoined->True, PlotRange -> All,
                 AspectRatio->Automatic,Frame -> True,
                 FrameLabel -> {"q","p"},
                 RotateLabel -> False,
                 PlotStyle -> Thickness[0.003] ]
phase3:= Euler[{D[hamilton, p], -D[hamilton, q]},
                {q, p}, {phi0, p0}, {10, dt}]
plot3:= ListPlot[phase3,PlotJoined -> True, Frame -> True,
                 AspectRatio -> Automatic,
                 FrameLabel -> {"q","p"},
                 RotateLabel -> False,
                 PlotRange -> {{-2.3,2.3},{-1.85,1.85}}]
```

4.2 Chaotisches Pendel: pendel.c

```
/*****  Chaotisches Pendel *****/

#define float double
#include <graphics.h>
#include <stdlib.h>
#include <math.h>
#include <stdio.h>
#include "\tc\recipes\nr.h"
#include "\tc\recipes\nrutil.h"
#include "\tc\recipes\nrutil.c"
#include "\tc\recipes\odeint.c"
#include "\tc\recipes\rkqc.c"
#include "\tc\recipes\rk4.c"

double dt=.1,r=.25,a=.5,pi,ysc=5.;
int poincare=0,done=0;
```

```
int maxx,maxy,xbild,ybild;
void print(),derivs(double,double*,double*),event(double*);


main()
{
  int gdriver=DETECT,gmode;
  int i,nok,nbad,xalt,yalt,xneu,yneu;
  double y[3],f[3],t=0.,eps=1e-08;

  initgraph(&gdriver,&gmode,"\\tc");
  maxx=getmaxx();
  maxy=getmaxy();
  xbild=maxx-20;
  ybild=maxy-110;
  pi=acos(-1.);

  y[1]=pi/2.;
  y[2]=0.;
  print();
  while(done==0)
   {
    if(kbhit()) event(y);
    if(poincare==1)
     {
      odeint(y,2,t,t+3.*pi,eps,dt,0.,&nok,&nbad,derivs,rkqc);
      xalt=fmod(y[1]/2./pi +100.5,1.)*xbild;
      yalt=y[2]/ysc*ybild/2+ybild/2;
      rectangle(xalt,yalt,xalt+1,yalt+1);
      t=t+3.*pi;
     }
    else
     {
      xalt=fmod(y[1]/2./pi +100.5,1.)*xbild;
      yalt=y[2]/ysc*ybild/2+ybild/2;
      odeint(y,2,t,t+dt,eps,dt,0.,&nok,&nbad,derivs,rkqc);
      xneu=fmod(y[1]/2./pi +100.5,1.)*xbild;
      yneu=y[2]/ysc*ybild/2+ybild/2;
      if(abs(xneu-xalt)<xbild/2) line(xalt,yalt,xneu,yneu);
      t=t+dt;
     }
   }
  closegraph();
}
```

```
void event(double y[])
{
 switch (getch())
  {case 'e': done=1;break;
   case 'c': print();break;
   default : getch();break;
   case 'h': a=a+.01;print();break;
   case 't': a=a-.01;print();break;
   case '+': ysc=ysc/2.;print();break;
   case '-': ysc=ysc*2.;print();break;
   case 's': y[1]=pi/2.;y[2]=0.;print();break;
   case 'u': poincare=!poincare;print();break;
   case 'a': outtextxy(50,50,"a=?");scanf("%lf",&a);
             print();break;
  }
}

void derivs (double t,double *y ,double *f)
 {
        f[1]=y[2];
        f[2]=-r*y[2]-sin(y[1])+a*cos(2./3.*t);
 }

void print(void)
 {
  char string[100];
  setviewport(1,1,maxx,maxy,1);
  clearviewport();
  settextstyle(1,0,1);
  outtextxy(20,ybild+20,"Getriebenes Pendel");
  settextstyle(0,0,1);
  outtextxy(10,ybild+70,
          "Befehle:exit,hoch(a),tief(a),start,a einlesen"
          ",+,-,umschalten,clear,beliebig");
  rectangle(9,9,xbild+11,ybild+11);
  sprintf(string,"a=%lf, r=%lf" ,a,r);
  if(poincare==1)
   sprintf(&string[strlen(string)]," - Poincare");
  outtextxy(220,ybild+40,string);
  setviewport(10,10,xbild+10,ybild+10,1);
 }
```

4.3 Stationäre Zustände: schroed.c

```
/*****   Schroedinger-Gleichung  *****/

#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

double e=.5,a=0.1;

main()
{
  void bild(void), print( double,double,double );
  double k(double),
         step(double *x,double dx,double*y,double*ym1);
  int gdriver=DETECT,gmode;
  int xb,ybneu,ybalt;
  double dx=10./500.,y,ym1,yp1,destart=.05,de=.05,x,xp1,xm1 ;
  initgraph(&gdriver,&gmode,"\\tc");
  bild();

  while(1)
   {
    x=dx/2.;y=ym1=1.;ybalt=1;
    for(xb=1;xb<500;xb++)
     {
      yp1=step(&x,dx,&y,&ym1);
      ybneu=(1.-yp1)*120;
      if(abs(ybneu)>10000) break;
      line(xb-1,ybalt,xb,ybneu);
      ybalt=ybneu;
     }
    switch (getch())
     {case 'e': closegraph();exit(1);
      case 'c': print(e,de,dx);break;
      default : getch();break;
      case '+': e=e+de;print(e,de,dx);break;
      case '-': e=e-de;print(e,de,dx);break;
      case 's': de=destart;print(e,de,dx);break;
      case 'k': de=de/10.;print(e,de,dx);break;
      case 'd': dx=dx/2.;print(e,de,dx);break;
     }/* switch */
   }/* while */
}/* main */
```

```
double step ( double *xa, double dx,double*ya,double*ym1a)
{
    long i,n;
    double k(double);
    double yp1,x,y,ym1,xp1,xm1;
    x=*xa;y=*ya;ym1=*ym1a;
    n=ceil(10./dx/500.);

    for(i=1;i<=n;i++)
       {
             xp1=x+dx;xm1=x-dx;
             yp1=(2.*(1.-5./12.*dx*dx*k(x))*y
                  -(1.+dx*dx/12.*k(xm1))*ym1)/
                  (1.+dx*dx/12.*k(xp1));
             xm1=x;x=xp1;
             ym1=y;y=yp1;
       }
    *xa=x;*ya=y;*ym1a=ym1;
    return yp1;
}

void print (double e,double de,double dx)
{
 char str[100];
 clearviewport();
 line(1,120,500,120);
 sprintf(str," E= %12.8lf, de=%12.8lf, dx= %12.8lf ",
        e,de,dx);
 outtextxy(10,290,str);
 return;
}

void bild()
{
  settextstyle (1,0,1);
  setlinestyle(0,0,3);
  rectangle(9,9,511,311);
  outtextxy(30,350," Schroedinger-Gleichung ");
  settextstyle (0,0,1);
  outtextxy(20,390,
      "Befehle : exit,klein(de),+(e),-(e),start(de),dx/2,"
      "clear screen");
  setlinestyle(0,0,1);
  setviewport(10,10,510,310,1);
}
```

```
double k(double x)
{  return (-pow(x,2)-2.*a*pow(x,4)+2.*e); }
```

4.4 Solitonen: soliton.m

```
Print["          Solitonen"]
soliton = -2 Sech[x-4t]^2
plot1:= Plot3D[soliton,{x,-5,5},{t,-1,1},PlotPoints->50]
gl=D[soliton,t]-6*soliton*D[soliton,x]+D[soliton,{x,3}]==0
max:= Ceiling[20/dx]
ustart:=Table[-6 Sech[(j-max/2)dx]^2//N,{j,0,max}]
step[u_]:= (Do[uplus[k]=RotateLeft[u,k],{k,3}];
               u+dt*(6u*(uplus[1]-u)/dx -
                     (uplus[3]-3uplus[2]+3uplus[1]-u)/dx^3))
plot2[i_:3]:=(dx=0.05; dt=0.02; upast=ustart; zeit=0;
              Do[upres=step[upast];
                 upast=upres;
                 Print["Zeit ",zeit=zeit+dt],{i}];
                 xulist=Table[{(j-max/2)*dx,upres[[j]]},
                              {j,max}];
            ListPlot[xulist,PlotJoined->True,PlotRange->All,
                       Frame -> True, Axes -> {True,False},
                       FrameLabel ->{"x","u(x,0.06)"} ] )
firststep[u_]:=(up1=RotateLeft[u]; up2=RotateLeft[up1];
                um1=RotateRight[u]; um2=RotateRight[um1];
                u+dt*(3*u*(up1-um1)/dx-
                     (up2-2up1+2um1-um2)/(2dx^3)) )
step2[u_,w_]:= (up1=RotateLeft[u]; up2=RotateLeft[up1];
                um1=RotateRight[u]; um2=RotateRight[um1];
                w+dt(2(um1+u+up1)*(up1-um1)/dx -
                      (up2-2up1+2um1-um2)/dx^3 ) )
init:=(dx = 0.18; dt=.002; upast=uPast=ustart; zeit = dt;
       dt=dt/10; upres = firststep[upast]; dt = zeit-dt;
       ufut=step2[upres,upast];upast=upres;upres=ufut;
       upast=uPast; dt = zeit; )
plot3[i_:10]:=(If[ Not[NumberQ[dt]]||dt != .002, init];
               If[zeit==.002,fin=i-1,fin=i];
               Do[ ufut=step2[upres,upast];
               upast=upres;upres=ufut;zeit=zeit+dt,{fin}];
               Print["Zeit ",zeit];
               xulist = Table[{(j-max/2)*dx,ufut[[j+1]]},
                              {j,0,max}];
               zs = StringJoin["u(x,",ToString[zeit],")"];
               uu = Interpolation[xulist];
```

```
                Plot[uu[x],{x,-10.,10.},PlotRange->All,
                    Frame -> True, Axes -> None,
                    PlotStyle -> Thickness[0.002],
                    FrameLabel ->{"x",zs}] )
u2[x_,t_]=-12(3+4Cosh[2x-8t]+Cosh[4x-64t])/
              (3Cosh[x-28t]+Cosh[3x-36t])^2
plot4:=Plot3D[u2[x,t],{t,-.2,.2},{x,-5,5},PlotPoints->50,
               PlotRange -> {-10,0},
               Shading -> False,
               MeshStyle -> Thickness[0],
               AxesEdge -> {Automatic,Automatic,{-1,1}},
               ViewPoint -> {-1.78095, -2.06202, 2.5},
               DefaultFont ->{"Times-Roman",16},
               AxesLabel -> {"t","x","u"}]
plot5:=ContourPlot[-u2[x,t],{t,-1,1},{x,-10,10},
                    PlotPoints->100,
                    ContourShading->False,
                    PlotRange -> All,
                    ContourSmoothing -> 4,
                    FrameLabel -> {"t","x"},
                    RotateLabel -> False,
                    Contours -> {0.1,0.6,1.1,1.6,2.6,4.0,5.6}]
plot6[tt_:0.3]:= (uprime[x_]=D[u2[x,tt],x]; x=-10.0;
         exactlist={{x,u2[x,tt]}}; d=0.07; upl=uprime[x]^2;
         While[x<10., upr=uprime[x+d]^2;
               d = 0.07/Sqrt[1+Max[upl,upr]];
               x = x+d; AppendTo[exactlist,{x,u2[x,tt]}];
               upl=upr ];
         ListPlot[exactlist, PlotRange -> All,
                  PlotStyle -> PointSize[0.002],
                  Frame -> True, Axes -> None] )
```

4.5 Zeitabhängige Schroedinger-Gleichung: welle.c

Nur auf Diskette vorhanden.

5.1 Zufallszahlen: zufall.m und mzran.c

```
x0 = 1234
 a = 106
 c = 1283
 m = 6075
zufall[x_] = Mod[a*x + c,m]
uniform = NestList[zufall,x0,m+1]/N[m]
tripel = Table[Take[uniform,{n,n+2}],{n,1,m,3}]
unitvectors = Map[(#/Sqrt[#.#])&,tripel]
```

```
plot1:=Show[Graphics3D[{PointSize[0.004],
            Map[Point,unitvectors]}],
            ViewPoint -> {2,3,2}]
v0 = {276., 164., 442.}/m  (* vector closest to {0,0,0} *)
(* b1,b2,b3 span primitive cell *)
b1 = {-113., 172., 7.}/m
b2 = {173., 113., -172.}/m
b3 = {345., 120., 570.}/m
vp = m*(9*b1+4*b2)+{0,0,100}

plot2 := Show[Graphics3D[{PointSize[0.004],
              Map[Point,tripel]}],
              ViewPoint -> vp]
uni = Table[Random[],{m+2}];
tri = Table[Take[uni,{n,n+2}],{n,m}]
univec = Map[(#/Sqrt[#.#])&,tri]

plot3:=Show[Graphics3D[{PointSize[0.004],
            Map[Point,univec]}],
            ViewPoint->{2,3,2}]
```

mzran.c

```
/*****  Zufallszahlen  *****/

#define N 1000000
unsigned long x=521288629, y=362436069, z=16163801,
                       c=1, n=1131199209;
unsigned long mzran()
{ unsigned long s;
   if(y>x+c) {s=y-(x+c); c=0;}
     else    {s=y-(x+c)-18; c=1;}
    x=y; y=z; z=s; n=69069*n+1013904243;
    return (z+n);
}

main()
{
     double r=0.;
     long i;
     for (i=0;i<N; i++)
        r+=mzran()/4294967296.;
     printf ("r= %lf \n",r/(double)N);
}
```

5.2 Fraktale Aggregate: dla.c

```
/*****  Fraktale Aggregate  *****/

#include <graphics.h>
#include <stdlib.h>
#include <math.h>

#define lmax  220
#define rs (rmax+2.)
#define rd (rmax+5.)
#define rkill (100.*rmax)

char xf[lmax][lmax] ;
int rx,ry,maxx,maxy;
double rmax=1.,pi;

void main ()
{
    void besetze(),huepfe(),aggregate(),kreissprung();
    char pruefe();
    int driver=DETECT,mode;
    int i,j;

    initgraph (&driver, &mode, "\\tc");
    maxx=getmaxx();
    maxy=getmaxy();
    randomize();
    pi=acos(-1.);
    outtextxy(50,maxy-30,"Befehl: exit");
    for(i=0;i<lmax;i++)
       for(j=0;j<lmax;j++)  xf[i][j]=0;
    xf[lmax/2][lmax/2]=1;

    besetze();
    huepfe();

    while(1)
    {
       switch(pruefe())
         {
           case 'v':besetze();huepfe();break;
           case 'a':aggregate();besetze();huepfe();break;
           case 'h':huepfe();break;
           case 'k':kreissprung();break;
```

```
         }
       if (kbhit())
          switch(getch())
            { case 'e': closegraph();exit(1);
              default : getch();break;}
    }/* while */
}/* main */

void huepfe()
 {
    switch(random(4))
    {
      case 0: rx+=1;break;
      case 1: rx+=-1;break;
      case 2: ry+=1;break;
      case 3: ry+=-1;break;
    }
 }

void aggregate()
 {
   double x,y;
   xf[rx+lmax/2][ry+lmax/2]=1;
   x=rx;y=ry;
   rmax= max(rmax,sqrt(x*x+y*y));
   if(rmax>lmax/2.-5.) {printf("\7");getch();exit(1);}
   circle(4*rx+maxx/2,4*ry+maxy/2,2);
 }

void besetze()
 {
   double phi;
   phi=(double)rand()/RAND_MAX*2.*pi;
   rx=rs*sin(phi);
   ry=rs*cos(phi);
 }

void kreissprung()
 {
   double r,x,y,phi;
   phi=(double)rand()/RAND_MAX*2.*pi;
   x=rx; y=ry; r=sqrt(x*x+y*y);
   rx+=(r-rs)*sin(phi);
   ry+=(r-rs)*cos(phi);
 }
```

```
char pruefe()
 {
   double r,x,y;
   x=rx;
   y=ry;
   r=sqrt(x*x+y*y);
   if(r>rkill)   return 'v';
   if(r>=rd) return 'k';
   if(xf[rx+1+lmax/2][ry+lmax/2]
     +xf[rx-1+lmax/2][ry+lmax/2]
     +xf[rx+lmax/2][ry+1+lmax/2]
     +xf[rx+lmax/2][ry-1+lmax/2]>0) return 'a';
   else return 'h';
 }
```

5.3 Perkolation: perc.c und percgr.c

percgr.c ist nur auf Diskette vorhanden.

```
/***** Perkolation *****/

#include <graphics.h>
#include <conio.h>
#include <stdlib.h>
#include <math.h>

main()
{
  int gdriver=DETECT,gmode;
  double p=.59275;
  int i,j,pr,L=700;

  pr=p*RAND_MAX;
  initgraph(&gdriver,&gmode,"\\tc");
  for(i=0;i<L;i++)
  for(j=0;j<L;j++)
          if(rand()<pr)  putpixel(i,j,WHITE);
  getch();
  closegraph();
}
```

5.4 Polymer-Ketten: reptation.c

Nur auf Diskette vorhanden.

5.5 Ising-Ferromagnet: ising.c

```
#include <time.h>
#include <conio.h>
#include <stdlib.h>
#include <math.h>
#include <stdio.h>
#define L 20
#define VSEG 0xb800

int s[L+2][L+2],bf[3],done=0;
double temp;

main()
{
 void setT(double);
 void event(void);
 void rahmen(int , int);
 char ch;
 int mcs,x,y,e,v;
 clock_t start,end;
 clrscr();
 randomize();
 mcs=0;
 gotoxy(50,1);
 printf("ISING-Ferromagnet ");
 gotoxy(50,2);
 printf("Monte-Carlo-Simulation");
 gotoxy(50,3);
 printf("Systemgroesse %d * %d",L,L);
 gotoxy(1,25);
 printf(
 " Befehle: t(iefer),h(oeher),s(prung),e(xit),beliebig");
 rahmen(2*L+3,L+2);

 for(x=0;x<L+2;x++) for(y=0;y<L+2;y++) s[x][y]=1;
 for(x=1;x<L+1;x++) for(y=1;y<L+1;y++)
   {
    gotoxy(2*(x-1)+3,y+1);
    putch(1);
   }
 getch();
 setT(2.269);
 start=clock();
```

```
 while(!done)
  {
   event();
   for(x=1;x<L+1;x++) for(y=1;y<L+1;y++)
    {
     e=s[x][y]*(s[x-1][y]+s[x+1][y]+s[x][y-1]+s[x][y+1]);
     if( e<0 || rand()<bf[e/2] )
      {
       s[x][y]=-s[x][y];
       v=2*(x*80+2*(y-1)+2);
       ch=(s[x][y]+1)*15;
       poke(VSEG,v,0xf00|ch);
      }
    }
   for(x=1;x<L+1;x++)
    {
     s[0][x]   =s[L][x];
     s[L+1][x]=s[1][x];
     s[x][0]   =s[x][L];
     s[x][L+1]=s[x][1];
    }

   mcs++;gotoxy(50,15);printf(" mcs/Spin : %d",mcs);
   end=clock();
   gotoxy(50,16);
   printf(" CPU Zeit : %.3e",(end-start)/CLK_TCK/(L*L)/mcs);
  }
}
void setT(double t)
 {
    temp=t;
     bf[2]=RAND_MAX*exp(-8./temp);
     bf[1]=RAND_MAX*exp(-4./temp);
     bf[0]=RAND_MAX/2;
     gotoxy(50,10);printf(" Temperatur: %.2f",temp);
 }

void event()
 {
   char ch;
   if(!kbhit()) return;
   ch=getch();
      switch(ch)
      { case 'h':setT(temp+=.05);break;
        case 't':setT(temp-=.05);break;
```

```
        case 's':setT(1.) ;break;
        default :getch();break;
        case 'e':done=1;break;
      }
 }

void rahmen(int xmax,int ymax)
{
    int i=0;
    gotoxy(1,1);
    while(i++,i<xmax) putch(205);
    for(i=1;i<ymax-1;i++)
    {
     gotoxy(1,i+1);
     putch(186);
     gotoxy(xmax,i+1);
     putch(186);
    }
    i=0;
    gotoxy(1,ymax);
    while(i++,i<xmax) putch(205);
    gotoxy(1,1);putch(201);
    gotoxy(xmax,1);putch(187);
    gotoxy(1,ymax);putch(200);
    gotoxy(xmax,ymax);putch(188);
}
```

5.6 Kürzeste Rundreise: travel.c

Nur auf Diskette vorhanden.

Sach- und Namenverzeichnis

A

Abbildung
  logistische 89, 98
  nichtlineare 102
Ableitung 42, 92
  diskretisierte 158
Ableitungsoperator 39
`Abs` 9
Abweichung, quadratische 25
`AccuracyGoal` 94
Aggregat 183
  fraktales 188
aliasing 18
Amplitudengleichung 155, 158
Anfangsbedingung 67, 92, 134, 181
Anfangsverteilung 162
  symmetrisierte 163
Anfangswert 128, 145
Anfangswertproblem 181
Anfangszustand 92, 167
Anharmonizität 54
Antisymmetrie 78
Antriebsperiode 136
`Apply` 4
Approximation 53, 153
`ArcSin` 9, 235
Arrhenius-Gesetz 219, 224
`AspectRatio` 115, 236
`atoi` 227
Attraktor 95, 114, 135
  chaotischer 140
  periodischer 136
  seltsamer 136
Aufenthaltswahrscheinlichkeit 51, 162, 171
Ausdehnung, mittlere 182, 191
Ausdruck, logischer 59
Auszahlungsmatrix 46, 49
`Automatic` 115, 236

B

Bahn
  chaotische 96, 108, 110, 142
  periodische 96, 98
  superstabile 92, 94, 98
Band 68, 71, 85, 99, 142
  chaotisches 91, 94
Barriere, gaußförmige 172
Beleuchtung, stroboskopische 136
Bénard-Experiment 143
Besetzungswahrscheinlichkeit 204, 212
Besetzungszahl 79
Besselfunktion 23
`BesselJ` 23, 236, 241
Bewegungsgleichung 65, 130, 133
  lineare 65
Bloch-Theorem 71
`Block` 4, 94, 241
Boltzmann-Faktor 216, 223, 226
Boltzmann-Konstante 42, 210
bond percolation 194
Boolesche Funktion 118
`break` 123, 251
Bulirsch-Stoer-Methode 131

C

`case` 123
`CDF` 27
`ceil` 147
Chaos 91, 96
  deterministisches 89, 91
chaotisch 89, 108, 110, 135, 170
`char` 5, 247

χ^2-Test 25
χ^2-Verteilung 26, 29
Chirikov-Abbildung 103
`ChiSquareDistribution` 27
`Chop` 67, 239
`Circle` 34
`clock_t` 5
Cluster 182, 188, 190, 193, 195, 198
`ContourPlot` 9, 28, 35, 236
Contour-Plot 157
`Contours` 9
Courant-Bedingung 155
critical slowing down 220

D

`D` 39, 43, 242
Daten
  geglättete 23
  verrauschte 23
`default` 123, 251
`define` 5, 249
detailed balance 205
Determinante 73, 78
`DiagonalMatrix` 53, 67, 243
Diagonalmatrix 53
Differentialgleichung 90, 127, 225
  gewöhnliche 127, 225
  lineare 51, 65
  nichtlineare 127
  partielle 150, 159, 164
Differenzenform 152
Differenzengleichung 165
Diffusion 180, 216, 219
Diffusion Limited Aggregation (DLA) 181
Diffusionsgleichung 181
  radialsymmetrische 183
Dimension, fraktale 112, 136, 181, 188, 191, 193, 201, 209
Dipolmoment 33, 35
Dipolnäherung 37
`Do` 4, 94, 152, 156, 241
Domänenwand 216, 219
Doppelbesetzung 83, 85
Doppelmuldenpotential 56, 133, 149
`Dot` 34, 244
`double` 5, 247
Drehmoment, periodisches 135
Dreiecksgitter 190
`Drop` 34
Dualitätssatz 48
Dynamik
  kontinuierliche 103
  symbolische 93, 98, 101

E

Eigenfunktion 147
Eigenfunktionsentwicklung 168
Eigenmode 154, 172
Eigenschwingung 67, 69
`Eigensystem` 82, 244
`Eigenvalues` 53, 67, 82, 244
Eigenvektor 51, 65, 67, 72, 83
Eigenwert 52, 54, 65, 67, 71, 73, 76, 83, 85, 145, 160
Eigenwertgleichung 51, 56, 67
  verallgemeinerte 66
Eigenwertproblem 72
Eigenzustand 51, 71, 85, 160
Einbettungsdimension 112
Einheitszelle 65, 70
Einteilchenniveau 84
Einteilchenzustand 77
Einzugsgebiet 136
Elementarzelle 65
`EllipticF` 8
`EllipticE` 38
`EllipticK` 8, 38
`else` 207, 250
End-zu-End-Abstand 205
Energie 210, 223
Energieband 71
Energiebarriere 219
Energie-Eigenwert 54, 145
Energiegebirge 230

Energieniveau 45, 56, 84, 148
Entartung 85
Entwicklung, symbolische 12
Entwicklungskoeffizienten 161
Erzeugungsoperator 52, 78, 80
Euler-Schritt 164
Euler-Verfahren 128, 132
`Evaluate` 9
`Expand` 82, 237, 246
Explosionskriterium 148
Exponent, kritischer 191, 200, 202, 213, 220

F

Faltung 15, 22
Fast Fourier Transform (FFT) 16
Fehler, lokaler 130
Fehlerintervall 27
Feigenbaumkonstante 91, 98
Feld 41
  elektrisches 33, 35
  magnetisches 69
  zweidimensionales 185
Feldlinienbild 38
`feof` 123
`fgetc` 123
Fibonacci-Zahl 21
Filter 61
  elektrisches 67
`FindMinimum` 27
`FindRoot` 44, 92, 94, 107, 240, 245
finite size scaling 193, 196, 200, 214, 220
`First` 104
Fit, nichtlinearer 25, 27
Fixpunkt 91, 95,
  stabiler 96
`Flatten` 68, 81
`float` 5, 247
Flory, P. J. 201
Fluktuation 193, 212, 216, 218, 220
Fluß
  flächenerhaltender 103
  magnetischer 70
Flüssigkeit 43
Flußquantum 70
`fopen` 122
`For` 4, 241, 242
`for` 5, 35, 75, 147, 197, 217, 248, 251
Form, quadratische 35
`Fourier` 9, 16
Fourierkoeffizienten 60
Fourierreihe 9, 59
Fouriertransformation 8, 11, 13, 22, 56, 60, 66, 181
  inverse 60
Fouriertransformierte 14, 168, 232
Fraktal 112, 115, 136, 188, 191
Frequenz, kommensurable 107
Frequenzspektrum 91
Fugazität 220
Fünferzyklus 96, 101

G

Gammafunktion, unvollständige 26
Gas 43
Gaußfunktion 22, 162
Gaußverteilung 179
`gcc` 6, 256
Generator, linear kongruenter 174
Gesamtladung 33
Gesamtwahrscheinlichkeit 164
Geschwindigkeit 39
Gewicht
  statistisches 203
  synaptisches 117, 121, 123
Gewichtsvektor 119
Gewinn, mittlerer 46, 48
Gewinntabelle 46
Gitter
  kubisches 181
  quadratisches 190
Gittergas 220
Gitterschwingungen 65, 68
Gittervektoren, primitive 175

Glättung 21
Gleichgewicht 212
  detailliertes 205, 211, 214, 223
  thermisches 211, 216, 219, 223, 224
Gleichung
  lineare 51
  transzendente 45
Gleichungssystem 51, 57, 60
  lineares 65
Gradient 33, 39
`Graphics` 34, 115
`Graphics3D` 175
Graphik 216
Graphikobjekt 34, 115, 175
Grassberger, P. und *Procaccia, I.* 137
Gravitationskraft 133
Grenzwertsatz, zentraler 211
Grundfrequenz 161
Grundzustand 82, 85, 104, 110, 145
Grundzustandsenergie 54, 86, 224

H

Hamiltonfunktion 132
Hamiltonmatrix 54, 78, 82
Hamiltonoperator 52, 56, 77, 79, 82, 165
  skalierter 160
Handlungsreisender 222
Harper-Gleichung 70, 72, 76
Hermitepolynome 52
Histogramm 95
Hofstadter, D. 69
Hofstadter-Schmetterling 76
Höhenlinien 9, 35
Hoshen, J. und *Kopelman, R.* 195
Hubbard, J. 78
Hubbard-Modell 78
Hüpfmatrixelement 70
Hyperebene 118, 119

I

`If` 45, 82, 241, 242
implizit 165
Impulsoperator 70
`include` 5, 138, 249
Informationsentropie 137
Informationsdimension 137
Initialisierung 152
`int` 5, 247
Integerzahlen, vorzeichenlose 177
Integral
  elliptisches 7
  vollständiges elliptisches 8
`Integrate` 9, 12, 40, 242
Integration, numerische 145, 164
Integrationsprozedur 139
Integrationsschritt 147
Interferenz 161, 169
`Interpolation` 156
`Inverse` 67, 93, 244
`InverseFourier` 23
irrational 104, 108, 110
Ising-Ferromagnet 213, 218
Ising-Modell 210
Isotherme 41
Isotropie, räumliche 182
Iteration 93
  inverse 93, 98
  nichtlineare 103

J

`JacobiAmplitude` 8
`JacobiSN` 8
`Join` 94

K

KAM-Trajektorie 108, 110
Kern 22
Kirchhoffsche Regeln 57
Knobelspiel 50
Knotensatz 74, 145
Kochsche Kurve 116
Koexistenz 44

kommensurabel 71, 104, 107, 110
kompilieren 6
Konfiguration 203
Konfigurationsraum 212
Kontraktion 35
Kontrollparameter 220
Konturen 30
Konvergenzbeweis 119
Konvektionsrollen 142
Konzentration, kritische 190
Korrelation 175
Korrelationslänge 193, 213, 218
Korteweg-de-Vries-Gleichung 150
Kraft, periodische 135
Kreisfrequenz 57
Kristall, aperiodischer 20

L

Ladungsverteilung 32
Leapfrog-Methode 131
Leath, P. L. 195
Leistung 63
Leistungsspektrum 20
Leiterschleife, kreisförmige 38
`Length` 4, 82, 238
Lernen 118, 120
Lernregel 117, 119
`Line` 34
`LinearProgramming` 46, 49
`Listable` 39, 238
Listen 5, 237
`ListPlot` 11, 68, 104, 152, 238, 239
Ljapunow-Exponent 92, 100
`long` 5, 247
Lorenz, E. N. 142
Lorenzmodell 143
Lösung
  analytische 44
  numerische 44, 51, 56, 77, 127, 156

M

Magnetfeld 38
Magnetisierung 213
  spontane 219
`main` 5, 247, 248
`Map` 68, 104, 115, 174
Marsaglia, G. und *Zaman, A.* 177
Massenmatrix 66
Maßstab 111
Mastergleichung 204, 212
Matrix 5, 52, 71, 80, 84, 149, 252, 255
  diagonale 52
  hermitesche 66
  tridiagonale 73, 154, 159
Matrixdarstellung 53
`MatrixForm` 54, 243
Matrixgleichung 154
`MaxIterations` 94
Maxwell-Gerade 42, 44
Maxwell-Konstruktion 41
`memcpy` 228
Meßdaten 22
Metropolis-Algorithmus 212, 214, 230
Minimax-Theorem 47
Minimum, absolutes 224
Mittel
  gewichtetes 22
  statistisches 202, 212
  thermisches 220
Mittelwertbildung 153
`Mod` 105, 174
Modellfunktion 25
`Module` 131
Modulo 105, 177, 206, 250
Modulo-Abbildung 105
Modulo-Funktion 174
Modulo-Operation 177, 207
Molekulardynamik 131
Molekularfeldtheorie 202
Monomer 112

Monte-Carlo-Schritt 227, 232
Monte-Carlo-Simulation 87, 215, 217, 223, 226
Multipol-Entwicklung 32

N

`NDSolve` 127, 245
`Needs` 28, 240
`Nest` 92, 100, 104, 241, 242
`NestList` 104, 132, 174, 241, 242
Netzwerk
  elektrisches 56, 57
  lineares 59
  neuronales 117
Neumann, J. von 46
Neurocomputer 117
Neuron 118
Neuronales Netzwerk 117
Neuronenaktivität 121
Nichtlinearität 8
`NIntegrate` 13, 243
Niveau 55
`NonlinearFit` 27, 240
Normalmoden 66
Nullstellenbestimmung 71
Numerov-Verfahren 146

O

`OddQ` 94
`odeint` 138, 139
Ohmsches Gesetz 56, 58
Onsager, L. 213, 218
Operator 78, 81
  unitärer 164
Operatoralgebra 78, 81
Optimierung 223
  lineare 46, 48, 223
Optimierungsaufgabe 45
Optimierungsproblem
  diskretes 223
  kombinatorisches 224
Ordnung, magnetische 190
Ordnungsparameter 220
Ortsdarstellung 51, 160
Oszillation, numerisch bedingte 153
Oszillator
  anharmonischer 52, 145
  harmonischer 54

P

Parallelkreis 59, 64
Parameterraum 30
Parametervektor 25, 30
`ParametricPlot3D` 39, 236
Pauli-Prinzip 77, 79
`PDF` 28
Peierls-Trick 70
Pendel 6, 132
  getriebenes 140
  nichtlineares 135
Periode, teilerfremde 178
Periodendauer 162
Periodenlänge 174, 177
Periodenverdoppelung 91, 98, 141
Periodizität 72
Perkolation 189
Perkolationscluster 191
Perkolationsschwelle 190, 193, 198
Permutationen 222
`Permutations` 81, 239
Perzeptron 117, 123
Perzeptron-Lernregel 119
Phänomen, kritisches 189
Phase 61
Phasendifferenz 57
Phasenfaktor 161
Phasengeschwindigkeit 168
Phasenraum 9, 11, 103, 132, 135
Phasenraumdiagramm 105
Phasenübergang 42, 91, 190, 192, 202, 213, 219
Phasenumwandlung 210
Phononen 65, 68

Pixel 75, 95, 113, 148
Plastizität, synaptische 118
`Plot` 8, 156, 236
`Plot3D` 35, 236
`PlotRange` 8
`PlotVectorField` 35, 38
`Plus` 4
Poincaré, H. 134
Poincaré-Schnitt 90, 103, 136, 140
`Point` 175
`Polygon` 115
Polymer 201, 206, 209
Polymerdynamik 202
Polymerkonfiguration 203
Polymermolekül 112, 201
Polynom 71, 75
  charakteristisches 73
Populationsdynamik 90
Potential 37, 39, 40, 144, 160
  anharmonisches 52, 148
  chemisches 220
  elektrisches 118
  elektrostatisches 32
  kommensurables 71
  periodisches 102, 107
  quadratisches 65
  symmetrisches 145
Potentialmulde 133
Potentialtopf 45
Potenzsingularität 213
`Precision` 100
Predictor-Corrector-Methode 131
`Print` 156, 241, 242
`printf` 5, 248
Produktansatz 51
Prozedur 4
Prozeß, stochastischer 212, 214, 223
Pseudozufallszahl 173, 175, 177

Q

Quadrupolmoment 33, 35
Quadrupolnäherung 37
Quadrupoltensor 33
Quanten-Monte-Carlo-Methode 87
`Quantile` 28
Quasi-Ellipsoid 31

R

`rand` 5, 113, 174, 187, 195, 228, 253
Randbedingung 146, 163, 181, 216
  periodische 66, 79, 84, 216
`RAND_MAX` 5, 186, 253
`Random` 4, 106, 174, 176, 237
random walk 113, 181, 201, 209
Rechteckimpuls 17
Rechteckspannung 64
Regel 28, 43
Reibungskraft 133
Reihendarstellung 160
Reihenentwicklung 9
Rekursion 166
Relaxationszeit 136, 216, 219
Renormierungsgruppentheorie 192, 213
`ReplacePart` 81
Reptation 203, 206
Reptationsalgorithmus 205
Resonanz 58, 61, 63
Resonanzfrequenz 59, 61, 64
Resonanzkurve 61
`return` 5, 147, 254
Richardson-Extrapolation 131
Rosenblatt, F. 119
Rosenblatt-Regel 121
`RotateLeft` 23, 152, 156
`RotateRight` 156
Rotation 40
`Round` 132
Rückwärts-Iteration 166
Rückwärtsrekursion 102
Ruhelage 65, 102, 103, 133, 135, 140
Rundweg 222
Runge-Kutta-Verfahren 129, 138, 143, 146

S

Sägezahnspannung 60
Satz von Gerschgorin 74
Satz von Liouville 103
`scanf` 249, 257
Schema, implizites 159
Schieberegister-Generatoren 176
Schießverfahren 145, 149
Schmetterling 69
Schnitt, goldener 108
Schrittweite 130, 133, 158, 169
Schrittweitenkontrolle, adaptive 138
Schröder, M. 116
Schrödinger-Gleichung 51, 70, 144, 151, 160, 164, 168
  diskrete 70, 76
  stationäre 52, 144
  zeitabhängige 159
Schwarzsche Ungleichung 120
Schwellenwert 118, 190
Schwingkreis 57, 61
Schwingung
  gedämpfte 140
  harmonische 10, 136
  longitudinale 65
Schwingungsdauer 7
Schwingungsform 65
Schwingungsmoden 67
`Sech` 152
selbstähnlich 114, 218
Selbstüberschneidung 204
self-avoiding walk 201
Sequenz, periodische 174
Serienkreis 58
Serienschaltung 59, 61
`Series` 9, 243
`Show` 34, 236, 239
`ShowProgress` 28
Sierpinski-gasket 114
`Simplify` 41, 242
simulated annealing 224, 231
Simulation 178, 194, 202, 216, 223
  numerische 163
site percolation 194
Skalar 33
Skalarprodukt 34, 39, 82
Skalenexponent 193
Skalengesetz 192, 213
Skalenverhalten 225
Skalierung 231
Soliton 151
`Solve` 43, 45, 59
`Sort` 82, 238
Spektrum 69
Spielstrategie 45, 46
Spieltheorie 46
Spinglas 223
Spinkonfiguration 211
Spinkorrelationen 87
Spitzenspannung 20, 232
Spur 71
`Sqrt` 9, 53, 234
Stabilität 158
Stabilitätsbedingung 159
Stabilitätsgrenze 159
Standardabbildung 103
Standardabweichung 194
Steuerparameter 143
Störung 54, 91
Strategie 46, 47
  optimale 48, 50
  stochastische 47
Stromdichte 162, 183
`struct` 113, 196, 206, 227
Sturmsche Kette 73
Subharmonische 91
subtract-with-borrow generators 177
`Sum` 35, 82
Supercomputer 211
Suszeptibilität, magnetische 213, 219
`switch` 123, 186, 251
Symmetrie 15, 72, 74, 76, 80, 85, 87, 145, 147, 161, 203, 214
Symmetriebrechung, spontane 213
Synapse 117
Synapsenstärke 119

System
  deterministisches 124
  mechanisches 134

T

`Table` 4, 82, 128, 152, 156, 238
`Take` 174
Tastaturpuffer 227
Taylor-Entwicklung 128, 146, 169
Taylorreihe 129, 153
Temperatur
  kritische 42, 45
  skalierte 45
Tensor 5, 33
Teufelstreppe 109
`TeXForm` 12, 83
`Thickness` 34
`Thread` 82, 131, 244
Torus 216
Trajektorie 107, 135
  chaotische 110
Transfermatrix-Methode 211
Transformation, inverse 14
Translationen 66
Translationsoperator 70
travelling salesman problem 221
Trefferrate 121, 123
Tridiagonalmatrix 53, 72, 165
`True` 53, 240, 243
Tunneleffekt 145, 172
`typedef` 177, 206, 227
Typendeklaration 113

U

Überdeckung 136
Übergangswahrscheinlichkeit 204, 211, 214, 217, 223
Überholvorgang 151, 157
Überlagerung 51, 67, 164
  unendliche 161
Universalität 89, 189, 191, 213
`unsigned long int` 177

V

Vakuum 78
van-der-Waals-Gas 41
van-der-Waals-Gleichung 41, 43
Variable, abgeschirmte 4
Varianz 25, 179, 221
Vektor 5, 33, 252, 255
Vektoranalysis 39
Vektorfeld 39
Vektoroperationen 39
Vektorpotential 38, 70
Verallgemeinern 118, 121
Verhulst, P. F. 90
Verlustleistung 64
Vernichtungsoperator 52, 78, 80
Verschiebungssatz 48
Verteilung, stationäre 204
Vertrauensintervall 26, 29
Vertrauensniveau 31
Verzweigungspunkt 92
Vibrationen 69
Vielfaches, irrationales 136
Vielteilchensystem 78, 223
Vielteilchenzustand 77, 81, 210
Viererzyklus 91, 99
`ViewPoint` 175
`void` 186, 254
Vorwärts-2-Punkt-Formel 152
Vorzeichenwechsel 73, 75

W

Wachstumsalgorithmus 196, 198
Wachstumsprozeß 180, 195
Wahrscheinlichkeit 47, 160
Wahrscheinlichkeitsdichte 181
Wärme, spezifische 213, 220
Wärmegleichgewicht 210
Wechselspannung 56
Wechselstrom 56

Wechselwirkung
  attraktive 202
  magnetische 190
Wechselwirkungspotential 102
Wegintegral 39
Weglänge, skalierte 230
Welle, stehende 161
Wellenfunktion 51, 70, 74, 78, 144, 148, 160, 163, 166
Wellenpaket 151, 161, 168, 170
  Gaußsches 167
  lokalisiertes 159
  numerisch integriertes 171
Wert, irrationaler 71
`while` 95, 123, 139, 251
Widerstand, komplexer 56, 57
Wiederkehrzeit 159
Wilson, K. G. 213
Windungszahl 104, 107, 109
  rationale 107
Witten, T. A. und *Sander, L. M.* 182
`WorkingPrecision` 94

Z

Zahl, irrationale 71, 76, 104
Zeiger 5, 252, 257
Zeit, charakteristische 168
Zeitableitung 155
Zeitentwicklung 164, 169
  quantenmechanische 160
Zeitentwicklungsoperator 164
Zeitmittel 63
Zeitreihe 121
Zeitskala 168, 219
Zufallsbewegung 112, 180
Zufallsvektor 112
Zufallsweg 182, 201, 202
Zufallszahl 4, 23, 49, 113, 124, 173, 207, 226
  gleichverteilte 179, 207, 215
  normalverteilte 32
Zufallszahlengenerator 179, 218
Zuordnung 59
Zustand 103
  fastperiodischer 107
  inkommensurabler 107, 110
  kommensurabler 104, 107
  stabiler 102
  stationärer 51, 144, 161, 212
Zustandsgleichung 41
Zustandssumme 210
Zweierzyklus 91
Zwei-Personen-Nullsummen-Spiel 46, 49
Zweiphasengemisch 43
Zwei-Soliton-Lösung 158
Zyklus 97, 141